NOTE TO USERS

MCGILL UNIVERSITY

DATA INDEXING AND UPDATE IN XML DATABASE

JIAFENG WU
MASTER OF SCIENCE
SCHOOL OF COMPUTER SCIENCE
MCGILL UNIVERISTY, MONTREAL

A THESIS SUBMITTED TO MCGILL UNIVERSITY IN
PARTIAL FULFILLMENT OF THE REQUIRMENT OF
THE DEGREE OF MASTER IN SCIENCE
DECEMBER, 2003

Your file Votre référence
ISBN: 0-612-98762-0
Our file Notre référence
ISBN: 0-612-98762-0

Table of Contents

Acknowledgement

The author would like to express her appreciation to her supervisor Professor Bettina Kemme for the encouragement and guidance in the graduate studies in the School of Computer Science, McGill University, also thanks to Professor Bettina Kemme for her valuable suggestion in the research of this thesis, carefully reading this thesis and correcting it.

Professor Bettina Kemme has very profound academic attainments in database systems and XML data manipulation. The author has learned a lot from her supervision in the study of this project. It is also the author's pleasure to study and work under Professor Bettina Kemme's supervision. All these will produce deep influence on the future study and work of the author.

Finally, the author would like to thank the thesis examiners, department staff, other people and friends in the School of Computer Science of McGill University.

Abstract

XML, the eXtended Markup Language, is well believed to be the most common tool of the future for all data manipulation and data transmission. As a result, a lot of research and work have been done on XML. However, the current efforts on XML only focus on data queries. So far, there does not exist an XML data manipulation language that has reached its maturity to be accepted widely. Most of the existing XML databases only implement some kind of XML data query language and do not support data update operations. This thesis tries to make up the absence of XML data updates, and it begins with a through study on topics such as XML language features, its history and development, the existing XML query languages and update languages, XML database management systems, and XML indexing structures, etc. Based on the research, McXML, a native XML database management system, is proposed, which supports both data queries and data updates, with emphasis on data updates. In order to optimize the performance of McXML, some indexing structures are developed on it, which overcome the difficulties incurred by the support of data updates and make McXML work more efficiently and flexibly.

Résumé

On pense que XML (eXtended Markup Language) va être l'outil du future le plus commun pour toutes les manipulations et transmissions de données. En conséquence, beaucoup de recherche et de travaux ont été menés sur XML. Cependant, les efforts courants sont concentrés seulement sur les requêtes de données. Jusqu'à maintenant, il n'existe pas un langage de manipulation de données XML assez mature pour être accepté largement. La plupart des bases de données XML existantes mettent en application un simple langage de requête de données et ne soutiennent pas des opérations de mise à jour. Premièrement, dans cette thèse, pour remplir le manque de mise à jour de données XML, une grande quantité de recherche a été faite pour donner un aperçu sur les caractéristique du langage, son histoire et développement, les langages d'interrogation et de mise à jour XML existants, des systèmes de gestion de base de données XML, des structures d'indexation XML, etc. Basé sur cette recherche, on propose McXML, un système de gestion de base de données XML primitif, qui soutient des opérations de mise à jour et d'interrogation de données, dont l'accent est sur les mises à jour. Afin d'optimiser l'exécution de McXML, quelques structures d'indexation ont été développées, qui surmontent les difficultés encourues sur le support des opérations de mise à jour de données et qui rend le travail de McXML plus efficace et plus flexible.

Introduction

XML, the eXtended Markup Language, is a meta language that provides a consistent syntax for describing a variety of documents and their structures. It is well believed that XML will be the most common tool for all data manipulation and data transmission in the future. In Chapter 1, a tour is given on XML. The history, syntax rules, structures and advantages of XML are reviewed. The knowledge closely related to XML, such as DOM, DTD, XSD, are also studied and described here.

A lot of efforts have been given to XML recently. For example, many XML query languages, such as XQuery, XQL, XML-QL, XML-GL, XPath and LOREL, for extracting and restructuring the XML content have been proposed, some in the tradition of database query languages, others more closely inspired by XML. Among them, XQuery, which attempts to combine the best features of the leading XML query languages, is expected to become the "SQL of XML". In Chapter 2, features of these XML query languages are studied and compared with one another.

Surprisingly but understandably, XML update languages are not as much studied as XML query languages. So far there are no specifications or standards for XML data updating, and formalisms for XML update languages are currently still underdeveloped. However, some groups have foreseen the need and are working in the field. Tatarinov et.al. of the University of Washington have proposed and developed an extension of the XQuery language to support XML updates, referred as XQuery Extension. Other examples of XML update languages are XUpdate and MMDOC-QL. In Chapter 3, these XML update languages are described and analyzed.

The unresisting XML popularity even attracts the database solution providers' attention. All the well-known ones rapidly XML-enabled their existing products in the hope that this would help them better position their "new" products in the already strenuously competitive database market. The marketing departments at IBM, Microsoft, Oracle and

others have forecast a flourishing XML scene. However, "XML-enabled" databases are not XML databases but relational databases with an added XML interface. In Chapter 4, the research on XML enabled databases is reviewed.

Different from XML enabled databases, native XML databases are designed especially to store XML documents. They support features like transactions, security, multi-user access, programmatic APIs, query languages, and so on. Their internal model is based on XML only. Natix developed by the University of Manheim and Lore developed by the University of Stanford are two examples of native XML databases. In the belief that XML native databases are more promising in manipulating XML data effectively, the author presents her research on Natix and Lore in Chapter 5.

One thing is common to both XML-enabled database systems and the native XML database systems. The tradeoff between efficient query performance versus space and update cost must be considered. Despite the cost of index maintenance, the added storage and the added complexity in the query engine, indices have shown themselves to be useful and integral part of all database systems. Especially in an XML database system, many application of XML data is read-intensive, in which case the balance falls towards maintaining extensive indexing structures to speed up query processing. The research on the XML indexing structures is presented in Chapter 6. Full-text indices, structural indices and hybrid indices together with the related techniques such as compression techniques are described here.

Based on the above-mentioned researches, McXML, the native XML database management system that the author developed under the supervision of Professor Bettina Kemme, is proposed in Chapter 7, and 8. McXML supports XML data updates, as well as XML data queries, with emphasis on data updates. It is implemented using JAVA programming language. It runs on Linux, Solaris and Windows operating systems. The work on McXML has started since July 2002, recently the version 1.0.1 was released.

McXML uses the file systems for physical data storage, XML DOM tree as its logical data model, XQuery for data queries and XQuery Extension for data updates. McXML was designed to be a Client/Server system so to be potentially multi-user and multi-transaction supportive. The McXML Sever handles the business logic. It is powerful in that it not only supports all the update operations that XQuery Extension XML update language specifies but also has its own special features, such as it recognizes IDREF expressions, it handles complex updates containing sub updates well, it has fully implemented functionalities for predicate evaluation, it uses DOM Pooling to make XML data loading more efficient , it developed its own indexing structures to optimize its performance on XML data query, update and commit operations, etc. The McXML client handles application logic. It is very user friendly. The storage, logical data model, XML manipulation languages, and the Client/Server architecture of McXML is described in Chapter 7.

Unlike most of the existing XML indexing structures, which are only designed to optimize XML data queries, and thereof either are not integral to XML database systems that support data updates or would deteriorate the XML data update performance of the systems once integrated, CIndex and QIndex, the two indexing structures on McXML, are both efficient and compatible to XML updates. Their design, implementation and performance are presented in Chapter 8.

Chapter 1

XML

1.1. Overview

To understand the current forces for XML databases, it is instructive to give a brief outline on the language. XML [WXML03], the eXtended Markup Language, is a meta language that provides a consistent syntax for describing a variety of documents and their structures. Due to the great need to share information on the Web, XML is viewed as a way to describe structured data – the sort of information that lives in rows and tables of relational database, and this use of XML for data has been growing exponentially. It is well believed that XML will be the most common tool for all data manipulation and data transmission in the future.

Just like HTML [HTML03], XML is a markup language. Unlike HTML, which was designed to display data and focus on how data looks, XML was designed to describe data and focus on what data is. HTML file contains the data to be presented and the information how the data should be presented (bold, italic, etc.). In contrast, an XML document contains the real data plus the information about the structure of the data.

HTML uses tags to define the presentation of the data. XML uses tags to structure, store and send information. HTML uses only a set of predefined tags, while XML allows the author to define his/her own tags and his/her own document structure. Figure 1-1 shows an example of an XML document, in which tags *<bib>*, *<book>*, *<title>*, *<author>*, *<publisher>*, *<editor>*, *<prices>*, *<first>*, *<last>*, *and <affiliation>* are used to structure and store the information on books. There are two books. The books have author, publisher, editor and price information related to them.

The syntax rules of XML are simple and strict. To understand them well, let's have a close look at the example in Figure 1-1:

- The first line of the XML document is the *XML declaration*, which defines the XML version and the character encoding used in the document. In this case, the document conforms to the 1.0 specification of XML and uses the ISO-8859-1 (Latin-1/West European) character set.
- The main building block of an XML document is *element* represented by a tag, which may have some related *attributes* and child nodes, which are again elements. In *bib.xml*, *<book>* is an element, and it has one attribute *year* as well as five children *<title>*, *<author>*, *<publisher>*, *<editor>*, and *<price>*.
- All XML elements must have a closing tag. XML tags are case sensitive. All XML elements must be properly nested.

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.  <!—This is an example XML document -->
3.  <!DOCTYPE bib SYSTEM 'bib.dtd'>
4.  <bib>
5.    <book year="1994">
6.      <title>Science library</title>
7.      <author publisher = "p1">
8.        <last>Stevens</last>
9.        <first>W.</first>
10.     </author>
11.     <publisher ID = "p1">Addison-Wesley</publisher>
12.  <price>65.95</price>
13.   </book>
14.   <book year="2000">
15.     <title>Data on the Web</title>
16.       <author publisher = "p2">
17.         <last>Abiteboul</last>
18.         <first>Serge</first>
19.       </author>
20.       <author>
21.         <last>Buneman</last>
22.         <first>Peter</first>
23.       </author>
24.       <editor>
25.         <last>Gerbarg</last>
26.         <first>Darcy</first>
27.         <affiliation>CITI</affiliation>
28.       </editor>
29.       <publisher ID = "p2">MIT</publisher>
30.       <price>129.95</price>
31.    </book>
32. </bib>
```

Figure 1-1: bib.xml

- There must be only one *root* element in an XML document, which is an ancestor to all the other elements in the XML document. In *bib.xml*, *<bib>* is the root element.
- In XML, an *attribute* has a *name* and a *value* with it. Attribute values must always be quoted. As in *<book year="1994">* (Line 5 of *bib.xml*), the attribute of the element *book* is named *year* and its value *"1994"* is well quoted.
- In XML, IDREF is a special attribute that allows an element to refer to another element with the designed key. As in *<author publisher = "p1">* (Line 7 of *bib.xml*), the element *author* refers to the element *publisher* by its id *"p1"*, which is defined in *<publisher ID = "p1">* (Line11 of *bib.xml*). Note that the designed keys always have the reserve word *ID* as their attribute name.
- Similar to IDREF, IDREFS is a special attribute that allows an element to refer to more than one other elements with their designed keys.
- XML comments are enclosed between <!-- and -->, as in line 2 of *bib.xml*.

1.2. History of XML

XML was developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996.

XML was derived from the other two makeup languages SGML [SGML03], the Standard Generalized Markup Language, and HTML, the HyperText Markup Language.

Conceived notionally in the 1960s - 1970s, SGML, the international standard for marking up data, has been used since the 80s. SGML is an extremely powerful and extensible tool for semantic markup, which is particularly useful for cataloging and indexing data. However, SGML is complex, expensive, and the commercial browsers made it pretty clear that they did not intend to ever support SGML.

HTML on the other hand was free, simple and widely supported. HTML was originally designed around 1990 to provide a very simple version of SGML, which could be used by "regular" people. As everyone knows, HTML spread like wildfire. Unfortunately, HTML had serious defects, too. There is no single HTML standard. Different browsers support

different/unique HTML tags. Different hardware affects final result. As a result, a HTML based web page design will not necessarily appear the same on every computer.

Hence in 1996, discussions began on how to define a markup language that combines the power and extensibility of SGML with the simplicity of HTML in order to meet the challenges of large-scale electronic publishing. The World Wide Web Consortium (W3C) decided to sponsor a group of SGML gurus including Jon Bosak from Sun.

Essentially, Bosak and his team sliced away all of the non-essential, unused, cryptic parts of SGML. What remained was a lean, mean marking up machine: XML. Nevertheless, all the useful things, which could be done by SGML could also be done with XML.

Over the next few years, XML evolved. By the summer of 1997, Microsoft had launched the Channel Definition Format (CDF) [CEL97] as one of the first real-world applications of XML. Finally, in 1998, the W3C approved Version 1.0 of the XML specification and a new language was born. In October 2002, the W3C released XML Version 1.1.

1.3. Advantages of XML

Compared with SGML and HTML, XML has some advantages. Besides being simpler, it is more flexible, portable, and standardized.

Rather than providing a set of pre-defined tags, as in the case of HTML, XML specifies the standards with which users can define their own markup languages with their own sets of tags. The rules specified by the tags need not be limited to formatting rules. XML allows users to define all sorts of tags with all sorts of rules, such as tags representing business rules or tags representing data description or data relationships. Custom tags bring meaning to the data being displayed, making data extremely portable because it carries with it its description rather than its display.

XML is system, platform and vendor independent. It can be transformed to produce different types of outputs for different media devices (Web browser, paper, CD-ROM)

without the need to modify the original content. XML data exists as plain text, which gives data a longer life span, future readability, and reusability.

XML has open standards.

Due to its advantages, XML is now playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.

1.4. XML Structure

Each XML document has both a logical and a physical structure.
Physically, an XML document contains text, a sequence of characters. In XML, legal characters are tab, carriage return, line feed, and the legal characters of Unicode and ISO/IEC 10646. Text in XML can be parsed (PCDATA) or unparsed (CDATA) data.

Most text in an XML document is parsed data (PCDATA), and will be parsed by the parser. When an XML element is parsed, the text between the XML tags is also parsed, because XML elements can contain other elements, like in *bib.xml*, where the *<author>* element contains two other elements (*<first>* and *<last>*). Text like *<author><last>Abiteboul</last><first>Serge</first></author>* will be parsed and broken up into sub-elements like this:

< author >
<last> Abiteboul </last>
<first> Serge </first>
</author >

In the parsed text section, *entity references* are used to replace illegal XML characters. For example, if a character like "<" is placed inside an XML element, it will generate an error because it would be interpreted as the start of a new element. Such "<" character should be replaced with an entity reference *<*. There are five predefined entity references: *<* for '<' as less than, *>* for '>' as greater than, *&* for '&' as ampersand, *'* for ''' as apostrophe, and *"* for '"' as quotation mark. The author may define his/her own entities in the Data Type Definition (DTD) entity list and reference them in the XML document. Entity references always start with the '*&*' character and end with the '*;*' character. (See Section 1.6 for more information on DTD.)

An XML document may contain unparsed text, which belongs to a CDATA section. Everything inside a CDATA section is ignored by the parser. For example, a paragraph of text (like program code,) that contains a lot of '<' or '&' characters can be defined as a CDATA section. A CDATA section starts with *"<![CDATA["* and ends with *"]]>"*. A CDATA section cannot contain another CDATA section.

Logically, the document contains one or more *nodes*, each of which has a type. Node types are declarations, elements, comments, entity references, processing instructions, notations, etc. As briefly mentioned in section 1.1, e*lement* nodes are the main building blocks that carry the data info. They are identified by name, and may have a set of *attribute* specifications in name/value pairs to provide additional information. All elements are proper nested. As a consequence, the XML document has a mechanism to maintain parent-children relationship among the elements. We will have a closer look at the logical structure of XML documents in the next section.

1.5. XML Document Object Model (DOM)

Closely related to XML documents, the XML Document Object Model (DOM) is a programming interface for XML documents. It defines the way an XML document can be accessed and manipulated. It is designed to be used with any programming language and any operating system. With the XML DOM, a programmer can create an XML document, navigate its structure, and add, modify, or delete its elements.

The DOM represents a logical tree view of the XML document. The root of a DOM tree is a *documentElement*, which has one or many *childNodes* that represent the branches of the tree. XML elements, texts, attributes, entities, notations, processing instructions, etc, are represented as nodes in the DOM tree.

The *Node* object represents a node in the DOM tree. In XML DOM, there are different types of node objects:

- The *Document* object is the root element in the node tree. All nodes in the DOM tree are *childNodes* of the *Document* element. The *Document* element is required in all XML documents.

- The *DocumentType* object in the DOM provides an interface to the list of entities that are defined for the document.
- The *DocumentFragment* is a "lightweight" or "minimal" *Document* object.
- The *Element* object represents the elements in the document. If the element contains text, this text is represented as a *Text* object.
- The *Attr* object represents an attribute of an *Element* object.
- The *Text* object represents the text inside an element as a node.
- The *CDATASection* object represents the CDATA sections in a document. The *CDATASection* object is used to escape parts of text that normally would be recognized as markup.
- The *Comment* object represents the comments in a document.
- The *Entity* object represents the <!ENTITY ..> declarations in an XML Document Type Definition (DTD) entity lists (See Section 1.6 for more infomation on DTD).
- The *ProcessingInstruction* object represents a processing instruction, used in XML as a way to keep processor-specific information in the text of the document.
- The *Notation* object represents a notation declared in the DTD. A notation either declares, by name, the format of an *unparsed entity* [BPS98], or is used for formal declaration of *processing instruction targets* [BPS98] .
- The *EntityReference* object represents an entity reference in the XML document.

A Node object in DOM has some properties, among which the most important are the *name* and *value* properties. Table 1-1 summarizes the name and value properties of different types of Node objects [WDOM00].

According to the Table 1-1, a DOM tree view on *bib.xml* (See Figure 1-1) should be as shown in Figure 1-2. Note that in the figure, there are two nodes named *bib,* the first of which is a *documenttype* node, while the second of which is an *element* node.

Some methods are defined on the Node object. Table 1-2 summarizes the most basic ones [WDOM00].

With these methods, it is easy to access, manipulate, and modify the underlying XML document through its DOM tree.

nodeType	nodeName	nodeValue
element	tagName	null
attribute	name	value
text	#text	content of node
cdatasection	#cdata-section	content of node
entityreference	entity reference name	null
entity	entity name	null
processinginstruction	target	content of node
comment	#comment	comment text
document	#document	null
documenttype	doctype name	null
documentfragment	#document fragment	null
notation	notation name	null

Table 1-1: nodeName and nodeValue of different nodes in XML DOM

Method	Description
appendChild(newChild)	Appends the node newChild at the end of the child nodes for this node
cloneNode(boolean)	Returns an exact clone of this node. If the boolean value is set to true, the cloned node contains all the child nodes as well
hasChildNodes()	Returns true if this node has any child nodes
insertBefore(newNode,refNode)	Inserts a new node, newNode, before the existing node, refNode
removeChild(nodeName)	Removes the specified node, nodeName
replaceChild(newNode,oldNode)	Replaces the oldNode, with the newNode

Table 1-2: methods on Node object

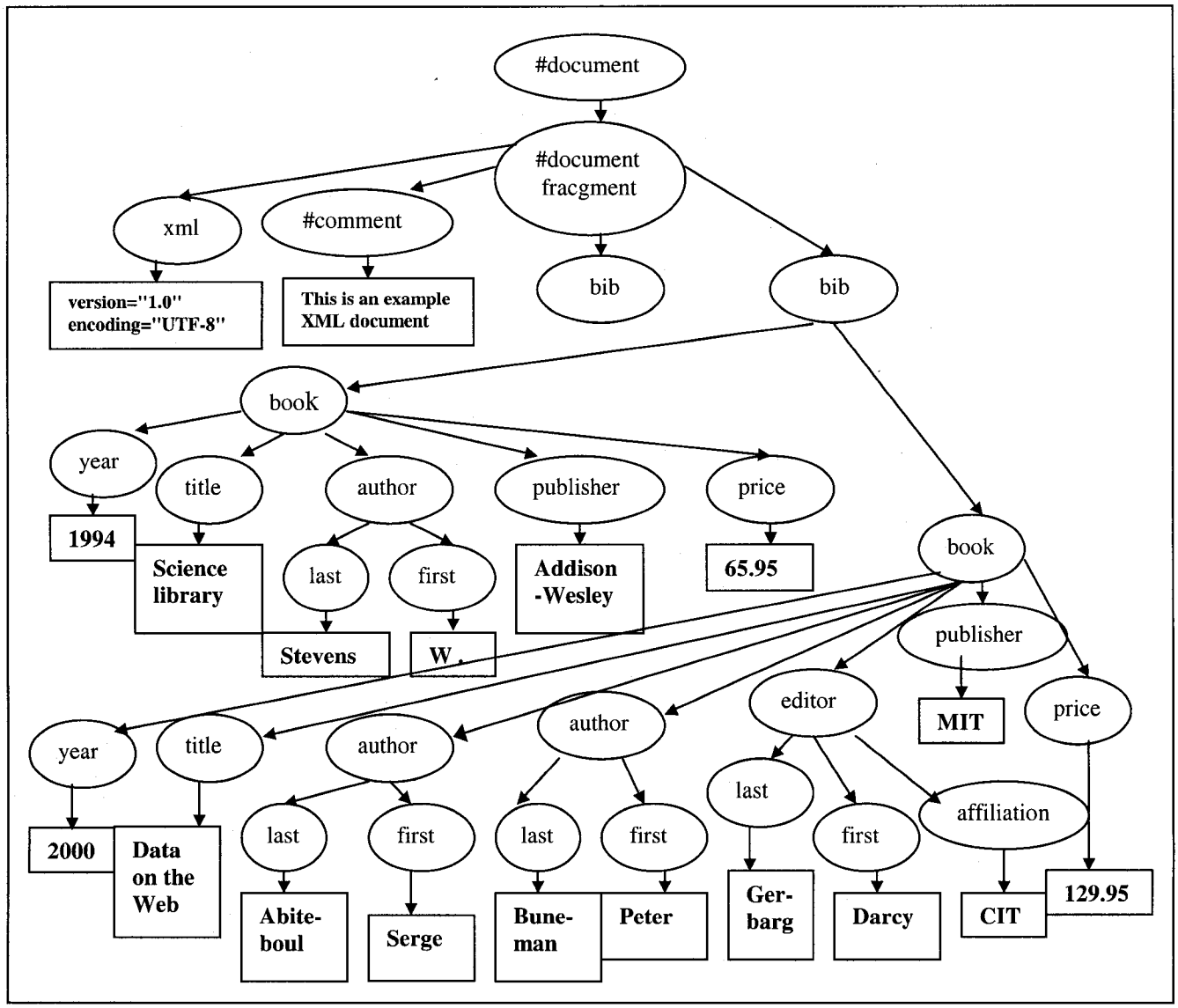


Figure 1-2 The Logical Structure of *bib.xml*

1.6. Document Type Definition (DTD)

A "Valid" XML document is a "Well Formed" XML document, which also conforms to the rules of a Document Type Definition (DTD). The purpose of a DTD is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. With DTD, an XML file can carry a description of its own format with it, and independent groups of people can agree to use a common DTD for interchanging data. A DTD can be declared inline in the XML document, or as an external reference. In

DTD, there are mainly DTD element declarations, ATTLIST declarations, and entity declarations, which declare XML elements, attributes and entity references respectively. For detailed syntax of these DTD declarations, please refer to [RHJ99].

Line 3 in Figure 1-1 contains an external reference to a DTD file *bib.dtd*, which is shown in Figure 1-3. In *bib.dtd*, the element list declares twelve elements: *bib* element is the root element and it contains one or more *book* elements as child nodes (Line 3). *book* elements also has a sequence of child nodes: one *title* element, one or more *author* elements, one *publisher* element, zero or one *editor* element and one *price* element (Line 4). *author* element has two child nodes: a *last* element and a *first* element (Line 6). *editor* element has one *last* element, one *first* element and one *affiliation* element as child nodes (Line 8). *title*, *publisher*, *price*, *last*, *first*, *affiliation* are leaf elements and all of them contain parsed text data (Line 5, 7, 9, 10, 11, 12) . The ATTLIST declares one attribute named *year* that belongs to the element *book* (Line 14). The entity list is empty.

```
1.  <!DOCTYPE bib [
2.
3.  <!ELEMENT bib (book+)>
4.  <!ELEMENT book (title,author+,publisher,editor?,price)>
5.  <!ELEMENT title (#PCDATA)>
6.  <!ELEMENT author (last,first)>
7.  <!ELEMENT publisher (#PCDATA)>
8.  <!ELEMENT editor (last, first,affiliation)>
9.  <!ELEMENT price ((#PCDATA)>
10. <!ELEMENT last (#PCDATA)>
11. <!ELEMENT first (#PCDATA)>
12. <!ELEMENT affiliation (#PCDATA)>
13.
14. <!ATTLIST book year CDATA #REQUIRED>
15.
16. ]>
```

Figure 1-3: bib.dtd

An XML document may have an inline DTD declaration instead, in which case, the DTD is included in the XML source file. The inline DTD should be wrapped in a DOCTYPE definition with the syntax:

<!DOCTYPE root-element [
element-declarations
ATTLIST
entity declarations
] >

For example, we may substitute Line 3 in *bib.xml* shown in Figure 1-1 with the entire text content (Line 1 to Line 16) of *bib.dtd* shown in Figure 1-3 and get a valid XML document with an inline DTD.

XSD is known as XML Schema Language, also referred as XML Schema Definition. XSD is a successor of DTD, and recommended by W3C. XSD defines the legal building blocks of an XML document, just like a DTD. Compared with DTD, XSD has more advantages. XSD supports data types, it uses XML syntax, it is more secure in data communication, and it is more extensible. For more information on XSD, please refer to [XSD99].

Chapter 2
XML Query Languages

As discussed in Chapter 1, XML is becoming more and more important for data representation and exchange on Internet. Languages for extracting and restructuring the XML content have been proposed, some in the tradition of database query languages, others more closely inspired by XML. In the chapter, we present a description to some of these XML query languages, with emphasis on XQuery, a Word Wide Web Consortium recommendation, which attempts to combine the best features of the leading XML query languages and is expected to become the "SQL of XML".

XML documents are logically tree-structured, so in the discussion we use conventional terminology for trees. Particularly, except specified otherwise, we refer to each entity in the XML tree as a *node* that is identical to itself, and not identical to any other, or by its type such as *element*, *attribute*, etc, as discussed in section 1.5.

2.1. XPath

XPath was released as a W3C Recommendation in 1999 as a language for addressing parts of an XML document. XPath is designed to be embedded in a host language such as XSLT [XSLT99] or XQuery [MMA03].

XPath uses a path notation as in URLs for navigating through the hierarchical structure of an XML document. In XPath, a location path can be absolute or relative. An absolute location path starts with the root node, while a relative location path does not. A location path returns the set of nodes selected by the path.

A simplified syntax for XPath is as follows (for full details, the reader may refer to the [JCSD99] paper):

LocationPath : = RelativeLocationPath | AbsoluteLocationPath
AbsoluteLocationPath := "/" RelativeLocationPath ? | "//" AbsoluteLocationPath
RelativeLocationPath := Step | RelativeLocationPath "/" Step |
RelativeLocationPath "//" Step

As ilustrated by the above syntax grammer, an XPath consists of a series of one or more *steps*. A step generates a sequence of nodes and then filters the sequence by zero or more predicates. The value of the step consists of those nodes that satisfy the predicates. This sequence of steps is then evaluated from left to right. Steps in XPath are separated by "/" or "//", and optionally beginning with "/" or "//". A "/" at the beginning introduces a absolute path that contains the context node. A "//" at the beginning establish a relative path that contains all nodes in the same tree as the context node.

XPath supports various kinds of expressions such as numerical expressions (+, -, *, *div, mod*), equality expressions (=, *!=*), relational expressions (<,>,<=,>=), and Boolean expressions (*or*, *and*). XPath defines a library of standard functions for converting and translating data, including string functions (eg. *concat(), start-with(), substring(), etc.*), number functions (eg. *ceiling(), floor(), etc.*), and Boolean functions (eg. *not(), false(), true(), etc.*), and node set functions (eg. *count*(), *last*(), *id*(), etc) that performs on the set of nodes selected by the location path.

Figure 2-1 shows an XPath example that selects all the direct or indirect child elements of <*book*> with name *author*, which has at least one child element <*last*> valued "*Gerbarg*" and at least one child element <*first*> valued "Darcy".

```
/book//author[last = "Gerbarg" and first = "Darcy" ]
```

Figure 2-1: An XPath Example

2.2. *XQuery*

XQuery [MMA03] is a W3C's proposed standard for an XML query language that started its life as Quilt [JCF00], primarily a test vehicle for user-level syntax. Quilt itself was spearheaded and it in turn borrowed features from several other languages, including XPath [JCSD99], XQL [RLS98], XML-QL [DFFL98], SQL [WSQL03], and OQL [MPE02]. XQuery turned out to be a powerful and convenient language in processing XML data, including files in XML format, and other data such as databases whose structure is similar to XML as well.

XQuery is in a sense an *expression language,* for in XQuery everything is an expression, which evaluates to a value. That is, an XQuery program or script is an expression, together with some optional function and other definitions. There are no updates in the XQuery standard yet.

The basic syntax of XQuery is summarized as follows (for detailed syntax, please refer to [MMA03]):

FLWORExpr::= (ForClause | LetClause)+ WhereClause? "RETURN" Expr
*ForClause ::= "FOR" Variable "IN" Expr ("," Variable "IN" Expr)**
*LetClause ::= "LET" Variable ":=" Expr ("," Variable ":=" Expr)**
WhereClause ::= "WHERE" Expr
*Expr ::= ExprSingle (", " ExprSingle) **
ExprSingle := FLWORExpr | QuantifiedExpr | TypeswitchExpr | IfExpr | OrExpr
IfExpr := "IF" "(" Expr ")" "THEN" Expr "ELSE" ExprSingle

As an expression language, XQuery supports different type of expressions. Among them:

- The typeswitch expression (*TypeswitchExpr*) chooses one of several expressions to evaluate based on the dynamic type of an input value.
- The quantified expression (*QuantifiedExpr*) supports existential and universal quantification. An existential predicate over a set of instances is satisfied if at least one of the instances satifies the predicate. A universal predicate over a set of instances is satisfied if all the instances satisfy the predicate. The value of a quantified expression is always true or false.

- A conditional expression (*IfExpr*) is made up of 3 parts: the expression following the *if* keyword is called the *test expression*, and the expressions following the *then* and *else* keywords are called the *then-expression* and *else-expression*, respectively. If the boolean value of the *test expression* is true, the value of the *then-expression* is returned. Otherwise, the value of the *else-expression* is returned.
- A logical expression (*OrExpr*) is either an *and-expression* or an *or-expression*. The value of a logical expression is always true or false.
- XQuery embeds XPath, using it as its path expressions to locate elements or attribute nodes within an XML tree (as explained in section 1.4, an XML document is logically a tree structure).

Now consider a simple XQuery example:

```
FOR $b IN document("bib.xml")//book
WHERE $b/publisher = "Addison-Wesley" AND $b/@year = "1994"
RETURN  $b/title
```

The XQuery queries on the XML document *"bib.xml"*. It first selects the *<book>* elements that have a *<publisher>* child element whose value is *"Addison-Wesley"* , and an attribute *year* whose values is *"1994"*, then it returns the child elements *<title>* of such *<book>* elements.

The data types used in XQuery are:

- Primitive data types

  The primitives data types in XQuery are numbers integer or floating-point, the Boolean values true and false, strings of characters, like *"Hello world!"*, and various types to represent dates, times, and durations. There are also a few XML-related types.
- Node types

  XQuery also has the necessary data types needed to represent XML values. It does this by using *node* values, of which there are seven kinds: *element*, *attribute*, *namespace*, *text*, *comment*, *processing-instruction*, and *document* (root) nodes. These are very similar to the corresponding DOM classes such as Node, Element and so on, as described in section 1.5.

- Sequence

  The primitive data types and *node* types are simple values, while some XQuery expressions actually evaluate to *sequences* of simple values. The comma operator can be used to concatenate two values or sequences. For example, *3,4,5* is a sequence consisting of three integers.

XQuery allows defining variables for later references. A variable is a name that may be bound to a value. In XQuery, a variable is used to represent an instance of data of one of the above mentioned data types. In XQuery a variable may be defined in *ForClause*, *LetClause* and some expressions such as *FLWORExpr*. Variable names are always preceded by a *$* character to distinguish them from string literals.

XQuery supports both standard functions and user defined functions:

- Various standard XQuery functions create, evaluate or return nodes.

  For example, the *document* function reads an XML file specified by a URL argument and returns a document root node. The *element* constructor function creates new node objects directly in the program. The *count* function takes a sequence as an argument and returns the number of values in that sequence. The *children* function returns the sequence of the child nodes of the argument. *Sortby* function takes an input sequence and one or more ordering expressions and returns the sequence sorted according to the values of the ordering expressions. As a strongly typed programming language, XQuery uses *instance of* function to check the data types of the input element. Another convenient function *typeswitch* matches a value against a number of types.
- XQuery wouldn't be much of a programming language without user-defined functions. Such function definitions appear in the *query prologue* of an XQuery program. The following is an example of use-defined function which does what its name suggests:

```
define function descendant-or-self ($x)
{
        $x,
        for $y in children($x)
        return descendant-or-self($y)
}
```

In summary, XQuery is an XML query language that is concise, easily understood and flexible enough to retrieve and interpret information across diverse data sources. As a result, it is getting more and more industry-wide attention and support.

2.3. Other Query Languages

2.3.1. LOREL

LOREL [AQMW97] was developed as the query languages of the Lore prototype data management system at Stanford University, which started as a semi-structured data query language and now is extended to an XML query language. Semi-structured data is data with more structure than a conversation, but less structure than a telephone book. A good example is a resume (curriculum vitae). Semi-structured data is irregular and exhibits type and structural heterogeneity, so LOREL performs type coercion to overcome the inappropriateness of strict typing of OOL in the semi-structured data context. LOREL is a user-friendly language in SQL\OQL style.

LOREL has a powerful support for path expressions to allow flexible navigation access on data. In other words, LOREL is like SQL plus path expression.

The simplified grammar for LOREL selections is (for details, the reader can refer the [AQMW97] paper):

Query := "SELECT" select_expr ("FROM" from_expr)? ("WHERE" where_expr)?

Figure 2-2 shows a simple LOREL query:

```
SELECT X
FROM book.(author | editor).last X
WHERE X = "Gerbarg"
```

Figure 2-2: A LOREL Query

The above query returns all elements under the path *book/author/last* or *book/editor/last*, whose value is "*Gerbarg*".

2.3.2. XML-QL

XML-QL [DFFL98] was proposed in 1998 as a query language for XML data by AT&T labs.

XML-QL can express *queries*, which extract pieces of data from XML documents, as well as *transformations*, which integrate XML data from multiple XML data sources and map XML data using DTDs. XML-QL has a *constructor* operator that builds the document resulting from the query and uses *element patterns* to match data in an XML document.

A simplified syntax for defining a query in XML-QL is:

Query := "WHERE" Predicate "CONSTRUCT" "{" Query "}"

Figure 2-3 shows a typical example of XML-QL query:

```
WHERE <book>
            <title>Science Library</title>
            <author> $a</author>
            <publisher>$p</publisher>
            <editor>$e</editor>
      </book> IN "bib.xml"
CONSTRUCT $n
```

Figure 2-3: An XML-QL query

Informally, this query matches every <*book*> element in the XML document *bib.xml* that has at least one <*editor*> element, one <*author*> element, one <*pubisher*> element, and one <*title*> element that equal to *"Science Library"*. For each such match, it binds the variables *a*, *p* and *e* to every *author*, *publisher* and *editor* group. The result is the list of *books* bound to *n*.

2.3.3. XML-GL

XML-GL was designed at Politecnico di Milano, Dipartimento di Elettronica e Informazione [CCDF99]. It is a graphical query language for XML documents, which uses a visual formalism for representing both the content of XML documents (and of their

DTDs). Both its syntax and semantics are defined in terms of graph structures and operations.

The XML-GL Data Model (XML-GDM) consists of three concepts: *objects* that indicate abstract items without a directly representable value, *properties* that indicate representable values (e.g., a character data or parsed character data string), and *relationships* that indicate semantic associations (e.g., containment or reference), which are respectively represented by rectangles, labeled circles and directed arcs.

XML-GL permits the formulation of queries for extracting information from XML documents and for restructuring such information into novel XML documents. An XML-GL query consists of four parts:

- The *extract* part indicates both the target documents and the target elements inside these documents. With respect to SQL, the extract part corresponds to the *from* clause.
- The *match* part (optional) specifies logical conditions that the target elements must satisfy in order to be part of the query result. With respect to SQL, the condition part corresponds to the *where* clause.
- The *clip* part specifies the child elements, of the extracted elements that satisfy the match part, to be retained in the result. With respect to SQL, the clip part corresponds to the *select* clause.
- The *construct* part (optional) specifies the *new* elements to be included in the result document and their relationships to the extracted elements. With respect to SQL, the construct part can be seen as the extension of the *create view* statement, which permits the user to design a new relation from the result of a query.

Graphically, an XML-GL query is a pair of XML-GDM graphs, displayed side by side and separated by a vertical line; the left-side graph visually represents the *extract* and *match* parts, while the right-side graph conveys the *clip* and *construct* parts.

Figure 2-4 shows an example of XML-GL query, which finds all <*book*> elements with a child element <*affiliation*> in a specified document *bib.xml,* and returns all its child elements <*affiliation*>.

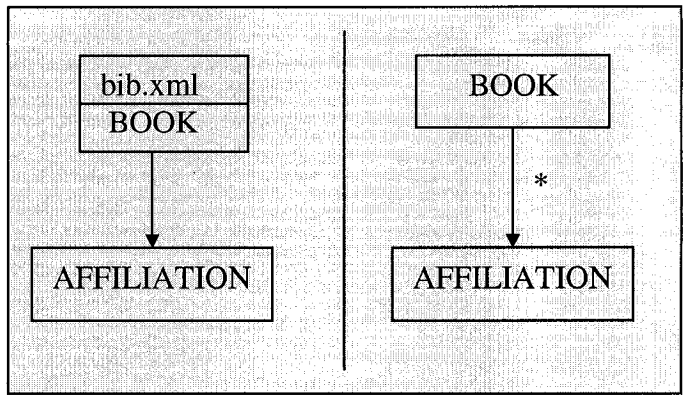


Figure 2-4: An XML-GL Query

2.3.4. XQL

XQL [RLS98] is a query language for XML documents designed by Texcel Inc., webMethods Inc., and Microsoft Corporation.

The basic constructs of XQL correspond directly to the basic structures of XML. XQL is designed to be syntactically simple and compact, using a syntax that may be used in XML attributes, embedded in programming languages, or incorporated in URIs. It allows users to combine information from multiple data sources, use the relationships expressed in links as part of a query, and search based on text containment.

A simplified syntax for XQL is (see [RLS98] for complete syntax):

Query := ("./" | "/" | "//" | ".//")? Element ("[" Predicate "]")? Path?
Path := ("/" | "/")? Element ("[" Predicate "]")? Path?

Figure 2-5 shows an example of XQL query that returns the 1st <*author*> child element of <*book*> element whose attribute *year* is equal to *"2000"*.

```
//book[@year= '2000]/author [1]
```

Figure 2-5: An XQL query

2.4. *Comparison of the XML query languages*

In this section, we present a comparison of the six above-mentioned XML query languages, XQuery, LOREL, XML-GL, XML-QL, XQL, and XPath, highlighting their common features and differences.

The comparison is done in the following categories:

- Data Model: This category shows whether the XML query languages use their own special data model or rely on the data-modeling feature of XML.
- Handling IDREFs: IDREFs can be interpreted as references between elements. This category shows whether the XML query languages define some mechanism for handling IDREFs.
- Support Joins: A join condition compares two or more XML attributes or data belonging to the same document or to two different documents. This category shows whether the XML query languages support joins.
- Support partially specified path expressions: It is convenient to use path expressions. The most powerful form of path expressions does not need to list all the elements of the path, as it uses wildcards and regular expressions. This category shows whether the XML query languages support such partially specified path expressions.
- Grouping result elements: The category shows whether the XML query languages support aggregation or reorganization of elements of the result as specified by means of special functions such as *group by*.
- Support aggregate functions: Aggregate functions compute a scalar value out of a multi-set of values. This category shows whether the XML languages support aggregate functions like *min*, *max*, *sum*, *count*, *avg*, etc.
- Support subqueries: This category shows whether the XML languages support nested subqueries.
- Support set operations: As in SQL, a query can be binary, composed of the union, intersection, or differences of subqueries. This category shows whether the XML languages support such set operations as *union*, *intersect*, *minus*, *except*, etc.

- Order management: Elements of the query result can be ordered differently: ascending or descending, or in the same way as the original document. This category shows whether the XML query languages support ordering elements in the result.
- Type coercion: This category shows whether the XML query languages support implicit data casting among different types as well as the ability to compare the values represented with different type constructors.
- Support for insert, delete and update of element: This category shows weather the XML query languages support XML data updates as well.

The comparison result is summarized in Table 2-1:

	XQuery	**Lorel**	**XML-QL**	**XML-GL**	**XQL**	**XPath**
Data Model	XML implied Model	Its Own Special Data Model	Its Own Special Data Model	Its Own Special Data Model	XML implied Model	XML implied Model
Handling IDREFs	Yes	Yes	No	No	No	Yes
Support Joins	Yes	Yes	Yes	Yes	No	Yes
Support partially specified path expressions	Yes	Yes	Yes	Partially	Yes	Yes
Support aggregate functions	Yes	Yes	No	Yes	Partially	Yes
Grouping result elements	Yes	Yes	No	Yes	No	Yes
Support subqueries	Yes	Yes	Yes	No	No	Yes
Support Set Operations	Yes	Yes	Partially	Yes	Yes	Yes
Order Management	Yes	Yes	Yes	Yes	Partially	Yes
Type coercion	Yes	Yes	No	No	Partially	Yes
Support Data Updates	No	Yes	No	Yes	No	No

Table 2- 1: Comparison of Different XML Query Languages

Chapter 3

XML Update Languages

In order for XML to fully evolve into a universal data representation and sharing format, updates to XML documents must be allowed and techniques to process the updates efficiently must be developed. This calls for XML update languages. So far there are no specifications or standards for XML data updating and formalisms for XML update languages are currently still underdeveloped. XML languages are not as much studied as XML query languages. However, some groups have foreseen the need and are working in the field. In this chapter, we study three XML update languages: XQuery Extension [TIHW01], XUpdate [LMA00] and MMDOC-QL [PLLH01].

3.1. XQuery Extension

Tatarinov et.al. [TIHW01] of University of Washington have proposed and developed an extension of the XQuery language to support XML updates, and they implemented these operations using relational technology. (For convenience, in the following discussion we refer to this extension as *Xquery Extension.*)

XQuery Extension includes a set of constructs for expressing updates in XML documents. These constructs are embedded into the syntax of the XQuery language.

XQuery Extension uses the simplified version of the World Wide Web Consortium's XML Query Data Model [MFJR00], which views an XML document as a node-labeled tree with references, and models all attributes uniformly, including those with specially meanings such as IDREFs.

XQuery Extension describes a set of update operations on XML documents, which are aimed to update not simply scalar or leaf-node values (as in Lorel), but also complex, structured, and irregular types. The update operations take a set of parameters. It also

inherits from XQuery the path expression matching operation that binds the variables to nodes.

In XQuery Extension, an update is a sequence of primitive operations of the following types. They are always operated in the context of a target node.

- Delete(*childnode*): delete the *childnode* from the target node.
- Rename(*childnode*, *newname*): rename the *childnode* of the target node to the *newname*.
- Insert(*newcontent*): insert the *newcontent,* which can be PCDATA, element, attribute or reference, into the target node as a child node.
- InsertBefore(*noderef*, *newcontent*): insert the *newcontent* as a left sibling of the node referenced by *noderef,* which is a child node of the target node.
- InsertAfter(*noderef*, *newcontent*): insert the *newcontent* as a right sibling of the node referenced by *noderef,* which is a child node of the target node.
- Replace(*childnode*, *newcontent*): replace the child node *childnode* of the target node with the *newcontent*.
- Sub-Update(*pathPatternMatch*, *predicates*, *updateOp*): performs sub updates recursively with *updateOp* starting at the target node, on all the qualified bindings filtered by the *predicates* invoking the *pathPatternMatch.*

The above update operations are mapped into the XQuery language syntax. XQuery Extension extends XQuery with a FOR ... LET ... WHERE ... UPDATE structure for updates. The following is the basic form of the update syntax:

```
Update    := FOR $binding1 IN path expression,...
             LET $binding2 := path expression, ...
             WHERE predicate {,predicate}*,
             UpdateOp {,UpdateOp}*
UpdateOp := Update $binding {SubOp {,SubOp}*}
SubOp     := DELETE $child |
             RENAME $child TO name |
             INSERT content (BEFORe|AFTER $child) |
             REPLACE $child WITH $content |
             For $binding IN path expression,
                 WHERE predicate list,
                 UpdateOp {,UpdateOp}*
```

Now consider an example:

```
For $b IN document("bib.xml")//book
 Let $t =  $b/title,
     $a = $b/author,
     $e = $b/editor,
     $p = $b/publisher
Where $b/@year = "2000"
UPDATE $b
{
        DELETE $t,
        RENAME $a To "writer",
        INSERT <edition>1st</edition> BEFORE $e,
        REPLACE $p WITH <published>
                                        <by>Kluwer Academi
                                Publishers</by>
                                        <at>Washington<at>
                                </published>
}
```

The above example updates the *<book>* in *bib.xml* whose attribute *year* is *2000*. It deletes all the child nodes *<title>*, renames its child nodes *<author>* to the new name "*writer*", inserts a new child node *<edition>* before each of its child node *<editor>*, and replace its child nodes *<publisher>* with a new child node *<published>*.

3.2. XUpdate

XUpdate [LMA00], also known as Lexus, is developed by XML:DB organization. It is an XML update language that uses XML format. In other words, an update in the XUpdate is expressed as a well-formed XML document.

XUpdate makes extensive use of the expression language defined by XPath for selecting elements for updating and for conditional processing.

In XUpdate, an update is represented by an *xupdate:modifications* element in an XML document. An *xupdate:modifications* element must have a *version* attribute, indicating the version of XUpdate that the update requires. The value should always be *1.0*. The *xupdate:modifications* element may contain the following types of elements:

- *xupdate:insert-before*
- *xupdate:insert-after*
- *xupdate:append*
- *xupdate:update* :

- *xupdate:remove*
- *xupdate:rename*
- *xupdate:variable*
- *xupdate:value-of*
- *xupdate:if*

The first seven elements have a required *select* attribute, which specifies the node selected by a path expression. This select expression must evaluate to a node-set, which is the target for update.

xupdate:insert-before and *xupdate:insert-after* elements are used to describes XUpdate instructions that directly inserts nodes in the XML result tree. A *xupdate:insert-before* element inserts the new node as a left sibling of the selected node-set, while a *xupdate:insert-after* element inserts the new node as a right sibling. The *xupdate:append* element allows a node to be created and appended as a child of the target node. It may have an optional *child* integer attribute that specifies the position of the new node to be appended. By default, the new child is appended as the last child of the selected target node. The *xupdate:update* element can be used to update the content of the selected target nodes. The *xupdate:remove* element allows a node to be removed from the selected target nodes. The *xupdate:rename* element allows an attribute or a child element node of the selected target nodes to be renamed after its creation.

xupdate:insert-before, xupdate:insert-after and *xupdate:append* elements may contain the following types of elements for constructing new nodes:

- *xupdate:element*: construct a element node
- *xupdate:attribute* : construct a attribute node
- *xupdate:text* : construct a text node
- *xupdate:processing-instruction* : construct a processing instrument node
- *xupdate:comment* : construct a comment node

The following is an example that inserts a new child node <*editor*> into the first <*book*> element in *bib.xml*:

```
<?xml version="1.0"?>
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate">
      <xupdate:insert-after select="/bib/book[1]" >
            <xupdate:element name="editor">
                  <last>Martin</ last >
            <first>Lars</first>
      </xupdate:element>
      </xupdate:insert-after>
</xupdate:modifications>
```

XUpdate supports variable binding by using *xupdate:variable* and *xupdate:variable-of* elements. The *xupdate:variable* element has a required name attribute, which specifies the name of the variable. For example, *<xupdate:variable name="editor" select="/bib/book[0]/editor"/>* binds the selected node to the variable named *editor*. The *xupdate:variable-of* element references a variable by the variable name. For example, *<xupdate:value-of select="$editor"/>* refers to the node bound in the previous example. XUpdate uses *xupdate:if* element for condition processing, which is still an open issue.

3.3. MMDOC-QL

MMDOC-QL [PLLH01] is an XML manipulation language. It has been developed by Siemens Corporate Research, Inc., USA in 2001. In this section, we focus on the update aspect of the language.

MMDOC-QL uses DOM tree as data model and uses Path Predicate Calculus as formalism to support XML update as well as XML query. In short, Path Predicate Calculus uses element predicates as atomic logic formulas and describes XML nodes by specifying path predicates that the tree nodes must satisfy. Element predicates and path predicates are designed for asserting logical truth statements about nodes in an XML DOM tree.

In MMDOC-QL, XML update operations are deletion, insertion, and update of XML structures and content. There are four clauses: operation clauses (GENERATE, INSERT, DELETE, or UPDATE), among which INSERT, DELETE, and UPDATE are used for XML update. The PATTERN clause is used to describe the domain constraints of

variables by regular expression. The FROM clause is used to describe the source document for querying. CONTEXT clause is used to describe logic assertions about nodes. In MMDOC-QL, the variables are indicated by "%".

The following is an MMDOC-QL example that inserts *<editor name="Jerry Ferentino">* as a child node of *<book>* element in *bib.xml* where the attribute *year* has a value between *1990 and 1999*:

INSERT: (<book> with year = %y)
DIRECTLY CONTAINING
(<editor> WITH name="Jerry Ferentino")
PATTERN: {199[0-9]/ %y};
FROM : bib.xml
CONTEXT: {TRUE}

3.4. Summary

The three XML update languages XQuery Extension, XUpdate and MMDOC-QL all support insertion, deletion, update operations on XML documents. By comparison, XQuery Extension has the advantages of following the styles of XQuery. XUpdate is neat in that it uses the XML format. MMDOC-QL seems a little complicated in syntax, but it integrates XML updates easily with XML queries and is sufficiently original in that it created its own calculus for formalism.

Actually, XML update operations can be categorized according to the target node types (leaf, or non-leaf). Some operations can be done only on leaf nodes, such as changing attribute value, changing text node value, etc, which would only affect the data content but not the data structure. Other operations must be done on non-leaf nodes, such as changing element name, changing attribute name (this operation has to be done though the parent node of the attribute), etc., which would affect the XML data structure. Thus these operations might invalidate the DTD, and give rise to some subtle issues in the multi-transaction XML data management environment. An XML update language would be more flexible and integral if it not only implements different XML update operations but also categorizes them. Regretfully, so far no XML update language has accomplished this.

Chapter 4

XML-Enabled Relational Database Systems

It is well believed that in the future a lot of data will be generated, stored, and maintained using XML. Solutions that combine the advantages of XML technology with classical database technology make sense. All the well-known providers of database solutions rapidly XML-enabled their existing products in the hope that this would help them better position their "new" products in the already strenuously competitive database market. The marketing departments at IBM, Microsoft, Oracle and others have forecast a flourishing XML scene. However, "XML-enabled" databases are not XML databases but relational databases with an added XML interface.

4.1. DB2

DB2 supports XML in the base DB2 product, in the XML Extender [IDXE03] and Text Extender [IDTE03], and in its Web services framework (DB2 WORF) [ISVL02]. Support in DB2 itself consists of support for the publishing functions in SQL/XML [ISVL02].

4.1.1. DB2 XML Extender

DB2 XML Extender provides new data types to store XML documents in DB2 databases and new functions to work with these structured documents.

The XML Extender stores XML documents in the database in one of two ways: "XML columns" and "XML collections". XML columns store entire documents as VARCHARs, CLOBs, or files using user-defined types like XMLVARCHAR, XMLCLOB, or XMLFILE. XML collections map non-XML data to an XML document. Two different mappings, SQL mapping and RDB node mapping, are supported. The former can be used only to transfer data from the database to an XML document. It is a template-based

language in which the user specifies SELECT statements and states where the results should be placed within the template. The latter is an object-relational mapping and can be used to transfer data both to and from the database.

The XML Extender provides stored procedures to store and retrieve complete documents or individual elements. The major purpose of these procedures is to translate DB2 data to and from XML documents.

4.1.2. DB2 Text Extender

DB2 Text Extender provides flexible full-text search functionality, using SQL for linguistic functionality. It enables thorough searching of documents where the need is complex and the quality and precision of the search results outweigh the retrieval time. Besides a variety of search technologies such as fuzzy searches, synonym searches, and searches by sentence or paragraph, it can perform proximity search for words within structured XML document sections.

4.1.3. DB2 WORF

The Web services object runtime framework (WORF) provides an environment to easily create simple XML based Web services that access DB2 data and stored procedures. WORF uses Apache Simple Object Access Protocol (SOAP) 2.2 or later and the Document Access Definition Extension (DADX). A DADX document specifies a Web Service using a set of operations that are defined by SQL statements or XML Extender Document Access Definition (DAD) documents. DAD is an XML file that governs the particular tagging scheme and the shape of the XML document in the database.

4.1.4. DB2 SQL/XML Publishing Functions

SQL/XML is an emerging part of the ANSI and ISO SQL Standard, specifying the way SQL can be used to relate to XML and define the core definitions for an XML data type in the SQL language. The built-in SQL/XML publishing functions offer DB2 users and application developers ways to publish XML from DB2 data.

The key SQL/XML functions are scalar constructor functions to build elements and attributes (XMLELEMENT and XMLATTRIBUTES), aggregate function to group child elements (XMLAGG), and subqueries to specify complex nesting/structuring . XMLELEMENT constructs an XML ELEMENT item given an SQL identifier for the tag name, an optional list of expressions for attribute name/value items, and an optional list of values for the content of this element, and return an XML fragment. XMLATTRIBUTES is used to produce an attribute for an element. Each attribute is constructed from an expression and an optional alias. XMLAGG is an aggregate function that can be used to produce a forest of XML elements, with a specific ordering, from a collection of individual elements.

4.2. Oracle

In late 1999, Oracle 8i was released. It was integrated with support for XML to meet the emerging requirements for exchange of data in XML form. However it was a loose integration. The content transformation was performed externally to the database itself. The XML components, an XSQL Servlet, and XML parser and an XSL transform engine, all ran separate form the database itself.

In June 2001, Oracle9i was released. Oracle added XML support directly to the database, this time to improve performance. The existing XML components of Oracle8i were enhanced, and new XML components were added. They enabled the transfers of XML document into and out of database. In addition, two new data types, XML Type and URI-Ref were added for direct XML storage. New table functions were added to fragment XML documents across multiple tables. New SQL operators were introduced to extract data in XML document format using familiar SQL syntax. However in Oracle9i Release 1, XML datatypes must be stored as binary (as CLOB), which limited the flexibility of XML data processing.

Oracle9i Database Release 2 became available in early 2002. It is equipped with a new feature – Oracle XML DB, which is claimed to provide a high-performance, **native** XML storage and retrieval technology. With Oracle XML DB, the W3C XML data model is

absorbed into the Oracle9i Database, navigated and queried with standard access methods. Oracle XML DB is implemented at the database server level. It supports two kinds of XML storage, an XML repository and a native XML Type.

4.2.1. XML Type

In Oracle9i Release 2, the XML Type preserves the structured-data aspect of XML and supports for XML Schema, XPath, XSL-T, DOM, etc. The XML Type is just like any other data type. It allows XML Type data to be stored in either of two ways: with object-relational storage or as CLOB. The storage options are interchangeable. XML data can be either stored as an XML Type column in a relational table or as an XML object in an XML Type table. Non-Schema based XML is always stored as CLOB. Schema based XML can be stored as a CLOB or a set of objects. The constructors of the XML Type allow XML Type to be created from VARCHAR and CLOB. Relational and external data can be exposed as XML views. The view can be a relational view containing a column of XML Type or can be an XML Type view, that is, the view can be constructed over any data, regardless of whether it is relational data or XML data. Inserting into a table with an XML Type column is like inserting into any other table.

Oracle XML DB provides some new SQL operators for performing SQL queries over XML content. These operators help view XML data as relational data and vice versa. extractValue() provides a useful way for viewing XML data as relational data by retrieving the document fragment matching a path expression as an XML Type object. schemaValidate() constrains an XML Type column to a particular XML schema. existsNode() checks if a node specified by an path expression exists. xmltable() creates a table from a set of nodes filtered by a path expression, using a table-based mapping. transform() applies an XSLT stylesheet [XSLT99]. sys_xmlgen() uses a table-based mapping to create one XML document per row from a result set. xmlelement ()creates XML elements.

Figure 4-1 shows an example of XML Type data processing using these SQL operations.

```
SQL> create table BOOK _TABLE
  2 {
  3    BO_NUMBER number(16),
  4              BOOK xmltype
  5 }
  6 /

Table created.

SQL> insert into BOOK_TABLE
2 values (1234, xmltype(
3 '<Book year = "1994">
  4    <title>Science library</title>
  5    <author>
  6       <last>Stevens</last>
  7       <first>W.</first>
  8    <publisher>Addision-Wesley</ publisher >
  9    <price> 65.95</ price >
10 </Book>'))
11/

1 row created.

SQL>  select extractValue(P.BOOK, '/Book/title')
    2    from BOOK_TABLE  P
    3    where existsNodes(P.BOOK, '/Book [publisher = "Addision-
         Wesley"]') == 1
    4    /

EXTRACTVALUE(P.BOOK, '/BOOK/TITLE)
--------------------------------------------------------------------------------
SCIENCE LIBRARY

SQL >
```

Figure 4-1 An Example Of processing XML Type data

4.2.1. XML Repository

The Oracle XML DB repository is designated for "content-oriented" (semantic) data access. It makes it possible to use a familiar file/folder metaphor to store, organize and access XML content stored in the database. All meta-data is managed by the Oracle XM DB repository. All content, other than schema-based XML, is stored in the repository. XML Type objects (regardless of whether they actually contain XML data or are just XML views over relational data) can be assigned a path and a corresponding URL in the repository hierarchy. These can then be accessed via WebDAV, FTP, JNDI, and SQL. There are some special SQL operators for this purpose. In addition, the repository maintains properties for each object, such as owner, modification date, version, and access control.

Chapter 5

Native XML Database

5.1. Overview

Native XML databases are designed especially to store XML documents. Like other databases, they support features like transactions, security, multi-user access, programmatic APIs, query languages, and so on. The only difference from other databases is that their internal model is based on XML and not something else, such as the relational model.

In detail, according to XML:DB Initiative [XDBI03], a Native database is one that:

- Defines a (logical) model for an XML document -- as opposed to the data in that document -- and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order. An example of such models is the models implied by the DOM.
- Has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.
- Is not required to have any particular underlying physical storage model. For example, it can be built on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files.

Native XML databases differ from XML-enabled databases (as discussed in the previous chapter) in three main ways:

- Native XML databases can preserve physical structure (entity usage, CDATA sections, etc.) as well as comments, PIs, DTDs, etc. While XML-enabled databases can do this in theory, this is generally not done in practice.
- Native XML databases can store XML documents without knowing their schema (DTD), assuming one even exists. Although XML-enabled databases could generate

schemas on the fly, this is impractical in practice, especially when dealing with schema-less documents.

- The only interface to the data in native XML databases is XML and related technologies, such as XPath, the DOM, or an XML-specific API. XML-enabled databases, on the other hand, offer direct access to the data, such as through ODBC.

It is well believed that native XML databases that store XML in "native" form are more promising in manipulating XML data effectively. In this chapter, we look at two native database systems: Lore and Natix.

5.2. LORE

Lore [WBGA01] had been under development at Stanford University since 1995 and was declared as a success in 2000. It is a database originally designed for storing semi-structured data and has been migrated for use as an XML database, for XML is well-suited to storing semi-structured data and shares a feature common to many semi-structured data models: it is self-describing.

5.2.1. Lore's query language

Lore includes a query language LOREL, as described in section 2.3.1.

5.2.2. Lore's data model

Lore has an XML-based data model, an XML element is a pair<*eid*, *value*>, where *eid* is a unique element identifier, and *value* is either an atomic text string or a complex value containing the following four components [GMW99]:

- A string-valued tag corresponding to the XML tag for that element
- An ordered list of *attribute-name/atomic-value* pairs, where each *attribute-name* is a string and each *atomic-value* has an atomic type drawn from *integer*, *real*, *string*, etc., or ID, IDREF, or IDREFS.
- An ordered list of *crosslink subelements* of the form<*label*, *eid*>, where *label* is a string. *Crosslink subelements* are introduced via an attribute of type IDREF or IDREFS.

- An ordered list of *normal subelements* of the form<*label*, *eid*>, where label is a string. *Normal subelements* are introduced via lexical nesting within an XML document.

Following some rules, an XML document can be easily mapped into the Lore data model and visualized as a directed, labeled, ordered graph:

- Text between tags is translated into an atomic text element.
- An XML element is translated into a complex data element.
- The tag of the data element is the tag of the XML element.
- The list of the *attribute-name/atomic-value* pairs in the data element is derived directly from the XML element's attribute list.
- For each attribute value *i* of type IDREF in the XML element, or component *i* of an attribute value of type IDREFS, there is one crosslink subelement <*label*, *eid*> in the data element, where *label* is the corresponding attribute name and *eid* identifies the unique data element whose ID attribute value matches *i*.
- The subelements of the XML element appear, in order, as the normal subelements of the data elements. The label for each data subelement is the tag of that XML subelement, or **Text** if the XML subelement is atomic.

With the Lore data model, XML data can be viewed in two modes: *semantic* or *literal*. Semantic mode is used when the user or application wishes to view the database as an interconnected graph, where attributes of type IDREF or IDREFS are omitted, and the distinction between crosslinks and subelements are gone. In literal mode, the user views the database as an XML document. Crosslinks are invisible. The database is a tree.

Figure 5-1 shows a sample of XML and the graph representation in Lore data model. Element identifiers (*eids*) appear within nodes and are written as &1, &2, etc. Attribute-name/atomic-value pairs are shown next to the associated nodes and are surrounded by { }, with IDREF attribute in italic. Subelements are solid and crosslinks are dashed. In semantic mode, the database in Figure 5-1 does not include the IDREF attributes. In literal mode, the crosslink are not included.

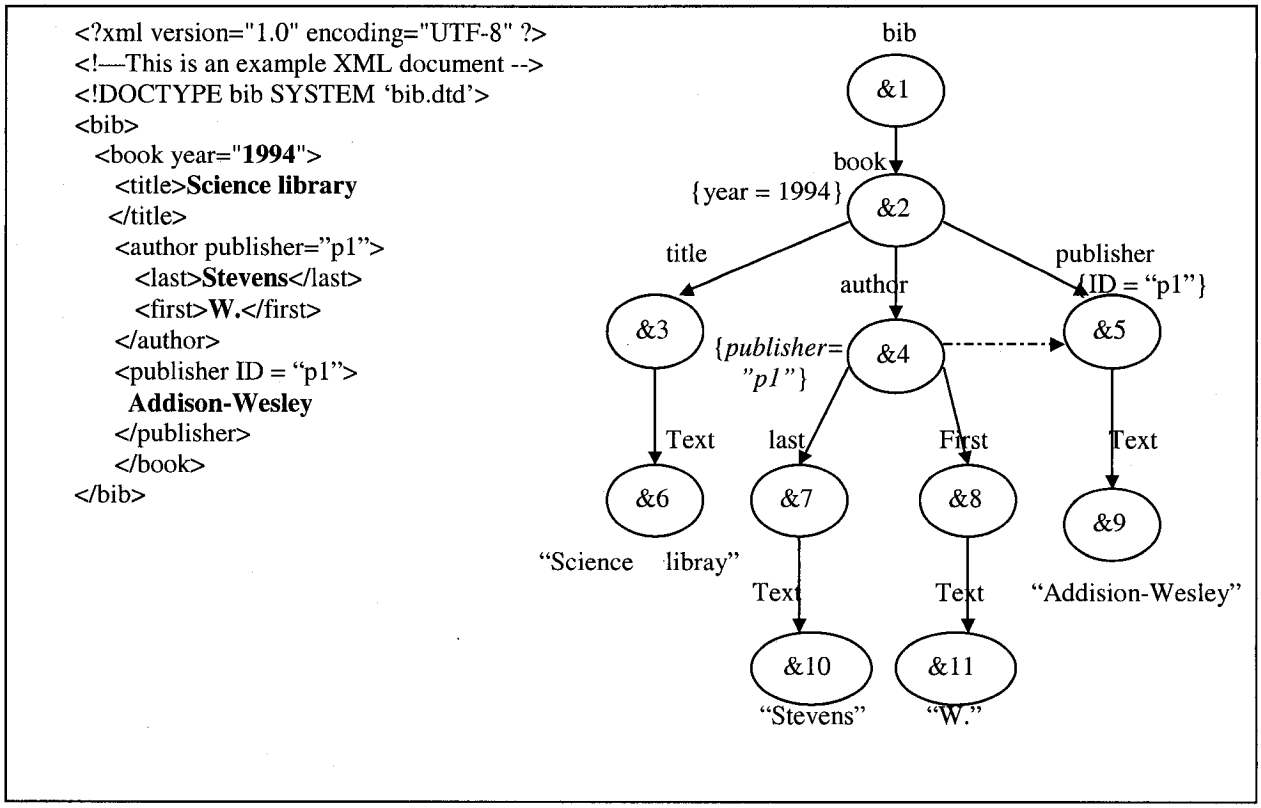


Figure 5-1: An XML document and its Lore data model graph

5.2.3. Other features

Lore has four kinds of indexing structures [MWAL98], Vindex, Pindex, Lindex and Tindex, which we will discuss in Chapter 6. Lore uses its cost-based query optimizer to piece together the available indices to create efficient query plans. Lorel also has multi-user support, logging, recovery, and novel technologies such as DataGuides, management of external data, and proximity search. For detailed information, please see [WBGA01].

5.3. Natix

Natix [FHKM02] is a database management system for storing and processing XML data. It is developed by University of Manheim based on several master & PhD thesis. It is developed in C++ on Unix. The system is quite ambitious. It has an architecture with different layers. It develops an efficient way of storing data with a B-tree like data structure. It supports XPath and XQuery query languages. It has two kinds of data indexing to fasten the data processing. What's more, its transaction management

comprises recovery and multi-user synchronization. However, Natix does not support data updates so far.

5.3.1. Architecture

Natix's components form three layers, as shown in Figure 5-2. The bottommost layer is the storage layer, which manages all persistent data structures. On top of it, the service layer provides all DBMS functionality required in addition to simple storage and retrieval. These two layers together form the Natix engine. Closest to the application is the binding layer, which consists of all the modules that map application data and requests from other application programming interfaces to Natix Engine interfaces and vice versa.

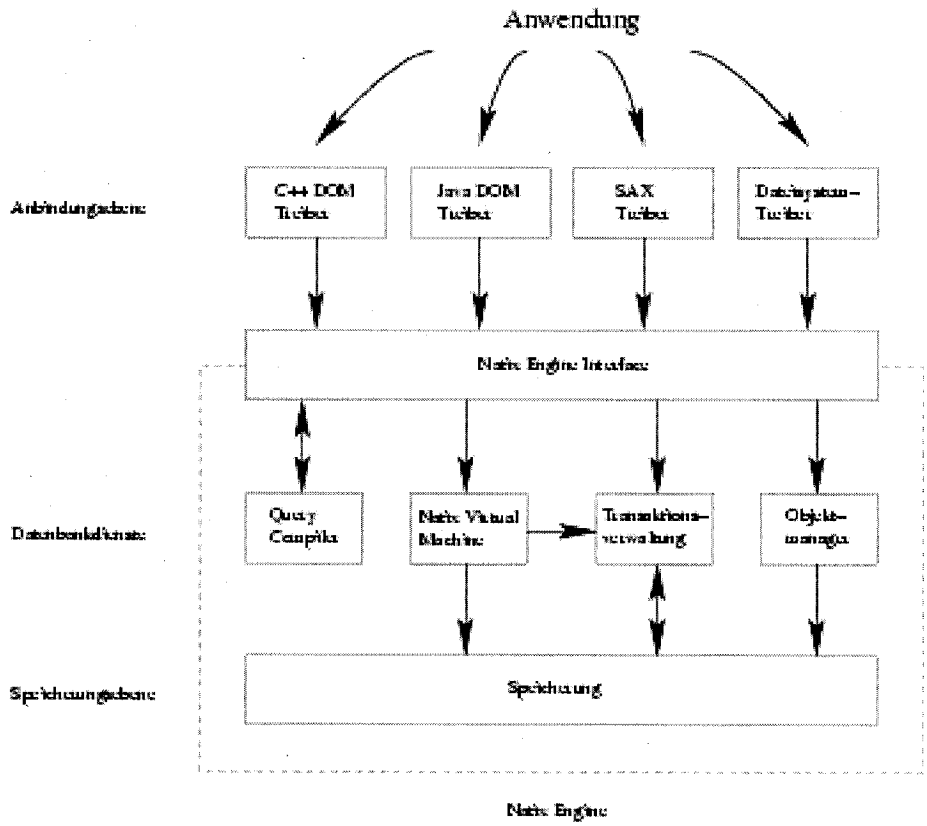


Figure 5-2: Architectural Overview of Natix

Among the service components that implement the functionality needed for the different request are:

- *The Natix Query Execution Engine (NQE)* is used to efficiently evaluate queries. It contains the *Natix Physical Algebra (NPA)* and the *Natix Virtual Machine (NVM).*

- NPA exists to support query specification. NPA supports the standard algebraic operators, such as binding, combination, and selection borrowed from relational context. From the object-oriented context, it supports some operators like the d-join and the unary join and binary grouping operators. NPA also provides several scan operations to generate variable bindings for path expressions.
- NVM interprets commands on register sets. NVM tries to avoid unnecessary copying as much as possible by managing register sets carefully. NVM can interpret more than 1500 commands.

- The *Query Compiler* translates queries expressed in XML query languages into execution plans for NQE.
- The *Transaction Manager* contains classes that provide ACID-Style transactions. Components for recovery and isolation are located here. In my opinion, this is a feature for the future, for at present, Natix does not support XML updates.
- The *Object Manager* factorizes the representation-independent management of documents and their components nodes.

5.3.2. Natix's data storage

There are three approaches to store XML documents, *Flat Streams*, *Metamodeling* and *Mixed* [FHKM02]:

- In the *Flat Stream* approach, the documents are serialized into byte streams. For large streams, some mechanism is used to distribute the byte streams on disk pages. This method is very fast when storing or retrieving whole documents or big continuous pars of documents. Accessing the documents' structure is only possible through parsing.
- *Metamodeling* stores the documents or data trees using some conventional DBMS and its data model. In this approach, interacting with structured databases in the same DBMS is easy, while scanning a whole document is slower. Also complex mapping operations are needed to reproduce a textural representation, as a result query processing is slowed down.

- *Mixed* approach merges the first two methods in two ways, *redundant* and *hybrid* merging:
    - In *redundant* approach, data is held in two redundant repositories, one flat and one metamodeled. This allows fast retrieval, but leads to slow updates and incurs significant overhead for concurrency control.
    - In the *hybrid* approach, a "threshold" is maintained. Structures coarser than this granularity live in a structured part of the database; finer structures are stored in a "flat object" part of the database.

The storage organization of Natix, *Native XML Repository*, is similar to the hybrid approach, with two extensions: First, the "flat" parts of the database are not completely flat, but clustered groups of tree nodes treated as atomic records by the lower level of Natix. Second, the "threshold" can be a dynamic value, adapted to the size and structure of documents at run time. In Figure 5-3, the different modules of the storage subsystem of Natix and their call relationships are shown.

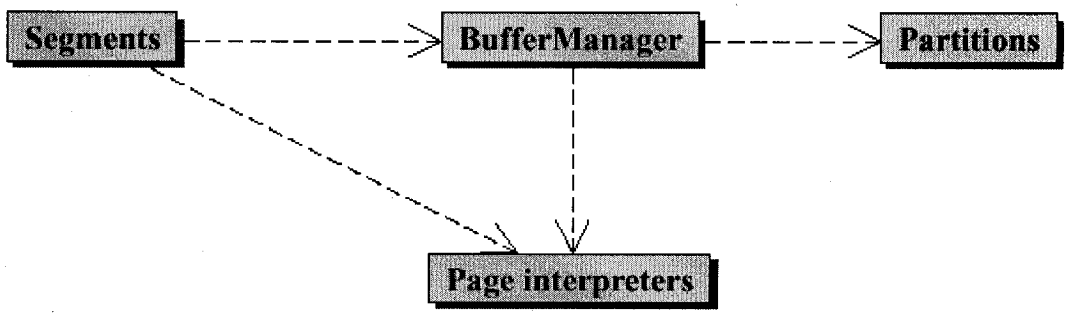


Figure 5-3: Storage Engine Architecture

Storage in Natix is organized into *partitions*, which represent an abstraction of random-access block storage devices that can randomly read and write a fixed number of disk pages.

Disk pages are grouped in *segments*. Segments implements large, persistent object collections. Disk pages resident in main memory are managed by the *buffermanager*, which is responsible for transferring pages between main and the secondary memory, and synchronizing page access by multiple threads by means of a latch.

The content of disk pages are accessed using *page interpreters*. While a page resides in main memory, it is associated with a page interpreter object that abstracts from the actual data format on the page. The existence of page interpreters separates intra-page data structure management from inter-page data structure.

5.3.3. Natix's data model

Natix uses ordered trees as logical data model, which is very similar to XML DOM tree (as discussed in section 1.5). The logical data tree is materialized as a physical data tree, which contains the original logical nodes and additional nodes needed to manage the physical structure of large trees. In a physical data tree, there are three kinds of nodes. *Aggregate* nodes are inner nodes of the tree. *Literal* nodes are leaf nodes. *Proxy* nodes are nodes that point to different records. They are used in representation of large trees.

In Natix, whole documents (or subtrees of documents) are stored together in one atomic record. The record size has an upper limit, the page size. So the physical model provides a mechanism for distributing data trees over several pages. Large documents are semantically split based on the underlying tree structure. The data tree is partitioned into subtrees, which are stored in a single record less than a page in size. Proxy nodes, which consist of the RID of the records, which contains the subtree they present, are added to connect subtrees residing in other records. Substituting all proxy nodes by their respective subtrees reconstructs the original data tree. A sample is shown in Figure 5-4:

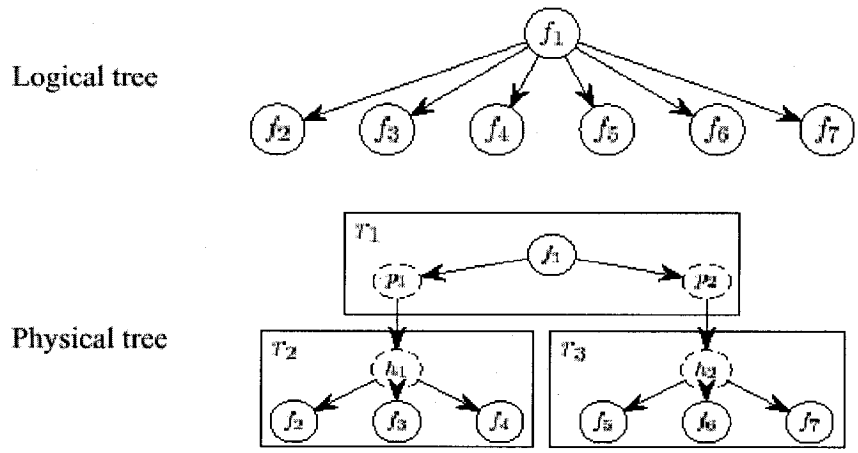


Figure 5-4: One possibility for distribution of logical nodes on records

Chapter 6

XML Database Indexing Structures

In any DBMS, the tradeoff between efficient query performance versus space and update cost must be considered. Indexing allows fast access to data by essentially replicating portions of the database in special-purpose structures. However, these structures must be kept up-to-date incrementally: each change to the base data must be reflected in all applicable indices. Despite the cost of index maintenance, the added storage and the added complexity in the query engine, indices have shown themselves to be useful and integral part of all database systems. Especially in an XML database system, many application of XML data is read-intensive, in which case the balance falls towards maintaining extensive indexing structures to speed up query processing.

In this chapter, we have a look at various XML data indexing structures: full-text indices, structural indices and hybrid indices.

6.1. Three Types of XML Indexing Structures

- **Full text indexing** methodologies, like the inverted file [PEB03] and signature file approaches [DKM96], enjoy applicability in the modern Information Retrieval (IR) environment. The inverted file approach is characterized by its efficiency in text retrieval operations whereas the signature file involves a simple structure and requires significantly less storage overhead.
- **Structural Index** is used to support queries of structural XML query languages such as XQuery or XPath, which makes regular path expressions part of the search criteria. Indexing the structure of the XML data is import for search engines as well as IR.
- **Hybrid Index** is useful in the environment where it is advantageous to combine full-text index with structural index.

6.2. *Some Indexing Structures*

6.2.1. Lore's Indexing Structures

Lore has four kinds of indexing structures [MWAL98]:

- Indexing atomic values (*Vindex*) in the graph-based data model allows the query engine to quickly locate specific leaf objects. Vindex is implemented as a B+ tree structural index to support inequality as well as equality lookups.
- Taking into consideration that almost all queries also explore the data via labelled traversals through the graph, Lore also introduces two other structural indices: (*Pindex*) and link index (*Lindex*) that efficiently locate paths and edges through the data.
- *Lindex* provides a mechanism for retrieving the parents of a node via a given label. A Lindex takes a child node c and a label l, and returns all parents p such that there is an l-labelled edge from p to c. Lindex is implemented using *extendible hashing* since Lindex always does equality lookups.
- A *Pindex* lookup for a path p returns the set of nodes O reachable via p. Currently Pindex does not support regular expressions. Pindex is implemented using Lore's DataGuide, where the set of reachable nodes for each path are stored. DataGuide is a "structural summary of all paths in the database". Unlike structured databases, in which the structure is specified first and data is added according to that structure, data is entered first into Lore and the structure is then summarized.
- Lore also includes a simple full-text indexing system (*Tindex*) that efficiently supports information retrieval style predicates within Lore's query language. Tindex is implemented using *inverted list* [PEB03], which maps a given word w and label l to a list of atomic values with incoming edges l that contain word w.

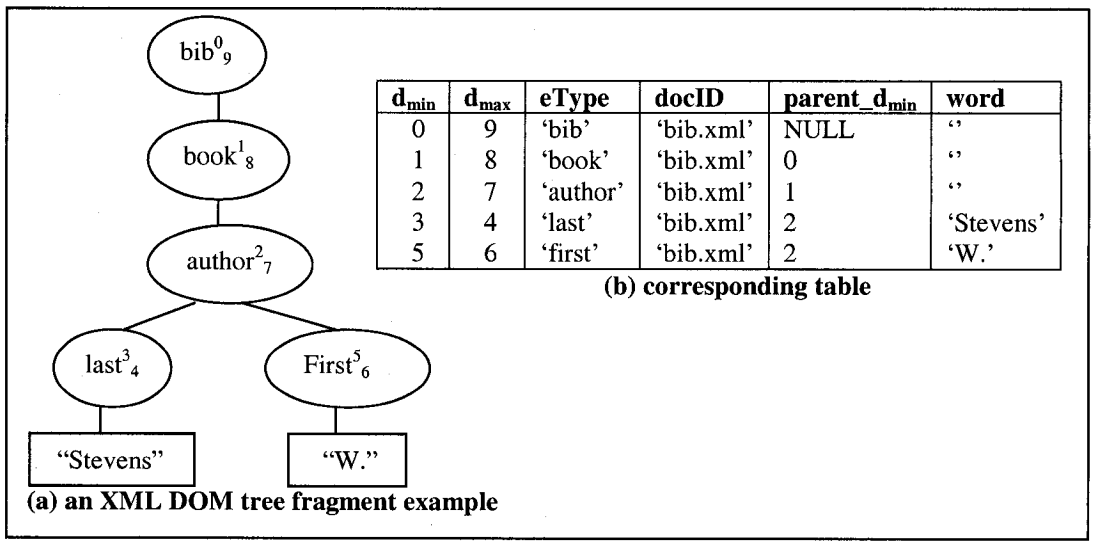
6.2.2. Natix's Indexing Structures

Natix developed powerful indexing structures to support query evaluation [JMI01]. It enhanced a traditional full-text index in such a way as to be able to cope with semi-structured data. It also has a structural index called XSAR (eXtended Access Support Relations).

Natix establishes its full-text index by using inverted files [PEB03] with Elisa Gamma Coding Compression technique [POJ00]. In Natix, the full-text index framework offers two major functionalities: bulkLoad() and find(). The former accomplishes an efficient inserting of large data sets to build indices on words. The latter checks data in the index.

XASR is an index structure proposed by the University of Manheim as a component of a search engine for XML data called Mumpits [TFGM00], which supports structural query on XML data. XASR is also integrated into Natix to fasten data queries.

XSAR works on top of a relational DBMS. It has a XSAR table stored in RDBMS that has the attributes *docID*, *eType* (name of the tag), *word* (search term), *dMin*, *dMax*, *parent_dMin*, etc. Among them, dMin is a number value assigned to a non-text-typed node when we enter the node for the first time in depth-first traversal. dMax is the number value when we return from the traverse back to this node. Each node of the document tree has an index tuple stored in the XSAR table.



(a) an XML DOM tree fragment example

d_{min}	d_{max}	eType	docID	parent_d_{min}	word
0	9	'bib'	'bib.xml'	NULL	''
1	8	'book'	'bib.xml'	0	''
2	7	'author'	'bib.xml'	1	''
3	4	'last'	'bib.xml'	2	'Stevens'
5	6	'first'	'bib.xml'	2	'W.'

(b) corresponding table

Figure 6-1: XSAR table

A path in a query is translated into a sequence of SQL-joins on the XSAR table. For example, let X_{i+1} and X_i be two nodes in the same path. Depending on the path connector ('/' or '//'), the join predicate of the SQL query is

$X_i.docID = X_{i+1}.docID$ *and*
$X_i.dMin = X_{i+1}.parent\_dMin$, *('/')*
Or
$X_i.docID = X_{i+1}.docID$ *and*
$X_i.dMin < X_{i+1}.dMin$ *and*
$X_i.dMax > X_{i+1}.Max$. *('//')*

Querying on the XSAR table attribute *word*, XSAR allows looking for nodes containing. certain words. Querying on the attribute *eType*, XSAR allows looking for nodes of certain type.

6.2.3. A Hybrid Index Structure

In [EKO02], a hybrid-indexing mechanism is proposed. The approach combines the inverted file with a path index:

- Organize index structure by setting up the summary tree, removing any path and text duplications. The resulting summary tree is similar to and, in principle, smaller than the original XML DOM tree. Just Like Lore's DataGuide, it is a "structural summary of all paths" of the XML data. Figure 6-2 shows the summary tree of *bib.xml* (See Section 1.1).
- Load the summary tree into the index structure. This involves the separation of content data from path data. The former is raw text data aimed to be stored in the inverted file and the latter is the structural text aimed to be stored in the path index (See Figure 6-3).
    - Build the path index with path data. The path index is a hierarchy of tags, which records every single path in the collection.
    - Create the inverted file that stores the literal content of document nodes by adding all the content texts to the vocabulary of the inverted file.
    - Establish necessary link between the nodes in the path index with their corresponding literal content in the inverted file.

The algorithm of using the hybrid index to fasten the query evaluation is:

- Decompose the received query into its constituent conjunctive and disjunctive terms.
- Separate each term into a path part and a raw text part.

- The path is checked against the path index. A candidate list *A* of documents, which has matching inverted lists, and a candidate list *T*, which contains all the vocabulary terms that happens to be at the end of this path, are returned.
- The literal part is checked against the candidate terms in *T* and for those terms that there is a match, the document entries are retrieved from the inverted lists and a second candidate set of documents *B* is returned.
- The answer is the intersection of *A* and *B*. s

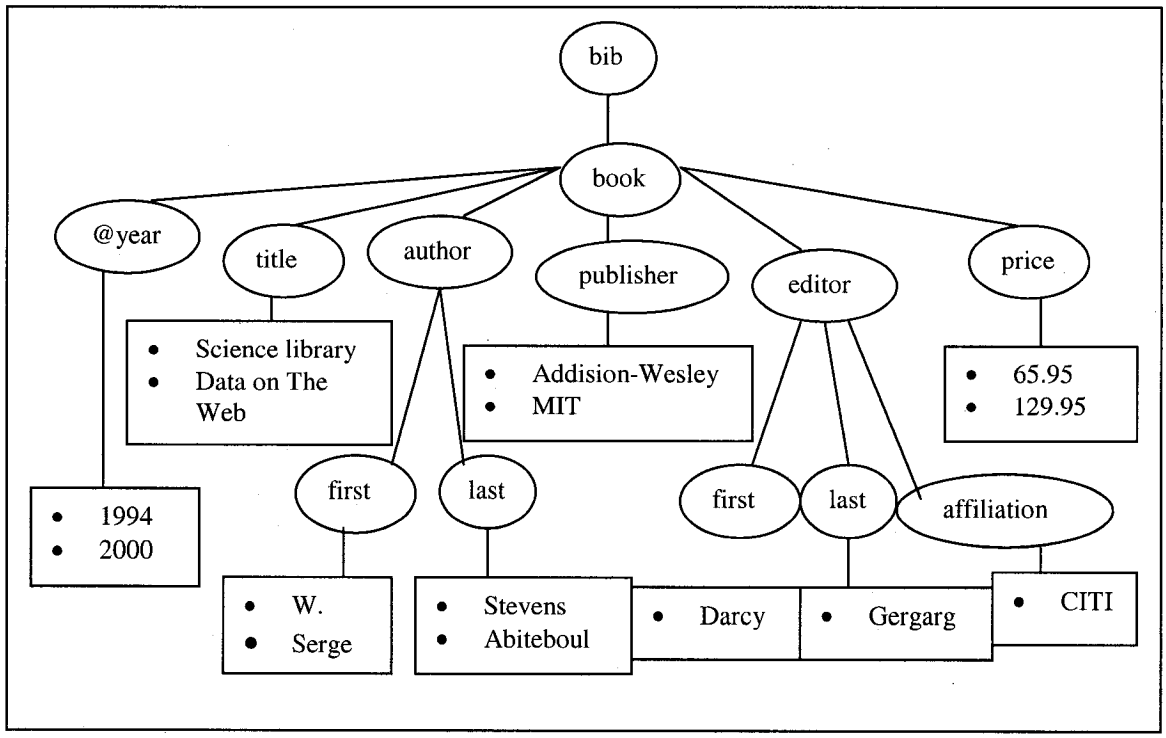


Figure 6-2: An example of Summary Tree on bib.xml

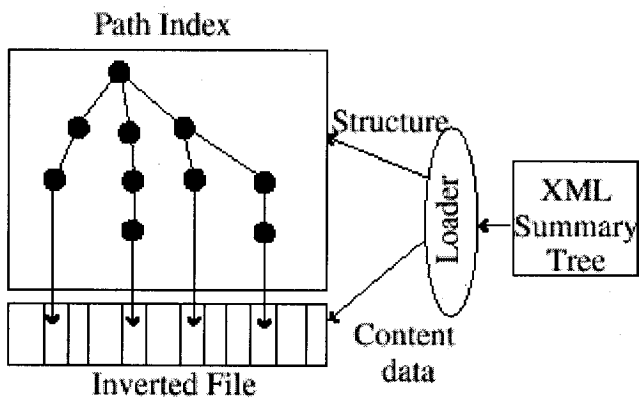


Figure 6-3: Loading the summary tree into the index structure

Chapter 7

McXML

-- A Native XML Database Management System Emphasizing Data Updates

7.1. Overview

McXML, a Native XML database management system, was developed by the author as the thesis research project. The work on McXML has started in July 2002. Recently, Version 1.0 was released.

McXML was designed as a Client/Server system so to be extendable to support multi-users. It is a breakthrough, because unlike most of the existing native XML databases, it supports XML data updates, as well as XML data queries, with emphasis on data updates. McXML is also equipped with its own indexing structures, which optimizes its performance on XML data queries, updates and commits. McXML is implemented using JAVA programming language. It runs on Linux, Solaris and Windows operating systems.

7.1.1. Physical Storage Model

Most XML DBMS are built on relational DBMS. We prefer to have a native XML DBMS. Hence, McXML built its physical storage model directly on top of the file systems. McXML has a root storage directory, under which each user of McXML is granted a sub directory as *user account* whose name is the same as the *username*. Under the user account directory, special system directories and files like *"/.index"*, *".pswd"*, *".quota"*, etc. are established to store index files, *user password*, *account quota*, etc. respectively. The user is authenticated by the username and password, and may store and manage XML data in the account. The quota specifies the size limit of the account, that is,

the user may only store as much XML data as the quota allows. Besides being simple and native, this approach is advantageous in that it saves physical data storage space.

7.1.2. Data Model

McXML uses XML DOM tree as its logical data model. For detailed information on XML DOM tree, please refer to Section 1.5.

7.1.3. XML Query & Update Languages

McXML supports both XML data updates and XML data queries. For data queries, McXML implements the XQuery language. For data updates, McXML implements XQuery Extension. For detailed information about these two languages, please refer to Section 2.2 and Section 3.1.

7.2. Client/Server Architecture of McXML

McXML is designed as a Client/Server architecture. This is to make McXML component-based, extendable, and enabled to support multiple clients. Figure 7-1 shows the architecture diagram of McXML:

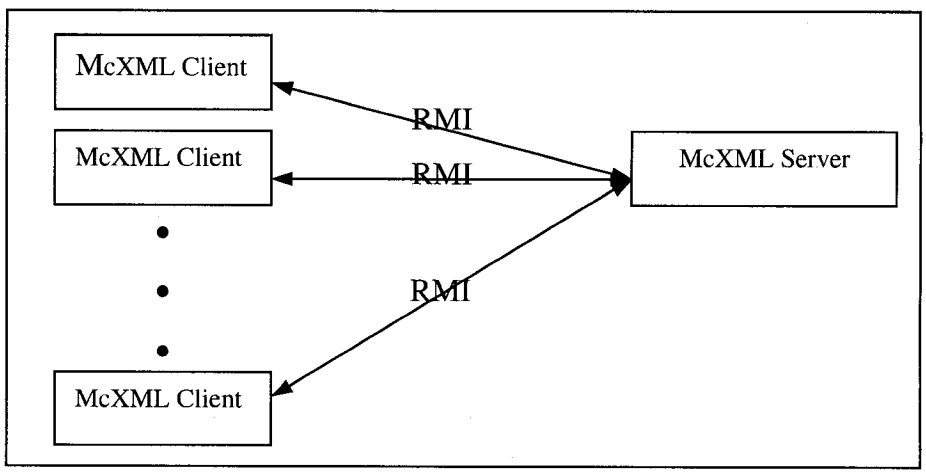


Figure 7-1: Client/Server Architecture of McXML

As a Client/Server system, McXML supports multiple clients. The McXML clients can run on various sites. The McXML server and the McXML clients communicate with one

another by means of Jave RMI [AWJW00]. All the functionalities for XML data processing including data queries, updates, indexing, storage, etc. are located on the McXML server. This means the McXML client is rather thin. It provides an interactive user interface. In other words, the McXML server handles business logic, while the McXML client handles application logic.

7.2.1. McXML Server

The McXML Server is a thick server. It provides all the functionalities for data queries, updates, indexing, storage, etc. Its features include:

- It supports all the update operations that XQuery Extension XML update language specifies (See McXML.stmt in Section 7.2.1.2).
- It recognizes IDREF expressions (See McXML.expr.xpath in Section 7.2.1.2).
- It is capable of executing complex updates containing sub updates (See McXML.stmt in Section 7.2.1.2).
- It has fully implemented functionalities for predicate evaluation (See McXML.expr.condition in Section 7.2.1.2).
- It uses DOM Pooling to make XML data loading more efficient (See Section McXML.server.database.driver & McXML.Util in Section 7.2.1.2).
- It developed its own indexing structures to optimize its performance (See Chapter 8).

7.2.1.1. Package Diagram

Figure 7-2 shows the package diagram of the McXML server that gives a overview of how it is organized. Briefly, the McXML Server accepts commands (which are XQuery command or XQuery Extension command strings) from the McXML clients. Upon receiving a command, it calls the drivers to execute the command. There are two drivers: the McXML.server.database.driver.KAWADriver for XML data queries and the McXML.server.database.driver.XMLDBDriver for data storage, data updates and data indexing. The drivers invoke worker functionalities provided by the McXML.Util package to check command syntax, parse commands semantically, bind variables, and manage XML data accordingly.

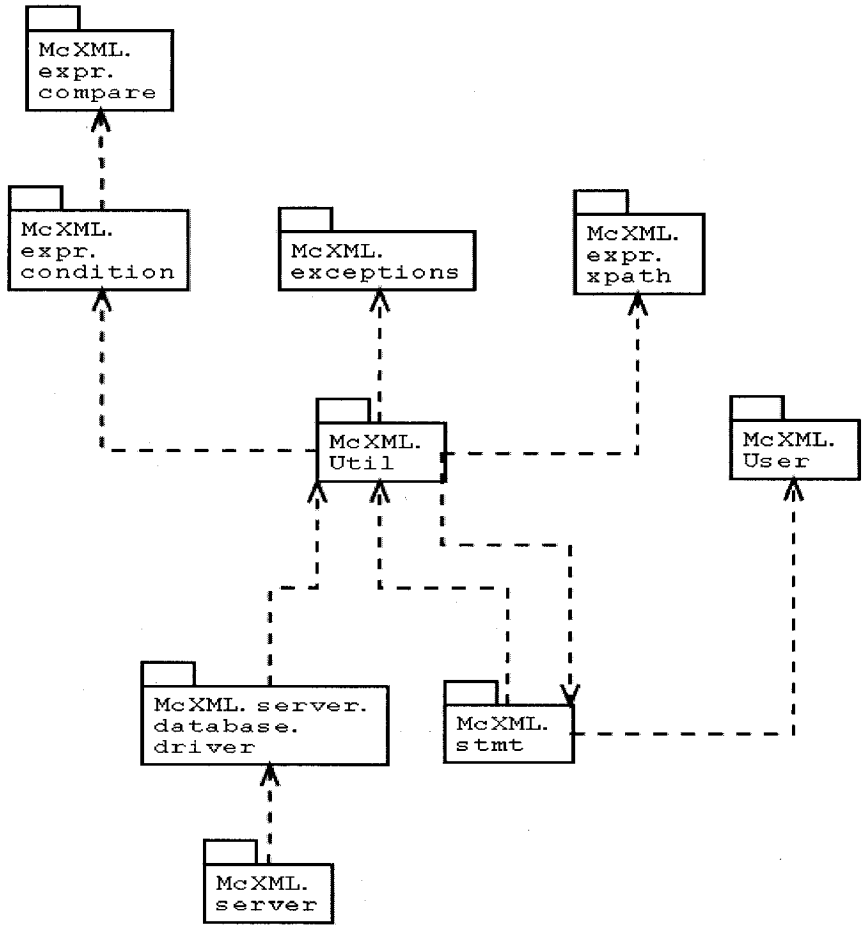


Figure 7-2: Package Diagram of the McXML Server

The McXML.stmt package contains several kinds of McXML.stmt.XMLUStmt objects, which create, delete, insert before, insert after, insert into, delete from, rename, or replace XML data.

The McXML.expr.condition package contains serveral kinds of McXML.expr.condition.CondExpr objects, which evaluate different predicate conditions: the predicates enclosed by '[' and ']' and tail the XPath expression in the update command and those introduced by "where" in the where-clause of the update command.

The McXML.expr.xpath package contains several kinds of McXML.expr.xpath.XpathExpr objects, which interpret various kinds of XPath expressions including the IDREF expression. It is useful for variable binding.

And the McXML.exceptions package provides functionalities for both syntax and semantic error handling.

The McXML.Util package is dependant on these packages.

The McXML.User package has a McXML.User.CurrentUser object, which manages current user information, such as the user account location, the user name, the user password, the user quota and so on. The McXML.stmt package depends on it to fetch the concerned XML data from the corresponding user account for processing.

The McXML.expr.compare package contains some comparator objects, which act as '<', '>', '<=', '>=', '= =', and '! ='. And the McXML.expr.condition package depends on it, for the McXML.expr.condition.CondExpr Objects use these comparators to evaluate predicate conditions.

7.2.1.2. McXML Server Packages

As described in the previous section, there are several packages used in the McXML server organization. Some of the packages are worth more detailed description:

- **McXML.server.database.driver**

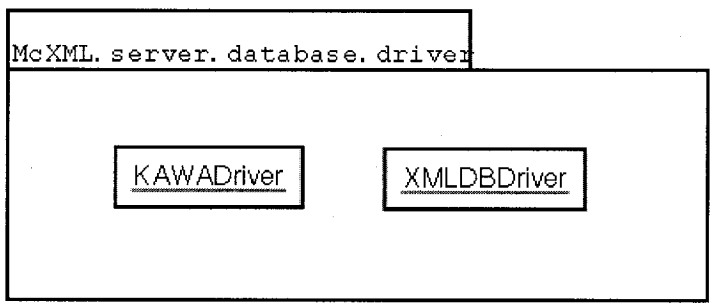


Figure 7-3: McXML.server.database.driver package

In this package, there are two classes: the KAWADriver and the XMLDBDriver.
The KAWADriver uses the XQuery engine of the KAWA compiler [PBRM03] to perform XML data queries. The KAWA compiler is both a framework for implementing, compiling, and running programming languages in Java, and also includes implementations of Scheme, XQuery, Emacs Lisp, etc. KAWA is free software with a "modified Gnu Public License". It was integreted into McXML for XML data queries. Some tiny modifications were made on KAWA to make it more efficient. For instance, the KAWA compiler loads XML document from disk to memory every time an XQuery command is issued and discards the memory copy directly after the execution of the command. If several commands on the same XML document are submitted, this is

certainly not efficient. Hence KAWA was modified to use McXML.Util.DOMPooling to avoid repetitive loading of the same XML documents, which will be described in detail in the latter part of this section.

We developed the XMLDBDriver on our own. It is responsible for XML data storage, XML data updates, XML data indexing, etc. The XMLDBDriver works independently of the KAWADriver. It interpretes the XPath expression, evaluates predicate conditions and handles the variable binding of the XML update commands on its own. If we had extended the XMLDBDriver to be able to return the result data of an XQuery, we could have used it for XML data queries instead of KAWADriver. However, XML data queries are not our main concern, and for now we chose to focus our efforts on XML data updates, so the aforesaid functionality has not yet implemented. The XMLDBDriver does its job by using functionalities provided by the McXML.Util package.

- **McXML.Util**

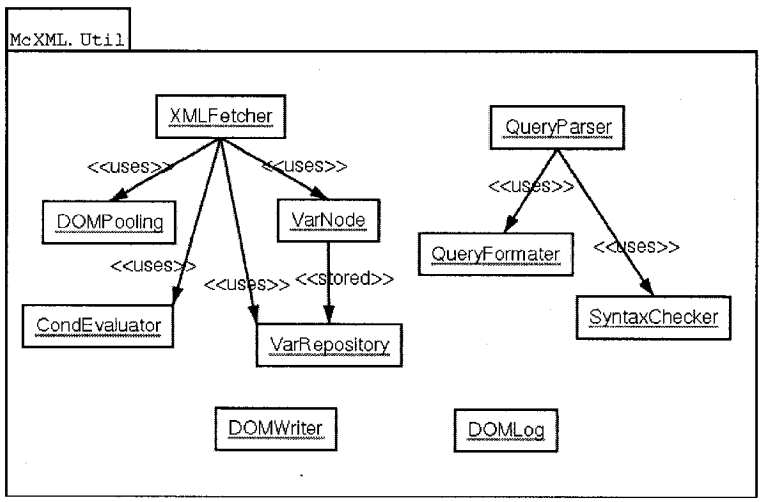


Figure 7-4: McXML.Util Package

This is a very import package that provides very handy functionalities for XML data processing.

The XMLFetcher object fetches XML nodes according to the XPath expression in the XQuery Extension command string. It uses the DOMPooling to avoid loading the same XML document repetitively. Basically, the XML DOM trees of the XML documents are stored in the DOMPooling the first time they are loaded. When an XML DOM tree is re-

asked for updates, instead of being reloaded from the disk, it is fetched from the DOMPooling. There is a limit on the number of the DOM trees active in the pooling, and the first-in-first-out rule is applied. The XMLFetcher uses the CondEvaluator to filter the list of XML nodes it fetched. Only those that satisfy the predicate conditions in the XQuery Extension command string and evaluated true by the CondEvaluator are returned. It is possible that the same XML node may be updated more than once in a command. Hence, in order to optimize the performance, every XML node fetched by the XMLFetcher is bound to a variable represented by a VarNode, and stored in the VarRepositioy for reuse purpose.

The QueryParser object parses the XQuery Extension command string into a McXML.stmt.XMLUStmt for execution. Firstly, it uses the QueryFormater to delete all the unnecessary white spaces, new line characters, etc. from the command string to make the command neater and easier for further parsing. Then, it uses the SyntaxChecker to check whether the command string is syntactically correct according to the syntax rules of the XQuery Extension language. Lastly, it parses the command string into for-clause, let-clause, where-clause, update-clause, etc., and uses the clauses to compose a McXML.stmt.XMLUStmt object, which uses the aforementioned XMLFetcher to solve variable bindings.

The DOMWriter writes DOM trees into strings and saves them as XML documents back to disk when necessary.

The DOMLog is used to log the index of updated XML nodes. This optimizes commit and rollback performance, which we will explain in more details in Section 8.1. of Chapter 8.

- **McXML.expr.condition**

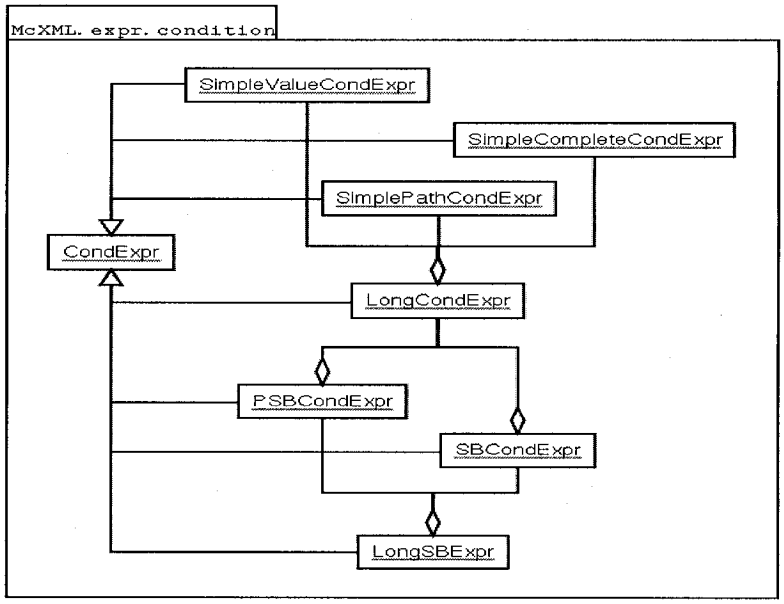


Figure 7-5: McXML.expr.condition Package

This package contains some objects that represent different kinds of predicates. According to XPath syntax grammar, where-, for-, let-clauses all may contain predicates. The predicates in for- and let-clauses are enclosed in '[' and ']'. In this package, the predicate expression objects are defined according to the following grammar:

SimpleValueCondExpr := '=' + Value;
SimplePathCondExpr := XpathExpr;
SimpleCompleteExpr := XpathExpr + '=' + Value;
LongValueExpr := ((SimpleCompleteExpr (and | or)?)?
(SimplePathExpr (and|or)?)?
*(SimplePathExpr (and|or) ?)?) **
SBCondExpr := '[' + LongValueExpr + "]";
PSBCondExpr := XPath + SBCondExpr;
LongSBCondExpr := ((SBCondExpr (and|or) ?)?
*(PSBCondExpr (and | or)?)?)**

The CondEvaluator object in McXML.util evaluates these predicate expressions on the corresponding XML nodes.

- **McXML.expr.xpath**

This package contain several XPathExpr objects, which represents different kinds of XPath expressions. Among them, the IDRefXPathExpr interprets IDREF, which recognizes IDREF attributes and treats them accordingly. The AttrXPathExpr interprets a string of format "*@attrname*" into an attribute XPath expression. The DocXPathExpr

represents XPath expressions that begin with "*document("xmlfilename")*". The ContentXPathExpr intepretes a valid XML content fragment string into a regular XPath expression. The OrdXPathExpr represents the regular XPath expressions, which deals with the wildcard '//' by searching all the sub paths of the path preceding "//" and returning all those whose last tag name is the same as the one that follows "//".

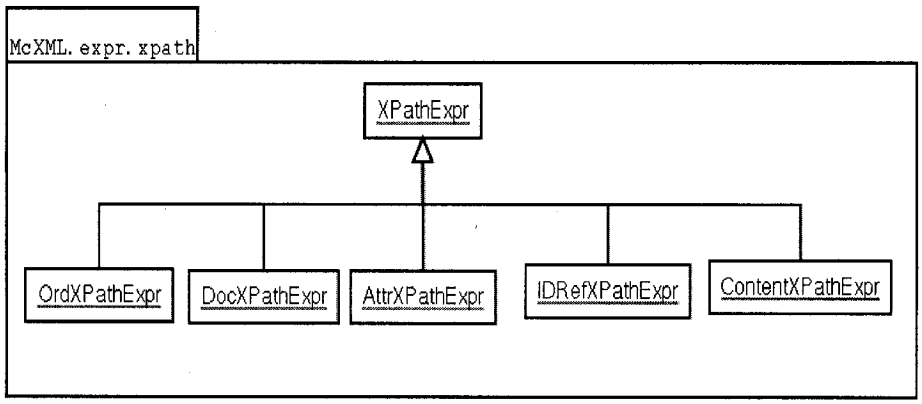


Figure 7-6: McXML.expr.xpath Package

- **McXML.stmt**

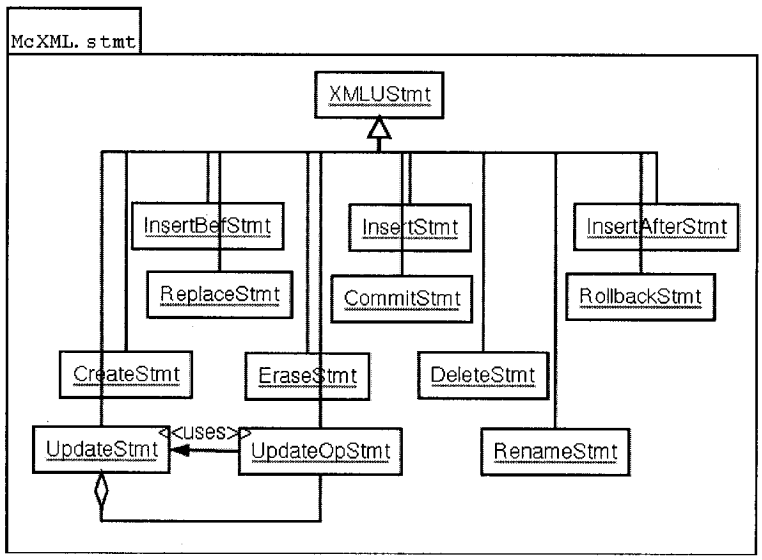


Figure 7-7: McXML.stmt package

This package contains some objects that do what their name suggests. Among them, the CommitStmt commits all the uncommitted updates, and RollbackStmt aborts all the uncommitted updates. With them, McXML is fully extendable to support transactions. For instance, in the future, we may implement BeginTransaction and EndTransaction

statements which enclose in a transaction all the operations in between. We may optimize the CommitStmt (RollbackStmt respectively). When provided with a transaction as parameter, it only commits (aborts respectively) those operations that belong to the transaction. We may also set an auto commit flag. When it is set true, any operation that does not belong to a transaction should be committed immediately after the execution. Otherwise, the operation has to be explicitly committed.

The CreateStmt creates an XML document from scratch. The DeleteStmt deletes an XML document from the user account.

Each of the following statements has one or more XML nodes as input parameters. The EraseStmt removes some XML nodes from its parent nodes. The InsertBefStmt (InsertAfterStmt respectively) inserts a sibling before (after respectively) the indicated XML node. The InsertStmt appends a child node to the child node list of the indicated XML node. The RenameStmt renames the indicated XML node. The ReplaceStmt replaces some content in the indicated XML node with something different.

The relationship (composition & association) between the UpdateStmt and the UpdateOpStmt is to support sub updates, which is illustrated by the following example: We have the following XQuery Extension command string that contains a sub update:

```
For $b in document("bib.xml")//book,
    $t in $b/title
update $b
{
        delete $t,
        for $a in $b/author
        update $a
        {
                rename $a to "writer"
        }
}
```

The command is parsed by the McXML.Util.QueryParser into an UpdateStmt as shown in Figure 7-8:

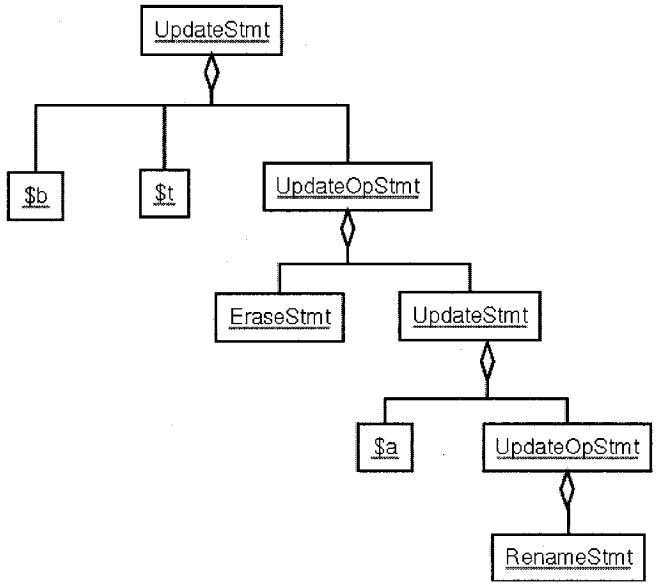


Figure 7-8: A Sub Update

An UpdateStmt has a list of variables and an UpdateOpStmt. When being executed, it first solves bindings for its variables, and then it executes the UpdateOpStmt to update the XML data bound to these variables.

An UpdateOpStmt has a sequence of XMLUStmts (XMLUStmt is the supper class in the McXML.stmt package, see Figure 7-7 for the inheritance relationship). When being executed, it executes these XMLUStmts one by one.

So, the execution of the above UpdateStmt are carried out smoothly as follows:

I. *$b* and *$t* are bound to some XML nodes by the McXML.Util.XMLFetcher.

II. The XML nodes bound to *$t* are iterated and removed from the corresponding parent XML node bound to *$b* by the EraseStmt.

III. *$a* is bound to some XML nodes by the McXML.Util.XMLFetcher.

IV. The XML data nodes bound to *$a* are iterated and renamed by the RenameStmt.

7.2.1.3. A Collaboration Diagram

We have got an idea how McXML server is organized. Figure 7-9 shows a simplified collaboration diagram of the McXML server.

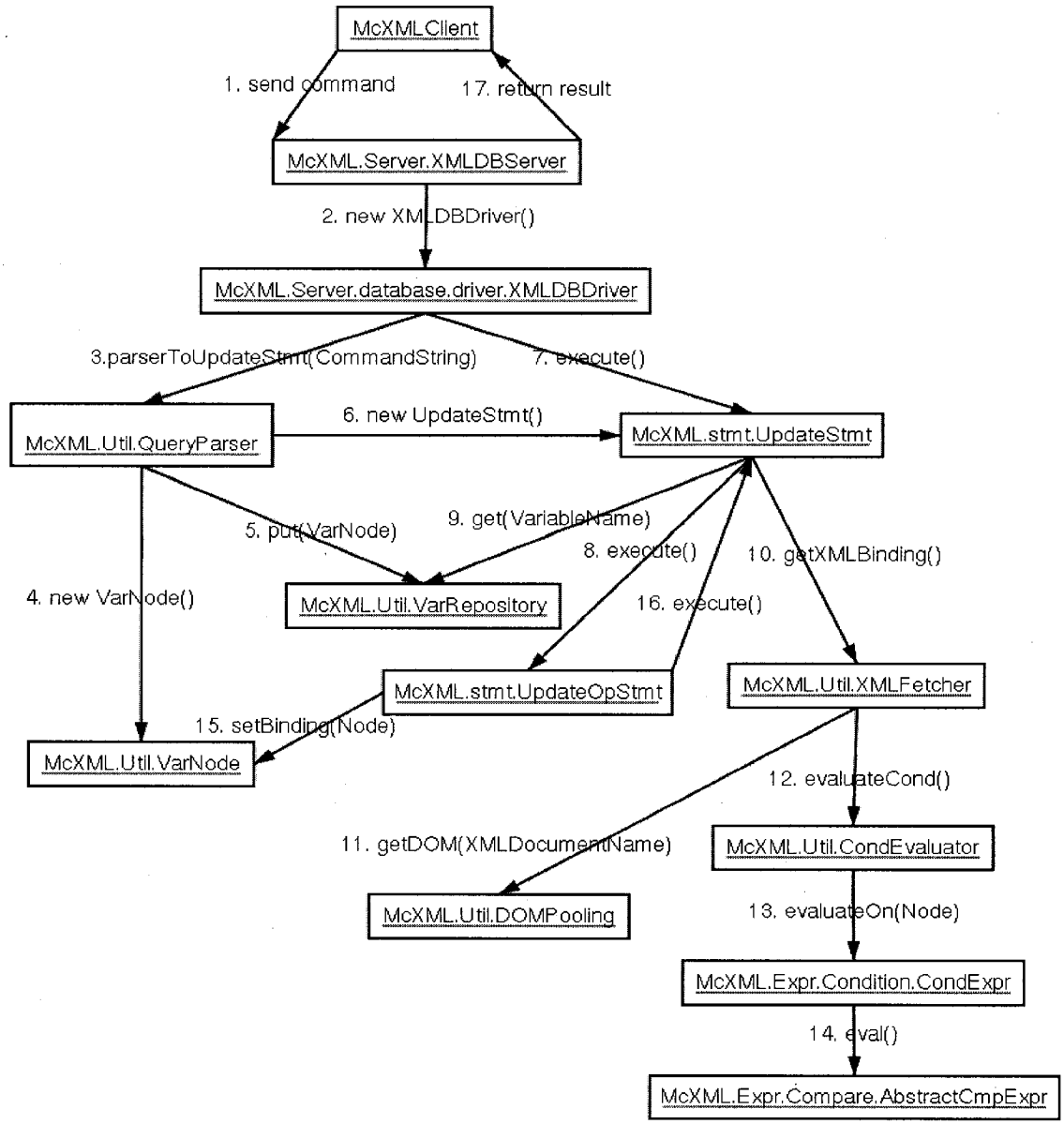


Figure 7-9: A Collaboration Diagram of the McXML Server

It shows a scenario how the McXML components collaborate with one another to execute an XQuery Extension command.

- The McXML server receives an XQuery Extension command from the McXML client (Step 1).
- It then creates an instance of McXML.server.database.driver.XMLDBDriver to execute the command (Step 2).
- In executing the command, the driver calls McXML.Util.QueryParser to parse the command string into a McXML.stmt.UpdateStmt (Step 3-6).
    - In the process of parsing, the QueryParser creates a McXML.Util.VarNode for each variable in the command string, and stores them in the static McXML.Util.VarRepository. At this point, the variables are not yet bound.
    - The UpdateStmt maintains the list of the VarNodes as well as a McXML.stmt.UpdateOpStmt.
- The driver then executes the newly created UpdateStmt, which
    - first solves bindings for its variables represented by the VarNodes with the help of McXML.Util.XMLFetcher and McXML.Util.DOMPooling (Step 9-15),
    - and then executes its UpdateOpStmt, which has a sequence of McXML.Util.XMLUStmts, and executes them one by one to update the bound XML data (Step 16).
- The result of the execution is returned by the McXML server to the McXML client (Step 17).

7.2.2. McXML Client

The McXML client is a thin client. It contains the iterative user interfaces and handles application logic. It is very user friendly. Figure 7-10 shows its component diagram:

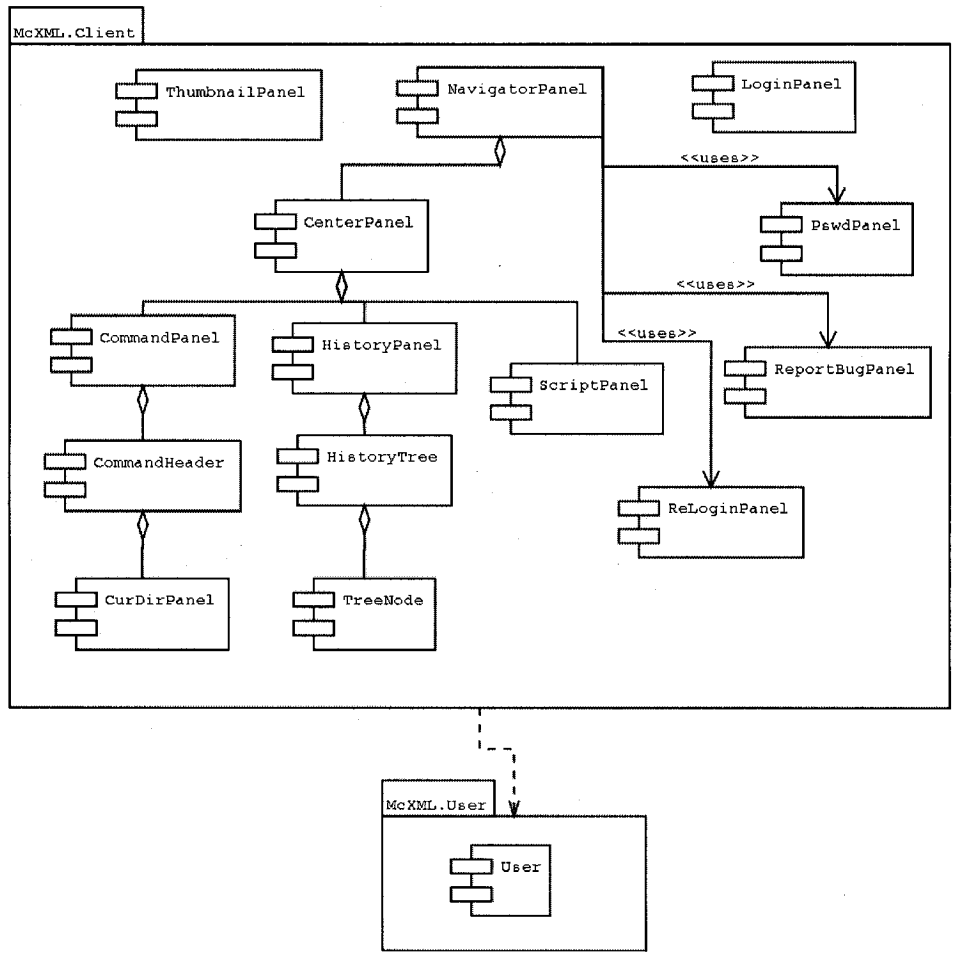


Figure 7-10: Client Component Diagram

- ThumbnailPanel

  The ThumbnailPanel shows up when the McXML application starts. It serves as a thumbnail icon of the application.

Figure 7-11: ThumbnailPanel

- LoginPanel

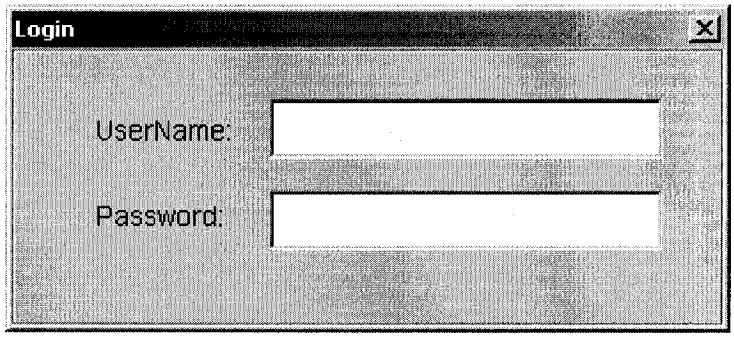


Figure 7-12: LoginPanel

The LoginPanel opens as soon as ThumbnailPanel closes. It prompts user for username and password.

- NavigatorPanel

  The NavigatorPanel is the main component of the McXML client (See Figure 7-13). It consists of the toolbar and the CenterPanel. On the toolbar of the NavigatorPanel, there are some menus, *File*, *Edit*, *Workspace*, *Admin*, *Window*, and *Help*. Among them, *Workspace* menu allows the user to specify the working directory, *Admin* menu allows the user to change password or login as another user, *Window* menu allows the user to hide and show the HistoryPanel, CommandPanel and ScriptPanel in the work place, and *Help* menu allows the user to go to the documentation websites for McXML or report bugs through email.

- CenterPanel

  The CenterPanel is the work place for the user. As shown in Figure 7-13, the CenterPanel is located below the toolbar in the NavigatorPanel, and it contains three panels, the HistoryPanel (the one on the left), the CommandPanel (the one in the middle) and the ScriptPanel (the one on the right).

- CommandPanel

  The CommandPanel is designed for the user to type commands and run commands one by one. It is very user friendly. The user may type a command after the prompt sign >>. Text after the active prompt sign is editable, while that before it is not. The user may also user Up and Down keystrokes on the keyboard to retrieve the historical commands. When the user types a character ';' followed by a enter keystroke, the

command gets executed, and the execution result will be shown below the command (See Figure 7-13).

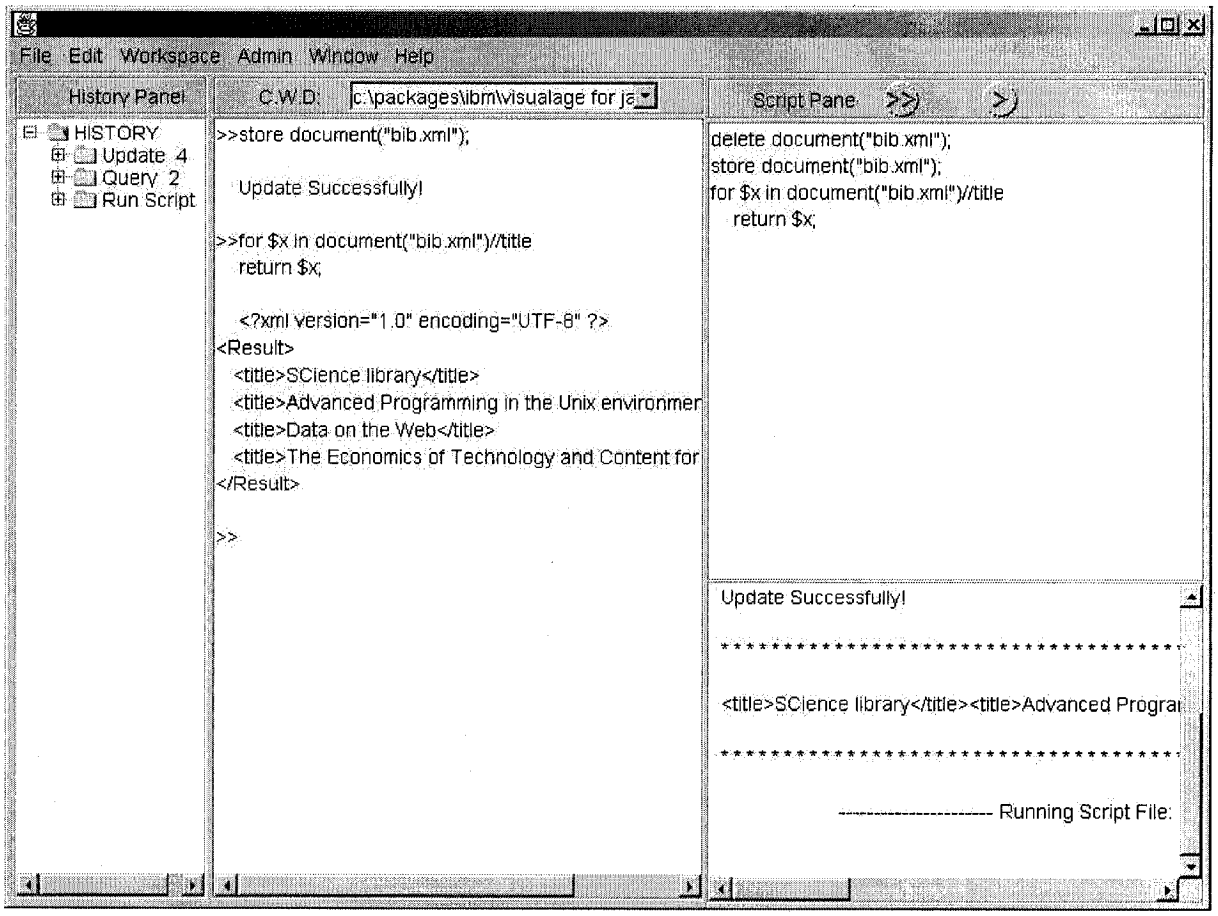


Figure 7-13: NavigatorPanel

- ScriptPanel

  The ScriptPanel is designed for the user to run commands in bunch. The user may create or import a script file containing a list of commands and run it through the ScriptPanel. There are two round button at the top of the ScriptPanel, the *Fast* (with two arrows on it) and *Step* (with one arrow on it) buttons. A click on the former executes all the commands in the script file in one run, while clicking the latter executes one command at a step. The ScriptPanel is also user friendly. It recognizes and skips comment lines that begin with "//".

- HistoryPanel

The HistoryPanel has a HistoryTree in it, which is a tree of recently executed commands and their execution results (See Figure 7-13). The HistoryPanel is synchronized with the CommandPanel and the ScriptPanel. Once a command is executed through the CommandPanel or a script is run through the ScriptPanel, the command string together with the command execution result is inserted as a tree node to the HistoryTree. The user may save the whole tree or some of the tree nodes into XML files. The user may also clear the whole tree or delete some of the tree nodes from the tree.

- CommandHeader

  The CommandHeader is located at the top of the CommandPanel. It has a dropdown menu item that shows and allows the user to specify the current working directory.

- CurDirPanel

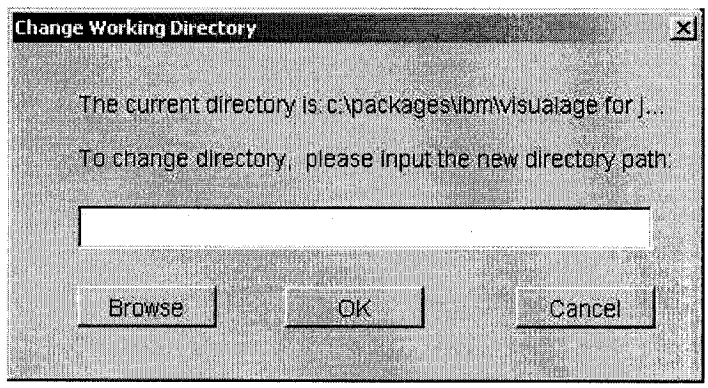


Figure 7-14: CurDirPanel

The CurDirPanel opens when the user clicks the *Current Working Directory* menu item on the *Workspace* menu on the toolbar of the NavigatorPanel. It allows the user to change the current working directory.

- PswdPanel

  When the user clicks the *Change Password* menu item on the *Admin* menu, the PswdPanel opens. It prompt user for information to change the password.

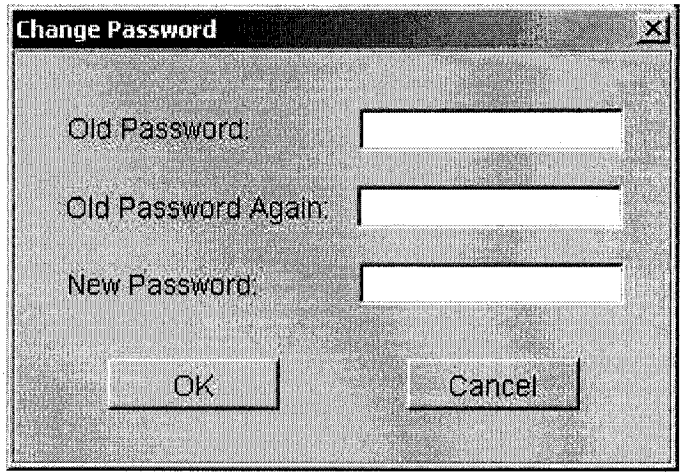


Figure 7-15: PswdPanel

- ReloginPanel

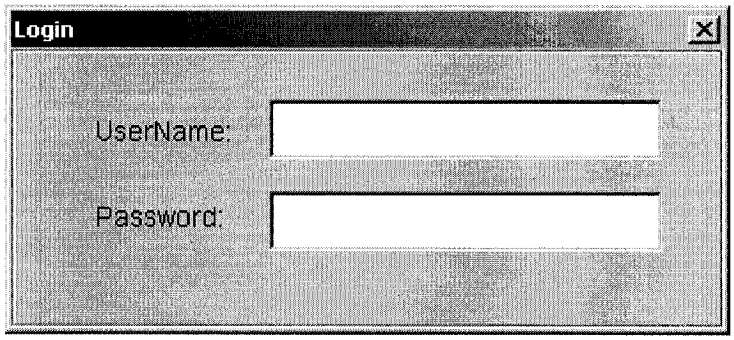


Figure 7-16: ReLoginPanel

McXML supports multi-users. When the user clicks the *Login As Another User* menu item on the *Admin* menu, the ReloginPanel opens. It prompt user for information to log in as anther user.

- ReportBugPanel

When the user clicks Report Bug menu item on the Help menu, the ReportBugPanel shows up. It prompt user for information to report the bug by sending email to the administrator of McXML.

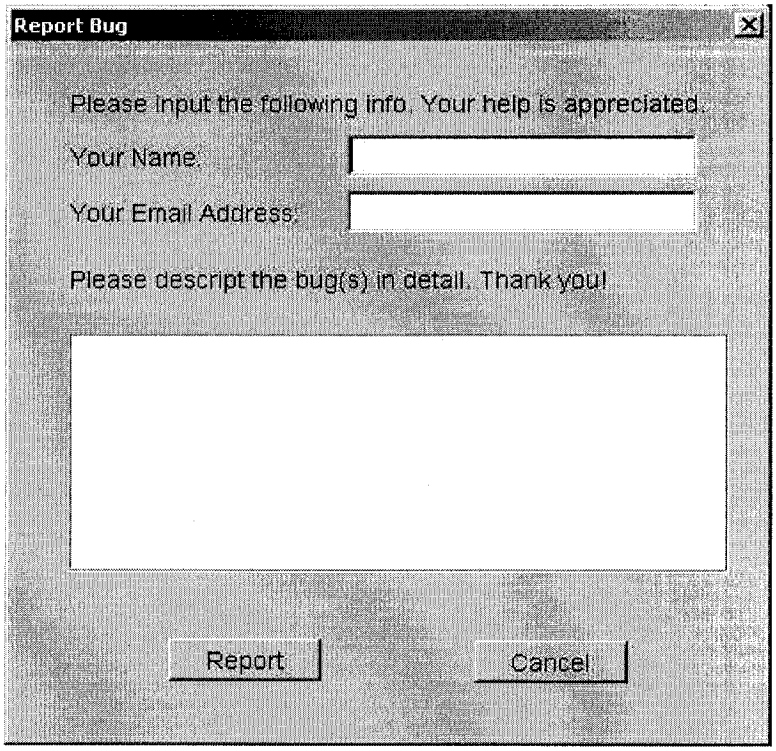


Figure 7-17: ReportBugPanel

Chapter 8

Indexing Structures of McXML

McXML has two indexing structures in order to offer fast storage and retrieval of XML data. The first one is the Commit Indexing Structure (CIndex), which is designed specially to optimize commit performance of McXML. The second one is the Query/Update Indexing Structure (QIndex), which is designed to optimize XML data queries in McXML. In this chapter, we describe the purpose, design idea, and the performances of these two indexing structures.

8.1. Commit Indexing Structure (CIndex)

McXML has developed its own unique Commit Indexing Structure (CIndex) to make the commit operation efficient. When commit operation is invoked, updated data needs to be saved by being written back to disk. Without CIndex, the entire XML document in question has to be rewritten to disk. This includes serializing the entire XML DOM tree. As a result, the commit time is proportional to the size of the document. This is neither efficient nor necessary since normally only a small part of the document data is affected by updates. It makes sense to optimize the commit time to be proportional to the size of the affected data. CIndex has been developed for this purpose. With CIndex, instead of the whole data, only the affected data is rewritten to disk.

CIndex contains a list of document IDs followed by the hierarchical address paths of the affected nodes in the XML document identified by the document IDs. The address path is itself a sequence of tag names and ordinal numbers.

8.1.1. A CIndex Example

Figure 8-1 shows an example:

The italic underlined numbers after each node indicate its ordinal in the siblings with the same name. Say the XML document has the identifier *bib.xml*, and in it all the bolded nodes (in Line 8, Line 22, and Line 30) are updated, the CIndex would have an entry like this:

bib.xml
bib , 1 , book , 1 , author , 1 , last 1
bib , 1 , book , 2 , author , 2 , first 1
bib , 1 , book , 2 , price , 1

```
1.  <?xml version="1.0" encoding="UTF-8" ?>
2.  <!—This is an example XML document -->
3.  <!DOCTYPE bib SYSTEM 'bib.dtd'>
4.  <bib>                                          1
5.    <book year="1994">                           1
6.      <title>SCience library</title>             1
7.      <author>                                   1
8.        <last>Stevens</last>                     1
9.        <first>W.</first>                        1
10.     </author>
11.     <publisher>Addison-Wesley</publisher>      1
12.     <price>65.95</price>                       1
13.   </book>
14.   <book year="2000">                           2
15.     <title>Data on the Web</title>             1
16.     <author>                                   1
17.       <last>Abiteboul</last>                   1
18.       <first>Serge</first>                     1
19.     </author>
20.     <author>                                   2
21.       <last>Buneman</last>                     1
22.       <first>Peter</first>                     1
23.     </author>
24.     <editor>                                   1
25.       <last>Gerbarg</last>                     1
26.       <first>Darcy</first>                     1
27.       <affiliation>CITI</affiliation>          1
28.     </editor>
29.     <publisher>MIT</publisher>                 1
30.     <price>129.95</price>                      1
31.   </book>
32. </bib>
```

Figure 8-1: An CIndex Example in bib.xml

The first line is the document identifier. The other lines are the hierarchy address paths of the updated nodes. In this case, the CIndex entry shows that there are three nodes updated in the XML document identified by *bib.xml*. For an instance, the third line of the entry

indicates that the 1st of the nodes with tag name *first* in the 2nd of the nodes with tag name *author* in the 2nd of the nodes with tag name *book* in the 1st of the nodes with tag name *bib* in the document identified by *bib.xml* is updated and needs to be rewritten to disk when commits.

8.1.2. McXML.util.DOMLog Object

In McXML, a McXML.util.DOMLog object is implemented to represent CIndex. Each hierarchy address path of the updated nodes in an CIndex entry corresponds to a log record in an instance of McXML.util.DOMLog object. Besides the hierarchy address path, a log record also stores a pointer to the updated node. Any XML document in memory that has unsaved updates must have an entry in the CIndex, thus, must have some log records in the McXML.util.DOMLog object.

When the commit operation is executed, the disk copy of such an XML documents is scanned, the locations of the affected nodes in the disk copy are calculated according to their hierarchy address paths stored in the log records. The memory copies (which are XML DOM Node objects, see Section 1.5.) of these nodes are fetched through the pointers stored in the log records. With the help of McXML.util.DOMWriter, these memory copies are patched to overwrite the corresponding parts of the disk copy of the XML document.

In the McXML.util.DOMLog, the log records of the affected nodes are sorted according to the order these nodes are encountered in depth-first traversal of the XML DOM tree. This order is the same as the order in which these nodes appear in the textual disk copy of the XML document, so committing all the updates in an XML document needs only one efficient forward disk write.

8.1.3. Compatibility to Updates

Update operations might change the ordinals of right siblings of the updated nodes. For instance, the delete of an *<author>* element decreases the ordinals of the right *<author>* siblings of the deleted one. Insert, rename and replace have similar affects. In XQuery

Extension update language that McXML implements, however, all update operations including insert, delete, rename, replace, etc., are done through the parent node. For example, in the following XQuery Extension update command, the real target node is *$a*, but the update is done on its parent node *$b*. Therefore *$b* is considered as the updated node, and if necessary, a log record is created for *$b* instead of *$a*.

```
for $b in document("bib.xml")//book
let $a = $b/author
where $a/first = 'Serge'
update $b
{
        delete $a;
}
```

This makes it irrelevant that the update operations affect the ordinals of siblings of the target node. The update on a target node is embodied by its parent node, for the memory copy of the child node is a sub set of that of the parent node. Therefore, in case that an update operation on a parent node exists or happens, it is not necessary any more to keep track of the update operations on the child nodes. In CIndex of the McXML, if a child node is updated before its parent nodes is, upon the creation of the log record of the parent node, the existing log record of the child node is deleted. Otherwise, no log record is created for the child node. More generally, an update operation that is done on a bigger XML sub-tree overwrites (embodies, respectively) all the previous (subsequent, respectively) update operations that are done on the smaller XML branches. In CIndex of the McXML, only the updates on the bigger XML sub-trees have update log records.

In the above example, if before the delete operation, we already has a log record with the pair of value {*bib, 1, book 2, author, 2* ; a pointer to the <*author*> element }, upon the delete operation, the log record becomes invalid, because the ordinal of the <*author*> element is decreased from 2 to 1, and the last number 2 in the hierarchy address path is wrong. It does not matter, for the log record should be deleted anyway. If necessary, a log record is created for the delete operation, which stores the pair of value {*bib, 1, book, 2* ; a pointer to *$b*}. Later if there are updates on any of the descendant nodes of *$b*, no log record would be created.

8.1.4. Advantages of CIndex

As explained in Section 8.1.2, CIndex is sorted in such a way that committing all the updates in an XML document needs only one forward disk write. It is efficient. As explained in Section 8.1.3, in CIndex, invalidated log records are deleted immediately. Furthermore, only necessary log records are created and stored. CIndex contains no redundant information. It is lean, and small in size.

The other advantages of CIndex are obvious too:

- CIndex is set up on the fly in the process of variable binding. It almost takes no time.
- CIndex is dynamic. Once the commit operation is done, it is simply deleted. It needs no maintenance.
- CIndex resides in main memory only. No extra disk storage required.
- CIndex is easy to implement.

8.1.5. Performance

CIndex greatly improves the performance of the commit operation in McXML. For an instance, to commit updates on a 2.4M XML document, without CIndex, it takes about 120 seconds. With CIndex, the commit only takes about 4.5 seconds.

Chart 8-1 shows the comparison between the commit performances of McXML before and after CIndex is installed. The x-axis shows the size of the XML data. The y-axis shows the commit time in seconds. The thin line curve shows the commit performance when CIndex is not used. The thick line curve shows the commit performance when CIndex is used. It is very clear that without CIndex the commit time is linear to the data size. With CIndex, the commit performance is much better.

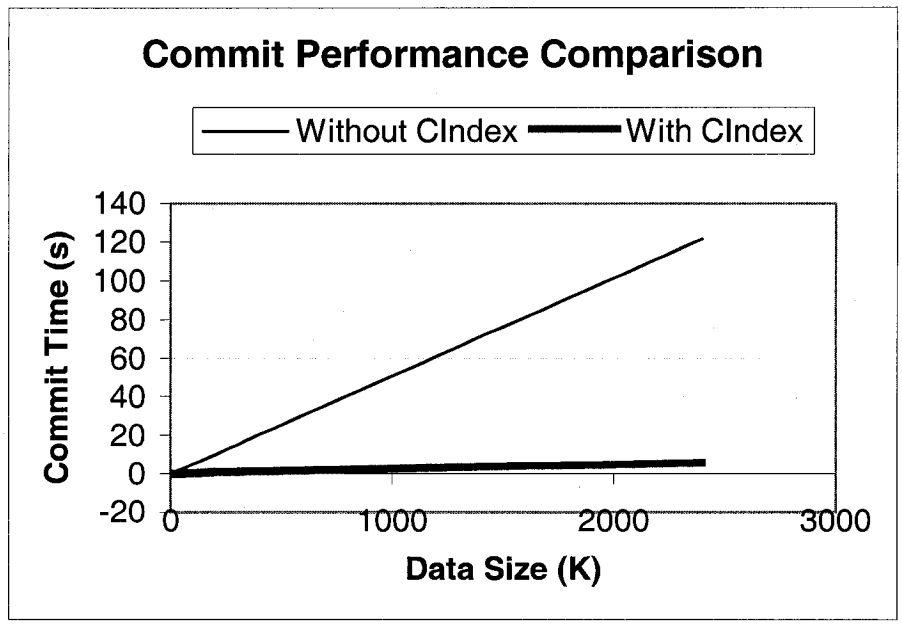


Chart 8-1: Commit Performance Comparisons

Chart 8-2 shows the commit performance of McXML with CIndex installed. The tested XML data is 2.4M in size. The x-axis shows the amount of the updated XML data. The y-axis shows the commit time in seconds. It is clear that with CIndex, the commit time is proportional to the updated data size. The intercept of the curve on y-axis is not zero, which suggests that there is some overhead involved. This is inevitable, for some time is spent on scanning the entire XML data. In this case, it takes about 4 seconds to scan the entire 2.4M XML data.

Chart 8-3 shows the comparison between the commit performance of McXML before and after CIndex is installed. The tested XML data is 2.4M in size. The x-axis shows the amount of the updated XML data. The y-axis shows the commit time in seconds. The thin line curve shows the commit performance without using CIndex. The thick line curve shows the commit performance when CIndex is used. It is very obvious that with CIndex the commit performance is much better.

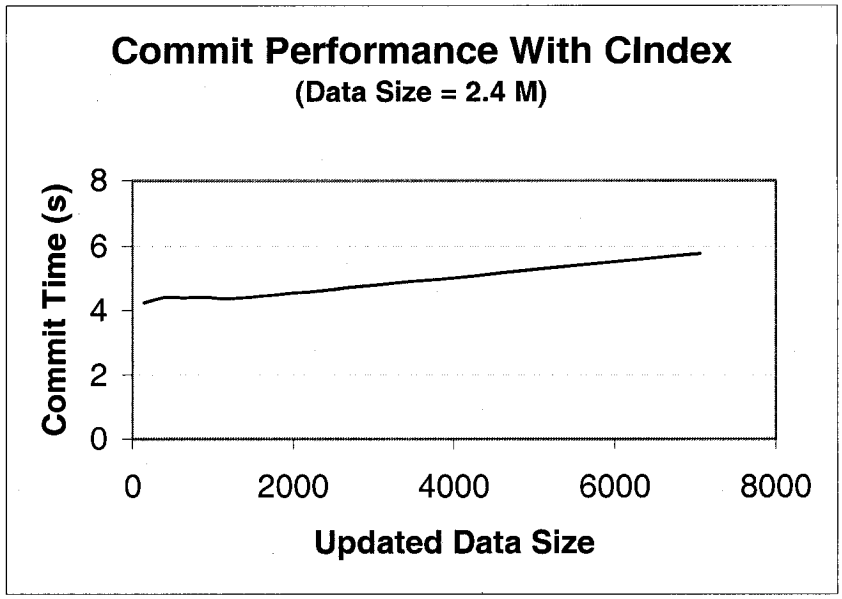


Chart 8-2: Commit Performance With CIndex

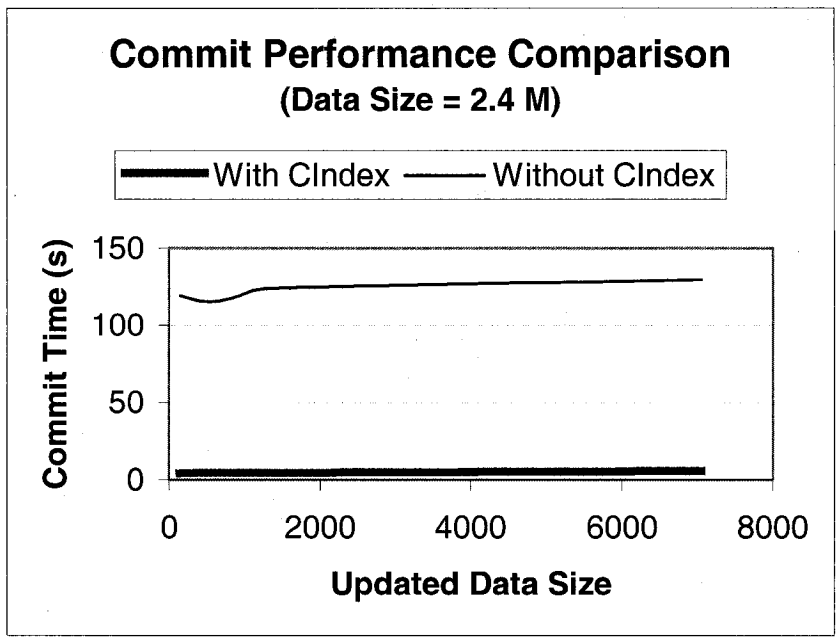


Chart 8-3: Commit Performance Comparison (With CIndex vs. Without CIndex)

8.2. Query Indexing Structure (QIndex)

The Query Indexing Structure (QIndex) in McXML is designed to optimize XML data queries. It is a new feature that was added to McXML version 1.0.1 in May 2003.

8.2.1. Difficulties

It is not trivial at all to design an indexing structure that optimizes XML data queries in McXML, for there are some extra difficulties.

Unlike many existing XML databases, McXML supports XML data updates as well as XML data queries, which makes it more difficult to build an indexing structure. We have to guarantee that the indexing structure not only improves the performance of XML data queries, but also improves, or at least not degrades, the performance of XML updates. This is difficult, since the indexing structure might be affected by data updates. It requires some index maintenance in order that the index to be up-to-date. This imposes extra burden to XML data updates and might slow down the update performance to some extent.

Though McXML uses the file system for physical data storage, it uses XML DOM tree as the logical data model. In other words, disk copies of XML data are all flat *.xml* files, while the memory copies of the XML data are XML DOM trees. This design is both simple and robust. However it leads to another requirement: the indexing structure must be built on XML DOM trees instead of *.xml* files, for all the XML data processing is done in memory on the XML DOM trees. This requirement narrows our options, for it technically excludes using signature files or inverted files for implementation.

Also we want the design of the indexing structure to be simple, efficient, and most importantly, original. Despite the difficulties we came up with QIndex, which in my opinion is a great success.

8.2.2. Position Path Expression (PPE)

QIndex works very well in improving XML data query performance. One of the reasons is that it finds a better way of addressing desired XML nodes.

The traditional way of addressing an XML element is to traverse the DOM tree in depth-first order. To find the desired element, the traversal usually covers a big portion of the DOM tree. For example in Figure 8-2, to locate element 18 in Figure 8-2, the traversal covers elements 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 and 18.

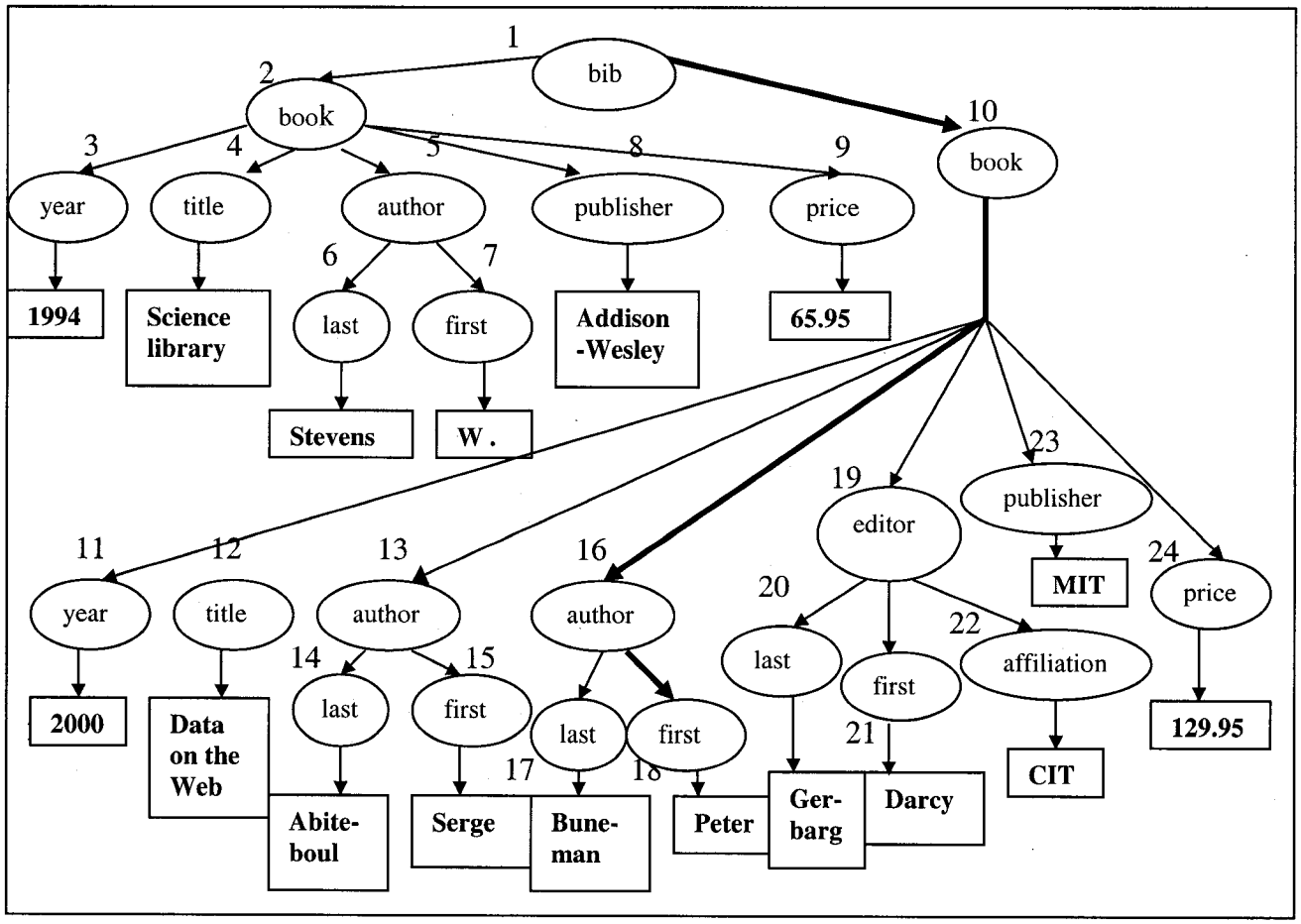


Figure 8-2: A Simplified XML DOM tree on bib.xml

This is not efficient. Instead, QIndex uses a different way of addressing XML elements in XML DOM tree that requires traversing only a very small portion of the XML DOM tree. In a DOM tree, the path from the root node to an XML node is unique. For example, in Figure 8-2, the path in bold is the path from the root node 1 to node 18. In QIndex, a

special path expression that we defined as Position Path Expression (PPE) is used to identify this path. The PPE is a sequence of '/' delimited numbers. Each number indicates the ordinal of the node among its siblings. For example, the PPE of node 7 is *1/1/3/2*, that of node 15 is *1/2/3/2*, and that of node 18 is *1/2/4/2*.

With PPE, to address element 18, we need only visit four nodes: node 1, 10, 16, and 18, invoking *getChildElementAt (int pos)* operations, which does what its name suggests, on the ancestor nodes all the way down.

PPE is perfect to serve as index on XML DOM tree. However, it is not a good idea to store PPEs directly in the indexing structure. Firstly, it is difficult to lower the cost of the maintenance required to keep such an indexing structure up-to-date, for it is susceptible to XML data updates. An insertion or deletion causes all the right siblings of the inserted or deleted node **and** all their descendant nodes to change their PPEs, as a result, a big portion of the indexing structure needs to be fixed. Secondly, such an indexing structure is redundant. A PPE might appear in the indexing structure more than once. Once as the PPE of an XML node, **and** several times as prefix of the PPEs of the descendant nodes of that XML node supposing the XML node is not a leaf.

8.2.3. Relative Position Path Expression (RPPE)

In QIndex, we store Relative Position Path Expressions (RPPE) instead. We then use the RPPEs to construct the PPEs indices. The RPPE contains three numbers: the first one is the ordinal number at the last level in the corresponding PPE, which indicates the *ordinal* of the node among its siblings; the second and third numbers are a *Cormen's number pair*. It is easy to construct the PPEs from RPPEs.

8.2.3.1. Cormen's number scheme

Early in 1982, Dietz [PFD82] proposed a numbering scheme to use tree traversal order to determine the ancestor-descendant relationship between any pair of tree nodes:

> *For two given nodes x and y of a tree T, x is an ancestor of y if and only if x occurs before y in the preorder traversal of T and after y in the postorder traversal.*

In 1989, Thomas H. Cormen et.al. [CLRS89] proposes a similar numbering scheme:

> *For two given nodes x and y of a tree T, x is an ancestor of y if and only if x is visited before y is visited and x is exited after y is exited in the depth-first traversal of T.*

Compared with the Dietz's numbering scheme, which requires two traversals, the Cormen's numbering scheme is simpler in that it requires only one traversal on the tree. Figure 8-3 demonstrates the Cormen's numbering scheme. Each element of the DOM tree is annotated by a Cormen's number pair: an *entry number* when the element is first visited and an *exit number* when the element is left by the depth-first traversal.

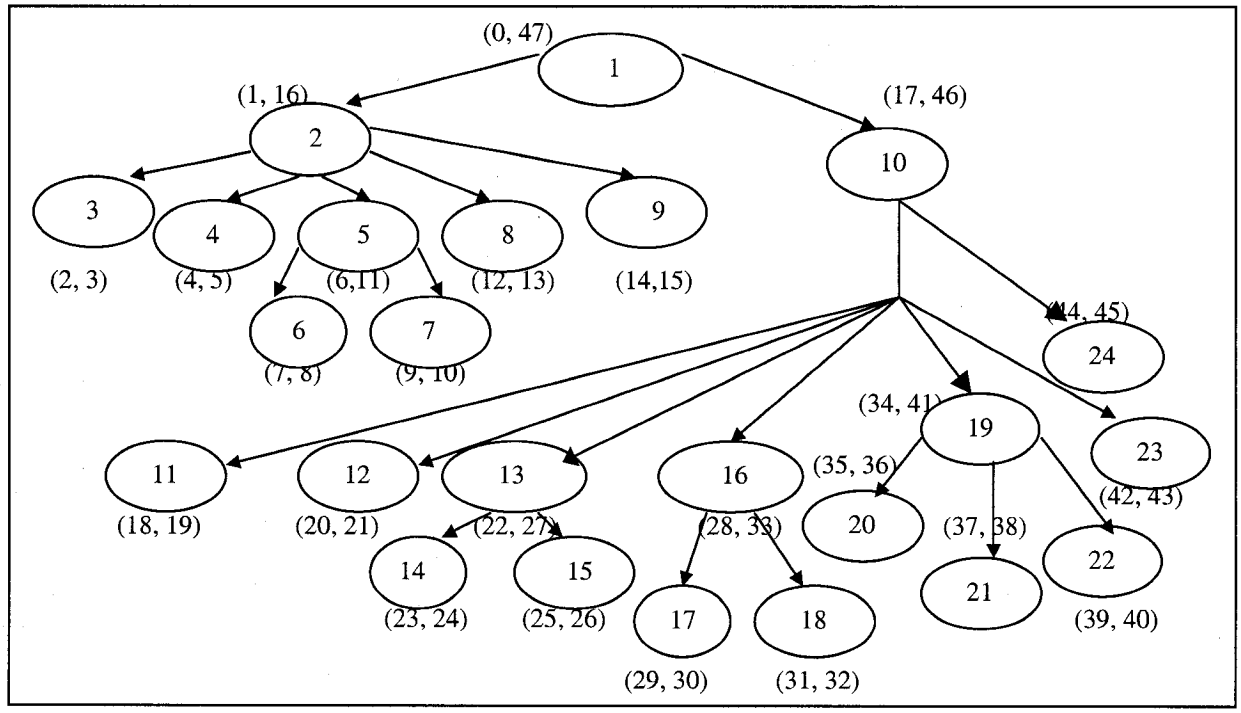


Figure 8-3: Cormen's Number Pair

The Cormen's numbering scheme is elegant and convenient. But it has one disadvantage. When an insertion or deletion occurs, the Cormen's number pair of all the nodes that were visited successively after the inserted or deleted one in the depth-first traversal must be recalculated. This could be costly, especially when the XML data is huge.

To get around the problem, we introduce a extending factor E_F. Instead of increasing the entry and exit numbers by one, we increase them by E_F, so to reserve extra spaces to

accommodate future insertion. Figure 8-4 shows the improved numbering scheme with E_F = 10.

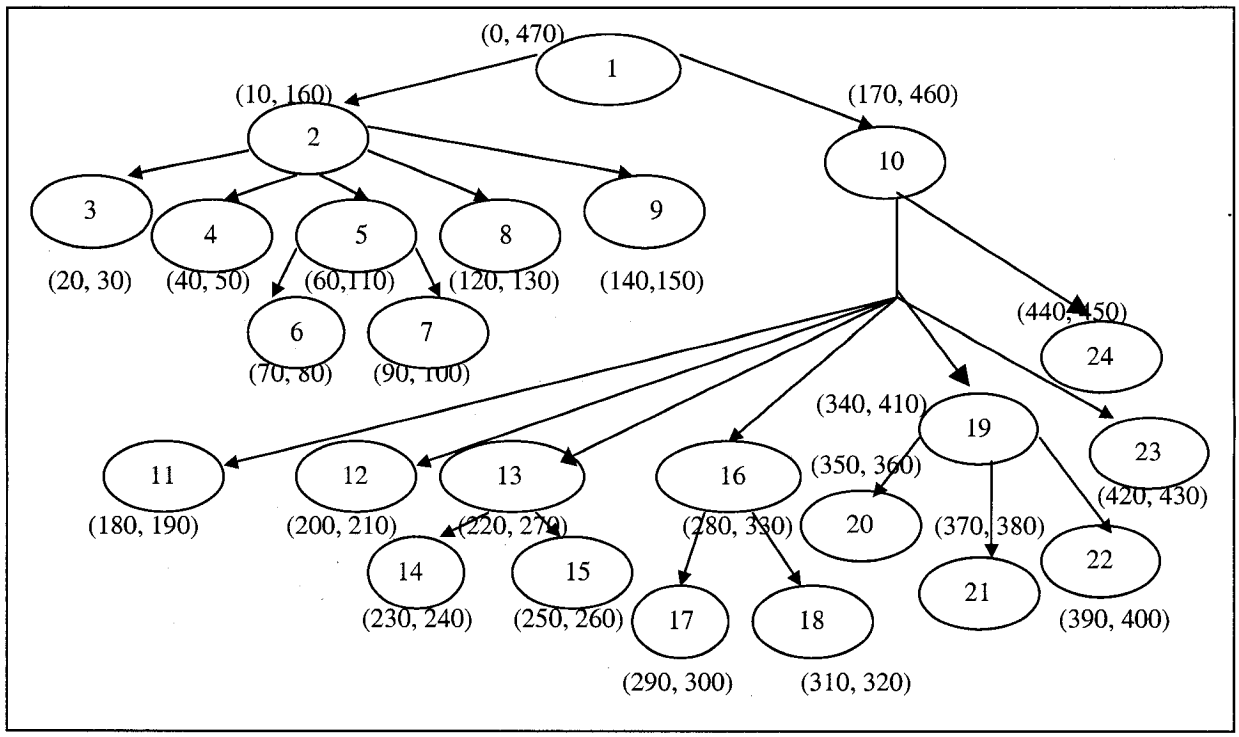


Figure 8-4: Improved Cormen's Number Scheming (E_F = 10)

After the improvement, the number scheme is more flexible. Now the entry and exit numbers need not to be recomputed every time an update occurs. For example, to insert a node before node 3, we may simply assign (13, 17) as the Cormen's number pair of the newly inserted node and keep those of the others unchanged.

An alternative solution is Li-Moon's proposal in [QLBM01]. Q.Li and B.Moon proposed a numbering scheme that uses *an extended preorder* and *a range of descendants*. The number scheme associates each node with a pair of numbers <*order, size*> as follows:

- *For a tree node y and its parent x, order(x) < order (y) and order(y) + size(y) ≤ order(x) + size(x). In other words, interval [order(y), order(y) + size(y)] is contained in interval [order(x), order(x) + size(x)].*
- *For two sibling nodes x and y, if x is the predecessor of y in preorder traversal, order(x) + size(x) < order (y).*
- *For a tree node x, size(x) can be an arbitrary integer larger than the total number of the current descendants of x.*

In Figure 8-5, the Li-Moon's numbering scheme is illustrated.

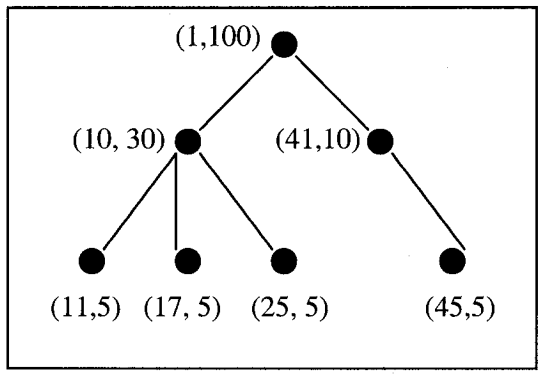


Figure 8-5: Li-Moon's Numbering scheme

Both Li-Moon's numbering scheme and our improved numbering scheme accommodate future insertions gracefully. We choose ours over Li-Moon's because we have already invented it and applied it to McXML before we explore Li-Moon's numbering scheme.

8.2.3.2. An Example of QIndex

QIndex is like a map, the keys of which are XPath expressions, and the values of which are RPPEs. The QIndex of an XML document is first created in the memory from scratch by parsing the XML DOM tree. Then it is saved in the disk as an index file for reuse purpose. Table 8-1 shows an example of QIndex on *bib.xml*:

Key	Value
bib	1, (0, 470)
bib/book	1, (10, 160) ; 2, (170, 460)
bib/book/year	1, (20, 30) ; 1, (180, 190)
bib/book/title	2, (40, 50) ; 2, (200, 210)
bib/book/author	3, (60, 110) ; 3 (220, 270) ; 4 (280, 230)
bib/book/author/last	1 , (70, 80) ; 1 ,(230, 240) ; 1 (290, 300)
bib/book/author/first	2, (90, 100) ; 2 (250, 260) ; 2 (310, 320)

Key	Value
bib/book/publisher	4, (130, 140) ; 6, (420, 430)
bib/book/price	4, (150, 160) ; 7, (440, 450)
bib/book/editor	5, (340, 410)
bib/book/editor/last	1, (350, 360)
bib/book/editor/first	2, (370, 380)
bib/book/editor/affiliation	3, (390, 400)

Table 8 - 1: An Example of QIndex

8.2.4. Maintenance Cost

By using RPPEs, QIndex has minimized the maintenance requirement. An update operation would not cause any Cormen's number pairs in the RPPEs to change assuming E_F is big enough. An update causes only the right siblings of the updated nodes to change their ordinal numbers in the RPPEs. The maintenance that QIndex needs is of low cost. In this section, we analyze the cost of the maintenance in terms of time complexity. To keep it simple, we assume the XML DOM tree involved is a balanced tree (which is usually true in reality), with n total number of nodes, m as its average scaling factor.

The algorithm to maintain QIndex is presented as follows.

```
1.    MaintainQIndexAfterInsert( InsertedNode )
2.    {
3.          RPPExpr ← construct a new RPPE;
4.          XPathExpr ← the XPath expression of the InsertedNode;
5.          insert {RPPExpr ; XPathExpr} into QIndex;
6.
7.          for each right sibling of the InsertedNode
8.          do      increase the ordinal number in the RPPE by 1;
9.          endfor
10.   }

1.    MaintainQIndexAfterRename( RenamedNode )
2.    {
3.          RPPExper ← the RPPE of the RenamedNode;
4.          OldXpathExpr ← the old XPath expression of the RenamedNode;
5.          delete {OldXPathExpr ; RPPExpr} from QIndex;
6.          NewXPathExpr ← the new XPath expression of the RenamedNode;
```

```
7.          insert { NewXPathExpr; RPPExpr} to QIndex;
8.
9.          for each child node ChildNode of the RenamedNode
10.         do      MaintainQIndexAfterRename(ChildNode);
11.         endfor
12.   }

1.    MaintainQIndexAfterDelete( DeletedNode )
2.    {
3.          delete the QIndex entry of the DeletedNode from QIndex;
4.          for each right sibling of the DeletedNode,
5.          do      decrease the ordinal number in the RPPE by 1;
6.          endfor
7.          for each child node ChildNode of the DeletedNode
8.          do      MaintainQIndexAfterDelete(ChildNode)
9.          endfor
10.   }

1.    MaintainQIndexAfterReplace(NewNode, OldNode)
2.    {
3.          MaintainQIndexAfterDelete(OldNode);
4.          MaintainQIndexAfterInsert(NewNode);
5.    }


1.    MaintainQIndex( UpdateType )
2.    {
3.          switch(UpdateType)
4.          {
5.                case Insert:
6.                      InsertedNode ← the inserted node;
7.                      MaintainQIndexAfterInsert(InsertedNode);
8.                      break;
9.                case Delete:
10.                     DeletedNode ← the deleted node;
11.                     MaintainQIndexAfterDelete(DeletedNode);
12.                     break;
13.               case Rename:
14.                     RenamedNode ← the renamed node;
15.                     MaintainQIndexAfterRename(RenamedNode);
16.                     break;
17.               case Replace:
18.                     OldNode ← the node to be replaced;
19.                     NewNode ← the node to replace the old one;
20.                     MaintainQIndexAfterReplace(NewNode, OldNode);
21.                     break;
```

```
22.                 default:
23.                         throw WrongUpdateTypeException;
24.             }
25.     }
```

QIndex is implemented as a Hashtable. Looking up a RPPE by an XPath expression takes O(1) in average case. The total time complexity of searching all the right siblings of a node is O(n/m^{h+1}), if the node is at the height of h, for at the height of h, there are at most $\lceil n/m^{h+1} \rceil$ nodes. Thus *MaintainQIndexAfterInsert( )*, in which the most expensive step is to increase the ordinal number of the of RPPEs of all the right siblings (Line 7 to Line 9), takes O(1) + O(n/m^{h+1}) = O(n) time complexity.

MaintainQIndexAfterRename( ) takes O($log_m{}^n$) * m * O(1) = O($log\ n$) time complexity, for the height of the DOM tree is $\lfloor log_m{}^n \rfloor$, and the most expensive step in *MaintainQIndexAfterRename( )* is the recursive call on all the child nodes (Line 9 to Line 11). The *MaintainQIndexAfterDelete( )* spends most of the time on the recursive call on the child nodes too (Line7 to Line 9). It also spends some time on decreasing the ordinal numbers of the RPPEs of the right siblings (Line 4 to Line 6). Its total time complexity is (O(1) + O(n)) * (O($log\ n$) * m * O(1)) = O($n\ log\ n$).

MaintainQIndexAfterReplace( ) contains *MaintainQIndexAfterInsert( )* (Line4)and *MaintainQIndexAfterDelete( )* (Line 3), so its complexity is O(n) + O($n\ log\ n$) = O($n\ log\ n$). Hence, *MaintainQIndex( )* takes max(O(n) , O($log\ n$) , O($n\ log\ n$) , O($n\ log\ n$)) = O($n\ log\ n$) time complexity. This is very low cost.

8.2.5. Advantages of QIndex

The advantages of QIndex are obvious:

- It is tight in structure.

  QIndex uses XPath expressions as keys. Normally, an XML document is not very deep in structure. Even very big XML documents are often wide but not deep. So, the number of unique XPath expressions in an XML document are usually not very big, which means there are not a lot of entries in QIndex.

  The values of QIndex are RPPEs. Though there could be a big number of RPPEs, they are all grouped by the XPath expressions. Besides, for each XPath expression, the list of the RPPEs is sorted by the value of their Cormen's number pairs.

- It is small in size.

  QIndex stores RPPEs instead of PPEs, as a result, redundantly repetitive information is avoided. Also, compared with PPEs, RPPEs are much smaller in size.

- It does not require very much maintenance. The time complexity of the maintenance is only O($n \log n$), where n is the total number of nodes in the XML DOM tree (See Section 8.2.4.).

- It improves the efficiency of addressing XML nodes.

  With QIndex, it is simple and efficient to address an XML node:

  Step 1: Find the RPPE of the XML node by looking up QIndex, using its XPath expression as the key. In case that wildcards are used in the XPath expressions of an update command, the XPath expressions are first reinterpreted to regular XPath expressions.

  Step 2: Construct the PPE of the XML node from its RPPE.

  Step 3: Locate the XML node in the XML DOM tree, following the path from the root node to the XML node specified by the PPE.

  Step 1 and Step 3 are straightforward. The algorithm of constructing PPEs from RPPEs is as follows:

```
String ConstructPPE (String XpathExpr, String RPPExpr)
{
        if (XpathExpr contains no '/') then
                return RPPExpr;
        XpathExpr = XpathExpr.substring(0, XpathExpr.lastIndexOf('/'));
        String [] newRPPExprList = the list of RPPEs stored in QIndex
                                        under the key XpathExpr;
        for each newRPPExpr in newRPPExprList  do
                if  the Cormen's  number pair of newRPPExpr strictly
                    contains that of RPPExpr
                then  return ConstructPPE(XpathExpr , newRPPExpr) +
                    '/' + RPPExpr;
        throw WrongInputException;
}
```

8.2.6. Performance

In this section, we present some test data to show the performance of the QIndex. The test on the QIndex is done on several XML documents. Table 8-2 shows the statistics on these XML documents.

XML document	Size (K)	Depth	Scaling Factor	Fan Out
bib.xml	1	4	2	22
Project.xml	14	4	3	401
Course.xml	36	4	5	1,201
Studentproject.xml	230	4	10	70,00
Student.xml	400	4	7	13,602
Bigstudent.xml	2,400	4	7	70,000

Table 8 - 2: Statistics on The Tested XML Document

8.2.6.1. Query-All & Update-All Performances

Some queries returns the results that scatters the whole data, which in this thesis is called "Query-All" for simplicity. And we call those updates that affect the whole data set "Update-All". For example, the following query returns a result that spreads the whole data set, for in *bib.xml*, the *<book>* nodes are child nodes of the root:

```
for $b in document("bib.xml")//book
return $b
```

The following update command is an "Update-All":

```
for $b in document("bib.xml")//book,
    $a in $b/author
update $b
{
        rename $a to "writer";
}
```

Chart 8-4 shows the Query-All performances of McXML before and after QIndex is installed. The x-axis shows the size of the XML data being queried. The y-axis shows the query time in milliseconds. The chart shows that when the data size is big, the Query-All performance with QIndex is not as good as that without QIndex. This makes sense, for as any indexing structure, QIndex does not optimize the performance of Query-All queries. On the contrary, since using QIndex enviably adds some overheads, the Query-All performance is even slightly degraded.

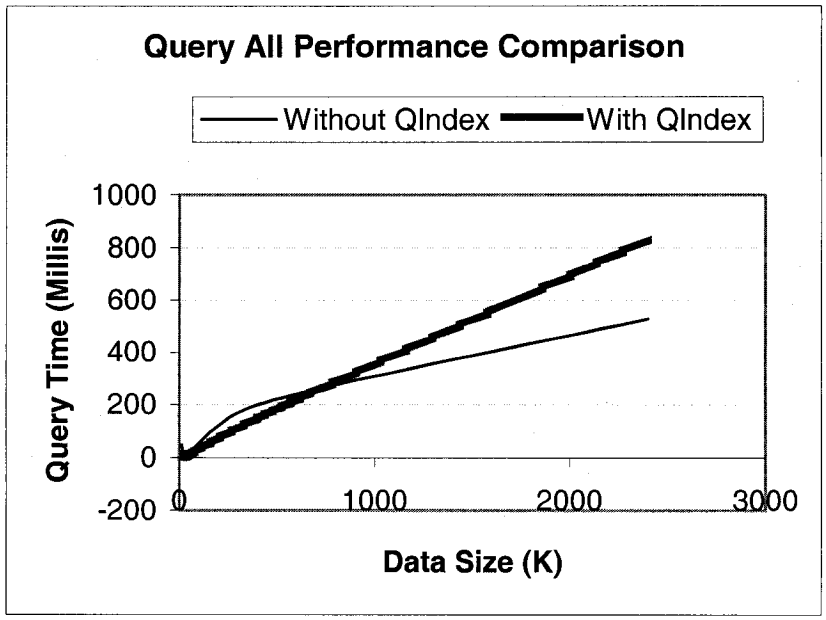


Chart 8-4: Query-All Performance Comparison (With QIndex vs. Without QIndex)

Chart 8-5 shows the Update-All performances of McXML before and after QIndex is installed. The x-axis shows the size of the whole XML data. The y-axis shows the update time in seconds. For the same reason that we stated in the previous paragraph, the Update-All performance with QIndex is not as good, though the difference is only slight.

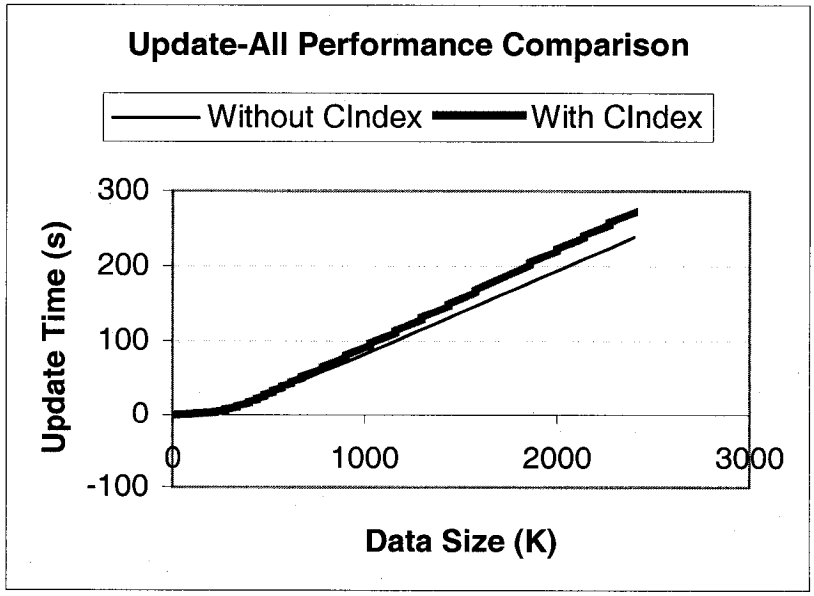


Chart 8-5: Update-All Performance Comparison (With QIndex vs. Without QIndex)

In conclusion, for Query-All and Update-All commands, we should not use any index structure including QIndex. Normal way of execution is better.

8.2.6.2. Query-None & Update-None performance

Some queries return no data, which in this thesis is called "Query-None" for simplicity. And we call those updates that affect no data "Update-None". For example, the following query asks for data under an non-existing path, and returns no data:

```
for $x in document("bib.xml")//book/author/last/name
 return $x;
```

The following update command is a "Update-None":

```
for $x in document("bib.xml")//book/author/last/name,
let $y = $x/alias
 update $x
{
        delete $x;
}
```

Chart 8-6 shows the Query-None performance of McXML before and after QIndex is installed. The x-axis shows the size of XML data being queried. The y-axis shows the query time in milliseconds. As we can see, with QIndex, the result returns immediately, while without QIndex, it still takes some time to figure out that no data is qualified to be returned.

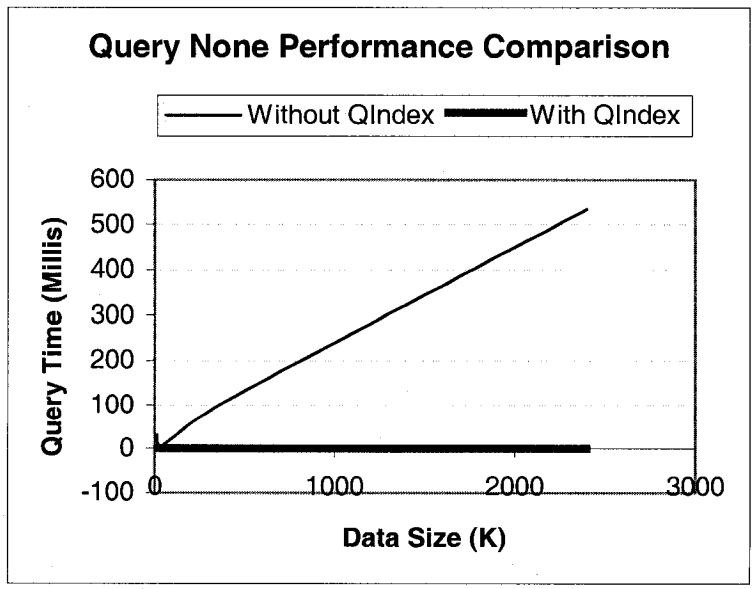


Chart 8-6: Query-None Performance Comparison (With QIndex vs. Without QIndex)

Chart 8-7 shows the Update-None performances of McXML before and after QIndex is installed. The x-axis shows the size of the whole XML. The y-axis shows the update time in milliseconds. Similarly, with QIndex, the Update-None performance is better.

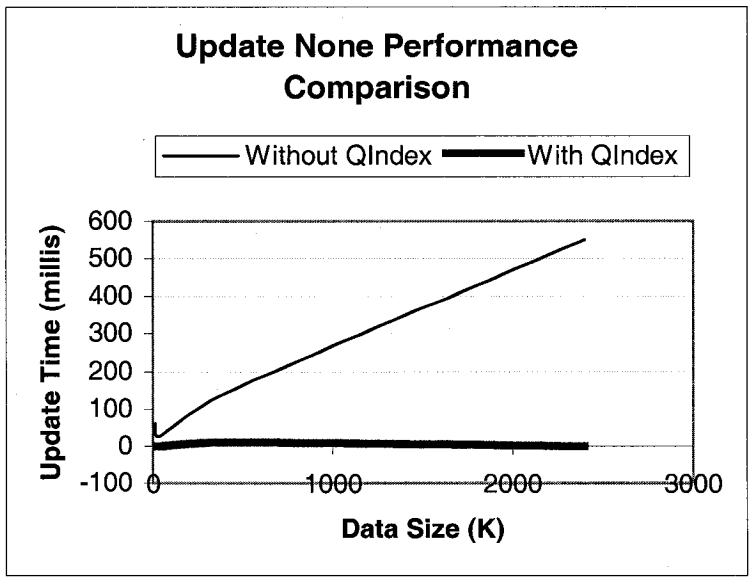


Chart 8-7: Update-None Performance Comparison (With QIndex vs. Without QIndex)

8.2.6.3. Query-Part & Update-Part Performance

Most of the query returns a small part of the data as result, which in this thesis are called "Query-Part" for simplicity. And we call those updates that affect a small part of the data "Update-Part". For example, the following query returns only a small XML data branch:

```
for $f in document("bib.xml")//book/editor/affiliation
return $f;
```

The following is an "Update-Part":

```
for $e in document("bib.xml")//book/editor
let $f = $e/affilliation
update $e
{
        insert <middle>N.A.</middle> before $f;
}
```

Chart 8-8 shows the Query-Part performances of McXML before and after QIndex is installed. The x-axis shows the size of the data being queried. The y-axis shows the query time in milliseconds. The tested query returns one record as result. In this case, the Query-Part performance with QIndex is optimized.

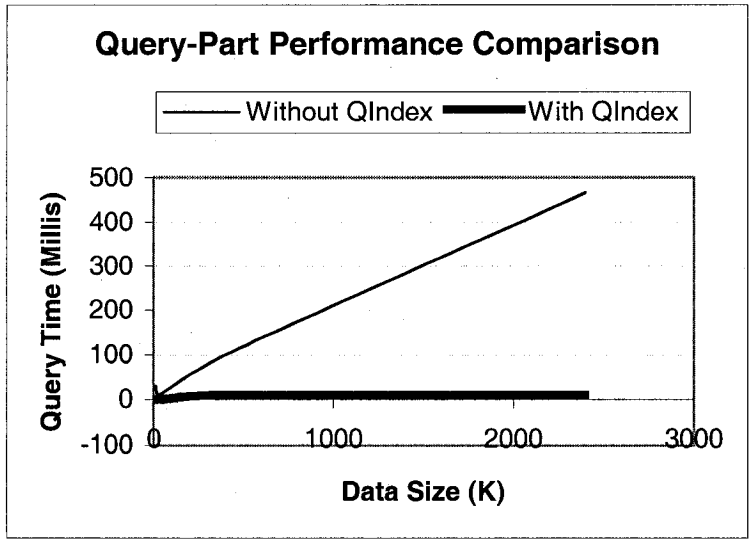


Chart 8-8: Query-Part Performance Comparison (With QIndex vs. Without QIndex)

Chart 8-9 shows the Update-Part performances of McXML before and after QIndex is installed. The x-axis shows the size of the whole data. The y-axis shows the update time is milliseconds. The tested update affects one particular record. As explained in Section 8.2.1., QIndex requires some maintenance to be up-to-date. This is a side affect of QIndex that might slow down the update performance to some extent. However, in this case, the update performance with QIndex is better than that without QIndex. This is because update operations contain solving variable binding, which is actually a data query part. QIndex works so well in optimizing the data query performance that the overhead of maintaining QIndex is not visible.

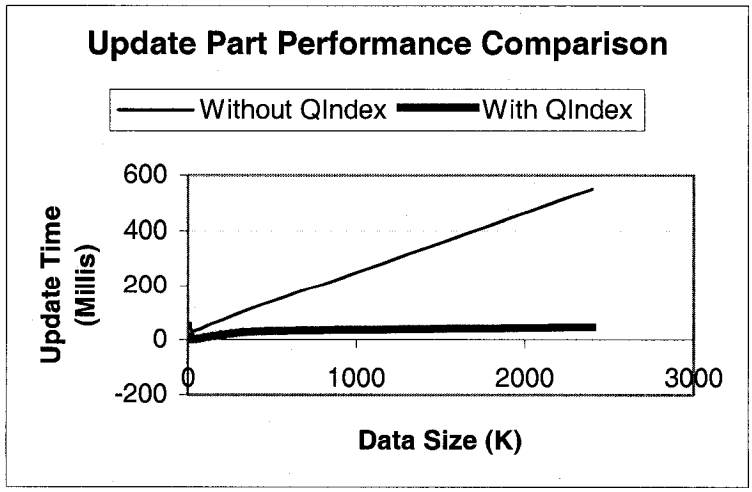


Chart 8-9: Update-Part Performance Comparison (With QIndex vs. Without QIndex)

Note that in the above charts, the update measurements seem linear when QIndex is used, whereas, in Section 8.2.4, we have analyzed that computational complexity for update maintenance for QIndex is O(*n log n*). This is simply because the test XML data is not large enough to show the superlinear curvature.

8.2.7. Comparison With Other Indexing Structures

In Chapter 6, some indexing structures are described. In this section, let's compare QIndex with some of those indexing structures to show more clearly the advantages and disadvantages of QIndex. Unfortunately, there is not enough information on the performances of these indexing structures. Especially, there is no information on their update & maintenance performances. As a result, we are unable to give a comparison in performance.

Lore has four index structures. One of them, Tindex is a full text index, which will not be discussed here. The other three are all structural indices. Vindex quickly locates specific leaf objects. Lindex retrieves the parents of a node via a given label. Pindex lookups for a path *p* returns the set of nodes *O* reachable via *p*. QIndex of McXML is like a combination of Vindex and Pindex. The main functionality of QIndex is to look up a XPath expression and returns all the nodes under it. What Vindex does is a special case for QIndex. What Pindex returns (all the nodes reachable by the path) is a super set of what QIndex returns (all the nodes under the path). As far as data retrieval by path is concerned, QIndex is more specific. Lindex of Lore is only useful in Lore's data model. McXML uses DOM as its logical data model, which itself provides the functionality that Lindex has except that Lindex is also capable of finding the referencing node. QIndex is advantageous in that its functionalities are more specific and it is integral to all the XML database systems that use DOM or some similar data model. Lore's indexing structures are advantageous in that they provide more functionalities. Lore's functionalities are distributed, which could be desirable and undesirable. Lore has a cost-based query optimizer to piece together all the indexing structures to create efficient query plans, in this sense, it is good to have the functionalities distributed. However, the distribution makes it very complicated to synchronize all the indexing structures in case of XML data

updates. Of course, this is not an issue for Lore, for it does not support XML data updates. But the indexing approach itself is disadvantageous in this sense.

Natix has a structural index, XSAR, too. Like QIndex, it also uses the Cormen's number scheme to indicate parent-child relationship among nodes. Actually, QIndex was to a great extent enlightened by XSAR. XSAR works on top of a relational database, which gives it access to the functionalities of the relational database to simplify its work. However, at the same time, it loses its independency. Besides, XSAR does not reserve extra number pair space to accommodate future insertions and it does not provide a mechanism to synchronize itself with update operations. Therefore XSAR is only useful where update operations are not supported. In actual, XSAR was originally a component of a XML data search engine called Mumpits [TFGM00]. Whereas, QIndex of McXML is independent, for it creates its own index files instead of relying on relation database tables. Also, QIndex improves Cormen's numbering scheme by introducing a factor E_F. It is compatible to update operations and does not require a lot of maintenance. (See Section 8.2.3.1.)

In [EKO02], a hybrid-indexing mechanism is proposed. It is like a combination of Lore's Pindex and Vindex with inverted file implementation. (In Lore, Vindex is implemented with B+ tree). It does not provide a way of synchronizing itself with XML update operations. What is worse, it is almost impossible to be improved otherwise, for inverted files are too much affected by XML updates. Plus inverted files are often very large in size. Maintaining them would require far too much effort. It is safe to say, such a hybrid index structure could not survive XML updates.

In summary, QIndex of McXML is advantageous in that it functions well where XML updates are supported. There is a lot for McXML to learn from other indexing structures, too. For example, it could learn from Lindex of Lore to add the functionality of retrieving the referencing node of an XML node.

Summary

McXML is a successful piece of work, with which, the author showed that her approach of designing a native XML database management system that supports both XML queries and XML updates is feasible. Some indexing structures were built on McXML. They were proved by solid test data to be efficient in lowering XML query cost without increasing XML update cost. This proved that indexing structures are useful and should be an integral part of XML database systems.

However McXML is just a prototype, which is open to future improvements:

- McXML uses the XQuery engine of the KAWA complier (called KAWADriver in McXML) for XML query commands. McXML has developed its own driver, XMLDBDriver, which now is only used for handling XML update commands. The XMLDBDriver is capable of handling XML queries, too. What lacks is a ReturnStmt that returns all the XML data found. In the future, such a statement should be added. Then the XMLDBDriver would deal with all commands and the KAWADriver should be removed.
- In the future, McXML could develop a cost-based query optimizer, which is responsible to create efficient query plans that decide, for executing the XML query or update commands, whether to use an indexing structure or not, and if yes, which indexing structure should be used.
- As a Client/Server architecture, McXML is extendable to support multiple clients. In the future, this feature could be added to McXML.

References

[AQMW97]: S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, "*The Lorel Query Language for Semistructured Data*", International Journal on Digital Libraries, 1(1):68-88, April 1997.

[AWJW00]: Ann Wollrath and Jim Waldo, "Trail: RMI", 2000, http://java.sun.com/docs/books/tutorial/rmi.

[BPS98]: Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, "*Extensible Markup Lauguage (XML) 1.0*", February. 1998, http://www.xml.com/axml/testaxml.htm.

[CCDF99]: Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, etc, "*XML-GL, a Graphical Language for Querying and Restructuring XML Documents*", Technical Report, Universita di Milano, March 1999.

[CEL97]: Castedo Ellerman, *"Channel Definition Format (CDF)"*, March 1997, http://www.w3.org/TR/NOTE-CDFsubmit.html.

[CLRS89]: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein, *"Introduction to Algorithms"*, MIT Press and McGraw-Hill, 1989.

[DFFL98]: Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu, "*XML-QL, A Query Language for XML*", August 1998, http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/.

[DKM96]: D.Dervos, P.Linardis and Y.Manolopoulos, "*Perfect Encoding: a Signature Method for Text Retrieval*", Proceedings 3rd International Workshop on Advances in Databases and Information Systems (ADBIS'96), Moscow, Russia.

[EKO02]: Evangelos Kotsakis, "*Structured Information Retrieval in XML documents*", SAC 2002.

[FHKM02]: Fiebig, T.; Helmer, S.; Kanne, C.-C.; Moerkotte, G.; Neumann, J.; Schiele, R.; Westmann, T., *"Natix: a technology overview"*, Web, Web-Services, and Database Systems. NODe 2002 Web- and Database-Related Workshops, Oct. 2002.

[GMW99]: R. Goldman, J. McHugh, and J. Widom, "*From Semistructured Data to XML: Migrating the Lore Data Model and Query Language*", Proceedings

of the 2nd International Workshop on the Web and Databases (WebDB '99), Philadelphia, Pennsylvania, June 1999.

[HTML03]: W3C, *"HyperText Markup Language (HTML)"*, http://www.w3.org/MarkUp.

[IDTE03]: IBM, "", January 2003, "*DB2 Text Extender*", http://www-3.ibm.com/software/data/db2/extenders/text/index.html

[IDXE03]: IBM, "*DB2 XML Extender*", January 2003, http://www-3.ibm.com/software/data/db2/extenders/xmlext/

[ISVL02] IBM Silicon Valley Lab, "*Database Technology for e-Business*", December 2002 http://www7b.software.ibm.com/dmdd/library/techarticle/0212malaika/0212malaika.html

[JCSD99]: James Clark and Steve DeRose, "*XML Path Language (XPath) Version 1.0*", November 1999, http://www.w3.org/TR/Xpath.

[JMI01]: Julia Mildenberger, "*A generic approach for document indexing: Design, implementation, and evaluation*". Master's thesis, University of Mannheim, Mannheim, Germany, November 2001 (in German).

[LMA00]: Lars Martin, "XUpdate – XML Update Language", November 2000, http://www.xmldb.org/xupdate/xupdate-req.html.

[MFJR00]: M. Fernandez and J. Robie, "*XML query data model, W3C working draft 11 May 2000*", Technical report, World Wide Web Consortium, May 2000, http://www.w3.org/TR/2000/WD-query-datamodel-20000511.

[MMA03]: Massimo Marchiori, "*XML Query (XQuery)*", September 2003, http://www.w3.org/XML/Query.

[MPE02]: Michalis Petropoulos, "*OQL Tutorial*", February 2002, http://feast.ucsd.edu/People/michalis/notes/O2/OQLTutorial.htm.

[MWAL98]: J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman, "*Indexing Semistructured Data*", Technical Report, January 1998.

[PBRM03]: Per Bothner and R. Alexander Milowski, "*Kawa, the JAVA based Scheme System*", June 2003, http://www.gnu.org/software/kawa.

[PEB03]: Paul E. Black, "Inverted File Index", April 2003, http://www.nist.gov/dads/HTML/invertedFileIndex.html.

[PFD82]: P. F. Dietz. "*Maintaining order in a linked list*", ACM Symposium on Theory of Computing, pages 122--127, 1982.

[PLLH01]: Peiya Liu and Liang H. Hsu, "*A Logic Approach to XML Document Update Query Specifications*", Internationales Congress Centrum (ICC) 21-I5 May 2001.

[POJ00]: Pasi Ojala, "*Compression Basics*", 2000, http://www.cs.tut.fi/~albert/Dev/pucrunch/packing.html.

[QLBM01]: Quanzhong Li, and Bongki Moon, *"Indexing and Querying XML Data For Regular Path Expressions"*, 27th VLDB Conference, Roma, Italy, 2001.

[RCF00]: Jonathan Robie, Don Chamberlin, Daniela Florescu, "*Quilt, An XML Query Language*", March 2000, http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html.

[RHJ99]: Dave Raggett , Arnaud Le Hors, and Ian Jacobs, "*Document Type Definition*", December 1999, http://www.w3.org/TR/REC-html40/sgml/dtd.html.

[RLS98]: J. Robie, J. Lapp and D. Schach, "*XML Query Language (XQL)*", Proc. of the Query Languages workshop, Cambridge, Mass., Dec 1998, http:://www/w3.org/TandS/QL98/pp/xql.html.

[SGML03]: W3C, "Overview of SGML resources", http://www.w3.org/MarkUp/SGML.

[TFGM00]: Thorsten Fiebig, Guido Moerkotte, *"Evaluating Queries on Structure with eXtended Access Support Relations*", WebDB (Informal Proceedings), pages 41--46, Dallas, Texas, 2000.

[TIHW01]: Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld, "*Updating XML*", SIGMOD conference 2001.

[WBGA01]: Jennifer Widom, Andre Bergholz, Roy Goldman, Serge Abiteboul, etc, "Lore", 2001, http://www-db.stanford.edu/lore.

[WDOM00]: W3C, "*Document Object Model (DOM) Level 2 Core Specification*", November 2000, http://www.w3.org/TR/DOM-Level-2-Core.

[WSQL03]: W3Schools, "SQL Tutorial", 2003, http://www.w3schools.com/sql/.

[WXML03]: W3C, *"Extensible Markup Lanaguage"*, http://www.w3c.org/xml.

[XDBI03]: The XML:DB Initiative, "*XML:DB Initiative for XML databases*", 2003, http://www.xmldb.org/index.html.

[XSLT99]: W3C, "*XSL Transformations (XSLT)*", November 1999, http://www.w3.org/TR/xslt.

[XSD99]: W3C, "*XSL Transformations (XSLT)*", November 1999, http://www.w3.org/TR/xslt.