Updating XML Views

by

Ling Wang

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

Aug 23, 2006

APPROVED:

Prof. Elke A. Rundensteiner Advisor	Prof. Dan Dougherty Committee Member
Prof. Murali Mani Co-advisor	Prof. Susan Davidson University of Pennsylvania External Committee Member
Prof. Michael Gennert Head of Department	

Abstract

Update operations over XML views are essential for applications using XML views. In this dissertation work, we provide scalable solutions to support updating through XML views defined over relational databases. Especially we focus on the update-public semantic, where updates are always public (made to the public database), and the update-local semantic, where update effects are first kept local and then made public as and when required.

Towards this, we propose the clean extended-source theory for determining whether a correct view update translation exists, which then serves as a theoretical foundation for us to design practical XML view updating algorithms.

Under update-public semantic, state-of-the-art view updating work focus on identifying the correct update translation purely on the data. We instead take a schema-centric solution, which utilizes the schema of the underlying source to effectively prune updates that are guaranteed to be not translatable and pass updates that are guaranteed to be translatable directly to the SQL engine. Only those updates that could not be classified

using schema knowledge are finally analyzed by examining the data. This required data-level check is further optimized under schema guidance to prune the search space for finding a correct translation.

As the first work addressing the update-local semantic, we propose a practical framework, called LoGo. LoGo Localizes the view update translation, while preserves the properties of views being side-effect free and updates being always updatable. LoGo also supports on-demand merging of the local database of the subject view into the public database (also called global database), while still guaranteeing the subject view being free of side effects. A flexible synchronization service is provided in LoGo that enables all other views defined over the same public database to be refreshed, i.e., synchronized with the publically committed changes, if so desired.

Further, given that XML is an ordered data model, we propose an order-sensitive solution named O-HUX to support XML view updating with order. We have implemented the algorithms, along with respective optimization techniques. Experimental results confirm the effectiveness of the proposed services, and highlight its performance characteristics.

Acknowledgments

This dissertation and the growth in my knowledge over the last few years owe a great deal to many professors, colleagues, and friends. First among them is my advisor, Prof. Elke A. Rundensteiner. She inspired my interests in database research and gave me direction by suggesting interesting problems. It has been my luck to have her as my advisor. Her technical and editorial advice was essential to the completion of this dissertation. I express my sincere thanks for her support, advice, patience, and encouragement throughout my graduate studies. Her persistence in tackling problems, confidence, and great teaching will always be an inspiration.

My thank goes to the members of my Ph.D. committee, Prof. Murali Mani, Prof. Dan Dougherty and Prof. Davidson, who provided valuable feedback and suggestions to my comprehensive-exam, my dissertation proposal talk and dissertation drafts. All these helped to improve the presentation and contents of this dissertation. I thank Prof. Kathi Fisler for her time and efforts discussing with me in my research qualifying exam.

I would like to thank Ming Jiang for his collaboration on the HUX system. I thank Song Wang for his valuable discussions on part of my first

dissertation task and on the Rainbow system integration. My thanks also go to all other previous and current Rainbow team members for their useful discussions and feedback. The long hours spent in the Fuller Lab would not have been possible but for the company of wonderful office colleagues around me such as Hong Su. The friendship of Song Wang, Xin Zhang, Li Chen and all the other pervious and current DSRG members is much appreciated. They have contributed to many interesting and good-spirited discussions related to this research.

I also thank Worcester Polytechnic Institute and the Computer Science Department for giving me the opportunity to study and also providing TAship during my studies.

My parents receive my deepest gratitude and love for their dedication and the many years of support during my studies.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivating Problem

1.1.1 Applications Using XML Views

In many database applications views play an important role as a means to structure information with respect to specific users' needs. Views also provide the support for logical data independence. That is, the changes in the conceptual schema of the database can be shielded from applications. Views are valuable in the context of security. Namely, we can define views that give a group of users access to just the information they are allowed to see [AHV95].

Extensible Markup Language (XML) [W3C98] is increasingly considered the format of choice for the exchange of information among various applications on the Internet. The popularity of XML is due in large part to its flexibility for representing many kinds of information. The use of tags

makes XML data self-describing, and the extensible nature of XML makes it possible to define new kinds of documents for specialized purposes.

XML makes it possible for applications to exchange data in a standard format that is independent of storage. For example, one application may use a native XML storage format, whereas another may store data in a relational database. Since XML is emerging as a standard for data exchange, it is natural that queries among applications should be expressed as queries against data in XML format. This use gives rise to a requirement for a query language designed expressly for XML data sources. XQuery is designed for this purpose.

XML data are different from relational data in several important aspects that influence the design of a query language. Relational data tend to have a regular structure, which allows the descriptive meta-data for these data to be stored in a separate catalog. XML data, in contrast, are often quite heterogeneous, and distribute their meta-data throughout the document. XML documents often contain many levels of nested elements, whereas relational data are flat. XML documents have an intrinsic order, whereas relational data are unordered except where an ordering can be derived from data values.

With XML [W3C98] becoming the standard for interchanging data between web applications, XML views over various data storage medium, typically relational databases or native XML documents, are commonly used by many applications. Among the key benefits of XML are its vendor and platform independence and its high flexibility. XML is a data model suited for any combination of structured, unstructured, and semistructured

data. XML data is easy to extend because new tags can be defined as needed. XML documents can easily be transformed into different looking XML and even into other formats such as HTML. Furthermore, XML documents can easily be checked for compliance with a schema. All this has become possible through widely available tools and standards such as XML parsers, XSLT (Extensible Style sheet Language Transformation), and XML Schema. They greatly relieve applications from the burden of dealing with the particularities of proprietary data formats. In an era where message formats, business forms, and services change frequently, XML reduces the cost and time required to maintain application logic.

Below we list some of such applications using XML views.

- **Biological Applications.** The *Protein Information Resource (PIR)* located at Georgetown University Medical Center (GUMC), is an integrated public bioinformatics resource to support genomic and proteomic research, and scientific studies. PIR provides the Protein Sequence Database (PSD) of functionally annotated protein sequences, which grew out of the Atlas of Protein Sequence and Structure (1965-1978). PSD is created either as an XML database or a relational database [Res].

  Another example of a biological application is the Human Genome Project [HGP], which has involved thousands of scientists distributed over many universities and industry during a 13 year period to identify all the genes in the human DNA, to store, share and further analyze this information.

A biological application over these databases might create an XML view that collects only proteins having experimental evidence that really functions as described. Such an XML view can be used among biologists to share or exchange the data.

- **Geographical Applications.** The *MONDIAL database* is a world geographic database integrated from the CIA World Factbook, the International Atlas, and the TERRA database among other sources. The geographical data is stored either in XML documents or in relational tables [May]. A particular user might only be interested in the geographical data from a certain region. User specific XML views can be employed for this purpose.

- **Online DBLP Applications.** The *Digital Bibliography Library Project* (DBLP) [Pro] provides bibliographic information on major computer science journals and proceedings. The database researchers might prefer to monitor only certain journals and conferences, such as SIGMOD, VLDB, ICDE, etc. User specific XML views could be utilized to extract those data of high interest to certain online user groups.

- **Existing Relational Databases.** Most business data is currently stored in relational database systems. XML views are a general and flexible way to publish relational data as XML. Typical XML applications publish data from a relational database via XML views.

In response, major efforts from both commercial database systems [Rys01, BKKM00, CX00] as well as research projects [CKS+00, FKS+02, JAKC+02]

focus on supporting XML views.

1.1.2 Updating Through Views

Frequently, users do not directly access the underlying data storage, in fact, they may not even be familiar with or even be aware of it. Instead they access derived information in terms of views as part of an external schema, usually by specifying queries against these virtual XML views. These queries are translated to queries against the underlying data storage through query modification [Rys01, BKKM00, CX00, CKS+00, FKS+02, JAKC+02, ZDW+03]. However, while the support of queries on views is necessary, this is clearly not sufficient. Support must be provided for update operations over views as we will introduce below.

Update operations are essential for applications using XML views, especially in dynamic environments for the following reasons:

- Without the capability of updating the stored data, the stored data often becomes quickly out of date and less valuable.
- Supporting updates through the XML view will provide users with a uniform interface (XML and XQuery), independent of what the underlying data storage is, or which query and update language is being used. It is thus convenient and easy to use.

Consider a typical view application domain such as scientific data sharing in the Human Genome Project [HGP]. A public database (such as the NCBI gene bank) has first been built and thereafter has been commonly used and extended as appropriate by scientists in related areas. Scientists

use this as well as other public databases by either directly querying over the database or through a view. Such a public database represents a valuable common resource that must be properly maintained and frequently updated to ensure that the data is up-to-date, as new discoveries are made and then contributed by different laboratories. This maintenance could be accomplished by allowing a qualified user to update the public database through her view assuming appropriate permission.

Given the importance and popularity of XML views, it is also essential to support updates over XML views.

1.2 A Bird's Eye View of the View Updating Problem

In current practice, however, updates must be specified against the underlying data storage rather than against the view, because updating through views is often ambiguous. In other words, there are usually different ways of translating an update, each of which leaves the database in a different state. Update requests through views are difficult in the sense that they have to be translated into "appropriate" updates on the underlying databases. In particular, the translation of updates often need to be handled transparently by the database system as much as possible, i.e., be invisible to users, while the effects of translated updates should at the same time meet the user expectations.

To address this ambiguity, different *update semantic* can be defined. An *update semantic* is a description of a general approach for how updates on the view should be translated.

1.2.1 View Update Semantics

Given that views are usually virtual, view instance data is obtained by applying the view query on the base tables. The most common semantics require view updates be achieved in a way that the view state after the update is the same as what we would obtain if the update had been applied directly to a corresponding materialized view instance. This is referred to as updates through views with the guarantee of being free of side effects. Update semantics without side effect free guarantee are also used in some scenarios [Kel86b, BKT01, CWW00], which requires either always favoring view side effect minimization [BKT01, CWW00], or, user communication to agree with potential view side effects [Kel86b]. In this dissertation, we focus on the side effect free view update semantic. In most cases, users would like the update to be achieved exactly as they required, instead of bearing some extra side effect. Thus the side effect free semantic is also the default semantic used by commercial databases and also commonly studied by majority of research works [Kel85, BS81, CP84, DB82]. In addition, non-side-effect-free update semantics usually involve view semantic and user interaction, which is not the focus of this dissertation.

We classify update semantics based on the following two aspects. First, what is the goal of the approach. Are view side effects allowed? Do all updates need to be translatable, or, could some updates be rejected for not being translatable. Second, what are the restrictions for the particular semantic? Can any local data be associated with the view for updating purpose? Is any change made to the view always supposed to be written through

into the underlying database? If so, is the database schema allowed to be changed?

In this dissertation, we focus on two semantics, namely, the update-public semantic and the update-local semantic as described below. The rest of semantics, we call them the hybrid update semantics as shortly described below, are not the focus of this dissertation. The reason is that the approaches of handling them can be easily obtained by merging the update-public and the update-local semantic.

Update-public semantic. This is the "traditional" view update semantic used by most relational view update solutions [Kel86b, Kel85, BS81, CP84, DB82]. According to this semantic, all updates on the view need to be achieved by mapping them into updates over the base data only. No schema change can be made to the base database. No local data can be directly associated with the view and used to compute the view content. If a view side effect free translation exists, the view update is accepted and translated. Otherwise, the view update is rejected.

This is a very strict update semantic since many restrictions apply for a correct translation to exist. Updates are thus not always translatable. Depending on the view definition and the update specified on the view, most of the updates might be rejected in most cases. It also requires a complex update translatability checking model that can identify a view side effect and reject updates which do not have any side effect free translation.

However, it is a simple yet useful update semantic because all views defined over the same database are always "synchronized" — the content

of each view is extracted from the same database state, and views are aware of changes made by each other. Also, there is no data replication since no local data copy is ever associated with the view compared to the update-local semantic described below.

Some commercial relational database systems, such as Oracle [BKKM00], DB2 [CX00] and SQL-Server [Rys01], use even stricter semantic: an update is accepted and translated only if it has single unique side effect free translation.

So far, most of the view updating works [Kel86b, Kel85, BS81, CP84, DB82] focus on the update-public semantic. The two major aspects to be tackled for this problem include:

- **Update Translatability**. Is the update specified over the view mappable to updates over the base data storage? In other words, does there exist at least one sequence of updates over the base data storage which "correctly" translates the given view update? Any particular view update request may result in a view state that does not correspond to any database state. Such a view update request should be rejected. This decision depends not only on the database instance, the view definition and the update operation, but also on the *update translation policy*. This policy determines the database updates to be generated for a given view update.

- **Translation Strategy**. That is, assuming the view update is indeed identified as being translatable, the translation strategy selects how to translate the updates on the XML view into the equivalent tuple-

based SQL updates or XML document updates on the base data. Ideally, there will be precisely one way to perform the database update that results in the desired view update. However, if the view is many-to-one, the new view state may correspond to many database states. There thus exists ambiguity.

Update-local semantic. Now consider another scenario where scientists with update capabilities may prefer to first keep their research results (updates) "local" instead of always immediately updating the public database through their views with all their newly discovered findings. Reasons for this are plentiful. For example, newly identified gene information still needs to be verified before it goes public. Another reason will be competitiveness. Scientists may want to keep their discoveries private as long as possible, that is, as long as permitted by regulation (e.g., for Human Genome Project, anyone with government funding had six months maximum limitation of public disclosure), or until the results had been successfully published first. This requirement is called *update-local*.

Once the local data is ready (e.g., scientific articles reporting the finished genome sequence have been published), scientists (*subject view* user) may want to (or have to) release the qualified data into the "public" database, so that other scientists (*object view*) can gain access to this shared data. Other scientists (*object view* users) can choose either to "synchronize" their local database (if it is not empty) with the public database, or, to leave their local customized data as it is. We call the two requirements *data merging* and *synchronization* respectively.

In response to the above scenario, we design the update-local semantic. (1) *Localize* the view update translation, while preserving the properties of views being side-effect free and updates being always updatable. (2) On-demand merging of the local database of the *subject view* into the public database, while still guaranteeing the subject view to be free of side effects. (3) On-demand synchronizing the local data of the *object view* with the public database updated by the *subject view*.

This is the most general update semantic in the sense that all updates are translatable in a side effect free manner under this semantic. It also provides the maximum flexibility to the user such that they can keep their localized data and get all their changes written through to the database whenever they want.

However, it is a complex update semantic in the sense that (i) the view contents are now extracted from both a "public" database and a "local" database associated with the view; (ii) different views defined over the same database are not always synchronized since each of them cannot keep their own local data which might be hidden from others; (iii) data replication is often used to eliminate view side effects (e.g., using stored local data copy to restore elements disappearing due to a view side effect).

Update hybrid semantic. There are other semantics which are not as strict as the update public semantic and not as flexible as the update-local semantic either. We call them *update hybrid semantic*. One typical hybrid semantic is used in the recent work [YK06]. Any changes made by the user through her view are encoded using special identifiers in the underlying database.

This then requires both data and schema changes. All view updates are thus translatable, in the sense that they can be encoded, namely, first cloned then some portion of the clones are updated. It also ensures that side-effects are not visible to users.

1.2.2 Limitations of Existing Works

There are several implementations of XML storage that are independent of relational databases [Mic01, Sah02, XH, dbX, JAKC+02, FHK+02, ST00, Sch00]. Several of these are driven by the document (or programming language) community, rather than the database community. **Natix** [FHK+02] has been developed as a storage manager suitable for XML data. The focus is on efficient management of tree-structured data at the level of page allocation and physical placement. **Timber** [JAKC+02] is based upon a bulk algebra for manipulating trees, and natively stores XML. It also developed new access methods to evaluate queries in the XML context. **Tamino** [ST00, Sch00] is a leading commercial "native" XML database, which uses a nested relational engine as its data store, with the bulk of the innovation in the product coming from new index structures, support for handling XML schematic information, and the web interface layer.

XML databases have also been implemented on top of an object oriented database [FE01, LAW99, RP02, eXc] and a semi-structured database [GMW99, ML01].

Several mapping techniques have been proposed [FK99, KM00, SYU99] to express tree-based XML data to flat tables in a relational schema. Many recent XML data management systems [CKS+00, SKS+01, FMST01, MFK01b,

PB02, DT03, KCKN03, DTCO03, CKN03, KKN02] support queries over XML views of the relational data.

In **XPeranto** system from IBM [CKS+00, SKS+01], a framework for processing complex XQuery queries over XML views is presented. The given user and view queries are converted to the XQGM (XML Query Graph Model) representation, and optimized by eliminating the construction of intermediate XML fragments and pushing down the computation. A single SQL query is generated according to the optimized XQGM inside the relational engine.

SilkRoute [FMST01] proposes an algorithm for translating an XQuery expression into SQL by using the *View-Forest* methodology to separate the structure of the output XML document from the computation that produces the document's contents. Especially, it addresses the issue of how to decompose an XML view over a relational database into an optimized set of SQL queries. **Agora** [MFK01b] uses an LAV approach to translate XQuery FLWR expressions into SQL. **Rolex** [PB02] addresses the issue of evaluating a series of navigation operations on a virtual DOM wrapping a relational database into SQL. **MARS** [DT03] proposes an XML-to-SQL translation algorithm when both GAV-style and LAV-style views are present. The system uses the constraints on both relational and XML data to achieve optimized performance.

Commercial database systems, such as Oracle, DB2 and SQL-Server, also provide XML support. **Oracle** XML DB [BKKM00] provides SQL/XML as an extension to SQL, using functions and operators, to include processing of XML data in relational stores. The SQL/XML operators make it

possible to query and access XML content as part of normal SQL operations. It also provides methods for generating XML from the result of an SQL Select statement. The **IBM DB2** XML Extender [CX00] provides powerful user-defined functions to store and retrieve XML documents in XML columns, as well as to extract XML elements or attribute values. In **Microsoft SQL Server2000** [Rys01], SQLXML supports the evaluation of XPath queries over the annotated XML Schema.

Despite the importance of the XML view update problem, it has not yet received much attention from the database research community.

So far, most existing work is designed under update-public semantic. Translating updates issued on virtual views is a long standing difficult issue [BS81, CP84, DB82, Kel85, Kel86a, Kel86b] under this semantic even in the relational scenario. The reason is that a side effect free update mapping from a view update to base updates does not always exist and, when it does exist, it may not be unique.

[DB82] is a practical approach dealing with view update translatability checking for relational databases. It stipulates the notions of *correct translation* and *clean source*. It also presents an approach for performing a careful semantic analysis of the view definition to determine the existence of a unique or at least a small set of update translations. The update translation strategy has been studied for the Select-Project-Join views on relational databases [Kel85, Kel86a, Kel86b]. These works have been further extended for object-based views in [BSKW91], when the view is anchored in a *pivot relation* and updates are specified only in those *well-nested* relations. With XML now as a nested data model, these works might not be suitable and

thus need to be further examined.

The update translation for XML view updating scenario has also been explored to some degree in recent works such as [BKKM00, BDH03, BDH04, CX00, Rys01, TIHW01]. Under the assumption that the given update is translatable, [BDH03, BDH04] propose an update translation strategy for converting the given XML view update into a relational view update. The main result of [TIHW01] is a proposal of an XQuery update grammar. It also studies the execution performance of translated updates.

Commercial database systems such as SQL-Server2000 [Rys01], Oracle [BKKM00] and DB2 [CX00] also provide system-specific solutions for restricted update types. **Oracle** XML DB [BKKM00] provides SQL/XML as an extension to SQL, using functions and operators to query and access XML content as part of normal SQL operations. It also provides methods for generating XML from the result of an SQL Select statement. The **IBM DB2** XML Extender [CX00] provides user-defined functions to store and retrieve XML documents in XML columns, as well as to extract XML elements or attribute values. However, neither IBM nor Oracle support update operations. [Rys01] introduces XML view updates in the **SQL-Server2000** based on a specific *annotated schema* and update language called *updategrams*. Instead of using update statements, the user provides a before and after image of the view. The system computes the difference between the images and generates corresponding SQL statements to reflect changes on the relational database.

However, none of these works consider any of the following basic questions for XML views: (i) what are the possible correct translations and how

to find them? (ii) Which is the most suitable one and how to identify it? To the best of our knowledge, no work has been done under the update-local semantic to address the view updating problem either in the relational or the XML scenario.

The first update hybrid semantic is proposed and has been used in the very recent work [YK06] for the relational scenario. It has not yet been adapted to the XML scenario because it lacks view side effects checking mechanism and thus cannot reject updates with side effects.

1.2.3 Supporting Updates over XML Views

The ambiguity shown above exists in general for the problem of updates through views. When the view in question is in XML format, new challenges arise as summarized below.

- The mismatch between the XML hierarchical view model and the base model. That is the nested structure imposed by an XML view may be in conflict with the hierarchy explicitly or implicitly defined by the underlying base data model. In other words, if the base is relational, the constraints of the relational schema imply the base hierarchical structure. If the base is an XML document, its schema expresses the hierarchy. This mismatch will affect the translatability of the view updates. In particular, the challenge arises from the fact that the XML view does not determine a unique relational database schema or XML document schema underneath. Hence assumptions about the specific nature of the base data storage cannot be built into the view-update

algorithm.

- Updates can be specified on any XML view element. Compared to the fixed tuple-based update in the relational view update scenario, this flexible update granularity now causes new issues. As an example, although the relational database can be normalized and the XML document can be well-formed to follow a certain schema, XML views can be very complex and potentially contain data duplications. The flexible granularity of XML updates thus could touch several duplicates, while leaving others untouched. Such an update would not be translatable without any side effect.

- XML is an order-sensitive data model. An order-sensitive update can delete or insert a view element in a certain position of the XML view. Order-sensitive update translation is a problem specific for XML views, because both relational and the object-oriented data model are both ordered models.

Motivating Examples. Given the fact that (1) XML views are hierarchically structured and (2) updates can happen on any element along the view hierarchy, XML view updating is more complex than relational view updating. Fig. 1.1(a) shows a running example of a relational database for a course registration system. A virtual XML view in Fig. 1.1(c) is defined by the *view query* in Fig. 1.1(b). The following examples illustrate cases of classifying updates as translatable or not translatable. The XML update language from [TIHW01] or the update primitives from [BDH04] are used to define update operations. For simplicity, in the examples below we only

Professor

	pid	pname
t_1	p1	David Finkel
t_2	p2	Tim Merrett

Key:{pid}

Course

	cid	cname	pid
t_1	c1	Math	p1
t_2	c2	Physics	p1
t_3	c3	English	p2

Key={cid}

FK: pid → Professor.pid

Student

	sid	sname	cid
t_1	s1	Chun Zhang	c1
t_2	s2	Mike Fisher	c1
t_3	s3	Feng Lee	c2

Key={sid, cid}

FK: cid → Course.cid

(a) Relational Database

```
FOR $p IN DOCUMENT(Professor/ROW),
    $c IN DOCUMENT(Course/ROW)
WHERE $p.pid = $c.pid
RETURN
    <ClassInfo>
        <Course>
            $c/cname/text()
        </Course>,
        <Professor-Student>
            <Professor>$p/pname/text()</Professor>,
            FOR $s IN DOCUMENT(Student/ROW)
            WHERE $s.cid = $c.cid
            RETURN
                <Student>
                    $s/sname/text()
                </Student>
        </Professor-Student>
    </ClassInfo>
```

(b) View query

```
CI1     <ClassInfo>
CI1.C1      <Course>Math</Course>
CI1.PS1     <Professor-Student>
CI1.PS1.P1      <Professor>David Finkel</Professor>
CI1.PS1.S1      <Student>Chun Zhang</Student>
CI1.PS1.S2      <Student>Mike Fisher</Student>
            </Professor-Student>
        </ClassInfo>
CI2     <ClassInfo>
CI2.C1      <Course>Physics</Course>
CI2.PS1     <Professor-Student>
CI2.PS1.P1      <Professor>David Finkle</Professor>
CI2.PS1.S1      <Student>Feng Lee</Student>
            </Professor-Student>
        </ClassInfo>
CI3     <ClassInfo>
CI3.C1      <Course>English</Course>
CI3.PS1     <Professor-Student>
CI3.PS1.P1      <Professor>Tim Merrett</Professor>
            </Professor-Student>
        </ClassInfo>
```

(c) XML view

Figure 1.1: The running example for the course registration system

use a delete primitive with the format *(delete* `nodeID`*)*, where *nodeID* is the abbreviated identifier of the element to be deleted [1]. For example, `CI1` indicates the first *ClassInfo* element, while `CI1.PS1` the first *Professor-Student* element of the first *ClassInfo* element. We use $Professor.t_1$ to indicate the first tuple of relation *Professor*.

Now let's consider an example of an update under the update-public semantic.

Example 1 *Consider the update* $u_2 = \{$*delete* `CI1.C1`$\}$*.*
The appearance of the view element `CI1.C1` *is determined by two tuples:* $Professor.t_1$ *and* $Course.t_1$*. There are three choices for achieving this update:* T_1=$\{$*delete* $Professor.t_1\}$*,* T_2=$\{$*delete* $Course.t_1\}$ *and* T_3=$\{$*delete* $Professor.t_1$*, delete* $Course.t_1\}$*. All three translations would cause a view side effect, namely, the whole ClassInfo element would disappear. From the view query, we see that any ClassInfo element must always have a pair of Professor and Course sub-elements. Deleting the course element would break this join condition and thus make the whole ClassInfo element disappear.*

As we can see from the above examples, not only view updates can happen anywhere along the view hierarchy, but also side effects can appear anywhere in the view. The XML view updating is thus more complex than in the relational case.

[1] Note that the view here is still *virtual*. In reality, this `nodeID` is achieved by specifying conditions in the update query [WRMJ05].

1.3 Contributions of This Dissertation

In this dissertation work, we focus on the problem of updating XML views, which wrap relational data. The main contributions of this dissertation work include the following.

- We propose the clean extended source theory to serve as theoretical foundation for the study of the view update problem under all different update semantics.
- We propose our XML view updating approach specific to both the update-public semantic and the update-local semantic. We call them *HUX* and *LoGo* respectively.
- We have designed and implemented the HUX and LoGo algorithms, along with respective optimization techniques in our XML management system *Rainbow*.
- We propose an approach named *O-HUX* to extend HUX and LoGo with ordered semantics.
- We report experiments assessing the performance and usefulness of proposed algorithms.
- We prove the correctness of the proposed theory and algorithms.

1.3.1 Clean Extended Source Theory

Here we first explore the fundamental theory to determine whether a given update over the XML view is indeed translatable. We call it *clean extended*

source theory, which provides the theoretical foundation for the study of the view updating problem under all different semantics [WR04, WRMar].

In the relational scenario, [BS81, CP84] propose a complementary theory that requires a correct mapping to avoid view side effects as well as database side effects. Database-side-effect-free means that for a translation to be considered correct, it cannot affect any part of the database that is "outside" the view. This correctness criteria, however, is too restrictive to be practical. [DB82] relaxes this condition to only require that no view side effect occurs. In other words, a translation is correct as long as it corresponds exactly to the specified update, and it does not affect anything else in the view. Using the concept of "clean source", it also characterizes the schema conditions under which an update of a relational view is translatable.

With the hierarchical structure of the XML data model in consideration, our work treats an XML view as a "composition" of a set of relational views by following the approach in the literature [FKS+02, BDH04]. We extend the concept of a "clean source" for relational databases [DB82] into the concept of a "clean extended source" suitable for XML. This extension takes the foreign key constraint in the relational data model into consideration, since it is very commonly used as join condition to form the XML view hierarchy. We propose a clean extended source theory for determining the existence of a correct relational update translation for a given XML view update.

We now exploit this proposed clean extended source theory to serve as a theoretical foundation for solving the XML view updating problem under

various update semantics. Our main contributions in this direction include: (1) We characterize the *update translatability* problem for XML views and identify key factors affecting the translatability. (2) We then use the theory for determining whether a correct view update translation exists. (3) We prove the correctness of our *clean extended source theory*.

1.3.2 XML View Updates Handling Under Update-public Semantic (HUX)

Here we design practical algorithms to determine whether a given update over the XML view is indeed translatable under the update-public semantic. We propose a schema-centric approach named *HUX* — Handling Updates in XML [WRM06, WRMJ06].

HUX first bridges the XML and relational view update problem by treating the XML view as a "composition" of a set of relational views. An update over a schema node is treated as an update over its relational mapping view. This in turn can be handled as relational view update problem. However, such a simple transformation between the two problems is not sufficient. The relationship between a parent SQL view and its children SQL views is critical in the XML view scenario for side effect-free checking. The XML view update problem can be viewed as the problem of updating multiple SQL views, with restrictions regarding how updates on one SQL view can affect other SQL views.

HUX utilizes the schema of the underlying source to effectively prune updates that are guaranteed to be not translatable and pass updates that

are guaranteed to be translatable directly to the SQL engine. Updates that could not be classified using schema knowledge are finally analyzed by examining the data. This required data-level check is further optimized under schema guidance to prune the search space for finding a correct translation. Extensive experiments illustrate the reliability and performance benefits of HUX.

We make the following contributions in this direction: (1) We propose the pure data-driven strategy for XML view updating, which guarantees that all updates are fully classified. (2) We also propose a schema-driven update translatability reasoning strategy, which uses schema knowledge including now both keys and foreign keys to efficiently filter out untranslatable and identify translatable updates when possible. (3) We then design an interleaved strategy that optimally combines both schema and data knowledge into one update algorithm, which performs a complete classification in polynomial time for the core subset of XQuery views. (4) We have implemented the algorithms along with respective optimization techniques in a working XQuery view system named HUX. We report experiments assessing its performance and usefulness.

1.3.3 XML View Updates Handling Under Update-Local Semantic (LoGo)

In this work, we propose a practical framework, called LoGo, that provides flexible view updating services under update-local semantic. LoGo achieves update translation (mapping updates over views to updates over

the data), while guaranteeing *side-effect free* semantics as well as *update-local* as required by update-local semantic.

Further, LoGo supports writing through from the local database to the public (global) database whenever desired. A flexible synchronization service is provided that enables all other views defined over the same public database to be refreshed, i.e., synchronized with the publically committed changes, if so desired.

LoGo is the first solution that provides a flexible view updating service. Experimental results confirm the effectiveness of the proposed services, and highlight its performance characteristics.

To summarize, our work in this direction makes the following contributions: (1) We propose a new view update semantics which relies on localized behavior to guarantee: (i) all view updates are translatable in a view side effect-free manner, (ii) local user updates can be separated from the global database when desired and (iii) views are independent from each other in terms of update effects. (2) We propose the LoGo framework which fulfills our newly proposed update semantics, yet supports synchronization between local and global behavior when so desired. (3) We implement the LoGo system. Experiments are also conducted to assess the performance of LoGo.

1.3.4 XML View Update Handling with Order (O-HUX)

Here we study the problem of updating XML views with order being considered, we call it O-HUX. O-HUX classifies the order syntax in the XML view definition into different categories. For each category, we design a set

of rules that identify order-sensitive candidate update translations.

Our contributions include: (1) To the best of our knowledge, we are the first to study updating order-sensitive XML views. (2) We extend the clean source theory to order-sensitive semantics. (3) Based on the order-sensitive clean source theory, we develop the O-HUX algorithm that guarantees view side effect free semantics while considering most of the XQuery order constructs. (4) Our O-HUX algorithm relies largely on SQL, and hence can be easily adopted by relational database systems to support order-sensitive view updating.

1.4 Dissertation Organization

Chapter 2 describes the background data model used in XML view updating. After that, the dissertation is organized into four parts. The first part, focusing on the clean extended source theory, is described in Chapter 3. The second part, described in Chapter 4, depicts the updating algorithm of XML views under update-public semantic. While the third part, focusing on the XML view updating under update-local semantic, is described in Chapter 5. Chapter 6 describes the order-sensitive XML view updating solution. Conclusions of this dissertation and future work are described in Chapters 7 and 8 respectively.

Chapter 2

Background

2.1 XQuery

The query language XQuery [Sco02] is a language to express to extract and manipulate data from XML documents or any data source that can be viewed as XML, such as relational databases or office documents. XQuery uses XPath expression syntax to address specific parts of an XML document. It supplements this with a SQL-like "FLWOR expression" for performing joins. A FLWOR expression is constructed from the five clauses after which it is named: FOR, LET, WHERE, ORDER BY, RETURN. The language also provides syntax allowing new XML documents to be constructed. Where the element and attribute names are known in advance, an XML-like syntax can be used; in other cases, expressions referred to as dynamic node constructors are available. All these constructs are defined as expressions within the language, and can be arbitrarily nested.

XQuery language is based on a tree-structured model of the information

content of an XML document, containing seven kinds of nodes: document nodes, elements, attributes, text nodes, comments, processing instructions, and namespaces. The type system of the XQuery language models all values as sequences (a singleton value is considered to be a sequence of length one). The items in a sequence can either be nodes or atomic values. Atomic values may be integers, strings, booleans, and so on: the full list of types is based on the primitive types defined in XML Schema [W3Ca].

2.2 XQuery Views over a Relational Database

XML views can be defined over a relational database using XQuery. An **XML view** V is specified by a **view definition** DEF_V over a given relational database D. In our case, DEF_V is an XQuery expression [W3C03] called a *view query*.

Fig. 1.1(a) in Chapter 1 shows a running example of a relational database for a course registration system. A virtual XML view in Fig. 1.1(c) is defined by the *view query* in Fig. 1.1(b).

2.3 Updates Language for Modifying XML Data

Several update languages have been used to update XML data (views or XML documents) [BDH03, TIHW01, XD, W3Cb, WLL03].

[TIHW01] proposes an XQuery like update language to change XML views, which includes a set of basic update operations for both ordered and unordered XML data. The update language used in [TIHW01] is shown in

Figure 2.1(a). For example, the deletion of the view element `CI1.C1` from the XML view in Figure 1.1 is expressed as in Figure 2.1(b).

```
  FOR $binding1 IN XPath-expr, ...
  LET $binding := XPath-expr, ...
  WHERE predicate1, ...
  updateOp, ...

where updateOp is dened in EBNF as:
  UPDATE $binding f subOp f, subOp g

and subOp is:
  DELETE $child j
  RENAME $child TO name j
  INSERT content [BEFORE j AFTER $child ] j
  REPLACE $child WITH $content j
  FOR $binding IN XPath-subexpr, ...
    WHERE predicate1, ... updateOp
```

(a)

```
FOR $ci IN document(``View.xml'')/ClassInfo,
    $c  IN $ci/Course
WHERE $c.text() = 'Math'
UPDATE $ci {
  DELETE $c }
```

(b)

Figure 2.1: Update language used by [TIHW01]

The XUpdate language proposed in [XD] defines the syntax and semantics of XUpdate, which is a language for updating XML documents. XUpdate is designed to be used independently of any implementation. An update is represented by an `xupdate:modifications` element in an

XML document. An `xupdate:modifications` element must have a version attribute, indicating the version of XUpdate that the update requires. The xupdate:modifications element may contain the following types of elements:

```
xupdate:insert-before
xupdate:insert-after
xupdate:append
xupdate:update
xupdate:remove
xupdate:rename
xupdate:variable
xupdate:value-of
xupdate:if
```

Figure 2.2: XUpdate language used by [XD]

For example, an update `<xupdate:remove select=/ClassInfo/Course [text()='Math']/>` will also delete the view element `CI1.C1` from the XML view in Figure 1.1.

[BDH03] proposes to use update primitives in updating XML data. The proposed update primitive model is very simple, and allows the insertion of a subtree at a given node, the deletion of the subtree rooted at a given node, or the modification of a node's context.

Definition 1 *An update operation u over an XML view V is a tuple <u,δ,ref>, where u is the type of operation (insert, delete, modify); is the XML tree to be inserted, or (in case of a modification) an atomic value; and ref is the address of a node in the XML tree. Deletions do not need to specify a δ since all the nodes under ref will be deleted.*

The reference *ref* can be obtained by an addressing scheme such as DOM. For example, to delete the course `CI1.C1` "Math", we specify: u = delete, *ref = /vendors/ vendor/products/book[btitle="ComputerNetworks"]*.

Recently the World Wide Web constitution proposes the XQuery update facility to perform the update operations. For example, the update operation: `do delete fn:doc(‘‘view.xml’’)/ClassInfo/Course [text()='Math']` deletes the `CI1.C1` element.

Each of the above proposals is suitable for updating XML data. [BDH03, XD, W3Cb] are simple solutions using XPath, while [TIHW01] is an XQuery language for more complex update operations, such as sequence updates. However, in this dissertation we consider only single element updates. Either update language proposal above thus is suitable for our purpose. Henceforth we thus only indicate which element to insert, delete, or modify without pointing to the specific update language.

2.4 XML View Updating Problem

Let $\mho$ be the domain of update operations over the view. Let $u \in \mho$ be an update on the view V. An *insertion* adds while a *deletion* removes an element from the XML view. A *replacement* replaces an existing view element with a new one. A taxonomy of the view update domain $\mho$ is shown in Fig. 2.3(a). A **valid view update** is an insert, delete or replace operation that satisfies all constraints in the view schema. All updates discussed in this dissertation are assumed to be valid.

Definition 2 *A relational update sequence U on a relational database D is a*

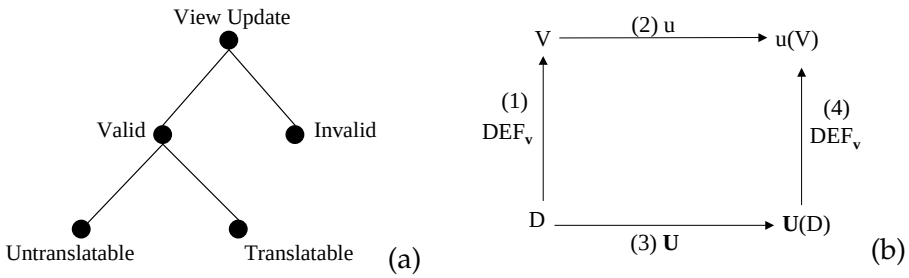


Figure 2.3: (a) The partition of view update domain $\mho$ and (b) the correct translation of view updates

correct translation *of a valid update* u *on the view* V *iff (i)* $u(DEF_V(D)) = DEF_V(U(D))$ *and (ii) if* $u(DEF_V(D)) = DEF_V(D) \Rightarrow U(D) = D$.

A correct translation means the "rectangle" rule shown in Fig. 2.3(b) holds. Intuitively, this implies that the translated relational updates exactly perform the view update and nothing else, namely, without view side effects. In addition, if an update operation does not affect the view, then it should not affect the relational base either. This guarantees that any modification of the relational base is indeed done for the sake of the view. This second criterion is guaranteed if the translation is done by composing the view query and the update query. Hence it generally can be achieved.

For a valid update, if a correct translation does not exist, u is **untranslatable**. Otherwise, it is said to be **translatable** (Fig. 2.3(b)). We consider the effects of foreign key constraints on update translatability since they are widely used in defining XML views, while being largely ignored in existing work even for relational view updates [Kel86b, Kel85, DB82].

We assume a view update u is a valid view update and the update translation policy is the same type update translation.

2.5 Modeling XML Views using Schema Graphs

Similar to SilkRoute [FKS+02], we consider the XML view as the "composition" of a set of relational views as will be introduced in Section 2.6. We use a set of schema graphs to capture the mapping relationship between an XML view and its relational views, as well as relationships among relational views. In the rest of this dissertation, we make extensive use of these schema graphs to handle XML view updating under different semantics.

2.5.1 View Annotated Schema Graph

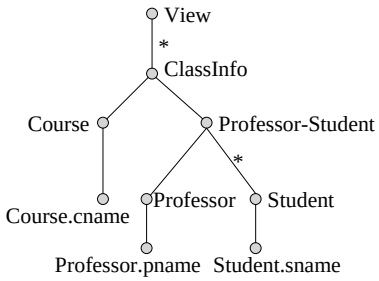
We use the *View Annotated Schema Graph* to separate the structure of the XML view from the computations that produce the content of the view, with the latter expressed in SQL.

A **view annotated schema graph** (ASG), denoted by $\mathcal{G}_V$, is a forest in which each node is labeled with an XML label and an SQL query over the relational database. Fig. 2.4 depicts the view ASG for the XML view in Fig. 1.1. Nodes $v \in \mathcal{G}_V$ include root, leaf and complex nodes. The XML label of the root and complex nodes is the corresponding element or attribute name; for leaf nodes the label is its corresponding relational column name. The SQL annotation of a node v represents the computations that produce the atomic values contained in v.

The nodes, edges, and XML labels of the view ASG represents the structure of the XML view, while the SQL query represent the computation performed by the relational database in order to construct the view. Fig. 2.4(a) shows the structure of the XML view in Figure 1.1. The XML view can be

computed by executing these SQL queries, and constructing a distinct XML node from these SQL answers. The view ASG in Fig. 2.4(b) shows the SQL queries for each node of our example view [1]. Given two nodes v_1, v_2 in $\mathcal{G}_V$, the edge (v_1, v_2) represents that v_1 is a parent of v_2 in the view hierarchy.

The view ASG exists because XQuery supports sequences whose items may be arbitrary XML elements. In practice, view ASGs are often trees. We now describe how to obtain the view ASG from the view query. This description is similar to [FKS+02].



(a)

```
CI: Select *
    from Professor P, Course C
    where P.pid=C.pid
C : Same as CI node
PS: Same as CI node
P : Same as CI node
S : Select *
    from Professor P, Course C, Student S
    where P.pid=C.pid and C.cid = S.cid
```

(b)

Figure 2.4: The view ASG for the XML view in Fig. 1.1

We first associate with each view ASG node an SQL fragment. The SQL fragment consists of a required select clause, containing a single value of the corresponding atomic type and optional from and where clauses. For exposition purposes, we use only the string atomic type and assume all other values are cast into strings. We require that there is a tuple variable that is bound to each table occurring in a from clause, and that every tuple variable used in the SQL fragment of some node v is bound in the from

[1] In the rest of the dissertation, we use v to indicate a complex node. The treatment of other nodes is either trivial or similar to that of complex nodes.

clause of n or in the from clause of one of v's ancestors. The SQL fragments for our running example view query in Figure 1.1 are in Figure 2.5.

```
FOR $p IN DOCUMENT(Professor/ROW),
    $c IN DOCUMENT(Course/ROW)
WHERE $p.pid = $c.pid
RETURN
  <ClassInfo>                                ;from Professor p, Course c where p.pid=c.pid
    <Course>                                 ;from ()
       $c/cname/text()                       ;select p.cname
    </Course>,                               ;
    <Professor-Student>                      ;from ()
        <Professor>                          ;from ()
           $p/pname/text()                   ;select p.pname
        </Professor>,                        ;
        FOR $s IN DOCUMENT(Student/ROW)      ;
        WHERE $s.cid = $c.cid                ;
        RETURN                               ;
           <Student>                         ;from student s where s.cid = c.cid
               $s/sname/text()               ;select s.sname
           </Student>                        ;
    </Professor-Student>                     ;
  </ClassInfo>                               ;
```

Figure 2.5: The SQL fragments of XQuery view in Figure 1.1

The SQL fragments of the internal nodes, e.g., CI-node, C-node and PS-node, contain FROM and WHERE clauses, whereas the leaf nodes, e.g., $c/cname/text(), contain only SELECT clauses. The FROM or WHERE clauses may be empty, in which case we omit them (e.g., in the leaf nodes), or represent them with from () (e.g., P-node. These queries are fragments: a WHERE or SELECT clause in a fragment may have tuple variables that are not defined in that fragment. However, each such tuple variable must be defined in the FROM clause of an ancestor. For example, CI-node defines the tuple variable c, which is used in in the WHERE clauses of S-node.

We associate with each node v a complete SQL query, Q_v, as follows. The FROM clause of Q_v is the concatenation of all FROM clauses of v and all v's ancestors; the WHERE clause of Q_v is the conjunction of all WHERE

clauses of v and all v's ancestors; and if v is a leaf, the SELECT clause of Q_v is that of v, otherwise it is `SELECT *`. Notice that Q_v is complete, i.e., all tuple variables used in Q_v are defined in the FROM clause. Moreover, if $v1$ is the parent of $v2$, then all tuple variables bound in the FROM clause in Q_{v1} are also bound in the FROM clause of Q_{v2}. Finally, notice that Q_v is of the form SELECT-FROM-WHERE, not SELECT-DISTINCT-FROM-WHERE, thus duplicate values may occur in the answer. Fig. 2.4(b) shows the associated SQL queries for each node of our example view (Figure 1.1).

2.5.2 Computation Dependency Graph

For each view ASG node v, the **computation dependency graph** introduced below represents the cardinality between the referenced relations specified by the view query.

Definition 3 Computation Dependency Graph $\mathcal{G}_C$.

1). *Given a view ASG node v computed by SQL query Q_v. Let $R_1, R_2, \ldots, R_n$ be relations referenced by Q_v. Each R_i, $1 \leq i \leq n$ is represented as a node named R_i.*

2). *Let R_i, R_j be two nodes ($R_i \neq R_j$). There is an edge $R_i \rightarrow R_j$ if Q has a join condition of the form R_i.a = R_j.b and R_j.b is UNIQUE in R_j.*

3). *If Q has a join condition R_i.a = R_j.b where R_i.a is UNIQUE for R_i and R_j.b is UNIQUE for R_j, then there are two edges $R_i \rightarrow R_j$ and also $R_j \rightarrow R_i$.*

Fig. 2.6(a) shows the computation dependency graph for the `CI`-node, with `C`-node, `PS`-node, and `P`-node graphs being identical. Fig. 2.6(b) shows the graph for `S`-node.

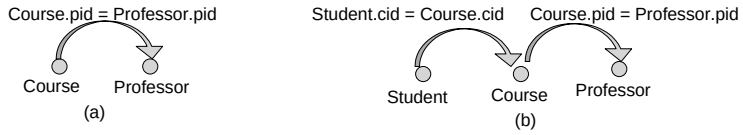


Figure 2.6: (a) $\mathcal{G}_C$ of CI-node and (b) $\mathcal{G}_C$ of S-node

Proposition 1 *Given a view ASG node v computed by SQL query Q_v. Given two nodes R_i, R_j in $\mathcal{G}_C$ of v. If there is a path from R_i to R_j in $\mathcal{G}_C$, then each tuple in R_i will be joined with at most one tuple in R_j. Let V_R be the relational view defined by Q_v. If a node R in $\mathcal{G}_C$, which can reach all other nodes, then there is a 1-1 mapping from V_R to R.*

The property of this 1-1 mapping is useful [Kel86b, DB82, Kel85]. For instance, in $\mathcal{G}_C$ for the CI-node (Fig. 2.6(a)), the *Course* relation can reach all relations in the graph. Therefore there is a 1-1 mapping from the elements of the CI-node in the view to the tuples in the *Course* relation. Thus we can delete a *ClassInfo* element by deleting the corresponding tuple from the *Course* relation without causing side effects on other *ClassInfo* elements.

2.5.3 Foreign Key Graph

Since foreign key propagation could cause side effects, we now introduce the foreign key graph for each view ASG node.

Definition 4 Foreign Key Graph *($\mathcal{G}_{FK}$).*

1). Given a view ASG node v computed by SQL query Q_v. Let $R_1, R_2, \ldots, R_n$ be relations referenced by Q_v. Each R_i, $1 \leq i \leq n$, is denoted as a node named R_i.

2). Let R_i, R_j be two nodes ($R_i \neq R_j$). There is an edge $R_i \rightarrow R_j$ iff Q has a foreign key constraint of the form $R_i.fk \subseteq R_j.key$, where $R_i.fk$ is a foreign key of R_i and $R_j.key$ is the primary key of R_j.

Fig. 2.7 is the foreign key graph of S-node. If a relation R_i in $\mathcal{G}_{FK}$ can reach R, we say R is **referenced** by R_i.

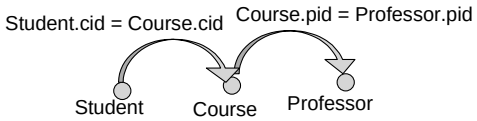


Figure 2.7: $\mathcal{G}_{FK}$ of S-node

We define the **entailment** relationship $\models$ between Q_v and the relational foreign key constraints as follows. Consider the foreign key constraint $R_2.fk \subseteq R_1.key$, denoted as c_{FK}. If Q_v has a join condition of the form $R_2.fk = R_1.key$, then we say that Q entails c_{FK}, denoted by $Q_v \models c_{FK}$. Similarly, let C_{FK} be a set of foreign key constraints. We say $Q_v \models C_{FK}$ if $Q_v \models c_{FK}$ for all $c_{FK} \in C_{FK}$. Intuitively, the entailment means the view follows the foreign key and keeps the 1-1 mapping.

2.6 Bridging the XML and Relational View Update Problem

One possible direction for handling updates over XML views may be to "convert" the XML view update problem to the equivalent relational view update problem (if possible). For this purpose, let us treat the XML view as a "composition" of a set of relational views by following the approach from

the literature [FKS+02, BDH04]. Here, each node in the view annotated schema graph (ASG) of the view (Fig. 2.4) can be considered as generated by a relational view, with an associated SQL query. The set of instances of a schema node is therefore given by this SQL query. Intuitively, an update over a certain schema node can be treated as an update over its relational mapping view. This in turn can be handled as relational view update problem.

However, such a simple transformation between the two problems is not sufficient. The relationship between a parent SQL view and its children SQL views is critical in the XML view scenario for side effect-free checking.

The relationship between the parent SQL view and the child SQL view is explicitly defined by the join constraints specified in the view query (typically, this join is specified in terms of foreign key constraints). Secondly, the relationship is also restricted by the update behavior. In this dissertation work, we will assume the most commonly used update behavior, namely, when we delete an element in the XML view, we will delete all its children elements as well.

The XML view update problem can be viewed as the problem of updating multiple SQL views, with restrictions regarding how updates on one SQL view must affect other SQL views. Therefore, the XML view update problem is more complex than that of pure relational views [BS81, CP84, Kel86b, Kel85, DB82]. Not only do all the problems in the relational context still exist, but we also have to address new challenges derived from the XML hierarchical data model and its flexible update operations. On the other hand, the relational view update problem can be mapped to a special

XML view update problem, where the view only includes a single XML schema node.

2.7 The Restrictions on XQuery Views

The XQuery language provides many features that make queries simpler to write and use, but are also redundant. For instance, complex FLWR expressions can be rewritten as the composition of individual FOR, LET, and IF-THEN-ELSE expressions. The XQuery Formal Semantics [W3C03] defines a proper subset of the XQuery language, called the XQuery Core language, and gives rules that rewrite or normalize every XQuery expression as a XQuery Core expression. The static (type) and dynamic (value) semantics of XQuery is defined on this core language.

XML views defined by XQuery core can be very complex and thus might not be suitable for the purpose of updating. XQuery language supported in this dissertation work as defined in Figure 2.8 is a proper subset of the XQuery Core language. The following restrictions are applied to XQuery core:

- It excludes recursive functions and operators.
- It excludes aggregation functions and operators. This includes count, avg, min, max, sum.

In addition, given that the XML view is specified over the relational database (a flat data model), the backward axis such as parent, ancestor will not appear in the view definition query.

Expr	::=	Literal	
	|	`element` QName Expr	;Element constructor
	|	`attribute` QName Expr	;Attribute constructor
	|	()	;Empty sequence
	|	Expr1 , Expr2	;Sequence constructor
	|	Var	
	|	Expr1 BinOp Expr2	
	|	Expr1 EqOp Literal	
	|	UnaryOp Expr	
	|	(For | Let)+ [Where] [Orderby] return Expr	
	|	Var	;Single step path expression
Literal	::=	String |Integer |Float | . . .	
UnaryOp	::=	+ | - | not	
BinOp	::=	eq | ArithOp | SetOp | LogicalOp	
ArithOp	::=	+ | - | * | div | mod	
LogicalOp	::=	and | or	
SetOp	::=	union Node set operator	
EqOp	::=	eq | lt | le | gt | ge | ne	
Axis	::=	self | child | descendant-or-self | descendant	
For	::=	for $var in Expr	
Let	::=	let $var := Expr	
Where	::=	where Expr	
Orderby	::=	order by Expr	

Figure 2.8: XQuery views handled by our dissertation work

In short, we only consider XML views which can be mapped to a set of relational Select-Project-Join-Union (SPJU) views (with join being an equality join). The reason is that views involving functions or operations, which can not be mapped to SPJU views, are generally not updatable even in the relational scenario. Since the base data is still in the relational data format, such XML views would still be not updatable.

Chapter 3

Theoretical Foundation

3.1 Introduction

3.1.1 Motivation

Given the inherence of the ambiguity of the view updating problem, it is quite common that we cannot find a correct update translation satisfying the criteria under certain update semantics. Such an update should be rejected since it violates the specified update semantic. The *update translatability* issue thus needs to be solved before any update translation procedure can ever be applied. As shown by Example 1 in Section 1.2.3, the mismatch between the hierarchical XML view model and the flat relational base model further complicates the update translatability issue.

Studying update translatability is important in terms of both correctness and performance. Without translatability checking, blindly translating a given view update into relational updates can be dangerous. Such

blind translation may result in unintended view side effects. To identify this, the view before the update and after the update would have to be compared. To adjust for such an error, the view update would have to be rejected and the relational database would have to be recovered for example by rolling back. This would be time consuming and depends on the size of the database. However, by performing an update translatability analysis, such ill-behaved updates could be identified early on and rejected at compile time. The latter would be less costly, depending only on the view query size.

However, before any update translatability checking algorithm under certain semantic can ever be designed, a general purpose theory is needed to guide such algorithm design. This theory should aim to answer the following question:

- How to trace the data from the view to the underlying database? This is important since this relationship represents the data trace or the fragment in the underlying database, which we should consider to change for achieving the given view update.
- How to achieve the given view update? It is essential to identify possible changes to the underlying relational database, which in turn can achieve the given view update.
- How to identify the view side effect? Assuming we choose one translation to achieve the given view update, will this chosen update cause any view side effect? If it does, where could the side effect happen?

The theory should also have the property of being independent from the

view type (XML or relational) and the view definition language (XQuery or SQL), even though the specific computation used to answer these questions could be different. For example, a different SQL query might be issued to identify the data trace for a given view update when view is in relational or XML.

3.1.2 State-of-Art

Significant effort in theory has been made in the relational context to solve the view updating problem.

[BS81, CP84] propose a complementary theory that requires a correct mapping to avoid view side effects as well as database side effects. Database-side-effect-free means that for a translation to be considered correct, it cannot affect any part of the database that is "outside" the view. This correctness criteria, however, is too restrictive to be practical. This is firstly because the view usually only exposes a subset of data from the underlying relational database. Thus there is always some data which is outside of the view (not being exposed). Also, it is possible that there is some "connection" between those exposed and not-exposed data, which could be affected by the view updating behavior. For example, deleting a tuple from one table used by the view, will trigger the foreign key (delete cascading) to delete more tuples from other tables, which might not be exposed in the view.

[DB82] relaxes this condition to only require that no view side effect occurs. In other words, a translation is correct as long as it corresponds exactly to the specified update, and it does not affect anything else in the view.

Using the concept of "clean source", it also characterizes the schema conditions under which an update of a relational view is translatable. Under this relaxed criteria, [Kel86b, Kel85, Kel86a] study the view update translation mechanism for SPJ queries on relations that are in BCNF.

When view is in XML, these theories need to be adjusted, given that the data model is now hierarchical rather than flat. We extend these theoretical works in the following aspects.

3.1.3 Contributions

First, with the hierarchical structure of the XML data model in consideration, our work [WR04, WRMar] extends the concept of a "clean source" for relational databases [DB82] into the concept of a "clean extended source" suitable for XML. We propose a clean extended source theory for determining the existence of a correct relational update translation for a given XML view update.

Second, using the complementary theory, we study the update translatability of XML views over the relational database in the special "round-trip" case [WMR03], which is characterized by a pair of reversible lossless mappings for (i) first loading the XML document into the relational database for storage, and (ii) extracting an XML view identical to the original XML document back out of it. We prove that any *valid* update operation over such XML views, given a pair of round-trip mappings, is always translatable.

3.2 Clean Extended Source Theory

Much work has been done on the existence of a correct translation for various classes of view specifications [BS81, DB82] in the relational context. Especially, Dayal and Bernstein [DB82] use the concept of "clean source" to characterize the schema conditions under which a relational view over a single relational table is updatable. We call it the *clean source theory* [DB82], which has been widely used as theoretical foundation to solve the relational view update problem [CWW00, BKT01].

However, the relational view update translatability problem addressed in [DB82] is different from the XML view update translatability problem we described in Section 1.2.3. In addition, Dayal and Bernstein [DB82] only consider the functional dependencies inside a single relation. However, we notice that the integrity constraints such as foreign keys also deserve careful consideration since (i) in most practical cases, nesting in XML views is done through the Join operation between key and foreign key constrained hierarchies and (ii) the update propagation through the foreign key, which is used to maintain the referential integrity, is one major reason causing view side effects. Considering integrity constraints makes the view update problem harder than considering only updates over a single relation, for such *propagated updates* may again cause view side effects.

We extend it now as *clean extended source theory* to determine whether a given translation is correct for the XML view update problem when foreign key constraints are also considered. The new critical concepts are listed below.

e	A view element
$g(e)$	The generator of e
s	The source of e
$extend(s)$	The extended source of s

We use the following as running example through this section. Fig. 3.1(a) shows a running example of a relational database for a course registration system. A virtual XML view in Fig. 3.1(c) is defined by the *view query* in Fig. 3.1(b).

3.2.1 Extended Source and Clean Extended Source

The key concepts used by our *clean extended source theory* include *extended source* and *clean extended source*.

Definition 5 *Let $R_1, R_2, ..., R_n$ be the set of relations referenced by the SQL query Q of a given view ASG node v. Informally, the* **generator** *of a view element e, denoted by $g(e)$, is a set $\{t_1, t_2, ..., t_n\}$ where $t_i \in R_i$ $(i = 1..n)$, that contains exactly the tuple in R_i used to decide the appearance of e in the view. We say g is the* **generator** *of v, and $\forall t_i \in g$ is a* **source-tuple** *in D of v.*

For example, the generator of the *ClassInfo* element `CI1` in Fig. 3.1(c) is $g(\texttt{CI1}) = \{Professor.t_1, Course.t_1\}$. Note that if Q references a relation more than once (self join), we would use their alias and denote the tuple from each of these relation alias separately in $g(e)$. Our definition of generator follows [DB82] and is the same as *Data lineage* [CWW00] and *Why Provenance* [BKT01].

Definition 6 *Let V^0 be a set of view elements in a given XML view V. Let $G(V^0)$ be the set of generators of V^0 defined by $G(V^0) = \{g \mid g$ is a generator of a view-*

Professor

	pid	pname
t_1	p1	David Finkel
t_2	p2	Tim Merrett

Key:{pid}

Course

	cid	cname	pid
t_1	c1	Math	p1
t_2	c2	Physics	p1
t_3	c3	English	p2

Key={cid}

FK: pid → Professor.pid

Student

	sid	sname	cid
t_1	s1	Chun Zhang	c1
t_2	s2	Mike Fisher	c1
t_3	s3	Feng Lee	c2

Key={sid, cid}

FK: cid → Course.cid

(a) Relational Database

```
FOR $p IN DOCUMENT(Professor/ROW),
    $c IN DOCUMENT(Course/ROW)
WHERE $p.pid = $c.pid
RETURN
    <ClassInfo>
        <Course>
            $c/cname/text()
        </Course>,
        <Professor-Student>
            <Professor>$p/pname/text()</Professor>,
            FOR $s IN DOCUMENT(Student/ROW)
            WHERE $s.cid = $c.cid
            RETURN
                <Student>
                    $s/sname/text()
                </Student>
        </Professor-Student>
    </ClassInfo>
```

(b) View query

```
CI1     <ClassInfo>
CI1.C1      <Course>Math</Course>
CI1.PS1     <Professor-Student>
CI1.PS1.P1      <Professor>David Finkel</Professor>
CI1.PS1.S1      <Student>Chun Zhang</Student>
CI1.PS1.S2      <Student>Mike Fisher</Student>
            </Professor-Student>
        </ClassInfo>
CI2     <ClassInfo>
CI2.C1      <Course>Physics</Course>
CI2.PS1     <Professor-Student>
CI2.PS1.P1      <Professor>David Finkle</Professor>
CI2.PS1.S1      <Student>Feng Lee</Student>
            </Professor-Student>
        </ClassInfo>
CI3     <ClassInfo>
CI3.C1      <Course>English</Course>
CI3.PS1     <Professor-Student>
CI3.PS1.P1      <Professor>Tim Merrett</Professor>
            </Professor-Student>
        </ClassInfo>
```

(c) XML view

Figure 3.1: The running example for the course registration system

element in V^0}. For each $g \in G(V^0)$, let $H(g)$ be some nonempty subset of g. Then $\cup_{g \in G(V^0)} H(g)$ is a **source** *in D of V^0, denoted by s. If $G(V^0) = \emptyset$, then V^0 has no source in D.*

Definition 7 *Let s be a source of V^0. Let E be the set of tuples $\{t_j\}$ from the relations $rel(DEF^V)$, where $\exists t_i \in s$ such that t_j refers to t_i through foreign key constraint(s). We say $s_e = s \cup E$ is an* **extended source** *in D of V^0.*

A *source* includes the underlying relational part of a set of view elements V^0, which is sufficient to decide the appearance of V^0. For example, there are two possible sources of `CI1`, namely, s_1=$\{Professor.t_1\}$, s_2=$\{Course.t_1\}$.

For example, in our example view (Fig. 3.1), consider V^0 as all the *Student* elements of `CI1.PS1` . We have $G(V^0) = \{g_1, g_2\}$, where g_1 =$\{Professor.t_1$, $Course.t_1$, $Student.t_1\}$, $g_2 = \{Professor.t_1$, $Course.t_1$, $Student.t_2\}$. Then $H(g_1)_1 = \{Professor.t_1\}$, $H(g_1)_2 = \{Course.t_1\}$, $H(g_1)_3 = \{Student.t_1\}$. And $H(g_2)_1 = \{Professor.t_1\}$, $H(g_2)_2 = \{Course.t_1\}$, $H(g_2)_3 = \{Student.t_2\}$. Any combination of $H(g_1)_i$ and $H(g_2)_j$ will be a source of V^0, for example, $s_1 = \{Professor.t_1\}$ and s_2 = $\{Student.t_1$, $Student.t_2\}$. Assuming that there is a foreign key from the Course relation to the Professor relation, the extended source of s_1 is given by s_{e1} = $\{Course.t_1$, $Student.t_1$, $Studenst.t_2\}$, while S_{e2} = S_2.

Definition 8 *Let $D = \{R_1, ..., R_n\}$ be a relational database. Let V^0 be a set of view elements in a given XML view V and S_e be an extended source in D of V^0. S_e is a* **clean extended source** *in D of V^0 iff ($\forall v \in V - V^0$), ($\exists g$) such that g is a*

generator in $(R_1 - S_{e1}, ..., R_n - S_{en})$ *of* v*. Or, equivalently,* S_e *is a clean extended source in* D *of* V^0 *iff* $(\forall v \in V - V^0)(S_e \cap g = \emptyset)$*, where* g *is the generator of* v*.*

A **clean extended source** is a source of a view element used only by this particular element and no other one. For instance, s_2 is a clean extended source of e, but s_1 is not since s_1 is also part of the generator of `CI2`.

3.2.2 Clean Extended Source Theory

We now establish a connection between *clean extended source* and update translatability by introducing a series of theorems. The following theorems form a **clean extended source theory**. This serves as the base theory for identifying whether an update is *translatable*. Although somewhat similar to [DB82], the theorems below differ in several important ways. Most notably, (i) the key concepts, such as the *generator, source, extended source* and *clean extended source*, now follow the new definitions from Section 3.2.1 and (ii) XML view elements have different granularity, instead of just the uniform granularity for relational view tuples.

Lemma 1, 2 and 3 below are used to prove Theorem 1 and Theorem 2. Let relational database $D = \{R_1, ..., R_n\}$. Let $dom(V)$ denote the domain of the view. Let V^0 be a set of XML view elements in a given XML view V and $v \in V^0$ indicates that v is a view-element inside V^0.

Lemma 1 *Given a view* V *defined by* DEF^V *over* D*. (1)* S_e *is an extended source in* D *of* V^0 *iff* $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) \subseteq V - V^0$*. (2)* S_e *is a clean extended source in* D *of* V^0 *iff* $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) = V - V^0$*.*

Proof.

(1) *If.* Suppose $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) \subseteq V - V^0$ but S_e is not an extended source in D of V^0. Let $G(V^0)$ be the set of generators of V^0. From definition 7, $\exists(t_1, ..., t_p) \in G(V^0)$ be a generator of $v \in V^0$, such that $(\forall t_i \in R_x) \Rightarrow t_i \notin S_{e_x}$. That is, $t_i \in R_x - S_{e_x}$. Thus $v \in DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n})$. But, $(t_1, ..., t_p)$ is a generator of $v \in V^0$. That is $v \notin V - V^0$. Hence, we have $v \in DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n})$ and $v \notin V - V^0$, a contradiction with the hypothesis that $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) \subseteq V - V^0$.

Only if. Suppose S_e is an extended source in D of V^0 but $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) \not\subseteq V - V^0$. Then, $\exists v$ such that $(v \in DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n})) \wedge (v \in V^0)$. This implies that there is a generator $(t_1, ..., t_p)$ of $v \in V^0$ such that $\{t_i \mid t_i \in R_x \text{ and } R_x \in rel(DEF^V)\} \cap S_e = \emptyset$, contradicting the hypothesis that S_e is an extended source in D of V^0.

(2) *If.* Suppose $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) = V - V^0$ but S_e is not a clean extended source in D of V^0. From (1), S_e is an extended source in D of V^0. By Definition 8, $(\exists v \in V - V^0)$ such that there is no generator $g \in \prod_{R_x \in rel(DEF^V)}(R_x - S_{e_x})$ of v, and hence $v \notin DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n})$, a contradiction.

Only if. Assume that S_e is a clean extended source in D of V^0. By (1), $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) \subseteq V - V^0$. Assuming $V - V^0 \not\subseteq DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n})$, that is, $(\exists v \in V - V^0)$ such that $(v \notin DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}))$. Then there is no generator $g \in \prod_{R_x \in rel(DEF^V)}(R_x - S_{e_x})$ of v. Hence, by Definition 6, there is no source in $(R_1 - S_{e_1}, ..., R_n - S_{e_n})$ of $v \in V - V^0$, which contradicts the hypothesis that S_e is a clean extended source in D of V^0. □

Lemma 2 *Let u^V and U^R be updates on V and D (respectively). Let $v \in V$. Then (U^R deletes an extended source of v and U^R does not insert source-tuple of v) iff $v \notin DEF^V(U^R(D))$.*

Proof.

Let $R'_x = U^R(R_x)$ be one of the updated relation $R_x \in rel(DEF^V)$. Let $T = D - U^R(D)$.

U^R deletes an extended source of $v \in V$

$\iff T$ is an extended source in D of v

$\iff DEF^V(R_1 - T_1, ..., R_n - T_n) \subseteq V - v$ (lemma 1)

$\iff v \notin DEF^V(R_1 - T_1, ..., R_n - T_n)$

$\iff v \notin DEF^V(R_1 \cap R'_1, ..., R_n \cap R'_n)$ since $R_x - T_x = R_x \cap R'_x$

$\overset{(1)}{\iff}$ There is no generator of v in $(R_1 \cap R'_1, ..., R_n \cap R'_n)$.

U^R does not insert an extended source-tuple of $v \in V$

$\overset{(2)}{\iff} \forall R_x \in rel(DEF^V)\ \forall t_i \in R'_x - R_x$, there is no $t_j \in R'_y - R_y$ where $R_y \in rel(DEF^V)$, $x \neq y$, such that $(t_1, ..., t_p)$ is a generator of v.

(1) and (2) hold iff there is no extended-generator in $U^R(D)$ of v. The proposition then follows. □

Lemma 3 *Let u^V, U^R, V, D be as in Lemma 2. Let $v \in dom(V) - V$. Then U^R inserts source-tuples of v iff $v \in DEF^V(U^R(D))$.*

Proof.

U^R inserts source-tuples of v

$\Longleftrightarrow (\exists R_x \in rel(DEF^V), \exists t \in R'_x - R_x)$($t$ is a source tuple in $U^R(D)$ of v)
$\overset{(1)}{\Longleftrightarrow} (\exists g = (t_1, ..., t_p) \in \prod_{R_x \in rel(DEF^V)} R'_x)$($g$ is a generator of v).
$\Longleftrightarrow v \in DEF^V(R'_1, ..., R'_n) = DEF^V(U^R(D))$.

(1) is proven as below:

If. Follow directly from Definition 6.

Only If. Assume that $g = (t_1, ..., t_p)$ is a generator of v, but $\forall\, R_x \in rel(DEF^V)$, $t_i \in R_x$. Then $g \in \prod_{R_i \in rel(DEF^V)} R_i$ and so $v \in DEF^V(R_1, ..., R_n) = DEF^V(D)$, a contradiction. □

The following theorems form the core of the clean-extended source theory. The intuition behind is that an update translation is correct if and only if it deletes or inserts a clean source of the view tuple. Intuitively, it means that the update operation only affects the "private space" of the given view element and will not cause any view side effect. A deletion or insertion is translatable as long as there is a clean extended source of the view element being deleted or inserted.

Theorem 1 *Let u^V be the deletion of a set of view elements $V^d \subseteq V$. Let τ be a translation procedure, $\tau(u^V, D) = U^R$. Then τ* **correctly translates** *u^V to D iff U^R deletes a clean extended source of V^d.*

Proof.

By lemma 1(b), U^R deletes a clean source of V^d
$\Longleftrightarrow DEF^V(R_1 - T_1, ..., R_n - T_n) = V - V^d = u^V(V)$
$\Longleftrightarrow DEF^V(U^R(D)) = u^V(V)$

$\Longleftrightarrow DEF^V(U^R(D)) = u^V(V)$, since $R_i - T_i = R_i'$

$\Longleftrightarrow \tau$ correctly translates u^V to U^R. □

By Definition 2, a correct delete translation is one without any view side effect. This is exactly what deleting a clean extended-source guarantees by Definition 8. Thus Theorem 1 follows.

Theorem 2 *Let u^V be the insertion of a set of view elements V^i into V. Let $V^- = V - V^i$, $V^u = V^i - V$. Let τ be a translation procedure, $\tau(u^V, D) = U^R$. Then τ* **correctly translates** *u^V to D iff (i) $(\forall v \in V^u)$(U^R inserts a source tuple of v) and (ii) $(\forall v \in dom(V) - (V^u \cup V^-))$($U^R$ does not insert a source tuple of v).*

Proof.

By Lemma 3, condition (i) iff $V^u \subseteq DEF^V(U^R(D))$.

Also, since $type(u^V) = insert$ and $type(U^R) = type(u^V)$, $DEF^V(U^R(D)) \supseteq V \supseteq V^-$.

Hence, $V^u \cup V^- \subseteq DEF^V(U^R(D))$.

By Lemma 3, condition (ii) iff $(dom(V) - (V^u \cup V^-)) \cap (DEF^V(U^R(D))) = \emptyset$.

Hence, $DEF^V(U^R(D)) \subseteq V^u \cup V^-$.

Thus, condition (i) and condition (ii) iff $DEF^V(U^R(D)) = V^- \cup V^u = u^V(V)$, that is τ correctly translates u^V to U^R. □

Since $dom(V) - (V^u \cup V^-) = (dom(V) - (V^i \cup V)) \cup (V^i \cap V)$, Theorem 2 indicates a correct insert translation is one without any duplicate

insertion (insert a source of $V^i \cap V$) and any extra insertion (insert a source of $dom(V) - (V^i \cup V)$). That is, it inserts a clean extended source for the new view-element. Duplicate insertion is not allowed by BCNF, while extra insertion will cause a view side effect.

3.2.3 Clean Source Theory on Schema

Section 3.2.2 described the *clean extended source theory*, which is used to determine whether a given translation is correct for the XML view update problem when foreign key constraints are also considered. In short, an update translation is correct if and only if it deletes or inserts a clean source of the view tuple. Intuitively, it means that the update operation only affects the "private space" of the given view element and will not cause any view side effect. The new critical concepts are listed below.

e	A view element
$g(e)$	The generator of e
s	The source of e
$extend(s)$	The extended source of s

However, the update translatability checking based on the clean extended source theory above must examine the actual base data. Unfortunately, as shown by [BKT01], the number of potential translations of a given update can be exponential. Therefore we propose instead to use the schema knowledge to filter out the problematic updates. This prunes the search space in terms of candidates we must consider. We thus introduce a set of corresponding schema-level concepts as below.

Given a view element e and its schema node v. *Schema-level generator* $G(v)$ indicates the set of relations from which the generator $g(e)$ is ex-

v	A view schema node
$G(v)$	The schema-level Generator of v
S	The schema-level Source of v
$Extend(S)$	The schema-level Extended Source of S

tracted. Similarly, S and $Extend(S)$ denote the set of relations the source s and the extended source $extend(s)$ are derived from, named *schema-level source* and *extended source*, respectively. For example, G(CI)={*Professor*, *Course*}. Schema level sources include S_1={*Professor*}, S_2={*Course*}. And $Extend(S_1)$ = {*Professor, Course, Student*}.

3.3 Complementary Theory

3.3.1 Review of the Complementary Theory

The view complement theory in [BS81] 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 [Kel87], 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. We study the complementary theory, which is reviewed below, to solve the XML view updating problem.

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 domain of the database states, denoted by S, is the set of all possible database states. A *data update* of a relational database is a mapping from S into S,

denoted as $\hat{u} : S \rightarrow S$. A *view* V of a given relational database 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.

Definition 9 *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 10 *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 11 *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 12 *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 9 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 11 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 12, 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 3.7 define a mapping f, the query in Figure 3.10 defines a mapping g, then $f \geq g$ and $g \times f \equiv 1$. f is complement of g.

Lemma 4 *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 [DB82] as the "absence of side effects" feature. For the proof of this lemma, please refer to [BS81].

3.3.2 A running example

In the rest of the section, we use the following running example. Figures 3.2 and 3.3 respectively show an XML schema and document representing a book list from an online book store application.

Many XML applications use a relational data store by applying loading strategy such as [STH$^+$99, DD99]. Figures 3.4 and 3.5 show an example relational database generated from the XML schema and data of our running example using a shared inlining loading strategy [STH$^+$99]. 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

```
<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>
```

Figure 3.2: Example XML schema

this table element. Figure 3.6 depicts the default XML view of the database (Figure 3.4).

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 *virtual view*. Such a *virtual view* can be specified by an XQuery expression called a *view query*. Several recent systems such as

```
<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>
```

Figure 3.3: Example XML data

XPERANTO [CKS+00], SilkRoute [FMST01] and Rainbow [ZDW+03] 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 3.7 shows the view query defining a virtual view identical to the originally loaded XML document in Figure 3.3.

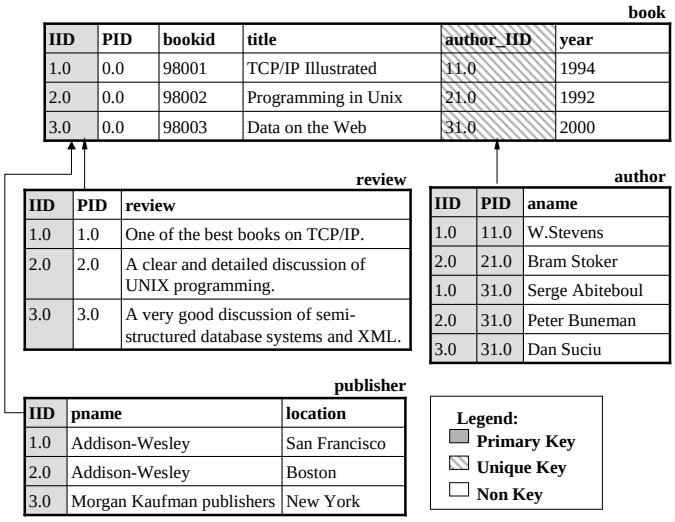


Figure 3.4: Relations in database

```
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 3.5: Database schema of Figure 3.4

```
<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 3.6: Default XML view of database shown in Figure 3.4

3.3.3 The Round-trip XML View Updating (RXU)

We focus on one important case which we name the **round-trip XML view updating scenario** (RXU). Given an XML schema and a valid XML document, by using a suitable loading algorithm, such as inlining [STH$^+$99], edge or universal [DD99], accompanied with a constraint-preserving mapping such as described in [LC00], assume we built a relational database. We

```
<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 3.7: Virtual XQuery view over default XML view shown in Figure 3.6 producing the XML data in Figure 3.3

call it a **structured database**. Further we specify an 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 scenario* (Fig-

ure 3.8).

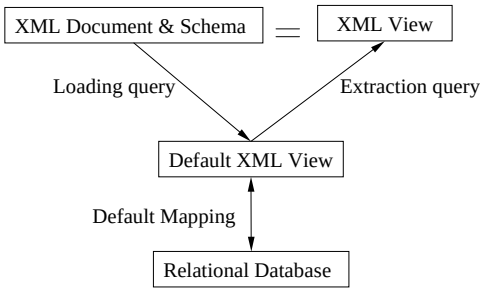


Figure 3.8: Round-Trip update problem

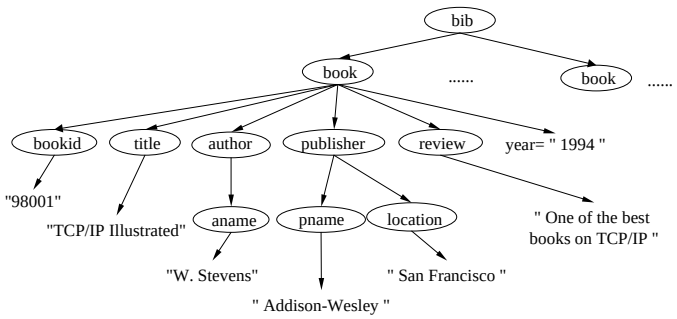


Figure 3.9: Tree representation for XML document shown in Figure 3.3

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 [LC00, STH$^+$99, DD99]. Not only the XML document, but also the XML schema is typically captured in this procedure, which are called data and constraint information respectively.

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* must have exactly the same content as the original document , independent on whatever we may do to the structure.

Definition 13 *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 $D_r \xrightarrow{L'} D_x$.*

Figure 3.9 is a tree structured representation of the XML document in Figure 3.3, while Figure 3.4 is a structured database resulting from applying the inlining loading to that XML document. The extraction query in Figure 3.7 will generate the *twin-view* from the *structured database* of Figure 3.4. Thus this loading is a lossless data loading by Definition 13.

A lossless data loading guarantees to capture all leaves in the XML tree-structured representation (Figure 3.9). 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 [STH+99] and Edge [DD99], are all lossless data loadings.

Constraint Loading Completeness. Given a relational database schema S_r and a view query Q, we define the constraints implied by the XML

```
<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>
```

Figure 3.10: XQuery example

view as *XML View Schema*, which can be extracted by a mapping named *constraint extraction mapping* denoted by $\hat{e}$. In RXU, we assume a lossless constraint loading as defined below.

Definition 14 *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 $S_v = S_x$.*

An XML to relational database loading is a **lossless loading** iff it is both a lossless data loading as defined by Definition 13 and a lossless constraint loading as defined by Definition 14. 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 [DD99] is a lossless data loading, while it is not a lossless constraint loading. In order for such loading strate-

gies to be usable for RXU, it must accompany a constraint preserving loading such as proposed in [LC00].

3.3.4 On the View Updatability in RXU

We now study the updatability of views in the RXU space. The view complementary theory fits our RXU scenario 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.

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.*

Proof.

(i) Since the mapping query defining the default XML view is **1**, according to Definition 10, 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 **0** is the complement of **1**, while $f \equiv \mathbf{1}$, then **0** is the complement view of f. (iii) $\forall u \in U^v$, let $f(s') = uf(s)$, then $0(s') = 0(s)$. Thus, by Lemma 4, u is always translatable. □

Chapter 4

HUX: Schema-centric XML View Updating

4.1 Introduction

As described in Section 1, the update-public semantic requires all updates on the view to be achieved by mapping them into updates over the base data only. No schema change can be made to the base database. No local data can be stored at the view side with the view and used to compute the view content. If a view side effect free translation exists, the view update is accepted and translated. Otherwise, the view update is rejected.

Although Section 3 proposes the clean extended source theory to solve the problem, the update translatability checking directly applying the theory must examine the actual base data. Unfortunately, as shown by [BKT01], the number of potential translations of a given update can be exponential.

On the other hand, the schema knowledge of both the base and the view could be utilized to optimize this analysis. For example, research in the relational database context [DB82] has proposed to identify side effects for some Select-Project-Join views by exploring schema knowledge. A pure schema-based approach is efficient, but rather restrictive. Even for many relational view update cases, it is impossible to decide the translatability by only examining the schema.

4.1.1 Motivating Examples

Given the fact that (1) XML views are hierarchically structured and (2) updates can happen on any element along the view hierarchy, XML view updating is more complex than relational view updating. Fig. 4.1(a) shows a running example of a relational database for a course registration system. A virtual XML view in Fig. 4.1(c) is defined by the *view query* in Fig. 4.1(b). The following examples illustrate cases of classifying updates as translatable or not translatable. XML update language from [TIHW01] or update primitives from [BDH04] is used to define update operations. For simplicity, in the examples below we only use a delete primitive with the format *(delete* `nodeID`*)*, where *nodeID* is the abbreviated identifier of the element to be deleted [1]. For example, `CI1` indicates the first *ClassInfo* element, while `CI1.PS1` the first *Professor-Student* element of the first *ClassInfo* element. We use $Professor.t_1$ to indicate the first tuple of relation *Professor*.

Example 2 *Update* u_1 ={*delete* `CI1.PS1.S2`} *over the XML view in Fig. 4.1*

[1] Note that the view here is still *virtual*. In reality, this `nodeID` is achieved by specifying conditions in the update query [WRMJ05].

Professor

	pid	pname
t_1	p1	David Finkel
t_2	p2	Tim Merrett

Key:{pid}

Course

	cid	cname	pid
t_1	c1	Math	p1
t_2	c2	Physics	p1
t_3	c3	English	p2

Key={cid}

FK: pid → Professor.pid

Student

	sid	sname	cid
t_1	s1	Chun Zhang	c1
t_2	s2	Mike Fisher	c1
t_3	s3	Feng Lee	c2

Key={sid, cid}

FK: cid → Course.cid

(a) Relational Database

```
FOR $p IN DOCUMENT(Professor/ROW),
    $c IN DOCUMENT(Course/ROW)
WHERE $p.pid = $c.pid
RETURN
    <ClassInfo>
        <Course>
            $c/cname/text()
        </Course>,
        <Professor-Student>
            <Professor>$p/pname/text()</Professor>,
            FOR $s IN DOCUMENT(Student/ROW)
            WHERE $s.cid = $c.cid
            RETURN
                <Student>
                    $s/sname/text()
                </Student>
        </Professor-Student>
    </ClassInfo>
```

(b) View query

```
CI1     <ClassInfo>
CI1.C1      <Course>Math</Course>
CI1.PS1     <Professor-Student>
CI1.PS1.P1      <Professor>David Finkel</Professor>
CI1.PS1.S1      <Student>Chun Zhang</Student>
CI1.PS1.S2      <Student>Mike Fisher</Student>
            </Professor-Student>
        </ClassInfo>
CI2     <ClassInfo>
CI2.C1      <Course>Physics</Course>
CI2.PS1     <Professor-Student>
CI2.PS1.P1      <Professor>David Finkle</Professor>
CI2.PS1.S1      <Student>Feng Lee</Student>
            </Professor-Student>
        </ClassInfo>
CI3     <ClassInfo>
CI3.C1      <Course>English</Course>
CI3.PS1     <Professor-Student>
CI3.PS1.P1      <Professor>Tim Merrett</Professor>
            </Professor-Student>
        </ClassInfo>
```

(c) XML view

Figure 4.1: The running example for the course registration system

deletes the student "Mike Fisher". We can delete Student.t_2 to achieve this without causing any view side effect. This can be concluded by looking at the schema of the view. From the view query in Fig. 4.1(b), we know that each student can only appear once in the view, namely, in the ClassInfo element that represents its course-professor-student relationship. In general, deleting any student element in the view can always be translated as deleting the student tuple without causing any side effect. **The schema knowledge is sufficient to decide if an update is translatable**.

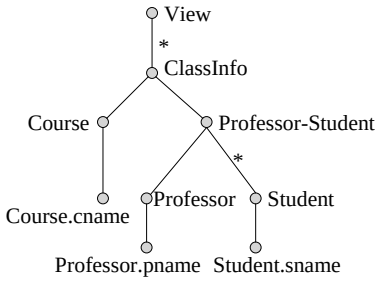
Example 3 *Consider the update u_2 = {delete* `CI1.C1`*}.*
The appearance of the view element `CI1.C1` *is determined by two tuples: Professor.t_1 and Course.t_1. There are three choices for achieving this update: T_1={delete Professor.t_1}, T_2={delete Course.t_1} and T_3={delete Professor.t_1, delete Course.t_1}. All of three translations would cause a view side effect, namely, the whole ClassInfo element would disappear. This conclusion again can be made based on schema knowledge. From the view query, we see that any ClassInfo element must always have a pair of Professor and Course sub-elements. Deleting the course element would break this join condition and thus make the whole ClassInfo element disappear.* **The schema knowledge is sufficient to classify the update as untranslatable**.

Example 4 *For update u_3 = {delete* `CI1`*}, it is easy to see that T_1={delete Course.t_1} will achieve the update without causing any view side effect for the same reason as Example 2. On other hand, T_2={delete Professor.t_1} will cause a side effect since* `CI2` *would disappear. For update u_4 = {delete* `CI3`*}, we find that T_1'={delete Course.t_3} is a correct translation for the same reason as Example 2. T_2'={delete Professor.t_2} is a correct translation since* `CI3` *is the only class Prof. Tim Merrett*

teaches. The difference here indicates that the schema knowledge itself is not sufficient for deciding translatability. **The translatability depends on the actual base data.**

4.1.2 HUX: Handling Updates in XML

As we can see from the above examples, not only view updates can happen anywhere along the view hierarchy, but also side effects can appear anywhere in the view. The XML view side effect checking is thus more complex than in the relational case. A view update can be classified as **translatable** or **untranslatable** using either schema or data knowledge. We aim to support updates of XML views by (i) extending the relational view update solution and (ii) utilizing schema knowledge as much as possible. We thus propose our **schema-centric XML view updating** system named **HUX** (Handling Updates in XML).



CI: Select *
from Professor P, Course C
where P.pid=C.pid
C : Same as CI node
PS: Same as CI node
P : Same as CI node
S : Select *
from Professor P, Course C, Student S
where P.pid=C.pid and C.cid = S.cid

Figure 4.2: Schema graph of the XML view

As we will show in the later sections, the XML hierarchical structure, which is expressed by constraints between the different relational views, increases the complexity of the XML view update problem, but at the same time, this view composition opens more optimization opportunities for ef-

ficient update checking and translation. In other words, we are now able to decide if an update is translatable or untranslatable much earlier, than would otherwise have been possible. As we will show, our schema-centric XML view updating algorithm makes translatability decisions, either in the stage of pure schema-based checking or in the early stage of data-based checking. For instance, in our Example 3, the hierarchy between the relational views of the Course and the Professor tells us that deleting a Professor should not affect any Course element (even in the same CourseInfo element). However, the constraints between the two view elements also indicate that any appearance of a course implies the appearance of the professor. Thus it is impossible to find a correct translation for either deleting a course or deleting a professor.

HUX is a schema-centric solution. Fig. 4.3 shows the HUX framework. The **Schema-driven TranslAtability Reasoning** (STAR) process first filters out all untranslatable updates and classifies some updates as definitely translatable based purely on the schema. Determining these two classes of updates takes polynomial time in the view query size. For updates that cannot be classified by the STAR process, the **Schema-directed Data Checking** (SDC) process examines a subspace of the data (guided by the schema knowledge) to definitely decide whether the update is or is not translatable.

Untranslatable updates are directly rejected with an appropriate error message (indicating the potential side effects). Updates, that successfully pass the STAR or SDC process, are forwarded to the *SQL Update Genera-*

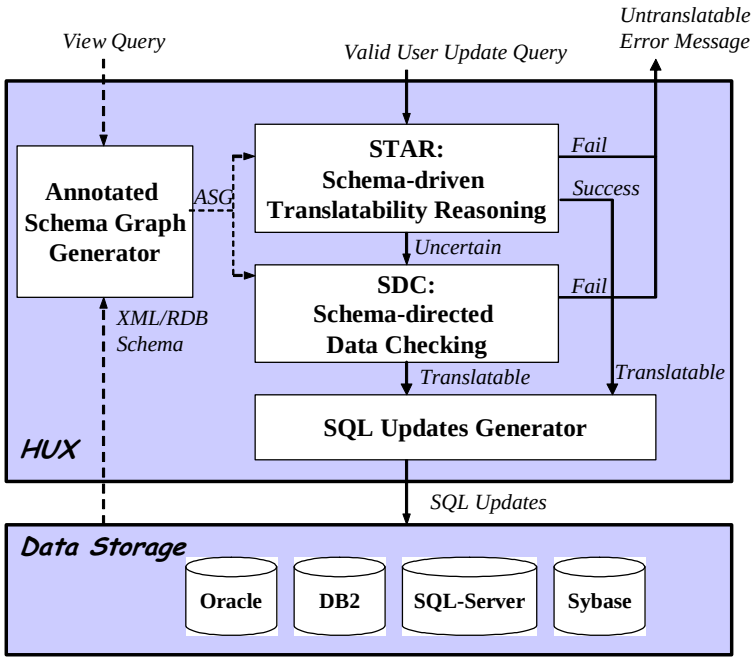


Figure 4.3: The system framework of HUX

tor to produce the correct SQL update statements to be executed over the underlying relational database. HUX guarantees that the generated SQL updates are view side effect-free. Requiring no extra side effect checking or roll back results in a major performance benefit.

Let us illustrate HUX for the motivating examples. During the schema level check, update u_1 is classified as translatable. We translate this update by deleting the corresponding tuple in the $Student$ relation. Update u_2 will be found to be untranslatable by the schema-level check. Updates u_3 and u_4 cannot be classified as translatable or untranslatable by the schema-level check. Therefore we proceed to the data-level check, where we find that u_3 and u_4 are both translatable, and the candidate translations are suggested.

4.1.3 Contributions

In short, we make the following contributions. (1) We propose the first pure data-driven strategy for XML view updating, which guarantees that all updates are fully classified. (2) We also propose a schema-driven update translatability reasoning strategy, which uses schema knowledge including now both keys and foreign keys to efficiently filter out untranslatable and identify translatable updates when possible. (3) We then design an interleaved strategy that optimally combines both schema and data knowledge into one update algorithm, which performs a complete classification in polynomial time for the core subset of XQuery views. (4) We have implemented the algorithms, along with respective optimization techniques in a working XQuery view system named HUX. We report experiments assessing its performance and usefulness.

4.2 Data-driven Side-effect Check

Using clean source theory, most commercial relational database systems [Rys01, BKKM00, CX00] and some research prototypes [BKT01, CWW00] directly issue SQL queries over the base data to identify view side effects. If any clean source (exclusive data lineage [CWW00]) is found, then this source can be a correct translation. Below we extend this approach to find a clean source for updating elements in an XML view. We illustrate the ideas using only deletion examples in our discussion below. However, a similar discussion follows for insertion as shown in Section 4.5.

4.2.1 Partitioning XML View Elements

Suppose we want to update a view element e of a schema node v. Now the translation of this update can update any descendant elements of e in addition to updating e. This translation should *not* affect any of the non-descendant elements of e. We classify these non-descendant elements into three groups as shown in Figure 4.4. **Group-NonDesc** includes view elements whose schema nodes are non-descendant ones of v (does not include v itself). **Group-Self** includes those whose schema node is v. **Group-Desc** includes those whose schema nodes are descendants of v (except e's descendants).

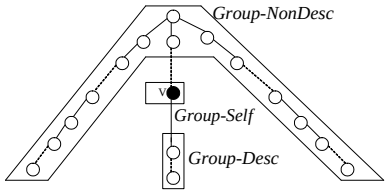


Figure 4.4: Schema Tree Structure

For example, let the view element e be `CI1.PS1`. Then Group-NonDesc includes `CI1, CI2, CI3, CI1.C1, CI2.C1, CI3.C1`. Group-Self includes `CI2.PS1, CI3.PS1`. Group-Desc includes `CI2.PS1.P1`, `CI2.PS1.S1`, `CI3.PS1.P1`.

For updating a view element e, if there is a translation that deletes/inserts e without deleting any existing element or inserting any new element of any of the three groups, then this is the correct translation of the given update. On the other hand, if every translation will affect some element in one of these three groups, then this update is untranslatable.

Given the generator $g(e) = \{R_1^*, R_2^*, ..., R_n^*\}$ of a view element e of a schema node v. Intuitively, deleting any R_i^* from the generator will certainly delete the element e. If R_i^* is not used to generate any element other than e(clean source), then deleting R_i^* is a correct translation that will not cause any side effects. Our three rules below will identify whether R_i^* is used to generate any other element in Group-NonDesc, Group-Self, or Group-Desc respectively.

4.2.2 Checking Side Effects

Let us first consider Group-NonDesc elements. Let v' be a schema node of a Group-NonDesc element, whose corresponding SQL query is $Q^{v'}(\ldots, R_i, \ldots)$. In other words, $Q^{v'}$ uses the relation R_i. The rule below says that R_i^* is not used to generate any element of v' if the result of executing $Q^{v'}(\ldots, R_i^*, \ldots)$ is empty. Here R_i^* is used instead of R_i, while all other relations stay the same.

Rule 1 *Consider Group-NonDesc node v' with $Q^{v'}(\ldots, R_i, \ldots)$ as its SQL annotation. Deleting R_i^* from $g(e)$ will delete the element e without causing side effect on any element e' of v' if $Q^{v'}(\ldots, R_i^*, \ldots) = \emptyset$.*

Proof.

If $Q^{v'}(\ldots, R_i^*, \ldots) = \emptyset$, then $\forall e'$ being an instance element of v', $R_i^* \cap g(e') = \emptyset$. That is, the generator of e' will not be affected by any deletion over R_i^*. Thus, there is no side effect on e'. □

In Example 2, for the element `CI1.PS1.S2`, its generator is {*Professor*,t_1, *Course*.t_1, *Student*.t_2}. Let R^* = {*Student*.t_2}. The Group-NonDesc nodes are `CI, C, PS, P, S`. Since the *Student* relation is not used by any of them, deleting R^* will not cause any side effect on any of their elements.

Now let R^* = {*Course*.t_1}. The *Course* relation is used by all Group-NonDesc nodes. Let us pick the `CI`-node as an example. By executing the SQL query of the `CI`-node: `select * from Professor P,` R^* `where P.pid=`R^*`.pid` we get the result {*Professor*.t_1, *Course*.t_1}. This means *Course*.t_1 is also used by `CI1`, and deleting it will cause side effects.

Let us now consider Group-Self elements. R_i^* is not used by any view element in Group-Self if the result of executing the SQL view query of v over R_i^* only generates e, as shown by the rule below.

Rule 2 *Deleting R_i^* from $g(e)$ will delete e without causing side effect on any other view element e' of v, iff*
$Q^v(\ldots, R_i^*, \ldots) = e$, where Q^v is the SQL annotation of v.

Proof.
If $Q^v(\ldots, R_i^*, \ldots) = e$, then $\forall e'$ being an instance element of v, where $e' \neq e$, $R_i^* \cap g(e') = \emptyset$. That is, the generator of e' will not be affected by any deletion over R_i^*. Thus, there is no side effect on e'. □

Again consider deleting `CI1.PS1.S2`(Mike Fisher). The Group-Self elements are: `CI1.PS1.S1` (Chun Zhang), `CI2.PS1.S1` (Feng Lee). Consider R^* = {*Student*.t_2}. By executing the SQL query: `select * from Professor P, Course C,` R^* `where P.pid=C.pid and C.cid=`R^*`.cid`, we find that

the result is only the element `CI1.PS1.S2`. This means deleting R^* will not cause any side effect on any other Student elements.

On the contrary, if R^*=$\{Course.t_1\}$, then besides "Mike Fisher"(`CI1.PS1.S2`), the query result will also include `CI1.PS1.S1` (Chun Zhang). Thus $Course.t_1$ is also used by `CI1.PS1.S1` and deleting it will cause view side effects.

Let us now consider Group-Desc elements. Consider a schema node v' of a Group-Desc element, whose corresponding SQL query is $Q^{v'}$ (R_1, R_2, ..., R_n, ...). Note that the SQL query corresponding to any schema node that is a descendant of v will include all the relations specified in the SQL query corresponding to v, namely, $R_1, R_2, \ldots, R_n$. The rule below indicates that a Source R_i^* is not used to generate any view element in Group-Desc node v' if the result of executing SQL queries of v' includes only those descendants of e.

Rule 3 *Consider a schema node v' of a Group-Desc element. Let $Q^{v'}$(R_1, R_2, ..., R_n, ...) be its SQL annotation. Deleting R_i^* from $g(e)$ will delete e without causing side effects on any element e' of v' if $T_2 - T_1 = \emptyset$ holds, where T_1 = $Q^{v'}$(R_1^*, R_2^*, ..., R_n^*, ...) and T_2 = $Q^{v'}$(R_1, R_2, ... , R_{i-1}, R_i^*, R_{i+1}, ..., R_n, ...).*

Proof.

First, T_1 includes all view elements whose ancestor includes e. Second, T_2 includes all view elements which uses R_i^* while its schema node is a descendent of v (e's schema node). Thus $T_2 - T_1$ includes all view elements whose schema node is a descendent of v, except descendent elements of e, which is exactly Group-Desc. If $T_2 - T_1 = \emptyset$, then Deleting R_i^* from $g(e)$

will delete e without causing side effects on Group-Desc. □

Consider the element e be `CI1.PS1` of the `PS`-node. Its generator is {*Professor*.t_1, *Course*.t_1}. Let R_1^* = {*Professor*.t_1} and R_2^*={*Course*.t_1}. To find out whether it is used by any sub-elements of e, we first execute the SQL query: `select * from` R_1^*, R_2^* `where` R_1^*`.pid=`R_2^*`.pid`. We find that the descendents of e, denoted by T_1, include: `CI1.PS1.S1` (Chun Zhang) and `CI1.PS1.S2` (Mike Fisher). By executing SQL query: `select * from Professor P,` R_2^* `where P.pid=`R_2^*`.pid`, we find that R_2^* has been used to generate the Student elements (T_2): `CI1.PS1.S1` (Chun Zhang), `CI1.PS1.S2` (Mike Fisher) and `CI2.PS1.S1` (Feng Lee). The difference T_2-T_1 indicates that R_2^* is also used to generate `CI2.PS1.S1` (Feng Lee), which is the side effect of deleting R_2^*.

Observation 2 *If deleting R_i^* from $g(e)$ does not cause side effects on any Group-NonDesc element (using Rule 1), on Group-Self element (using Rule 2), on any Group-Desc element (using Rule 3), then R_i^* is a* **clean source** *of e. Deleting it will not cause any view side effect.*

Proof.

According to Rule 1, deleting R_i^* from $g(e)$ does not cause side effects on any Group-NonDesc element; according to Rule 2, deleting R_i^* from $g(e)$ does not cause side effects on any Group-Self; according to Rule 3, deleting R_i^* from $g(e)$ does not cause side effects on any Group-Desc element.

Given an element e' in the view which is different from e, according to the node partition for the XML views (Section 3.2.3), e always belongs to

one of the three groups: Group-NonDesc, Group-Self, Group-Desc.

Therefore, deleting R_i^* from $g(e)$ does not cause any view side effect. □

When we have foreign key constraints, the checking becomes more complex. To determine whether any side effects are caused by deleting R_i^*, it is also required that tuples being deleted by the foreign key propagation should not in the generators of any other elements. For example, consider a course `CI1.C1 (Math)`. If we delete *Course*.t_1, the *Student*.t_1 and *Student*.t_2 will also be deleted. If any other elements, such as `CI1.PS1.S1`, use *Student*.t_1, the element `CI1.PS1.S1` will disappear from the view as side effect.

4.3 Schema-driven Side-effect Checking

In the previous section, we have described the approach of identifying side effects by *examining the actual base data*. In general this approach is *correct* and *complete*, namely, we can reject all untranslatable updates and identify all translatable updates by always directly examining the data. This step could be quite expensive, however, as we need to ensure no side effects for every schema node by issuing a probe query.

In this section, we propose a more effective solution by using the schema knowledge. The available schema knowledge for XML view updating problem, including the view definition and relational database schema, can be represented by a set of graphs introduced in Section 2.

4.3.1 Schema-level Untranslatable Updates

Utilizing the schema knowledge only, we will now divide an update as *untranslatable* (Example 3) or *translatable* (Example 2). Those we can not make the decision will be labeled as *uncertain*(Example 4), which means that updates are *data dependent*. In other words, the update is translatable for some view instance, while for others the update is untranslatable.

In particular, in this section, utilizing only the schema knowledge, we will illustrate that certain updates can be identified as definitely causing view side effects. These updates will be labeled as *untranslatable*.

Rule 4 *Given a view ASG Node v. Let U be a translation which achieves the deletion of an element of v by deleting the source from the relation $R \in G(v_p)$, where $G(v_p)$ is schema-level generator of the parent node of v. U is guaranteed to always cause a side effect on elements of v_p.*

Proof.

Given any view element e of the ASG node v. Let e_p be the parent view element of e, whose schema node is v_p. Let $t \in R$ be the tuple in R to be deleted to achieve the given view update (deleting e). Then $t \in g(e)$ and $t \in g(e_p)$. Deleting t will make e_p to disappear from the view as a side effect. □

Since $G(v_p) \subseteq G(v)$ always holds, deleting from $G(v_p)$, shared by $G(v)$, will definitely cause a side effect, namely the parent element will disappear. For example, to delete `CI1.PS1.S1`, both *Professor*.t_1 and *Course*.t_1 will cause side effects on their parent elements `CI1` and `CI1.PS1`.

Given an ASG node v, let's now consider the non-ancestor Group-NonDesc nodes v'. Deleting an element of v will cause a side effect on some element of v', if there is an onto mapping from elements of v' to elements of v. Equivalently, if an element e of v exists, there is always an element e' of v', such that $g(e) \subseteq g(e')$. Clearly, if we were to delete e, then in that case e' will also disappear.

For example, in our example view, there is an onto mapping from the P-node to the C-node. Whenever a *Professor* element appears in a *ClassInfo* element, there is always a *Course* element in the same *ClassInfo* element, and vice versa. Hence, deleting either of them will certainly cause view side effects on the other one.

Below we introduce the *participation graph* $\mathcal{G}_P(v', v)$ of the ASG node v' with respect to node v, which is used to identify the onto mapping from elements of v' to elements of v.

Definition 15 Participation Graph $\mathcal{G}_P(v', v)$.

1). *Given a view ASG node v computed by SQL query Q^v. Let v' be a non-ancestor Group-NonDesc group node computed by SQL query $Q^{v'}(R_1, R_2, \ldots, R_n)$. Each R_i ($1 \leq i \leq n$) is denoted as a node with name R_i.*

2). *Let R_i, R_j be two nodes ($R_i \neq R_j$). There is an edge $R_i \rightarrow R_j$ iff $Q^{v'}$ includes a join condition of the form R_i.fk = R_j.key, where R_i.fk is the foreign key of R_i and R_j.key is the primary key for R_j. There is an edge R_i — R_j iff $Q^{v'}$ has a general join constraint R_i.a = R_j.b.*

3). *A set of nodes $R_{i1}, ..., R_{ik}$ form a* **partition** *P iff they are directly or indirectly connected.*

4). Let R_i, R_j be two nodes such that $R_i \in P_i$, $R_j \in P_j$ and $P_i \neq P_j$. There is an edge $R_i \rightarrow R_j$ in $\mathcal{G}_P$ iff there is an edge $R_i \rightarrow R_j$ in $\mathcal{G}_{FK}$.

5). Let R_i, R_j be two nodes ($R_i \neq R_j$). There is an edge $R_i \leftrightarrow R_j$ iff $C_v \models C_{v'}$, where C_v is the set of constraints between R_i and R_j in Q^v, while $C_{v'}$ is the set of constraints between R_i and R_j in $Q^{v'}$.

Consider Example 3 of deleting the professor "David Finkel" (`CI1.PS1.P1`). Here v is the `S`-node. Its non-ancestor Group-NonDesc nodes are {`S`-node, `C`-node}. As shown below the participation graph with respect to `P`-node for the `C`-node is shown in Fig. 4.5(a) and for the `S`-node is shown in Fig. 4.5(b).

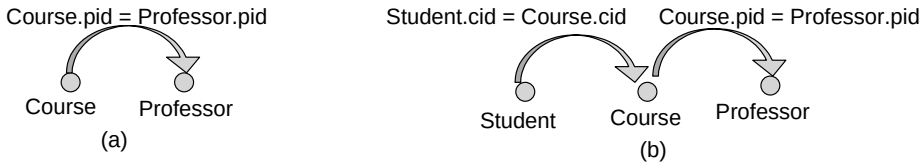


Figure 4.5: (a) $\mathcal{G}_P(C, P)$ and (b) $\mathcal{G}_P(S, P)$

Intuitively, the directed path $R_1 \rightarrow R_2 \rightarrow \ldots \rightarrow R_n$ in $\mathcal{G}_P(v', v)$ means that if any tuple $t_1 \in R_1$ is used by a view element e' of v', there is always a set of tuples $\{t_2,...,t_n\}$ being used by e' as well. Thus there is a one-to-one mapping from tuples in R_1 to view elements of v'. Proposition 2 indicates this one-to-one mapping. Lemma 5 is used to prove this proposition.

Lemma 5 *Given two nodes R_i and R_j in $\mathcal{G}_P(v', v)$, if there is a directed path from R_i to R_j, then $\forall t \in R_i$ participates in the generator of an element of v'), denoted by $g(e')$, $\exists t' \in R_j$ such that t' participate in $g(e')$ as well.*

Proof.

We prove this lemma by examining each item in the definition of the participation graph (Definition 15). We point each of them by the step number.

Step #2. An edge $R_i \rightarrow R_j$ from a join condition R_i.fk = R_j.key certainly satisfies. Since foreign key is forced by the relational database schema, the Lemma holds. A general join condition $R_i.a = R_j.b$, however, overwrites any foreign key constraint, which might exist between R_i and R_j. The reason is that the matching tuple in R_j would not participate in the same generator $g(e)$.

Step #4. Now consider edges between the partitions, which is implied by the relationship between v and v'. Again, the foreign key expression, which is not exposed in the view, implies that $\exists t' \in R_j$ such that t' participates in $g(e')$. Thus there should be an edge from R_i to R_j.

Step #5. Given any view element e of v, the condition C_v is certainly satisfied. Since $C_v \models C_{v'}$, $C_{v'}$ is satisfied too. Then $\forall t \in R_i$ participates in the generator of an element of v'), denoted by $g(e')$, $\exists t' \in R_j$ such that t' participates in $g(e')$ as well. □

Intuitively, Lemma 5 means that a path in the participation graph $\mathcal{G}_P(v', v)$ implies the following: whenever there is a tuple in R_i that participates in generating the view element e', there always exists a tuple t' in R_j, which also contributes to generating e'.

Proposition 2 *Given a non-ancestor Group-NonDesc node v' and its participation graph $\mathcal{G}_P(v', v)$. If a node R in $\mathcal{G}_P(v', v)$ can reach every other nodes through a directed path, then there is a onto mapping from tuples in R, which participates*

in generating elements of v, to the view elements of v'.

Proof.

According to Lemma 5, if R in $\mathcal{G}_P(v', v)$ can reach every other node through a directed path, then $\forall t \in R$, there exist a set of tuples $\{t_1, t_2, ..., t_n\}$, which (along with t) will form the generator for an element e' of v'. Thus there is a onto mapping from tuples in R, which participates in generating elements of v, to the view elements of v'. □

For example, according to Fig. 4.5(a) there is a one-to-one mapping from the *Course* relation to the *Course* elements in the view; according to Fig. 4.5(b), there is a one-to-one mapping from the *Student* relation to *Student* elements in the view.

To guarantee the onto mapping from elements of v' to elements of v, three conditions have to hold. First, the schema-level generator of v', denoted by $G(v')$, and the schema-level generator of v, denoted by $G(v)$, share some common relations. Without shared relations, deleting v will certainly not cause any side effect on v'. Second, for shared relations, the tuple set used by the to-be deleted element e is also used by some elements of v'. Third, if there are other relations referenced by v' but not v, there always exists a set of tuples in these relations, which will certainly produce an element of v'.

Proposition 3 *Let $T = G(v) \cap G(v')$ and $\tilde{T} = G(v') - G(v)$. There is an onto mapping from elements of v' to elements of v if the following two conditions hold. (i) T is a strongly connected component in $\mathcal{G}_P(v', v)$. (ii) Either $\tilde{T} = \emptyset$, or, $\tilde{T} \neq \emptyset$*

but $\forall R \in T, \forall R' \in \tilde{T}$*, there is a directed path from* R *to* R' *in* $\mathcal{G}_P(v', v)$*.*

Proof.

(i) If T is a strongly connected component in $\mathcal{G}_P(v', v)$, then $\forall R \in T, \forall R' \in T$ where $R \neq R'$, there is a directed path from R to R'.

(ii) if $\tilde{\mathrm{T}} = \emptyset$, then $\forall R \in G(v')$, R can reach every other relation in $\mathcal{G}_P(v', v)$. According to Lemma 5, there is an onto mapping from R to elements of v'. This implies that there is an onto mapping from elements of v to elements of v'.

(ii) If $\tilde{\mathrm{T}} \neq \emptyset$ but $\forall R \in T, \forall R' \in \tilde{\mathrm{T}}$, there is a directed path from R to R' in $\mathcal{G}_P(v', v)$. Then $\forall R \in T$, R can reach every other relation in $\mathcal{G}_P(v', v)$. According to Lemma 5, there is an onto mapping from R to elements of v'. This also implies that there is an onto mapping from elements of v to elements of v'. □

In our example, to check whether there is an onto mapping from elements of `C`-node (v') to elements of `P`-node (v), we compute T = {*Professor, Course*} and $\tilde{T} = \emptyset$. It is easy to see that T is a strongly connected component in Fig. 4.5(a). The onto mapping exists. Deleting any source of `CI1.PS1.P1` will certainly cause side effects on `CI1.PS1.C1`. Thus based on the schema knowledge, we can infer that updates on any *Professor* element are never translatable. This conclusion is *data independent*.

On the contrary, let v' = `S`-node and v = `P`-node. Then we have T = {*Professor,Course*} and $\tilde{T}$ = {*Student*}. There is no directed path from the *Professor* node or the *Course* node to the *Student* node in $\mathcal{G}_P(v', v)$ (Fig. 4.5(b)).

Thus, there is no onto mapping from *Student* elements to *Professor* elements. We cannot infer anything about side effects on *Student* elements. The following rule describes the scenario of guaranteed side effects on non-ancestor Group-NonDesc nodes.

Rule 5 *Given a view ASG node v. Deleting an element of v will cause side effects on some element of v', which is v's non-ancestor nodes from Group-NonDesc, iff there is an onto mapping from elements of v' to the elements of v.*

Proof.

The onto mapping from elements of v' to the elements of v means that if there exists an element e' of v in the view, there certainly exists an element e' of v' in the view. Thus deleting e from the view will certainly cause side effect of making e' disappear. □

4.3.2 Schema-level Translatable Updates

For some updates, the schema knowledge alone can be utilized to decide whether the update is **translatable**, meaning translatable independent of the actual data. In Example 2, deleting any *Student* element is always translatable. To find this, we have to check whether a clean source always exists for any update on the schema node.

The following rules are used to identify whether it is possible to delete a source without ever causing any side effects on Group-NonDesc, Group-Self and Group-Desc nodes.

First, deleting the source S of v will not cause any side effect on v' if its generator never uses any relation from the schema-level extended source Extend(S).

Rule 6 *Given a view ASG node v. Deleting a source $S \in G(v)$ of v will not cause any view side effect on view element of node v' in Group-NonDesc if Extend(S) $\cap$ G(v') = $\emptyset$.*

Proof.

By the definition of the *Extended Source*, *Extend(S_i)* includes all the relations that will be affected if the generator tuple from S_i is deleted. Let v' be a non-descendent schema node of v. Since Extend(S_i) $\cap$ G(v') = $\emptyset$, relations in $G(v')$ will not be updated by this deletion. Therefore, any view element of v' will not be affected. □

Second, deleting from the source S of an element e of v will not cause side effects on other elements of v, as long as (i) there is a one-to-one mapping from S to the elements of v and (ii) any foreign keys referring to S are entailed by the SQL query of v (Rule 7).

Rule 7 *Given a view ASG node v and its computation dependency graph $\mathcal{G}_C$. Deleting a source S of v will not cause any view side effect in any view element of Group-Self if the following two conditions hold: (i) The corresponding node of S in $\mathcal{G}_C$ can reach all other nodes. (ii) Let $C_{FK}(S)$ be the foreign key constraints from any relation $R \in G(v)$ referring to S. Then $Q \models C_{FK}(S)$, with $\models$ be the entailment defined in Section 2.5.*

Proof.

Let V_R be the relational view defined by Q. First, according to Proposition 1, if there exists a node R in $\mathcal{G}_C$ which can reach all other nodes, then there is a 1-1 mapping from view tuples of V_R to base tuples in the relation R. Let e be an element of v to be deleted. Let's use s_i to denote the source of e from relation R. Thus if condition (i) is satisfied, this indicates that the source s_i does not contribute to any view element of v other than e.

Further, when foreign key constraints exist, if $Q \models C_{FK}(S_i)$ then any tuple reference to s_i indeed contributes to the element e only through the foreign key connection, rather than other elements. Namely, the 1-1 mapping does not conflict with the foreign key constraints.

In summary, if both conditions (i) and (ii) hold, then this means that deleting s_i will not affect any other view element of its schema node. □

For example, deleting a student element `CI1.PS1.S1` (Chun Zhang) by deleting *Student*.t_1 for the *Student* relation will not cause side effects on any other student element (e.g., Mike Fisher, Feng Lee), since the above rule holds. However, if we choose to delete *Professor*.t_1, then student "Feng Lee" will also disappear from the view.

Third, deleting from the source S of an element e of v will not cause side effects on any other element of the descendent node of v, if those descendent nodes are well nested. For example, deleting the *professor* element `CI1.PS1.P1` will not cause side effects on *student* elements `CI1.PS1.S1` and `CI1.PS1.S2`, since the Student relation is well nested with respect to the

foreign keys.

Rule 8 *Deleting a source S of v will not cause any view side effect in any view element of node v' in Group-Desc if $C_V \models C_{FK}(S)$, where $C_{FK}(S)$ is the set of foreign key constraints from any relation $R \in G(v')$ referring to S.*

Proof.

If $Q \models C_{FK}(S)$, then let s_i be the generator tuple from the Source S_i. Let E_d be the set of tuples which refer to s_i directly or indirectly through foreign key constraints. Then for any element $e_{d_j} \in E_d$, whose schema node is v', if e_{d_j} is not descendent of e, then the tuple in E_d will not contribute towards generating e_{d_j}. Thus no side effect on any view element of node v' in Group-Desc will arise. □

Observation 3 *For a given view update u, if a translation U satisfies rules 6, 7 and 8, U is guaranteed to be view side effect-free.*

Proof.

Let e be the view element to be deleted from the view by u. Let v be the schema node of e. Then a different view element e' will certainly belong to Group-NonDesc, Group-Desc or Group-Self. If the translation U will not cause any side effect on any of those three groups, then U deletes e only (side effect free). □

For example, deleting any student element is always translatable. This is why we call Example 2 as data independent translatable.

4.3.3 Data Dependency

If we cannot classify an update into either translatable or untranslatable using Observation 3, we say that the translatability of an update is *data dependent*. Techniques from Section 4.2 could now be utilized. While more expensive, they guarantee to find a definite answer.

4.4 Schema-centric XML View Updating Algorithm

In Section 4.2, we show that performing fully data-driven check will correctly classify updates according to their translatability. While correct, this classification is very expensive. One critical issue in XML view updating is thus how to prune the search space for correct translations. Section 4.3 proposes a schema based translatability checking technique that identifies all the data independent untranslatable or translatable cases. Thus it efficiently prunes the search space. Ideally the search space could be fully examined using only schema-based checks. However, given the flexibility of the view definition, we may still require to examine the data for the remaining relations. In this section, we propose a schema-centric algorithm to combine the power of both schema-based and data-based approaches. This includes a schema-based translatability reasoning (STAR) step, a schema-directed data checking (SDC) step, and finally an efficient SQL update generation step. The algorithm is fully implemented in HUX system as shown in Fig. 4.3.

4.4.1 STAR: Schema-driven TrAnslatability Reasoning

Given an update u deleting a single view element e from an XML view. Let v be the schema node of e in the view ASG $\mathcal{G}_V$. By default, u can be achieved by deleting any tuple $t \in g(e)$. However, to guarantee the view side effect free property, all elements of schema nodes from Group-NonDesc, Group-Self and Group-Desc need to be examined for side effects. Initially, the search space SS_0 of v for finding a schema-level clean extended source is generated using Algorithm 1.

Algorithm 1 Initialize the search space

```
Let G(v) be schema-level generator of v.
for all relation R_i of schema-level generator G(v) do
  Add R_i as column name in SS_0
  for all R_j ∈ Extend(R_i) − G(v) do
    Add R_j as extended column name in SS_0
  end for
end for
for all view ASG nodes v' do
  Add v' as row name in SS_0
  for all R_i ∈ G(v') do
    Add × into cell (v', R_i) in SS_0
  end for
end for
```

The initial search space SS_0 for the S-node in our example is shown below. Here we use R_P, R_C and R_S to denote the *Professor*, *Course* and *Student* relations.

(a) The Generators

nodes	Generator
CI-node	(R_P, R_C)
C-node	(R_P, R_C)
PS-node	(R_P, R_C)
P-node	(R_P, R_C)
S-node	(R_P, R_C, R_S)

(b) The search space SS_0 of S-node

	Sources		Extended
nodes	R_P	R_C	R_S
CI-node	×	×	
C-node	×	×	
PS-node	×	×	
P-node	×	×	
S-node	×	×	×

Figure 4.6: Initialize the search space of S-node

The **schema-level untranslatable updates** rules (Rule 4 and 9) are first applied as described in Algorithm 2.

Algorithm 2 STAR-Untranslatable

```
Let current search space SS = SS_0
/*Horizontal-Prune*/
for all row v' in SS, where v' ≠ v do
  Let p be the parent node of v in G_v
  if G(v') − G(p) = ∅ then
    Delete row v
  end if
end for

/*Vertical-Prune-I (Rule 4)*/
for all column R_i in SS do
  if (v_p, R_i) is initialized then
    Delete column R_i
  end if
end for

if size(Sources) ≠ ∅ then
  /*Vertical-Prune-II)*/
  for all column R_i in SS do
    if (Rule 9 holds then
      Delete column R_i
    end if
  end for
end if

if size(Sources) ≠ ∅ then
  return SS
end if
```

Below we show step-by-step how the **STAR-Untranslatable** algorithm is used to prune the initial search space SS_0. First, the *Horizontal-prune* reduces the search space by eliminating two rows from Fig. 4.6(b): `P`-node and `C`-node. Both of them share exactly the same generator with their parent node (`CI` and `PS`). Any side effects on them will also be captured by their parent node. Thus it is sufficient to just check side effects on `CI` and `PS` nodes. Second, the *Vertical-prune-I* will eliminate two columns from Fig. 4.6(b): R_P and R_C, since deleting from them will certainly cause side effects on `PS`-node, which is the parent of v (`S`-node). The `Vertical-Prune-II` will not further reduce the search space since the non-ancester Group-NonDesc nodes are empty.

Next, we can identify translatable updates by purely using the schema knowledge. Algorithm 3 applies **schema-level translatable** rules.

For example, the search space for `S`-node after the STAR-translatable

After Horizontal-Prune

nodes	R_P	R_C	R_S
CI-node	×	×	
PS-node	×	×	
S-node	×	×	×

After Vertical-Prune-I

nodes	R_S
S-node	×

After Vertical-Prune-II
(Same as above)

Figure 4.7: Search space of S-node after STAR-untranslatable

Algorithm 3 STAR-Translatable

Let current search space be SS

for all Group-NonDesc node v' in SS **do**
  for all column R_i in Sources of SS **do**
    if Rule 6 holds **then**
      mark $\surd$ in the cell (v', R_i)
    end if
  end for
end for

for all Group-Self node v in SS **do**
  for all column R_i in Sources of SS **do**
    if Rule 7 holds **then**
      mark $\surd$ in the cell (v, R_i)
    end if
  end for
end for

for all Group-Desc node v' in SS **do**
  for all column R_i in Sources of SS **do**
    if Rule 8 holds **then**
      mark $\surd$ in the cell (v', R_i)
    end if
  end for
end for

Let CS be the set for clean sources
for all column R_i in Sources of SS **do**
  Let clean = TRUE
  for all row v in SS **do**
    if mark(v,R_i) = × **then**
      clean = FALSE; Continue
    end if
  end for
  if clean = TRUE **then**
    Add R_i to CS
  end if
end for
return CS

algorithm (Algorithm 3) is shown in Fig. 4.8. At this point, we can conclude that deleting an element of S-node can always be achieved without causing any side effect by deleting from the Student relation.

nodes	R_S
S-node	$\surd$

Figure 4.8: Search space of S-node after STAR-translatable

4.4.2 SDC: Schema-directed Data Checking

Given the current search space SS of node v from the STAR algorithms, each $\times$ mark left in the cell (v', R_i) indicates the potential view side effect on v', if the view update is achieved by deleting from R_i. The certainty, however, depends on the actual relational data. For these nodes, we need to issue a probe query over the relational database to check for potential side effects (Section 4.2).

nodes	R_P	R_C	R_S
CI-node	$\times$	$\surd$	
S-node	$\surd$	$\surd$	$\surd$

Figure 4.9: Search space of CI-node after STAR algorithm

Consider our motivating example 4. The search space of the CI-node is shown in Fig. 4.9. To decide whether we can delete CI1 by deleting from the *Professor* relation, we need to check whether deleting the tuple *Professor*.t_1 will cause any side effect on other *ClassInfo* elements. Let R_C^* = {*Professor*.t_1} be the result of the probe query `Select * from` R_P^* `P, Course C where P.pid = C.pid`, which includes both CI1 and CI2. This means *Professor*.t_1 is also used by CI2. Thus it is not a clean source for CI1.

However, assume we want to delete CI3. Let R_C^*={Professor.t_2}. The same probe query will generate CI3 only. This means {*Professor*.t_2} is not used by any other *ClassInfo* element. Thus we can now change the $\times$ mark in the cell (CI-node, R_P) to $\surd$. Thus R_C^*={*Professor*.t_2} is a clean source of CI3.

If our goal is to get all possible translations, all these probe queries need to be performed over the actual data. If our goal is to find the first correct

translation, then the data check is performed only if the STAR algorithm does not find any clean source ($CS = \emptyset$). Since the latter is commonly used by both commercial and research projects, Algorithm 4 corresponds to this strategy.

Algorithm 4 Schema-directed Data Checking

```
Let CS = ∅.
Let SS be the search space from STAR
Compute the generator of e: R1*, R2*, ..., Rn*
for all column Ri in SS do
  for all row v do
    if Rule 2 does not hold then
      Continue
    end if
    if Rule 1 does not hold then
      Continue
    end if
    if Rule 3 does not hold then
      Continue
    end if
    Add Ri into CS
    Return CS
  end for
end for
```

4.4.3 SQL Update Generation

Updates that successfully pass the STAR procedure and the SDC procedure will finally reach the *SQL update generator* to form the SQL update statements. The suggestions on possible correct translations are also carried along. Algorithm 5 creates the SQL updates.

In Example 2, the generator of *CI1.PS1.S1* is $\{Professor.t_1, Course.t_1, Student.t_1\}$. Let R_C^*=$\{Student.t_1\}$. The finally generated SQL update is "`Delete from Student where rowid in` R_C^*". Since the view side effect has been checked by the STAR and SDC procedure, the generated SQL updates will be executed over the relational database directly, without worrying about any

Algorithm 5 SQL Update Generation

```
/* To delete an element e of node v from the view */
Compute the generator of e: R*_1, R*_2, ..., R*_n
Let CS be the schema-level clean sources from STAR algorithm
if CS ≠ ∅ then
   Pick the first R_i from CS
   Generate SQL statement:
   DELETE FROM R_i WHERE ROWID IN R*_i
end if
```

view side effect.

4.5 Schema-driven Side Effect Checking For Insertion

Insertion is supported by HUX in a similar fashion with deletions. By view side effect in insertion, we mean that inserting an element e might cause inserting an element e' ($e \neq e'$) into the view. Intuitively, to avoid the side effect, we need either (i) the translation of e will not form a generator for e' or (ii) even if it does form the generator for e', it can be eliminated without affecting anyone else in the view. Using clean source theory, we have the following observation.

Proposition 4 *To achieve insertion of e into the view by inserting T=$\{t_1$,t_2...,$t_n\}$ into the relations $R_1, ..., R_n$ will not cause view side effect of inserting another view element e', if: (i) $\nexists g(e')$ such that e' appears in the view, or, (ii) $\exists g(e')$ and e' can be safely deleted by removing $t \in g(e') - T$.*

This proposition is straightforward, that is, side effect will only appear if the generator of another view element has been formed and can not be removed without touching those inserted data.

The challenge now is how to achieve this goal by utilizing the schema knowledge. Given a view ASG node v. Let $INS(v)$ denote the set of relations into which an insertion of a v's element will insert. The $INS(v)$ is computed as follows: $INS(v)$ = ($G(v) \cup G$(Group-Desc)) - $G(v_p)$, where G(Group-Desc) is the union of the generators of Group-Desc nodes of v, and v_p is the parent node of v. For example, INS(CI) = {*Professor, Course, Student*} and INS(S)={*Student*}.

For illustration purpose, we extend the motivation example in Section 4.1.1 by adding a new ProfessorList node. Figure 4.10 shows the view and Figure 4.11 shows the ASG for this new XML view.

```
FOR $p IN DOCUMENT(Professor/ROW),
  $c IN DOCUMENT(Course/ROW)
WHERE $p.pid = $c.pid
RETURN
  <ClassInfo>
    <Course>
      $c/cname/text()
    </Course>,
    <Professor-Student>
      <Professor>$p/pname/text()</Professor>,
      FOR $s IN DOCUMENT(Student/ROW)
      WHERE $s.cid = $c.cid
      RETURN
        <Student>
          $s/sname/text()
        </Student>
    </Professor-Student>
  </ClassInfo>
FOR $p IN DOCUMENT(Professor/ROW)
RETURN
  <Professor-List>
    $p/pname/text()
  </Professor-List>
```

```
CI1       <ClassInfo>
CI1.C1      <Course>Math</Course>
CI1.PS1     <Professor-Student>
CI1.PS1.P1  <Professor>David Finkel</Professor>
CI1.PS1.S1  <Student>Chun Zhang</Student>
CI1.PS1.S2  <Student>Mike Fisher</Student>
            </Professor-Student>
          </ClassInfo>
CI2       <ClassInfo>
CI2.C1      <Course>Physics</Course>
CI2.PS1     <Professor-Student>
CI2.PS1.P1  <Professor>David Finkle</Professor>
CI2.PS1.S1  <Student>Feng Lee</Student>
            </Professor-Student>
          </ClassInfo>
CI3       <ClassInfo>
CI3.C1      <Course>English</Course>
CI3.PS1     <Professor-Student>
CI3.PS1.P1  <Professor>Tim Merrett</Professor>
            </Professor-Student>
          </ClassInfo>
PL1       <ProfessorList>David Finkel<ProfessorList>
PL2       <ProfessorList>Tim Merrett<ProfessorList>
```

Figure 4.10: The view query used for insertion illustration

Again, we can perform a complete classification by examining side effects on each of the three groups in Fig. 4.4. This is very similar to the dele-

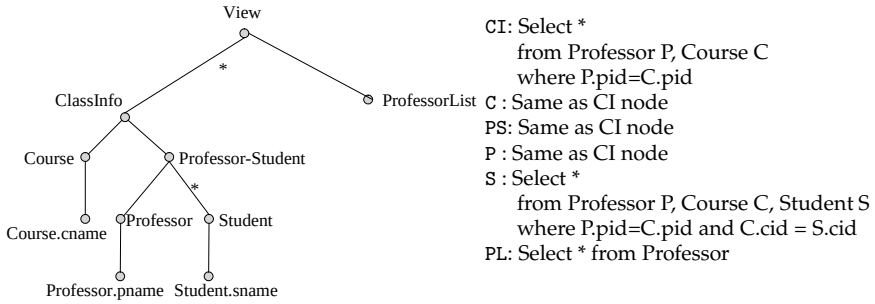


Figure 4.11: Schema graph of the XML view

tion scenario except that we are now examining every relation in $INS(v)$ instead of $G(v)$.

Algorithm 6 InsT-Mark — STAR marking algorithm for insertion

Input:
$\mathcal{G}_V$, $\mathcal{G}_C$, $\mathcal{G}_{FK}$
Output: The marked $\mathcal{G}_V$

for every schema node v of $\mathcal{G}_V$ **do**
  InsT-NonDesc-SideEffect(v, $\mathcal{G}_V$, $\mathcal{G}_C$, $\mathcal{G}_{FK}$)
  if InsT(v) = unsafe-insert **then**
    CONTINUE
  end if
  InsT-Self-SideEffect(v, $\mathcal{G}_V$, $\mathcal{G}_C$, $\mathcal{G}_{FK}$)
  if InsT(v) = unsafe-insert **then**
    CONTINUE
  end if
  InsT-Desc-SideEffect(v, $\mathcal{G}_V$, $\mathcal{G}_C$, $\mathcal{G}_{FK}$)
  if InsT(v) is marked as unsafe-insert **then**
    CONTINUE
  else
    InsT(v)=safe-insert
  end if
end for

First of all, the schema-level untranslatable insert rule (similar to Rule 9 in the deletion scenario) is still applicable for insert scenario.

Rule 9 *Given a view ASG node v. Inserting an element of v will cause side effects on some element of v', which is v's non-ancestor nodes from Group-NonDesc, iff*

there is an onto mapping from elements of v' to the elements of v.

Proof.

Let T be the set of tuples to be inserted into the database in order to achieve the view update (inserting an element e of v into the view). Then T will at least form the generator of e, namely, $g(e) \subset T$. Since there is an onto mapping from elements of v' to elements of v, then there exists an element of v', whose generator will also be formed. Namely, $g(e') \subset T$. This onto mapping also implies that e' can not be eliminated without making e disappear. Thus, there will always be a side effect on elements of v'. □

4.5.1 Step1 — Group-NonDesc Examination

We first examine the view ASG nodes of Group-NonDesc. To insert a new Course of `C`-node into view, we check whether any other view elements will be inserted (e.g., a new ClassInfo element of `CI`-node).

When we insert an element e of a view ASG node v, we first determine which view ASG nodes have side-effects – that is, for which view ASG node an element could get inserted. Note that an element of a node v' will not get inserted if (i) the generator of v' does not overlap with the relations into which we insert tuples when we insert e, or, (ii) we know that there is a relation R' required by the generator of v', but the tuple in R' does not exist currently (e.g., this can be guaranteed by foreign key constraints), and so no element of v' will be generated.

Rule 1 *(**Side effects examination for Group-NonDesc**)*

1). *Inserting an element of v will not cause any view side effect in Group-NonDesc node v' if $G(v') \cap INS(v) = \emptyset$.*

2). *Inserting an element of v will not cause any view side effect in Group-NonDesc node v' if $\exists R \in (G(v') \cap INS(v))$, $\exists\ R' \in (G(v') - INS(v))$ that references R through foreign key constraint(s) $C_{FK}(R, R')$ and $Q \models C_{FK}(R, R')$.*

Proof.

First, by the definition of $INS(v)$, it includes all relations into which we could possibly insert. Let v' be a non-descendent schema node of v. Since $G(v') \cap INS(v) = \emptyset$, none of relations in $G(v')$ will be updated. Thus any view element of v' will not be affected.

Second, let $T = G(v') \cap INS(v)$. If $T \neq \emptyset$, then there could be potential view side effects on v'. However, given a relation R, if there is a relation R' $\in (G(v') - INS(v))$ referring to R through a foreign key, and this foreign key condition is entailed by the view query, then the generator of v' can never be formed. The reason is that the tuple of R', which is required by the join condition through the foreign key to form the generator of v', does not exist. Thus any view element of v' will not be affected. □

If condition (i) is satisfied, the generator $G(v')$ does not overlap with the relations being inserted $INS(v)$, and will not be affected at all. For example, consider inserting a Student of `S`-node. Since INS(`S`-node) = {Student}. This insertion will not affect any Course element of `C`-node, nor Professor element of `P`-node.

If condition (ii) is satisfied, the generator of any element of v' cannot be formed. To insert an element of v, we will insert into one or more relations in $INS(v)$. Suppose we insert a tuple t into relation R; as R' references R, no tuple in R' references t. For example, inserting a new ProfessorList PL3 of PL-node will not cause side effect on CI-node. The reason is that we need an *existing* Course tuple with PL3 as its professor to form a ClassInfo element in the view. This cannot happen since there is a foreign key from Class to Professor table.

Rule 2 is implemented by Algorithm 7. Fig. 4.12 shows the progressive application of the algorithm **InsT-NonDesc-SideEffect** to identify side-effect on Group-NonDes. For example, inserting a ClassInfo of CI-node will cause side effect on ProfessorList of PL-node, and inserting a professor of a ClassInfo (CI-node) will cause side effect on Course (C-node).

View ASG Node		CI-node	PS-node	S-node	PL-node
G(v)		{Professor, Course}	{Professor, Course}	{Professor, Course, Student}	{Professor}
INS(v)		{Professor, Course, Student}	{Professor, Student}	{Stdent}	{Professor}
CS(v)		{Course}	{}	{Student}	{}
Step 1	1	{PL-node}	{CI-node,S-node,PL-node}	{CI-node,S-node,PL-node}	{CI-node,S-node,PL-node}
	2	{PL-node}	{CI-node,S-node,PL-node}	{}	{CI-node,S-node,PL-node}
	3	{PL-node}	{CI-node,S-node}	{}	{CI-node,PS-node,S-node
	4	{PL-node}	{CI-node,S-node}	{}	{CI-node,S-node}
	5	{PL-node}	{CI-node}	{}	{}
	6	unsafe-insert	unsafe-insert	-	-

Figure 4.12: Step 1 of STAR marking for insertion

Rule 2 only determines view ASG nodes where there could be side-effects. Below we further examine whether these side-effects can be removed. We will do this in two steps. First, we need to find view ASG nodes where side-effect "actually" happens. Second, we will examine whether the side-effects on such nodes can be removed. The following proposition

Algorithm 7 InsT-NonDesc-SideEffect — Side effect examination for Group-NonDesc nodes

Input:
v: The schema node of the element e (to be inserted)
$\mathcal{G}_V, \mathcal{G}_C, \mathcal{G}_{FK}$
Output:

Compute $INS(v)$
Let $Group_{NDesc}$ be the non-descendent nodes set from $\mathcal{G}_V$
for every node v' in $Group_{NDesc}$ **do**
  Compute the Generator $G(v')$
  Let *Intersect* = $G(v') \cap INS(v)$
  if *Intersect* = $\emptyset$ **then**
    Remove v' from $Group_{NDesc}$, CONTINUE
  end if
  Let *Diff* = $G(v') - INS(v)$
  Let *IsRemoved* = FALSE
  while !*IsRemoved* AND *Intersect* $\neq \emptyset$ **do**
    Get next relation $R \in$ *Intersect*
    while !*IsRemoved* AND *Diff* $\neq \emptyset$ **do**
      Get next relation $R' \in Diff$
      Compute $C_{FK}(R, R')$
      Let Q be the SQL query of v'
      if ($C_{FK}(R, R') \neq \emptyset$) AND ($Q \models C_{FK}(R, R')$) **then**
        Remove v' from $Group_{NDesc}$
        IsRemoved = TRUE
      end if
    end while
  end while
end for
if $Group_{NDesc}$ is not empty **then**
  InsT(v)=unsafe-insert
end if

identifies view ASG nodes for which side-effects actually happen.

Proposition 5 *Let v' be a Group-NonDesc node, and p be the parent node of v'. Assume $(G(v') - G(v_p)) \cap INS(v) = \emptyset$. (i) An insertion causing side effects on v' also causes side effect on v_p. (ii) Eliminating side effect on v_p will also eliminate the side effect on v'. We say that side effect does not actually happen on v'.*

Proof.

First, assume $G(v') - G(v_p) = \emptyset$, then $G(v') = G(v_p)$. Thus both (i) and (ii) hold. Second, Let *Diff* = $(G(v') - G(v_p))$ and $Diff \neq \emptyset$. Since $Diff \cap INS(v) = \emptyset$, if the view side effect ever appears, it is never caused by *Diff*. In other words, the side effect can only be caused by updating a relation in $G(v_p)$. Thus any side effects on v' will also be side effect on v_p. (ii) follows trivially. □

For example, we do not consider $Student$, because G(S-node) - G(`CI`-node) includes only Student, which does not overlap with INS(PL-node). Any side effect that appears on S-node again implies side effect on `CI`-node, and can be eliminated by removing side effects on `CI`-node.

As another example, when inserting a new ProfessorList `PL3` of `PL`-node into the view, there could be a side effect on `PS`-node. But, whenever there is a side effect on `PS`-node (a professor appears under the ClassInfo), there certainly will be a side effect on `CI`-node as well (a ClassInfo element appears in the view). Therefore, we say that side-effect does not actually happen for `CI`-node, and we need not examine `PS`-node's side-effect. Instead, we need to consider only `CI`-node's side-effect. Also, if we eliminate the

side effect on ClassInfo (delete the Course tuple), we can also eliminate the side effect on its ProfessorStudent (PS elements disappear as well).

For these nodes where $(G(v') - G(v_p)) \cap INS(v) \neq \emptyset$, we eliminate the side-effects of v' by the following rule.

Rule 2 *(**Side effects elimination for Group-NonDesc**) Consider a Group-NonDesc node v' where there is a side effect. Let v_p denote v's parent and $(G(v') - G(v_p)) \cap INS(v) \neq \emptyset$. The view side effect on v' can be eliminated if $CS(v') - INS(v) \neq \emptyset$.*

Proof.

If $CS(v') - INS(v) \neq \emptyset$, then $\forall R \in (CS(v') - INS(v))$, R is a clean source. We thus can always use it to eliminate the view side effects. □

For example, to insert a new professorList `PL3` of `PL`-node into the view, we have INS(`PL`-node)={Professor}. We need to consider the side effect on `CI`-node. From our STAR-marking deletion algorithm (Section 4.4), we get the clean source candidates CS(`CI`-node)={Course}. And, since CS(`CI`) - INS(`PL`-node)={Course}, we can always eliminate the side-effect by deleting the corresponding Course.

4.5.2 Step 2 — Group-Self Examination

For those ASG nodes that passed Step1 examination, in this step we check the side effect of inserting an element of v on other view elements of v. For instance, we determine whether an insertion of a new review `CI1.PS1.S2` of `S`-node will cause the insertion of another new Student element.

Rule 3 *(***Side effects examination for Group-Self***) Inserting an element of v will not cause any view side effect of inserting an element of Group-Self, if $\forall R \in (G(v) - G(v_p))$, R can reach all other nodes in $\mathcal{G}_C$.*

Proof.

First, according to the proposition 1 in Section 2.5, since R can reach all other nodes in $\mathcal{G}_C$, there is a 1-1 mapping from tuples of any relation $R \in G(v) - G(v_p)$ to elements of v. Thus inserting into R will not cause any side effect in Group-Self. Second, any other relation S in $INS(v)$, namely, $INS(v) - (G(v) - G(v_p))$ is never been used by $G(v)$. Thus inserting into S will not cause any side effect in Group-Self either. □

It is straight forward that if a schema node is generated from single relation only, inserting an element will not cause any side effect of inserting a new element of this schema node. In our example, adding a new publisher `PL3` of `ProfessorList` will not cause side effect of another `ProfessorList` element (a new professorList) to appear.

As another example, consider the extreme case, where each Professor can only teach one course. Namely, let's assume the *pid* is a unique key of Course relation as well. Then the computation graph of `CI`-node will be: Course $\rightleftharpoons$ Professor. INS(`CI`-node) = {Professor, Course, Student}. Each element of `CI`-node will map to a unique tuple in Course, as well as in Professor. Thus inserting into Course or Professor will not cause side effect on other `CI`-node element in this case. On the other hand, inserting into

Student will not cause side effect either.

Algorithm 8 InsT-Self-SideEffect — Side effects examination for Group-Self nodes

Input:
v: The schema node of the element e (to be inserted)
$\mathcal{G}_V, \mathcal{G}_C, \mathcal{G}_{FK}$
Output:

Compute the Generator $G(v)$, $G(v_p)$
for all relation $R \in G(v) - G(v_p)$ **do**
  for all relation $R' \in (G(v) - G(v_p) - R)$ **do**
    if R can not reach R' in $\mathcal{G}_C$ **then**
      InsT(v) = unsafe-insert, RETURN
    end if
  end for
end for

Algorithm 8 (**InsT-Self-SideEffect**) is used to implement Rule 3. Similarly, some side effects can be eliminated using the rule below.

Side-effect Eliminating Rule. *View side effect in Group-Self node v caused by inserting an element of v can be eliminated if DelT(v)=safe-delete.*

Proof.

Assume inserting a view element e of v causes view side effects of inserting another view element e' of v. If DelT(v)=safe-delete, then $\exists R \in G(v)$, deleting the generator tuple of e' from R can safely remove e', thus eliminate the side effects. □

Side-effect eliminating in this step is also straightforward. If v is marked as safe-delete, we always can eliminate the side effect safely. For example, we try to insert a new ClassInfo element `CI4` of `CI`-node. Since `CI`-

node is marked as safe-delete, it thus passes the side-effect eliminating test. To achieve the update by inserting into the Professor relation might cause other CI-node element, say CI5 to appear in the view, but CI5 can always be removed by deleting the *Course* tuple from its generator.

4.5.3 Step 3 — Group-Desc Examination

Having the result of Step 2, we now check whether we can insert a view element e without affecting other view elements whose schema nodes are in Group-Desc. For example, to insert CI4 into Fig. 4.1, we need to examine whether any other new view element (e.g.,CI2.P1) will also be inserted.

Rule 4 *(**Side effects examination for Group-Desc**) Inserting an element of v will not cause any view side effect in Group-Desc node v', if $\exists R \in INS(v)$, $\exists\ R' \in (G(v') - INS(v))$ that references R through a foreign key constraint(s) $C_{FK}(R, R')$ and $Q \models C_{FK}(R, R')$.*

Proof.

same as the proof of Rule 2. □

As an example, consider inserting an element e of CI-node in our example view. This insertion will not cause any view side effect on any element of S-node. The reason is that G(S-node) - INS(CI-node) = {Student}. The *Student* relation refer to *Course* relation through a foreign key C_{FK}($Course$, $Student$) = (*Student.cid* $\subseteq$ *Course.cid*). As we can see, $Q \models C_{FK}(Course, Student)$. Thus no side effect appears on any element of S-node.

Algorithm 9 InsT-Desc-SideEffect — Side effects examination for Group-Desc nodes

Input:
v: The schema node of the element e (to be inserted)
$\mathcal{G}_V, \mathcal{G}_C, \mathcal{G}_{FK}$
Output:

Compute $INS(v)$
Let $Group_{Desc}$ be the non-descendent nodes set from $\mathcal{G}_V$
for every node v' in $Group_{Desc}$ **do**
  Compute the Generator $G(v')$
  Let *Diff* = $G(v') - INS(v)$
  Let *IsRemoved* = FALSE
  while !*IsRemoved* AND $INS(v) \neq \emptyset$ **do**
    Get next relation $R \in INS(v)$
    while !*IsRemoved* AND *Diff* $\neq \emptyset$ **do**
      Get next relation $R' \in$ *Diff*
      Compute $C_{FK}(R, R')$
      if ($C_{FK}(R, R') \neq \emptyset$) and ($Q \models C_{FK}(R, R')$) **then**
        Remove v' from $Group_{Desc}$
        IsRemoved = TRUE
      end if
    end while
  end while
end for
if $Group_{Desc}$ is not empty **then**
  InsT(v)=unsafe-insert, RETURN
end if

Algorithm 9 (**InsT-Desc-SideEffect**) can be used to implement Rule 4. Similar to Step1, to determine whether these side-effects can be removed, we need the following two steps. First, we need to find view ASG nodes where side-effect "actually" happens. Second, we will examine whether side-effects on such nodes can be removed. Proposition 5 also holds for Group-Desc nodes and will be used to identify view ASG nodes for which side-effects actually happen. The side effect eliminating is also the same with Step1 as described below.

Side-effect Eliminating Rule. Consider a Group-Desc node v' where there is a side effect. Let v_p denote v's parent and $(G(v') - G(v_p)) \cap INS(v) \neq \emptyset$. The *view side effect on v' can be eliminated if $CS(v') - INS(v) \neq \emptyset$.*

Proof.

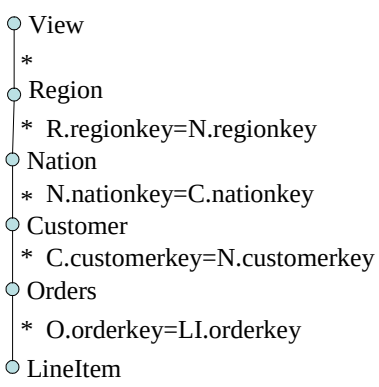
Same as the proof of the side-effect eliminating rule of Step 1. □

As long as the generator of descendant node v' cannot be formed, there is no way the side effect could appear. Even if the side effect could appear, we still have a clean extended source to eliminate the side effect.

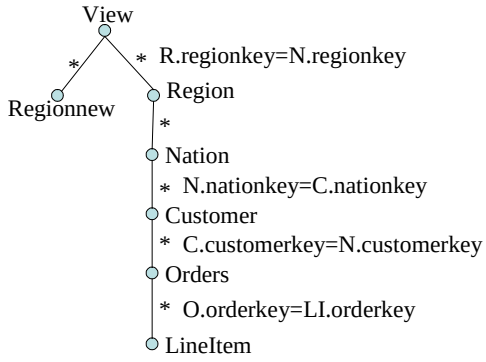
4.6 Evaluation

We conducted several experiments to address the performance impact of our system. The test system used is a dual Intel(R) PentiumIII 1GHz processor, 1G memory, running SuSe Linux and Oracle 10g. The relational

database is built using TPC-H benchmark [TPC]. Two views are used in our experiments, with their view ASGs shown below.



WellNestedView



DupSiblingView

We note that the schema-level translatability decision holds for the same-type of updates on the given schema node. Thus we can perform STAR algorithm at compile-time. The view ASG is marked according to the decision of STAR algorithm. When an update comes, it is first evaluated by checking the compile-time mark. The performance of STAR marking and STAR checking in HUX system is shown by Fig. 4.13. The time of STAR marking procedure increases linearly with the size of the view query. The STAR checking time stays stable.

We now compare the performance of our system with other possible update systems.

4.6.1 HUX vs. Naive View Update System

We compare the performance of HUX against two different naive approaches. The first naive approach, which we call the *N.G.System*, performs a view

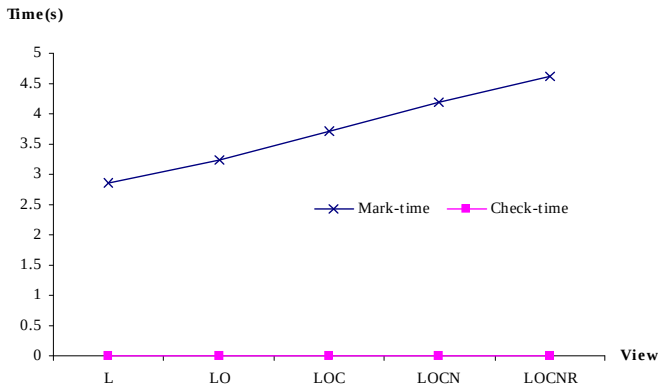


Figure 4.13: The performance of HUX system

update without checking for any side effects. For instance, given a view element to be deleted, the N.G.System deletes one of the tuples in its generator. Note that the N.G.System has to do little work to find the translation. For an accurate comparison in our experiments, we will assume that the N.G.System performs the same translation as what HUX would have found.

Fig. 4.14 shows the performance of HUX vs. N.G.System, when only key constraints are considered for *DupSiblingView*. The database used is 1G. As we can see, HUX takes a little bit more time (around 10ms) for deleting a customer, lineitem, order or region element, which are all identified as *translatable* by our STAR-translatable algorithm. This difference is the time spent on the STAR algorithm to get the translatability decision. Now let us consider deleting nation element (which is not translatable). The naive system chooses to delete the source from the relation Nation. This will cause view side effect. Our HUX system instead will reject it directly using the result from STAR algorithm. The cost is negligible. The difference in

deleting the *regionnew* is huge, since HUX now has to perform the SDC to decide whether it is side effect free. The N.G.System, however, finds the correct translation directly (not possible in reality), and only have the cost of actually deleting the data from the database.

Similar conclusion holds when the foreign key constraints are considered (Figure 4.15), except that the time spent on deleting a tuple is much more due to the delete cascading.

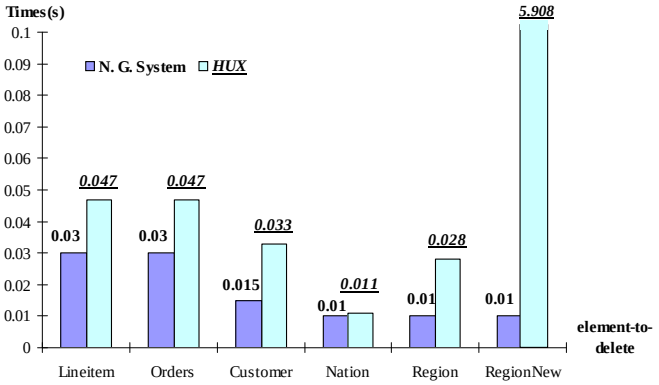


Figure 4.14: HUX vs. N.G.System with only key constraints

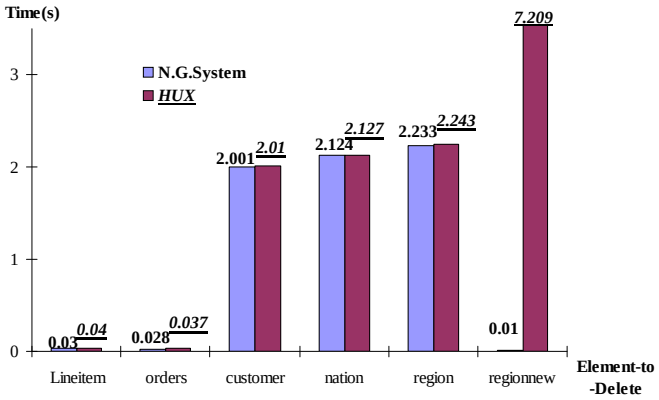


Figure 4.15: HUX vs. N.G.System with foreign keys

The second naive approach against which we compare HUX is one that checks for side-effects by comparing the views before and after the update. This approach is very expensive.

4.6.2 HUX vs. Data-driven View Update System

Several relational database systems [BKT01, CWW00], which support the view updating based on pure data checking, are available. We compare HUX with these systems by performing updates only on the "highest" complex node of the view, as shown in Fig. 4.16. For our *WellNestedView*, we only delete a region element. We use five different well nested views, each with a different number of relations. The update is always to delete an element from the bottom most schema node. HUX only takes the STAR checking time, which is very small. The pure data-driven system, however, has to perform probe queries on the actual data to find the correct translation. In the best case, it will find the correct one by the very first probe. This already takes much more time than HUX. In the worst case, it will find the correct translation in the last probe.

Motivated by this result, we compare HUX with the system which performs pure data-driven check as in Section 4.2. As shown by Fig. 4.17, the cost of pure data-driven increases rapidly with the view size, while HUX stays efficient.

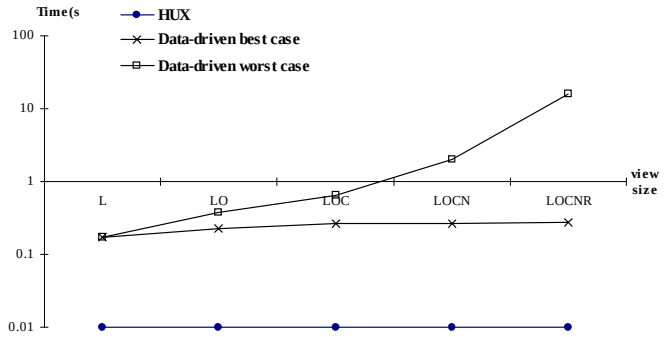


Figure 4.16: HUX vs. the relational view update system

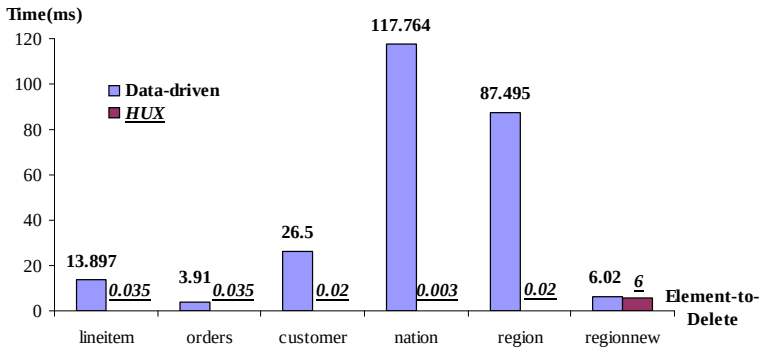


Figure 4.17: HUX vs. Pure data-based XML view update system

4.6.3 Complexity and Usefulness of HUX

We restrict our queries to XQuery queries that can be translated into a view ASG. The view ASG used in HUX has the same limitations as the view forest from SilkRoute [FKS+02]. This property requires these queries to be in XQueryCore [W3C03], with the exception of recursive functions, order related functions and aggregation functions. In addition, the SQL query of each schema node in the view ASG is of the form SELECT-FROM-WHERE,

not SELECT-DISTINCT-FROM-WHERE. Thus duplicate values may occur in the view. In [BKT01], the authors study the complexity of the update translatability problem in the case of deletion over relational SPJU views. They show that this problem is poly-time solvable with respect to the size of the database for SPU and SJ views, whereas it is NP-hard for PJ and JU views. Note that Project here implicitly eliminates the duplicates. However, when Project does not eliminate duplicates, the translatability of updates over PJ views (actually over SPJ views) is polynomial.

Since we restrict the view query handled by our ASGs, our views are actually a combination of SPJ views (in XML format), where the project does not eliminate duplicates. Therefore our overall algorithm is polynomial — our STAR procedure runs in poly-time in the size of the view query (also shown by our experiment in Fig. 4.13); the SDC uses the SQL engine, and runs in poly-time in the database size.

To examine the practicality of our approach, we studied the Protein Sequence Database (PSD) from [Res]. From typical user studies over this domain [Res] gained by discussion with biologist (Ryder, Elizabeth F) at WPI, we observed that the *well-nested view* assumed by [BDH04], where the nesting "follows" the key and foreign key constraints, is not often the case in this domain. Our approach hence provides a practical solution to this domain, because it supports even non-well-nested views.

4.7 Related Work

[Kel86b, Kel85, Mas84] study the view update translation mechanism for

SPJ queries on relations that are in BCNF. These works have been further extended for object-based views in [BSKW91]. Commercial database systems, such as Oracle [BKKM00], DB2 [CX00] and SQL-Server [Rys01], also provide XML support. [TIHW01] assumes that the update is indeed translatable and has in fact already been translated into updates over a relational database. They also study the performance of executing the translated updates by using relational techniques, such as triggers or indices. Our work addresses the *view update translatability*, an aspect different than *update translation strategy*.

An abstract formulation of the update translatability problem is given by the *view complementary theory* in [BS81, CP84]. It uses the invariance of the complement of a view, namely *database side-effect free*, to decide the translatability of a given update. However, this property is too restrictive to be practical. [DB82] relaxes the criteria for a correct translation as only requiring *view side-effect free*. Based on the notion of a *clean source*, it presents an approach in the relational context for determining the existence of update translations by performing a syntax analysis of the view definition.

Recent works [BDH04, BDHar] study the update over *well-nested* XML views. They assume joins are through keys and foreign keys, and nesting is controlled to agree with the integrity constraints and to avoid duplication. [LL92] develops a theory within the framework of the ER approach to characterize the conditions under which mappings exist. It is further extended in [CLL02] to guide the design of valid XML views. Valid views based on this design approach are a proper subset of general XML views studied in this paper. [CLL02] avoids the duplication from joins and multi-

ple references to the relations. Our work in this paper is *orthogonal* to these works by addressing new challenges related to the decision of translation existence when no particular restrictions have been placed on the defined views. That is, in general, conflicts are possible and a view cannot always be guaranteed to be well-nested [BDH04] or valid [CLL02] (as assumed by these prior works).

Commercial database systems, such as Oracle, DB2 and SQL-Server, also provide XML support. **Oracle** XML DB [BKKM00] provides SQL/XML as an extension to SQL, using functions and operators to query and access XML content as part of normal SQL operations, and also to provide methods for generating XML from the result of an SQL Select statement. The **IBM DB2** XML Extender [CX00] provides user-defined functions to store and retrieve XML documents in XML columns, as well as to extract XML elements or attribute values. However, neither IBM nor Oracle support update operations. [Rys01] introduces XML view updates in **SQL-Server2000**, based on a specific *annotated schema* and update language called *updategrams*. Instead of using update statements, the user provides a before and after image of the view. The system computes the difference between the image and generates corresponding SQL statements to reflect changes on the relational database.

As part of our data-level check we are able to analyze the performance of existing work [BDH04]. This leads us to suggest alternative approaches that can work with existing DBMS without imposing additional requirements, and that yields better performance.

Recent works [BKT01, CWW00] indicate a loose connection between

data provenance [BKT01] or *lineage* [CWW00] and the view update problem. The distinction between "why provenance" and "where provenance" is used to guide the view update process to find an appropriate update translation. Their work has several similarities with ours, e.g., to try to find the data trace (provenance) at the query syntax level. However, we utilize this data trace or provenance for a different purpose. The question that [BKT01] tries to answer is: given two equivalent queries that are rewritings of each other, when are the provenance guaranteed to be identical? Instead, we use the provenance to determine if a correct translation exists, for a given update.

In addition to the relational framework [BS81, CP84, DB82, CA81], the view update problem has been attacked from various logical vantage points. One method extends the semantics of the database to express some or all the possible correct translations of a view update [FUV83, RN89, Wil86] or to directly store the view updates and provide new semantics for the database [LLS93]. This approach increases the complexity of query processing. The opposite approach to restrict the class of translations in an attempt to compute a unique result [Heg90], has also been studied. We believe this work is limited in scope. Another approach classis and deals with each type of ambiguity in computes the implications of a view update translation and presents decisions to resolve ambiguity to the database administrator. We believe that correctness is a property of interest to database administrators and would be reported by these editors. The addition of integrity constraints clearly impacts translations and [TA91] considers deletion but defines insertion as the insertion of ID facts. For data log, [JMN83]

considers view updates for deletion but defines insertion as the insertion of IDB facts. Another methods [Bry90, Dec90, KM90], closely related to conjunctive query containment, generate all possible translations of a view update. The number of possible translations of a view update is very large, and we believe that usually a database administrator knows the correct translation of a view update and simply needs a language to express the translation.

Chapter 5

LoGo: Localized Write-through View Updates Services

5.1 Introduction

So far, we have introduced HUX (Chapter 4) for update-public semantic. Update-local semantic is also very common. We address it in this chapter.

5.1.1 Motivating Problem

Consider a typical view application domain such as scientific data sharing, e.g., in the Human Genome Project [HGP].

A public database (such as the NCBI gene bank) has first been built and thereafter has been commonly used and extended as appropriate by scientists in related areas. Scientists use this as well as other public databases by either directly querying and updating over the database or through a view.

In many practical cases, scientists with update capabilities may prefer to first keep his research results (updates) "local" instead of always immediately updating the public database through their views with all their local new findings. Reasons for this are plentiful. For example, new identified gene information still needs to be verified before it goes public. Another reason will be competitiveness. Scientists may want to keep their discoveries private as long as possible. This local data can be modified by the user for the purpose of overwriting any previous change. This requirement is called *update localization*.

Once the local data is ready (e.g., scientific articles reporting the finished genome sequence have been published), scientists (*subject view* user) may want to (or have to) release the qualified data into the "public" database, so that other scientists (*object view*) can gain access to this shared data. Other scientists (*object view* user) can choose either to "synchronize" their local database (if it is not empty) with the public database, or, to leave their local customized data as it is. We call these requirements *data merging* versus *synchronization*.

This is referred as the update-local semantic (introduced in Chapter 1).

5.1.2 State-of-Art

The above identified requirements suggest that a new system framework is required to support such flexible updates through the view. Unfortunately, none of the research projects or commercial systems meet the above requirements as we will review below.

First of all, for many common update scenarios no side-effect free trans-

lation may exist. This led researchers to permit side-effects [Kel86b, BSKW91, TIHW01], to develop algorithms to detect them [BS81, CP84, DB82, WR04, BKT01, CWW00], or to restrict the kind of updates that can be performed on a view [LL92, CLL02, BDH04, BKKM00, CX00, Rys01]. For many applications where views need to be handled in the same way just as base tables on updates, accepting side-effects or having such stringent restrictions is simply unacceptable. Alternatively, updates without any side effect free translation counterpart are often rejected.

Under the traditional view update semantics [BS81, CP84, DB82], the above two requirements of updating through views, namely, (1) the view update translation being side effect free and (2) a correct translation to always exist, conflict with each other. In other words, if the view updating systems guarantee updates to not cause any view side effect, then most view updates will be considered untranslatable and thus must be rejected. On the other hand, if the view updating system guarantees to translate all updates, then many of these updates will end up causing undesirable view side effects. However, from the user's point of view, the ideal view updating system needs to achieve both of the above goals. Such a win-win solution can be achieved by relaxing the traditional view update semantics, namely, the requirement that the instance of a view is equal to the execution of its view query on the base tables.

Recent work [YK06] proposes a new update semantic for the relational view updating problem. The main idea is that any changes made by the user through her view are encoded using special identifiers in the base data. This encoding assures the uniqueness of data tracing for each view tuple,

namely, no two view tuples ever share the same base data. All view updates are thus translatable, in the sense that they can be encoded, namely, first cloned then updated. It also ensures that side-effects are not visible to users. Beyond the fact that encodings are now being carried along with actual base data, they can be hidden from the view by an extended view query.

However, by allowing every update to be translated, this approach now conflicts with the spirit of data sharing. Updates from each user will always change the public database, leaving other users (using different views) with side effects. The situation could get out of control by interleaving the update effect and the side effects. The whole database might end up being unusable.

Most commercial database systems [BKKM00, CX00, Rys01] provide some support for version management, for example, for extracting (static) snapshots. The user can work on their own local cached data copy. But such snapshots tend to be disconnected from the original database and from future changes by users.

Some GUI-centric software such as CVS repository also provides the capability of backing up the whole database into local storage. Although one could update through views and change the public database by specializing a data merging procedure, it is not practical due to the high cost of backing up the whole database and the manual task of having to attempt to resolve conflicts for the purpose of later merging. Also, it is hard to fulfill the goal of data sharing between scientists, given the large amount to be compared and merged.

5.1.3 LoGo: A Local vs. Global Flexible Write-through Solution

In brief, a solution needs to be developed that supports the following scenarios. (1) *Localize* the view update translation, while preserving the properties of views being side-effect free and updates being always updatable. (2) *On-demand merge* of the local database of the *subject view* into the public database (also called global database), while still guaranteeing the subject view to be free of side effects. (3) *On-demand synchronize* the local data of the *object view* with the public database updated by the *subject view*.

We now propose a solution that successfully tackles all the above requirements within one uniform framework named *LoGo* (*Local-to-Global view update services*). LoGo provides a localized and flexible write-through solution for the view updating problem. The framework of LoGo is shown in Figure 5.1.

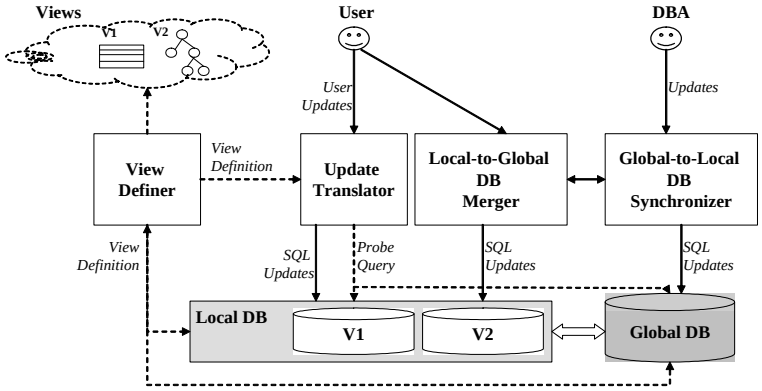


Figure 5.1: The framework of LoGo

The first key issue in the design of the *update localization* solution is how to design the local database schema. This schema needs to be capable of

memorizing update translations in the local database as well as to be sufficient to preserve data used for canceling-out update side effects. It also needs to be compatible with the public database schema so that the local and the public data can be easily merged without any schema changes. The second issue is how to memorize the update translation, including deletion, insertion and modification, in our local database. In addition, LoGo needs to identify any view side effect and propagate the counter-part into the local database to cancel out the view side effect. Finally, the view query needs to be rewritten to smoothly puzzle together the view content from both the local and the public databases. The *update translator* in Figure 5.1 is employed for this purpose. Section 5.2 will describe LoGo update localization solution and its properties.

Updates-writing-through means to merge the localized changes back to the public database. The key issues here include: how to encapsulate various changes from different update types in the local database into one single public database, and how to re-construct the view content after merging. The *local-to-global database merger* is used for writing through. When multiple views are available, this writing through capability will affect other views. Whether or how should other views be synchronized with the updated new public database must be addressed. Updates writing through and synchronization can interleave with each other. Namely, a view can be the subject view (the one merging data into the global database) during one merge procedure and it can also be an object view (the one refreshing according to the global data) during another synchronization procedure. Section 5.3 describes the merging techniques used by the local-to-

public merger in Figure 5.1. Section 5.4 describes the synchronizing techniques used by the public-to-local synchronizer in Figure 5.1.

5.1.4 Contributions

To summarize, we make the following contributions: (1) We propose a new view update semantics which relies on localized behavior to guarantee: (i) all view updates are translatable in a view side effect free manner, (ii) user updates are separated from the global database and (iii) views are independent in terms of update effects. (2) We propose the LoGo framework which fulfills our newly proposed update semantics, yet supports synchronization between local and global behavior when so desired. (3) We implement LoGo system. Experiments are also shown for the performance of LoGo.

5.2 LoGo-basic: Updating Through Views by Localization

Instead of mapping the given view update into updates over the public database, LoGo achieves the update translation using the following steps. First, LoGo maps the given view update into update operations over a *local* database of the given view, which record the pre-selected update translation achieving the given view update if issued over the public database. Second, the view construction query is now changed to query over both the public and local database. As result, LoGo guarantees that (1) all updates are translatable with a side effect free guarantee and (2) all update effects are isolated from the public database. Below we describe how these

goals are achieved.

5.2.1 Running Example

We use the following running example to illustrate the basic idea of LoGo for updating the Select-Project-Join relational views. Figure 5.2 shows the relational database of a campus online registration system. Figure 5.3(a) depicts the relational view definition which extracts the courses and their corresponding teachers into a view (Figure 5.3(b)).

Professor

tid	pid	pname
t_1	p1	David Finkel
t_2	p2	Tim Merrett

Course

tid	cid	cname	credit
t_1	c1	Math	3
t_2	c2	Physics	2
t_3	c3	Math	2

Teaching

tid	pid	cname
t_1	p1	Math
t_2	p1	Physics
t_3	p2	Math

Figure 5.2: A running example: the global database

```
CREATE VIEW V1 AS {
SELECT P.pname AS professor,
       C.cname AS course,
       C.credit AS credit
FROM Professor P, Course C,
     Teaching T
WHERE P.pid = T.pid
      AND T.cname = C.cname}
```

(a)

professor	course	credit
David Finkel	Math	3
David Finkel	Physics	2
David Finkel	Math	2
Tim Merrett	Math	3
Tim Merrett	Math	2

(b)

Figure 5.3: A relational view (b) defined by the view query (a) over the relational database in Figure 5.2

```
u1: DELETE FROM V1 WHERE professor = ''David Finkel''
    AND course = ''Math'' AND credit = 3
```

Figure 5.4: An update $u1$ over the view in Figure 5.3

Now assume the update $u1$ in Figure 5.4 over the relational view, which deletes the three-credit "Math" class taught by professor "David Finkel". The relational tuples used to compute this to-be-deleted view tuple (so

called *generator*) include: Professor.t_1, Course.t_1, Teaching.t_1. There are many possible ways to achieve this update: (1) delete Professor.t_1; (2) delete Course.t_1; (3) delete Teaching.t_1, or (4) any combination of two or three of the above base updates. Basically, to delete any part of the generator will always achieve the given view update. However, it is obvious that none of them is a correct translation in the sense of guaranteeing the view side effect free property. The reason is that our view definition involves an $m : n$ join, namely, *Course.cname = Teaching.cname*. As result, Math can be taught by different professors, while each professor can teach different Math classes with different credits. Deletion of any single teaching relationship from the view, as done by $u1$, cannot be achieved by simply deleting one or more tuples from the public database. This has illustrated in our example. Traditional view updating systems including commercial databases will reject the view update in this case. Below we now describe how LoGo handles this view update and succeeds to make it translatable.

5.2.2 Local Database and Update Translation

Create Local Database. LoGo builds a *local database* to: (i) store the chosen update translation in the public database and (ii) restore or eliminate the view data disappearing or appearing from the view respectively, which otherwise would have caused undesired view side effects. This way, LoGo guarantees that all updates are translatable in a view side effect free manner. This is achieved without modifying the public database.

We build a *local database* D_L for each view allowed to be updated. The

local database includes a $\overline{\Delta}$ table, which memorizes the base table name and the tuples in the table to be deleted to achieve the given view update. It also includes a delta table for each table being referenced by the view definition (e.g., ΔProfessor for Professor table). This delta table is used to restore view elements which might disappear as view side effects. For example, Figure 5.5 depicts the local database schema for the view in Figure 5.3. The *linkID* column in delta tables (e.g., ΔProfessor) is used to connect the $\overline{\Delta}$ table and Δ table, since one transaction of deletion will affect both. The *cloneID* is used to uniquely identity each restore procedure, which eliminates view side effects. The cloneID is automatically generated and unique to each update translation transaction.

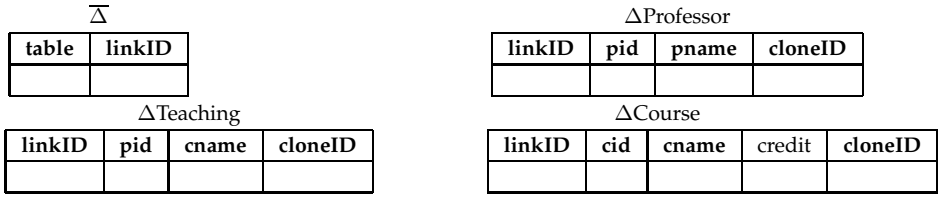


Figure 5.5: The local database state after a deletion ($u1$)

Translate View Deletions. Using the update $u1$ as an example, we now illustrate the schema of the local database and how it is used in the update translation procedure to keep all user updates localized. Assume that the user or the system decides to choose the update translation (1) to achieve $u1$, namely, to delete the tuple t_1 from the *Professor* relation.

LoGo first adds the tuple <Professor, t_1> into $\overline{\Delta}$ table to memorize the delete translation (See the figure below). This achieves the deletion $u1$.

$\overline{\Delta}$

table	linkID
Professor	t_1

However, as side-effect the second and the third view tuples (Physics and Match taught by David Finkel) would also be disappearing from the view. We call them *to-be-disappeared* tuples. To eliminate this side effect, we now extend the local database to re-store these to-be-disappeared view tuples.

Each *restore procedure* is uniquely identified by an assigned cloneID, e.g., cloneID =1 for our example. The restore procedure is achieved by considering join relationships in the view query step by step.

We start from the join between the *Professor* and *Teaching* tables. The tuple professor.t_1 from the table *Professor* in the public database is cloned. It is then inserted into the Δ*Professor* table in the local database as two tuples, both with linkID = t_1 but cloneID = "d" and cloneID = "1" respectively. This starts the re-store procedure corresponds to the deletion of Professor.t_1 memorized in $\overline{\Delta}$ table.

ΔProfessor

linkID	pid	pname	cloneID
t_1	p1	David Finkel	1
t_1	p1	David Finkel	d

All tuples from the *Teaching* table in the public database, which joins with this professor.t_1 by the view query, are cloned. The cloned Teaching tuples (except Teaching.t_1) are added into the Δ*Teaching* table in the local database with cloneID = 1. The clone of Teaching.t_1, however, is inserted

into the ΔTeaching table in the local database with cloneID = "d".

ΔTeaching

linkID	**pid**	**cname**	**cloneID**
-	p1	Physics	1
-	p1	Math	d

Second, the join between *Teaching* and *Course* is processed. The cloned *Teaching* tuple with cloneID = "d" is cloned again, but with cloneID = 1. All *Course* tuples joined with any *Teaching* tuple, which has been cloned in the first step, are cloned. The cloned Course tuples (except Course.t_1) are added into the Δ*Course* table in the local database with cloneID = 1. The clone of Course.t_1 is inserted into ΔCourse with cloneID = "d".

ΔTeaching

linkID	**pid**	**cname**	**cloneID**
-	p1	Physics	1
-	p1	Math	d
-	p1	Math	1

ΔCourse

linkID	**cid**	**cname**	credit	**cloneID**
-	c2	Physics	2	1
-	c3	Math	2	1
-	c3	Math	3	d

In essense, after the restore procedure, the local database contains not only the generator of the to-be-deleted view tuple, but also generators of all to-be-disappeard view tuples. At the last step, the generator of the to-be-deleted view tuple, which has been marked as "cloneID = d", is removed. Figure 5.6 depicts the local database state after the translation.

We slightly modify the view definition Q (Figure 5.3) to produce the new query named *local view query* Q_L (Figure 5.7). Q_L re-stores the tu-

$\overline{\Delta}$

table	linkID
Professor	t_1

ΔProfessor

linkID	pid	pname	cloneID
t_1	p1	David Finkel	1

ΔTeaching

linkID	pid	cname	cloneID
-	p1	Physics	1
-	p1	Math	1

ΔCourse

linkID	cid	cname	credit	cloneID
-	c2	Physics	2	1
-	c3	Math	2	1

Figure 5.6: The local database state after a deletion ($u1$)

ples, that would otherwise have disappeared as the view side effect under the initial view query Q, back into the view (the second and third tuples in Figure 5.3). Q_L is rewritten by adding cloneID join condition for each pair of joined tables in the original view query. In Figure 5.7, two cloneID join conditions are added into the original view query, namely, P.cloneID = T.cloneID and T.cloneID = C.cloneID.

```
SELECT P.pname AS professor, C.cname AS course, C.credit AS credit
FROM ΔProfessor P, ΔCourse C, ΔTeaching T
WHERE P.pid = T.pid AND P.cloneID = T.cloneID
      AND T.cname = C.cname AND T.cloneID = C.cloneID
```

Figure 5.7: The local view query Q_L for the view defined in Figure 5.3

Translate View Insertions. Now let's consider an example of insertions. We always translate an insertion over the view directly into insertions over the local database. This translation will never change the public database. Thus we do not need to write into $\overline{\Delta}$ table in the local database. For example, assuming that originally the local database is empty, the insertion u_2 will result in the new local database state as shown in Figure 5.8. Note that we use I as *linkID* to indicate the newly inserted tuple. This indicator

is used later on for the local-to-global database merger.

```
u2 = INSERT INTO V1 VALUES (Peter Griffen, English, 3)
```

$\overline{\Delta}$

table	linkID

ΔProfessor

linkID	pid	pname	cloneID
I	PI1	Peter Griffen	1

ΔTeaching

linkID	pid	cname	cloneID
I	PI1	English	1

ΔCourse

linkID	cid	cname	credit	cloneID
I	CI1	English	3	1

Figure 5.8: The local database state after an insertion ($u2$)

Translate View Modifications. Modification is treated as a deletion followed by an insertion in LoGo during the translation procedure. For example, u_3 in Figure 5.9 modifies the professor of the 3-credit Math class from "David Finkel" to "Peter Griffen". LoGo will consider u_3 as first deleting the 3-credit Math class taught by professor "David Finkel", then inserting a 3-credit Math class taught by professor "Peter Griffen". Assuming that originally the local database is empty, then the local database state after translating u_3 is shown in Figure 5.9. Note that two different cloneIDs (1 and 2) are chosen for the deletion and the insertion, namely, we assume them as two different restore procedures. For the deletion, the linkID of Professor tuple is t_1. For the insertion, the linkID of the professor tuple is specially coded as M-t_1. The similarity identifies them both as the restore procedures resulting from one view modification. In addition, instead of assigning randomly selected or default values for the columns invisible to the view, we also choose the value from the generator of the view tuple to-be-modified. In our example, the generator of the to-be-modified tuple is: {Professor.t_1, Course.t_1, Teaching.t_1}. We thus choose *p1* as *pid* for the

tuple to be inserted to ΔProfessor table.

```
u3 = UPDATE V1 SET Professor = ''Peter Griffen''
```

$\overline{\Delta}$

table	linkID
Professor	t_1

ΔProfessor

linkID	pid	pname	cloneID
t_1	p1	David Finkel	1
M-t_1	p1	Peter Griffen	2

ΔTeaching

linkID	pid	cname	cloneID
null	p1	Physics	1
null	p1	Math	1
null	p1	Math	2

ΔCourse

linkID	cid	cname	credit	cloneID
null	c2	Physics	2	1
null	c3	Math	2	1
null	c1	Math	3	2

Figure 5.9: The local database state after a modification ($u3$)

5.2.3 LoGo-Basic Algorithm

Algorithm 10 LoGo-Basic for deletion

Input:
D_P: The public database
Q: The view definition over the relational database D_P
u: A user update over the view V defined by Q
D_L: The current local database of V
Output:
U: A sequence of SQL update over D_L

Compute the generator g of the tuple to be deleted
Choose the source tuple $R_i.t_j$ to achieve the deletion
Insert into $\overline{\Delta}$ the new tuple ("R_i", "t_j")
Generate the new cloneID cID for restore procedure
Insert into R_i the cloned tuple ("t_j", t_j, cID)
Let T be the set of current to-be-joined tuples
$T = \{R_i.t_j\}$
for each $R_k \neq R_i$ referenced by V **do**
  Let T_k be the set of tuples in R_k joined with a tuple in T
  Insert T_k into ΔR_k with cloneID = cID and linkID = null
  $T = T_k$
end for
delete the generator of the to-be-deleted tuple from D_L

Algorithm 10 is the LoGo basic algorithm for deletion. Figure 5.10 shows the walk-through steps of applying Algorithm 10 for u_1. First, the generator of the to-be-deleted tuple is computed [WRMar]. The computa-

tion is done by issuing a query combining the view query and the update query, namely, add the where conditions in update query into the where clauses of the view query. Deleting any of the tuples in the computed generator set will achieve the deletion. In the second step, a translation is chosen (e.g., Professor.t_1). This can be done either by communicating with the user [Kel86b], or by a system automatic choice. How to choose the best translation [Kel85, WRMar] is an orthogonal problem and thus not further discussed here. The chosen update is memorized in $\overline{\Delta}$. That is, a new tuple (Professor, t_1) is inserted into the delete differential table $\overline{\Delta}$. Finally, a restore procedure for side effect eliminating purpose changes all Δ tables to recover view tuples that otherwise would be disappearing from the chosen translation. The tuple Professor.t_1 is first cloned into ΔProfessor table with cloneID =1 and linkID = t_1. All tuples from the Teaching table, which join with Professor.t_1, are then cloned into ΔTeaching. Similarly, tuples from the Course table, which join with the cloned Teaching table are also cloned into ΔCourse table. Finally, we delete the chosen generator tuple of the to-be-deleted tuple from the local database D_L.

The algorithm for insertion is very easy: we only need to insert tuples into local database. These tuples will be treated as "new", namely, without any connection with the global database. The algorithm for modification translation is first performs the deletion algorithm and then perform the insertion algorithm.

1. *Compute the generator into G*:
   Select P.ROWID, C.ROWID, T.ROWID
   From Professor P, Course C, Teaching T
   Where P.pid = T.pid and T.cname = C.cname and C.credit = 3
   and P.pname = "David Finkel" and T.cname = "Math";
2. *Choose the translation*:
   Assuming Professor.t_1 is chosen; generated cloneID = 1
3. *Update* $\overline{\Delta}$:
   Insert into $\overline{\Delta}$ values ("Professor", "t_1")
4. *Update* Δ:
   Insert into ΔProfessor Select "t_1", pid, pname, 1 From Generator;
   Insert into ΔTeaching Select null, T.pid, T.cname, 1
   From Generator G, Teaching T Where G.pid = T.pid;
   Insert into ΔCourse Select null, C.cid, C.cname, C.credit, 1
   From ΔTeaching DT, Course C
   Where DT.cloneID = 1 and DT.cname = C.cname;
   Delete from ΔCourse DC
   Where DC.cid not in (Select cid from Generator)

Figure 5.10: Walk-through Algorithm 10 for u_1

5.2.4 View Re-construction

By introducing the local database, the view now needs to be constructed over both the data from the local and the public databases. Given a view V defined by the view definition query Q over the public database D_P = $\{R_1,...,R_n\}$. Let D_L = $\{\overline{\Delta}, \Delta R_1,...,\Delta R_n\}$ denote the local database associated with the view V. We first compute the view tuples generated from the "updated" public database, namely, $Q_P(D_P - \overline{\Delta})$, where Q_P is a slightly rewritten query of Q. For example, the first half of the query in Figure 5.11 (before UNION) is Q_P and used for this purpose. Second, the view tuples generated from the local database D_L need to be computed. This is done by issuing a probe query Q_L similar to the view query over D_L with the following changes: (i) replace each public table R_i in Q with the corresponding local table ΔR_i; (ii) add the cloneID join conditions. In our example, the second portion of the query in Figure 5.11 (after UNION) is used

for this purpose. Finally, by union-ing the view tuples computed from the public and those from the local database, we get the re-constructed view. In other words, the updated view is now computed as: $Q_P(D_P - \overline{\Delta}) \cup Q_L(\Delta R_1,...,\Delta R_n)$.

```
Select P.pname as professor, C.cname as course, C.credit as credit
From Professor P, Course C, Teaching T
Where P.pid = T.pid and T.cname = C.cname
    and P.ROWID not in (Select linkID From Δ̄ Where table="Professor")
    and T.ROWID not in (Select linkID From Δ̄ Where table="Teaching")
    and C.ROWID not in (Select linkID From Δ̄ Where table="Course")
UNION
Select DP.pname as professor, DC.cname as course, DC.credit as credit
From ΔProfessor DP, ΔCourse DC, ΔTeaching DT
Where DP.pid = DT.pid and DT.cname = DC.cname
    and DP.cloneID = DT.cloneID and DT.cloneID = DC.cloneID
```

Figure 5.11: The rewritten view query Q'

5.2.5 Property of LoGo Basic

So far, we have only considered updating the view tuple generated from the public database. However, given that the local database may not be empty once we start updating through the view, a view tuple could be produced either from the public database D_P or the local database of the view D_L. Observation 4 shows that according to the LoGo update translation methodology, it is impossible for a view tuple to have a generator composed of tuples from both the public and the local database (Observation 4).

Observation 4 *Given a view V and a tuple $t \in V$. Let $g(t)$ be the generator of t. Then $g(t) \in D_P$ or $g(t) \in D_L$, but not both.*

Proof.

When the local database is empty (before any update over the view has ever

applied), all view tuples are generaged form the public database. We now prove that the deletion, insertion and modification translations in LoGo will all keep the observation hold.

Delete. Let two tuples t_i, t_j from $g(t)$ satifies the condition $t_i.col1 = t_j.col2$ in the view query. Let $t_i \in D_L$. Let us assume $t_j \in D_P$. According to Algorithm 10, t_j will be cloned into the local database, since it has joined with the tuple t_i. This conflicts with our assumption of $t_j \in D_P$. Therefore after any deletion, Observation 4 holds.

Insert. Since our translation never updates the public database, each insert is translated into a sequence of insertions to the local database. The newly inserted view tuple is generated from the local database, which is essentially connected by the cloneID.

Modify. Given that we treat a modification as a deletion followed by an insertion during translation. Observation 4 holds naturally. □

However, it is critical to distinguish between the pure public database generator and local database generator for the following reasons. First, in typical applications, the local databases tend to be fairly small. Computing the local-generator is rather quick. LoGo should only search the public database – a more costly operation, if and only if a local-generator cannot be found.

Different update translation algorithms must be developed for updating tuples generated from global or local database respectively. The view tuple generated from the public database (with a public-generator) are updated using algorithms in Section 5.2.3. View tuples with a local-generator

are updated using an algorithm similar to Algorithm 10 with only a simple change: instead of memorizing the tuple to be deleted, the chosen translation is directly achieved by actually deleting from the local database.

professor	course	credit
David Finkel	Physics	2
David Finkel	Math	2
Tim Merrett	Math	3
Tim Merrett	Math	2

Figure 5.12: A relational view over the global relational database in Figure 5.2 and local database in Figure 5.6

As an example, consider the current local database as in Figure 5.6. Let us assume the current view as in Figure 5.12, which is the result view of deleting the first view tuple from Figure 5.3(b). Now the view tuples (David Finkel, Physics, 2) and (David Finkel, Math, 2) are generated from the local database, while view tuples (Tim Marrett, Math, 3) and (Tim Marett, Math, 2) are generated from the public database. Now assuming a view update $u4$ deletes the view tuple (David Finkel, Math, 2), which has a local-generator {ΔProfessor(t_1, p1, David Finkel, 1); ΔTeaching(null,p1, Physics,1); ΔCourse(null, c2, Physics, 2,1)}. Assume the chosen delete translation is to delete from ΔProfessor. That is, the tuple (t_1, p1, David Finkel, 1) is deleted from ΔProfessor in the local database. The restore procedure now clones this tuple as (t_1, p1, David Finkel, 2), with cloneID = 2. The rest of the restore procedure is the same as that for the public-generator case. The resulting local database is shown in Figure 5.13. As an optimization step, we could also delete all tuples with cloneID = 1 from the local database but do not contribute to the view tuple generation anymore. The

result local database will not only contain tuples with cloneID = 2.

$\overline{\Delta}$

table	linkID
Professor	t_1

ΔProfessor

linkID	pid	pname	cloneID
t_1	p1	David Finkel	2

ΔTeaching

linkID	pid	cname	cloneID
-	p1	Physics	1
-	p1	Math	1
-	p1	Physics	2
-	p1	Math	2

ΔCourse

linkID	cid	cname	credit	cloneID
-	c2	Physics	2	1
-	c3	Math	2	1
-	c3	Math	2	2

Figure 5.13: The local database state after a deletion ($u4$)

Proposition 6 shows the correctness of Alforithm 10.

Proposition 6 *Given a view V defined by a view query Q over the public database D_P and the local database D_L. Let Q_P be the part of query Q querying against D_P, while Q_L be the part of Q querying over D_L. Given a view deletion u, Algorithm 10 achieves a side effect free translation.*

Proof.

According to Observation 4, the view update u could be deleting a view tuple t with either a public-gnerator or a local generator, but not both. We thus need to prove that for both cases, Algorithm 10 will achieve the side effect free translation. By side effect free, we need to show that when t disappears from the view, no other existing view tuples t' besides t would be deleted from the view, no any new view tuples being inserted into the view either. Namely, we should have $V = Q(D_P, D_L) = Q_P(D_P - \overline{\Delta}) \cup Q_L(D_L)$.

Public-generator. Assuming the tuple $R_i.t_j$ is chosen to be deleted to achieve the view deletion u. According to Algorithm 10, any view tuple, whose

public-generator never uses $R_i.t_j$, will not be affected. Any view tuple with a public-generator using tuple $R_i.t_j$ will be affected. These to-be-affected view tuples cannot be formed from D_P anymore since $R_i.t_j$ is removed by performing $Q_P(D_P$ - $\overline{\Delta})$. Algorithm 10 clones all tuples ever connected with $R_i.t_j$ through joins into the local database D_L during the restore procedure. Thus all generators of these view tuples are now located in D_L.

Among these to-be-affected tuples, (1) t will not appear anymore since its generator is removed from D_L by the last step of the algorithm; (2) Let $t' \in V$ be an existing view tuple (to-be-affected), its generator is thus all cloned into D_L. t' thus will not disappear from the view after the view re-construction. (3) Assuming t'' be a new view tuple appearing in the view after the reconstruction. Then the generator of t'' is located in D_L and satisfies all join conditions. Since that Q_L is a rewritten query of Q by adding new join conditions between cloneID into the where clauses, Q_L is thus "stricter" than Q in filtering and joining. Thus the generator for t'' will certainly satisfy all join condition of Q. This implies that t'' indeed had been already in the view before the update. This contradicts over assumption in the proposition. Thus no new view tuple will appear in the view.

From the above cases, we now conclude that the proposition holds for deleting a view tuple with a public generator.

Local-generator. The only difference between the public-generator and local-generator is that we do not need to memorize the chosen update translation. Instead, we directly delete from the corresponding relation in the local database. The rest of the proof for the public generator above will still hold.

□

5.3 Local-to-Global Database Merging

Update localization (Section 5.2) prevents user views from affecting each other. It allows each user to work on their own local database, whenever they so desire. Once the user is satisfied with changing and preparing their private/customized local data, they might want to publish some or all of their data to share it with the other users. This is achieved by merging the local database explicitly into the global (public) database.

5.3.1 Data Merging Service

We divide the data in the local database into two categories: (i) $\overline{\Delta}$ and tuples in ΔR_i produced by the restore procedure of the deletion processing and (ii) new tuples in ΔR_i produced by the insertion processing. This classification can be easily identified by searching the insertion mark "I" in the *tid* column of each local table, which indicates that the tuple belongs to the insertion instead of the deletion data restore procedure. For example, assume the user wants to merge the local database in Figure 5.8 into the public database. Category (ii) includes all tuples in ΔR_i whose *tid*s are indicated as "I", while the remaining tuples fall into category (i).

Tuples in category (i) will be merged into the public database. First, all tuples indicated by $\overline{\Delta}$ will be removed. We call this *public database purging*. For instance, the first tuple of the Professor relation with "*tid* = t_1" in the public database will be removed during the public database purging

time. Then, all tuples in ΔR_i, which have been linked with the just purged tuple in $\overline{\Delta}$ will be inserted into the public database, namely, *local database purging*. LoGo will automatically produce new *tid*s for each purged tuple, which participate joins in the view reconstruction time (discussed in Section 5.3.2). The tid is generated by combining the cloneID of the tuple and assign a new tuple id in each clone. For deletion, tid = "D" + cloneID + "-" + tupleID. For our example, the local database purging is achieved by inserting tuples from the local database, which are identified by the query below, into the global database. Assuming the public database as in Figure 5.2,

```
SELECT DP.ROWID, DC.ROWID, DT.ROWID
FROM ΔProfessor DP, ΔCourse DC, ΔTeaching DT
WHERE DP.pid = DT.pid AND DP.cloneID = DT.cloneID
      AND DT.cname = DC.cname AND DT.cloneID = DC.cloneID
      AND DP.linkID = t1
```

after merging the local database (Figure 5.6) into the global database, the global database state is shown in Figure 5.15. The sequence of updates used for this merging are shown in Figure 5.14.

```
DELETE FROM Professor P WHERE P.tid = t1
INSERT INTO Professor VALUES (D1-t1, p1, David Finkel)
INSERT INTO Course VALUES (D1-t1, c2, Physics, 2)
INSERT INTO Course VALUES (D1-t2, c3, Math, 2)
INSERT INTO Teaching VALUES (D1-t1, p1, Physics)
INSERT INTO Teaching VALUES (D1-t2, p1, Math)
```

Figure 5.14: SQL updates used to update the global database in the merging procedure

Tuples in category (ii) will be merged into the public database similarly with newly generated *tid*s. For insertion, tid = "I" + $t_{cloneID}$. For example, assuming the public database as in Figure 5.2, after merging the local

Professor

tid	pid	pname
t_2	p2	Tim Merrett
D1-t_1	p1	David Finkel

Course

tid	cid	cname	credit
t_1	c1	Math	3
t_2	c2	Physics	2
t_3	c3	Math	2
D1-t_1	c2	Physics	2
D1-t_2	c3	Math	2

Teaching

tid	pid	cname
t_1	p1	Math
t_2	p1	Physics
t_3	p2	Math
D1-t_1	p1	Physics
D1-t_2	p1	Math

Figure 5.15: The global database after merging the local database in Figure 5.6

database (Figure 5.8) into the global database, the global database state is shown in Figure 5.16.

Professor

tid	pid	pname
t_1	p1	David Finkel
t_2	p2	Tim Merrett
I-t_1	PI1	Peter Griffen

Course

tid	cid	cname	credit
t_1	c1	Math	3
t_2	c2	Physics	2
t_3	c3	Math	2
I-t_1	PI1	English	3

Teaching

tid	pid	cname
t_1	p1	Math
t_2	p1	Physics
t_3	p2	Math
I-t_1	PI1	English

Figure 5.16: The global database after merging the local database in Figure 5.8

5.3.2 View Re-construction

We now describe the changes needed after the local-to-global merging, so that the view definer in Figure 5.1 can extract the virtual view content. Note that the procedure we described below corresponds to the general view reconstruction, namely, not necessarily to be the view just performed the local-to-global merging, which have an empty local database.

Let D_P denote the original tuples in the public database (tid = t_i). Let $D_{\Delta P}$ denote the tuples newly inserted tuples in the public database by a merging procedure, thus $tid \in \{It_i, Dt_i, Mt_i\}$). Let $\overline{\Delta}$ and D_L denote the $\overline{\Delta}$ and ΔR tables in the local database of the view. Then the view content is

computed by: $V = Q_P(D_P - \overline{\Delta}) \cup Q_{\Delta P}(D_{\Delta P}) \cup Q_L(D_L)$. In other words, the view content is computed from three parts. The first part, denoted by Q(D_P - $\overline{\Delta}$), is the original view query over the original public database, which now has some tuples removed according to the local database. The second part, denoted by $Q_{\Delta P}(D_{\Delta P})$, corresponds to the rewritten view query executed over the public delta data (coming from merging). The latter could only be empty if we never performed a view merging procedure before. Here $Q_{\Delta P}$ denotes the rewritten query with each Join condition being associated with an extra *cloneID* Join, which is achieved by the function *getCloneID(tid)*. For example, $Q_{\Delta P}$ query for Q (Figure 5.3) is shown in Figure 5.17. The third part, denoted by $Q_L(D_L)$, corresponds to the rewritten view query executed over the local delta tables. The latter tables would not be empty if we ever performed any update through the view which has not yet been pushed public (merged with the public database). Here Q_P and Q_L are the same as in Section 5.2.4 (e.g., Figure 5.11).

```
SELECT P.pname AS professor, C.cname AS course,
    C.credit AS credit
FROM Professor P, Course C, Teaching T
WHERE P.pid = T.pid AND getCloneID(P.tid) = getCloneID(T.tid)
    AND T.cname = C.cname AND getCloneID(T.tid) = getCloneID(C.tid)
```

Figure 5.17: $Q_{\Delta P}$: view query for $D_{\Delta P}$

5.4 Public-to-Local Database Synchronization

If the view is truly virtual (without any data captured in the associated local database), the updates affecting the base data are naturally reflected by

the view as well. We call this the *default refresh*. LoGo naturally permits this default refresh. However, when the user view updates have been accumulated over time in the local database, we may want to synchronize with the public database at some point. Reasons for this include the following. First, a database administrator can modify the database directly. Such an update would be public, thus possibly affecting all user views. Therefore, the local database needs to be synchronized. Second, since a user (called *subject view user*) can merge her locally collected data residing in the local database into the public database (Section 5.3), this implies that the updates to the public database should then be public, namely, other user views (called *object views*) should be aware of these changes and synchronize with them.

Now data in the public database can be either original or collected from the merging procedure, given that we enable the local-to-global database merging. Now updates over the public database could be requested either indirectly due to data merging or directly by the database administrator. The delta data ($D_{\Delta P}$) generated due to the merging procedure of a subject view has never been shared with other views (objective views). Thus these data are automatically refreshed to these objective views. When an update happens to the data from the original table, since this part of the data is shared by other views, the synchronizer will force to refresh the local databases of the object views to reflect the update.

First, if any deletion operation of the original data matches the tuple deletion memorized in a local $\overline{\Delta}$ table, this deletion operation should be propagated to the local database by eliminating the memorized deletion. The matching tuple in $\overline{\Delta}$ table should thus be deleted. For other tuples

"linked" to it (through linkID in the same cloneID transaction), we could either leave them untouched or eliminate them to save space. The former will leave dangling tuples and thus waste the space over time. The latter, however, is more space efficient. We call it *synchronization purging*.

Let's use an example to illustrate synchronization purging. Assume another view $V2$ identical to the view in Figure 5.3 has the same local database state as in Figure 5.8 at a certain point of time. Now after we have merged the local database associated with $V1$ in Figure 5.3 (subject view) into the global database (Figure 5.15) using a sequence of SQL updates (Figure 5.14), we can synchronize the local database of $V2$ (object view). The first deletion in Figure 5.14 matches the first memorized deletion in the $\overline{\Delta}$ table of $V2$. Thus this tuple can be removed from $\overline{\Delta}$. Meanwhile, all tuples linked with this just removed $\overline{\Delta}$ tuple will also be eliminated from the local database.

On the other hand, if the deletion from an original table does not match any $\overline{\Delta}$ tuple in the local database, no synchronization is forced. The refresh will be automatically achieved whenever we re-construct the view by executing the view query over the public database, that is, by the traditional view mechanism.

Similarly, an insertion to the public database carries the semantics in our context that the inserted tuple has never been visible to other views before. This is guaranteed due to our merging strategy of treating inserted tuples as completely new, each with a new tid. Such insertion thus never needs to be propagated to the other local databases. The view re-construction will also automatically reflect this change. In our example, after the global-to-local synchronization, the local database of $V2$ is now empty.

professor	course	credit
David Finkel	Physics	2
David Finkel	Math	2
Tim Merrett	Math	3
Tim Merrett	Math	2

Figure 5.18: The view $V2$ after synchronization

By providing services for merging and synchronizing, we now encounter a new view update translation scenario as below. A synchronized view, such as $V2$, can again be updated. Sometimes this update will be translated into updating the delta data in the public database. Certainly we could use the LoGo basic approach by simply treating $D_{\Delta P}$ as the original public database. The only difference here is that we query over $Q_{\Delta P}$ instead of Q_P to identify tuples which have been linked together through Joins, and then propagate them into the local database. For example, assuming that we have an update specified over the refreshed view $V2$ in Figure 5.18: `delete from` $V2$ `where professor=‘‘David Finkel’’ and course=‘‘Math’’ and credit=2`. First, we search D_P and do not find the generator. We then search $D_{\Delta P}$ and identify the generator (Professor.D1-t_1, Teaching.D1-t_1, Course.D1-t_1). Assume we choose to delete Professor.D1-t_1 to achieve the view update. Then the local database after the update translation is shown in Figure 5.19.

$\overline{\Delta}$

table	linkID
Professor	D1-t_1

ΔProfessor

linkID	pid	pname	cloneID
D1-t_1	p1	David Finkel	4

ΔTeaching

linkID	pid	cname	cloneID
-	p1	Physics	4

ΔCourse

linkID	cid	cname	credit	cloneID
-	c2	Physics	2	4

Figure 5.19: The local database state of $V2$ after refresh

5.5 LoGo-XML: Updating XML Views Through Localization

As described in Section 2.6, an XML view over a relational database can be considered as a combination of a set of relational views [WRMar]. Thus we map the XML view updating problem into a set of relational view updating problems. This mapping provides us the opportunity of extending the LoGo basic approach to also cover the XML scenario as we will describe in this section.

5.5.1 Running Example

Let us consider the following relational database (Figure 5.20). For illustration purpose, we add two tables, Student and Enrollment, into the global relational database in Figure 5.2.

We use an XML view (Figure 5.21) defined over this extended global database as our example. The view annotated schema graph of this XML view is shown in Figure 5.22(a). Figure 5.22(b) shows the SQL query corresponding to each schema node.

Professor

tid	pid	pname
t_1	p1	David Finkel
t_2	p2	Tim Merrett

Course

tid	cid	cname	credit
t_1	c1	Math	3
t_2	c2	Physics	2
t_3	c3	Math	2

Teaching

tid	pid	cname
t_1	p1	Math
t_2	p1	Physics
t_3	p2	Math

Students

tid	sid	sname
t_1	s1	Chun Zhang
t_2	s2	Mike Fisher
t_3	s3	Feng Lee

Enrollment

tid	sid	cid
t_1	s1	c1
t_1	s1	c2
t_2	s2	c2
t_3	s3	c3

Figure 5.20: Additional tables added into the global database in Figure 5.2

5.5.2 Update Translation and Data Sharing

Given that we map the XML view updating problem into the relational view updating problem, the proposed LoGo-basic approach can now be used to support updates on elements of each schema node. This naturally leads us to the very first naive extension for solving the XML view updating problem, namely, we treat each schema node separately as a relational view and apply the LoGo basic approach to it. Since LoGo-basic will guarantee the property of update localization, namely, no other relational views (map to other schema nodes) will be affected, then the view side effect free property is naturally guaranteed.

However, this simple solution requires a local database for each XML schema node. This would lead to a major data explosion problem. In fact, it is possible to share data between XML schema nodes. The view hierarchy essentially implies the *data sharing*. The question we now must address is how best to design the update translation since each node (mapped to a relational view) is now sharing certain portions of the data with other

```
FOR $c IN DOCUMENT(Course/ROW),
    $t IN DOCUMENT(Teaching/ROW)
    $p IN DOCUMENT(Professor/ROW)
WHERE $c/cname = $t/cname AND $p/pid = $t/pid RETURN
    <ClassInfo>
        <Course>$c/cname/text(), $c/credit/text()</Course>,
        <Professor>$p/pname/text()</Professor>,
            FOR $s IN DOCUMENT(Student/ROW)
                $e IN DOCUMENT(Enrollment/ROW)
            WHERE $s/sid = $e.sid AND $e/cid = $c/cid
            RETURN
                <Student>$s/sname/text()</Student>
    </ClassInfo>
```

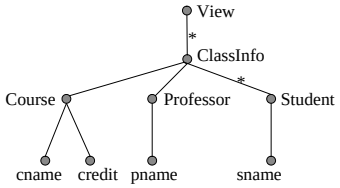
(a) View query

```
CI1     <ClassInfo>
CI1.C1      <Course>Math, 3</Course>
CI1.P1      <Professor>David Finkel</Professor>
CI1.S1      <Student>Chun Zhang</Student>
        </ClassInfo>
CI2     <ClassInfo>
CI2.C1      <Course>Physics, 2</Course>
CI2.P1      <Professor>David Finkel</Professor>
CI2.S1      <Student>Mike Fisher</Student>
CI2.S2      <Student>Feng Lee</Student>
        </ClassInfo>
CI3     <ClassInfo>
CI3.C1      <Course>Math, 2</Course>
CI3.P1      <Professor>David Finkel</Professor>
CI3.S1      <Student>Feng Lee</Student>
        </ClassInfo>
CI4     <ClassInfo>
CI4.C1      <Course>Math, 3</Course>
CI4.P1      <Professor>Tim Merrett</Professor>
CI4.S1      <Student>Chun Zhang</Student>
        </ClassInfo>
CI5     <ClassInfo>
CI5.C1      <Course>Math, 2</Course>
CI5.P1      <Professor>Tim Merrett</Professor>
CI5.S1      <Student>Feng Lee</Student>
        </ClassInfo>
```

(b) XML view

Figure 5.21: The XML view (b) defined by the view query (a)

nodes (other relational views).



`CI`: Select *
From Professor P, Course C, Teaching T
Where P.pid=T.pid and T.cname = C.cname
`C` : Same as CI node
`P` : Same as CI node
`S` : Select *
From Professor P, Course C, Teaching T, Student S, Enrollment E
Where P.pid=T.pid and T.cname = C.cname
and E.cid = C.cid and S.sid = E.sid

Figure 5.22: The schema graph for the view in Figure 5.21

Deletion. Let v be the schema node to be updated by a view update u. Let $G(v)$ be the schema level generator of v (relations referenced by the mapping of the relational view query of v, defined in Section 3.2.3). Let v_p be the parent schema node of v. Then the local database of v consists of the following: (i) $\bar{\Delta}$ and (ii) ΔR_i where $R_i \in G(v) - G(v_p)$. This implies that v shares ΔR_i ($R_i \in G(v_p)$) with its parent v_p, although ΔR_i is not necessarily in the local database of v_p (it might be in v_p's ancester nodes' local databases).

Assume a view update $u6$, expressed as:
`(delete; view/ClassInfo[Course=‘‘Math, 3’’ and Professor=‘‘David Finkel’’])`. Update $u6$ deletes `CI1` from the view. Let us assume that we choose to delete from the Professor relation. The local database of the `CI`-node is the same as in Figure 5.6. The local database state of `S`-node remains unaffected by this update translation.

Now consider another view update $u7$ to delete `CI4.S1`. The local database of `S`-node is shown in Figure 5.23, if we choose to delete from the *Student* relation. Here the translation will only write the local database of `S`-node, while leaving the local database of `CI`-node unaffected.

$\bar{\Delta}$

table	linkID
Student	t_1

ΔStudent

linkID	sid	sname	cloneID
t_1	s1	Chung Zhang	1

ΔEnrollment

linkID	sid	cid	cloneID
null	s1	c1	1

Figure 5.23: The local database state of S-node after $u2$

$\bar{\Delta}$

table	linkID
Student	t_1

ΔEnrollment

linkiD	sid	cid	cloneID
null	s1	c1	1
null	SI1	c1	2

ΔStudent

linkID	sid	sname	cloneID
t_1	s1	Chung Zhang	1
t_1	SI1	Peter Griffen	2

Figure 5.24: The local database state of S-node after $u3$

Note that here we always choose to update the non-shared data. It is also possible to update the shared data. However, by doing that, we break the sharing property between the parent and child node. The shared data now needs to be cloned in both local databases, which essentially is not shared anymore.

Insertion. Insertion is also handled similarly to the LoGo-Basic approach. But again, we insert only into the local database without touching the shared data. For example, consider the insertion of a new student element into `CI4`. This update is expressed as:

$u8$ = `(Insert; View/ClassInfo[Course=‘‘Math, 3’’ and Professor=‘‘David Finkel’’]; <Student>Peter Griffen</Student>)`. The updated local database of `S`-node is shown in Figure 5.24.

The data sharing we proposed here is maximal data sharing. The reason is that if there is another relation R which can be shared between a parent and a child node, then R $\in$G(v)$\cap G(v_p)$, which has already been included by our sharing policy.

5.6 Evaluation

We have conducted experiments to address the performance impact of LoGo. The test system used is a dual Intel(R) Pentium-III 1GHz processor, 1G memory, running SuSe Linux and Oracle 10g. The relational database is built using TPC-H benchmark [TPC].

5.6.1 Performance of Relational View Updating

```
CREATE VIEW EV1{
SELECT *
FROM region R, nation N, customer C, orders O, lineitem L
WHERE N.N_REGIONKEY = R.R_REGIONKEY AND C.C_NATIONKEY = N.N_NATIONKEY
      AND O.O_CUSTKEY = C.C_CUSTKEY AND L.L_ORDERKEY = O.O_ORDERKEY};
```

Figure 5.25: Relational view for evaluation

Figure 5.25 shows the relational view used in our experimental study. A deletion over the view EV1 is to delete a single tuple. An insertion is to insert a new tuple into the view.

Update Performance. Deleting a view tuple can be achieved by deleting any tuple in the generator. For example, deleting from the *Region* relation can achieve the deletion, while deleting from Lineitem can also achieve a

view tuple deletion. Similarly, deleting from the *Customer*, *Orders* or *Nation* tables can all achieve the view tuple deletion. Figure 5.26 shows the performance difference among these candidate translations. Deleting from the *Lineitem* table is the cheapest one since it will not cause any view side effect, thus no any data propagation to the local database is needed. We refer to this translation as *LoGo-best* in the rest of the section. Deleting from the *Region* is very expensive (referred as *LoGo-worst* later on), since it will cause side effects by making all view tuples from that region disappear. To eliminate the side effect, generators of all these to-be-disappeared view tuples are propagated and preserved in the local database to restore these view tuples. Figure 5.27 also shows that the local database space required for deleting from the Region relation is much larger than other translations.

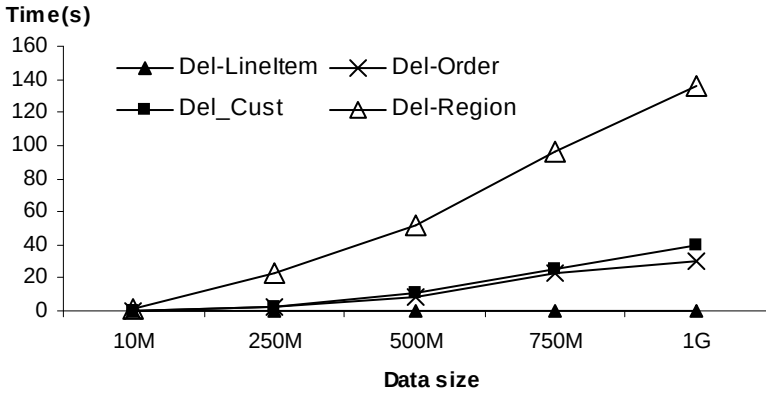


Figure 5.26: Performance among different delete translations

Figure 5.28 shows the performance of a single deletion and insertion. Insertion takes very little time, given that it only visits the local database. The latter usually is very small. Even the best translation of deletion (LoGo-best) is much more expensive compared with insertion.

DataSize	Orig-DB	Del-Lineitem	Del-Order	Del-Cust	Del-Region
10M	12.4M	0.383M	0.383M	0.383M	0.383
250M	284.1M	0.383M	0.383M	0.383M	24.4
500M	574.1M	0.383M	0.383M	0.383M	47
750M	855.1M	0.383M	0.383M	0.383M	71.1
1G	1100M	0.383M	0.383M	0.383M	100.2

Figure 5.27: Space increasing caused by update localization

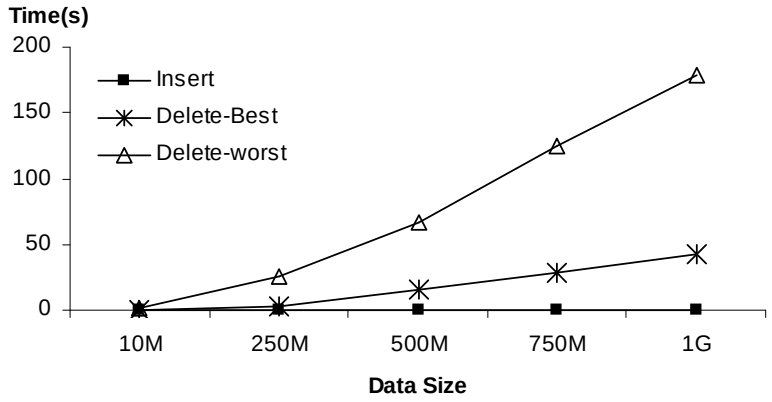


Figure 5.28: The performance of a single view update

Figure 5.29 shows an analysis of each update translation. For deletion, the system always computes the generator of the to-be-deleted tuple, in order to even identify any candidate translation. Here we assume that the cost of making choice among these translations is negligible. This is a major portion of time spending for translating a view deletion. Actually, the real update cost is actually very cheap for delete-best, which is comparable to the insertion time.

Figure 5.30 shows that the performance gains when the public database includes indices.

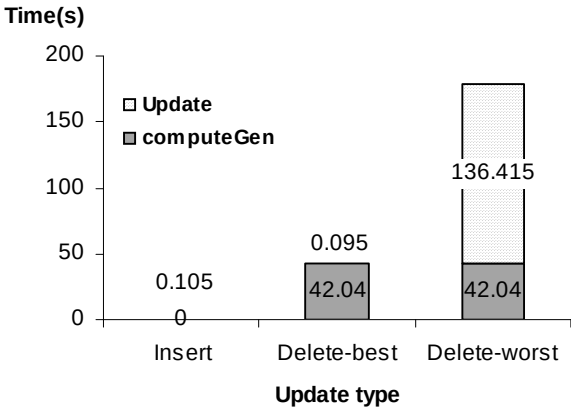


Figure 5.29: Time spreading for each update (public DBsize = 1G)

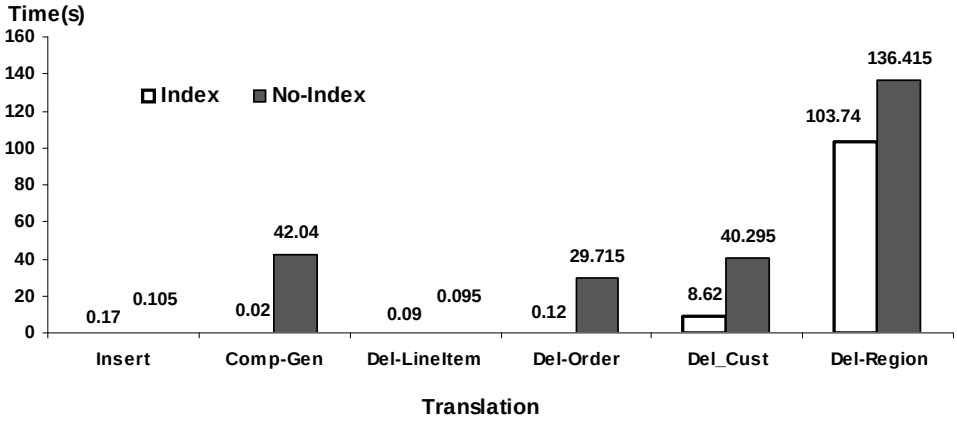


Figure 5.30: Performance of index during update translation (public DBsize = 1G)

Performance of view construction. Figure 5.31 shows the cost of computing the view content. The initial view computation time refers to the cost of extracting the view content, when no local database has ever been used. LoGo view computation time includes commutating the view content from both the local database and the public database. The best case of LoGo al-

lows view updating by only memorizing the translation in $\overline{\Delta}$ without propagating any extra data into Δ tables to cancel-out side effects. The performance of LoGo-best is even comparable to the original view construction time. However, the worst case will increase the local database size, thus it takes more time to re-compute the view content. Figure 5.32 also shows an analysis of the time spent on querying the public and the local databases respectively.

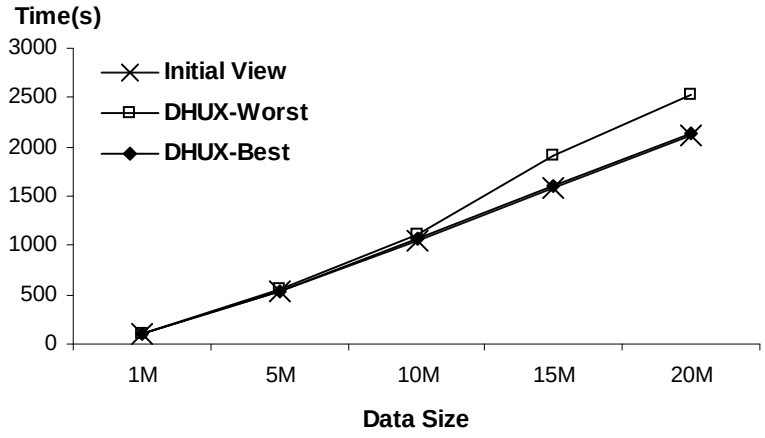


Figure 5.31: Performance of view construction

Performance of merging and synchronization. Figure 5.33 shows the performance of the merge and synchronize services. Here we assume that the system always select the best translation for deletion, namely, it always delete from the Lineitem table. This assumption implies that all Δ tables include only tuples from insertion translation, while deletion translation only writes into the $\overline{\Delta}$ table. We compare the performance of merging a local database constructed due to deletions only and another local database

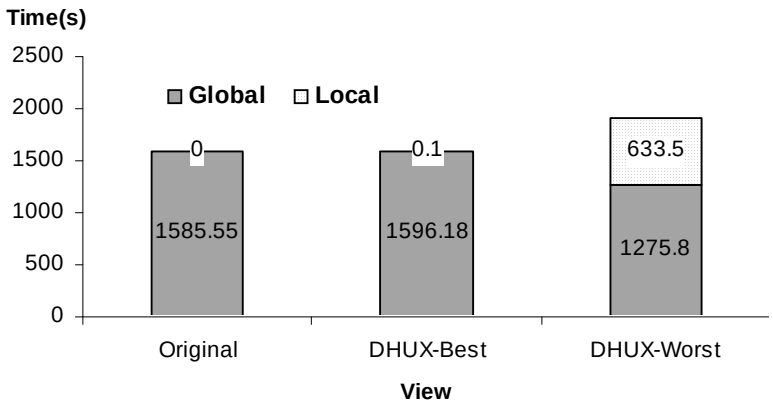


Figure 5.32: Time spreading for view construction (public DBsize = 15M)

from the same number of insertions only. The deletion merging is always cheaper than the insertion merging, since that the insertion merging involves adding new tuples into the public database in addition to the local operations.

Synchronizing is very efficient as shown in Figure 5.33. The reason is that (i) the local database is usually small if we always choose the best translation for deletion translation and (ii) newly inserted data is not forced to be synchronized rather it would be automatically refreshed when executing the rewritten view query by a further user request.

5.6.2 Performance of XML view updating

Figure 5.35 depicts the structure (schema) of an XML view constructed by nesting through the foreign key (Figure 5.34).

There are two extreme sharing policies for XML view update translation, namely, no-sharing or maximally sharing. Without any sharing, the

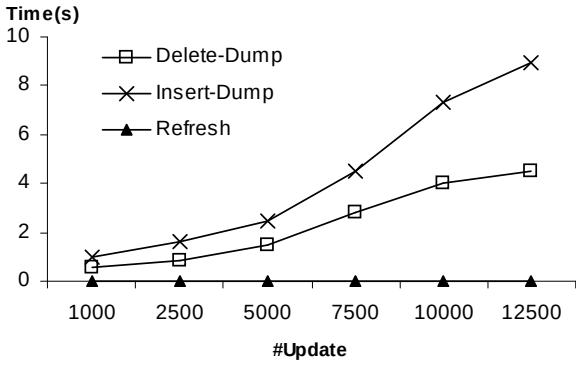


Figure 5.33: Performance of merging and synchronization

```
<View>
FOR $r IN DOCUMENT(Region/ROW)
RETURN
 <Region>
   $r.R_regionkey,
   FOR $n IN DOCUMENT(Nation/ROW)
   WHERE $n/regionkey = $r.R_regionkey
   RETURN
     <Nation>
        $n.N_nationkey,
        FOR $c IN DOCUMENT(Customer/ROW)
        WHERE $c/nationkey = $n.nationkey
        RETURN
          <Customer>
             $c.C_customerkey,
             FOR $o IN DOCUMENT(Order/ROW)
             WHERE $c.C_customerkey = $o.customerkey
             RETURN
               <Order>
                 $o.O_orderkey,
                 FOR $li IN DOCUMENT(Order/ROW)
                 WHERE $o.O_orderkey = $li.orderkey
                 RETURN
                   <Lineitem> $li/Li_lineitemkey </Lineitem>
               </Order>
          </Customer>
       </Nation>
 </Region>
</View>
```

Figure 5.34: The XML view for LoGo-XML performance study

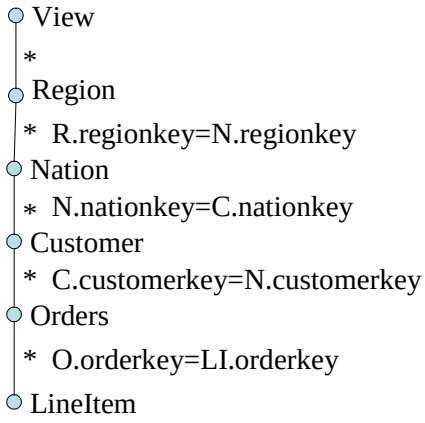


Figure 5.35: The schema graph for LoGo-XML view in Figure 5.34

data explosion in the local database is significant as shown in Figure 5.36. This is because side effects appearing on any node will also cause side effects on its children (if any). Side effects need to be eliminated by propagating to the local database of a parent node as well as to its children nodes. This causes data republication. Maximal data sharing will reduce the size of the local database significantly, as shown in Figure 5.36.

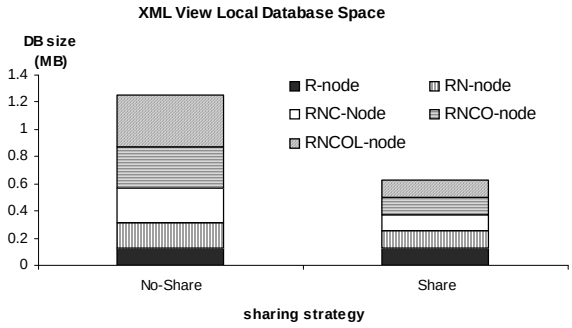


Figure 5.36: Space performance of updating XML views (public DBsize = 1G)

Chapter 6

O-HUX: XML View Updating Handling with Order

6.1 Introduction

When the view is in XML, the problem of updating through the view becomes more complex. In particular, compared to the non-ordered flat relational data model, XML is an ordered hierarchical data model. Correspondingly, XQuery can be order sensitive. However, despite the fact that order is an essential aspect of XML, it is not well addressed by the database community. That is, all previous XML view updating work [BDH04, CLL02, WRMar] assumes a non-ordered semantic.

In this chapter, we focus on the order specific issues of updating XML views defined over a relational database, namely, handling cases when both the XML view as well as the XML update query are order sensitive. We now

consider two XML views to be equal if and only if they include not only the same view content, but also the same order among the elements.

We classify the order syntax in the XML view into different categories, according to their potential effects on the view updating problem. We design a set of rules for each category, which are used for identifying potential update translations.

Our contributions include: (1) We are the first to study updating order-sensitive XML views, to the best of our knowledge. (2) We extend clean source theory to order sensitive semantics. (3) Based on the order-sensitive clean source theory, we develop the O-HUX algorithm that guarantees view side effect free semantics while considering most of the XQuery order constructs. (4) Our O-HUX algorithm relies largely on SQL, and hence can be easily adopted by relational database systems to support order-sensitive view updating.

6.2 Background

6.2.1 Order in XML

Order comes in various forms. Generally speaking, an ordered data model and the order-based functionality present in standard query language are key ingredients that contribute to order.

First of all, XML is an ordered data model. An XML document represents a tree structure; a pre-order traversal of this tree representation indicates the *document order*. However, if we consider the default mapping of the relational database as a canonical XML view, it does not have implicit

document order. The reason is that the relational data model is not ordered.

A view definition using a standard query language, such as XQuery, can overwrite the implicit document order (if any) using explicit *query order*. Such explicit query order can be specified using the following constructs.

- *Order-based axes in XPath.* XPath includes following-sibling and following axes as well as their backward counterparts. These axes are called the order-based axes. Order-based axes in XPath only "expose" the existing order, but never overwrite it.

- *Order By clauses.* Using *Order By* in XQuery will produce new orders by overwriting the existing order.

- *Position and Range function.* Position and range functions can be used to select certain parts of the elements from the previous query result. For example, an XPath query to select the third book is: `/book[3]`; XPath to find all books after the fourth one `/book [position() > 4]`.

- *Relative order functions*. XQuery standard defines two order functions in XPath, namely, first() and last(). We call the ordered XPath using these functions as *relative order functions*. For example, an XPath query to select the second last book is: `/book[last()-1]`.

- *Order expression.* Some expressions using position and range functions will also generate ordered result. For example, `/book[position() mod 2 = 0]` exposes all elements at all even numbered positions.

XML query result reflects both the implicit document order (if any) and the explicit query order in an interleaved manner. Also, when the document is not ordered, such as the relational database considered in this paper, the XML query result might only be *partially ordered*. Namely, only part of the view result maybe ordered by the explicit order specified in the query, while others remain with non-ordered semantics.

6.2.2 Running Example

Professor

	pid	pname
t_1	p1	David Finkel
t_2	p2	Tim Merrett

Key:{pid}

Course

	cid	cname	pid
t_1	c1	Math	p1
t_2	c2	Physics	p1
t_3	c3	English	p2
t_4	c4	Biology	p1
t_5	c5	Chemistry	p1
t_6	c6	Spanish	p2

Key={cid}

Figure 6.1: An example relational database

Let us consider the following relational database (Figure 6.1) as a running example to illustrate the order sensitive XML view updating problem. An XML view (Figure 6.2) can be defined using order constructs introduced above, as shown in Figure 6.3. In our view updating scenario, as the view is defined over the relational database, the order-based axes would never be used. We will refer to any expression that uses position, range, or relative order functions as **order filter**. Thus, the order constructs we addressed include OrderBy and order filters.

We use the update syntax from [TIHW01] to specify an update opera-

```
<ClassInfo>
    <Professor>David Finkel</Professor>
    <CourseSecond>Physics</CourseSecond>
    <CourseMore>Math</CourseMore>
    <CourseMore>Physics</CourseMore>
    <CourseMore>Biology</CourseMore>
</ClassInfo>
<ClassInfo>
    <Professor>Tim Merrett</Professor>
    <CourseSecond>Spanish</CourseSecond>
    <CourseMore>English</CourseMore>
    <CourseMore>Spanish</CourseMore>
</ClassInfo>
```

Figure 6.2: XML view *ClassView*

```
FOR $p IN DOCUMENT(Professor/ROW)
ORDER BY $p.pid
RETURN
  <ClassInfo>
    <Professor> $p/pname/text() </Professor>,
    FOR $c IN DOCUMENT(Course/ROW)
    WHERE $p.pid = $c.pid
    ORDER BY $c.cid
    RETURN
      FOR $c1 IN $c[2]
      RETURN
        <CourseSecond> $c1/cname/text() </CourseSecond>,
      FOR $c2 IN $c[1 to 3]
      RETURN
        <CourseMore> $c2/cname/text()</CourseMore>
  </ClassInfo>
```

Figure 6.3: View query for *ClassView*

tion (insert, delete or replacement) over the view. An update may or may not specify order syntax. An example update query (that does not specify order) is shown below in Figure 6.4.

```
FOR $ci IN DOCUMENT(View.xml)/ClassInfo,
    $cm = $ci/ClassMore
Where $cm /text() = "Physics"
Update $ci{
    Delete $cm}
```

Figure 6.4: An update over XML view in Fig. 4.1

6.3 Order-sensitive Clean Source Theory

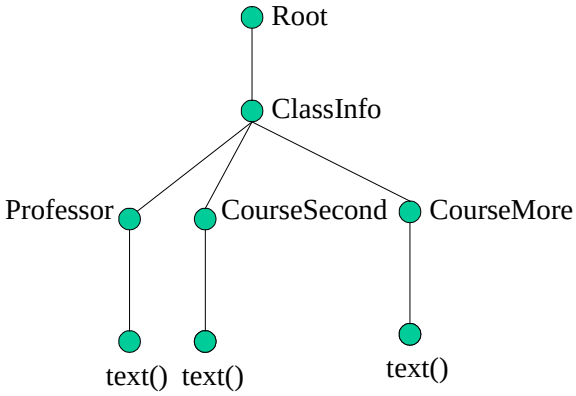
The *clean source theory*, proposed in [DB82], has been widely used to solve the relational view update problem [CWW00, BKT01]. It is further extended in [WR04] (Chapter 3) into the *clean extended source theory* to serve as a theoretical foundation for solving the XML view update problem. It determines whether a given translation mapping is correct for the XML view update problem. In order to handle order semantics, we now first extend the key concepts shown in Figure 6.5. We then form our *order-sensitive clean source theory*.

e	A view element
$g(e)$	The generator of e
s	The source of e
$extend(s)$	The extended source of s

Figure 6.5: Key concepts of the clean extended source theory

As introduced in Section 2.6, we consider an XML view as the combination of a set of relational views. In other words, elements in each XML schema node are considered to be generated by issuing an SQL query to the relational database. This in turn defines the *mapping relational view* of the given schema node. For example, the **annotated schema graph (ASG)** for the view in Fig. 6.2 is depicted in Fig. 6.6. We use the abbreviation $\texttt{CI}_i$

to represent the *i*-th `ClassInfo` element.




```
CI: SELECT * FROM Professor
P:  Same as CI
CS: SELECT A2.cname FROM Q2 AS A2 WHERE A2.pos = 2,
CM: SELECT A2.cname FROM Q2 AS A2
    WHERE A2.pos >= 1 and A2.pos <= 3
For CS and CM, Q2 is defined as following:
       Q2 = SELECT A1.cname, row_number() OVER
            (partition by A1.pid order by A1.cid) pos
            FROM Q1 AS A1
       Q1 = SELECT * FROM Professor P, Course C
            WHERE P.pid = C.pid
```

Figure 6.6: XML view schema graph and its SQL mapping

Let $R_1, R_2, ..., R_n$ be the set of relations referenced by the SQL query Q of a given view ASG node v. Informally a view element e's **generator** $g(e)$ is $\{R_1^*, R_2^*, ..., R_n^*\}$, where $R_i^* \subseteq R_i$ $(i = 1..n)$ contains exactly those tuples in R_i that are used to derive e. The formal definition can be bound in Section 3.2. For example, the generator of the `ClassInfo` element `CI1` in Fig. 6.2(b) is $g(\texttt{CI1}) = \{\text{Professor}.t_1\}$.

We now propose the following extension to the generator definition. If Q specifies order constructs (the order construct can be OrderBy or order filter), then we first remove all the order constructs and compute e's gen-

erator as mentioned above. We refer to each R_i^* as a **non-ordered trace** of e. Further, if Q specifies an order construct over R_i, then we define an **ordered trace** $\overset{o}{R}_i^*$ to include all tuples in R_i that participate in the order computation. For example, the non-ordered traces of the `CourseSecond` element `CI1.CS1` is R_1^* = {Professor.t_1}, R_2^* = {Course.t_2}. The only ordered trace is $\overset{o}{R}_2^*$ = {Course.t_1, Course.t_2, Course.t_4, Course.t_5}. That includes all tuples from the Course relation which participate in the order computation (Order By $c.cid).

We now define the concept of **source**, which determines the existence of e in the view. Each non-ordered trace R_i^* is also a **non-ordered source** of the view element. Computing an **ordered source** from an ordered trace $\overset{o}{R}_i^*$ is more complex, and will be discussed in Section 6.4.3. Intuitively, an ordered source satisfies the property that updating this source will not cause any side-effects on the relational mapping views due to the position change of the view elements.

For example, consider the view element `CI1.CS1` discussed above. Its relational mapping view is shown in Figure 6.6. The two non-ordered sources are: s_1={Professor.t_1} and s_2={Course.t_2}. This element also has an ordered trace $\overset{o}{R}_2^*$ = $\overset{o}{Course}^*$ = {Course.t_1, Course.t_2, Course.t_4, Course.t_5}. An ordered source for this element (as will be discussed in Section 6.4.3) is $\overset{o}{s}_2$ = {Course.t_2, Course.t_4, Course.t_5}. Note that if we do not delete Course.t_4 and Course.t_5 while deleting `CI1.CS1`, then it will cause view side-effects: another Course will appear as the CourseSecond element as side effects in the XML view, as well as in its mapping relational view.

Clean source theory [DB82, WR04] (Section 3.2) says that an update

translation is correct if it deletes or inserts a **clean source** of the view element. We now extend the concept of clean source as: a *clean source* is an (ordered or non-ordered) source of an element used only by this particular element and no other one. Intuitively, this means that the update operation only affects the "private space" of a given view element and thus will not cause any view side effect. The clean source theory becomes order-sensitive by utilizing the order specific concepts as mentioned above.

Determining whether a source is a clean source is quite straight forward. For instance, one could compare the view before and after the update, that is, deleting or inserting the source, as done in [Rys01]. Optimizations for this step have been studied in [BKT01, CWW00], as well as our previous work [WRMar] (Chapter 4). We will not focus on this issue. Instead, we will only focus on how to find the ordered sources from ordered traces (Section 6.4). We identify ordered sources by only considering whether deleting or inserting it will cause side effects on the mapping relational view of the schema node. To identify whether an ordered source is clean, side effect checking on the ancestor and descendent schema nodes are required, which essentially are the same as we described in Chapter 4 or in the literature [BKT01, CWW00].

6.4 O-HUX: XML View Update Handling With Order

Based on the order-sensitive clean source theory stated in Section 6.3, we now introduce the general framework named **O-HUX** for XML view update handling with order (reads right to left).

6.4.1 O-HUX Algorithm

The detailed algorithm of **O-HUX** is shown in Algorithm 11. O-HUX takes the order-sensitive view definition query Q and the update over the view u as input. It then generates the first correct SQL translation it identifies.

Algorithm 11 O-HUX: Order-sensitive XML view updating

Input:
V: XML view defined by XQuery DEF_V
u: The update over the XML view to delete/insert an element e
Output: U: The sequence of SQL updates

/* Step 1: Update transformation */
Compute the ASG $\mathcal{G}_V$ of V
Identify the update type $type$
Identify the schema node v in $\mathcal{G}_V$ to be updated by u
/* convert non-ordered into ordered update (Section 6.4.2)*/
u' = ConvertNonOrderToOrderUpdate(u)

/* Step 2: Computing the sources */
Compute the non-ordered and ordered traces of e
Compute the set of non-ordered sources NOS of e
Compute the set of ordered sources OS of e using rules from Section 6.4.3

/* Step 3: Identifying clean sources */
/* Identify the first clean source from non-ordered sources */
for all Source $s_i \in NOS$ **do**
  boolean sideEffectFree = CheckSideEffect(s_i, $\mathcal{G}_V$)
  if sideEffectFree **then**
    U = GenerateSQLUpdates(s_i, $type$)
    RETURN U
  end if
end for
/* Identify the first clean source from ordered sources */
for all Source $s_i \in OS$ **do**
  boolean sideEffectFree = CheckSideEffect(s_i, $\mathcal{G}_V$)
  if sideEffectFree **then**
    U = GenerateSQLUpdates(s_i, $type$)
    RETURN U
  end if
end for

There are three steps in O-HUX. The first step transforms a non-ordered view update u into an ordered update u'. This step is needed because the update statement u might not specify the position of the element to be up-

dated. For instance, update in Figure 6.4 does not specify the position of the `CourseMore` element to be updated, while the XML view definition (Figure 6.3) extract view content based on the order information. This position is needed to compute the ordered traces in the next step, also as described in Section 6.4.2.

In the second step, O-HUX computes the non-ordered and ordered traces of the to be inserted or deleted view element e. Further, it identifies non-ordered and ordered sources of e. Section 6.4.3 shows how to compute the ordered sources.

In the third step, O-HUX identifies the clean sources (if any). Once a clean source is identified, the corresponding SQL update will be generated; otherwise, the view update is rejected. Our algorithm favors non-ordered sources over ordered sources, as non-ordered sources tend to touch less data in order to achieve the given view update.

6.4.2 From Non-ordered to Ordered View Updates

As mentioned in Section 6.4.1, we need to transform the non-ordered view update into ordered update. For instance, consider the update in Figure 6.4, we will first issue a probe query to identify the position of the `CourseMore` element to be deleted. This probe query can be obtained by combining the SQL query of `CM` node with the update statement. This is given by: `SELECT A2.pos FROM Q2 AS A2 WHERE A2.cname = ''Physics''`, where `Q2` is defined by the SQL mapping of `CM` node in Figure 6.6. The query result shows that "Physics" is the first course before order-filter $c[1 to 3] is applied. Therefore we get the ordered update statement as in Figure 6.7.

```
FOR $ci IN DOCUMENT(View.xml)/ClassInfo,
    $cm/CourseMore[1]
UPDATE $ci{
    DELETE $cm }
```

Figure 6.7: Converted ordered update for update in Figure 6.4

6.4.3 Identification of the Ordered Sources

Consider a view element e, whose corresponding SQL query Q references relations $R_1, R_2, ..., R_n$. Q can specify different order constructs on each R_i (Section 6.2.1). Computing the ordered sources from the ordered trace R_i^* depends on the order constructs specified on R_i, as discussed below.

(1) Ordered traces with only OrderBy clauses. This is the case when only OrderBy clauses are specified on R_i in SQL query Q, namely, there are no order filters. In our example view (Figure 6.2), if we omit the order filter `[1 to 3]` specified on the `CourseMore` element, the ordered trace from the Course relation, denoted by $Course^*$, is defined only using OrderBy clauses (`OrderBy cid`).

Definition 16 *Given an XML view element e computed by the SQL query Q. Assume Q specifies only OrderBy on R_i. The non-ordered source s_i is also the ordered source $\overset{o}{s_i}$.*

As described in Section 6.3, an ordered source needs to satisfy the property that updating this source will not cause any side-effects due to the position change of the view elements. Apparently, change the order among view element using Order By clauses will not change the update behavior. Namely, no side effects will appear just because the position change.

For example, consider the `CourseMore` node without the order filter [1 to 3] as mentioned earlier. An element `CI1.CM1` has an ordered trace $\overset{o}{Course}^*$ = {Course.t_1, Course.t_2, Course.t_4, Course.t_5}. A non-ordered source of this element `CI1.CM1` is Course.t_1. Therefore, from Definition 16, we obtain an ordered source of `CI1.CM1` as {Course.t_1}.

(2) Ordered traces with order filter using only one range. Consider an order filter with one single range, namely, of the form *[position()=k]* or *[position() = m to n]* specified on R_i in SQL query Q [1]. In our running example in Figure 6.2, both `CS` and `CM` elements are defined using an order filter on the Course relation with a single range.

Intuitively, if we want to delete the view element derived from the tuple at position k in the ordered trace $\overset{o}{R}_i^*$, we need to delete not only the tuple at position k, but also everyone after the end of the range (i.e., after position n).

Definition 17 *Consider deleting an XML view element e computed by the SQL query Q. Assume Q specifies an order filter on R_i as [m to n]. Let a non-ordered source s_i = $\overset{o}{R}_i^*$ $[k]$, $m \leq k \leq n$, that is s_i has position k in the ordered trace. An ordered source $\overset{o}{s}_i$ is defined as $\overset{o}{R}_i^*$ $[k] \cup \overset{o}{R}_i^*$ $[position() > n]$.*

Let us now consider an example of deleting a `CM` element, say the course `Math` (`CI1.CM1`) taught by professor `David Finkel`. An ordered source is given by {Course.t_1; Course.t_5}. As another example, consider the view update to be deleting the `CourseSecond` element (`CI1.CS1`). An ordered source for this delete is { Course.t_2, Course.t_4, Course.t_5 }.

[1] Note that [position()=k] is equal to [position() = k to k]

Definition 18 *Consider inserting view element e, where Q is the same as in Rule 17. Also Q specifies an order filter on R_i as [m to n]. Let the non-ordered source s_i = $\overset{o}{R}{}_i^*\ [k]$, $m \leq k \leq n$, that is it has position k in the ordered trace. If $\overset{o}{R}{}_i^*\ [n] = \emptyset$, an ordered source is defined as $\overset{o}{R}{}_i^*\ [k]$; otherwise no ordered source exists for e.*

Definition 18 follows the same intuition as the non-ordered source definition for deletion (Definition 17). If an element already exists in the view that is derived from $\overset{o}{R}{}_i^*\ [n]$, then this view element will disappear after insertion (view side effect caused by position change). Thus there is no ordered source exists and we should reject this view insertion.

As an example, assuming an insertion to insert "History" as a new CourseMore element of professor Tim Merrett. We will insert Course tuple (c31, History, p2). We consider c31 $>$ c3 in lexicographic order.

As another example, consider inserting a new course *Music* as the first course for professor *David Finkel*. This update will always cause view side effect, as it will cause the course *Biology* to disappear from the view. Here, Rule 18 will say that there is no ordered source for this insertion.

(3) Ordered traces with order filter using multiple ranges. Consider an order filter with multiple ranges defined by [r_1, r_2,...,r_n] (multiple positions) specified on R_i. For example, consider the view in Fig. 6.8. The view query is slightly modified from the view query in Fig. 6.2 by replacing the binding of `c2` with $c[1,3,5].

Let us consider deleting the CourseMore element `Math` (`CI1.CM1`) from this view. This view element can be deleted by deleting Course.t_1 and

```
FOR $p IN DOCUMENT(Professor/ROW)
ORDER BY $p.pid
RETURN
    <ClassInfo>
        <Professor> $p/pname/text() </Professor>,
        FOR $c IN DOCUMENT(Course/ROW)
        WHERE $p.pid = $c.pid
        ORDER BY $c.cid
        RETURN
            FOR $c2 IN $c[1,3,5]
            RETURN
                <CourseMore> $c2/cname/text()</CourseMore>
    </ClassInfo>
```

```
<ClassInfo>
    <Professor>David Finkel</Professor>
    <CourseMore>Math</CourseMore>
    <CourseMore>Biology</CourseMore>
</ClassInfo>
<ClassInfo>
    <Professor>Tim Merrett</Professor>
    <CourseMore>English</CourseMore>
</ClassInfo>
```

Figure 6.8: View query for *ClassView*

Course.t_2. The intuition behind such a deletion is as follows: We need to maintain the successor-predecessor "distance" even after the deletion (this distance was always 1 in the previous case, and hence can naturally be maintained).

Definition 19 *Consider deleting an XML view element e computed by the SQL query Q. Assume Q specifies an order filter as multiple ranges on R_i as $[r_1, r_2, \ldots, r_n]$. Let d_j denote $r_{j+1} - r_j$, $1 \leq j \leq n-1$. Let a non-ordered source be $s_i = \overset{o}{R}{}_i^* [r_k]$, where $1 \leq k \leq n$. The ordered source for deleting e is defined as follows.*

- *Case-Dec: if $d_{k+j} > d_{k+j+1}$ (distances between ranges decreases), for $1 \leq j \leq n-k-2$, then no ordered source exists for e.*

- *Case-Inc: otherwise, the ordered source consists of elements of R_i^* from the following three sets of positions: (1) elements at positions r_k to $r_{k+1} - 1$, (2) elements at positions $r_{k+j+1} + 1$ to $r_{k+j+1} + d_{k+j+2} - d_{k+j+1}$, $1 \leq j \leq n - k - 2$, and (3) elements at positions after r_n.*

For example, consider the view defined in Figure 6.9, which is a slightly modification for our motivating view which is a slightly modified view query from Fig. 6.2 by replacing the binding of `c2` with $c[1,3,4].

Let us consider deleting the CourseMore element `Math` (`CI1.CM1`) from this view. This view element cannot be achieved without causing any view side effects on CourseMore node. We need to maintain the successor-predecessor "distance" even after the deletion (this distance was now decreasing now and hence cannot be maintained).

Consider the following example (Figure 6.10) for Case-inc. Let us consider deleting the CourseMore element `Math` (`CI1.CM1`) from this view. This view element can be deleted by deleting Course.t_1 and Course.t_3. The intuition behind such a deletion is as follows: We need to maintain the successor-predecessor "distance" even after the deletion. This distance is increasing, and hence can be maintained by removing those increased part. In our example, we need to remove c[3] to keep the distance to 1. We also need to remove all courses after c[4], if there is any.

The definition for insertion is similar to that of deletion as described below.

Definition 20 *Consider inserting element e, where Q, d_j and s_i are as defined in Rule 19 above. The ordered source for inserting e is defined as follows:*

```
FOR $p IN DOCUMENT(Professor/ROW)
ORDER BY $p.pid
RETURN
    <ClassInfo>
        <Professor> $p/pname/text() </Professor>,
        FOR $c IN DOCUMENT(Course/ROW)
        WHERE $p.pid = $c.pid
        ORDER BY $c.cid
        RETURN
            FOR $c2 IN $c[1, 3, 4]
            RETURN
                <CourseMore> $c2/cname/text()</CourseMore>
    </ClassInfo>
```

```
<ClassInfo>
    <Professor>David Finkel</Professor>
    <CourseMore>Math</CourseMore>
    <CourseMore>Biology</CourseMore>
    <CourseMore>Chemistry</CourseMore>
</ClassInfo>
<ClassInfo>
    <Professor>Tim Merrett</Professor>
    <CourseMore>English</CourseMore>
</ClassInfo>
```

Figure 6.9: View query for *ClassView*

- *if* $\overset{o}{R}{}_i^* [r_n] \neq \emptyset$*, then there is no ordered source.*
- *If Case-Dec (as defined in Definition 19) holds, then there is no ordered source.*
- *Case-Inc (as defined in Rule 19): otherwise, the ordered source consists of* $\overset{o}{R}{}_i^* [r_k]$*, as well as dummy tuples inserted into the following positions (1) positions* $r_k + 1$ *to* $r_{k+1} - 1$*, (2) elements at positions* $r_{k+j+1} + 1$ *to* $r_{k+j+1} + d_{k+j+2} - d_{k+j+1}$*,* $1 \leq j \leq n - k - 2$*.*

(4) Ordered traces with general order filter expression. Consider an order filter with order filter expression, such as [position() mod d = n] (evenly dis-

```
FOR $p IN DOCUMENT(Professor/ROW)
ORDER BY $p.pid
RETURN
    <ClassInfo>
        <Professor> $p/pname/text() </Professor>,
        FOR $c IN DOCUMENT(Course/ROW)
        WHERE $p.pid = $c.pid
        ORDER BY $c.cid
        RETURN
            FOR $c2 IN $c[1, 2, 4]
            RETURN
                <CourseMore> $c2/cname/text()</CourseMore>
    </ClassInfo>
```

```
<ClassInfo>
    <Professor>David Finkel</Professor>
    <CourseMore>Math</CourseMore>
    <CourseMore>Physics</CourseMore>
    <CourseMore>Chemistry</CourseMore>
</ClassInfo>
<ClassInfo>
    <Professor>Tim Merrett</Professor>
    <CourseMore>English</CourseMore>
    <CourseMore>Spanish</CourseMore>
</ClassInfo>
```

Figure 6.10: View query for *ClassView*

tributed, but infinite). For example, the view in Fig. 6.11 choose all courses at odd position.

Definition 21 *Consider deleting an XML view element e computed by the SQL query Q. Assume Q specifies an order filter on R_i as [position() mod d = n]. Let a non-ordered source $s_i = \overset{o}{R}{}_i^*\ [k]$, where $k \bmod d = n$. The ordered source consists of tuples in $\overset{o}{R}{}_i^*$ from position k to $k + d - 1$.*

For example, consider deleting the CourseMore element `Math` (`CI1.CM1`) from this view. This view element can be deleted by deleting Course.t_1 and Course.t_2.

```
FOR $p IN DOCUMENT(Professor/ROW)
ORDER BY $p.pid
RETURN
    <ClassInfo>
        <Professor> $p/pname/text() </Professor>,
        FOR $c IN DOCUMENT(Course/ROW)
        WHERE $p.pid = $c.pid
        ORDER BY $c.cid
        RETURN
            FOR $c2 IN $c[position() mod 2 = 1]
            RETURN
                <CourseMore> $c2/cname/text()</CourseMore>
    </ClassInfo>
```

```
<ClassInfo>
    <Professor>David Finkel</Professor>
    <CourseMore>Math</CourseMore>
    <CourseMore>Biology</CourseMore>
</ClassInfo>
<ClassInfo>
    <Professor>Tim Merrett</Professor>
    <CourseMore>English</CourseMore>
</ClassInfo>
```

Figure 6.11: View query for *ClassView*

Definition 22 *Let e and Q be the same as in Rule 21. Also Q specifies an order filter on R_i as [position () mod d = n]. Let the non-ordered source $s_i = \overset{o}{R}{}_i^* [k]$, where $k \bmod d = n$. The ordered source consists of s_i and a set of dummy tuples from position $k+1$ to $k+d-1$.*

(5) Ordered traces with relative order functions.

Order filter with relative order function first() can always be converted into an absolute order function, since first() will always return the fixed number 1. For example, `$c[position() = first()]` can be converted into `$c[1]`.

On the other hand, relative order function last() cannot be converted

into an absolute order function, since last() is evaluated as different number for different instances. However, views in this case can always be considered as symmetric to the corresponding case with first(). For example, the view element e defined by `[position()=last()-k]` on $\overset{o}{R}{}_i^*$ can be deleted by deleting tuples from position 1 to `[last() - k]`. Other categories can be derived similarly.

6.5 Related Work

A lot of effort has been put into building XQuery engines in research [CKS+00, SKS+01, FKS+02, MFK01b, PB02, DT03] and in commercial database systems [BKKM00, CX00, Rys01]. Order as a key issue in XML query processing has not been addressed adequately so far.

To the best of our knowledge, however order as a key issue specific to the XML data model has not yet been addressed by any of those research projects nor by any of the commercial systems.

[TVB+02] assessed order issue in the XML querying processing context. [TVB+02] proposed three *order encoding methods* to represent XML order in the relational data model and an algorithm for translating ordered XPath expressions into SQL using these encoding methods. The performance of the ordered-encoding methods on a workload of ordered XML queries are also presented. However, the proposed algorithm is *dependent* on and *specific* to the loading and encoding strategy used in building the relational database. That is, (1) the knowledge of loading and encoding is required by the translation algorithm (2) different loading and encoding strategy re-

quire different translation algorithm. In addition, not only the translation strategies proposed, but also the performance studies described, mainly concentrate on the correctness and efficiency of XPATH translation and evaluation. The complexity of handling the order-sensitive XQuery nested structure is not addressed.

The Agora system [MFK01a], which stores XML in relational tables, is one of the few systems that provides support for handling order-sensitive XQuery expressions. XQuery queries are first normalized, then translated and rewritten into SQL queries to be executed over the relational tables. However, this solution is limited to XQuery queries that semantically match SQL and can successfully be translated and rewritten into SQL. Such supported category of queries handling order is an expensive process where an XQuery is translated into many SQL queries requiring several passes and materializing of intermediate XML results. [SKS$^+$01] introduces mechanisms to publish relational data as XML documents using an extension to SQL. The use of a sorted outer union approach is proposed to retrieve the relational data needed for constructing XML documents when the resulting XML document does not fit into main memory. However, this approach performs unnecessary additional work to support user-defined ordering as it produces total ordering even when only partial ordering is needed. Timber [JAKC$^+$02], a native XML data management system, has the ability to deal with order in query processing. However to preserve order, sorting for some of the intermediate results appears to be required during execution [JAKC$^+$02]. The order handling strategy in Timber is built on top of the node $start$-end-$level$ labeling described above. Hence, it suffers from

the disadvantages described above.

Chapter 7

Conclusions of This Dissertation

In many database applications views play an important role as a means to structure information with respect to specific users' needs. Views also provide the support for logical data independence. With XML [W3C98] becoming the standard for interchanging data between web applications, XML views are commonly used by many applications. Update operations are essential for applications using XML views, especially in dynamic environments.

In this dissertation work, we provide scalable solutions to support updating through XML views defined over the relational databases. We classify the view updating problem into different semantics. We have mainly focused on two particular semantics, namely, the update-public semantic and the update-local semantic, given their popularity, importance and gen-

erality. We have provided a novel schema-centric approach for XML view updating under the update-public semantic as well as a local-to-global service framework for XML view updating under the update-local semantic. The conclusions of this dissertation work are listed below.

In part I, we extend the concept of a "clean source" for relational databases [DB82] into the concept of a "clean extended-source" suitable for XML. We propose the clean extended-source theory for determining whether a correct view update translation exists. We prove the correctness of our clean extended source theory, which then serves as a theoretical foundation for XML view updating problem under various update semantics.

In part II, we design practical algorithms to determine whether a given update over the XML view is indeed translatable under the update-public semantic. We propose a schema-centric approach named HUX. HUX first bridges the XML and relational view update problem by treating the XML view as a "composition" of a set of relational views. Existing solutions from the relational scenario are thus applicable. HUX then addresses the hierarchical model property of the XML view updating problem by considering the relationship among the mapping relational views.

HUX is a schema-centric solution given that it utilizes the schema of the underlying source to effectively prune updates that are guaranteed to be not translatable and pass updates that are guaranteed to be translatable directly to the SQL engine. Only those updates that could not be classified using schema knowledge are finally analyzed by examining the data. This required data-level check is further optimized under schema guidance to prune the search space for finding a correct translation. We have imple-

mented the algorithms, along with respective optimization techniques in HUX. We also report experiments assessing its performance benefit and usefulness in practical scenarios.

In Part III, we propose a practical framework, called LoGo, that provides flexible view updating services under update-local semantic. LoGo localizes the view update translation, while preserves the properties of view updates being side-effect free and being always updatable.

LoGo supports on-demand merging of the local database of the subject view into the public database (also called global database), while still guaranteeing the subject view being free of side effects. A flexible synchronization service is provided in LoGo that enables all other views defined over the same public database to be refreshed, i.e., synchronized with the publically committed changes, if so desired. Experimental results confirm the effectiveness of the proposed services, and highlight its performance characteristics.

In Part IV, we consider the XML view updating problem under order-sensitive semantic. We propose the O-HUX approach to classify the order syntax in the XML view definition into different categories. For each category, we design a set of rules that identify order-sensitive candidate update translations. We are the first to study updating order-sensitive XML views, to the best of our knowledge. We extend clean source theory to order sensitive semantics. Based on the order-sensitive clean source theory, we develop the O-HUX algorithm that guarantees view side effect free semantics while considering most of the XQuery order constructs.

Chapter 8

Ideas for Future Work

8.1 Condition-based Set Updates

So far all our discussion focuses on the single element updates. In the future, we would like to consider condition-based set updates, which means updating a set of elements that satisfy certain conditions. For example, the user can delete all students enrolled in the "Math" class.

Intuitively, we can treat a condition-based deletion as a "composition" of single view element deletions. The schema-level decision on updating a set of elements of v can stay the same as updating single element of v. For example , if deleting a single element of v is translatable based-on schema-level knowledge, then deleting a set of elements satisfying certain condition is also translatable.

However, this is an aggressive solution. It sometimes rejects updates, which otherwise may be translatable. For example, consider the view in Figure 8.2, which is defined over the relational database in Figure 8.1. Con-

sider a deletion removing a single view tuple, e.g., delete from V1 where professor= David Finkel and course = Math and credit = 3. This deletion is not translatable under the update-public semantic.

Professor

tid	pid	pname
t_1	p1	David Finkel
t_2	p2	Tim Merrett

Course

tid	cid	cname	credit
t_1	c1	Math	3
t_2	c2	Physics	2
t_3	c3	Math	2

Teaching

tid	pid	cname
t_1	p1	Math
t_2	p1	Physics
t_3	p2	Math

Figure 8.1: The example relational database

```
CREATE VIEW V1 AS {
SELECT P.pname AS professor,
       C.cname AS course,
       C.credit AS credit
FROM Professor P, Course C,
     Teaching T
WHERE P.pid = T.pid
      AND T.cname = C.cname}
```

(a)

professor	course	credit
David Finkel	Math	3
David Finkel	Physics	2
David Finkel	Math	2
Tim Merrett	Math	3
Tim Merrett	Math	2

(b)

Figure 8.2: A relational view (b) defined by the view query (a) over the relational database in Figure 8.1

Now consider three deletions, which removes the first, the second and the third tuples from the view V1 in Figure 8.2 respectively. None of these single tuple deletions is translatable. However, the condition-based deletion "Deleting from V1 where professor = David Finkel" can be easily achieved by deleting the professor David Finkel from the Professor relation. Thus by considering the condition-based deletion simply as a composition of a sequence of single element deletions, we are aggressively rejecting some translatable updates.

A better way of achieving the condition based update would be to perform an analysis on the view side effects, which are collected by decomposing the condition-based update into a sequence of single element deletions.

If side effects only appear on elements to be deleted by the condition-based deletion, then it will be canceled out internally. We thus should conclude that the condition-based update is translatable.

This checking can be done to some degree based on the schema knowledge, as done in the relational scenario [DB82]. It can also be performed by examining the actual data through an exhaustive search in the relational scenario [CWW00]. In the XML scenario, the problem needs to be further considered to fully explore the extra schema information that can be gathered from the XML hierarchy.

8.2 Updating XML Views Published over XML Documents

Given an XML view over XML data, the problem of how to check the updatability of the view elements and further give the correct and efficient translation of this view update still remains unsolved.

Due to the similarity of this problem to the relational view problem, many concepts and previous studies can be reused for this XML view update problem. However, because of the hierarchical structure of XML and expressive query statements, there are some situations that cannot be handled by former solutions.

Let us consider the XML document with its schema as in Figure 8.3. We also present the schema tree of this XML document in Figure 8.4. Note the base schema element $course$ is recursive, as a course may require some other courses as pre-requisite courses. In Figure 8.4, an arrow starts and

ends at $course$ to denote that it is recursive.

```
<!DocType root[
 <!Element root( institute*)>
 <!Element institute (name, department+)>
 <!Element department (name, professor+,
    course+)>
 <!Element professor( name, student*)>
 <!Element student( name)>
 <!Element course( name, course*)>
 <!Element name( #PCDATA)>]>
<root>
 <institute>
   <name> WPI </name>
   <department>
    <name> CS</name>
    <professor>
     <name>  Henry </name>
     <student>
      <name>John </name>
     </student>
     <student>
      <name>  Joe </name>
     </student>
    </professor>
    <course>
     <name>  Database</name>
     <course>
      <name> algorithm </name>
     </course>
    </course>
   </department>
 </institute>
</root>
```

Figure 8.3: XML document D with Schema(D)

Consider two queries over D, as shown in Figure 8.5 and 8.6. In Figure 8.5, (a)is the XQuery statement which defines the view. (b) is the view schema tree that corresponds to the XQuery. (c)is the view instance tree generated by the XQuery and XML document D. The same goes with Figure 8.6. Note in Figure 8.6(a), $course_1$ and $course_2$ correspond to the same view schema node, we use subscript to differentiate them.

A user may want to delete $student_1$ in Figure 8.5(c). We can try to delete $student_1$ in D. This update exactly performs the view update and is a correct translation. However, let us consider how to translate if the user wants to delete $course_1$ in Figure 8.6(c). If we delete $course_1$ in D, this update would cause $course_2$ and its descendants to be removed in Figure 8.6(c). Therefore, it is not a correct translation.

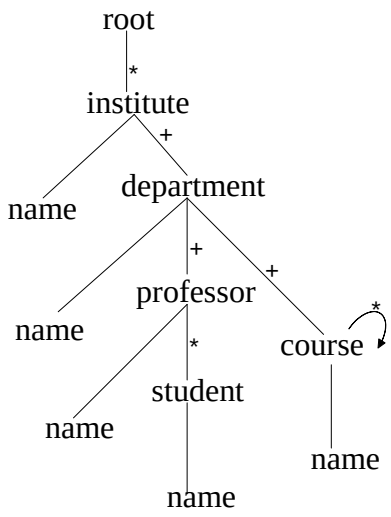


Figure 8.4: XML Schema tree

Intuitively, for the above example, the recursion of the underlying XML database makes the problems harder. XML schema contains cardinality information indicating the number of a certain kind of elements. Also XML query languages are more expressive. For example, "//" in XPath binds the variable to different elements that may appear at different locations of the XML document. Such elements could be ancestor-descendant relationships. For instance, $course_1$ and $course_2$ in Figure 8.3 bind to $\$course$ in Figure 8.6(a). Side-effects due to these features in XML should be considered.

Besides the recursion, the order issue also becomes complicated in this scenario, since the order-based axes in XPath introduced in Section 6.2.1 will now be used in the view query. This in turn may affect the clean extended source theory, as well as the practical approach of handling updates.

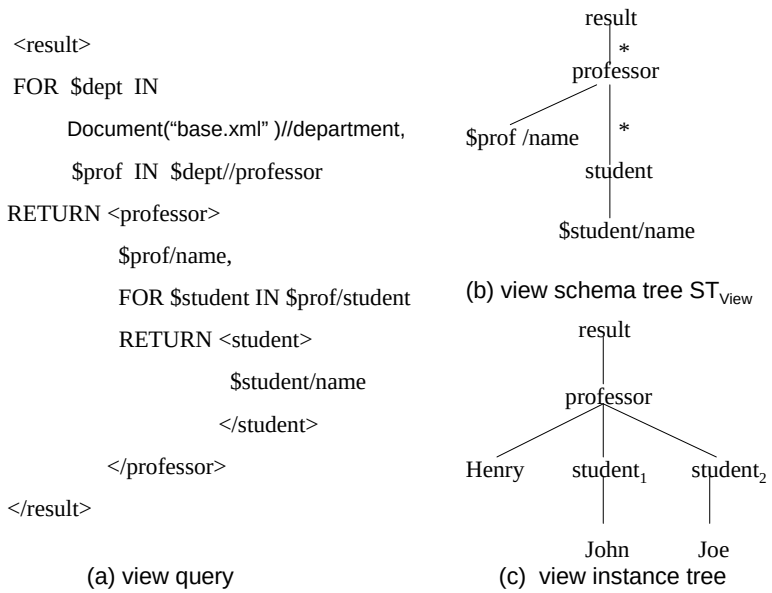


Figure 8.5: Query Q_1 and corresponding view

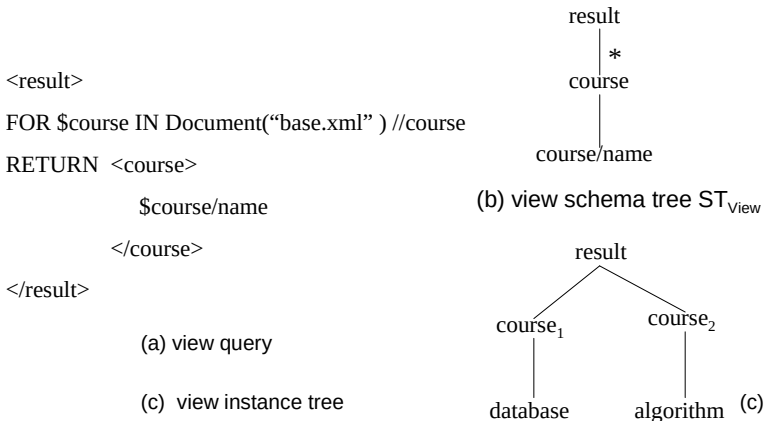


Figure 8.6: Query Q_2 and corresponding view

8.3 Additional Thoughts

Transaction based XML view updating. So far, we only considered a single update translation. Updates of different types over the same view can be grouped together into one transaction. New issues related to update translatability may arise. This may lead to new update semantics.

Update Interpretation. In this dissertation, we chose the most straightforward (and natural) way to interpret the behavior of a given update on the XML views. For example, to delete an element of the XML view is interpreted as deleting the whole document sub-tree, which is rooted at the to-be-deleted element. However, in some scenario, this deletion may need to be interpreted differently, such as to just delete the element tag. Especially, in cases when we can move an XML view element inside the XML view. The new way of interpreting the update behavior then would also need to be addressed. This may lead to the design of new update translation strategies.

Bibliography

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. Foundations of databases. In *Addison Weseley*, page 293, 1995.

[BDH03] Vanessa P. Braganholo, Susan B. Davidson, and Carlos A. Heuser. On the Updatability of XML Views over Relational Databases. In *WEBDB*, pages 31–36, 2003.

[BDH04] Vanessa P. Braganholo, Susan B. Davidson, and Carlos A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, pages 276–287, 2004.

[BDHar] Vanessa Braganholo, Susan Davidson, and Carlos A. Heuser. PATAXO: a framework to allow updates through XML views. In *TODS*, 2006 to appear.

[BKKM00] Sandeepan Banerjee, Vishu Krishnamurthy, Muralidhar Krishnaprasad, and Ravi Murthy. Oracle8i - The XML Enabled Data Management System. In *ICDE*, pages 561–568, 2000.

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[Bry90] Francois Bry. Intensional Updates: Abduction via Deduction. In *Proceedings of the seventh international conference on logic programming*, 1990.

[BS81] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. In *ACM Transactions on Database Systems*, pages 557–575, Dec 1981.

[BSKW91] Thierry Barsalou, Niki Siambela, Arthur M. Keller, and Gio Wiederhold. Updating Relational Databases through Object-Based Views. In *SIGMOD*, pages 248–257, 1991.

[CA81] C. R. Carlson and A. K. Arora. The Updatability of Relational Views based on Functional Dependencies. In *COMPSAC*, pages 415–420, 1981.

[CKN03] Surajit Chaudhuri, Raghav Kaushik, and Jeffrey F. Naughton. On Relational Support for XML Publishing: Beyond Sorting and Tagging. In *SIGMOD*, 2003.

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

[CLL02] Ya Bing Chen, Tok Wang Ling, and Mong-Li Lee. Designing Valid XML Views. In *ER*, pages 463–478, 2002.

[CP84] Stavros S. Cosmadakis and Christos H. Papadimitriou. Updates of Relational Views. *Journal of the Association for Computing Machinery*, pages 742–760, Oct 1984.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L. Wienner. Tracing the lineage of view data in a warehousing environment. In *ACM Transactions on Database Systems*, volume 25(2), pages 179–227, June 2000.

[CX00] Josephine M. Cheng and Jane Xu. XML and DB2. In *ICDE*, pages 569–573, 2000.

[DB82] Umeshwar Dayal and Philip A. Bernstein. On the Correct Translation of Update Operations on Relational Views. In *ACM Transactions on Database Systems*, volume 7(3), pages 381–416, Sept 1982.

[dbX] dbXML. dbXML Core. http://www.dbxml.org.

[DD99] Florescu Daniela and Kossmann Donald. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[Dec90] Hendrik Decker. Drawing Updates from Derivations. In *ICDT*, 1990.

[DT03] Alin Deutsch and Val Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *VLDB*, 2003.

[DTCO03] Davis Dehaan, David Toman, Mariano P. Consens, and M. Tamer Ozsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *SIGMOD*, 2003.

[eXc] eXcelon. eXcelon XML platform. http://www.exceloncorp.com/platform/extinfserver.shtml.

[FE01] Leonidas Fegaras and Ramez Elmasri. Query Engines for Web Accessible XML Data. In *VLDB*, 2001.

[FHK+02] Thorsten Fiebig1, Sven Helmer2, Carl-Christian Kanne3, Guido Moerkotte2, Julia Neumann2, Robert Schiele2, and Till Westmann. Natix: A Technology Overview. In *Proc. Web, Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops, 2002*, 2002.

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

[FKS+02] Mary Fernandez, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan. SilkRoute: A Framework for Publishing Relational Data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, 2002.

[FMST01] Mary F. Fernandez, Atsuyuki Morishima, Dan Suciu, and Wang Chiew Tan. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.

[FUV83] Ronald Fagin, Jeffrey D. Ullman, and Moshe Y. Vardi. On the Semantics of Updates in Databases. In *SIGMOD*, pages 352–365, 1983.

[GMW99] Roy Goldman, Jason McHugh, and Jennifer Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *WebDB (Informal Proceedings)*, pages 25–30, 1999.

[Heg90] Stephen J. Hegner. Foundation of Canonical Update Support for Closed Database Views. In *ICDT*, 1990.

[HGP] Human Genome Project. http://www.ornl.gov/sci/techresources/HumanGenome/home.shtml.

[JAKC+02] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, NuweeWi watwattana, Yuqing Wu, and Cong Yu. Timber: A native xml database. In *VLDB*, 2002.

[JMN83] Kioumars Yazdanian Jean-Marie Nicolas. An Outline of BD-GEN: a Deductive DBMS. In *R.E.A. Mason, editor, Proceedings of IFIP 83*, pages 711–717, 1983.

[KCKN03] Rajasekar Krishnamurthy, Venkatesan T. Chakaravarthy, Raghav Kaushik, and Jeffrey F. Naughton. Recursive XML Schemas, Recursive XML Queries and Relational Storage: XML-to-SQL Query Translation. In *ICDE*, 2003.

[Kel85] Arthur 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.

[Kel86a] Arthur M. Keller. Choosing a View Update Translator by Dialog at View Definition Time. In *VLDB*, pages 467–474, 1986.

[Kel86b] Arthur M. Keller. The Role of Semantics in Translating View Updates. *IEEE Transactions on Computers*, 19(1):63–73, 1986.

[Kel87] Arthur. M. Keller. Comments on Bancilhon and Spyratos' "Update Semantics and Relational Views". *ACM Transactions on Database Systems*, 12(3):521–523, 1987.

[KKN02] Rajasekar Krishnamurthy, Raghav Kaushik, and Jeffrey F. Naughton. Optimizing Fixed-Schema XML to SQL Query Translation. In *VLDB*, 2002.

[KM90] Antonis C. Kakas and Paolo Mancarella. Database Updates Through Abduction. In *VLDB*, 1990.

[KM00] Meike Klettke and Holger Meyer. XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statics. In *WebDB*, pages 151–170, 2000.

[LAW99] Tirthankar Lahiri, Serge Abiteboul, and Jennifer Widom. Ozone: Integrating Structured and Semistructured Data. In *DBPL*, 1999.

[LC00] Dongwon Lee and Wesley W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *ER*, pages 323–338, Oct 2000.

[LL92] Tok Wang Ling and Mong-Li Lee. A Theory for Entity-Relationship View Updates. In *ER*, pages 262–279, 1992.

[LLS93] Dominique Laurent, Viet Phan Luong, and Nicolas Spyratos. Updating Intensional Predicates in Deductive Databases. In *ICDE*, 1993.

[Mas84] Yoshifumi Masunaga. A Relational Database View Update Translation Mechanism. In *VLDB*, pages 309–320, 1984.

[May] Wolfgang May. Information Extraction and Integration with Florid: The Mondial Case Study. http://www.informatik.uni-freiburg.de/ may/Mondial/florid-mondial.html.

[MFK01a] Ioana Manolescu, Daniela Floresce, and Donald Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *VLDB, Roma, Italy*, pages 241–250, Sept. 2001.

[MFK01b] Ioana Manolescu, Daniela Floresce, and Donald Kossmann. Answering XML Queries over Heterogeneous Data Sources. In *VLDB*, 2001.

[Mic01] Microsoft Inc. XML Query Language Demo. http://131.107.228.20/xquerydemo/demo.aspx, April 2001.

[ML01] Pedro Joseeee Marroooon and Georg Lausen. On Processing XML in LDAP. In *VLDB*, pages 601–610, 2001.

[PB02] Henry F. Korth PPS Narayan Pradeep Shenoy Philip Bohannon, Sumit Ganguly. Optimizing View Queries in ROLEX to Support Navigable Result Trees. In *VLDB*, 2002.

[Pro] Digital Bibliography Library Project. DBLP Computer Science Bibliography. http://dblp.uni-trier.de/.

[Res] Protein Information Resource. Protein Sequence Database. http://pir.georgetown.edu/.

[RN89] Francesca Rossi and Shamim A. Naqvi. Contributions to the View Update Problem. In *Proceedings of the Sixth International Conference on Logic Programming*, 1989.

[RP02] Kanda Runapongsa and Jignesh M. Patel. Storing and Querying XML Data in ORDBMSs. In *EDBT XML-Based Data Management (XMLDB) Workshop*, 2002.

[Rys01] Michael Rys. Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems. In *VLDB*, pages 465–472, 2001.

[Sah02] Arnaud Sahuguet. Querying XML in the New Millennium. http://kweelt.sourceforge.net, 2002.

[Sch00] Harald Schoning. Tamino - A DBMS designed for XML. In *ICDE*, pages 149–154, 2000.

[Sco02] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, August 2002.

[SKS+01] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John Funerburk. Querying XML Views of Relational Data. In *VLDB*, 2001.

[ST00] Harald Schoning and J. Wasch. Tamino. Tamino - An Internet Database System. In *EDBT*, pages 383–387, 2000.

[STH+99] Jayavel Shanmugasundaram, Kristin Tufte, Gang He, Chun Zhang, David DeWitt, and Jeffrey Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, pages 302–314, September 1999.

[SYU99] Takeyuki Shimura, Masatoshi Yoshikawa, and Shunsuke Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *DEXA*, pages 206–217, 1999.

[TA91] Riccardo Torlone and Paolo Atzeni. Updating Deductive Databases with Functional Dependencies. In *DOOD*, 1991.

[TIHW01] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In *SIGMOD*, pages 413–424, May 2001.

[TPC] TPC Benchmark H (TPC-H). http://www.tpc.org/information/benchmarks.asp.

[TVB+02] Igor Tatarinov, Stratis D. Viglas, Kavin Beyer, Jayavel Shanmugasundaram, Eugena Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD*, 2002.

[W3Ca] W3C. XML Schema. http://www.w3.org/XML/Schema.

[W3Cb] W3C. XQuery Update Facility. http://www.w3.org/TR/xqupdate/.

[W3C98] W3C. XML^{TM} . http://www.w3.org/XML, 1998.

[W3C03] W3C. XQuery 1.0 Formal Semantics. http://www.w3.org/TR/query-semantics/, June 2003.

[Wil86] Marianne W. Wilkins. A Model-theoretic Approach to Updating Logical Databases. Technical report, Computer Science Department, Stanford University, 1986.

[WLL03] Guoren Wang, Mengchi Liu, and Li Lu. Extending XML-RL with Update. In *IDEAS*, page 66, 2003.

[WMR03] Ling Wang, Mukesh Mulchandani, and E. A. Rundensteiner. Updating XQuery Views Published over Relational Data: A Round-trip Case Study. In *XML Database Symposium*, pages 223–237, 2003.

[WR04] Ling Wang and E. A. Rundensteiner. On the Updatability of XQuery Views Publised over Relational Data. In *ER*, pages 795–809, 2004.

[WRM06] Ling Wang, E. A. Rundensteiner, and Murali Mani. UFilter: A Lightweight XML View Update Checker. In *ICDE, poster paper*, 2006.

[WRMJ05] Ling Wang, E. A. Rundensteiner, Murali Mani, and Ming Jiang. HUX: An Efficient Schema-centric Approach on Handling Updates in XML. Technical Report WPI-CS-TR-05-11, Computer Science Department, WPI, 2005.

[WRMJ06] Ling Wang, E. A. Rundensteiner, Murali Mani, and Ming Jiang. HUX: A Schemacentric Approach for Updating XML Views. In *CIKM, poster paper*, 2006.

[WRMar] Ling Wang, E. A. Rundensteiner, and Murali Mani. Updating XML Views Published Over Relational Databases: Towards the Existence of a Correct Update Mapping. In *DKE Journal*, 2005 to appear.

[XD] XML-DB. XUpdate — XML Update Language. http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html.

[XH] X-Hive. X-Hive/DB. http://www.x-hive.com.

[YK06] Yannis Velegrakis Yannis Kotidis, Divesh Srivastava. Updates Through Views: A New Hope. In *VLDB*, 2006.

[ZDW+03] Xin Zhang, Katica Dimitrova, Ling Wang, Maged EL-Sayed, Brain Murphy, Luping Ding, and Elke A. Rundensteiner. RainbowII: Multi-XQuery Optimization Using Materialized XML Views. In *Demo Session Proceedings of SIGMOD*, page 671, 2003.