XML-RL: AN XML QUERY SYSTEM

A Thesis

Submitted to the Faculty of Graduate Studies and Research

In Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science

University of Regina

By

Hongwei Qi

Regina, Saskatchewan

June, 2004

Your file Votre référence
ISBN: 0-612-96098-6
Our file Notre référence
ISBN: 0-612-96098-6

UNIVERSITY OF REGINA

FACULTY OF GRADUATE STUDIES AND RESEARCH

PERMISSION TO USE POSTGRADUATE THESIS

TITLE OF THESIS: XML-RL: An XML Query System

NAME OF AUTHOR: Hongwei Qi

DEGREE: Master of Science

In presenting this thesis in partial fulfillment of the requirements for a postgraduate degree from the University of Regina, I agree that the Libraries of this University shall make it freely available for inspection. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the professor or professors who supervised my thesis work, or in their absence, by the Head of the Department or the Dean of the Faculty in which my thesis work was done. It is understood that with the exception of UMI Dissertations Publishing (UMI) that any copying, publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Regina in any scholarly use which may be made of my material in my thesis.

SIGNATURE: _____

DATE: Apr 13, 2004

UNIVERSITY OF REGINA

FACULTY OF GRADUATE STUDIES AND RESEARCH

SUPERVISORY AND EXAMINING COMMITTEE

Hongwei Qi, candidate for the degree of Master of Science, has presented a thesis titled, ***XML-RL: An XML Query System***, in an oral examination held on April 13, 2004. The following committee members have found the thesis acceptable in form and content, and that the candidate demonstrated satisfactory knowledge of the subject material.

External Examiner:	Dr. Dianliang Deng, Department of Mathematics & Statistics
Supervisor:	*Dr. Mengchi Liu, Carleton University
Committee Member:	Dr. Samira Sadaoui-Mouhoub, Department of Computer Science
Committee Member:	Dr. Lawrence V. Saxton, Department of Computer Science
Chair of Defense:	Dr. Thomas Conroy, Faculty of Engineering

*Attended via video conference, is currently at Carleton University

Abstract

With the constantly increasing use of the Web, the problem of ensuring interoperability between different systems has aroused more interest. With the popularity of Extensible Markup Language (XML) on the Web come additional problems of XML storage, XML document transformation, XML documents integration, etc. In particular, a fundamental need is to provide an easy-to-understand, easy-to-use, and powerful query language for the purpose of advanced use of XML. Supporting specifications released include: the XML Query data Model; the XML Query Algebra; the XML Query Use Cases, which articulate goals, requirements, and usage scenarios for the W3C XML Query languages.

The W3C workshop on query languages for XML has produced a number of interesting proposals for extracting information more efficiently from XML documents. Among them, XQuery is a working draft of the latest public version of the Web Consortium's XML Query Specification. At this point, discussions are still going on.

XML-RL is introduced in this thesis as a rule-based declarative query language for XML. XML documents are viewed from a database point of view, which is similar to the complex object data model, so that query representation becomes also higher level. Unlike queries on traditional relational databases whose results are always flat relations, the results of XML queries can have complex structure. XML-RL provides a natural way of separating the querying part and result constructing part of query processing using rule body and rule head respectively. Thus it avoids a problem in XML query languages, where the querying part and result constructing part are intermixed in a nested way. Using several rules for a query, XML-RL can express complex queries in a simple and natural manner.

This thesis describes the design and implementation of an XML query system for processing XML documents, automatically retrieving information, constructing the result as new XML documents for the user, based on the declarative XML Query Language – XML-RL. The system not only provides an expressive yet very simple way for retrieving XML information, but also allows advanced processing, such as logic deduction, integration and transformation of the data on the XML documents.

Acknowledgements

I would like to take this opportunity to thank many people who made this thesis possible.

Many thanks go first to my supervisor, Dr. Mengchi Liu for his valuable advice and guidance on both my research and thesis, and also for his financial support during my study. I want to thank my family for being supportive over years of my study, especially my husband Sheng, for his tireless effort and support. Many thanks go to my friends: Hong Chen, Yibin Su, Jessica Zheng, Dan Wu, Magdalena Widjaja, Richard Kuo for their constant help and encouragement.

I would also like to recognize the efforts of the committee for evaluating this work. Finally, I am very appreciative to the funding agencies such as the Department of Computer Science, and the Faculty of Graduate Studies and Research of the University of Regina, Natural Science and Engineering Research Council of Canada, whose financial support made this thesis possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The information on the Web is exchanged and displayed using languages such as Hypertext Markup Language (HTML) [70] and Extensible Markup Language (XML) [69].

As opposed to from HTML, XML does not have a predefined tag set. XML provides a facility to define tags, document structures can be nested to any arbitrary level. It allows applications to invent tags according to their needs and define the semantics for them. This feature is very beneficial to a variety of richly structured applications over the web. XML also provides an optional description of its grammar if applications need to perform structural validation.

With extensibility, structural validation, data checking and many useful features like its media independence to publish content in multiple formats, and its vendor and platform independence to process any conforming document using standard commercial software or text tools, XML is fast emerging as the dominant standard for data representation and exchange on the Web.

The popularity of XML brought additional problems of XML storage, XML document transformation, integration of XML documents, etc. In particular, a fundamental need is to provide an easy-to-understand, easy-to-use, and powerful query language for the advanced use of XML.

With the wide adoption of XML, the goal for XML has been expanded far beyond

information representation and exchange over the Internet. Now more sophisticated needs arise, such as manipulation, reuse and integration of data from diverse resources. How to query XML documents has become a fundamental problem that arouses great research interest. Various XML query languages proposed in the past several years, such as XML-QL [24], Lorel [7], XML-GL [19], XPath [61], XSLT [62], YATL [21], XDuce [36], XQuery [11], XML-RL [49], etc. Some of them are in the tradition of database query languages, others are closely inspired by XML.

For a comparative analysis of some of these language, see [13]. The XML Query Working Group has recently published XML Query Requirements for XML query languages [26]. The discussion is going on within the Web Consortium, within many academic forums and within the IT industry, with XQuery being selected as the basis for an official W3C query language for XML. These languages bring facilities to specify part of XML content, query models to model the XML data and syntax suitable for the XML hierarchical features. Some recent efforts, like the implementation of XQuery conforming to the W3C Working Draft specification for XQuery 1.0 also addressed the issues and importance to design a "small, easily implementable language in which queries are concise and easily understood" [11]. Therefore a system that supports a simple, powerful, and high-level query language to explore the full power of XML is of great practical significance.

Different query languages use different constructs. Lorel uses SELECT-FROM-WHERE syntax similar to SQL, and XML-QL uses WHERE-CONSTRUCT clauses, with a WHERE clause to query the data and a CONSTRUCT clause to build the query result. XQuery employs the constructs of the FOR-LET-WHERE-RETURN (FLWR) expression: FOR and LET are used to query data and bind variables, WHERE is used to further filter the result and RETURN to construct the result. In our point of view, the main problem with the existing query languages is that they mix querying and result constructing, which in fact is an issue inherited from SQL. In order to deal with a complex query, nested queries have to be used in those query languages. Also because of the complex nature of XML, when compounded with other constructs, it is cumbersome for these languages to express a complicated query and also hard to comprehend. As we studied those XML query languages,

we noticed that it is likely not sufficient emphasis has been laid from a declarative language perspective to consider the possibility of accomplishing it.

One of the distinct features of XML-RL is that it clearly separates the query and result construction.

Another feature of XML-RL [49] is that it is a pure declarative XML query language. It allows the users to focus on *what* to query instead of *how* to query.

In some existing XML query languages, due to the low level representation and view of XML, users need to know more about the query procedure, such as how a query on a list-value is carried out or how to create a result from a set of values extracted from XML. For instance, in XQuery, the FOR clause is used to range over a list of values in an XML document and also used construct a list of values in the query result.

By using the mechanism of logical variables, XML-RL also accomplishes information extraction and result reconstruction in a simpler way. It makes the queries written in XML-RL more natural and comprehensible. Based on XML-RL, we have developed an XML query system.

1.2 Thesis

In this thesis, we discuss the design and implementation of XML-RL. You will find how the implementation of system supports and reflect the features of the XML-RL language. We will address the issues we met while developing the system as well as the experience gained on how we devised different techniques to solve those problems.

The XML-RL system is intended to provide a high level view of structures and data in XML format, while conforming to the existing standard in implementation. A key contribution of this prototype system is that it demonstrates the functionality of an XML query language that incorporates some of features of database and logic programming language and is easier for users to understand and use.

Components are designed to work together in order for the system to function. In the fast changing XML world, some of the standards are widely accepted while some are still evolving. We implemented the system with regards to standards and

other ongoing efforts both from W3C and other research groups, and leave some implementation open for change. In this way, more flexibility for future modification is provided.

1.3 Structure of Thesis

This thesis is organized as follows: the background information on XML development, basics and characterization is in Chapter 2. An introduction to XML-RL is provided and some comparison with other XML query languages are given in Chapter 3.

The system architecture and implementation issues for the XML-RL are discussed in Chapter 4. Chapter 5 discusses the changes needed to apply to the current system to implement the revised XML-RL language. The thesis closes with future work and conclusions in Chapter 6.

Chapter 2

Background

2.1 The History of XML

To fully understand the objectives of the research on XML query languages, let us first take a brief look at the development of XML from the historical point of view.

2.1.1 Markup

A *Markup language* is a set of defined codes and rules for specifying markup. XML, HTML, XHTML and SGML are all called *Markup Languages.* Although the Markup language is a fairly recent technology, "*markup*" has long been used. "Markup" was used to refer to the formatting languages that specify the layout. Nowadays it is a method of making the interpretation of a text explicit and therefore processible, as in XML, HTML, etc. Everything in a document that is not content is *markup.*

Markup is divided into two types: *procedural markup* and *descriptive markup. Procedural markup* specifies the physical appearance of the documents, such as the font size, the character encoding, the text layout, etc. Most electronic publishing systems and software use procedural markup, for example, PostScript of Adobe or RTF (Rich Text Format) of Microsoft Corporation. Since procedural markup mixed appearance and content together, it makes it very difficult to change format.

Descriptive markup specifies the structure of a document, such as chapters and sections. Descriptive markup is also called logical markup, and it is based on the

content of a document. Thus by separating presentation from structure, descriptive markup allows for multiple presentation.

2.1.2 Markup Languages

In this section, we introduce several markup languages from a historical point of view.

SGML: A General Markup Language *Standard Generalized Markup Language* abbreviated as *SGML*, is an international standard, published in 1986 [10, 71]. It defines a standard for device-independent, system-independent methods of representing texts in electronic form. SGML is descriptive rather than procedural. SGML provides a standard method for describing the structure of a document. SGML is defined very general, thus it has many options hardly ever used. It is a meta-language that may be used to define other markup languages. HTML and XML are all markup languages based on SGML. HTML is defined using SGML, while XML is a subset of SGML.

HTML: A Web Publishing Language *HyperText Markup Language* is a widely used markup language for presenting hypertext documents over the Web [70].

HTML has a predefined set of markups or tags, adapted to the requirements of hypertext, multimedia and presentation of relatively simple documents. Most of those tags are procedural markups, for instance, tags for fonts, tables, lists etc, which mostly concern with how the text should appear on the screen.

Therefore, on one hand, the confined tags makes it much easier to publish the document on the Web, while on the other hand, the mixture of content and appearance limits HTML in several important respects.

The explosive success of the Internet spurred information creation, distribution, and access. As the Web documents grew larger and more complex, Web content providers began to experience the limitations of the medium. Functionalities such as extensibility, structure and data checking are needed by large-scale commercial publishing and other more sophisticated applications. Those requirement were beyond what HTML can provide.

Users were no longer satisfied with only publishing the information, they also need to integrate the information, to issue queries against it and to reuse the information. The features of HTML make it inappropriate to represent richly structured documents and the dissemination of the information. The separation of the content of the information from its presentation became crucial.

XML: A Language for data Representation and Exchange XML, *Extensible Markup Language* was developed officially in 1996. It is a simplified *subset* of SGML specially for Web applications.

XML is a meta language, but not as complex as SGML to work with. In a sense HTML is an application of SGML as HTML was a markup language defined using SGML, XML is a subset of SGML. It differs from HTML in three major respects:

- Firstly, one of the greatest strength of XML is that XML provides a facility to define tags and structural relationships between them.
- Secondly, document structures can be nested to any level of complexity.
- Finally, any XML document may have an optional description of its grammar use for applications to perform structural validation.

XML is a content-based meta-language, which means that it does not include any information about how it should be displayed. This is one of the biggest difference from HTML. To view an XML document, style sheets coded with *Cascading Style Sheets* (*CSS*) [72] or *Extensible Style sheet Language* (*XSL*) [68] must be used to render the display.

Simple as they are, these distinctions make it possible for a program receiving XML documents to interpret, filter and reconstruct documents to meet the needs of different applications.

The use of XML is not confined to Internet for representing data to browse, XML is also a widely used method in industry for data exchange purposes.

data representation in an enterprise system is achieved by creating XML documents with elements containing the content. database data may also be represented

in XML format. The structure and semantic information is contained in DTD (*Document Type Declaration*) or a schema. Different organizations can easily create and exchange data by agreeing on a DTD or a schema. It is simpler and more portable than other methods, which need complicated protocols and specialized software and hardware, for example, Electronic data Interchange (EDI).

XHTML: A Migration to XML The problem with the Web is that there are huge number of poorly coded, unstructured HTML documents. Even though XML may be the answer to this issue, to convert everything to XML and use a style sheet to render them involves a great deal of work.

XHTML which stands for *Extensible Hypertext Markup Language* [58], is a markup language written in XML. It is similar in many ways to HTML 4.0. XHTML documents are based on the XML syntax, and can also be served as a media type with appropriate style sheet support. XHTML is a version of HTML that has been reformulated as an application of XML. It differs from HTML in many ways: XHTML documents must be well-formed, end tags are required for non-empty elements, document head and body can not be omitted, elements and attributes must be lower cased, and so on.

By expressing HTML as an XML application, XHTML makes it possible to get many benefits from XML with minimal effort.

2.2 Introduction to XML

2.2.1 XML Basics

An XML document consists two parts: markup and content. Markups that may occur in an XML document include: elements, attributes, entity references, comments, processing instructions, marked sections, and document type declarations.

Elements *Element* is the most common form of markup in XML. It is responsible for establishing the logical structure of XML documents. Element declaration is as follows:

<! ELEMENT ElementName Type>

where ElementName is the name of the element and types of an element can be:

Empty - The element doesn't contain any content

Element Only - Element only contain child elements

Mixed - The element contains a combination of child elements and character data

Any - The element contains any content allowed by DTD.

For example, <!ELEMENT bib (book*) >

specifies that bib is an element, which may contain zero or more book subelements.

Attributes An element represents an object, and an *attribute* represents a property or the characteristic of the object, i.e., an element may have some attributes. An attribute declaration is as follows:

<!ATTLIST ElementName AttrName AttrType Default>

where ElementName is the name for the element which this attribute belongs to. AttrName is the name for the attribute, AttrType denotes the type of the attribute. It can be *CDATA* (unparsed character data), *Enumerated* (series of string values), *ENTITY*, *ID* (unique identifier) etc.

Attributes may have default values, and may be constrained to a predefined list of values. Default can be:

REQUIRED - The attribute is required

IMPLIED - The attribute is optional

FIXED - The attribute has a fixed value

default - The attribute's default value

For example, <!ATTLIST book year CDATA # REQUIRED >

Sometimes we use element and attribute interchangeably to model information.

Entity References *Entities* are storage containers used to hold data and are the building blocks of XML documents. They are typically made up of other entities via entity references. Every XML document has at least one entity that serves as the base entity (called a document entity) for the document. It forms the root of the document hierarchy, with which every XML processor will begin.

There are two categories of entities: parsed entities and unparsed entities. Parsed entities contain only text that is parsed by an XML parser. Unparsed entities store data that is not parsed, which can be text or binary.

Entity reference is used as an alias to a piece of data in XML. Every entity must have a unique name. Entity reference begins with the ampersand and ends with a semicolon. Most entity references have to be explicitly declared before use. But some reserved characters like greater-than (>), ampersand (&), apostrophe ('), quote ("), less-than (<) are pre-declared as *>*, *&*, *'* *"* and *<*. e.g.

```
<company> Shopper &apos; S Drug &amp; Mart </company>
```

Comments Like comments in programming languages, XML *comments* are to provide descriptions of document data purely for the sake of users. They are usually ignored by the XML parser and applications.

Comments begin with '<!' and end with '->'. Comments can be used anywhere within an XML document where parsed character data appears. The following is a comment in XML documents:

```
<!- This is a comment.->
```

Processing Instructions *Processing instructions* are an escape hatch intended for use by the application. An XML processor is required to pass it to an application, instead of doing anything to them. The following processing instruction indicates that the document is based on XML version 1.0.

```
<? XML version= "1.0"? >
```

Document Type Declarations A *valid* document must have a *document type declaration* (DTD) and conforms to the rules in the document type declaration which is a grammar or a set of rules that define what tags can appear in the document and how one element should nest in another.

Document Type Declaration is a very important part of the XML document. It is responsible for specifying the document's root element, defining elements, attributes, and entities specific to the document (internal DTD) or identifying an external DTD for the document. Documents depend on DTDs for validation purposes. A declaration allows a document to communicate meta-information about its content to an XML processor. These include: allowed sequence or nesting of tags, attribute values, types and defaults, and names of external files that may be referenced, and entities that may be encountered.

The following document type declaration denotes that the root element is `bib` and `"bib.dtd"` is the external DTD.

```
<! DOCTYPE bib SYSTEM "bib.dtd" >
```

CDATA Sections *CDATA* is also called unparsed character data section. CDATA sections are used to block off text which is to be ignored by an XML processors. XML processor will pass them to the application without parsing them. A section of CDATA is enclosed within the strings of <![CDATA [and]] >. For example,

```
<![ CDATA[ <nickname> Rick </nickname> ]]>
<studentID> 200798422< /studentID>
<lastname> Brown </lastname>
<firstname> Richard </firstname>
```

2.2.2 XML Processing

In XML and SGML, an outstanding feature is that they separate the representation of the information structure and content from information processing specifications. Information objects modeled through XML or SGML are named and described

as elements and attributes, in a way how they are defined, rather than how they should be displayed or processed.

Well-formed and valid XML Documents A *well-formed* document is a document easy for a computer program to read, and ready for network delivery. Specifically, in a well-formed document:

- All the begin-tags and end-tags match up
- Empty tags use the special XML syntax (e.g. `<xxx/>`)
- All attribute values are nicely quoted (e.g. `<store name="Superstore">`)
- All entities are declared

The document type declaration is also used to declare entities, re-usable chunks of text that can appear many times but only have to be transmitted once.

Display XML HTML pages use predefined tags, and the meaning of these tags is well understood. For example, `<p>` means a paragraph and `<h1>` means a header, and the browser knows how to display these pages. But with XML we can invent and use any tags, and the meaning of these tags are not automatically understood by the browser: `<table>` could mean a HTML table or maybe a piece of furniture. Therefore, there is no standard way to display an XML document. Here is an example:

```
<?XML version="1.0" encoding="ISO8859-1" ?>
<catalog>
   <cd>
   <title>Empire Burlesque</title>
   <artist>Bob Dylan</artist>
   <country>USA</country>
   <company>Columbia</company>
   <price>10.90</price>
   <year>1985</year>a
   </cd>
```

```
   ...
</catalog>
```

To ensure the desired presentation of XML document in various contexts, some languages are used: Cascading Style Sheets (CSS), Extensible Stylesheet Language (XSL) and JavaScript. Basically, these languages specify the layouts for those structural elements.

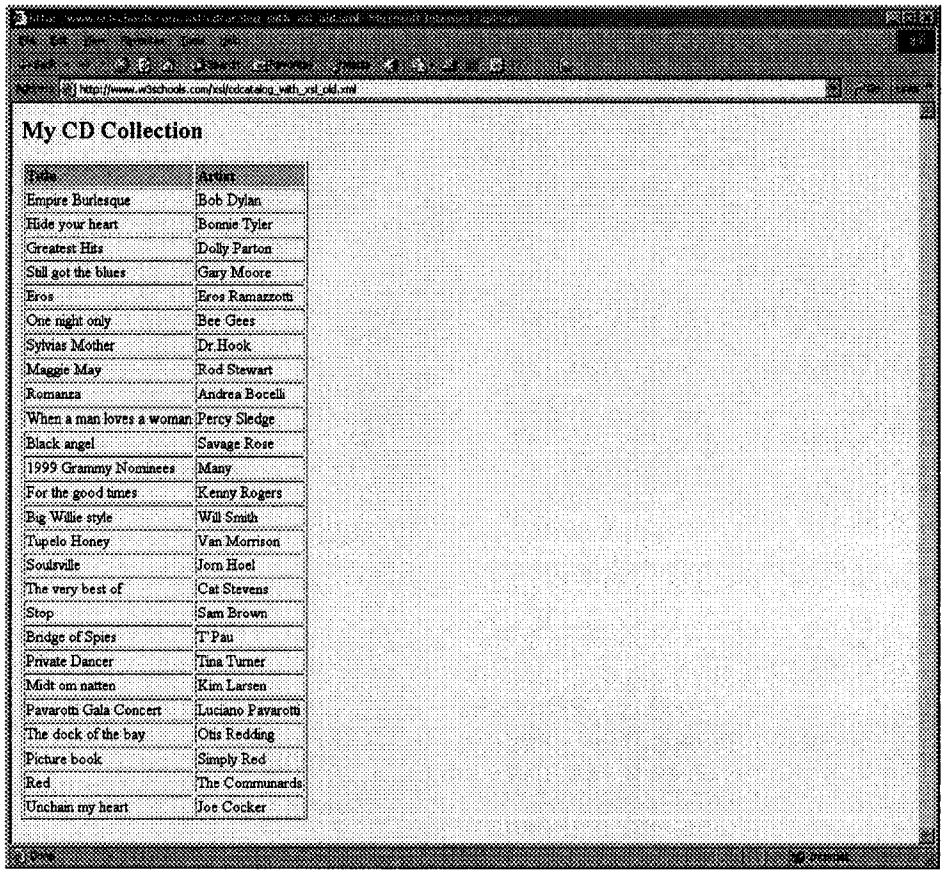
Consider the following example and see how an XML document is used as an HTML template to populate a HTML document with the given XML document. XML allows definition of how an XML file should be displayed by transforming the XML file into a format that is recognizable to a browser. Normally XML does this by transforming each XML element into an HTML element.

```
<?XML version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
   <html>
      <body>
         <table border="2" bgcolor="yellow">
         <tr> <th>Title</th> <th>Artist</th> </tr>
         <xsl:for-each select="CATALOG/CD">
         <tr>
            <td><xsl:value-of select="TITLE"/></td>
            <td><xsl:value-of select="ARTIST"/></td>
         </tr>
         </xsl:for-each>
         </table>
      </body>
   </html>
</xsl:template>
</xsl:stylesheet>
```

In the above file, the `xsl:for-each` element locates elements in XML document and repeats a template for each one. The `select` attribute describes the element in

the source document. The syntax for this attribute is called an XML *pattern*, and works like navigating a file system where a forward slash (/) selects subdirectories. The `xsl:value-of` element selects a child in the hierarchy and inserts the content of that child into the template.

Since an XML style sheet is an XML file itself, the file begins with an `XML` declaration. The `xsl:stylesheet` element indicates that this document is a style sheet. By adding a reference to the above stylesheet to the XML document (at line 2), the browser will transform the XML document into HTML. In a browser, it will look like the following:



My CD Collection

Title	Artist
Empire Burlesque	Bob Dylan
Hide your heart	Bonnie Tyler
Greatest Hits	Dolly Parton
Still got the blues	Gary Moore
Eros	Eros Ramazzotti
One night only	Bee Gees
Sylvias Mother	Dr.Hook
Maggie May	Rod Stewart
Romanza	Andrea Bocelli
When a man loves a woman	Percy Sledge
Black angel	Savage Rose
1999 Grammy Nominees	Many
For the good times	Kenny Rogers
Big Willie style	Will Smith
Tupelo Honey	Van Morrison
Soulsville	Jorn Hoel
The very best of	Cat Stevens
Stop	Sam Brown
Bridge of Spies	T'Pau
Private Dancer	Tina Turner
Midt om natten	Kim Larsen
Pavarotti Gala Concert	Luciano Pavarotti
The dock of the bay	Otis Redding
Picture book	Simply Red
Red	The Communards
Unchain my heart	Joe Cocker

Figure 2.1: Display XML with XML

2.3 Example XML Documents

We use W3C XML Query Use Cases [4] as the usage scenarios to test our query language. For the convenience of discussion, we give the XML documents and their corresponding DTDs. These DTD documents and XML documents are given in Appendix A.

As for the first XML sample document bib.xml, the DTD for bib.xml specifies a book element with year attribute and one title, at least one author or editor elements, one publisher and one price element. An author element has a last name and a first name. An editor element also contains an affiliation. Title, last name, first name, publisher and price are all parsed characters.

2.4 Summary of the Chapter

XML is a markup language. As HTML and SGML already exist, why XML becomes more popular and a focus in both in research and application. Through a comparison over the different markup languages, this chapter provides the overview of how XML came into being and its language features. The basic constructs of XML language: element, attribute, etc are introduced and lastly the chapter also discusses how an XML document is processed and displayed.

Chapter 3

Query Languages for XML

As XML technology is applied to an increasing number of fields, many issues arise, such as: defining message formats, storing structured information, printing XML documents on paper as well as publishing them on the Web and extending the expressive power of XML.

Meanwhile the research and new technology are fast emerging, in particular, the techniques to extract information from XML documents, to exchange XML data, to integrate XML from different sources. Among them are some supporting specifications, including:

- *XML Query data Model* [3], which is the foundation of XML Query Algebra.
- *The XML Query Algebra* [2] defines a formal algebraic model for an XML query language. Together with XML Query data Model, it provides semantics for the XML Query Language.
- *XML Query Use Cases* [4] specifies the usage scenarios for the W3C XML Query data model, algebra, and query language

Essentially these all concentrated on the problem of XML query technology. This chapter discusses in-depth several XML query languages, and make a comparison among them on their syntaxes, their query ability and their expressiveness. To address the issues involved in XML queries, firstly we take a look at various XML applications.

3.1 Various XML Applications

3.1.1 data Centric and Document Centric XML

XML documents vary as their applications differ. In order to decide what an application concerns and what strategy we shall use to deal with them, it is crucial to characterize the XML documents as of an appropriate type of application. XML documents fall into two broad categories: data-centric documents and document-centric documents [16].

data-centric documents are characterized by their fairly regular structure, fine-grained data, and little or no mixed content. They are usually designed for machine consumption where XML is used as a data transport, including sales orders, flight schedules, patient records and scientific data. A special case of data-centric documents is dynamic Web pages, such as online catalogs and address lists, which are constructed from known, regular sets of data. For the data-centric XML documents, the physical structures, such as the order of sibling elements, whether data is stored in attributes or PCDATA-only elements or whether entities are used, are often not significant.

Document-centric documents are characterized by their irregular structure, larger grained data, and lots of mixed content. They are generally designed for human consumption and their physical structures are important. Good examples of document-centric documents are email, advertisements, news reports, marketing brochures and almost all XHTML documents that contain larger chunks of information. These types of documents are usually managed on the file system.

Document-centric documents are easy to render on an output device. On the other hand, data-centric documents are typically easier to process with computer programs because the data are better organized.

In reality, the distinction between data-centric and document-centric documents is not always clear. An otherwise data-centric document such as an invoice might contain large-grained and irregularly structured data, such as a product part description. On the other hand, an otherwise document-centric document sometimes contains fine-grained, regularly structured data.

It is helpful to categorize the XML documents as data-centric or document centric

to decide the strategy in order to handle them properly.

3.1.2 database and Semistructured View Points

To manage XML data concerning different XML application areas, two point of views are brought up from the database and information retrieval communities [24].

A popular one from the database view point is to place those XML documents in a *database* framework. It stems from the similarities between the data-centric XML documents and databases. databases are applications that are specifically designed for the sole purpose of managing data, and are consequently very good at storing and retrieving data. DBMSs store data in rows and columns that make up tables.

Since databases are well understood and widely used, to tame XML with a database context to search, analyze and update the data can be a possible solution to the problem. Some projects have been proposed to build a database from XML documents, for instance, the work at University of Wisconsin [64], Migrated Lore at Stanford University [32]. Just as database schema is used for specifying meta information, in the context of XML, a DTD or an XML schema describes a model for the structure of the information, including the possible arrangement of tags and text in a valid document. The idea behind these projects is to convert XML documents and DTDs to rows and columns within a relational database management system.

The databases used for this purpose usually belong to two kinds: *XML enabled databases* and *native XML databases. XML enabled database* accepts XML, parses it into chunks that fit the database schema. When retrieving XML, the pieces are put together again. Some middle-ware is created to handle the translation process between XML and the internal format. There is also exploration in storing XML in database in its native form, referred as *native XML database.*

Many major database vendors have incorporated XML support into their products or provide tools to use XML with their databases. IBM has an XML Extender for DB2 to store XML documents in DB2 database; Microsoft also made extensions to SQL Server 6.5 and 7.0 to deal with XML; Oracle supports XML storage and indexing as well.

Another important view point came from the study of semistructured documents. The conventional approach of storing data in a file or document on a file system is quite different from the database mechanism. Though the self-describing feature of XML makes it distinct from HTML, researchers on semistructured data revealed many similarities between the semistructured data models and XML. Typically the semistructured data are tree-structured composed of edges and nodes, and have a notion of hierarchy and path to specify the part of a document.

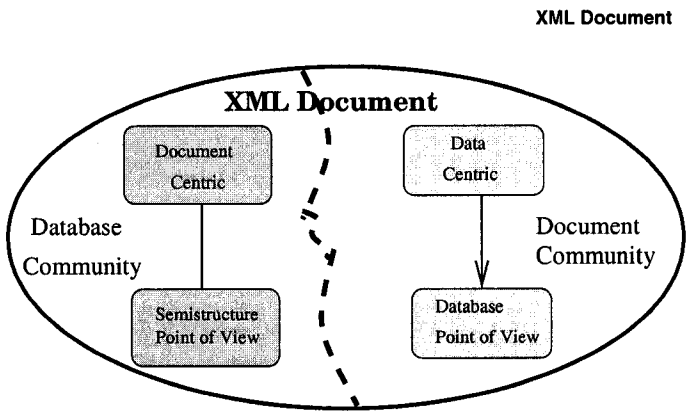


Figure 3.1: Views of XML From Semistructured Document and Database

The semistructured view and database view are not separated completely as shown in Figure 3.1. The techniques from both view points are helpful and complementary in dealing with XML. Semistructured document processing lacks many things found in databases: efficient storage, index, security, data integrity, multi-user access, though some of them are not quite necessary in XML, such as triggers, transaction and so on as in real databases. While in some other aspects, database technologies are not sufficient: order, hierarchy, fields of highly variable length, etc. where the strength of document management lays.

The study from diverse viewpoints reveals the opportunity to bring the power of different technologies together in solving the XML query problem.

This thesis later will introduce XML-RL and show how to treat the existing XML documents as extensional databases while incorporating the techniques to handle the

order and hierarchy, to make it more appropriate for query processing over XML documents.

3.2 Overview of Current XML Query Languages

To address the needs of various applications, several languages for extracting and restructuring the XML content have been proposed.

From the database perspective, the set of XML documents are considered to be a database, and the Document Type Descriptor to be the database schema. It borrows features of database query language to deal with XML data. Some Database Management Systems support XML document querying with a relational engine. For example, Oracle 8i provides basic query abilities over XML documents and the ability to convert the result to XML format. With some extensions to the SQL language, it is possible to bring the strength of database querying to XML. However due to the emphasis it lays on XML DTDs or the structural information, it poses limitation on the application. Currently not all XML documents have DTDs or their equivalent. Some of the systems require generating database schema based on DTD or schema information. As an instance, Oracle 8i provides built-in support for defining, parsing and rendering XML documents. Their users are required to supply a document descriptor to specify the XML document structure and how it should be stored in the database.

The solution based on the semistructured view mostly comes from the experience with HTML. It considers an XML document as an example of a semi-structured data set, which is a graph with nodes connected by labeled edges.

XML query languages are mainly for data extraction, transformation, and integration. For any query language, the *query data model* and the *syntax* are the most important and they are closely related.

data models are deployed for modeling XML document data and also for formulating the queries over the data. For example, in XML-QL, an XML document is modeled as a variation of the semistructured data model, represented as a graph. It is composed of nodes and edges and has a unique *root*. Edges are labeled with

element identifiers, nodes are labeled with sets of attribute-value pairs. Leaves are labeled with value strings.

Lore is a database management system for semistructured data [32]. In the Lore Project [32], the Object Exchange Model (OEM) is used to support the XML Query language. OEM is an self-describing, nested object model which can be regarded as a labeled directed graph. To handle XML documents, it made several extensions to the model, so that the XML documents can be mapped into a directed, labeled ordered graph. In this extended model, an XML element is an identifier-value pair represented as a node in the graph, and an edge represents the element-subelement relationship.

Since XML data have characteristics similar to both database and semistructured data, XML query languages may exhibit mixed syntax features from both database fields and the semistructured community. The difference of the data model can be seen from the syntax. Some XML languages borrowed the syntax from database framework and made modifications to meet the needs of XML. For example, many of the query languages now take the notion of XML tag syntax. There are still some other early work which has a great influence on the development of the W3C working draft. For a complete view, we also mention them in the following part.

XPath *XPath* [61], which stands for the *XML Path Language* is a string syntax for building addresses to the information found in an XML document. Originally XPath was designed as a helper language for XSL, the Extensible Style sheet Language, which is a rendering vocabulary describing the semantics of formatting information for different media.

It is also regarded as a prototypical, special purpose XML query language. It is widely used for its simple syntax and its convenient way to perform regular expression. Many later query languages accepted it as part of their syntax to express the navigation through the XML documents. The syntax of XPath is defined in Table 3.1:

Addressing information is only one aspect of querying information. As a query language, it still lacks some other useful aspects. For example, XPath does not allow

bib	matches a bib element
*	matches any element
/	matches the root element
bib/book	matches a book element following a bib element
bib//author	matches an author following a bib element at any depth
//title	matches a title at any depth
book/@ year	matches a year attribute of element book

Table 3.1: Syntax of XPath

constructing elements that are not in the original XML document, or sorting the result or querying document pieces from different XML documents.

XSL *XSL* [62] - *Extensible Styles Language* has facilities that could serve as a basis for an XML query language. XSL uses an XML-based syntax. Its syntax is described as the following. Reserved words or symbols are in bold font. The braces denote the part that may occur zero to many times.

```
<xsl: template [match PatternExpression ] >
{<ResultElement>}
{<xsl: Directives>}
{</ResultElement>}
</xsl: template>
```

Here the `PatternExpression` stands for an expression written in XPath. `Directives` are actions to be taken when the pattern is matched. `ResultElement` is used to specify the elements' tags created in the query result.

XSL style sheet uses `template` rules to express their query. By matching the template with the document content, the query executes corresponding actions to get the result.

```
<xsl:  template match="bib[book/price>=30">
   <title>
      <xsl:  value of select="title">
   </title>
</xsl:template>
```

The `xsl:value of` directive extracts information from a pattern expression and converts it into a string. XSL's strength lies in its support for document transformation: they allow cloning the queried documents and constructing result by creating new elements.

Lorel *Lorel* [7] was originally designed as a semistructured data query language for the Lore project at Stanford University. It uses a `SELECT-FROM-WHERE` clause to represent the query request. The syntax is similar to an SQL statement:

select {SelectExpression}
[**from** {FromExpression}]
[**where** {WhereExpression}]

The `select` clause contains the constructor, `from` clause and `where` clause specify the pattern and the filter constraint. The select expression, from expression and where expression can contain subqueries. A query is expressed as follows:

```
select B.title
from bib.book B
where B.price >= 30
```

XQL *XQL* [63], which stands for *XML Query Language* can be considered as an extension of XSL pattern syntax. The basic constructs of XQL correspond directly to the basic structures of XML, queries, transformation patterns and links are all based on patterns in structures found in possible XML documents.

XQL's syntax has some similarities with URI(Universal Resource Identifier) directory navigation, it uses single slash and double slash to traverse the XML hierarchy. XQL considers a XML document as an ordered, labeled tree, with nodes to represent the document entity, elements, attributes, processing instructions, and comments. An example query looks like the following :

```
bib [book/price >=30 ]
```

XML-QL *XML-QL* [24] is a prototypical XML query language designed at AT&T Labs. Like SQL or OQL, XML-QL allows nested queries. Seen from the language syntax, `QueryBlock` has two parts: a `WHERE` clause and a `CONSTRUCT` clause. `QueryBlock`

is recursively defined by specifying the clause CONSTRUCT also as a QueryBlock. A XML-QL usually is composed of one or more query blocks.

The basic abstraction of the XML-QL is as follows:

```
QueryBlock →
      WHERE Condition
      CONSTRUCT { QueryBlock }
```

Even though XML-QL has a WHERE construct, and a CONSTRUCT clause for building the query result from the query, it resembles an XML document more than a SQL statement. Pattern and filters are in the WHERE clause. For example, a query to find books with price above $30 is as follows:

```
WHERE <bib>
      <book>
         <price>$p</price>
      </book>
      </bib> ELEMENT_AS $r in bib.xml, $p>=30
CONSTRUCT $r
```

SXQL *SXQL - Structured XML Query Language* has a similar syntax with database SQL. It is compatible with the SQL-99 standard. It adopted a lot from SQL.

XQuery XQuery is derived from a query language called Quilt. It is a functional language where a query is represented as an FLWR expression, composed of FOR, LET WHERE, and RETURN clauses.

The syntax is as follows:

```
{FOR ForExpression}
{LET LetExpression}
[WHERE WhereExpression]
RETURN ReturnExpression
```

The FOR clauses and LET clauses serve to bind values to variables. Further filtering is done on each binding tuple generated by FOR and LET with an optional WHERE

clause. The RETURN clause generates the output of the FLWR expression with the bound variables.

```
<results>
   LET $doc := document("prices.xml")
   FOR $t IN distinct ($doc/book/title)
   LET $p := $doc/book[ title = $t]/price
   WHERE $p >= 30
   RETURN $t
</results>
```

Many of the existing query languages borrow from the database query language SQL and express the query over XML documents in a similar way. In our view, the main problem with existing query languages is that they mix querying and result constructing, which is an inherited issue from SQL. In order to deal with a complex query, nested queries have to be used in those query languages. Also because of the complex nature of XML, when compounded with other constructs, it is cumbersome in these languages to express complicated queries and also hard to comprehend. Concentrating more on the data centric XML applications, this thesis discusses a solution to the problem of XML query processing from a different angle. The query system we implement is intended to integrate the ideas from logic programming and databases, and allows the expression of XML queries with fewer lines and in a more readable manner.

3.3 A Rule-based Query Language for XML

In order to get an insight of strength of the XML-RL query languages, in this section, we take a further look at XML-RL from its syntax, how to model XML in XML-RL, and the features of the language, and demonstrate its capabilities to query XML data.

3.3.1 Model for XML-RL

"The purpose of models is not to fit the data but to sharpen the questions." [55]. Dr. Liu and Dr. Ling proposed a novel data model [47, 48] for XML that allows XML data to be viewed in a way similar to the complex object data model [6]. To make the syntactical features more meaningful and clear, we extend the model by bringing some new notions: *lexical object*, *attribute object*, *element object*, *tuple object*, *list object*.

As the fast emerging and dominant standard for data representation and exchange on the Web, we need to find some way to extract, synthesize, and analyze XML contents in a powerful and easily understandable way. Thus we require a high-level and a declarative fashion to access XML data. We resorted to declarative languages to get the solution.

The XML-RL model describes and views XML using objects. In this model, every component in XML language is treated as a type of *objects*. We use part of bib.xml (see Appendix A) as an instance to explain how an XML is represented using our model.

```
<book year="2000">
   <title>data on the Web</title>
   <author> <last>Abiteboul</last > <first>Serge</first> </author>
   <author> <last>Buneman</last> <first>Peter</first> </author>
   <author> <last>Suciu</last> <first>Dan</first> </author>
   <publisher>Morgan Kaufmann Publishers</publisher>
   <price>39.95</price>
</book>
```

Element is an important building block in XML document. In the XML-RL model, it is described as an *element object*, which is a name-value pair, with a string describing its element name and another object denotes the element's value. Examples are:

```
title:data on the Web
last:Buneman
price:39.95
```

Here the element tags, `title`, `last` and `price` and the text content of the elements such as `data on the Web`, `Buneman`, each is represented by a string of characters, which we call *lexical object.*

An attribute in XML is also represented with an *attribute object.* It is an attribute name-value pair, similar to element objects. As in other proposals [61, 7, 63] for XML querying, we also use the symbol @ in front of attributes to differentiate them from elements. As is seen, the year attribute is represented with `@year:2000`

A *tuple object* is used to describe an XML element's attributes and content. Its attributes denoted by attribute objects, together with its content which may include sub-elements denoted by element objects or plain text denoted by lexical object, are enclosed using brackets ([ ]).

For instance, the book element's attributes along with its contents can be denoted by a tuple object as follows:

```
[ @year:  2000,
title:  data on the Web,
author:  [ last:Abiteboul, first:Serge ],
author:  [ last:Buneman, first:Suciu ],
author:  [ last:Suciu, first:Dan ],
publisher:  Morgan Kaufmann Publishers,
price:  39.95 ]
```

Notice the first author element has no attribute and its content is denoted by [`last:Abiteboul, first:Serge`]. If an element has no attribute and its content is a lexical object, we can omit the brackets, which is in fact a lexical object; for example, element `title`'s and `publisher`'s content are represented as lexical objects.

If a tuple object contains a collection of elements with the same tag, such as `author` illustrated in the above example, and we are interested in all of its values, it can be represented as a *list object.* In other words, it may have two kinds of representation: a *flat representation*, with each element corresponding to an element object; or a *integrated representation*, with its attribute or elements' values represented by a list object instead.

Thus the tuple object can also be represented as follows:

```
[ @year:  2000,
title:  data on the Web,
author:  { [ last:Abiteboul, first:Serge ],
         [ last:Buneman, first:Suciu ],
         [ last:Suciu, first:Dan ] },
publisher:  Morgan Kaufmann Publishers,
price:  39.95 ]
```

Therefore, conversion between XML data and our model is straightforward.

Consider another example with DTD and its corresponding XML document with ID, IDREF and IDREFS attributes:

```
<!ELEMENT people (person+) >
<!ELEMENT person (name) >
<!ELEMENT name (♯ PCDATA >
<!ATTLIST person
         id ID ♯ REQUIRED
         mother IDREF ♯ REQUIRED
         children IDREFs ♯ REQUIRED >

<people>
   <person id ="o123"> <name> Jane </name> </person>
   <person id ="o234" mother="o456">
      <name> John </name>
   </person>
   <person id="o456" children="o123,o234">
      <name> Mary </name>
   </person>
< /people>
```

The above XML document is represented with XML-RL model as follows:

```
people:[ person:
```

```
{ [@id:o123, name:Jane],
   [@id:o234, @mother:  o456, name:John]
   [@id:o456, @children:{o123,o234}, name:Mary ] } ]
```

3.3.2 Syntax for XML-RL

XML-RL, which stands for *XML Rule-based query language* is a declarative language, which is similar to logic programming language in many ways [47, 48, 49]. For example, like the Prolog language which is a logic programming language, our query is made up of rules. Each rule consists two parts: the *rule body* and the *rule head.* The rule body specifies the source of the information. It employs the mechanism of variable binding to get the result. By binding each variable with the appropriate value retrieved from the XML document, XML-RL builds the result consistent to the structure specified in the rule head.

However some of the features make XML-RL different from those declarative languages. First, XML-RL uses *XPath* to specify the XML content. It is tailored for query languages for XML documents. Besides the common use of *content variable* for retrieving information in the content, the *structure variable* is employed in XML-RL to query the structural information of the XML document,

From data's view point, XML documents are considered as a deductive database, composed of two parts: the Extensional database (EDB) which stores the facts, and the Intentional database (IDB) is created by applying the rules on the EDB. This also leads to another important difference, the mechanism by which we bind the variables. In Prolog, after each variable in the body is bound using a fact, a result is obtained instantly in the head. But in XML-RL query rules, because reconstructions, such as getting minimum or maximum value and grouping are needed in order to create the result, the head of the rule is not decided until all necessary information for building the result is found. This is more of a database-like way to look at and manipulate the data in XML.

Variables Our objective for XML-RL is to extract data from XML documents in a declarative way, and we use logical variables for querying and constructing the result.

Based on the model we use, we need the variables to represent different objects: lexical object, attribute object, element object, list object, tuple object. As those objects have two kinds of representation, flat representation and integrated representation, we partition the variables into two kinds as well: *single-valued* variables and *list-valued* variables. A single-valued variable starts with a dollar sign ($); for example, `$a` may be represented either a *lexical object*, an *element object*, or a *tuple object* depending on the context. Its value denotes a possible binding to the element or attribute that the variable stands for. A list-valued variable is represented in a form `$$X`. Consider the following two examples.

Example 1 If we want to list all the title-author pairs, with each pair enclosed in a "result" element. The XML-RL query is expressed as follows:

```
bib.xml%/bib/book :$b,
$b/title:$t,
$b/author:$a
⇒ result.xml%/results/result:[ title:$t, author:$a ];
```

In the input XML *bib.xml*, the authors are grouped under the book title. This query requires each title and author pair, one-to-one listed separately. This query requires to flatten the nested structure and return title author as a pair. In XML-RL, a variable started with $ stands for each single element in a list.

Example 2 We want to list the title and authors inside a "result" element for each book in the bibliography.

```
bib.xml%/bib/book:$b,
$b/title:$t,
$b/author:$$a
⇒ result.xml%/results/result:[ title:$t, author:$$a ];
```

This query uses `$b/author:$$a`, which treats all the authors as a list-valued, as we are interested in all of its values, no matter there is one author or more than one in the list. It differs from `$b/author:$a`, where `$a` denotes each author or one author at a time.

In previous project of deductive database system RLOG [51], we had used a different mechanism where a same variable can represent both multiple values and

single value. For example, a variable `$a` can represent a single value if there is only one value in the database. Otherwise it represents multiple values.

Although the RLOG system is also rule-based and has a similar mechanism to construct result by binding variables, a very important difference from XML-RL system is that RLOG system is a typed database system. In the RLOG system, a variable is mainly a place holder, and its meaning can be decided by referring to the schema, while in XML-RL system, DTD may provide meta information just as schema does for database. However since a DTD is optional, and XML data structures often contain sequences where repetitive, optional and fixed parts are mixed together, the relationship between the DTD and its XML documents is not as tight as schema coupled with databases.

In XML-RL a variable is also used to specify the path for an element object. For example, `$a/last:$l`, `$a` specifies that `last:$l` is an element has the path with a variable `$a` bound to a certain value. This use of variable makes it necessary to distinguish the variables standing for multiple values from those for a singleton value. If `$a` stood for a collection of authors as well as an author, it would be intricate to figure out what `$a/last:$l` represents. Therefore, in XML-RL list-value variable and single-value variable are used respectively in different contexts to avoid ambiguity.

To query structural information, we introduced another type of variable, called a *structural variable*, which begins with ampersand (&). The structural variable stands for the element name or attribute name in the input document. The value of this kind of variable can only be a *lexical object.* In contrast, we refer to those variables starting with either $ or $$ as *content variables.* Here is an example.

Example 3 If we want to find books in which some element has a tag ending with "or" and the same element contains the string "Suciu" at any level of nesting. For each such book, return the title and the qualifying element.

```
bib.xml%//book:$b,
$b/&e(endwith "or"):$v(contain "Suciu"),
$b/title:$t
⇒ result.xml%/book:[ title:$t, &e:$v ];
```

As illustrated in this example, &e is a structure variable. It represents the tag for

the element $v whose content contains characters "Suciu".

Query and Result Construction In XML-RL language, a fundamental idea is to use the mechanism of variable binding to find the answers to query and then build the result into a desired form. In the following, we will demonstrate how to use variables and expressions to obtain information from XML documents and construct the result.

A XML-RL *query* is made up of a set of XML-RL *rules*. Each *rule* has two parts the *rule head* and the *rule body* connected by an arrow symbol ($\Rightarrow$). This provides a natural way for separating the querying portion and result constructing portion.

The *rule body* is used to query data or structure in XML documents. As in Example 1, its rule body has a form as follows:

```
bib.xml%/bib/book :$b,
$b/title:$t,
$b/author:$a
```

We include the syntax of XPath [61] and incorporate logic variables to navigate the hierarchy of XML documents. As the above shows, the first clause specifies the context of the query, then consequent query expressions use those defined variables to specify their paths. Using variable in the path expression gives much convenience especially when the path is repeatedly used in successive expressions. It improves the readability as well. Consider XML's hierarchy structure as an onion-like structure; following the variable bindings to explore the hierarchy is like unwrapping the skins of an onion, in a sense that the variable is bound to a value. The unknown inner part is treated as a whole.

Arithmetic and logic string comparison expressions can also be used as *constraints* in the rule body to eliminate undesired values in the input XML. Currently XML-RL provides many commonly used string comparison as constraints, for example, `endwith`, `beginwith`, `contain`. Here is an example.

Example 4

```
bib.xml%/bib/book:  $b,
$b/publisher:  $p(!="Addison-Wesley")
$b/@year:  $y(>1991),
```

```
$b/title:  $t
⇒ /bib/book:  groupby($a)[@year:$a, title:$t]
```

In the example, (>1991) and (!="Addison-Welsley") are constraints. The constraint expressions are put right after each specification expression, enclosed within parenthesis. This makes the syntax more compact and easier to read. The constraints are optional.

The *rule head* specifies how the result XML document is going to be constructed. Operators and functions can be used for constructing the result, which we refer to as the *constructive expression.* XML-RL supports arithmetic operators like `MULT`, `MINUS`, `DIV`, `PLUS`, and collection operators `UNION`, `INTERSECT`, `DIFF`. Functions are also supported, such as `sublist`, `MIN`, `MAX`, `AVG` and so on. In other words, XML-RL provides a method to create new elements in the result, by calculating, grouping, reordering values obtained from the XML documents.

Constructive expressions are different from the constraint expressions in a query. Constraint expressions are used to prevent the unqualified data from appearing in the result, without changing their values or structure in the input documents. For example, in Example 4, `$b/@year:$y(>1991)`, the constraint expression eliminate those with year attribute value smaller than 1991. The constructive expressions help to create new structures through the bound variables coming from the rule body. In the example mentioned, `groupby($a)` is a constructive expression.

Declarative Way of Querying As a rule based language, unlike other XML query languages, such as XML-QL, XQuery, or XQL, where complex queries are expressed in a nested form with queries containing subqueries, XML-RL takes a rather *"flat"* form. Take XQuery for example, the following query lists the author's name and the titles of all books by that author, for each author in the bibliography (bib.xml), grouped inside a "result" element.

```
<results>
      FOR $a IN distinct(document("http://www.bn.com")/author)
         RETURN
         <result>
```

```
        $a,
        FOR $b IN document("http://www.bn.com")/bib/book
           [author = $a]
        RETURN $b/title
        </result>
</results>
```

This requires the users to be well aware of the data retrieving procedure, such as how to iterate a collection to get the data. When the query is very complex, understanding the procedure will become a burden to the users.

The most distinct feature of declarative languages is to describe *what* the goal is rather than to provide the details on *how* to obtain it. In other words, the procedure needs not to be specified in the query. It is important for the users, since they do not have to care much about the underlying details of how a query is carried out, issues like when to use a loop in order to get each object in a list or to build a list of object through an iterator. The same query is expressed in XML-RL, as follows.

Example 5

```
bib.xml%/bib/book:  $b,
$b/title:  $t,
$b/author:  $a,
⇒ results/result:  groupby($a)[author:$a, title:$t]
```

Thus complex queries can be expressed in XML-RL in a simple and natural way.

Besides, in XML-RL the variables are effective only within each rule. This means in a rule, variables of a same name are uniquely bound; same named variables used in different rules are irrelevant. If a query has several rules, the variables appear in different rules are regarded as independent.

3.3.3 DTD and XML-RL

To reach XML's full potential of usefulness, there must be an agreement among inter-operating systems [64, 33]. A DTD [59] is considered as an agreement among XML documents sharing the same DTD. The DTD becomes more and more important in data exchange, and information integration. It describes the structure of a

document, types of data the document contains as well as the relation in a framework for the elements: some elements must have a single value, some may have more than one value, while others may be allowed to be empty. These descriptions help ensure that the XML document have a proper structure.

DTD can be used in two possible ways in a query language. One is to use a DTD to perform a structural validation check and assist in formulating the queries. In this case, the query strongly depends on the DTD, just as a database query is depending on its schema. In the database environment, users consult the schema in order to issue a meaningful query against the database. Using a DTD for an XML query in a way analog to a schema in a database is described in Figure 3.2.

However some factors make the "tight coupling" way of using DTD inconvenient in the XML world. Firstly, even though DTD is very popular and well-supported, many non-valid XML documents still exist, i.e. not all of the XML documents have DTDs. Secondly, as a file system, XML has its nature and flexibility in representing the data. Accordingly the XML DTD is less restrictive compared with database schema, and it permits data to have more variations. Some elements are optional, and some may have multiple values. Besides, there are other alternative techniques for defining XML meta information, such as XML schema [60] and the techniques are evolving quickly.

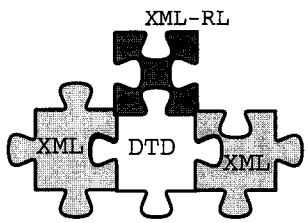


Figure 3.2: The Tight Coupling Between XML and DTD

Our system supports the validation features by conducting validation checking. It is optional. XML-RL allows users to formulate the query based on semantics of the XML query itself, as opposed to requiring users to issue queries in strict conformance

with the DTD. This way, XML query is less dependent on the DTD, as illustrated in Figure 3.3.

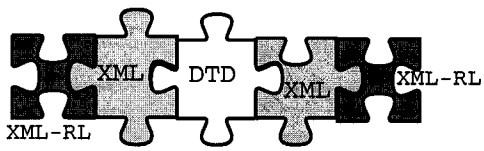


Figure 3.3: The Loose Coupling Between XML and DTD

It also provides more adaptability for system implementation when technologies evolve: if an existing DTD now is replaced by an XML schema, the users will not necessarily learn new technologies in order to issue the same query and the underlying processing program needs not change too much to adapt to the change. To support it, a uniformed variable representing method (refer to Section 3.3.2) and a consistent query processing based upon the representation are provided.

3.4 Summary of the Chapter

Both database and document communities are very actively researching XML, and several query languages have been proposed to bring the strength of the database and semistructure documents to extract the information from an XML documents. This chapter gives a brief overview of how these XML languages present queries. Then it introduces our XML-RL language, its syntax, data model, and describes with examples its capability to extract, transform and integrate information from an XML document.

Chapter 4

Design and Implementation of The XML-RL System

This chapter presents the design and implementation of our rule-based XML query system. This system has been developed under Line using Java©1.3.1. and also runs on Sun Solaris©2.7. Other development tools involved are: CUP©version 0.10, which is a system for generating parsers written in Java©, and JLEX©1.25, which is a lexical analyzer generator for Java. To evaluate its query capabilities from different perspectives, we also employed tens of queries among which are the typical queries in Query Use Cases [4].

This chapter discusses several issues we encountered while implementing XML-RL and how they were solved. Section 4.1 describes the system architecture. Then according to the system architecture, each component of the system is discussed specifically. Section 4.2 is focused on how XML documents are accessed and manipulated in XML-RL system. Section 4.3 discusses how an XML-RL query statement is analyzed and processed. Section 4.4 discusses the design and implementation of the classes in the query process control, query data management and result construction.

4.1 System Architecture

We first present a brief architectural overview. The architecture of XML-RL query system is shown in Figure 4.2. The *Query Interface* is the outermost part. It

is currently a textual interface, serving as the inter-medium between users and the XML query system. Running the query is very simple. We stored some queries in files, and we use the Java command to run the query under the textual interface. As seen in Figure 4.1, the command line is:

`java mmm.XMLQuery` $< q_1$

The result is displayed on the screen in XML format. It can be stored in an XML document as output as well.

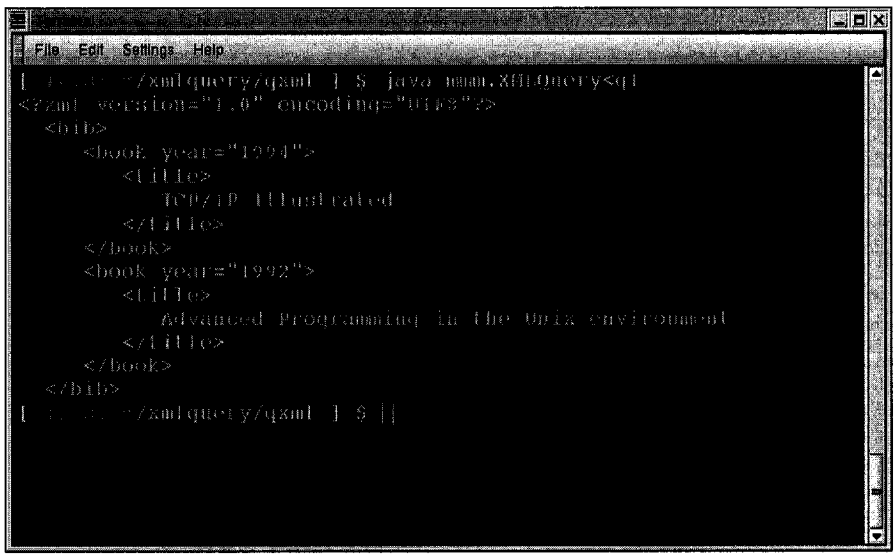


Figure 4.1: The Programs and Queries

This interface accepts a user query and passes it onto the Query Preprocessor. When the query result is found, the Query Interface displays it to the user.

The core component of the query system consists of the Query Preprocessor, the XML-RL Query Manager and the Information Retriever. The *Query Preprocessor* is responsible for checking the syntax and semantics. Firstly, the Query Preprocessor checks the syntactic correctness of the query and then with the assistance of the Information Retriever, performs semantic validation.

In cases when a query involves several XML documents, the *Query Manager* takes care of the communication among query sources and controls the query process. It

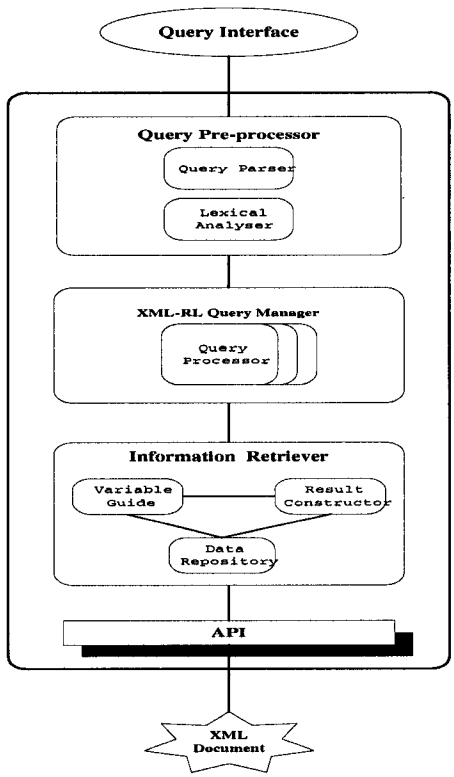


Figure 4.2: XML-RL System Architecture

deals with communication in a query level across different XML documents. Each Query Manager has at least one *Query Processor.* If a query involves several XML sources, e.g., comparison between variables belonging to two XML documents, a Query Manager is responsible for contacting the Query Processors. The *Query Processor* is in charge of the variables in the Information Retriever.

The *Information Retriever* has three parts: the *Variable Guide*, the *data Repository* and the *Result Constructor.* A more detailed explanation on each part is in Section 4.

The Information Retriever accepts the valid query and processes it according to the data model, and translates it into an API (Application Programming Interface) request.

The API interfaces with the XML documents. When a request from the Information Retriever is received, the XML parser loads the XML document, parses it, and make it ready for the Information Retriever to access. Then the Information Retriever gathers the XML data and the structural information, takes care of any necessary calculation, e.g., when a query has an arithmetic or deductive operation, or a query requires a transformation to new structure or an integration of XML data from multiple resources.

After that, the Information Retriever processes and assembles the information according to XML-RL data model. Error messages if any at the processing stage are passed to the users through the Query Interface.

4.2 Access and Manipulation on XML

4.2.1 JAVA for XML Querying

XML-RL query system is implemented using Java. Many of Java's features make it favorite for our system. First, software written in Java can run on different platforms, with a Java Virtual Machine, such as Unix, Window, Linux or Macintosh. In our case, the system is developed under Linux. When migrating it to a Sun workstation running the Solaris operating system with Java on it, we have no difficulty compiling the program after environment variables are set properly. It works as well as it executes under Linux.

Java's features that support an application for the Internet are also beneficial to the implementation. Java expands the universe of objects that can move around freely on the Internet. Besides the passive data which is designed to be viewed or browsed, the dynamic and active data can be transmitted to the client computer from the server side with Java. For example the Java applet can react to user input and change the data dynamically. In our case, even though so far we have not used this feature, it leaves us the possibility of implementing the query system as a Java Applet in the future, which can be accessed by the web browsers through the Web.

4.2.2 Programming Interfaces to XML

When we began to design the system, the first question we met is how to access XML in our query language. One way is to transform XML and store data in another form, such as a relational table or objects under database management and manipulate them directly in our query language.

Since XML was originally designed to be an exchange format for data over the Internet and in industry, another choice is to keep its format, then access the document through a processor with a parser as its key part. There are mainly two application interfaces (APIs) available over the Internet that are widely used by XML applications: *Simple API for XML* (*SAX*), and *Document Object Model* (*DOM*) [35]. As Figure 4.3 illustrates, SAX or DOM acts as an intermediate layer between applications and the XML documents being processed. Using APIs to access XML provides standard interfaces with other XML application and tools, thus giving more flexibility in implementation and also reducing our development time.

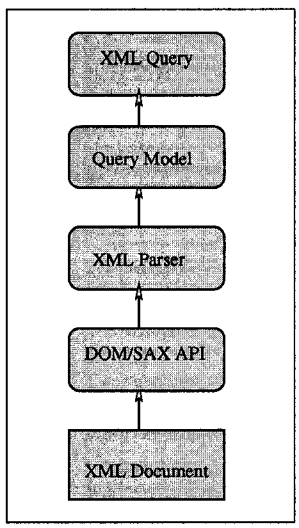


Figure 4.3: Programming Interfaces for XML Application

SAX provides a serial mechanism for accessing XML documents. It allows parsers to read through a document in a linear way, and then creates an event handler every

time a markup event occurs.

DOM is a standard tree-based API for XML and HTML documents. It compiles an XML document into a tree structure. DOM has a set of interfaces that provide programming access to the nodes of a document, beginning with the document's root node.

Document is the top level object that represents an XML document. *Document* holds the data as a tree of *Nodes*, where *Node* is a base type that can be an element, an attribute, or some other type of content. There are five objects defined by DOM: *DomDocument*, *DomNode*, *DomAttribute*, *DomDtd*, and *DomNamespace*.

Figure 4.4 illustrates a simple example XML document called "bib_exmp.xml" represented as a DOM tree.

```
<bib>
   <book year=''2000''>
      <title>XML and HTML</title>
      <author> <last>Lee</last> <first>Sara</first> </author>
      <author> <last>Brick</last> <first>Peter</first> </author>
   </book>
   <book year=''2002''>
      <title>Query on the Web</title>
      <author> <last>Lee</last> <first>Sara</first> </author>
   </book>
</bib>
```

We will use this XML example through the later sections to explain issues in class design and query processing. The root element is the node denoted as "bib".

DOM API and SAX API employ different philosophies for parsing. Their differences can be seen from the following aspects:

Ways to Obtain Information From the application point of view, to get information from a DOM parser, the application can go back and forth following the object reference from one to another in the memory. With SAX, the application will be

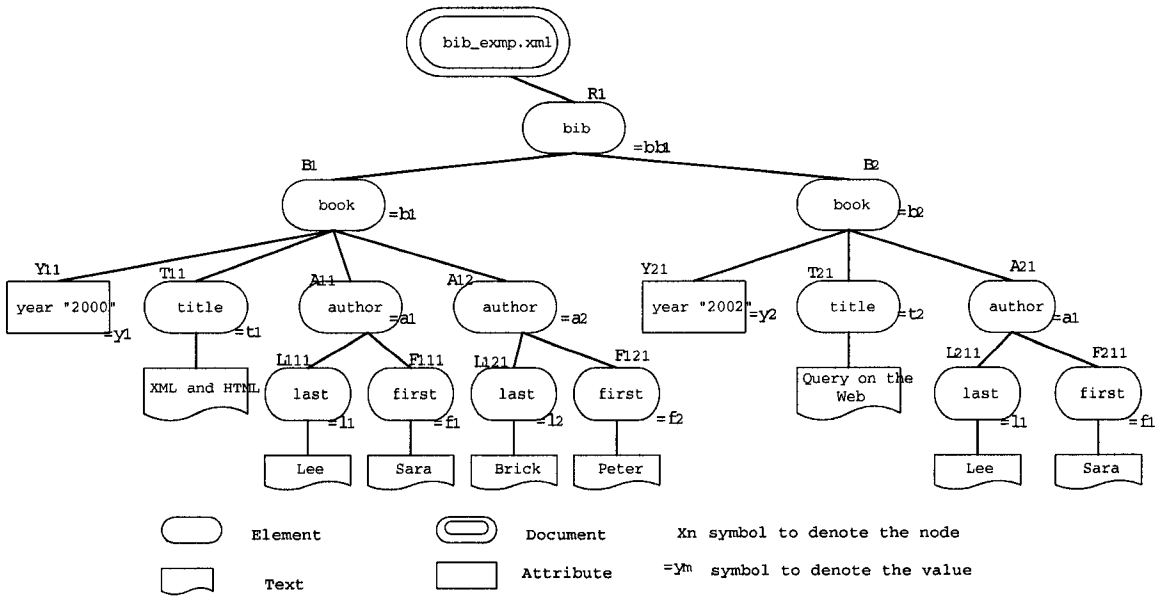


Figure 4.4: An XML Document for Query

notified by the parser with a series of parsing events, like reaching the end of element, starting from the document, etc.

Consider the following Java code using the SAX API to print the attribute of an element. Notice SAX handles XML as a stream of XML parser events, which is how it reacts when it meets an element start tag and an end tag.

```
public void startElement(String name, AttributeList attrs)
throw SAXException
{
int attLen = 0;
System.out.print("<" + name);
if(attrs != null && ((attLen = attrs.getLength()>0) {
   for (int i=0; i<attLen; i++ ) {
      System.out.print (" " + attrs.getName(i));
      System.out.print ("" + "='" + attrs.getValue(i)+ "'"); } }
System.out.print(">") }
```

```
public void endElement(String name)
throw SAX Exception
{ System.out.print("</" + name + ">"); }
```

The DOM API handles an XML document as an document tree. It has different kinds of tree nodes and related methods to interact with these tree nodes.

```
void print(Node node, PrintStream out) {
   int type = node.getNodeType();
   if (type = Node.ELEMENT_NODE)
   {
      System.out.print("<" + node.getNodeName());
      NamedNodeMap attrs = node.getAttributes();
      int len = attrs.getLength();
      for (int i=0; i<len; i++) {
         Attr attr = (Attr)attrs.item(i);
         System.out.print(" " + attr.getNodeName() + "='" +
         escapeXML(attr.getNodeValue()) + "'");
         }
      System.out.print(">"); } } }
```

Parsing Mechanisms From the parser's viewpoint, SAX and DOM also differ. SAX uses sequential parsing. It will parse the XML document by handling events. The DOM parser processes and manipulates XML by traversing the object tree [5]. That is, the difference between SAX and DOM is the difference between sequential, read-only access and random, read-write access.

Flexibility and Extensibility Each API has its own merits and drawbacks. Since the DOM API builds an in-memory parse tree, which makes queries and transformations easy, however, resource usage increases as the size of the document grows. On the other hand, event-based SAX requires fewer resources, since it only needs to deal with each piece of XML document sequentially. However, it makes arbitrary queries and transformations difficult.

As a query language, XML-RL needs the ability to access each part of an XML document easily as well as to transform it into some different structured result. Therefore after studying both the features of SAX and DOM, and our system needs, we decided to use the DOM API for our implementation as DOM API is ideal for interactive applications.

4.2.3 XML Parser

The XML parser is responsible for analyzing XML markup and the structure of the document. With APIs it is not hard to build an XML parser of our own; however there are already many existing ones that are widely used and proved to be efficient. In the fast evolving world of XML, an XML parser is said to be the most mature application. There are a wide choices of parsers available. Table 4.1 provides a list and comparison of them.

Name	*Company Person*	*Validating Nonvalidating*	*API Support*	*Coding Language*	*Other Features*
JAXP	Sun	Both	SAX 1.0/DOM	Java	faster
XML4J	IBM	validating	DOM	Java	
XP	J. Clark	nonvalidating	SAX 1.0	Java/C	with javadoc
Ælfred	Microstar	nonvalidating	SAX 1.0	Java	small
SXP	Silfide	validating		Java	extended QL
Aparche	Xerces		SAX 2.0/DOM	Java	
DXP	Microsoft	validating			XML support

Table 4.1: A Comparison Over Different XML Parsers

Several aspects were taken into account before we finally chose *Java API for XML Parsing* (*JAXP*) as the appropriate XML parser for XML-RL's implementation:

Processing Speed An XML parser's speed is the main concern in implementing XML-RL query system. According to Sun's testing result using JDK 1.1.6, Sun's

JAXP validating parser is significantly faster than the majority of the non-validating parsers tested and all-validating parsers. A validating parser is a parser that is able to perform structural validation on an XML documents.

Supported Features As we mentioned previously, SAX and DOM are the two programming interfaces to XML. DOM API is ideal for an interactive application, which is the case for our query system. Usually most parsers support one of them, for example, *XML4J* supports DOM while XP and *Ælfred* supports SAX. Only some of them support both, among which are *Apache* and *JAXP*. Also, *JAXP* has non-validating parser as well as a validating parser. This provides greater flexibility in developing our application. In the Java programming language, we can instantiate the parsers by using *JAXP*.

Good Documentation Good documentation is also beneficial in order for us to get a quick start and saves us time when some problem comes up. In comparison with other XML parsers, *JAXP* provides full documentations on installation, specification of classes, and even FAQ. There are other parsers that are well documented, such as IBM's *XML4J* and James Clark's *XP*.

4.2.4 Dealing with XML Meta Information

As XML becomes widely used throughout the Internet and industry, many XML Schema languages have been proposed to establish agreements and standards for processing and exchanging data in XML format. XML DTD based on SGML DTD is the standard of XML schema language of the past and present before XML schema arrives. Element and attribute are its primary building blocks. There were also some other efforts from different organizations: *XML schema* [60] by W3C, *XML data Reduced (XDR)* by Microsoft, *Document Structure Description (DSD)* [40] by AT&T, *Schematron* by Rick Jelliffe [42] .

XML is a very active research area and subjected to fast changes. Therefore, in our implementation, we try to keep some process relatively independent. Instead of applying the meta information directly to formulate the semantics of query language,

the XML-RL system only employs the schema information for validation. The validation procedure is separated from XML query processing. In this way, we are able to maintain a system's consistency as well as its readiness for changes with more ease. By explicitly specifying the meaning of each component of the query language, and putting validating and querying into separate procedures, we eliminate the tight coupling between the schema information and the query itself. The interpretation to the XML-RL query is not changeable as the schema information varies from one document to another. For example, a variable `author:$x` always stands for each single value, no matter whether the schema specifies author as single-valued or multiple-valued. The meaning of the query will not change according to the data being queried. In the XML-RL system, on one hand, the schema information ensures the validation of document when required and on the other hand it provides the users with more flexibility in writing queries.

4.3 Process for the XML-RL Query Statement

This section discusses how an XML-RL query statement is analyzed based on XML-RL syntax. Next we discuss how our data model works in between the query statement and XML input/output to accomplish the query.

4.3.1 The Query Parser and The Lexical Analyzer

The Query Preprocessor composed of the Query Parser and the Lexical Analyzer is mainly used to handle user query requests in XML-RL. By analyzing the query statement, it communicates to other components on what information needs to be retrieved and how the result should look. In implementing XML-RL, we used *JCUP*, an abbreviation for *Java based Constructor of Useful Parsers* and *JLEX*, an abbreviation for *Lexical Analyzer Generator for Java.*

JLEX is written for Java to create a lexical analyzer. It is similar to the well-known lexical analyzer LEX [44] for the UNIX operating system written in C. The JLEX utility is based upon the LEX lexical analyzer generator model. It takes a specification file, then creates a Java source file for the corresponding lexical analyzer. The lexical

analyzer works as the scanner for parser. First we use JLEX to construct a lexical analyzer to break characters up into meaningful tokens, such as keywords, numbers, and special symbols.

JCUP is a system for generating LALR(Look Ahead Left to Right Parsing) parsers from specifications. It serves the same role as the widely used program YACC and in fact offers most of the features of YACC(Yet Another Compiler-Compiler) [1]. We create a specification based on our query language grammar using JCUP. A piece of parser code looks like the following:

```
SingleVar ::= DOLLAR IDENTIFIER:id
      {:
         RESULT= "$" + id;
      :}
      ;
```

This shows when the XML-RL parser recognizes a single value variable, it returns a value in `RESULT`. The `::=` is used to define the syntax of a language. This forms part of our XML-RL query. Single variable (`SingleVar`) is a non-terminal, `DOLLAR` and `IDENTIFIER` are terminals. In `IDENTIFIER:id`, `id` is a variable which stores the value `IDENTIFIER` represents. Within the braces are the actions that the query parser will take when the query is parsed according to the syntax.

In our system, we separate the processing of the query part of the XML-RL rule from the result construct part. And we put the classes for the querying part and the result part into two Java Packages, *LeftRules* and *RightRules* respectively. The lexical file for the query is *leftrules.lex* and the parser file is *leftrules.cup*. The lexical file for result construction is *rightrules.lex* and parser file is *rightrules.cup*. Lexical files contain the regular expressions and actions. Parser files contain the grammar and actions. Running *rightrules.cup* through CUP produces files *rightparser.java* and *rightsym.java* which in turn generate *rightparser.class* and *righsym.class* that have constants and tokens which the lexical analyser should return. Similarly we can get *leftparser.class* and *leftsym.class*. Running lexical file: *leftrule.lex*, *rightrule.lex* through JLEX produces the leftrule.lex.java and rightrule.lex.java respectively. The lexical analyser actions return symbols, where the symbol type will be one of those

named in the *rightSym.class*, class *rightYylex.class* respectively.

Since both JCUP and JLEX are written for Java and in Java, Java code can be embedded in JLEX and JCUP. It produces the tokenizer and the parser which are implemented in Java. The syntax of our language expressed in BNF is in Appendix B.

4.3.2 The Data Model for Query Processing

As an application tool, the XML-RL query language provides an expressive and simple tool to represent the request for retrieving information from the XML documents.

We view XML data in a similar way to complex objects in the complex object data model. Figure 4.5 illustrates how we describe and view XML document in Section 4.2.2 from a database point of view.

bib

book			
year	title	author	
		last	first
2000	XML and HTML	Lee	Sara
		Brick	Peter
2002	Query on the We	Lee	Sara

Figure 4.5: XML Document Represented with XML-RL Model

As illustrated in Figure 4.6, to process an XML query, the XML Parser accepts XML documents as input, and outputs the query result in XML format. Note that the processing is not done directly on the XML document but through the XML-RL model, represented by those two shaded boxes between the XML documents and the XML Query.

Our data model for XML, on one hand, is tailored for our query language, which makes the query processing quite straightforward with respect to the model; on the

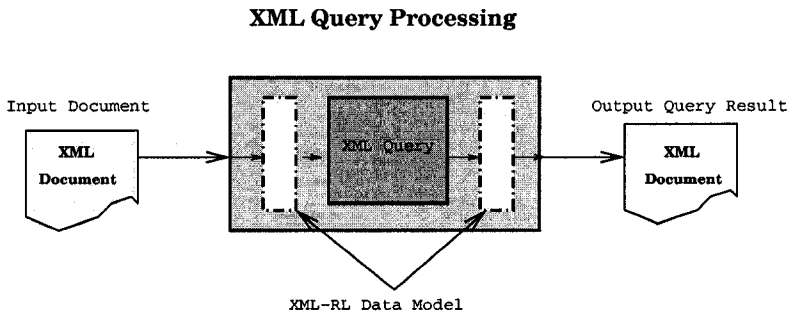


Figure 4.6: XML Query data Flow Diagram

other hand, it makes it necessary to provide a component to transform between the XML document and our data model. Through the data model, we can see various objects: as in Figure 4.5, element objects represent title elements and book elements; attribute objects describe the year attribute; list objects describe authors of a book. It is not hard to find the correspondence for the objects in the XML-RL model and those described as nodes in DOM or other lower level model.

To access to XML document, our query system also uses DOM. However, DOM in fact is only an abstract model and it declares a set of Application Programming Interfaces. After an XML document is parsed, the data is seen and manipulated in a way as our data model defined.

4.4 Classes for Design and Implementation

This section explains the design of some important classes in implementing the system. The design of the java class is diagramed in Appendix C. Their structures and functions are also discussed. Those classes are of three kinds: classes for query data management, classes for result construction, and classes for processing control. A sample query Q_1 is used through the subsections to explain how the classes and objects cooperate with one another in order to retrieve the information and construct the result. Q_1 is as follows:

```
bib.xml%/bib/book:$b,
```

```
$b/title:$t,
$b//author:$a,
⇒ result%/results/result:  groupby($a)[author:$a, title:$t ];
```

4.4.1 Classes for Query data Management

Variable Management In XML-RL, variables play a very important role. They are used to bind various components of XML documents. It is also one of the biggest challenges in implementing the system, as XML has a hierarchical structure, and a containment relationship between different variables are very common.

A direct implementation would store the variables in a linear structure, such as a stack or a list of nodes, and each node contains the variable's value and its related information, such as its type, and navigation path.

We solve this problem by devising a different method. For the variables, which are defined by the structural information, we store them in the *Variable Guide* which is also a tree-based structure. It keeps track of variable names, their hierarchical information and how one variable is related to another variable.

This idea comes from two considerations: first, the variables in XML-RL query are not simply a set of stand-alone values, like in a Logic Programming language, such as Prolog. They are structurally related, with either a sibling, parent-child or ancestor-descendent relationship. Second, we believe if we take the semantics of all underlying structure into account and facilitate that information into the representation of the variable itself, it would lead to a more efficient way to track the structural information of those variables. Figure 4.7 describes how the Variable Guide stores the information in a query head in Q_1.

The vital part of this method is that it specifies the portion of the structure which is of interest in that particular query. All structures and data irrelevant to the query are not included in the Variable Guide. We can see from two directions how it works: horizontally for a `bib` element, not all elements are recorded in the Variable Guide, as in the example, only book element is built in the tree, so the Variable Guide is not an exact structure mapping of the input XML.

However if we see the data that is not of interest as in a black box, the Variable

Guide reflects the structure of the XML document being queried. Consider the variables `$t` defined by `$b/title:$t`, and `$a` defined by `$b//author:$a`. In spite of the difference in their navigation path - one uses single slash (/) for one layer down the XML hierarchy and the other uses double slash (//) for multiple or unknown layers, `$b` is the parent variable of both `$t` and `$a`.

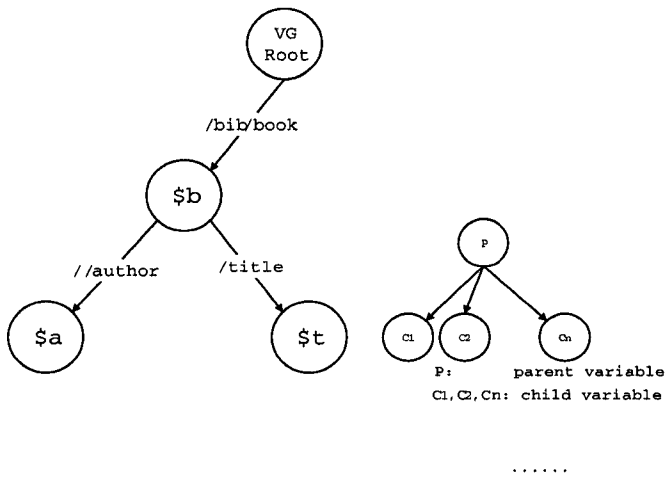


Figure 4.7: The Structure of The Variable Guide

data Repository The *data Repository* bridges between the querying and the result processing. It may be regarded as an intermediate result.

Functionally what we need is a list of variables with their possible values listed according to the precedence of time as they appeared in the query. In our system, it is implemented also with a binary tree. As seen in Figure 4.8, the nodes are linked with solid lines.

The *Repository Node*, which is a *RepoNode* object, can hold either the variable's meta information, such as its type and name, etc. or its value. In the Repository tree, nodes for storing variable meta information are attached to the root node as the left child. Each variable's value is attached to it as right child, with every node's right child set to variable's next possible value.

An example is given here as a further explanation. The XML document being queried is based on the example in Section 4.2.2. Note we give each element node and attribute node a name on the top of each node, in order to distinguish them.

The data Repository built for this query is illustrated in Figure 4.8. We gave symbols to the Repository tree node and its values to refer to them: each node is represented with a capital identifier with indices, and values with lowercase identifiers with indices. For example, the first author element node is denoted by $\mathtt{A}_{11}$, its value is referred to as $\mathtt{a}_1$, corresponding to author element with last name `Lee` and first name `Sara` in the XML mentioned in Section 4.4,

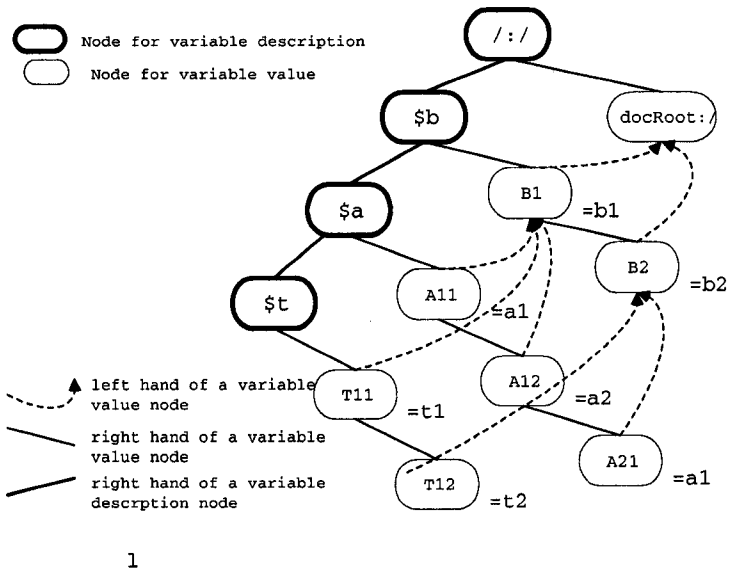


Figure 4.8: An Example of XML Query data Repository

As we noted, a variable is uniquely denoted with both its value and its hierarchical information. Two kinds of information are stored in the data Repository: meta data of each query variable (such as its type, name) and its possible bindings.

The data Repository tree is formed as follows: the meta information are directly linked to the root of the data Repository. In Figure 4.8, node `$b` and `$a` and `$t`, with a bold border, contain the meta information and are connected to the tree root of the

data Repository. Each of a variable's possible values is also linked to one another, and attached to the node containing the variable's meta data. For example, `$b` has two possible values, B_1 and B_2 linked to it. These values are in a order as they appear in the input XML document.

In the data Repository, each node keeps an indicator to its parent variable value. By tracking down the parent variable to its child variable, values to current variable are located in the XML document, and inserted into the data Repository.

The Variable Guide records the hierarchical structure of the queried data from the level of a variable's structure, whereas the data Repository stores the structural information to data from the level of a variable's binding values. That is, the Variable Guide tells which *variable* it is hierarchically related to, while the Repository specifies which *value* it related to in the XML hierarchical structure. The meta data are used to communicate between the Variable Guide and the data Repository.

To make this process clear, let us consider a simple example shown in Figure 4.9. It is based on the XML document shown in Figure 4.4 and query shown in Section 4.4. The procedure to fill the data into the data Repository is as follows: Every time the XML-RL query processor needs to obtain data from an XML document, it first consults the Variable Guide.

As for clause `$b/author:$a`, to insert values of `$a` into the data Repository, the query processor sends a request to the Variable Guide and gets the name of the parent variable, that is, `$b`. In the data Repository, all possible values to the parent variable are accessible to the query processor through its variable name.

Starting at XML document's root element, it finds book B_1, then finds author A_{11}, A_{12}, meanwhile A_{11} and A_{12}'s parent variables value are set as B_1. Similarly the parent variable's value binding of A_{21} is set as B_2 when it is traversing down the XML hierarchy from A_{21} to B_2.

4.4.2 Classes for Result Construction

It is an important use of the query language to transform the XML information by constructing the structure of the query result. Some main transformations include

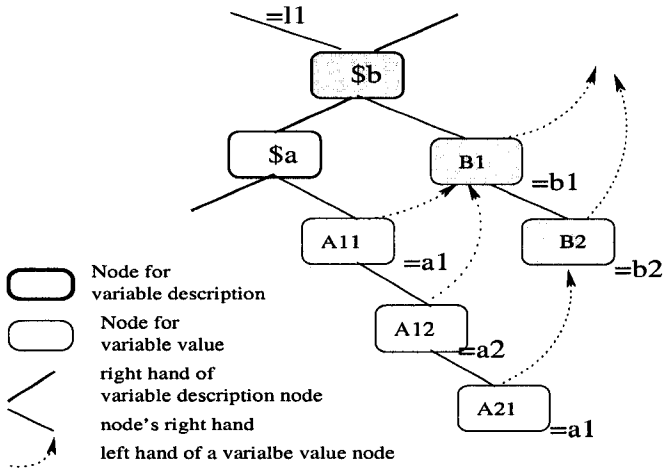


Figure 4.9: Filling data in data Repository

nesting, flattening and sorting the result. The major task in creating the *Result Tree* is data reorganization and reformatting.

data Organization and Reorganization The result is assembled through interacting with the data Repository. The goal for designing the Result Tree is to organize the variable values, so that the result construction tasks can be performed conveniently.

With different query requests and XML data, the insertion criteria varies, in here they are classified into two kinds, referred to as *direct insertion* and *cross insertion.*

Direct insertion happens when a child variable value is appended to the result tree directly under its parent variable. In this case, the parent variable is a leaf node, all the possible values of this child variable are simply inserted under its parent variable's binding. For example, in Figure 4.10, the value nodes A_{11}, A_{12} of `$a` are directly inserted under the node B_1 which is the value of its parent variable `$b`. In Figure 4.10, the rectangular box with a variable's name inside on its right top represents all the possible bindings after its parent variable is bound to a certain value. The circle box contains one binding of the variable.

Cross insertion is needed when a variable's values are indirectly inserted under

the binding of its parent variable. This takes place if before this variable is bound, another variable which has the same parent variable, has already inserted its values under its parent variable's binding. For example, in Figure 4.10, after $b is bound, $t and $a are bound subsequently. The bindings of $t and $a are relatively independent of each other. $t's or $a's values are only dependent on their parent variable $b's binding. It explains why $t's value T_1 is inserted under each leaf node reachable through B_1, i.e., under A_{11} and A_{12}.

This can easily accomplish a query (referred to as Q2), like listing all the title-author pair in the XML shown in Figure 4.4. Q2 is as follows:

```
bib.xml%/bib/book :$b,
$b/title:$t,
$b/author:$a
⇒ result.xml%/results/result:[ title:$t, author:$a ];
```

This query requests flattening the nested structure in the input XML, to get a one-to-one combination of variable $t and $a. As shown in Figure 4.10, we get the combination of $b, $t, $a, as B_1, T_1, A_{11}, B_1, T_1, A_{12} and B_2, T_2, A_{21}. As it only asks for the title and author in the result, we remove the book from the output. We get the title-author pairs are T_1, A_{11}; T_1, A_{12} and T_2, A_{21}.

Grouping and Aggregation Different from relational database queries, which requires the grouping and aggregation to work together, in XML query languages, grouping is allowed to work either with or without aggregation. The purpose of grouping is to split a collection of data and represent them by putting them into a new set of groups. It plays an important role in reconstruction.

With the Result Tree, it is convenient to achieve the transformation and integration. For the query Q1, in the result constructing part, it requires values of $t to be grouped by values of $a.

The initial step towards the grouping is to specify the structure for the grouping result. It serves as the template of the output. As shown in the right part of Figure 4.11. That is, all the values of variable $t are grouped together if they have a corresponding $a of a same value.

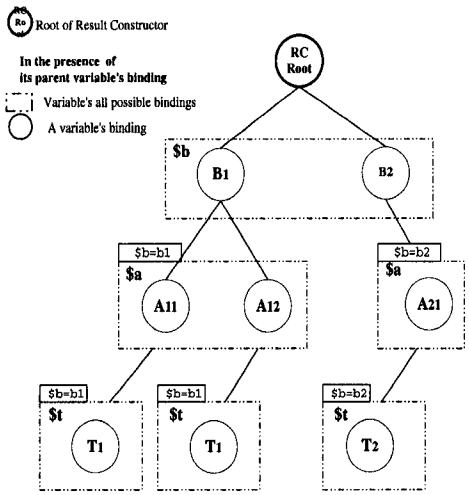


Figure 4.10: The Process of Result Construction

Now the question becomes a search of the node pairs throughout the Result Tree. In this step, it matches the qualified node pairs, combines them into the structure as required under the condition that they have the same variable value of `$a`.

In Figure 4.11, the edges between variables, for example, the edge between `$t` and `$a` represents the reachability, no matter how many intermediate nodes are in between. If there is a path from node A to another node B, A is reachable to B, vice versa. Starting from the root node, traversing the tree found the following node pairs: A_{11} and T_1; A_{12} and T_1; and $A_{21}$and T_2. After the matched node pairs are combined, A_{11} and A_{21} are merged into one, as they have the same value of a_1, and T_1, T_2 are grouped under the new `author` node.

The aggregation is based on the grouping result. XML-RL supports common functions such as `min`, `max`, `avg`, `sum`.

Consider two other queries which are similar to this one. First is Q2.

In Q2's input XML *bib.xml*, the authors are grouped under the book title. This query requires each title and author pair, one-to-one listed separately. This query requires flattening the nested structure and returning the title and author in pair. In

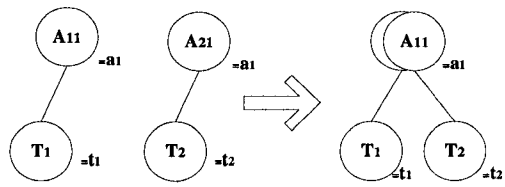


Figure 4.11: The Grouping Operation

XML-RL, a variable starting with $ stands for each single element in a list. So it is very natural to depict the flattening with variable `$t` and `$a`, and gain the Cartesian product of variable bindings.

Another query (referred as Q3) asks to list the title and authors inside a "result" element for each book in the bibliography. Q3 is as follows:

```
bib.xml%/bib/book:$b,
$b/title:$t,
$b/author:$$a
⇒ result.xml%/results/result:[ title:$t, author:$$a ];
```

In fact, this query requires preserving grouping of results by title. Different from Q2, in the query we use variable `$$a`. It is a collection variable in XML-RL, which means all elements with the same tag is treated as a whole, no matter whether it has a single value or multiple values. In this case, authors of the same book are treated as a list. This example query well shows the flexibility of the variable in our query language. Moreover, if we use `$$t` instead of `$t`, standing for all titles to each book, it produces same result, since in fact each book has only one title. However for the clarity of query semantics, the title is expressed in a singular variable form.

4.4.3 Classes for Processing Control

XML-RL has the capability to combine the information from two different sources, and uses the required element object to construct the result. In this system, some classes are responsible for coordinating between the sources, and managing query processing.

Query Processor, implemented with class *QProcessor*, is in charge of information concerning one XML document, including: XML's DOM tree, the variables, all the query related data and the *query result.*

The *Query Manager*, which is an instance of class *QManager*, is needed to coordinate the query processes among each XML data resources, in cases when a query involves several XML documents, or more than one copies of an XML documents.

For an example, a query asks to list the title of the book and its price from each source, for each book found in both *bib.xml* for bn.com and *reviews.xml* for amazon.com.

```
bib.xml%//book:$b,
$b/price:$bp,
$b/title:$t1 ,
reviews.xml%//entry:$a,
$a/price:$ap,
$a/title:$t2 ( =$t1 )
⇒ result.xml%/book-with-prices/book-with-price:[
   title:$t1,
   prices-amazon:$ap,
   price-bn:$bp ];
```

Another example is a query wants to find pairs of books that have different titles but the same set of authors (possibly in a different order).

```
www.bn.com/bib.xml%//book :$b1,
$b1/title:$t1,
$b1/author:$$a1,
www.bn.com/bib.xml%//book:$b2,
$b2/title:$t2(<$t1),
$b2/author:$$a2(~ $$a1)
⇒ result.xml%/bib/book-pair:[ title:t1, title:t2 ];
```

Note the symbol $\sim$ is the equal comparison between two list. Since in XML `$$a1`, and `$$a2` are ordered lists, the default equal comparison (=) requires that every list

member and also its order is identical. ~$$a2 is used to represent list members that are the same despite their orders.

This Query Manager object is in charge of an XML-RL query; it takes the user query as the input and pre-processes the query; if multiple XML files are involved in the query, it creates different Query Processor objects and then dispatches tasks to Query Processors; then each Query Process takes care of the query part concerning each XML file.

All the classes for the XML-RL Query Processor are in a Java package. File *XMLQuery.java* contains class *XMLQuery* and the `main()` method that sets up a Query Manager object.

The following is a piece of code from the Query Processor class. In order to process the query, the Query Processor object creates three objects: a variable guide, that is a `VarTree` object, a data repository, which is a `Repository` object, and query result, which is `ResTree` object.

```
QProcessor (String fileName, String inputStr, QManager qManager)
   {
      this.leftStr = inputStr;
      this.fileName = fileName;

      domTreeRoot = initDomTree( fileName );

      RepoNode repositoryRoot = new RepoNode (new Variable("/", "/"));
      RepoNode repoRootValue = new RepoNode (new DomNode (domTreeRoot));
      repository = new Repository (this, repositoryRoot, repoRootValue);

      Variable varTreeRoot = new Variable ("/", "/");
      varTree = new VarTree(this, varTreeRoot);

      resTree = new ResTree (this, new ResNode (repoRootValue));
      this.qManager = qManager;
   }
```

4.5 Summary of the Chapter

We have presented the design and implementation of XML-RL system in this chapter. The architecture of the system shows how the components fit together and then each component's function is explained, as well as how a query is processed by the system and through the interaction between different components. Some program detail is also given by looking into the classes design and some code of the objects.

Chapter 5

Design for Revised XML-RL

XML-RL is developed on the basis of a complex data model tailored for XML documents. This current system realizes the model in an open and modular way.

An implementation of XML-RL conformable to its latest syntax requires an understanding of the changes as well as the ramification of the changes from the implementation point of view. Since there is no fundamental change to the model, those syntax changes, will not affect the overall architecture of the system and the interactions between components. In the three constituent layers, the layer which consists of the Query Processor and the Query Manager is intact. The pre-processor layer made up of the Query Parser, Lexical Analyser needs some adjustment in order to reflect the new syntax. The changes will mainly occur in the Variable Guide and the data Repository, as the major modification of syntax is in the expression of variables and functions.

5.1 Variable Expressions

This section discusses the impact the changes of variable is going to have on the existing system from several perspectives.

5.1.1 Variable for Objects and Content

In the syntax of current XML-RL, content variables and structure variable have different forms and represent the content and the name respectively. In the new

syntax, when a variable is used to represent an element content, it is used the same way as in the current system. For example, in current system, `bib/book : $b` represents a book element's content, and it is represented similarly `bib/book ⇒ $b` The new syntax introduces a new usage to represent an object as a whole including its name and contents.

```
(bib.xml)/bib/book/$x
```

Here variable `$x` represents the sub-element object of `book`, one element at a time. Another example represents an attribute in a book element as follows:

```
(bib.xml)/bib/book/@$attr
```

In current implementation, variables representing content and structure are distinguished by its appearance. Variables starting with dollar sign are used to represent the content of an element or an attribute, for example, `@X` whereas the variables starting with ampersand represents an XML tag, for example, `&Y`

In the new syntax, variables all start with `$`. It uses a different way to describe its constraints; for instance, if we want to get an attribute whose name is `year`, in current syntax we use structure variable and content variable to represent each part separately, as the following:

```
bib.xml%//book:$b, $b/&e(=year):$v
```

In the new syntax, `@$attr` represent both the attribute as a whole, and specify the its attribute name in the bracket as part of the objects.

```
(bib.xml)/bib/book/@$attr(@year⇒$)
or (bib.xml)/bib/book/@$attr(@year)
```

As the new syntax allows representation of objects, either as a name-value pair or an object as a whole therefore no structure variable is used. The system must be able to react in different situations. This also requires some changes to the data structure of the variable, to include another piece of information in each node to identify what kind of object or what part of XML it represents.

5.1.2 Grouping Variables

The list-valued variable starting with double-dollar. In the new syntax, the grouping variable is used instead of the list-value variable. That is a variable starting with double-dollars, e.g. $$X is replaced by a single-value variable enclosed in braces, e.g. {X}.

Grouping variables can appear both in querying part as well as in the result constructing part. If appearing in the querying, it serves just as a list-variable, which is only interested in a list of values as a whole; for example if we want all subelements of a book element in the new syntax, it is as follows:

```
(book.xml)/bib/book⇒{$e}
```

When the grouping variable is used in the result constructing part, it implies a grouping operation. For example, for each author in the bibliography, if we want to list the author's name and the titles of all books by each author, we use the following:

```
querying
   (bib.xml)/bib/book ⇒[ title⇒$t, author⇒$a ]
constructing
   results/result⇒[author⇒$a, title⇒{$t}]
```

The grouping variable is used in this query to specify the result: $a and {$t} are put together in a tuple. It implies $t to be grouped by values of $a. In this case, this variable represents the same request as

```
bib.xml%/bib/book:$b,
$b/title:$t,
$b/author:$a,
⇒ result%/results/result:
   groupby($a)[author:$a, title:$t ];
```

To manage a grouping variable requires the parser to identify the use and behavior of those variables under different context, and modifications to the Variable Guide and the data Repository are also necessary in order to handle them accordingly.

5.2 Functions

The new syntax adopts an object oriented fashioned function. For instance, some aggregation functions, such as `max()`, `count()`, `avg()` , can be applied to a grouping variable, then the result value is assigned to a single-valued variable.

```
{$Nums}.avg()=$avgNum
{$Nums}.min()=$minNum
```

There are also some other built-in functions, for example:

```
{$authors}.firstTwo() ∋ $a
{$authors}.last() = $la
{$authors}.sort() = {$sa}
{$authors}.distinct() = {$da}
```

This in fact provides another way to define variables. In the current system, the variables are defined only through the hierarchical information and they directly correspond to objects in XML. A function is used in the result construction to get a new value from existing variables. It has an impact on the Variable Guide as well as the data Repository. Presently, the Variable Guide only keeps track of variables directly corresponding to an element or an attribute in XML documents. Each variable is organized based on its first occurrence and how it is hierarchically related to other variables. A variable, whose value is derived from a function would not fit in the Variable Guide in a same manner as variables defined by the hierarchical information in XML.

As indicated in the example, variables defined by functions could sometimes be numbers, such as `$avgNum`, `$minNum` defined by aggregation functions of `avg()` and `min()`. In other occasions, they could be a member of list of objects, for example, variable `$la` represents the last element of a grouping variable defined by the function `last()`. It could also be a list object after some type of conversion, as an example, variable `$sa` is defined as an object in the sorted list of authors.

Variables defined by a function can be handled with an additional data structure. The information required to be kept in this structure is: variable name, function used

to define it, as well as the variable that serves as the parameter for this function. The method to locate a variable's meta data in current system also needs to to change. When no matched variable is found in the Variable Guide, it should continue searching for it in the extended structure, in case the variable is defined through a function instead of reflecting directly a segment of the XML document.

In the Query Management layer of the system architecture, the Query Processor needs to initialize it as an extended part of Variable Guide. Consequently, the data Repository needs adjustment to accommodate the values of this kind of variables. With assistance of the Variable Guide, data Repository is able to calculate the value or values to the function and stores them in an appropriate place.

5.3 Other Changes

In addition there is a slight expansion in other areas of the syntax. We propose to introduce existential and universal quantifier to reduce the number of rules in XML-RL query. The design of current XML-RL system strives to allow flexibility with as little effort as possible. As a result, with some effort it should not be hard to realize the latest syntax upon the existing system despite of the changes and adjustment necessary.

5.4 Summary of the Chapter

The problem this chapter faced is to implement the changes of the existing system to reflect the evolution of the XML-RL languages. We discuss the ramification of these changes and feasibility and options for developing the revised system.

Chapter 6

Conclusion and Future Research

This thesis describes the XML-RL query system, which supports data extraction, transformation and integration over XML documents. In Chapter 4, it shows how to perform these functions with examples. The purpose of implementing the XML-RL query system is twofold: to investigate the feasibility, and at the same time experience the XML-RL language and to address any need to improve or redress the language.

From an implementation point of view, there are several open issues which still need to be addressed.

System Architecture and data Management In our prototype system, XML-RL depends on an API to parse the XML documents, and leaves the storage management to the algorithm of the XML processor. A different method is to model and store the semantics of the data, as in some proposals which XML documents are parsed and then stored in another form, such as a relational table or objects under database management systems [65] or clustered data according to the application's requirement. As an instance, *XML-OQL* [27] makes use of the database management technology for storing and managing XML data. The main components of the query system includes a type-checker and a query translator. It uses translation rules to translate XML-OQL query into OQL query and imports XML file into a lamda-DB database. Another well-known XML query language *Lorel* [7] also uses the *Lore* [53] database to manage its data. Lore's architecture has two layers: the Query Compilation layer and the data Engine. The Query Compilation layer is responsible for

parsing the query and transforms a Lorel query to an OQL-like query and then generates and optimizes the query plan and sends it to the data Engine layer. data Engine layer contains OEM object manager and other utilities to access the underlying data object.

Current XML-RL's implementation uses DOM programming interface to access and process the XML document. Before a query is executed, an XML document must be parsed into a DOM tree structure. Since DOM API builds an in-memory parse tree, it puts a strain on system resources when XML is too complex and large to fit in the memory. On contrast, as we have discussed in Chapter 3, an event-based API makes it possible to process a very large document using relatively constant resources. For example, SAX needs to deal with XML document piece by piece sequentially, thus requiring less resources. So we may also incorporate some techniques of SAX to reduced memory overhead for the DOM structures and the XML style sheet parsing times.

We are considering the possibility of utilizing the storage management techniques provided in deductive database. By making extensions to these techniques we may be able to replace or enhance the data management for XML-RL query system.

Compatibility and Extensibility A variety of related standards and recommendations have come into being and XML's enormous growth suggests still more to come. Presently, we see XPath, DTD and XML Schema for describing XML's structure and content, XLink and XPointer for linking and referencing information within documents, XML-PC and SOAP for making remote procedure calls over XML. Among them, some are widely accepted and used, such as XPath and DTD; however not all of the standards are incorporated in each processing tool or application. For instance, to this point, no query language supports XPointer or XLink.

In our perspective, it is neither necessary nor possible to support all the standards in an application at a time. It is a choice affected by the application requirements, problem scope and other factors. In order to improve the usability and strength of XML-RL, we need to keep up with the development of up-to-date XML technologies while at the same time maintain a user query interface relatively consistent to the

existing one.

Optimization and Performance Issues In the present work, we have done little performance analysis of the XML-RL query system. There are a number of performance aspects we want to consider, such as its performance on large XML documents, performing the operations and algorithm with a different order, etc.

The database community often measures the performance in terms of scale-up in the size of the data (the *data complexity*) and in the size of the queries (the *query complexity*), which is key for tasks like optimization. A query system's optimization is closely related to its language features, storage format and the needs of its own language.

In our experiment, we use the sample XML files listed in Appendix A. For the queries used in this thesis, Chapter 3 and 4, it took an average 1 to 2 seconds for each query to complete when testing was done on a Celeron 600 (452M) personal computer running Linux. As part of the experiment result, the following graph is a comparison of the query time of each query against XML documents in different sizes. The queries are list in Appendix D. The dots in the lower line represents the query time taken when the queries were run against the sample XML documents. The dots in the upper line records another set of time when same queries were run against documents with the same structure but a hundred times bigger.

The *data complexity* and the *query complexity* are not the only factors that affect the performance. With regards to the nature of XML and XML query language, there are other arguments of measurements, such as the *source complexity* and the *degree of irregularity*. The source complexity means the scale-up in the number of sources. XML query languages is able to integrate information from different sources, and it may encounter large number of relatively small files as well as small number of large XML files. The degree of irregularity is to measure how the flexibility of the XML data can affect the performance.

A Query Manager is used to coordinate the data from different resources, and each resource is controlled by a Query Processor. With Java's support of multi-tasking, it is possible to use this technique to improve the efficiency of the query system.

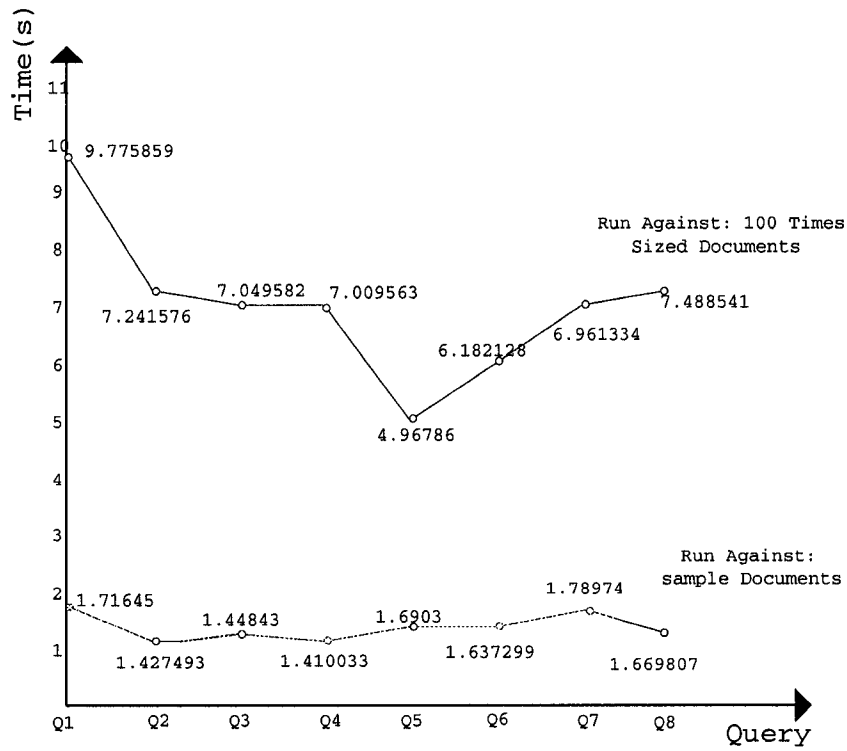


Figure 6.1: Comparison of Query Time against XML Documents of Different Sizes

There is also significant additional research to do in query optimization, including investigating the possibility of storing XML in a database system or other object management system, developing indexed on the data values, exploiting additional heuristic optimizations in data access and operation. We notice that for integration or other reasons, there are cases when XML-RL has to process different resources at one time. To this point, we only use single task to process the query.

XML also provides support for irregular data. How to characterize the kinds of irregularities, and how they affect the query processing is still an open question. Some measure is proposed to measure the performance with respect to the type and degree of irregularity in the data. It has major implications on how it is most efficiently stored and indexed, described, subsequently on query processing.

XML-OQL transfers its query to OQL and then relies on the underlying OD to perform the optimization. Lorel uses simple heuristic query optimization, and employs two kinds of indexes for optimization purpose, one for object's value and the other for object's path for speeding up the process to find an element's parent. XML-RL query system currently employs only a few simple heuristic query optimization rules. For example, the constraint is put near the variable; it eliminates the useless information by evaluating the constraint as soon as possible, which is a quite similar technique to pushing the filter in the algebraic expression in database query. XML-RL also takes advantage of accessing XML data through API, which not only loads XML into memory but also breaks documents into discrete content fragments. In the processing, when XML-RL needs to track the object, it uses reference to data in memory without copying the actual XML information, thus it also saves space and time.

Robustness There are occasions when an unexpected operation or an inappropriate query is run against the document. Currently we make use of the exception mechanism and extend it to handle these problems. When an error occurs, the exception is captured and the error message is reported to the user and no result is returned. However many times it is frustrating for the users to restart a query when a minor error causes the exception.

In order to make it more friendly and more efficient to use, it is necessary to provide choices when an exception is raised to take care of the situation and resume after where it was suspended. Useful as it is, a mechanism of recovery may also be delicate and demanding, since in the real world many of the exceptions are hard to expect. More time and consideration is needed before a more powerful exception handling component is developed in the query system.

In our tests cases, the load-time errors are perhaps the most common and are generally caused by problems in syntax. For example, when a query is issued without a semicolon, which tells the system it is the end of a query, a message will return: `Syntax error, couldn't repair and continue parse.` Other situations like misuse of function names or other constructs will also result in a syntax error. As another

example, when a variable is used before it is defined, system will return: `Variable not found in table:$varname`.

When a list-variable is used in the place where a single value is required: For a query as following:

```
bib.xml%/bib/book:$b,
$b/title:$t,
$b/author:$$a
⇒ result.xml%/results/result:[ title:$t, @author:$$a ];
```

In this example, the `author` attribute is expecting a single value, where a list variable `$$a` is given. An error will occur: `ERROR: Incompatible node type`.

Meantime, a query may fail to match anything in several situations. For example, when the file specified is not found, either a wrong file name is given or such file does not exist, the query will return immediately with an error: `No such file or directory`. Thus no result file is created.

Runtime errors are most often due to improper use of query constructs. In some cases, the error messages do not always indicate the actual error. Here is one such example; it happened when wrong object name was specified in the query or attribute symbol('@') is missing in an attribute name. The element or attribute required does not match any markup, an exception is raised by the system: `Java.lang.IndexOutOfBoundsException: Index:0, Size:0`. The error message is not very informative or instinctive. In order for the error message to be used to narrow down the source of the problems, we need to characterize different kinds of situations and identify them accordingly .

Being unable to find any data satisfying the query criteria will also make the query fail to return anything. For example,

```
bib.xml%/bib/book:$b,
$b/publisher:$pee(="Addison-Wesley"),
$b/@year:$ye(>1991),
$b/title:$t
⇒ result.xml%/bib/book:[ @year:$ye, title:$t ];
```

If there is no publisher element with a value equal to "Addison-Wesley", a file with empty root element is returned as the result.

Logic error is another type of errors that occur when users fail to use the language in the way it is designed. This type of error may not necessarily result in a system exception or any, instead it may give some result that is different from what user expects. Logic errors can be complicated and of a great variety. However logic errors can be reduced by a further study of the syntax and semantics of languages. A comparison of sample queries and results will also help to reduce the logic errors.

6.1 Summary of the Chapter

As the conclusion of the thesis, this chapter is a overview of perspectives where further consideration is needed in the existing system, such as other alternatives to implement storage of the XML, the flexibility to accommodate future needs, a more efficient and user-friendly error handling and a method to speed up the query.

Bibliography

[1] LEX and YACC Howto. *http://www.tldp.org/HOWTO/Lex-YACC-HOWTO.html.*

[2] W3C Working Draft 04 December 2000. The XML Query Algebra. *See at http://www.w3.org/TR/2000/WD-query-algebra-20001204/.*

[3] W3C Working Draft 15 February 2001. XML Query Data Model. *See at http://www.w3.org/TR/2001/WD-query-datamodel-20010215/.*

[4] W3C Working Draft 15 February 2001. XML Query Use Cases. *See at http://www.w3.org/TR/xmlquery-use-cases.*

[5] F. Arciniegas A. *XML Developer's Guide.* McGraw-Hill, 1 edition, 2001.

[6] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases.* Addison Wesley, 1995.

[7] S. Abiteboul, D. Quass, and et al. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[8] N. Ashish and C. Knoblock. Modeling Web Sources for Information Integration. In *Proceedings of Workshop on Management of Semistructured Data*, 1997.

[9] P. Atzeni, G. Mecca, and P. Merialdo. Semistructured and structured data in the web: Going back and forth. In *Workshop on Management of Semistructured Data*, 1997.

[10] H. Bingham. SGML Syntax Summary Table of Contents. *See at http://xml.coverpages.org/sgmlsyn/contents.htm.*

[11] S. Boag, D. Chamberlin, M. F. Fernandez, and et al. XQuery 1.0: An XML Query Language. *See at http://www.w3.org/TR/xquery/.*

[12] A. Bonifati. Technical Survey of XML Schema and Query Languages. *See at http://citeseer.nj.nec.com/402992.html.*

[13] A. Bonifati and S. Ceri. Comparative Analysis of Five XML Query Languages. *SIGMOD Record*, 29(1):68–79, 2000.

[14] C. Bornhvd. Semantic Metadata for the Integration of Web-Based Data for Electronic Commerce. In *International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems*, 1999.

[15] P. Bothner. Qexo: The GNU Kawa implementation of XQuery. *http://www.gnu.org/software/qexo/.*

[16] Ronald Bourret. XML and Databases. *http://www.rpbourret.com/xml/XMLAndDatabases.htm*, Nov 2000.

[17] F. Bry and N. Eisinger. Data Modeling with Markup Languages: A Logic Programming Perspective. In *15th Workshop on Logic Programming and Constraint Systems*, Berlin, August 2000.

[18] R. G. G. Cattell and D. Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[19] S. Ceri, S. Comai, and et al. XML-GL: A Graphical Language for Querying and Restructuring XML Documents. pages 151–165, 1999.

[20] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proceedings of WebDB 2000 Conference.* Springer-Verlag, 2000.

[21] S. Cluet and J. eon. Yatl: a functional and declarative language for xml, 2000.

[22] X-Hive Corp. X-Hive's XQuery demo. *See at http://support.x-hive.com/xquery/index.html.*

[23] R. Cover. XML and Query Languages. *http://xml.coverpages.org/xmlQuery.html*, November 2001.

[24] A. Deutsch, M. Fernandez, and et al. XML-QL: A Query Language for XML. *See at http://www.w3.org/TR/NOTE-xml-ql/.*

[25] A. Deutsch, M. Fernandez, and et al. XML-QL:A Query Language for XML. *See at http://www.research.att.com/sw/tools/xmlql/.*

[26] W3C Working Draft. XML Query (XQuery) Requirements. *http://www.w3.org/TR/xmlquery-req/.*

[27] L. Fegaras and R. Elmasri. Query Engines for Web-Accessible XML Data. In *The VLDB Journal*, pages 251–260, 2001.

[28] F. Fernandez, J. Simeon, and et al. XML Query Language Experience and Examples. *http://www.w3.org/1999/09/ql/docs/xquery.html.*

[29] M. Fernandez, D. Florescu, and A. Levy. A Query Language for a Web-Site Management System. *SIGMOD Record*, 26(3):4–11, 1997.

[30] T. Fiebig, J. Weiss, and G. Moerkotte. RAW: A Relational Algebra for the Web. *http://www.research.att.com/ suciu/workshop-paper/paper05.ps*, 1997.

[31] D. Florescu, A. Levy, and A. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, 1998.

[32] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *VLDB.*

[33] N. Guarino. Understanding, Building, And Using Ontologies. *International Journal of Human-Computer Studies*, 1997.

[34] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting Semistructured Information from the Web. In *Proceedings of Workshop on Management of Semistructured Data*, 1997.

[35] P. Le Hgaret, R. Whitmer, and L. Wood. Document Object Model(DOM). *See at http://www.w3.org/DOM.*

[36] H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language. In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.

[37] G. Huck and I. Macherius. GMD-IPSI XQL Engine. *See at http://xml.darmstadt.gmd.de/xql/.*

[38] Lucent Technologies INC. Galax: The XQuery implementation for discriminating hackers. *See at http://db.bell-labs.com/galax/.*

[39] V. Kashyap and M. Rusinkiewicz. Modeling and Querying Textual Data using E-R Model and SQL. *http://citeseer.nj.nec.com/kashyap97modeling.html*, 1997.

[40] N. Klarlund, A. Moller, and M.I. Schwartzbach. Document Structure Description (DSD): An XML Schema Language. *See at http://xml.coverpages.org/dsdAnn19991119.html.*

[41] C. A. Knoblock, S. Minton, J. L. Ambite, N. Ashish, P. J. Modi, I. Muslea, A. G. Philpot, and S. Tejada. Modeling Web Sources for Information Integration. In *Proceedings of the 15th National Conference on AI*, 1998.

[42] D. Lee and W. Chu. Comparative Analysis of Six XML Schema Languages. *SIGMOD Record*, 29(3):76–87, 2000.

[43] X. Leroy. The Objective Caml system release 3.06 Documentation and user's manual. *See at http://caml.inria.fr/ocaml/htmlman/.*

[44] M. E. Lesk and E. Schmidt, editors. *Unix Programmer's Manual.* Bell Laboratories, 1989.

[45] M. Liu and T. W. Ling. A Conceptual Model for the Web. In *Proceedings of the International Conference on Conceptual Modeling (ER 2000)*, pages 225–238, Salt Lake City, October 9-12 2000. Springer-Verlag LNCS 1920.

[46] M. Liu and T. W. Ling. A Data Model for Semistructured Data with Partial and Inconsistent Information. In *Proceedings of the International Conference on Advances in Database Technology (EDBT 2000)*, pages 317–331, Konstanz, Germany, March 27-31 2000. Springer-Verlag LNCS 1777.

[47] M. Liu and T.W. Ling. XML-RL: A Rule-based Language for XML. *See at http://citeseer.nj.nec.com/470369.html.*

[48] M. Liu and T.W. Ling. A Logical Foundation for XML. In *14th International Conference on Advanced Information Systems Engineering*, pages 568–583, Canada, May 2002. LNCS.

[49] M. Liu and T.W. Ling. Towards Declarative XML Querying. In *the 3rd International Conference on Web Information System Engineering*, Singapore, Decemeber 2002.

[50] M. Liu, L. Lu, and G. Wang. A Declarative XML-RL Update Language. In *22nd International Conference on Conceptual Modeling*, Chicago Illinois, October 2003.

[51] M. Liu and R. Shan. Design and Implementation of the Relationlog Deductive Database System. In *DEXA Workshop*, pages 856–863, 1998.

[52] XML-DEV mailing list. Simple API for XML(SAX). *http://www.saxproject.org/.*

[53] J. McHugh, S. Abiteboul, and et al. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[54] M. Morrison and et al. *XML Unleashed: From Knowledge To Mastery.* Sams Publishing, 1 edition, 2000.

[55] P. O'Neil and E. O'Neil. *Database Principles: Programming and Performance.* Morga Kaufmann Publishers, 2 edition, 2000.

[56] H. Qi and M. Liu. Design and Implementation of XML-RL Query System. *Database and Applications 2004(Accepted).*

[57] D. Raggett, A. L. Hors, and I. Jacobs. HTML 4.01 Specification. *See at http://www.w3c.org/TR/html401.*

[58] W3C Recommendation. XHTML 1.0 The Extensible HyperText Markup Language. *See at http://www.w3.org/TR/xhtml1/.*

[59] W3C Recommendation. XML Data Type Declaration. *See at http://www.w3.org/DTD.*

[60] W3C Recommendation. XML Schema. *See at http://www.w3.org/Schema.*

[61] W3C Candidate Recommendation. XML Path Language(XPath),Version 1.0. *See at http://www.w3.org/TR/xpath.*

[62] W3C Candidate Recommendation. XSL Transformations (XSLT), Version 1.0. *See at http://www.w3.org/TR/xslt.*

[63] J. Robie, J. Lapp, and D. Schach. XQL XML Query Language. *See at http://www.w3.org/TandS/QL/QL98/pp/xql.html.*

[64] J. Shanmugasundaram, K. Tufte, and et al. Relational Databases for Querying XML Documents: Limitation and Opportunities. In *Proceedings of the 25th VLDB Conference*, 1999.

[65] T. Shimura, M. Yoshikawa, and et al. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Database and Expert Systems Applications*, pages 206–217, 1999.

[66] D. Suciu. Semistructured data and XML. In *FODO*, 1998.

[67] D. Suciu. Managing Web Data. *SIGMOD Conference*, 1999.

[68] H. Thompson. Extensible Stylesheet Language(XSL). *See at http://www.w3.org/Style/XSL.*

[69] W3C. Extensible Markup Language (XML). *See at http://www.w3.org/XML/.*

[70] W3C. HyperText Markup Language (HTML). *See at http://www.w3.org/MarkUp/.*

[71] W3C. Overview of SGML Resources. *See at http://www.w3.org/MarkUp/SGML/Overview.html.*

[72] W3C. Web Style Sheets. *See at http://www.w3.org/Style/.*

[73] K. Williams, M. Brundage, and et al. *Professional XML Database.* Wrox Press Ltd., 2 edition, 2001.

Appendix A

Example XML Documents

We use W3C XML Query Use Cases as the usage scenarios to test our query language. The example XML documents with their DTD files are listed here.

DTD for "bib.xml"

```
<!ELEMENT bib (book* ) >
<!ELEMENT book (title, (author+  |  editor+ ), publisher, price ) >
<ATTLIST book year CDATA ♯ REQUIRED >
<!ELEMENT author (last, first ) >
<!ELEMENT editor (last, first, affiliation ) >
<!ELEMENT title (♯ PCDATA )>
<!ELEMENT last (♯ PCDATA )>
<!ELEMENT first (♯ PCDATA )>
<!ELEMENT affiliation (♯ PCDATA ) >
<!ELEMENT publisher (♯ PCDATA ) >
<!ELEMENT price (♯ PCDATA ) >
```

bib.xml

```
<bib>
   <book year="1994" >
      <title>TCP/IP Illustrated </title>
      <author> <last>Stevens </last > <first>W. </first > </author>
```

```
      <publisher>Addison-Wesley </publisher>
      <price> 65.95 </price>
   </book>
   <book year="1992" >
      <title>Advanced Programming in the Unix environment </title>
      <author> <last>Stevens </last > <first>W. </first> </author>
      <publisher>Addison-Wesley </publisher>
      <price>65.95 </price>
   </book>
   <book year="2000">
      <title>data on the Web </title>
      <author> <last>Abiteboul </last > <first>Serge </first> </author>
      <author> <last>Buneman </last> <first>Peter </first> </author>
      <author> <last>Suciu </last > <first>Dan </first > </author>
      <publisher>Morgan Kaufmann Publishers </publisher>
      <price> 39.95 </price>
   </book>
   <book year="1999">
      <title>The Economics of Technology and Content for Digital TV</title>
      <editor>
         <last>Gerbarg </last >
         <first>Darcy </first>
         <affiliation>CITI </affiliation></editor>
      <publisher>Kluwer Academic Publishers </publisher>
      <price>129.95 </price>
   </book>
</bib>
```

DTD for "review.xml"

```
<!ELEMENT reviews (entry*) >
<!ELEMENT entry (title, price, review) >
<!ELEMENT title (♯ PCDATA) >
```

```
<!ELEMENT price (♯ PCDATA) >
<!ELEMENT review (♯ PCDATA) >
```

review.xml

```
<reviews>
   <entry>
      <title> data on the Web </title>
      <price>34.95 </price>
      <review>
         A very good discussion of semi-structured database systems and XML.
      </review>
   </entry>
   <entry>
      <title> Advanced Programming in the Unix environment </title>
      <price>65.95 </price>
      <review>
         A clear and detailed discussion of UNIX programming.
      </review>
   </entry>
   <entry>
      <title>TCP/IP Illustrated </title>
      <price>65.95 </price>
      <review>One of the best books on TCP/IP. </review>
   </entry>
</reviews>
```

DTD for "prices.xml"

```
< !ELEMENT prices (book*) >
< !ELEMENT book (title, source, price) >
< !ELEMENT title (♯ PCDATA)>
```

```
< !ELEMENT source (♯ PCDATA)>
< !ELEMENT price (♯ PCDATA)>
```

prices.xml

```
<prices>
   <book>
      <title>Advanced Programming in the Unix environment </title>
      <source>www.amazon.com </source>
      <price>65.95 </price>
   </book>
   <book>
      <title>Advanced Programming in the Unix environment </title>
      <source>www.bn.com </source>
      <price>65.95 </price>
   </book>
   <book>
      <title> TCP/IP Illustrated </title>
      <source>www.amazon.com </source>
      <price>65.95 </price>
   </book>
   <book>
      <title> TCP/IP Illustrated </title>
      <source>www.bn.com </source>
      <price>65.95 </price>
   </book>
   <book>
      <title>data on the Web </title >
      <source>www.amazon.com </source>
      <price>34.95 </price>
   </book>
   <book>
      <title>data on the Web </title>
```

```
      <source>www.bn.com </source>
      <price>39.95 </price>
   </book>
</prices>
```

DTD for "books.xml"

```
<!ELEMENT chapter (title, section*)>
<!ELEMENT section (title, section*)>
<!ELEMENT title (♯ PCDATA)>
```

books.xml

```
<chapter>
   <title>data Model </title>
   <section>
      <title> Syntax For data Model </title>
   </section>
   <section>
      <title>XML </title>
      <section>
         <title>Basic Syntax </title>
      </section>
      <section>
         <title> XML and Semistructured data </title>
      </section>
   </section>
</chapter>
```

Appendix B

Syntax of XML-RL

Query	::=	*QueryRule* ' ; ' *QueryRule* ' . '
SourceItemList	::=	SourceItem
		| SourceItemList ',' SourceItem
SourceItem	::=	FullElementName ':' SingleVar
' (' Operator SingleOperand ')'		| FullElementName ':' SingleVar
		| FullElementName ':' CollectionVar ' (' Operator ColOperand '
		| FullElementName ':' CollectionVar
FullElementName	::=	Url ElementPath
		| SingleVar ElementPath
Url	::=	URL
		| FileName '%'
FileName	::=	IDENTIFIER '.xml'
TagNameConstraint	::=	' (' Operator SingleOperand ')'
		|
ElementPath	::=	ElementPathUnit
		| ElementPath ElementPathUnit
; ElementPathUnit	::=	DepthID ElementName
		| DepthID ADDR AttributeName
DepthID	::=	'/'
		| '//'
ElementName	::=	IDENTIFIER
		| & IDENTIFIER TagName Constraint

		| SingleVar
AttributeName	::=	IDENTIFIER
CollectionVar	::=	'$$' IDENTIFIER
SingleVar	::=	'$' IDENTIFIER
SingleOperand	::=	NUMBER
		| STRCONST
		| SingleVar
Operator	::=	>
		| <
		| >
		| <=
		| >=
		| ! =
		| ~
		| 'contain'
		| 'endwith'
		| 'beginwith'
ColOperand	::=	NUMBER
		| CollectionVar
		| '{' ColElement '}'
ColElement	::=	ColElement STRCONST
		| STRCONST
Result	::=	FileName '%' ResElementPath ':' TupleOption ResultValue SEM
TupleOption	::=	SortBy
| Group		
| Group ',' SortBy		
| SortBy ',' Group		
|		
Group	::=	'groupby' ' (' SingleVarList ')'
ResElementPath	::=	'/' EType
| ResElementPath '/' EType		
ResultValue	::=	'[' ResTuple ']'
| ElementValue		

```
SortBy              ::=   'sortby' ' (' SingleVarList ',' Order ')'
|  'sortby' ' (' SingleVarList ')'
ResTuple            ::=   ResTuple ',' ResultItem
|  ResultItem
ResultItem          ::=   EType ':' ElementValue ConstructCond
|  ElementValue ConstructCond
ElementValue        ::=   SingleVar STRCONST
|  NUMBER
|  'min' ' (' ElementValue ')'
|  'max' ' (' ElementValue ')'
|  CollectionVar
|  'sublist' ' (' CollectionVar ',' NUMBER ',' NUMBER ')'
|  'nill'
; ConstructCond     ::=   ' (' SingleValue Operator ElementValue ')'
|  ' (' SingleValue Operator SingleValue ')'
|  ' (' ColValue Operator SingleValue ')'
|
; Operator         ::=   >
                         |  <
                         |  >
                         |  <=
                         |  >=
                         |  ! =
                         |  ~
CollectionVar       ::=   '$$' IDENTIFIER
SingleVarList       ::=   SingleVar
|  SingleVarList ',' SingleVar
EType               ::=   IDENTIFIER
|  '&' IDENTIFIER
|  STRCONST
|  '@' IDENTIFIER
Order               ::=   'asc'
|  'desc'
```

SingleVar	::=	'$' IDENTIFIER
FileName	::=	IDENTIFIER '.xml'

Appendix C

Design of Java Class for XML-RL

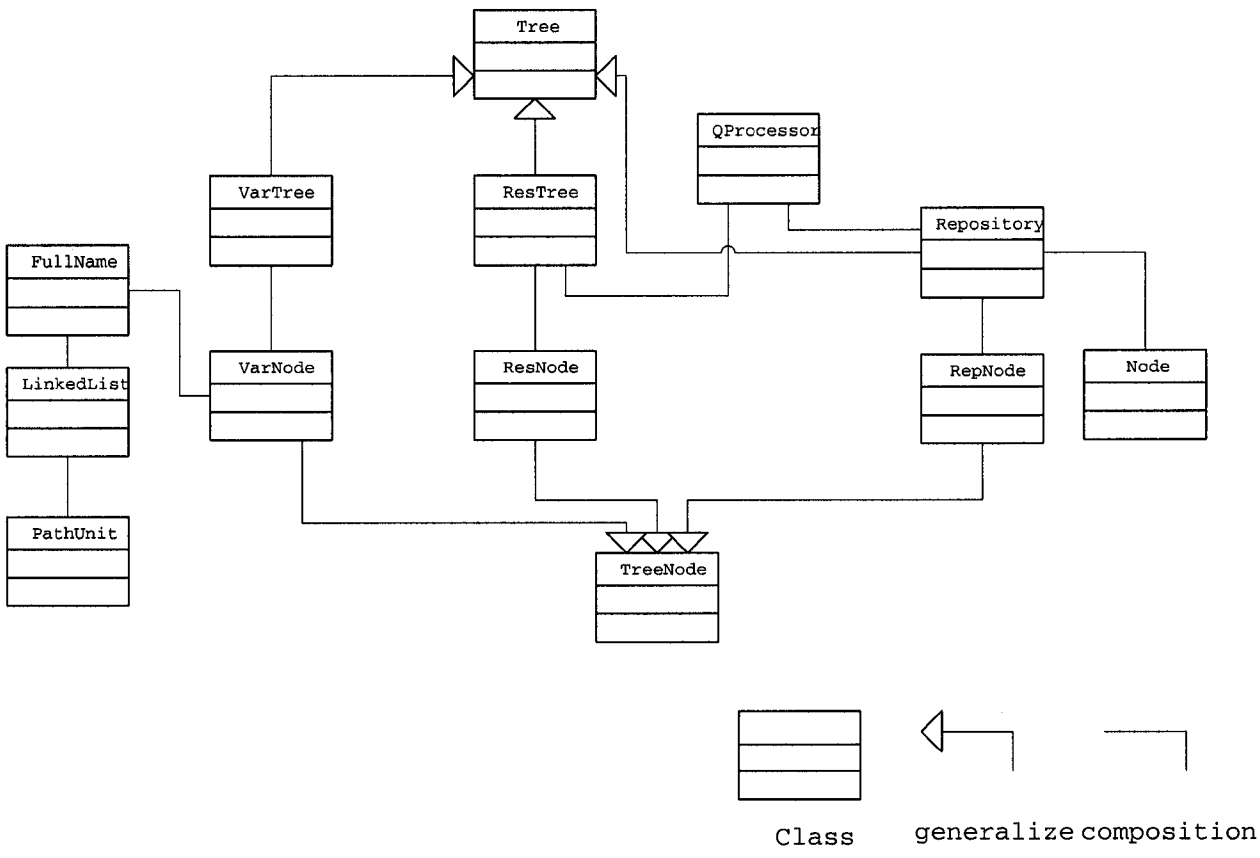


Figure C.1: The Design of Java Class for XML-RL

Appendix D

Exemplar Queries

We tested our language with various queries. But we only list some exemplary queries that illustrate the essential features of the query language. Most of the queries are from the *Query Use Cases* [4] recommended by W3C.

(1) For each author in the bibliography, list the author's name and the titles of all books written by that author, grouped inside a "result" element. This query requires a result in a different structure than the original document. In the result, all book titles of each author needs to be listed, where in the document being queried all authors of each book is grouped and sorted bythe author.

```
bib.xml%/bib/book:$b,
$b/title:$t,
$b/author:$a,
⇒ result%/results/result:
   groupby($a), sortby($a)[author:$a, title:$t ];
```

(2) requests to list books published by AddisonWesley after 1991, including their year and title.

```
bib.xml%/bib/book:  $b,
$b/publisher:$p(="Addison-Wesley"),
$b/@year:$y(>1991),
$b/title:$t
⇒ result.xml%/bib/book:[ @year:$y, title:$t ];
```

(3) asks for a list of all the title-author pairs, with each pair enclosed in a "result" element. The XML-RL query is expressed as follows:

```
bib.xml%/bib/book :$b,
$b/title:$t,
$b/author:$a
⇒ result.xml%/results/result:[ title:$t, author:$a ];
```

(4) asks to list the title and authors inside a "result" element for each book in the bibliography.

```
bib.xml%/bib/book:$b,
$b/title:$t,
$b/author:$$a
⇒ result.xml%/results/result:[ title:$t, author:$$a ];
```

(6) requests to find the minimum price for each book in the document "prices.xml", and return in the form of a "minprice" element with the book title as its title attribute.

```
prices.xml%/book:$b,
$b/title:$t,
$b/price:$p
⇒ result.xml%/results/minprice:groupby($t)[
   @title:$t,
   min($p) ];
```

(5) List the titles and years of all books published by Addison-Wesley after 1991, in alphabetic order.

```
bib.xml%//book:$b,
$b/@year:$y(>1991),
$b/title:$t,
$b/publisher:$p(="Addison-Wesley")
⇒ result.xml%/bib/book:sortby($t,asc)[
   @year:$y,
   title:$t
   ];
```

(7) asks to list the title of the book and its price from each source, for each book found at both bn.com and amazon.com.

```
bib.xml%//book:$b,
$b/price:$bp,
$b/title:$t1 ,
reviews.xml%//entry:$a,
$a/price:$ap,
$a/title:$t2 ( =$t1 )
⇒ result.xml%/book-with-prices/book-with-price:[
   title:$t1,
   prices-amazon:$ap,
   price-bn:$bp ];
```

(8) requests to find books in which some element has a tag ending in "or" and the same element contains the string "Suciu" at any level of nesting. For each such book, return the title and the qualifying element.

```
bib.xml%//book:$b,
$b/&e(endwith "or"):$v(contain "Suciu"),
$b/title:$t
⇒ result.xml%/book:[ title:$t, &e:$v ];
```

(9) In the document "books.xml" find all section or chapter titles that contain the word "XML" regardless of the level of nesting.

```
books.xml%/chapter:$c,
$c//title:$t(contain "XML")
⇒ result.xml%/results/title :$t;
```

(10) For each book with same author(s), return the book with its title and authors.

```
bib.xml%//book:$b1,
$b1/title:$t1,
$b1/author:$$a1,
```

```
bib.xml%//book:$b2,
$b2/title:$t2(<$t1)
$b2/author:$$a2( $$a1),
⇒ result.xml%/bib/book-pair:[ title1:$t1, title2:$t2];
```