# Rainbow: Relational Database Auto-tuning for Efficient XML Query Processing

Xin Zhang

A Proposal for a Dissertation in Computer Science

Worcester Polytechnic Institute, Worcester, MA

May 2001

**Committee Members:**
Prof. Elke A. Rundensteiner, Worcester Polytechnic Institute. Advisor.
Prof. Nabil I. Hachem, Worcester Polytechnic Institute.
Prof. Karen Lemone, Worcester Polytechnic Institute.
Dr. Gail Mitchell, BBN Technologies.
Dr. Wang-Chien Lee, Verizon Information Technology.

## Abstract

Recently, the XML representation has emerged as a promising new standard for modeling and sharing data on the web. Rather than developing XML data management services from scratch, there is great interest to exploit existing relational database technology as backend engine to store, retrieve, and query XML data set due to its maturity and performance. However one XML data set could be mapped to possibly many different relational schemata, each offering different query performance for a given XML-driven query workload. Hence, mechanisms that flexibly support and optimize such XML-to-relational mapping strategies are important for the success of XML management systems.

In this dissertation, we propose to design an XML management system, called Rainbow, for the storage, retrieval, and querying of XML documents based on relational database technology. We propose to equip the Rainbow system with a flexible mapping mechanism that not only allows for a wide variety of mapping XML documents into different appropriate relational schemata, but also explicitly models and manages the chosen mapping via a metadata model. This meta-data driven mapping model then in turn can be exploited by Rainbow for accomplishing database management services, such as loading, exporting, updating, and querying.

Furthermore, rather than requiring a database administrator to choose a particular mapping and thus to be very familiar with both the XML and the relational database technology, we propose to develop an automatic map designer, called XTuner, for Rainbow. Given an XML application workload that defined a set of XML queries expressed in the standard query language XQuery and their relative importance factors, the XTuner will automatically select an optimized mapping and hence resulting conceptual relational schema design that optimizes the workload performance. For this optimization, we will design a strategy for XQuery to SQL query translation as well as a cost model. The Rainbow system will be fully implemented using XQuery as XML query language and Oracle as backend relational store. Experimental studies will be conducted to evaluate the effectiveness of the Rainbow system for XML management in general, and the effectiveness of the XTuner tool for optimizing XML query processing in particular.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Touted as the ASCII format of the future, the Extensible Markup Language (XML) [3] has recently been adopted as markup language for information modeling and exchange in many web-based industries. By enabling automatic data flow between businesses, XML is pushing the world into the electronic commerce (*e*-commerce) era. Collecting, analyzing, mining, and managing XML data will hence become tremendously important tasks for future web-based applications.

Over the past decades, database systems have emerged as the traditional tools for managing data. After many years of development, relational database technology has matured and contributed significantly to the rapid growth of various industries. Relational database management systems (DBMS) are a proven technology for managing business data. Commercial relational database products embody years of research and development in areas as diverse as modeling, storage, retrieval, update, indexing, transaction processing, and concurrency control, to just name a few. Work now continues to add capabilities to a DBMS to address new kinds of data, such as multimedia [26], object-oriented [6], or spatial-temporal data types [38, 20]. With more and more data being captured in XML formats, it is an obvious next step to further extend relational or object-relational systems to accommodate XML data. Such an approach may avoid re-inventing database technology developed from the ground up to manage XML data. More importantly, it may succeed in taking advantage of the power of relational database technology and the wealth of experience in optimizing and using the technology.

There are however a number of technical issues to examine and overcome when bringing XML data into a relational database for management. These issues include for example defining a relational schema for the XML data, loading the XML data into the relational

database, and transforming XML queries expressed in an XML query language, such as XQuery [53], into meaningful SQL queries.

## 1.2 State of the Art for XML Data Management

Current enterprise DBMS vendors, such as DB2 [10] by IBM and Oracle 8i [36], provide XML extensions that require users to first design the desired relational schema for a given DTD and to also define a specific mapping between the DTD and the user-designed schema for the loading of XML documents into the DBMS system. Some tools are provided to assist the user in this manual process, such as DXX [10] and XSU [35]. Both of which are described in more detail in related work in Section 5. However, such a maintenance approach only works well for generating a relatively simple relational schema ideally without complex relationships between tables. It is also most appropriate for a small number of DTDs. This approach requires the database administrators to be experts of both XML and relational techniques. For rather complex DTDs and a more robust relational schema, the manual approach becomes more difficult, and requiring specialized expertise in both relational database design and XML mapping. Hence a more advanced approach to generating the relational schema and defining the mapping definition for the data loading process is needed.

For that purpose, [14, 44] have recently proposed automatic approaches for generating a relational schema for a given DTD or set of XML documents. However, to generate the schema, these approaches require either mining of the XML documents [14] or prior simplification of the DTD [44].

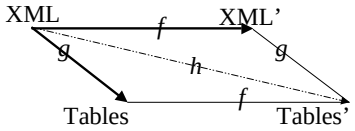


Figure 1.1: Mapping and Restructuring

*f* is restructuring, *g* is cross-model mapping, and *h* is a flexible mapping.

To be able to better characterize previous approaches, we distinguish between the two concepts of mapping as **cross-model mapping** between data models (such as between the XML model and the relational model), and as **restructuring** of data representations within the same data model (say inlining of a hierarchical XML structure). In Figure 1.1, *f* denotes the notion of restructuring, *g* denotes the notion of a fixed cross-model mapping, and *h* denotes the notion of a flexible mapping from XML data model to relational data model. DB2 DXX [10] first requires the DBA to specify the mapping from DTD to RDB manually,

and then loads the XML data based on that mapping. Their approach is very complex in order to achieve the flexibility of their mapping, which is shown as $h$ in Figure 1.1. Oracle XSU [35]'s approach to XML management is to first require the user to convert the desired XML data into a canonical XML format via XSL transformations illustrated as $f : XML \rightarrow XML'$ in Figure 1.1. Thereafter, it can automatically load the XML data using a predefined fixed mapping strategy (assuming the relational table schemata had been created beforehand). This corresponds to $f : XML \rightarrow XML'$ followed by $g : XML' \rightarrow Tables'$ in Figure 1.1, i.e., $g(f(xml))$. In this work, we now propose to take an alternate approach towards tackling this problem 1) by utilizing the $(f(g(XML))$ path to traverse the mapping space in order to move the computation complexity into relational database and 2) by completely automating the process.

## 1.3 Rainbow Framework

**Goals.** We now propose a new framework, called Rainbow, where the user can manage existing XML documents in one centralized repository, and can query the repository by issuing XML queries. The goal of the Rainbow system is to allow seamless use of XML without requiring awareness of the underlying relational database engine. Our system thus provides several key functionalities:

1. XML automatic loading;
2. Flexible mapping;
3. Update propagation through flexible mapping;
4. Query translation through the flexible mapping;
5. Schema optimization to select an optimized mapping for a given XML workload.

Our goal is to generate an XML repository system that has the following features:

- Flexible: Modeling of the mapping is flexible in order to be able to select different performance tradeoffs.
- Executable: All services including query translation and update propagation will automatically adapt their process based on the new chosen mapping without requiring any human interference.
- Efficient: Conceptual/physical schema will be auto tuned in order to reach the most efficient query performance for a given query workload.

**Architecture.** We propose the Rainbow framework to support efficient query processing of XML documents by exploiting relational database technologies. Our system is composed of a DTD manager, a default view manager, a schema creator, a restructurer, an XML query engine, and an optimizer called XTuner as depicted in Figure 1.2.

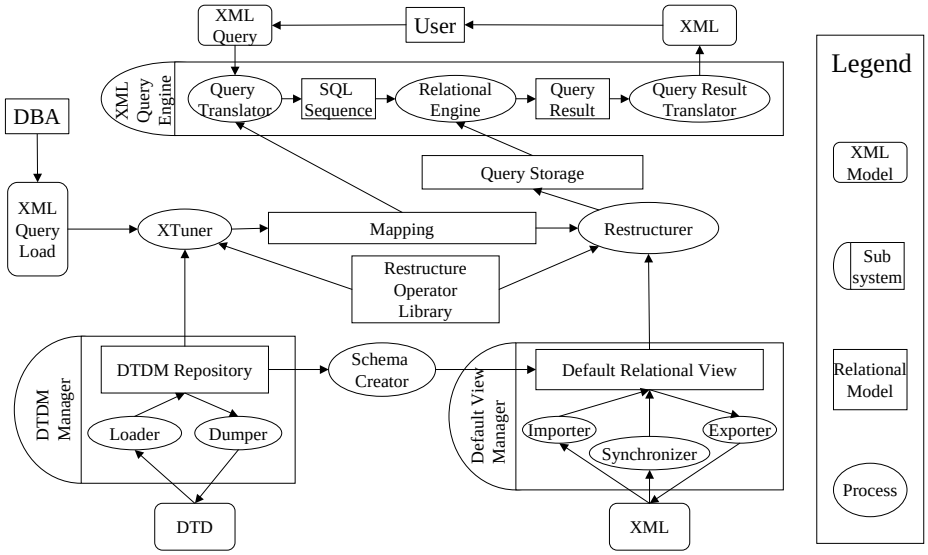


Figure 1.2: Framework of Rainbow.

DTD Manager will load DTD documents into our system by storing them in DTD Metadata (DTDM) tables as part of the system dictionary tables. DTDM tables model the DTD as a collection composed of items, attributes and nesting relationships. After the DTDM repository is loaded, a object-relational schema will be inferred from the DTDM repository by the schema creator.

The default view manager defines a default relational view on top of each XML document. The default view manager maintains XML documents with the help of three modules: an importer, an exporter, and a synchronizer. The importer imports XML compliant to a prior specified DTD into our system. The exporter will export the relational data into XML documents. The synchronizer is used to keep the internal relational representation and external XML representation consistent with each other under data updates.

The restructuring operator library stores a collection of restructuring operators for both DTD restructuring and query optimization purposes. The DTD restructuring operators will propagate the restructuring specified at the DTD level to the underlying relational data. The optimization operators will denormalize the relational schema for different query purposes. A mapping plan specifies a sequence of restructuring operators with their parameters.

The mapping plan is generated by a schema optimizer, called XTuner. First the XTuner translates a given XML query workload into a SQL workload, and then generates the mapping plan based on the database metadata.

The end user can issue XML queries through the XML Query Engine subsystem. The XML query will be translated into a sequence of SQL queries by the query translator based on the mapping provided by the optimizer. Then the relational query engine of the RDBMS will execute the SQL queries and return the corresponding relational query result. The query result translator will translate the query result back into the XML model and return it to the end user.

**Levels of Abstraction in Rainbow.** The data in the Rainbow is described at four levels of abstraction as illustrated in Figure 1.3. From top to bottom, they are XML external schema layer, XML conceptual schema layer, XML physical schema layer, and relational layer, which in turn can have multiple layers.

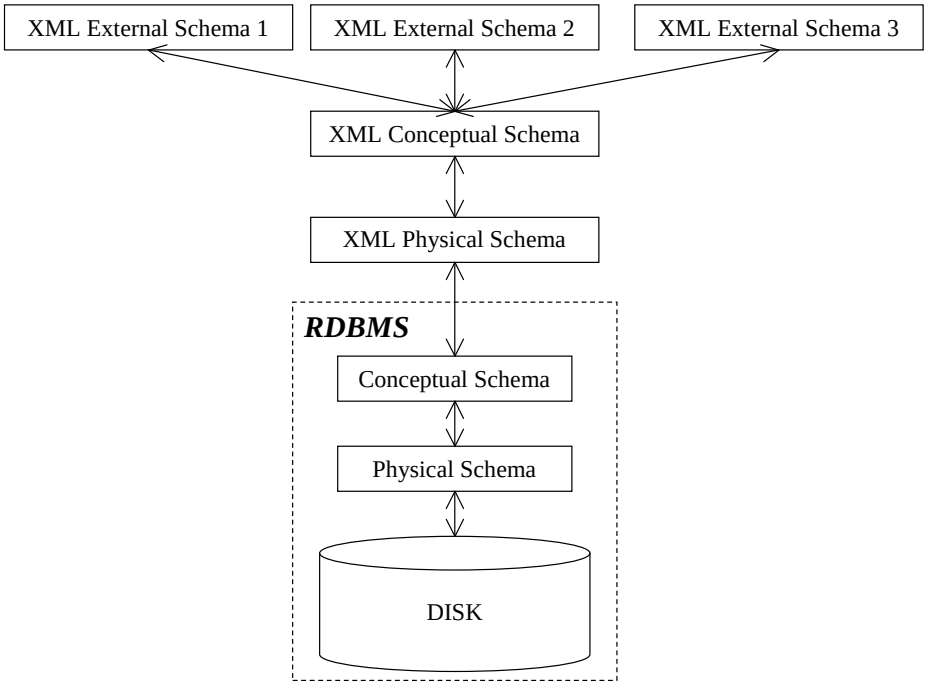


Figure 1.3: Levels of Abstraction in Rainbow.

The XML external schema is composed of XML views of the data tailored to different user groups. The XML conceptual schema describes the stored data in terms of the XML data model. The XML physical schema is a middle layer that wraps relational data into XML data. It also contains any special purpose XML indexes that cannot be applied to the

underlying RDBMS directly.

The relational layer is basically as RDBMS system, that again can be composed of a relational conceptual schema and relational physical schema that are used to store the XML documents.

## 1.4 Research Issues to be Tackled

Here is a list of what main tasks to be done:

### 1.4.1 Flexible Mapping

In our work, we now propose an automatic approach to this mapping problem that offers the following advantages:

- Flexible mapping to optimize diverse user requirements, such as, update or query workload.
- Management of different mapping approaches within one system to flexibly select the most desired one.
- Automatic generation of the relational schema and loading of XML data.

The flexible mapping includes two directions as depicted in Figure 1.4, namely, from XML to the relational data model and vice versa. Let's first look at the direction from XML to the relational model.

We propose to achieve this flexible mapping by a *default relational view* followed by powerful relational restructuring. A default relational view corresponds to a default mapping from the XML to the relational data model. The default relational view will be a set of virtual relational tables generated from a given DTD of the XML documents. These tables are typically normalized. The main purpose for the default relational view is to bring the XML documents into the relational data model, so that they then can be further flexible restructured on top of the default relational view purely within the relational territory, i.e., using SQL and database support.

The relational data to XML data mapping will follow the XQuery [53] standard, where a *default XML view* will be created on top of the relational tables. Here the flexibility is captured by the XQuery on top of the default XML view. The details of the two-way mapping is discussed in Section 3.2.

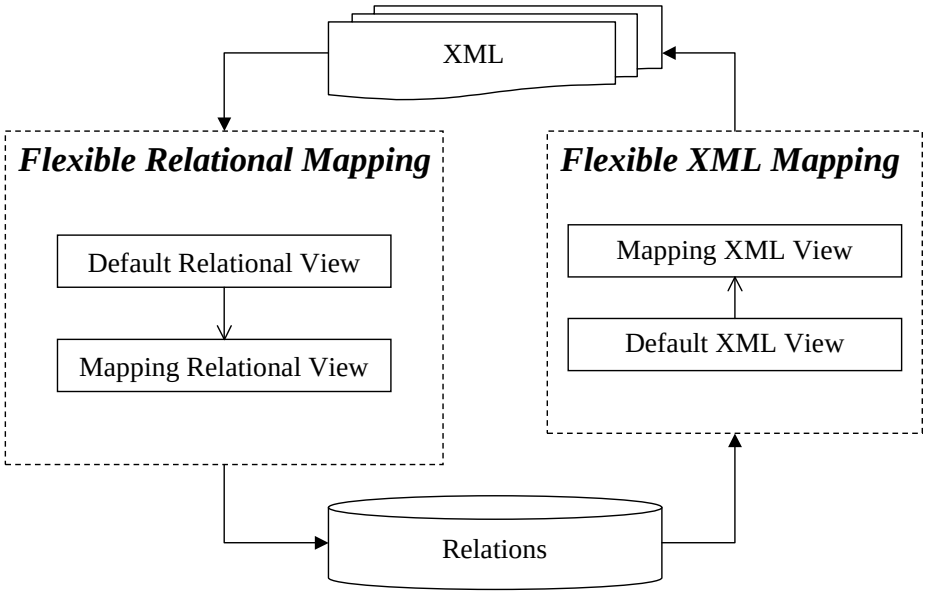


Figure 1.4: Two Way Flexible Mapping.

### 1.4.2 XML Update Processing

Once the XML data is loaded into the relational storage, we could reload the whole XML document for any update to the external XML data sources in order to keep it synchronized with the internal relational storage. However, this could be really expensive. Hence an incremental update strategy for the relational storage is introduced. There are four research issues to be tackled: 1) identify a set of update primitives, 2) bind the updates with one XML query language, in our case, XQuery, 3) identify the XML update and location of its corresponding relational storage, and 4) propagate updates through the mapping. The details are discussed in Section 3.3.

### 1.4.3 XML Query Processing

We plan to use relational database engine to process XML queries. Hence, we first must be able to translate an XML query into SQLs. We have done an initial analysis of this problem based on work proposed in the literature [5]. The details can be found in Section 4.1.

In this proposal we will not consider the XML query optimization in depth. Instead, we assume the XML query plan generated as result by the translator is already optimized. A discussion of related work in XML query optimization and XML indexing techniques can be found in Section 5.7.

### 1.4.4 XML Query Workload and Cost Model

In order to tuning the database, an XML query workload is required. The XML query workload will be composed of both queries and updates with their relative importance factors. In order to optimize the relational schema to achieve the best query and update performance for a given XML workload, a translation algorithm is required to get the corresponding SQL workload.

To compare different relational schemata, a cost model will be proposed to help us decide the cost of a query plan generated from a specific relational schema.

### 1.4.5 Schema Optimization

Schema optimization includes two aspects, conceptual schema optimization and physical schema optimization. Conceptual schema optimization deals with denormalization of the relational schema using materialized views. Physical schema optimization decides upon indexing, clustering techniques, and other such design choices.

In our proposal, we will focused more on the conceptual schema optimization. We assume once the conceptual schema design has been completed, then the physical schema optimization can be conducted by exploiting existing relational technologies plus possibly some novel XML-aware indices.

## 1.5 Targets of this Dissertation

In summary, we have identified the following targets.

1. We develop DTD metadata model and defined its relational schema in order to capture the XML schema information in the relational database.
2. We propose a lossless cross-model mapping from XML to the relational data model and vice versa.
3. We design and implement a flexible loading and dumping algorithm using the above mapping model with support of multiple XML and DTD documents.
4. Based on the default relational view we identify the relationship between the DTD change and the relational schema change.
5. We design an XQuery translation and processing strategy over our flexible mapping using relational database technology.
6. We design a cost model for the XML query plan.

7. We identify the guidelines to denormalize the conceptual schema in order to achieve higher query performance for a given query workload.

8. We implement the system as proof of concept and we discuss preliminary evaluations.

9. We setup the experimental environment to do experimental evaluation to figure out how good the Rainbow system can optimized relational schemata to suit a specific XML query workload.

Tasks 1 through 3 have been completed, tasks 4, 8 and 9 are partially implemented, while the remaining tasks are to be done.

## 1.6 Limitations and Assumptions

In order to achieve our proposal tasks, we have such limitations. In this work, we plan to focus on exploiting existing object-relational technologies to solve the XML query problem. To tackled the hierarchical and ordering nature of XML data, we will use the SQL3 standard, especially object extensions, procedure languages, etc.. In terms of the XML technology, we will handle data updates but not schema changes. For schema tuning, we assume that the XML query plan we generate is optimized, i.e., we do not put particular focus on query optimization itself. Also, we will primarily focus on the conceptual schema optimization, the physical schema optimization is only a secondary concern. We assume that user will not update the optimized query storage directly in our proposal, rather updates and queries are specified against the XML data model.

## 1.7 Outline

This proposal is organized in the following way. Chapter 2 includes the required background knowledge, such as of XML documents, DTDs, and their graph model, and XML query language XQuery. Chapter 3 describes what we have accomplished so far, such as DTD metadata model, lossless XML to relational mapping, import/export technology, and synchronization techniques. We list the remaining proposed tasks for this dissertation in Chapter 4. In that chapter, we will propose the restructuring, XML query translation, cost model, and schema optimization. Chapter 5 lists related work. Finally, we give the intended time schedule in Chapter 6.

# Chapter 2

# Background Material

We now review the technical background needed for this work, in particular, the XML and DTD data model, DOM (Document Object Model), and XML query languages. The following section 2.1 is directly cited from the technical report [55].

## 2.1 XML and DTD

XML (Extensible Markup Language) is currently used both for defining document markups (and thus information modeling) and for data exchange. XML documents are composed of character data and nested tags used to document the semantics of the embedded text. Tags can be used freely in an XML document (as long as their use conforms to XML specification) or can be used in accordance with *document type definitions* (DTDs) [4] to which an XML document declares itself conformance. An XML document that conforms to a DTD is a *valid* XML document.

A DTD is used to define the allowable structures of elements (i.e., it defines allowable tags and tag structure) in a valid XML document. A DTD can include four kinds of declarations: *element type*, *attribute-list, notation*, and *entity*. An element type declaration is analogous to a data type definition; it names an element and defines the allowable content and structure.

An element may contain only other elements (called *element content*) or may contain any mix of other elements and text, which is represented as PCDATA (called *mixed content*). An *EMPTY* element type declaration is used to name an element type without content (it can be used, for example, to define a placeholder for attributes). Finally, an element type can be declared with content *ANY* meaning the type (content and structure) of the element is arbitrary.

Attribute-list declarations define the attributes of an element type. The declaration

includes attribute names, default values and types, such as CDATA, NOTATION, ENTITY, etc.. Two special types of attributes, ID and IDREF, are used to define references between elements. An ID attribute is used to uniquely identify an element; an IDREF attribute can be used to reference that element[1]. Entity declarations facilitate flexible organization of XML documents by breaking the documents into multiple storage units. A notation declaration identifies non-XML content in the XML documents. In this paper, we assume that readers are familiar with the above terminology. For more details refer to [4].

Element and attribute declarations define the structure of compliant XML documents and the relationships among the embedded XML data items. Entity declarations, on the other hand, are used for physical organization of a DTD or XML document (similarly to macros and inclusions in many programming languages and word processing documents). We assume entities declarations can be substituted or expanded to give an equivalent DTD with only element type and attribute-list declarations, since they do not provide information pertinent to the modeling of the data. We call such a result a *logical DTD*. For the rest of this paper, we use DTD to refer to a logical DTD.

## 2.2 Running Example

We give out a running example of a DTD and a conforming XML document of a simple telephone bill application in Figures 2.1 and 2.2, respectively.

```
<!ELEMENT invoice (account_number,
                   bill_period,
                   carrier+,
                   itemized_call*,
                   total)>
<!ELEMENT account_number (#PCDATA)>
<!ELEMENT bill_period (#PCDATA)>
<!ELEMENT carrier (#PCDATA)>
<!ELEMENT itemized_call EMPTY>
<!ATTLIST itemized_call
          no ID #REQUIRED
          date CDATA #REQUIRED
          number_called CDATA #REQUIRED
          time CDATA #REQUIRED
          rate (NIGHT|DAY) #REQUIRED
          min CDATA #REQUIRED
          amount CDATA #REQUIRED>
<!ELEMENT total (#PCDATA)>
```

Figure 2.1: DTD Example of a Telephone Invoice.

```
<invoice>
  <account_number>555 777-3158 573 234 3
  </account_number>
  <bill_period>Jun 9 - Jul 8, 2000</bill_period>
  <carrier>Sprint</carrier>
  <itemized_call no="1" date="JUN 10"
    number_called="973 555-8888" time="10:17pm"
    rate="NIGHT" min="1" amount="0.05"/>
  <itemized_call no="2" date="JUN 13"
    number_called="973 650-2222" time="10:19pm"
    rate="NIGHT" min="1" amount="0.05"/>
  <itemized_call no="3" date="JUN 15"
    number_called="206 365-9999" time="10:25pm"
    rate="NIGHT" min="3" amount="0.15"/>
<total>$0.25</total>
</invoice>
```

Figure 2.2: XML Example of One Telephone Account.

[1] An IDREFS attribute can refer to multiple elements.

When we deal with XML queries, the result may always be a list of *XML fragments*, instead of one well-formed XML document. We call a list of *XML fragments* the *XML forest*.

## 2.3 Document Object Model (DOM)

DOM [50] is a platform and language neutral interface that allows applications to navigate and update the content, structure and style of documents. In the DOM, documents have a logical structure like a list of trees. The example in Figure 2.2 can be represented graphically as a DOM tree in Figure 2.3. The edges illustrate the nesting relationship between parent and child elements.

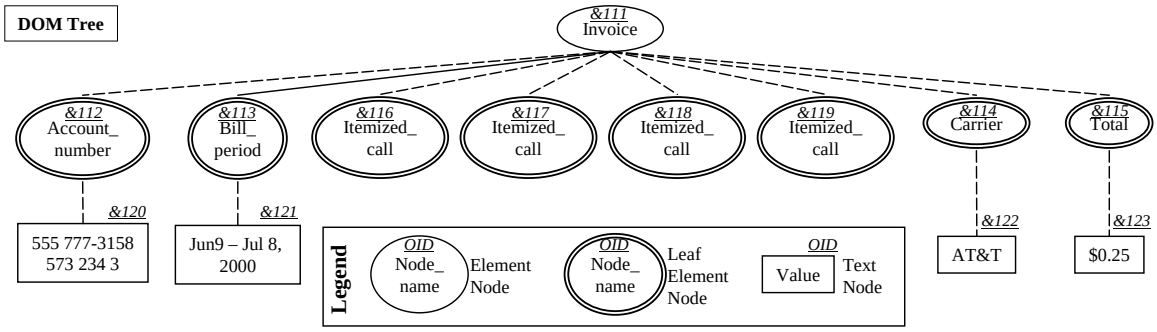


Figure 2.3: DOM Tree for XML in Figure 2.2.

## 2.4 XML Query Language – XQuery

XQuery [53] is designed to query different types of data represented by XML, including structured and semi-structured data, relational databases, and object repositories. XQuery is derived from Quilt [9] which is in turn derived from XPath [52], XQL [41], XML-QL [12], SQL, OQL, Lorel [2] and YATL [8]. XQuery experssions include the following principle forms:

1. Datatypes: XQuery is using the type system of XML Schema.
2. Path expressions: use the abbreviated syntax of XPath. For example:
   /invoice[1]/itemize_call[1].
3. Element constructors: An element constructor consists of a start tag and an end tag. For example:

<mybill> /invoice[1]/itemize_call[1] </mybill>

4. FLWR expressions: A FLWR expression is composed of FOR, LET, WHERE, and RETURN clauses. The variable bindings are also introduced in FLWR expression. For example, we want to get the amount spend to call area code 973 in the telephone bill, the XQuery is:

```
FOR $itemized_call IN /invoice/itemized_call
LET $amount := sum($itemized_call/@amount)
WHERE $itemized_call/@number LIKE '973%'
RETURN $amount
```

5. Expressions involving operators and functions: XQuery can use infix and prefix operators, and allow nested expressions. For example, BEFORE and AFTER infix operators, which are useful to search information by its ordinal position.

6. Conditional expressions: IF THEN ELSE.

7. Quantified expressions: Occasionally it is necessary to test for existence of some element that satisfies a condition. For example, SOME and EVERY quantifiers. They have similar semantics as ANY and ALL in SQL standard.

8. Filtering: Filter returns copies of some of the nodes that satisfied a special condition, while still preserving their nesting relationship and order.

9. Functions: There are core functions provided in XQuery, e.g., avg, sum, count, max, min, distinct, empty, etc.

10. User-defined datatypes: XML Schema also provides the definition facilities to construct user-defined datatypes.

11. Operations on Datatypes: For example, INSTANCEOF.

12. Querying relational data: Relational data can be directly queried by XQuery against a default schema mapping. This feature is used in our default XML mapping that maps relational data into XML data. Please see Section 3.2.1 for more details.

# Chapter 3

# Work Accomplished Thus Far

In this chapter, we review the initial research we have already done towards developing the proposed Rainbow framework, in particular, to address the basic issues of how to do the XML modeling, XML/Relational mapping, and update specification and propagation. We have already publish some works, i.e., Metadata-driven loading technical report [23] and Clock system [56], with collaboration from Verizon Laboratories Incorporated.

## 3.1 DTD Metadata Model

### 3.1.1 DTD Groups

For a detailed definition of DTDs see [4], while below we briefly clarify the meaning of groups in DTDs. A group is defined as a set of elements withint a pair of parenthesis possibly with a repetition operator, such as, ?, *, +. It's equivalent to the *content partial* defined in the XML 1.0 [51]. Each pair of parenthesis without a repetition operator except the outermost one will be treated as a group in our work. We don't consider the simplification of a DTD, such as <!ELEMENT E (a, (b, c))>, which is equivalent to <!ELEMENT E (a, b, c)>, i.e, it has one meaningless group (b, c). For example,

```
<!ELEMENT E (a, b)*> has one group (a, b).
<!ELEMENT E (a, b)> has no group.
<!ELEMENT E (a, (b|c), d)> has the group (b|c).
<!ELEMENT E (a | (b,c) | (d|e)*)> has two groups (b,c) and (d|e).
<!ELEMENT E (a | (b, (c, d)?)+)> has two groups (b, (c, d)?) and (c, d).
```

### 3.1.2 DTD Metadata Tables

We use object-relational technology to implement our mapping tools. Hence as first step, we model the DTD semantics in a relational format as well, referred to DTD metadata

(DTDM) tables. These DTDM tables are Rainbow system tables that store DTD as described next. Below we first introduce the table design and then illustrate them using our running example from Section 2.1.

DTDM has three tables: the DTDM_Item table stores the elements, groups and the text node called PCDATA; the DTDM_Attribute table stores the XML attributes; the DTDM_Nesting table stores the containment relationships between element types and groups.

The DTDM_Item table has the schema: ID, Name and Type. A group's name will be assigned by the system. Types are described in Table 3.1.

| Item Type | Meaning | Handling |
|---|---|---|
| ELEMENT.MIX | <!ELEMENT E (#PCDATA\|a\|b\|...)*> | Item (E) created; Nestings (* [a]) to PCDATA, a, b, ... created. |
| ELEMENT.PCDATA | <!ELEMENT E (#PCDATA)> [b] | Item (E) created; Nesting to PCDATA created. |
| ELEMENT.CHOICE | <!ELEMENT E (a\|b\|...)> | Item (E) created; Nestings to a, b, ... created. |
| ELEMENT.SEQUENCE | <!ELEMENT E (a, b, ...)> [c] | Item (E) created; Nestings to a, b, ... created. |
| ELEMENT.EMPTY | <!ELEMENT E EMPTY> | Item (E) created. |
| ELEMENT.ANY | <!ELEMENT E ANY> | Item (E) created. Nestings (*) to all the element types and PCDATA in this content (of the DTDM tables/views)[d]. |
| GROUP.CHOICE | (a\|b\|...) | Item (G) created. Nestings to a, b, ... created. |
| GROUP.SEQUENCE | (a, b, ...) [e] | Item (G) created. Nestings to a, b, ... created. |
| PCDATA | #PCDATA. Denotes a text node. | Item (PCDATA) created. Attribute (value[f]) created. |

Table 3.1: Types of Items

[a] We use * repetition operator because (a|b)* = (a*|b*)*.

[b] It's a special type of ELEMENT.MIX. Due to its popularity, we make it a separate type.

[c] If there is only one sub-element in the declaration, like <!ELEMENT E (a)>, we treat that as ELEMENT.SEQUENCE as defined in XML 1.0.

[d] ELEMENT.ANY can be treated as a special case of ELEMENT.MIX, which has relationships to all the possible element types.

[e] If there is only one sub-element in the group, e.g, (a), then the group type is GROUP.SEQUENCE.

[f] *Value* attribute has type CDATA and default #IMPLIED.

The DTDM_Attribute table has the schema: ID, PID, Name, Type, and Default_value. Type could be: StringType (CDATA), TokenizeType(ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS), and EnumerationType(NOTATION, <enumeration>). The default_value will be the default value for #FIXED, #REQUIRED, or #IMPLIED. We primarily focus on the logical structure of the attribute type, which is CDATA, ID, IDREF, IDREFS, and <enumeration>. The text node has one pre-defined attribute called "value" with type "PCDATA" in the DTDM_Attribute table.

The DTDM_Nesting table has the schema: ID, parentid, childid, multiple, optional, and position. Multiple and optional columns are used to represent different repetition operators, i.e., 1[1], +, *, ?, as depicted in Table 3.3. The position will record where this

[1] '1' represents the fact that no repetition operator exists. This means that it is required exactly once.

| characteristics | data type [a] | category |
|---|---|---|
| single-value | **CDATA**<br>NMTOKEN<br>ENTITY | |
| | **ID**<br>**IDREF** | reference |
| multi-value | **IDREFS** | |
| | ENTITIES<br>NMTOKENS<br>NOTATION<br>**<enumeration>** | |

Table 3.2: Possible Type of Attributes

[a] The types with bold font defines logical structure of a XML.

| Repetition Operator | Multiple | Optional |
|---|---|---|
| 1 (no repetition operator) | false | false |
| ? | false | true |
| + | true | false |
| * | true | true |

Table 3.3: Mapping between Occurrence Property and the Ratio and Optional Fields

relationship takes place (index starting from 0) in the parent element. For example, given <!ELEMENT E (a,b)>, for the nesting relationship $E \rightarrow a$ we record position 0; and for the nesting relationship $E \rightarrow b$, we record position 1. The position also applies in the same manner for the choice typed element content or group. For example, given <!ELEMENT E (a|b)>, then for $E \rightarrow a$, we have position 0, and for $E \rightarrow b$, we have position 1. The sibling relationships between the items are also captured in the DTDM_Nesting table by the fact that items have the same parent item.

In Figure 3.1 we depict the three DTDM tables for our running example from Figure 2.1. For example, the 'itemized_calls' item is stored as a tuple with ID equals 6 in the DTDM_Item table. Its attribute 'date' is stored as tuple with ID = 2 in the DTD_Attribute table in Figure 3.1. The nesting relationship between the 'invoice' item and the subitems 'itemized_calls' is stored as tuples with ID 4 in the DTDM_Nesting table. Also, we can see that the items 'account_number', 'bill_period', 'carrier', 'itemized_calls', and 'total' are all siblings.

The extension to support multiple DTDs is also straightforward. This can be achieved by, for example, adding a DTD catalog table to assign an ID for each DTD, and adding a DTD_id column to all the DTDM tables. It is omitted here for simplicity.

### 3.1.3 DTD Graph Model

To graphically illustrate the DTD metadata, we have proposed a DTD graph model. Figure 3.2 depicts the graphical notation to represent the DTD. DTD graph is an ordered graph that is composed of items, attributes, and nesting relationships. The items are represented a circle, the attributes are represented by a triangle, and nestings are represented by links between the items. The repetition operators are shown as labels on the edges. There are

**DTDM-Item**

| ID | Name | Type |
|---|---|---|
| 1 | PCDATA | PCDATA |
| 2 | invoice | ELEMENT.ELEMENT |
| 3 | account_number | ELEMENT.PCDATA |
| 4 | bill_period | ELEMENT.PCDATA |
| 5 | carrier | ELEMENT.PCDATA |
| 6 | itemized_calls | ELEMENT.EMPTY |
| 7 | total | ELEMENT.PCDATA |

**DTDM-Nesting**

| ID | FromID | ToID | Ratio | Optional | Index |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 1:1 | false | 1 |
| 2 | 2 | 4 | 1:1 | false | 2 |
| 3 | 2 | 5 | 1:1 | false | 3 |
| 4 | 2 | 6 | 1:n | true | 4 |
| 5 | 2 | 7 | 1:1 | false | 5 |
| 6 | 3 | 1 | 1:1 | false | 1 |
| 7 | 4 | 1 | 1:1 | false | 1 |
| 8 | 5 | 1 | 1:1 | false | 1 |
| 9 | 7 | 1 | 1:1 | false | 0 |

**DTDM-Attribute**

| ID | PID | Name | Type | Default |
|---|---|---|---|---|
| 1 | 6 | no | ID | #REQUIRED |
| 2 | 6 | date | CDATA | #REQUIRED |
| 3 | 6 | number_called | CDATA | #REQUIRED |
| 4 | 6 | time | CDATA | #REQUIRED |
| 5 | 6 | rate | (NIGHT\|DAY) | #REQUIRED |
| 6 | 6 | min | CDATA | #REQUIRED |
| 7 | 6 | amount | CDATA | #REQUIRED |
| 8 | 1 | value | CDATA | #IMPLIED |

Figure 3.1: DTDM Tables of Billing DTD from Figure 2.1.

three kinds of items: 1) sequence item, whose children are a list of items that separated by a comma in DTD syntax, 2) choice item, whose children are choices that separated by "|" in DTD syntax, 3) item reference, which denotes a reference to an item that has already been defined somewhere else. Figure 3.3 shows the DTD graph of the example DTD in Figure 2.1.

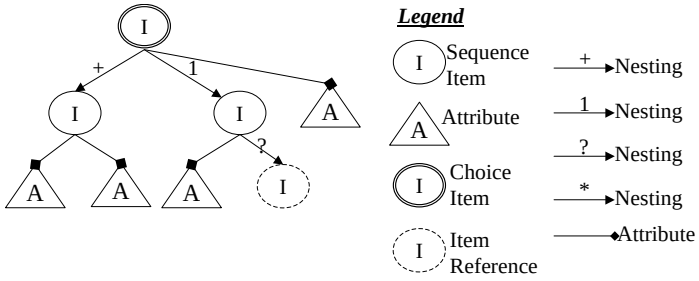


Figure 3.2: Notations of DTDM Graph.

## 3.2 Default Relational View and Default XML View

We need a cross-model mapping to bring the data across from the XML data model to the relational data model and vice versa. For this purpose, we define default relational view and default XML view (as depicted in Figure 1.4) for XML to relational data model mapping and relational to XML data model mapping, respectively.

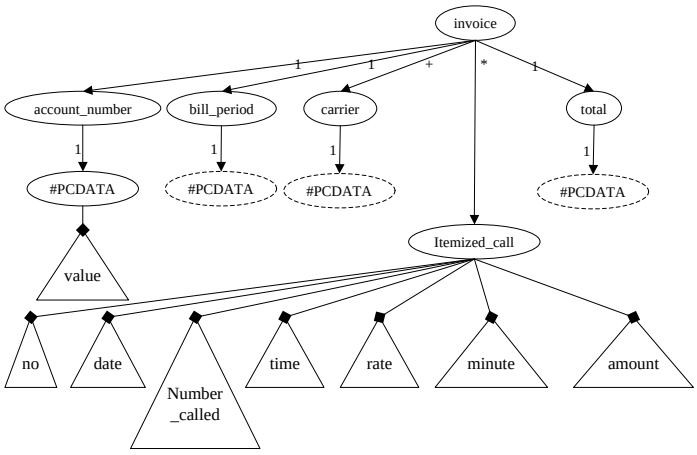


Figure 3.3: DTD Graph of DTD Example.

### 3.2.1 Default XML View

The default XML view concept has been proposed in XQuery [53]. For a given relation with schema $R(A_1, A_2, ..., A_n)$, a default XML view is very simple to generate as depicted in Figure 3.4. All the relational constraints, e.g., key constraints, foreign key constraints, data types, etc., are not captured by such mapping.

<$R$>
  <$R$_tuple>
    <$A_1$>(value of $A_1$)</$A_1$>
    <$A_2$>(value of $A_2$)</$A_2$>
    ...
    <$A_n$>(value of $A_n$)</$A_n$>
  </$R$_tuple>
  <$R$_tuple>
    <$A_1$>(value of $A_1$)</$A_1$>
    <$A_2$>(value of $A_2$)</$A_2$>
    ...
    <$A_n$>(value of $A_n$)</$A_n$>
  </$R$_tuple>
  ...
</$R$>

Figure 3.4: Default XML View for Relation with Schema $R(A_1, A_2, ..., A_n)$

### 3.2.2 Default Relational View

A default relational view is used to bring any XML documents into the relational data model, so that further restructuring can be performed on top of this xml data using relational technology. We give two guidelines to generate the default relational view upon any XML documents with knowledge of its schema defined in a given DTD.

**Item Mapping:** *For each ELEMENT and PCDATA typed item defined in the DTDM-Item table, create an application table named item.Name. The table has three default columns:* **iid**, **pid**, **position**. **iid** *represents an internal unique ID that will be generated when the XML data is loaded.* **pid** *represents the* iid *of its parent item and* **position** *represents the local order among sibling item instances.*

The **iid** and **pid** columns are used to capture the hierarchical information of XML documents, and the **position** column is used to capture the order information captured by XML documents. For example, by following the **Item Mapping** guideline, we create an empty *itemized_call* table from the tuple with $id = 6$ in the *DTDM_item* table of Figure 3.1.

**Attribute Mapping:** *For each tuple* t *in the DTDM-Attribute table, create a column named* t.Name *of type string in the relational table identified by* t.pid[2].

For example, the columns of table *itemized_call* in Figure 3.5 are deduced from the records defined by $PID = 6$ in the *DTDM_Attribute* table (Figure 3.1).

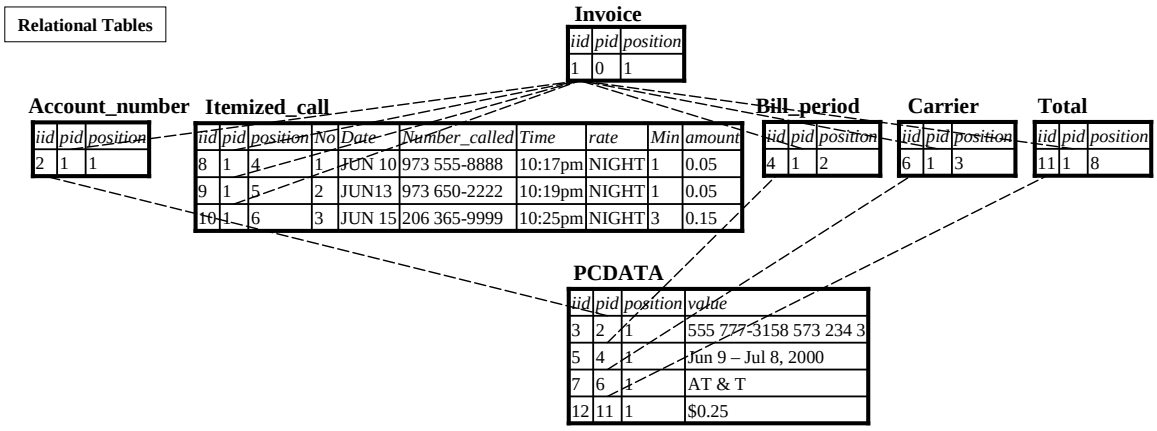


Figure 3.5: DOM Tree and Relational Tables Imported from XML in Example 2.1.

Once the application tables for this particular DTD have been created, XML documents can be loaded into these tables. The XML importer (in Figure 1.2) traverses a DOM tree uses the metadata mapping described above to store the XML data into relational tables.

[2]Further parsing on the value to decide its data type is the future work.

The Figure 3.5 shows the data loaded into the corresponding relational tables by traversing the DOM tree in Figure 2.3. The dashed line shows the hierarchical relationships between tuples.

### 3.2.3 Materialization of Flexible Mapping

In summary, we propose to achieve a flexible mapping by combining some default fixed cross-model mapping with some restructuring. The fixed default mapping is a default relational view whereas the restructuring is performed within the relational paradigm hence using SQL queries. We propose to apply materialized view technology for both query processing and automatic maintenance. Our goal will be to determine if traditional view technology is sufficient or if any programming extension or new query algebra operator are needed. An analysis of the tradeoff between the query performance and data redundency is required.

The fixed default mapping could be a virtual default relational view of the external XML documents, which does not necessarily need to be materialized. Once a flexible mapping can be represented by SQL queries over the default relational view, the optimized physical schema could be created directly without materializing the default relational view.

## 3.3 XML Data Update Synchronization

The default view manager in Rainbow framework in Figure 1.2 keeps internal relational data synchronized with external XML data. We have identified a complete set of data update primitives, i.e., *createLeafElement*, *deleteLeafElement*, *moveElement*, and *modifyElement*, as listed in Table 3.4. We have also designed update propagation and validation algorithms for the primitives. Our experiments confirm that the create element is the cheapest operation among those four primitives. Also, our experiments confirm that update synchronization is typically faster than performing a complete reload of modified XML documents. For details please refer to [56]. In the following sections we briefly review the update primitives and validation algorithms.

### 3.3.1 Propagation of XML Updates

We now show how to map the four primitive updates illustrated in Table 3.4 into operations on their relational counterpart by explicitly exploiting knowledge in the DTDM tables.

| Update | Interface |
|---|---|
| Create Leaf Element | CreateLeafElement(node_name, list_of_attributes) |
| Modify Element Operation | ModifyElement(iid, node_name, attribute_name, new_value) |
| Delete Leaf Element Operation | DeleteLeafElement(iid, node_name) |
| Move Element Operation | MoveElement(iid, iid_node_name, pid, pid_node_name, new_position) |

Table 3.4: Four XML Data Update Primitives with their Interfaces.

Definition 1 defines the sibling tuples and tables in the relational database of Rainbow system. They are used in the description of each data update primitives.

**Definition 1 Sibling Tuples of tuple `t`:** *The tuples corresponding to sibling nodes of the node corresponding to the tuple* `t` *are called* **sibling tuples of tuple `t`**.

**Sibling Tables of tuple `t`:** *The tables containing the sibling tuples of tuple* `t` *are called the* **sibling tables of tuple `t`.**

**Create Leaf Element Operation** creates a new leaf element and returns its new *iid*. This element will not be connected to any existing element yet. The connection could be created later by the *MoveElement* primitive [3].

On the relational model side, the *CreateLeafElement* operation will add one tuple to the table identified by *node_name* with the applicable attributes that can be gotten from the DTDM tables, and an empty due to the reason of no parent element yet. *pid* and *position*. A unique *new_iid* is generated by the system for the new tuple. The SQL template generated for the update is:

```
INSERT INTO <node_type> (iid, pid, position,
                         <list_of_attributes>)
VALUES (<new_iid>, null, null,
        <values_of_list_of_attributes>)
```

**Modify Element Operation** updates the attribute specified by *attribute_name* of the element identified by *iid* of type *node_name* with the new value *new_value*. The SQL template is:

```
UPDATE <node_type> SET <attribute_name>=<new_value>
WHERE iid = <iid>
```

**Delete Leaf Element Operation** deletes the leaf element identified by the *iid* of element type *node_name*. The *DeleteLeafElement* first gets a list of the sibling tables of the current element by querying the DTDM tables. Then, it goes through the sibling tables to decrease

[3] An insertion of a leaf element can be represented by the combination of *CreateLeafElement* and *MoveElement*.

the positions of the sibling tuples to the right of the to-be-deleted tuple. Third, we delete the to-be-deleted tuple identified by the *iid* from the table *node_name*. At last, if the to-be-deleted tuple is of type ELEMENT.PCDATA as noted in the *DTDM_item* table, we also delete the child tuple of the to-be-deleted tuple from the PCDATA table.

**Move Element Operation** moves the element identified by the *iid* of element type *iid_node_name* as the child of the element identified by the *pid* of element type *pid_node_name* into the position *new_position*.

The complexity of the relationship between old and new positions of an element complicates this operation. There are three kinds of relationships between the two positions. If the element is moved between different parent elements (either located in one table or two different tables), then increase the position of the tuples in the sibling tables that are larger than the *to* position, and decrease the position of the tuples in the sibling tables that are larger than the *from* position.

If the element is moved within the same parent, there are two cases. First, if the *from* position is less than the *to* position, then we decrease positions of the sibling tuples with the position greater than the *from* position but less than the *to* position. Second, if the *from* position is larger than the *to* position, we increase the positions of sibling tuples with the position greater than the *to* position but less than the *from* position.

Finally, we update the *pid* and position of the moved tuples.

### 3.3.2 General Update Propagation Process

We can see that the DOM operations are object-oriented operations, in the sense that every node is identified by its OID. We cannot guarantee that the OID used by the external XML data source will be the same as the internal IID used in the relational storage. To assure identification of the desired elements, such identification of items in DOM could be achieved by assuming that the XML data source uses the XPath to uniquely identify the to be modified node. For example, the XPath of the node with OID &115 in Figure 3.5 is "/invoice[1]/itemized_call[1]".

However, elements mapped into relational databases are identified by the pair *node_name* and *iid*, where *node_name* leads to the relational table in which data with the *iid* is to be found. The *iid* can be easily computed from XPath information by an XPath-to-iid index. The *node_name* can be gotten by parsing the XPath. For example, "/invoice[1]/itemized_call[1]" will be translated into "iid=8" and "node_name=itemized_call" (Figure 3.5).

Hence, it will be translated into the data update primitives: *DeleteLeafElement(8, "itemized_call")* into the DOM update *Node.removeChild*() for node "/invoice[1]" deleting the first child node "/invoice[1]/itemized_call[1]". In general the *synchronizer* first finds the correct

table name and update statement, and then the synchronizer will propagate those updates into the relational storage.

### 3.3.3 Validation of XML Updates

An XML document is said to be valid if it is compliant to a specific DTD. Though most of current available XML parsers can validate the whole XML document, they do not validate an individual XML update. If updates are specified without any kind of validation checking incrementally, the data would then be in a non-valid state according to its DTD. Hence, *synchronizer* will incrementally validate the update based on the DTD (i.e., the metadata captured in the DTDM tables) before executing the update operation. We support three kinds of update validations:

- *AttributeCheck(node_name, attribute_name, new_value)* will check whether the new value of the attribute satisfies the specification of that attribute. For example, it will also check the uniqueness of the ID/IDREF typed attribute using the DTDM_attribute table. This check will be used for the validation of *CreateLeafElement* and *ModifyElement*.

- *NestingCheck(node_name, iid)* will check the quantifier of the nesting relationship between the element identified by the *node_name* and *iid* with its parent element using the DTDM_nesting table. This will be used for the validation of *DeleteLeafElement*.

- *NestingCheck(from_node_name, from_iid, to_node_name, to_pid)* will check the quantifier of the nesting relationship between the element identified by *from_node_name* and *from_iid* with its parent and also the nesting relationship with the element identified by *to_node_name* and *to_pid* using the DTDM_nesting table. This will be used for the validation of *MoveElement*.

### 3.3.4 Experimental Study

We identify the update sychronization part of the Rainbow system as *Clock system*. We have implemented the Clock system and have conducted an experiment study. The experimental result is published in [56].

### 3.3.5 Next Tasks to be Done

Next we have two extensions to it: 1) binding our data update primitives with XQuery, and 2) integrated update propagationwith flexible mapping mechanism.

**XQuery Extension.** So far, XQuery hasn't defined any syntax for updates. The Microsoft's XML Query Language demo [30] has proposed the extended XQuery grammar to contain some update syntax. [48] also defines some syntax to specify XML updates using XML-QL and Quilt.

For our purpose, we extend the XQuery by four XML data update primitives. The grammar of the update portion is defined in Figure 3.6, where the grammar is from [53]. Here we revise the *returnClause* grammar to put the update statements. We introduced in Section 2.4, the XQuery already has the construction capability, hence we don't have additional syntax for *createElement* update.

```
returnClause := RETURN valueExpression |
                updateStatement;

updateStatement := deleteStatement |
                   modifyStatement |
                   moveStatement;

// VAR binds to elements.
deleteStatement := DELETE VAR;

// VAR binds to attributes.
modifyStatement := MODIFY VAR WITH
                   ArithmeticExpression;

// VAR binds to elements.
// PathExpression identify only one Element.
moveStatement := MOVE VAR BEFORE PathExpression |
                 MOVE VAR AFTER PathExperssion;
```

Figure 3.6: Extended XQuery Grammer for Update Statements.

For example, we want to update the carrier from "AT&T" to Sprint in the billing XML document from Figure 2.2. The syntax is like:

```
FOR $carrier IN /invoice/carrier
WHERE $carrier = 'AT&T'
MODIFY $carrier WITH 'Sprint'
```

**Integration with Flexible Mapping.** This extension is already in process. Once the research of restructuring techniques has been accomplished, we can start to integrate the synchronization with flexible mapping. We expected to see the integration to be relatively

straightforward while assuming update propagation through the relational views is supported.

# Chapter 4

# Proposed Work

In order to achieve schema optimization for a given query workload, we propose to tackle the following tasks:

1. Design XML query translation and execution using an SQL engine.
2. Define XML query cost model by utilizing SQL.
3. Develop a schema optimizer that does conceptual/physical database design based on an XML query workload.
4. Develop a set of restructuring operators for denormalization, including their management and execution.

In the rest of this chapter, we discuss each task in detail, propose a some initial solution strategy of how to approach each task, outline the outstanding research issues, and our plans of evaluation.

## 4.1 XML Query Translation

This proposal will use the W3C's XML query standard XQuery [53] as our query language. Shanmugasundaram et al. [5, 45] proposed one strategy for XQuery translation. First, they define an XML document view on top of the relations that store the shredded XML data from the XML documents. Second, the XML document view will be composed together with a user's XQuery into one composite XQuery, which has relations at its leaves. Then, an algebra tree composed of XML algebra and relational nodes will be translated into tagger operators that put tags around the relational data and generate XML documents out of it. Furthermore, by some rewriting rules (which have yet to be identified), the memory

and space intensive computations in the tagger operators will be pushed down into the relational databases as much as possible. Besides that, Manolescu et al. [25] discuss the techniques to push the XQuery into SQLs by directly translating the XQuery.

Inspired from the literature, we propose our own XQuery translation techniques based on the refined XML algebra model ([5, 45]).

- Once a fixed relational mapping has been generated for a given DTD and XML, we will generate the XML forest view (in XQuery) on top of the shredded data based on our fixed mapping.
- Translate the user's XQuery into an XML Algebra Tree (XAT), which explicitly defines the operators and the variable bindings.
- Translate the XML forest view into XATs.
- Compose the XATs for the user's XQuery and for the XML forest view into one XAT.
- Use equivalence rules to push down relational operators to the bottom of the XAT. So far, [5, 45] hasn't standardize those equivalence rules. A set of such rules has yet to be developed.
- Generate SQLs statements from the XAT, and execute the SQL statements using the relational query engine. As the result, one XQuery usually will typically be converted into multiple SQL queries.
- Tag the resulting relational data into XML forest in order to have a final XML document as output.

### 4.1.1 XML Query Algebra

Traditional relational algebra has a set of well defined operators [39], such as literal, relation, selection, projection, set-operations, renaming, cross product, division, sort, and aggregate operations.

W3C [XQA] haven't identified the final operators for the XQA. So far, they have projection, atomic data, selection, quantification, join, un-order, iteration, sort, aggregation, and functions. The first attempt on XQuery's operators has been made in [5, 45] with the goal to generate the XQGM (XML Query Graph Model).

In order to do the query translation, we propose our own query operators for XQuery execution. We define XAT (XML Algebra Tree) inspired by the idea from XQGM proposed by [5, 45].

**XAT** is composed of *nodes* and *edges*. Each *node* contains one operator and variable bindings for that operator. If the operator has no explicit variable binding, then an implicit variable will be inferred from the semantics of the operator. Each *Edges* will show the data flow connection between nodes. The data passed between operator nodes are a list of tuples. Each value in the tuple can be a list, an XML fragment, or an atomic value. Each variable will bind to one column in the list.

We have following guidelines when we pick XAT operators. They must be 1) powerful enough to capture the full generality of XQA or XQuery, and 2) easy to be converted into the SQL algebra, and thus translatable into SQLs. So far, we consider the following operators in XAT:

- **SQL**: A SQL operator where one SQL query is stored in the node. It will generate a list of tuples that are composed of atomic values. The most simplest node of type SQL would be "select * from < *a_table* >" to allow the processor to access a relation.
- **Projection**: Project out some variable bindings in the input list, or rename the variables.
- **Selection**: Filter the list by some predicates.
- **Join**: Join tuples from multiple lists. It will take multiple lists and generate cross product on the tuples of those lists, and finally generate a new list where each tuple is also a list. If a join condition is provided, the output list will be filtered by the join condition.

  For example, joining a list of two tuples [a1, a2] with a list of three tuples [b1, b2, b3] will generate a list of six tuples, and each tuple is also a list: [[a1, b1], [a1, b2], [a1, b3], [a2, b1], [a2, b2], [a2, b3]].
- **GroupBy**: Group tuples in a list by a key, and apply aggregation functions.
- **OrderBy**: Sort a list based on a key. This key has to be able to be sorted.
- **Set operators**: Given two lists as input, it performs set union, intersection, and difference based on value matching.
- **Navigation**: Perform path navigation in XPath expression on each tuple in the list and generate a list that each tuple will be the nodes reached by the XPath expression.
- **Tagger**: Put tags around tuples in the input list, corresponding to RETURN clause in XQuery. It will also return a list.

- **Unpack**: Iterate through the tuples in the list, and bind the output variable with each tuple in the list. At the end of list, an end-of-list signal is generated. It will split one input list into a sequence of output lists, each ended by an end-of-list signal.
- **Pack**: Retrieve a stream of multiple input lists until end-of-list signal is received, and treat each input list as a tuple in the output list. It will merge a sequence of lists followed by an end-of-list signal into one list.
- **Function**: Apply a function to variables and generate other variables. The function could be: *atomic data*, *quantification*, *parent and cast operators*, *references and node identity*, *aggregation function*, *view*, etc.

All the operators will have one or more input lists, except the SQL operator, and they are all order sensitive. Similar to the *nested relational model*, we need an iteration operator, which is Unpack in our case, to handle data structures like a list of lists. Further research is required to refine those operators. And, an algorithm is required to convert an XQuery statement into its XAT. The XAT is a super set of the relational algebra; hence we can use XAT to explain SQL statements also.

A user query expressed in XQuery syntax is specified in Figure 4.1. The user wants to know about calls which start with number "973". The XAT for this query is displayed in Figure 4.2.

```
FOR $itemized_call_user
    IN view("invoice")/itemized_call
WHERE $itemized_call_user/@number_called
      LIKE "973%"
RETURN $itemized_call_user
```

Figure 4.1: User Query for Phone Billing.

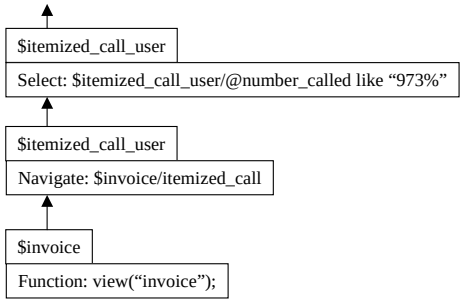


Figure 4.2: XAT for User-defined Query.

There are couple of issues need to solved in this situation. First, we have to identify a complete set of well-defined rewriting rules for XAT translation. Second, we need an algorithm to apply those rules. Third, if there are more than one translation is possible, we need to identify the criteria to pick the proper one for our schema optimization.

### 4.1.2 XQuery Transformation Rules

Once we can successfully generate the XML Algebra Tree (XAT) for a given XQuery, an XQuery will be translated into SQLs by merging projection, selection, join, groupby, or-

derby, and set operators with SQL operators. This will be achieved by a list of equivalent composition rules.

We will go through our running example to demonstrate the idea of translating XAT operators into one or more SQL operators. Recall the sample billing XML document shown in Figure 2.2. We have shredded the XML document and stored it in relations in Figure 3.5 by the default relational mapping. The XML forest view (in Figure 4.3) is created to map the shredded XML document back. We assume the XML forest view is created based on the default relational mapping and the DTD in Figure 2.1 to exactly regenerate the XML document in Figure 2.2.

```
CREATE VIEW invoice AS (
   <invoice>
     <account_number>
        FOR $PCDATA IN view("default")/PCDATA/row,
            $account_number in view("default")/account_number/row
        WHERE $PCDATA/pid = $account_number/iid
        RETURN
          $PCDATA/value
     </account_number>
     <bill_period>
        FOR $PCDATA IN view("default")/PCDATA/row,
            $bill_period in view("default")/bill_period/row
        WHERE $PCDATA/pid = $bill_period/iid
        RETURN
          $PCDATA/value
     </bill_period>
     (
       FOR $carrier IN view ("default")/carrier/row,
           $PCDATA IN view("default")/PCDATA/row,
       WHERE $carrier/iid = $PCDATA/pid
       ORDER BY $carrier/position
       RETURN
         <carrier>$PCDATA/value</carrier>
     )
     (
       FOR $itemized_call IN view ("default")/itemized_call/row,
       ORDER BY $itemized_call/position
       RETURN
         <itemized_call no="$itemized_call/no" date="$itemized_call/date"
                        number_called="$itemized_call/number_called"
                        time = "$itemized_call/time"
                        rate="$itemized_call/rate" min="$itemized_call/min"
                        amount = "$itemized_call/amount"/>
     )
     <total>
        FOR $PCDATA IN view("default")/PCDATA/row,
            $total in view("default")/total/row
        WHERE $PCDATA/pid = $total/iid
        RETURN
          $PCDATA/value
     </total>
   </invoice>
)
```

Figure 4.3: XML Forest View for Phone Billing in Figure2.2.

Figure 4.4 depicts the XAT for the XML fragment view in Figure 4.3 with two subqueries illustrated in Figures 4.5 and 4.6. Figure 4.2 depicts the XAT for the user query in Figure 4.1.

Then, we combine the user query and the XML forest query, and then optimize the composite query (see Appendix A for intermediate step of some possible optimization of this example query). After that the final optimized XAT for the composite query is described in Figure 4.7. The final XAT in Figure 4.7 is composed of one tag operator and

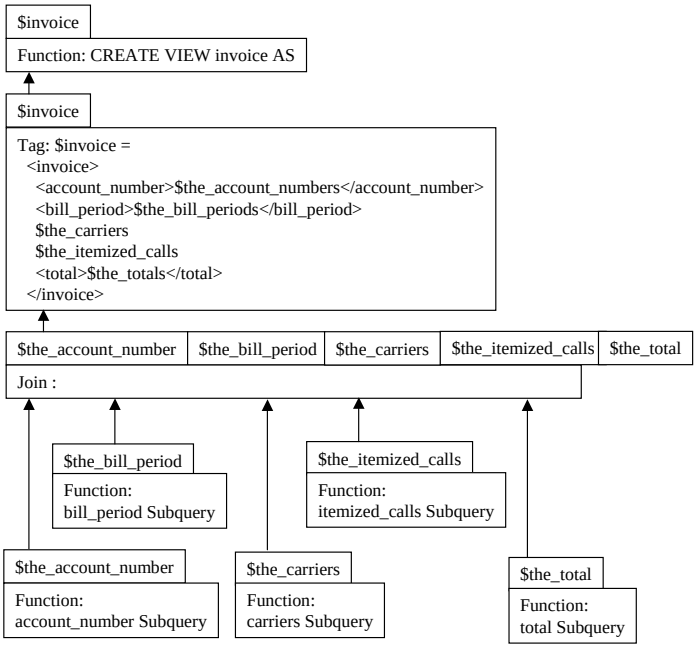


Figure 4.4: XAT for XML Fragment View.

one single SQL operator. The query result will contain all the itemized calls that start with "973". The steps to transition from XQuery to SQLs are demonstrated in Appendix A.

We propose the following guidelines for the SQL translation include:

- We generate SQL operators by merging XAT operators.
- If there is any Tag operator block in the middle, the disconnected SQL operators cannot be further merged.
- Tag operators should be pulled up as much as possible, and pure SQL operators should be pushed down as far as possible.

## 4.2 XQuery Workload

The underlying relational database used as storage medium for the Rainbow system needs to be tuned based on the given SQL workload and schema. A workload [39] is typically composed of a list of queries and updates and their related frequencies, and optional performance goals for each type of query and update.

We propose the specification of an XQuery-SQL workload both for design and tuning. Our workload will be composed of workload elements with importance and frequency attributes. The workload elements are organized by XQuery and SQL categories. Because

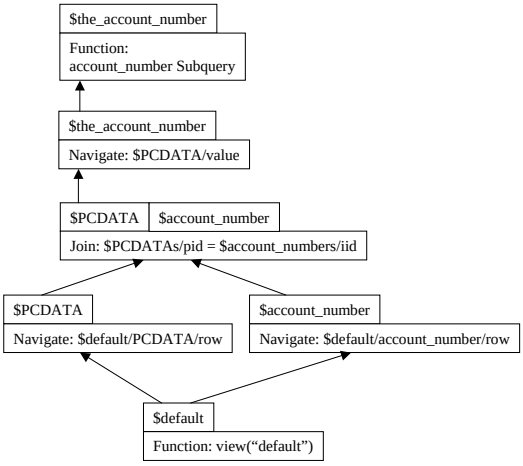


Figure 4.5: Account Number Subquery.

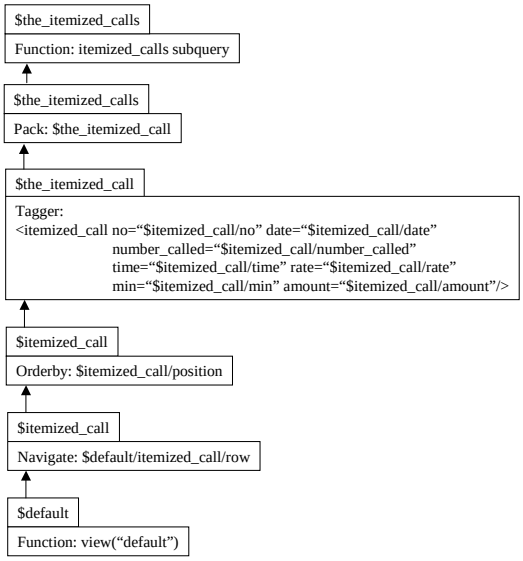


Figure 4.6: Itemized Call Subquery.

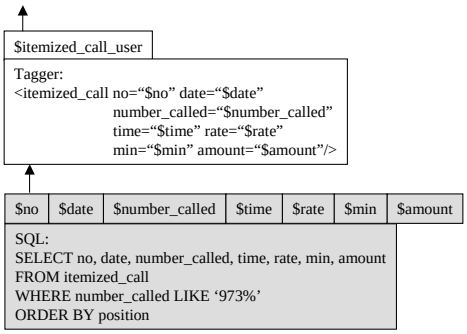


Figure 4.7: Final XAT for Composit Query.

we want to keep the relationship between the XQuery workload given by DBA and SQL workload generated by our system, we include both the XQuery and its translated SQLs in the workload. The importance attribute will be directly specified by the DBA to the initial XQuery workload, and the frequency value will be collected during the real execution for further tune up. In our proposal, we will only consider the importance attribute, and leave the database maintenance tune up to the future work. The workload specification also contains the "workload style" that specifies the weight of the importance value and the frequency value.

The workload for the XAT in Figure 4.7 is depicted in Figure 4.9 based on the DTD described in Figure 4.8. We can see that, each SQL workload will be bound to the XQuery workload. However, in this example we only have one query. The SQL workload will be generated from the XQuery workload during the XML query translation process. Basically,

```
<!ELEMENT workload (style, xquery+)>
<!ELEMENT style EMPTY>
  <!ATTLIST style
            importance_weight CDATA #REQUIRED
            frequency_weight CDATA #REQUIRED>
<!ELEMENT xquery (element, sql?)>
<!ELEMENT sql (element*)>
<!ELEMENT element (#PCDATA)>
<!ATTLIST element
          importance CDATA #REQUIRED
          frequency CDATA "0">
```

Figure 4.8: DTD of XQuery Workload.

the SQL queries will have the same important factor as their parent XQuery statement, but for nested XQuery statement or joins, the frequency values of SQL queries will be changed. We propose to solve the SQL query workload generation issue in our dissertation.

```
<workload>
  <style importantce_weight=\"5\" frequency_weight=\"5\"/>
  <xquery>
    <element importance_weight=\"1\">
      THE XQUERY.
    </element>
    <sql>
      <element important=\"1\">
        SELECT no, date, number_called, time, rate, min, amount
        FROM itemized_call_973
      </element>
    </sql>
  </xquery>
</workload>
```

Figure 4.9: An Example of XQuery Workload

## 4.3 Cost Model

In order to auto-tune the conceptual schema, the system needs some means to select among different query plans over different schemata. For this purpose, we design a cost model to estimate the cost of a XML query plan to be executed on top of a given conceptual (and physical) schema. The traditional method [39] of identifying the cost for a given query plan is based on the cost of each operator. In order to do that, we have to know:

- Cardinality: The number of tuples for each input list. Because the list could contain lists, the number of tuples will be the number of leaf tuples.
- Selectivities for join and predicates.
- Cardinality of the intermediate results and the final result and how each of them is sorted.

We assume there is a way those parameters are given to use in order to do the cost estimation for a given XAT. The gathering of those parameters is not our focus. We will first mainly make rough estimate using sizes of data and not precise disk layout and hence number of I/Os.

Though our cost model can calculate the total cost of an XAT during the conceptual schema optimization, we will only consider the cost of the SQL operators. Due to the reason that the relational conceptual schema will only affect the performance of evaluating the SQL operators, and the upper layer XML operators will be evaluated outside of the relational database.

One exception is the order of a list generated by SQL operators. A different order generated by the SQL operator will have a different cost in the higher level XML operator evaluation. This problem raised by different orders can be solved by forcing an *orderBy* operator right above the SQL operator.

Assume the cardinality of table $itemized_call$ is 5000 and the column *number*'s selectivity is 1/10. We can easily estimate the cost for the SQL operator in Figure 4.7 as 5000(scan) + 500(select) = 5500. This part of the cost estimation uses existing relational techniques.

## 4.4 Query Optimization

Query optimization and schema optimization are correlated. Query optimization includes two parts, namely: query plan optimization, e.g., pushup/down operators, and conceptual operator to physical operator binding, e.g., index selection and clustering.

The XML query plan optimization may change the XAT. Hence it increases the complexity of the schema optimization task. In our proposal, when we do the schema optimization we assume that the XAT has already been optimized and now it is static for our purpose. We will not consider the physical operator binding of the optimized XAT in this proposal.

There are couple of research issues in this part. First, order is a very important feature in the XML query evaluation, we have to figure out that how the order in the output list generated by the SQL operators will affect the overall cost of the XAT. Second, it's very

hard to say that an XAT with less but complex SQL operators will be always executed faster than an XAT with more but simple SQL operators. A further investigation is required.

## 4.5 Schema Optimization Techniques

Before we talk about the schema optimization, we first need to clarify the visibility of the different layers in the optimization. We have four layers in the Rainbow system defined in Section 1.3: XML external layer, XML conceptual layer, XML physical layer and relational layer. The relational layer contains its own conceptual schema layer and physical schema layer. The optimization only concentrates on the conceptual layer of the relational layer.

### 4.5.1 Relational Conceptual Schema Optimization

The conceptual schema optimization based on a given XQuery workload will not consider any indexing and clustering, which should be handled by physical schema optimization. Conceptual schema tuning includes restructuring and denormalization by materialized views, namely to denormalized data in the relational data model to assure better query performance.

Mullins [31] from Platinum Technology Inc. and Root [42] from Thisledown Consulting Services present a comprehensive discussion on different denormalization approaches. We summarize the techniques in Table 4.1. Sanders et. al. [43] also propose four kinds of denormalization strategies, e.g., 1) collapsing tables (one-to-one, many-to-many), 2) splitting tables (horizontal/vertical splitting), 3) adding redundant columns (reference data), and 4) derived attributes (e.g, summary, total, and balance). Most of the denormalization strategies in Table 4.1 can be implemented by SQLs except for *speedtables*. The *speedtables* can only be implemented by a programming language, e.g., COBOL, C, and Java, combined with SQLs.

We will contrate on the denormalization techniques that can help the hierarchical and order-sensitive XML queries. For example, the *pre-joined tables*, *report tables* and *speed tables* are of interest of XML data.

Materialized view technologies [24, 32, 46, 54, 1] are also used to improve query performance.

To demonstrate the performance improvement by materialized views, let's recall the SQL operator in Figure 4.7. We can split *itemize_call* table by their telephone area code to achieve a better performance. We can use materialized view technology to do the spliting as illustrated in Figure 4.10

| Techniques from [31] | or [42] | Description |
|---|---|---|
| Pre-Joined Tables | stored joins | used when the cost of joining is prohibitive. |
| Report Tables | | used when specialized critical reports (with **order**) are needed. |
| Mirror Tables | duplicated data | used when tables are required concurrently by two different types of environments. |
| Split Tables | vertical or horizontal segmentation | used when distinct groups use different parts of a table. |
| Combined Tables | | used when one-to-one relationships exist. |
| Redundant Data | | used to reduce the number of table joins required. |
| Repeating Groups | recuring data groups | used to reduce I/O and (possible) DASD. |
| Derivable Data | derived data | used to eliminate calculations and algorithms. |
| Speed Tables | | used to support **hierarchies**. |
| | surrogate keys | used to optimize the indices on the primary key. |

Table 4.1: Types of Denormalization from [31].

```
CREATE MATERIALIZED
VIEW itemized_call_973 AS
SELECT *
FROM itemized_call
WHERE number LIKE '973%'


CREATE MATERIALIZED
VIEW itemized_call_others AS
SELECT *
FROM itemized_call
WHERE number NOT LIKE '973%'
```

Figure 4.10: Materialized View used to Optimize the SQL Operator in Figure 4.7

Once the materialized view in Figure 4.10 is created, the SQL statement in the SQL operator is rewritten into:

```
SELECT no, date, number_called, time, rate, min, amount
FROM itemized_call_973
```

The cost for this SQL statement is only 500 for a single scan. Compared to the original cost 5500, it is 10 times faster.

There are couple of research issues in this area. First, we need to identify a proper set of denormalization techniques for the schema optimization. Second, we need to consider how to capture the order information at the conceptual schema. Third, we need to consider how the hierarchical information will be optimized at the conceptual schema.

### 4.5.2 Optimization Selection

From Section 4.2 and Section 4.1, we know that an XQuery workload is composed of multiple XQuery statements, and one XQuery statements can be translated into more than one SQL statements. Also, from Section 4.5.1, we know for each SQL statement, there could be more than one way to do the denormalization. In such a large search space, how can we find the best denormalization techniques that can benefits all the SQL queries in the translated SQL workload?

Below we sketch some initial simpliestic strategy for conceptual schema tuning. The two inputs are a given conceptual schema and a SQL workload (derived from the XQuery workload).

1. We generate the query plan (the XAT) for each SQL statement in the SQL workload.
2. Then we generate a matrix of the cost of the SQL statements is generated as in Table 4.2.

| Cost | $SQL_1$ | ... $SQL_j$ ... | $SQL_m$ | total |
|---|---|---|---|---|
| $schema_1$ | $cost_{1,1}$ | ... $cost_{1,j}$ ... | $cost_{1,m}$ | $\Sigma_{k=1..m} W_k \times cost_{1,k}$ |
| ... | ... | ... | ... | ... |
| $schema_i$ | $cost_{i,1}$ | ... $cost_{i,j}$ ... | $cost_{i,m}$ | $\Sigma_{k=1..m} W_k \times cost_{i,k}$ |
| ... | ... | ... | ... | ... |
| $schema_n$ | $cost_{n,1}$ | ... $cost_{n,j}$ ... | $cost_{n,m}$ | $\Sigma_{k=1..m} W_k \times cost_{n,k}$ |

Table 4.2: Cost Matrix for SQL Workload.

   The $SQL_i, (1 \leq i \leq m)$ represents the $i^{th}$ SQL statements from the SQL workload. The $W_k$ represents the weight of $k^{th}$ SQL statement in the SQL workload. The $schema_j, (1 \leq j \leq n)$ represent the $j$ relational schemata. The $cost_{i,j}$ identifies the cost for $SQL_j$ under $schema_i$. The number of schemata is decided by the number of iterations in our algorithm. The $schema_1$ is the initial schema generated by the fixed mapping. The total cost of the SQL workload for a given schema is stored in the *total* column.

3. Once we find the lowest total cost, the iteration terminates.

Now, let's roughly estimate the search space of this optimization problem. Assume the total number of optimization steps is $q$, and for each step we have $p_q$ different operators can be applied to the schema optimization. Then the search space of the optimal optimization plan is a function of $p$ and $p_q$, and $p_q$ is a function of $q$. As we can see, this is a NP problem.

There are couple of research issues in this area. First, we need to identify an enumeration method to iterate all the meaningful optimization plans. Second, we need a criteria to

stop the enumeration.

## 4.6 Evaluation

We are implementing the Rainbow system using Java for the basic logic, using Oracle8i as the backend relational storage, using JDBC techniques to integrate the Java code and backend database, using the XML4J package to manipulate XML documents, and reusing XQuery package (if any) to do the query parsing.

Once the system is implemented, an evaluation is required to validate our proposed idea. That will include an experimental design and evaluation.

The first purpose of the experiment evaluation is to compare the performance of executing XML queries with naive XML document oriented query engines and with a relational query engine. We can check the correctness of our XML query translation and execution by comparing the query results from these two engines.

The second purpose of the experimental evaluation is to test whether the query performance is increased after the optimization by comparing the query performance before and after the optimization.

The third purpose of the experimental evaluation is to try to assess how close to the optimal design our XTuner may get. For this, we will given out a human designed workload and expected optimized relational schema and test whether the optimizer can generate the intended relational schema. Also, a general evaluation of the quality of the optimization is required.

# Chapter 5

# Relation to Other Work

## 5.1 XML Query Languages

There are a bunch of XML query languages proposed in the literature. They are XSL [17], XPath [52], XQL [40], XML-QL [13], Lorel [2], YATL [8], Quilt [7], and XQuery [53].

XSL (extensible stylesheet language) [17] is proposed by W3C for expressing stylesheets. It consists of a language for transforming XML documents and an XML vocabulary for specifying formatting semantics.

XPath (XML path language) [52] has also been proposed by W3C for addressing parts of an XML document. It is designed to be used by both XSLT and XPointer.

XQL [40] is proposed by Microsoft as an extension to the XSL [17] pattern language by adding Boolean logic, filters, indexing into collections of nodes. It is proposed as a general purpose query language, providing a single syntax for queries, addressing, and patterns.

XML-QL [13] has been proposed by AT&T labs, University of Pennsylvania, and University of Washington as a general purpose query language for XML. Like SQL, it has the SELECT-WHERE construct and borrow features for semistructured data.

Lorel [2] has been proposed by Stanford university. It's a semistructured query language and implemented as the query language of the *Lore* prototype database management system.

YATL [8] has been proposed by the INRIA, France. It is a conversion language used in the YAT system. The YAT system provides a means to build software components based on data conversion, e.g., wrappers or mediators, in a simple and declarative way. It can be also used for the integration of heterogeneous data sources.

Quilt [7] has been proposed by the IBM Almaden research center. It's a general purposed XML Query language for heterogeneous data sources. It is derived from XPath, XQL, XML-QL, SQL, OQL, Lorel and YATL.

Most recently, XQuery [53] has been proposed by W3C as the XML query standard. XQuery is derived from Quilt. In this proposal, we adopt the XQuery standard.

## 5.2 XML Schema

XML is semistructured data. In order to communicate between different parties, they have to agree on a standard schema specification. One simple schema description comes with XML 1.0 [3], called Document Type Declaration (DTD). DTD specifies the syntax of a valid XML document, hence enables the same syntax between different communication parties, while detailed semantics of the data is not specified. A more comprehensive schema has been proposed W3C called XML Schema [49]. "XML Schema expresses shared vocabularies and allows machines to carry out rules made by people. They provide a means for defining the structure, content and semantics of XML documents." excerpt from [49].

The XQuery [53] we chose in this proposal is using XML Schema for its type system. In our proposal, we have started using the DTD which is simpler than XML Schema to illustrate our ideas. However it is possible to extend the work to support XML Schema.

## 5.3 Model Mapping

### 5.3.1 Mapping from XML to Relational

Recent studies investigate different approaches for storing XML data in RDBMS [16, 44, 14, 22, 21]. Work on translating XML queries into SQL statements and reconstructing the XML query results has also appeared in the literature [15, 5].

The STORED [14] project studies how to store XML data without a known DTD into relational databases. It creates a relational data schema by first applying data mining to a large number of similar XML documents, and then abstracting a relational data schema. For XML data that is too irregular, they use an overflow graph. Then, STORED loads the data into the relational tables.

Lee et. al. [22] consider to selectively keep the meta knowledge described in the DTD as constraints on the relational schema. Similar to this work, we also use metadata to keep the meta knowledge captured in a DTD. They first transfer the DTD before they use it into the mapping. They have three kinds of restructuring: 1) get rid of attributes and convert them into elements; 2) get rid of groups and change the nesting relationships accordingly; 3) get rid of duplicates and change the nesting relationships accordingly. We can formalize the above three restructuring techniques as operators applied on the DTDM within our framework. For this case, step 1 will be handled by the *pushUpAttribute*() operators,

step 2 by the *pushUpNesting*() operators, and step 3 by the *mergeNesting*() operators of our framework.

Kappel et. al. [21] use UML to describe a general approach towards integrating XML documents and relational schemata. Our approach is to store the data and metadata in the same relational model, and we instead focus on a flexible framework for optimization.

Instead of bringing the semi-structured data into the relational model, there are other approaches to bring the XML data into semistructured (e.g., Lorel [27], Araneus [28]), object-oriented, or object-relational DBMSs. Commercial RDBMSs, such as, DB2[10] or Oracle[33], have started to incorporate XML techniques into their databases, e.g., IBM DB2 XML Extender [10], and Oracle 8i [36].

Recently IBM Alphawork proposed a new set of Visual XML tools which can visually create and view DTDs and XML documents. In these tools, they have proposed an idea similar to ours of breaking DTDs into elements, notations, and entities as we did in our metadata. Especially, they use components such as *group* with properties of sequential, choice, etc., *attribute*, and *relationship* with properties of repetition, to construct DTDs. They also provide tools to do XML translation and XML generation from SQL queries. However, they have not proposed a general way for loading the XML documents into relational tables. We take one step further towards loading the XML documents into relational tables by our metadata-driven approach.

The DB2 XML Extender [10] can store, compose and search XML documents. For storing, they either store the XML document as a whole, called *XML column*, or store pieces of XML data into several tables, then called *XML collections*. Oracle 8i [35] extends Oracle DBMS to an XML enabled database server for XML restructuring and loading. They store an XML document over several object-relational tables [37]. Their XML SQL utility provides the ways of specifying the mappings. Their approach requires users to manually design the relational schema and to specify the mapping between the DTD and relational schema, while we instead perform mapping and loading automatically based on the characteristics of the DTD.

Our work is different from the above relational-to-XML mapping in two ways: 1) Ours are lossless mapping not only for the XML data, but also for the schema. 2) The whole mapping process is automatic without any human interference.

### 5.3.2 General Mapping from Relational to XML

There are a few projects at the direction of publishing relational data into XML documents. The two most impact papers are SilkRoute [15] and XPERANTO [5].

SilkRoute [15] has been proposed by AT&T and University of Pennsylvania. It provides

a framework to automatically publishes relational data into XML documents by a powerful declarative data transformation language called RXL (Relational to XML transformation Language). RXL is used with XML-QL. RXL and XML-QL will be merged into RXL queries and finally translated into SQL queries over the base relations. The relational data to XML data mapping is captured by RXL.

XPERANTO [5] has been proposed by the IBM Almaden research center. It provides another framework to publish object-relational data into XML documents. It uses a uniform XML-based query interface, i.e., XQuery. In their framework, they use the XQuery powerful relational query capability to query both XML data and relational data. The relation to XML mapping is captured by XQuery.

### 5.3.3 Mapping to Object-Oriented Database

Other related work on XML repositories are Object Design's Excelon [34] and POET'S Content Management Suite [19] and XEM [47]. They directly map the XML documents and manage them using object-oriented database management systems.

## 5.4 Querying XML by Relational Databases

Florescu et. al. [16] have conducted a benchmark test on the relational schemata generated from XML based on four basic mapping approaches. Different from our assumption, this work does not require a DTD. For the attribute-inline approach, they will create one table for each attribute with the attribute value inlined. While they claim to assume no DTD existence, they must be aware of the existence of the DTD in order to create the table schema. They will perform the restructuring of the DTD of getting rid of groups, which is applying our *pushUpNesting*() operators to an existing group.

Shanmugasundaram et. al. [44] have investigated schema conversion techniques for mapping a DTD to a relational schema. They give (and compare) three alternative methods based on traversal of a DTD tree and creation of element trees. They first simplify the DTDs and then map those into relational schemata. Instead we capture the whole DTD into metadata tables. Thus our approach captures more of the structure, properties and embedded relationships among elements in the XML documents, e.g., groups and difference between + and *.

## 5.5 Database Tuning

Database tuning has been always the main focus of database administrators. A lot of commercial tools have been developed to help with database tuning, for example, Microsoft SQL server [29], Oracle [33], IBM's DB2 [18], etc. Also, in the academic area, as mentioned in [39] the database tuning has been studied extensively.

Traditionally, database tuning includes workload specification, database statistics collections, physical schema optimization, and if possible, conceptual schema optimization.

Our proposal is focusing on the conceptual schema tuning for XML query evaluation by a relational query engine. Hence, we assume a default physical schema design for a given conceptual schema to reduce the search space of our conceptual schema optimizer. The tuning of the physical schema is going be the next task including XML navigation index and special clustering techniques.

When we use the denormalization techniques to optimize the relational schema, we have to consider the **rule of reconstruction** (ROR) [42] if we want to update the optimized relational schema. "When the Rule of Reconstruction is ignored and the data updated, and data is corrupted.", from [42]. The ROR means all optimization techniques have to be implemented inside RDBMS, and if possible using SQL only. [42] has proposed an initial investigation of the ROR in the denormalization techniques, such as, vertical and horizontal partition. They said that vertical partition can be implemented by projection/join pairs, and horizontal partition can be implemented by select/union pairs. They also propose more research issues for updating pre-join tables, 1) What happened to insert? 2) What happened to update a duplicated data during the join? 3) What happened to delete a duplicated data during the join?

## 5.6 Workload

Oracle's workload [33] is composed of categories and workload elements of each categories. The four categories are: application, business unit, transaction, and request. They use the importance values (ranking) and frequency values to compute the relative importance value of the workload elements. The frequency value is the number of times the element was executed each time its parent in the workload hierarchy was executed. A "workload analysis style" is used when to assign different weights to the importance value and the frequency value.

IBM DB2's workload [18] is stored in the ADVISE_WORKLOAD table. The table is composed of SQLs and their frequency in a given time period.

Microsoft SQL server's workload [29] has multiple representations. Basically, the work-

load is specified by a sequence of SQLs without requiring any additional information. A TSQL file will be generated from those SQLs or other monitoring applications that provides additional information for the index optimization.

## 5.7 XML Query Optimization and XML Indexing Techniques

XML queries are different from SQL queries. They have a lot of navigation queries and are order sensitive. The navigation queries are usually mapped down to join queries in SQL statements. Special indexes (hierarchy and order indexes) in addition to the traditional hash and tree typed indexes are required for XML query optimization. Deschler's MS thesis [11] is focusing for example on XML navigation indexing.

# Chapter 6

# Research Schedule

It is planned to finish work on the dissertation in Spring 2002. In order to achieve this goal, the following schedule of work is intended:

| | |
|---|---|
| now– May 2001 | Work on Proposal and Proposal Defense. |
| May 2001– June 2001 | Restructuring and Denormalization. |
| July 2001– September 2001 | Work on the Query Translation. |
| October 2001– November 2001 | Work on Cost Model. |
| December 2001– February 2001 | Work on Schema Optimization. |
| March 2001– April 2002 | Finishing Work on the Dissertation. |

# Bibliography


[1] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In ACM, editor, *Proceedings of ACM Symposium on Principles of Database Systems*, pages 254–263, New York, NY 10036, USA, 1998. ACM Press.

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

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

[4] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensaible markup language (xml) 1.0. http://www.w3.org/TR/REC-xml, Feburary 1998.

[5] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for publishing object-relational data as XML documents. In *The VLDB Journal*, pages 646–648, 2000.

[6] R. G. G. Cattell and T. Atwood, editors. *The Object Database Standard, ODMG-93*. M. Kaufmann, 1993.

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

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

[9] D. Chamberlin and J. Robie and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *ACM SIGMOD Associated Workshop on the Web and Databases (WebDB 2000), Dallas, Texas*, pages 53–62, May 2000.

[10] DB2 UDB XML Extender. XML Extender Administration and Programming. http://www-4.ibm.com/software/data/db2/extenders/xmlext/library.html, December 1999.

[11] K. Deschler. Xml navigation indexing. Master's thesis, Worcester Polytechnic Institute, 2001.

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

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

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

[15] M. Fernandez, W. Tan, and D. Suciu. SilkRoute: Trading between Relations and XML. http://www.www9.org/w9cdrom/202/202.html, May 2000.

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

[17] W. X. W. Group. Extensible Stylesheet Language (XSL). http://www.w3.org/TR/WD-xsl/.

[18] IBM. DB2 Product Family. http://www-4.ibm.com/software/data/db2/.

[19] P. Inc. Poet content management suite. http://www.poet.com/products/cms/cms.html, 1999.

[20] C. S. Jensen, J. Clifford, R. Elmasri, S. K. Gadia, P. Hayes, S. Jajodia, C. Dyreson, F. Grandi, W. Kafer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, P. Tiberio, and G. Wiederhold. A consensus glossary of temporal database concepts. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(1):52–64, 1994.

[21] G. Kappel, E. Kapsammer, S. Rausch-Schott, and W. Retschizegger. X-Ray - Towards Integrating XML and Relational Database Systems. In *International Conference on on Conceptual Modeling*, October 9-12 2000.

[22] D. Lee and W. W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *International Conference on on Conceptual Modeling*, October 9-12 2000.

[23] W. Lee, G. Mitchell, and X. Zhang. Integrating xml data with relational database. In *Int. Conference Distributed Computing Systems*, 2000.

[24] A. Levy, A. Mendelzon, and Y. Sagiv. Answering Queries Using Views. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 95–104, May 1995.

[25] I. Manolescu, D. Florescu, and D. Kossmann. Pushing XML Queries inside Relational Databases, 2001.

[26] S. Marcus and V. S. Subrahmanian. Foundations of Multimedia Database Systems. *Journal of ACM*, 1996.

[27] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. In *SIGMOD Record 26(3)*, pages 54–66, September 1997.

[28] G. Mecca, P. Merialdo, and P. Atzeni. Araneus in the era of xml. In *Bulletin of the Technical Committee on Data Engineering*, pages 19–26, September 1999.

[29] Microsoft Inc. Microsoft SQL Server. http://www.microsoft.com/sql/default.asp.

[30] Microsoft Inc. XML Query Language Demo. http://131.107.228.20/xquerydemo/demo.aspx, April 2001.

[31] C. S. Mullins. Denormalization Guidelines. *The Data Administration Newsletter*, 1.0, 1997. http://www.tdan.com/i001fe02.htm.

[32] I. S. Mumick. The Rejuvenation of Materialized Views. In *CISMOD*, pages 258–264, 1995.

[33] O. T. Network. Oracle8i. http://www.oracle.com/database/oracle8i, 2000.

[34] Object Design. Excelon Data Integration Server. http://www.odi.com/excelon, 1999.

[35] Oracle. Oracle xml sql utility for java. http://technet.oracle.com/tech/xml/oracle_xsu/, 1999.

[36] Oracle Inc. XML SQL Utility for Java. http://technet.oracle.com, 2000.

[37] Oracle Technologies Network. Using XML in Oracle Database Applications. http://technet.oracle.com/tech/xml/info/htdocs/otnwp/about_oracle_xml_products.htm, November 1999.

[38] J. Paredaens, J. V. den Bussche, and D. V. Gucht. Towards a theory of spatial database queries. In *Symposium on Principles of Database Systems*, pages 279–288, 1994.

[39] R. Ramakrishnan. *Database Management Systems*. WCB/McGraw-Hill, 1997.

[40] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html, September 1998.

[41] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL99/pp/xql.html, September 1999.

[42] D. Root. Denormalization and the Rules of Resconstruction. *The Data Administration Newsletter*, 14, 2000. http://www.tdan.com/i014ht04.htm.

[43] G. Sanders and S. Shin. Denormalization Effects on Performance of RDBMS. In *Proceedings of the 34th Hawaii International Conference on System Sciences*, 2001.

[44] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99)*, pages 302–314, Edinburgh, Scotland, UK, September 1999.

[45] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *The VLDB Journal*, pages 65–76, 2000.

[46] D. Srivastava, S. Dar, H. Jagadish, and A. Levy. Answering Queries with Aggregation Using Views. In *International Conference on Very Large Data Bases*, pages 318–329, 1996.

[47] H. Su, D. Kramer, L. Chen, K. T. Claypool, and E. A. Rundensteiner. XEM: Managing the Evolution of XML Documents. In *RIDE-DM*, pages 103–110, April 2001.

[48] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating xml. In *SIGMOD*, 2001.

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

[50] W3C. Document Object Model (DOM). http://www.w3.org/TR/REC-DOM-Level-1/, 1998.

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

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

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

[54] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *International Conference on Very Large Data Bases*, pages 136–145, 1997.

[55] X. Zhang, W.-C. Lee, and G. Mitchell. Metadata-driven Approach to Integrating XML and Relational Data. Technical Report TR-0404-12-00-4240, Verizon Laboratories Incorporated, 2000.

[56] X. Zhang, G. Mitchell, W.-C. Lee, and E. A. Rundensteiner. Clock: Synchronizing Internal Relational Storage with External XML Documents. In *RIDE-DM*, pages 111–118, April 2001.

# Appendix A

# XQuery Translation Case Study

This chapter shows one case of translating XQuery into SQLs.

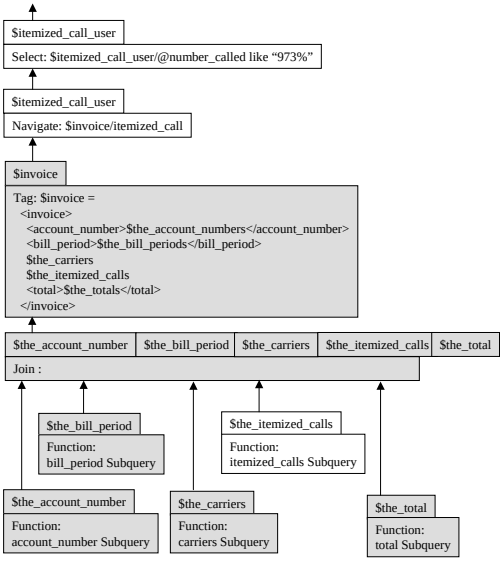


Figure A.1: XAT to SQL Translation Step1.

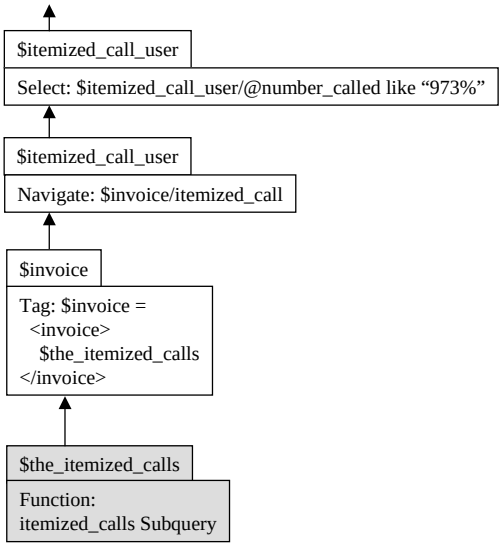


Figure A.2: XAT to SQL Translation Step2.

The intermediate steps are described from Figures A.1 to Figure A.8 . Each step of the transformation between figures are described below.

1. Figure A.1: Combine the user XAT with XML fragment view XAT.
2. Figure A.2: Cut unrelated subqueries, remove Join operator, and update Tag operator.
3. Figure A.3: Combine with itemized call subquery.
4. Figure A.4: Pullup Tag operator, selection operator is changed accordingly.
5. Figure A.5: Change variable $itemized_call.

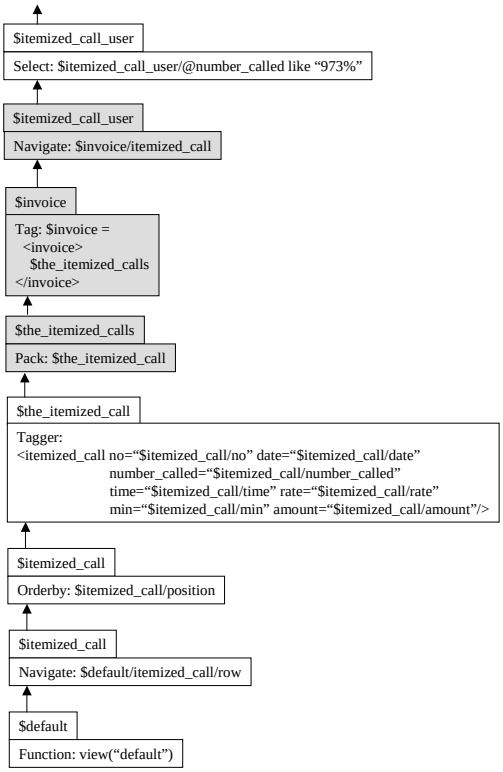


Figure A.3: XAT to SQL Translation Step3.

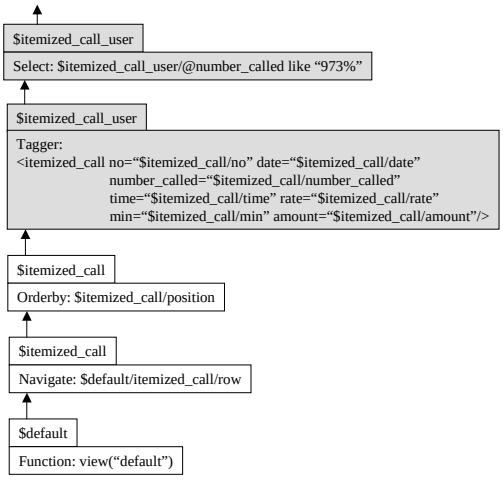


Figure A.4: XAT to SQL Translation Step4.

6. Figure A.6: Transfer into SQL operator.
7. Figure A.7: Merge Orderby operator.
8. Figure A.8: Merge Select operator.

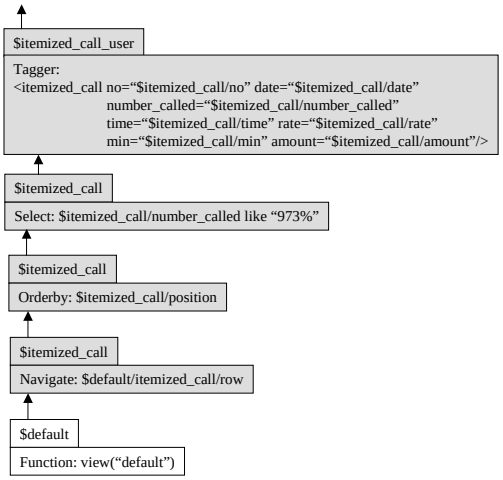


Figure A.5: XAT to SQL Translation Step5.

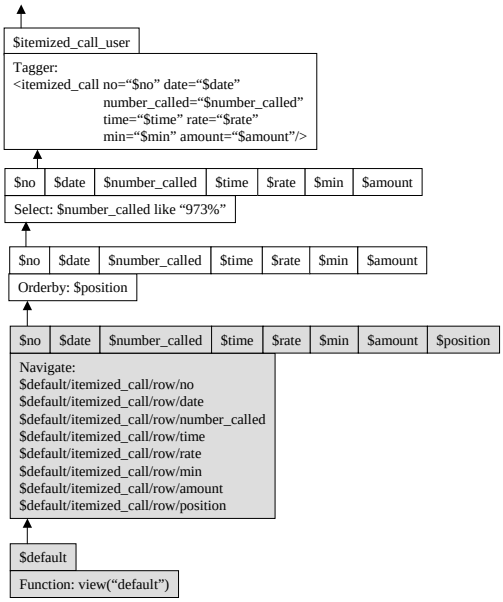


Figure A.6: XAT to SQL Translation Step6.

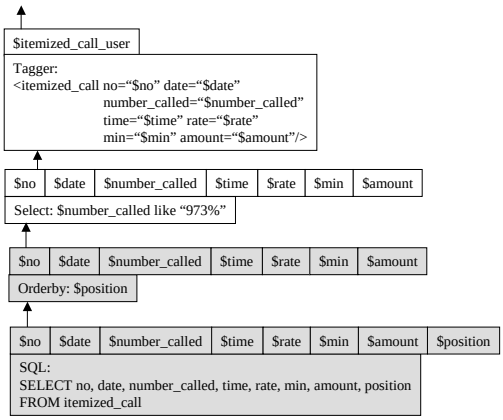


Figure A.7: XAT to SQL Translation Step7.

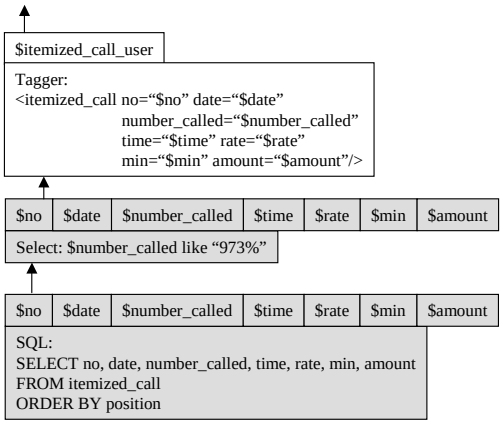


Figure A.8: XAT to SQL Translation Step8.

# Appendix B

# Default Relational View by XQuery

Reviewing what mapping mechanism we have in the Section 3.2, we notice that, we use three different languages in the two-way mapping. The languages are SQL, XQuery and a programming language. This section is attempt to use only one language, XQuery, to specify both relational to XML data model mapping and XML to relational data model mapping.

So far, the default relational view is implemented by an object-oriented language (Java) with database manipulation capabilities (JDBC). A more general way to describe the relation to XML mapping can be expressed purely by XQuery and a *reverse default XML view* for a given XML Schema.

*Reverse default XML view* is counterpart of the default XML view provided by XQuery. As we have described in Section 3.2.1, a default XML view will take any table and map it into an XML document. Then, a *reverse default XML view* will take a XML document that is compliant to the DTD in Figure B.1 (called Relational DTD) and convert it into relations. We call such XML *relational XML*. The constraint in the data is not considered as the default XML view proposed by XQuery.

```
<!ELEMENT DB (TABLE*)>
<!ELEMENT TABLE (TUPLE*)>
<!ATTLIST TABLE tablename CDATA #REQUIRED>
<!ELEMENT TUPLE (COLUMN*)>
<!ELEMENT COLUMN (#PCDATA)>
<!ATTLIST COLUMN columnname CDATA #REQUIRED>
```

Figure B.1: Relational DTD.

The XQuery translates the XML documents into the DTD specified by Figure B.1. Then the *relational XML* documents can be easily loaded into relational tables.

We have to point out that we only give out the syntax sugar by using the XQuery, so that, all the mappings (from relational to XML or vice versa) can be described in one **uniform** language. However, the *reverse default XML view* doesn't solve the problem of how an XML document will be translated into the relational DTD, which is discussed by different mapping approaches proposed in the literature.

The second advantage of using XQuery to specify the XML to relational mapping is **easy integration** with SQL statements if we think of SQL as a strict subset of XQuery.