NextDB : a native XML database system

Khin Myo Win

2005

Khin, M. W. (2005). NextDB : a native XML database system. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2671>

<https://doi.org/10.32657/10356/2671>

Nanyang Technological University

Downloaded on 26 Apr 2023 19:24:41 SGT

NEXTDB: A NATIVE XML DATABASE SYSTEM

KHIN MYO WIN

SCHOOL OF COMPUTER ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

2005

NextDB: A Native XML Database System

Khin Myo Win

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Doctor of Philosophy

2005

Acknowledgements

I would like to thank the University of Computer Studies, Yangon (UCSY) and the Nanyang Technological University (NTU) for giving me the opportunity to study and providing me the scholarship. I want to thank the School of Computer Engineering (SCE) for granting me the opportunity and providing a most conducive research environment in the Center for Advanced Information Systems (CAIS) to work my research activities.

I would like to first express my deepest gratitude to my advisor, **Associate Professor Ng Wee Keong**, for his constant encouragement, support and constructive advice. As a supervisor, he demonstrates knowledge and patience that I need as a research student. I am thankful to him for putting his trust in me and giving me this opportunity for doing the research on the most advanced research area — semistructured data. Without his stable support and enthusiastic encouragement, it would have been impossible for me to achieve the progress stated in this report. His efforts in organizing weekly presentation in CAIS give me encouragement and enriching. The discussions have enhanced the breath of my knowledge about the topic and improve my presentation skill. From his careful and meticulous approach, I really learn a lot.

Next, I also would like to thank **Associate Professor Lim Ee Peng** for his many invaluable suggestions, guidance and advice. Also thanks to the technical staffs **Mr. Lai Chee Keong** and **Ms. Tan Lay Choo**, for their assistance in providing the administration and technical support to me.

Special thanks go to **Woon Yew Kwong** and **Ong Kok Leong**, two research students in CAIS, for their generous sharing of ideas about research, as well as the help given in thorough reviewing the things I wrote. They spend much time with me to contribute suggestions in my research directions, and give encouragement.

I would also like to express my gratitude to my friends, **Yu Hai, Zhao Bin, Emerald Rosasillfiani** and **Liu Qincai**, creative students driven from the School of Computer Engineering, whom I have the honor to work with them. They spend much time to discuss my research issues and collaborate much in my research work.

My fellow research students in CAIS, **Myo Myo Naing, Liu Zehua, Wang Yida, Sun Aixin**, and **Li Zhao** certainly deserve much credit too for their help, listening ears and encouragement. Thank you all!

Last but not least, I would like to thank **my parents**. They have been looking after my daughter, **Yadi Nay Myo**, in my country during the times when I neglected them in the course of my work. Their understanding, love, and care give unwavering confidence in me. I could not have come this far without their love and support. I have to thank to my husband, **Nay Myo Aung**, the person without whom this report could have never been written. His support and unconditional love are beyond words. His understanding is certainly instrumental in helping me to concentrate in my work without reservations.

In short, this report cannot appear and my research will inevitably be destined for failure without these all above people.

Contents

Acknowledgements	ii
List of Figures	viii
List of Tables	ix
Summary	x
1 Introduction	1
1.1 Background	1
1.1.1 WWW and Semistructured Data	3
1.1.2 Markup Languages	6
1.1.3 XML Document	9
1.2 Motivations of the Thesis	14
1.3 Contributions of the Thesis	15
1.4 Thesis Organization	16
1.5 Summary	17
2 Managing XML Data in Native Structure	18
2.1 Related Work	18
2.1.1 Lore	19
2.1.2 Natix	19
2.1.3 TIMBER	20
2.1.4 Tamino	20
2.2 NextDB: A Native XML Database System	21
2.2.1 Storage & Index Structures	22
2.2.2 Query Processor	24

2.2.3	Transaction Management	24
2.2.4	Visualizing XML data	26
2.3	Summary	26
3	The Native XML Storage System	28
3.1	Related Work	29
3.1.1	File-Based Systems	30
3.1.2	Conventional Database Systems	30
3.1.3	XML-Based Native Systems	37
3.1.4	Indexing Schemes for XML Data	40
3.2	ENAXS	40
3.2.1	Storage Structure	41
3.2.2	Index Structure	50
3.2.3	Enhancing Index Capability with Value Index	55
3.3	Performance Evaluation	56
3.3.1	Experimental Environment	56
3.3.2	Experimental Data	56
3.3.3	Storage Space Utilization	57
3.3.4	Data Parsing and Storing Time	61
3.4	Summary	62
4	Query Processing	64
4.1	Related Work	65
4.2	NextDB Query Processor	67
4.2.1	Designing Query Processor	68
4.2.2	Data Operators in the Query Processor	69
4.3	Implementation of Query Processor	74
4.3.1	Parser	75
4.3.2	Tree Builder	75
4.3.3	Result Generator	76
4.3.4	Query Execution	76

4.4	Performance Evaluation	79
4.4.1	Experimental Environment	79
4.4.2	Experimental Results	81
4.4.3	Experiment with Improved Indexing Schemes	82
4.5	Summary	83
5	Transaction Management for XML Data	84
5.1	Related Work	85
5.2	Serializing Concurrent Transactions in NextDB	85
5.3	Designing Data Model and Basic Operations	86
5.3.1	Data Model	87
5.3.2	Basic Transactional Operations	87
5.4	Designing XStamps Protocol	89
5.4.1	Features of the XStamps	89
5.4.2	Data Access Rules in XStamps	90
5.5	Implementation of Transaction Management System	93
5.5.1	Transaction Manager	93
5.5.2	Resource Manager	94
5.5.3	Intercommunicating with Recovery Management	94
5.6	Performance Evaluation	95
5.6.1	Experimental Model	95
5.6.2	Experimental Results	96
5.7	Summary	97
6	Visual Interface for XML Data	99
6.1	Related Work	99
6.2	NextDB User Interface	100
6.2.1	Data Management Interfaces	102
6.2.2	Query Interfaces	105
6.3	Usability Study Results	107
6.4	Summary	109

7 Architectural Framework for Native XML Data Management	111
7.1 Related Work	112
7.2 Data Model and Design Methodology	113
7.2.1 The Importance of Architectural Design	113
7.2.2 Importance of Data Model	113
7.3 Design of Architectural Framework	114
7.3.1 Desirable Features and Constraints	114
7.3.2 Framework for Data Management	118
7.4 Validation	120
7.4.1 Natix	120
7.4.2 NextDB	121
7.4.3 Db4XML	121
7.4.4 TIMBER	122
7.5 Future Directions	123
7.6 Summary	125
8 Conclusions	126
8.1 Summary of the Thesis	127
8.2 Future Work	129
A List of Publications	131
References	132

List of Figures

1.1	A Sample XML document.	10
1.2	A Sample XML Schema.	13
2.1	The General Architecture of NextDB.	22
3.1	The General Storage Architecture of ENAXS.	44
3.2	External Node Record Format	45
3.3	The Internal Node Record Format.	46
3.4	The Index Structure of ENAXS	50
3.5	Extended Node Groups Record Format	51
3.6	The Sample DTD and an XML Document.	54
3.7	A Sample Schema Tree and Its Inverted Representation	55
3.8	Size of Data Files and Repositories for Shakespeare's Play (1)	58
3.9	Size of Data Files and Repositories for Shakespeare's Play (2)	58
3.10	Size of Repositories for Various XMark Data Sets	59
3.11	Block Allocations for Shakespear's Play and KVJ Data Sets	61
3.12	Block Allocations for XMark Data Set	61
3.13	Parsing and Storing Time for different Data Sizes of XMark	62
3.14	Performance Comparison for Parsing XML Documents	62
4.1	The Statement Format for NextDB Algebraic Operators.	74
4.2	Main Components of NextDB Query Processor.	75
4.3	The Parser.	75
4.4	The Query Execution in NextDB.	76
4.5	A Sample Document for a Bibliography Database.	77

4.6	A Sample Query Command File.	78
4.7	A Query Tree Built from the Sample Query.	78
5.1	Transaction Management System in NextDB	93
5.2	Throughput comparison for 10, 50, 100 and 200 transactions	97
6.1	Architecture of NextDB User Interface	100
6.2	Main Interface of NextDB	102
6.3	User Interface for Opening and Creating Database Contents	103
6.4	The Workflow of Importing and Exporting XML Data	104
6.5	The Interfaces for Importing and Exporting	105
6.6	Text-Based DTD Editor	106
6.7	Graphics-Based DTD Editor	107
6.8	XML Editor	108
6.9	The Workflow of Querying Data in NextDB	108
6.10	Query Interface	109
7.1	The General Architectural Framework for XML Data Management	118
7.2	Document Splitting in Natix	120
7.3	Node Record Format in NextDB.	122
7.4	The Record Format of an Element	122
7.5	The Record Format for Schema	122
7.6	Future Architectural Framework for XML Data Management	124

List of Tables

3.1	XML Data Sets for Performance Evaluation	57
3.2	Node Group Size for Shakespeare, KJV, and XMark Data Sets	60
4.1	Summary of Data Operators	69
4.2	Queries Used in Performance Evaluation.	80
4.3	Block I/Os for Query Evaluation	82
4.4	Comparison of Query Execution Time.	83

Summary

World Wide Web has been growing rapidly in the last few years and has become the main medium for information interchange among various communities. The concept of document preparation and distribution over the Web has been altered, and many recent applications contain semistructured data that is irregular and evolves rapidly. The Extensible Markup Language (XML) is a self-describing meta-language and fast emerging as a dominant standard for the Web data exchange among various organizations. As the growth of the Web and Internet applications have been producing and consuming large volumes of XML data, an efficient database management system is necessary to maintain the data. Conventional databases, which require all data to adhere to an explicitly specified rigid schema, are unable to provide efficient storage and retrieval of irregular and hierarchical-structured XML documents. A new database model that is specifically designed for XML is needed to address it.

In this thesis, we propose and present a design and architecture of a native XML database management system, *NextDB*, for large and complex XML documents. NextDB is composed of comprehensive database components that are specifically tailored to deal with the distinct characteristics of XML. This thesis explores the architecture of NextDB and the detailed descriptions of its components including a storage structure, a path-based and a value-based indexing schemes, a query processor, a transaction management scheme and a visual user interface.

Chapter 1

Introduction

1.1 Background

Within a few decades, the World Wide Web (the Web) has been growing incredibly and becomes the main information interchange among various communities. New Web servers and Web pages are in place, and a large volume of information are widely distributed on the Web in the form of pages and links created by using various markup languages. Among those, Hyper Text Markup Language (HTML) is the most popular language, and has been influencing the Web for several years as the standard data presentation language for delivering information over the Internet.

Over recent years, the concept of document preparation, distribution and publishing over the Web has been significantly altered, and semistructured data becomes in place in various arenas. More Web information are being published in form of semistructured data, and more Web-based applications have been producing and consuming such data for the purpose of information exchange. These data are hierarchically organized but flexible, irregular and rapidly changed in structure. Those are formed in such a way as semistructured documents in which data are represented as contents formatted by tags to describe the semantics of those data.

In practice, commonly used HTML has several limitations and incapacities to represent such data model and to meet the needs of new applications. Lack of data semantics, interoperability and automated information processing are main obstructions to adopt HTML for future Web data presentation language [SPS00]. To overcome those limitations intuitively, and to handle semistructured data efficiently, a

new markup language, called Extensible Markup Language (XML), has been introduced in recent years. XML is an emerging meta language that is a subset of SGML (Standard Generalized Markup Language). It allows users to compose semistructured documents, and automate Web information processing that is suitable for data exchange and interoperability. XML provides data-semantics and data-independence, and allows users to define their own set of markup tags. These markup tags are related to the contents of their documents for better self-describing of nested document structures. XML technology intends to enable media-independent electronic publishing, provide platform-independent protocols, allow automatic processing of receiving data, process data using inexpensive applications, and display data the way user wants.

With tremendous growth of XML data over the Web, an efficient data management system becomes a common requirement for any organization that needs to maintain, retrieve and manipulate XML data. XML itself is not a database. However, the other XML-based technologies and XML itself create a database-alike environment. On one side, this environment provides many of the things found in database such as storage (XML document), schemas (DTDs, XML schemas), query languages (XQuery, XML-QL, XPath, XQL), programming interfaces (SAX, DOM, and so on). However, on the other side, the features like indexing, security, transaction, multi-user access and triggers that are common in database environment are not actually provided by this environment.

The wide use of XML in data exchange and information representation demands persistent storage mechanisms that allow XML data to be managed in an effective way so that it is able to transfer data at any point of time without the needs of additional conversion. Moreover, it is capable to be efficiently looked-up, retrieved and manipulated with the support of indexes, transaction management and access control mechanisms. Those requirements inspire database researchers to intensively work in this area for appropriate resolution.

Several solutions have been introduced in recent years to address this problem. Some approaches focus on designing data models and storage structures, some on in-

dexing schemes, some on query languages and data retrieval techniques, and some on complete data management phenomenon. Most of these approaches use traditional database management systems such as Relational Database Management Systems (RDBMS), Object Oriented Systems (OODBMS), and Flat File Systems as underlying data model for managing XML data. The main reason could be that these databases are matured enough to handle large volume of data and provide robust data management features. For example, RDBMS gives a great number of advantages such as comprehensive query language and processing, transaction management, data security, multi-user access, backup and recovery and so on. However, these databases also have drawbacks and limitations in practical while dealing its rigid schema to XML's irregular structure practically. Data is necessary to be converted from XML to appropriate data format, for example, tables in RDBMS. This transformation is named *mapping* and it introduces an additional layer called *middleware* to perform mapping between two data structures. It is time-consuming and leads to performance degradation.

Similarly, OODBMS is based on the generic object structure that can conform the nested specifications of XML data, however, it demands large storage space that contributes to storage overhead. Like RDBMS, this approach also requires intensive mapping between object and XML data. In addition, unlike SQL of RDBMS, the object query language is not scalable enough to provide efficient and robust querying in the database that may introduce more weaknesses. *Flat-file system* is easy to implement and eliminates loading and reconstructing processes that are needed in storing and retrieving data. However, that approach does not provide efficient streamlined indexing and querying over the elements making the performance of data retrieval degraded.

1.1.1 WWW and Semistructured Data

The World Wide Web (WWW) is a network-based global information repository in which a collection of large amount of documents composing of different types of data are distributed over the Internet. With the rapid development of the Internet

technologies and usages, the Web becomes playing in a more vital role of our daily life. Since the beginning of the Web in 1989, it grows everyday at a surprising speed and the number of Web servers reaches 45,980,112 by the end of 2003 [Zak02]. The popularity of the Web has made it a prime vehicle for disseminating, sharing, transferring, storing, and publishing information over the Internet. People from all sorts of professions, educations, associations and organizations have been using the Web as a main media to conduct education, advertise and operate their business, communicate with one another, disseminate research, publish ideas, learn, play, and collaborate each other [NLH⁺98].

Obviously, anyone who wants to use the Web information efficiently should first understand the important characteristics of the Web. It includes the followings:

- The Web can be seen as a collection of directed graphs with Web documents corresponding to vertices and hyperlinks to directed edges.
- Web information is semistructured or unstructured, as opposed to more structured data found in conventional databases.
- Web information is dynamically and frequently updated. The owner of the information may modify it at any point of time.
- Web information can be discovered by either Web browsers that facilitate surfing the Web by navigating through links among Web pages, or search engines that support keyword-based queries over the huge collection of information.

Over the past years, the size of the Internet has become larger day after day and the volume of data available has been growing dramatically. Since the concept of document preparation, distribution and publishing has been changed, semistructured data becomes common in more Web pages and applications nowadays. Those data may have some structures, but it is not as rigid, regular and complete as data in traditional or object-oriented database management systems. They may evolve rapidly and change frequently. It is said to be “self-describing” as information inside is normally associated with a schema contained itself in the data to describe

association between data structure and semantics. It is not mandatory and there may be no schema associated with it. Such type of data normally has the following characteristics [Pet97, Suc98a, Suc98b]:

- *Implicit Structure*: There may or may not exist explicit description for some structure.
- *Irregular Structure*: Different patterns of data structure may exist. Some objects may have either missing attributes or multiple occurrences of the same attributes. Same attributes may have different types in different objects.
- *Implicit Schema*: Schema is not playing a vital role and is not explicitly required. It may exist in the document but it may be loose and is not strict. This is in contrast with conventional database systems in which schema is normative in nature.
- *Irregular Schema*: The size of schema may be large or small. Heterogeneous data will typically generate a large schema that is complicated and difficult to understand.
- *Heterogeneous Data*: Different types of structured and unstructured data from various internal and external sources may exist in a single document.
- *Irregular Data Type*: In contrast with conventional database systems, there is no strict type description in the schema. This type of inconsistency may lead to data heterogeneity.

Unlike structured data, semistructured data provides much flexibility in terms of data structure. However, this advantage may lead to other drawbacks such as data inconsistency, unreliability and incapacabilities. In general, there are several data sources that cannot be constrained by a specific schema. The Web itself is a data source that is treated as a heterogeneous database. A wide range of databases are distributed over the Web that motivates to have an extremely flexible format for data exchange among disparate databases. Even when dealing with structured data,

it may be helpful to view it as semistructured for the purpose of browsing. These several reasons are recently encouraging researchers and generating research interests in semistructured data [Pet97].

Since the beginning of the Web, much text-based data exchange and representation mechanisms have been introduced among data sources. Markup languages are one of those, and became quickly popular and commonly used by most of data sources today. The next section briefly introduces the most popular and widely-used markup languages over the Web.

1.1.2 Markup Languages

The Web is a system that supports a large collection of documents that is written and formatted using the same type of programming language called *Markup Language*. A markup language is a set of symbols that can be placed in the text of a document to delimitate and label the parts of that document [Ray01a]. These markup symbols are understood by the Web browsers, and the clients may view the formatted documents as Web pages through browsers. The capabilities of ease, simplicity and efficiency make markup languages the *de facto* standard for transmitting information over the Internet. Among them, SGML, HTML and XML are well-known and currently used as standard media for data presentation over the Web.

Standard Generalized Markup Language (SGML)

SGML is an international standard (ISO 8879) ¹ for more than ten years. It is originally designed to provide a way of describing text-based information and information exchange among organizations easily. Its main purpose includes to describe markup languages by allowing the author to provide a formal definition and to create their own tags that relate to their contents. SGML is not tied to a particular operating system and so it is portable from platform to platform. However, SGML is complicated to understand and difficult to integrate into an application. It is very powerful but very complex and many of its feature are rarely used. It is also difficult to interpret

¹<http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=16387>

without the definition of the markup language [FOJ⁺98]. In order to overcome such difficulties of SGML, another markup language, called *Hypertext Markup Language* (HTML), has been introduced.

Hypertext Markup Language (HTML)

HTML was originally an SGML application. It provides a rich set of markup symbols to format documents easily and presents them over the Web efficiently. Due to its efficiency, simplicity and ease of use, HTML becomes the standard hypertext language for distributing and exchanging information over the Internet within a short period. Currently, it is commonly used and supported by almost all Web browsers and Web page authoring tools. HTML brings along with many advantages [TH].

- HTML browsers are cheap or free, but very powerful and efficient.
- HTML is very simple and easy to learn. It provides a set of symbols and limited tags.
- HTML provides linking and navigating capabilities among Web pages.
- HTML supports some specialized structures but they are mostly used to effect a certain formatting look.
- HTML document browser interfaces are easy to build into existing products because of the simplicity of HTML.

Although HTML has many advantages, it also is a weak markup tool since it does not allow the creation of custom tags or the presentation of tags with different styles. It has a fixed tag set and there is no hierarchical relationship between HTML elements. It allows only the delivery of page oriented information and cannot carry information about the meaning of the content held within its tag. It is useful for data presentation and it does not support data processing and interoperability. Readers who wants to know more details about HTML are referenced to [Groc].

Web data today is predominantly semistructured. Although HTML provides a standard to create, display and access such data and navigate to different Web pages,

it does not provide a mechanism for describing contents and managing remote data. In addition, HTML provides a visualization of data contents but it does not describe how data should be represented and processed. There is no metadata information in an HTML text to facilitate other programs to understand the structure and content of the data. Due to these limitations, HTML lacks flexibility that is required for the syntactic specification of semistructured data [Zis00]. To address these limitations, a new markup language, called *eXtensible Markup Language (XML)*, has been introduced in recent years.

Extensible Markup Language (XML)

The World Wide Web Consortium (W3C) is an open, public organization whose task is to develop technology and standards for the Internet [Grob]. It has developed XML standards for efficient information exchange across via the Web. XML is initially derived from SGML and introduced as a subset of SGML that retains its flexibility but has a simplified design to suit the Web applications. The goal of the W3C's SGML activity is to enable the delivery of self-describing data structures of arbitrary depth and complexity to applications that require such structures. The idea behind the XML is to take benefits of SGML, remove the complicated parts, keep it light and make it work on the Web. The basic concept behind the XML is that data should be self-describing by means of tags. XML provides standards for users so that they can define their own tags and document structures. Indeed, XML grants a wide variety of flexibilities to the users. The followings are some of the benefits provided by XML.

- It merges multiple files together to form compound documents that make data consolidating and executions easy.
- It identifies where illustration is to be incorporated into text files, and the format used to encode each illustration.
- It provides processing control information to support programs such as documents validators and browsers.
- It allows the addition of editorial comments to files.

- It has a standardized template for producing particular types of documents.
- It simplifies business to business transactions on the Web.

XML is related to HTML, but XML is neither a predefined set of tags of the type defined for HTML, nor is it a standardized template for producing documents of different kinds. XML is a specification of a set of languages that can be used to annotate arbitrary character data, describes hierarchical structures and attributes meta-data to character data. XML mainly differs from HTML in three major aspects [Bos]:

- HTML has a fixed tag set but XML allows users to define new or own tags and attribute names.
- HTML is only a presentation technology and it has a simple document structure. On the other hand, XML is a data processing technology and its document structure can be nested to any level of complexity.
- HTML document contains markup codes to present its contents. XML documents can contain an optional description of its grammar for use by applications that need to perform structural validation.

XML appears as a content-based structure instead of the format-based structures imposed by designers. XML promises applications developers an extremely convenient format for storing many different kinds of information from dispersed databases. In the following section, we will shortly cover the structure of XML document and its components in order to provide preliminary knowledge to users to understand more clearly in the successive chapters.

1.1.3 XML Document

The basic of XML is an XML document. It is a text file with *.xml* extension that contains text, markup elements and actual data. Markup elements are created by means of user-defined tags and some constructs. User-defined tags extend the flexibility of XML that a user is not limited to a set of predefined tags in creating the XML document, rather user can create any tag that is specific to the application requirements.

```
<document>
  <person pnumber="a1">Susan &surname;</person>
  <person pnumber="a2">Jack &surname;</person>
  <person pnumber="a3" mother="a1" father="a2">Chelsea &surname;</person>
  <person pnumber="a4" mother="a1" father="a2">David &surname;</person>
</document>
```

Figure 1.1: A Sample XML document.

The XML standard provides a set of rules that specify some details, such as how to create tags and how an XML document can be structured, but within the XML framework, a user is free to define and use tags that best suit the data. A markup element in an XML document is actually the basis of the self-describing nature of XML, since it indicates what information is being described in the document.

As mentioned, XML documents are simple text documents composed of data and schema. XML documents comes in two flavors: *well formed* and *valid*. If an XML document complies to rules laid out in the XML specification, it can be said to be a well formed document. If the document complies to a document template for the data or to a schema, it is a valid document. The main difference between valid and well formed XML document is their relationship to a Document Type Definition (DTD) or schema. DTD serves as an implicit semantic schema for the XML document and it provides a set of rules for a document to be processed and displayed. Well formed documents are designed for use without DTD, and valid documents explicitly require it.

Generally, an XML document consists of XML declaration, elements, attributes, comments, processing instructions, and contents. Each XML element has a start tag and an end tag which serve as a metadata for defining document structure. The information between the tag is called the *content* of an element. In XML documents, the tags indicate the meaning of the data and not its appearance. In Figure 1.1, a simple XML document can be seen as an example.

Elements

An XML element consists of a start tag and an end tag pair along with the enclosed texts called *content*. An element may contain one or more sub-elements which are also known as *element content*. When an element content contains both elements and character data, it is known as *mixed element content*. XML also allows an element with no content termed as *empty content*.

XML elements are given names by the author to adequately describe the element's content. Furthermore, elements can have its ancestor, descendent and sibling elements to create structure. These traits provide the ability to construct documents with a semantically understandable structure and content. This should make it easier for developers and programmers to write programs which process the data enclosed in XML document structures.

Attributes

Elements, including empty elements, may have attributes associated with them which do not constitute the content of the element. Attributes may contain character data and may uniquely identify an instance of an element and refer to other locations in the document. It causes the inclusion of external binary data, or simply has a name value from a defined enumeration. Certain element's attributes may need to be defined or even given default values.

Comments

Comments are descriptive notes for XML code to provide better understanding to readers of the XML documents. Comments are text strings and may be found nearly anywhere in the XML document even after "root" element.

Processing Instructions

Processing instructions allow messages to be embedded within the XML code for external processing applications. Processing instructions may be found nearly anywhere in the XML document.

Text

XML documents are composed of sequences of text characters. The text contains markup and character data. The markup includes constructs such as tags, entity references and declarations that are provided to the XML processor. Within an XML document, data or content is stored as text which is a string of character data.

Character DATA (CDATA)

In some cases, text may contain numerous instances of the delimiter characters, but it does not contain any actual markup. It would be undesirable and difficult to convert all instances of those characters into entity references. XML allows one to store text as character data which may not contain other markup. Such a section of character data is known as CDATA.

Entity References

Entity references point to entities either defined in the document type definition (DTD) or required by the XML specification. There are four different types of entity references: *character reference*, *internal entity reference*, *external entity reference* and *Parameter Entity Reference*.

Document Type Definition (DTD)

The document type definition, or DTD, gives the definition of the structure of an XML document. DTD is used to decide whether a XML document is valid or invalid. It allows the definition of the document's schema through a definition of the document's entities and elements, the elements' attributes and contents, and the elements' relationships to other elements. A DTD may either be internal or external. Its syntax is based on Extended Backus-Naur Form (EBNF). Since EBNF is not an XML-based notation and has numerous limitations for non-traditional applications, alternate XML schema language is introduced to offer additional functionalities.

```
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="document">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="person" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="person">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string"/>
        <xs:attribute name="punmber" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
              <xs:enumeration value="a1">
              <xs:enumeration value="a2">
              <xs:enumeration value="a3">
              <xs:enumeration value="a4">
            </xs:simpleType>
          </xs:attribute>
          <xs:attribute name="mother" type="xs:hexBinary"/>
          <xs:attribute name="father" use="xs:hexBinary"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 1.2: A Sample XML Schema.

XML Schema

Generally, XML schema is a mechanism of defining structure for XML documents. With the help of an XML schema, a user can give a consistent structure to XML documents. An XML schema defines the elements and attributes that can appear in a document, define the number of child elements and the order in which they should appear, defines whether an element can include text or is empty, defines data types for elements and attributes and so on.

Although DTD is an initial form of describing structure of XML document, it suffers from many drawbacks. DTDs are composed of non-XML syntax which makes it non-extensible. DTDs are unable to represent data type of contents. Although there can be internal or external subsets of DTDs, there can be only a single DTD referenced

by an XML document. XML schema emerged as the solution to be a replacement of DTDs. An XML schema is written in XML syntax making it extensible for future addition, it supports different data types and inheritance of object-oriented concepts. An example of XML schema can be seen in Figure 1.2.

XML has been designed for maximum expressive power, maximum teachability, and maximum ease of implementation. The language is not backward-compatible with existing HTML documents, but documents conforming to the W3C HTML 3.2 specification, generic SGML documents and documents generated from databases can easily be converted to XML. With XML, the Web is turning into a worldwide heterogeneous and distributed database.

1.2 Motivations of the Thesis

XML is a self-describing meta-language and is fast emerging as a dominant standard for Web data exchange among various organizations. Since the use of XML data is increasingly in various communities, efficient database management system is necessary to maintain such data. Several XML storage solutions [AMS99, CACS94, CPHK01, FK99, JKG⁺99, KM00b, LK00, Ray01b] proposed in recent years are mostly based on the conventional databases, such as *Relational Databases* (RDB), *Object-Oriented Databases* (OODB) and *Object-Relational Databases* (ORDB). The main reason is that these databases are matured enough to handle large volume of data and provide robust data management features. However, in practice, these databases have limitations in dealing its rigid schema with XML irregular structure. They require additional transformation to map XML data into their formats and vice-versa. This process is complex and it requires more space when the structure of the document is deep and nested. In addition, users' XML queries must be converted into appropriate patterns understood by underlying query engines of these databases. This is time-consuming and there is performance degradation in data retrieval process.

There give impetus to database researchers to propose new data model that is specifically designed for XML data in order to overcome the deficiencies and limitations of the conventional database systems. The new database system maintains

XML data in its native hierarchical structure that makes storing and retrieving data efficient. It eliminates mapping process, which is a main obstruction of storing XML in the traditional databases, and it reduces execution time and the complexity of such mapping tools. In addition, it provides efficient indexing and query processing as well as document navigation.

Some researchers from both academic and industry have proposed native XML databases in these years [JAKC⁺02, KM00a, QWG⁺96, Sch01, SVC⁺02]. However, they have not addressed the issue of how to efficiently store a large collection of XML documents that are conforming to the same or similar hierarchical structure. Their solutions disregard the common structural properties of the vast number of elements. As a result, the storage consumes additional space to maintain redundant structural information while requiring more I/O accesses when elements or contents with the same path expression are maintained.

In this thesis, we present NextDB, a Native XML Database Management System, to address the above issue of degrading performance in existing native database management systems. NextDB is composed of a set of comprehensive database components that are tailored to deal with the distinct features of XML. It organizes complex documents with similar structures and stores collectively to provide efficient query processing and optimal storage utilization. It uses a query processing engine that can resolve user's native XML queries by the support of path-based and value-based indexes. A transaction management scheme provides atomicity at granularity of transactions and integrity of data in the database. Moreover, a visual user interface offers a rich variety of functions and operations, and provides usability and accessibility to users.

1.3 Contributions of the Thesis

Native XML database is a new and emerging technology in database and Web application areas. Doing research on XML provides opportunities as well as challenges. It is our hope that we contribute knowledge and findings from our work for further researches and improvements. The followings are the major contributions of the

thesis.

- We have proposed and implemented a new storage system, named *Efficient Native XML Storage System*(ENAXS). It is a novel storage strategy specifically designed for storing large and complex multiple XML documents in its native way to overcome the deficiencies of the conventional databases as well as achieve optimal storage utilization and support efficient query processing. In addition, we implemented a path-based indexing scheme that is embedded in ENAXS. By implementing the native storage prototype, we have verified that the fundamental ideas of native XML storage are realisable and have many promising aspects.
- We have proposed an architectural framework that is derived from our empirical experience on designing efficient storage for XML data management, ENAXS, by preserving desirable features and possible constraints.
- We have proposed and implemented a concurrency control protocol for XML data, named *XStamps*, that is designed based on multiversion timestamps concurrency control protocol. Additional features are added to enable flexible control of the isolation level of transactions and allow such transactions to commit early.
- We have implemented a visual interface for native XML database management system named *DB Manager*, which is composed of various database components that provides usability and accessibility to users.
- We have implemented a native XML database management system, named *NextDB*. It is composed of the comprehensive database components that are tailored to deal with the distinct characteristics of XML.

1.4 Thesis Organization

This remainder of this thesis proceeds as follows. After the introduction, Chapter 2 presents the management of XML data in native structure and describes an overview

of the Native XML Database Management System (NextDB) and introductory description of its database components. We discuss the design and implementation of native XML storage system and indexing scheme in Chapter 3. Chapter 4 describes the NextDB query processing. Chapter 5 covers concurrency control protocol for NextDB and visual user interface is presented in Chapter 6. We describe the architectural framework for native XML data management in Chapter 7. Finally, we summarize our contributions and suggest future works in Chapter 8.

1.5 Summary

In this chapter, we first describe an overview of semistructured data and the World Wide Web followed by presenting various markup languages and its strengths and weaknesses. Then, we introduce preliminary knowledge of XML documents and its components to readers in order to provide basic concepts to make reading next chapters more convenient. Motivations and main contributions of this thesis are also provided to give a clue to readers what we are going to introduce. Finally, we cover the overall picture of how this thesis is organized its contents in each chapter.

Chapter 2

Managing XML Data in Native Structure

The rapid proliferation of XML in different application areas results in a tremendously growing number of XML documents as well as research opportunities to design special storage to manage such data. Conventional database systems that require a predefined rigid schema are not completely adequate to maintain flexibly structured XML data. To overcome the inabilities and limitations of traditional databases, different flavors of new generation databases that are specifically designed for XML have been introduced in recent years. NextDB, a Native XML Database Management System for large and complex documents, is one of the such data management systems that is designed based on a novel approach of organizing and clustering XML nodes from multiple documents. To provide an overall picture of the thesis to readers, we briefly introduce NextDB and its components that are comprehensively tailored to deal with special features and distinct characteristics of XML. This chapter covers the introduction to state-of-the-art native XML database systems, the overall architecture of NextDB and its database components including storage structure, indexing schemes, query processor, transaction management scheme and the visual user interface.

2.1 Related Work

XML documents can be stored in a special purpose database system that is called XML based native database systems. Some native XML DBMSs have been proposed in the recent years. In this section, we briefly discuss some popular native database

management systems.

2.1.1 Lore

The *Lore* (The Lightweight Object Repository) is an XML database implemented in Stanford University, that is designed for storing semistructured data in its native format [QWG⁺96]. As an underlying storage, Lore adopts Object Exchange Model (OEM) [PMW95] that is a simple, schema-less and self-describing semistructured data model providing object nesting and identity [MAG⁺97]. The OEM is tailored to handle incompleteness of data and structure as well as type heterogeneity. XML documents in the Lore database are represented as *label-directed graphs*, and elements and attributes are organized as *objects* linked by *edges*, named as *labels*. Document navigation can be preformed by traversing over nodes via edges that are directed by the labels.

An efficient feature of Lore is a DataGuide [GW97] that is a tree representation of structural summary of possible paths in the database that captures the skeleton of XML documents with or without DTDs [Bou]. Unlike the traditional database systems that determine a structure first, followed by allocating data that conforms to the predefined schema, Lore allows importing XML data first followed by deciding the appropriate structure. The resulting dataguide can be effectively used to accelerate tree traversing, formulate the query processing and enable query optimization. Lore is integrated with a query language [AQM⁺97], indexing schemes, multi-user support, and recovery technique to be a complete XML data management system.

2.1.2 Natix

Natix, developed at the University of Mannheim, is a native XML repository designed from scratch that is customized to the requirement of storing and processing XML data [KM00a]. Natix storage adopts a hybrid approach in which a certain level of details of the data are stored in a structured part while finer structures are stored in flat object part of the database [Bou, KM00a]. Natix treats flat objects as atomic records, and allows run-time setting to dynamic threshold value based on the statistics

of size and the structure of the document in order to allocate them into physical pages. Instead of storing individual node in each record, Natix introduces a mechanism that splits an input XML document into sub-trees that are then clustered into compact records limited by the physical page-size.

Natix offers a set of XML documents by means of logical and physical partitions. Applications can specify logical schema to make grouping documents into logical collections that can be used to address data in query executions. On the other hand, physical schema handles storage primitives that are provided by the storage engine to be used to materialize the application's documents on physical disk storage. Natix integrates database components such as multiple indexing schemes [FM01b], algebra for XML construction, query evaluation and optimization [FM01a], query processing engine [MHKM04] and the concurrency control protocol [HKM04] to build a native XML database system.

2.1.3 TIMBER

The researchers of TIMBER project, at the University of Michigan, aim to develop a genuine native XML database engine, designed from scratch. TIMBER uses a bulk algebra called *TAX*, for manipulating sets of trees: each operator on this algebra would take one or more sets of trees as input and produce a set of trees as output [JAKC⁺02]. As an underlying storage management, TIMBER organizes XML documents as a collection of ordered-labelled trees, and stores in Shore [CDF⁺94] that is a hybrid data model of merging object-oriented database and file system. Additional data management features such as an algebra-based query engine and various indexes (tag, value and term) are embedded into the TIMBER storage engine to accelerate query processing and data retrieval in the database.

2.1.4 Tamino

Tamino from Software AG is one of the most popular commercial XML DBMS package that provides XML and Web technologies for platform-independent business-to-business collaboration strategies. It is designed to manage hierarchical structure of

XML in a set-oriented manner using X-Nodes [Sch01]. In Tamino database, multiple documents are grouped into *collections* by the definition of an open content model, and a common schema is defined for each document type in a collection. For each input document, Tamino compares and selects one matched document type and assigns appropriate schema for query processing and data retrieval. Indexing schemes and query processing engine are also integrated to resolve users' queries. In addition, user interface, multi-users accesses and comprehensive transaction managements are also introduced in Tamino to be a competitive and high performance data management platform.

Although the currently proposed native XML data management systems overcome the shortcomings of traditional storage systems, these systems do not have significant strength and improvement in maintaining huge collection of XML documents that have common and similar hierarchical structure. These databases normally retain document trees separately in the physical storage regardless of the schema or DTD that adheres to data. Hence, they require more spaces to keep the redundant structural information and more processing time to abstract schema from data. In addition, more I/O accesses are needed when executing the queries to retrieve a common element or content from multiple documents.

2.2 NextDB: A Native XML Database System

NextDB is an XML-based native database management system for large and complex documents that is specifically designed for XML data to deal with its special features and distinct characteristics. As described, XML data is hierarchically organized in a document. Thus, for a particular element or a node, the path expression and the ancestor-descendant relationship with its parent and child nodes are highly important. Those information can be efficiently employed in data manipulation in order to speed up query processing and data retrieval. We take an account such advantage into design consideration, and tailor NextDB and its database components to meet with it. The general architecture of NextDB and its key components are depicted in Figure 2.1.

The storage design of the NextDB focuses on the scenario that all nodes are

Figure 2.1: The General Architecture of NextDB.

clustered according to their paths, and collectively stored together in physical blocks so that the necessary spaces can be significantly optimized and the user queries can be resolved with less data loading and scanning. The path index is implemented to support direct access to the desired nodes without requiring exhaustive tree traversal and the schema tree is embedded in the index structure in order to deal with regular path queries. The value index is added for the execution of queries with predicate over values, and the comprehensive query processing engine is provided in order to resolve user queries. In addition, the transaction management scheme is introduced to assure the consistency of database state and integrity of data.

2.2.1 Storage & Index Structures

NextDB is implemented based on ENAXS, an efficient storage structure for XML data that organizes multiple documents with common or similar structures, and collectively stores elements and values according to paths attributes [WNL03c]. ENAXS is a path-based storage structure in which element and value nodes are separately stored in each repository. Thus, node operations (insert, update, delete) can be performed simultaneously and independently. With the support of indexes that are crucial parts of NextDB, query execution can be accelerated by eliminating exhaustive path

traversals along the document. Basically, ENAXS is composed of two main storage structures and three indexing components as described below.

- **Value Repository** is a collection of all contents and values from multiple XML documents by allocating each in a record that is a basic building cell of the repository.
- **Node Repository** is a collection of root, element and attribute nodes that are organized as clusters according to its path attributes.
- **Path Index** is a tree-based index structure constructed by the use of possible paths in the database in which all leave nodes of the index tree are pointing to the node groups. For a given path, the index produces a pointer or pointers referencing to a particular node group or groups. **Node Groups**, which is an integrated part of the Path Index, is a collection of groups representing all possible paths in the database. Typically, each group contains all reference pointers to those nodes in the node repository reachable via a specific path.
- **Schema Tree** is a tree representation of structural summary for all XML documents in the database. It is constructed by the use of either DTD or XML schema. It provides the ancestor-descendent relationship to a pair of nodes in order to deal with regular-path queries and document navigation. For a given regular path expression, NextDB traverses along the schema tree to find all possible full-paths that match a given path.
- **Value Index** is built over all content and attribute value nodes to provide an information retrieval style keyword search for a given word or a group of words. For a search, the value index locates value records containing a specific word(s), and returns a set of (DID, PID) that can be used, in turn, to identify a particular document and parent node it belongs to.

The detailed design and implementation of storage management in NextDB is further discussed in Chapter 3.

2.2.2 Query Processor

The query processor implemented in NextDB is an algebra-based query execution engine in which the algebra for data operators are formally defined in group theory as a pair $\langle \text{ObjectSet}, \text{OperatorSet} \rangle$. Typically, the objects that are used to manipulate operators, and the execution results are node sets that can be reconstructed to an XML tree. In order to formulate users' queries, ten basic data operators that are specifically designed for semistructured XML data are allowed in NextDB query processor. These operators basically perform the functionalities of node retrieving, tree traversing and set operations over the nodes maintained in the database.

To eliminate repeated and additional query execution steps, and to boost the performance of query processing, NextDB also introduces a query optimization scheme that engages efficient use of specially designed index and storage structures to optimize the execution of different query types including exact match, order access, regular path and path traversal.

NextDB query processor is made up of the following main components:

- **Parser** is a process that validates users' query statement synthetically.
- **Tree Builder** is a process that builds a query tree that is a tree representation of an input query statement. Upon completion of validation, each query is split into sub-statements that are subsequently used to construct a sub-tree that is finally linked with others to form a query tree.
- **Result Generator** is also a process that traverses the query tree, fetches an operator, and then executes using the operand(s) provided. The execution result is then reformatted into XML format that is returned back to user.

2.2.3 Transaction Management

Interactions among transactions may cause inconsistency in the database even though individual transaction performs correctly and there is no system failure. Thus, the order among different transactions must be regulated using rules [MUW02]. Concurrency control is a process that ensures to preserve consistency of the database state

when transactions are simultaneously executed [MUW02]. The concurrency control protocol in the NextDB's transaction management is named *XStamps* [WNLL03] that is designed based on the concept of multiversion timestamp ordering protocol (MTOS) [Ree78b] with additional improvements.

The implementation of the NextDB transaction management system involves two main parts: *transaction manager* and *resource manager*.

- ***Transaction Manager*** ensures that all transactions are executed correctly. It determines whether the transaction is granted, nullified, delayed or aborted based on the access rules by preserving the Atomicity, Consistency, Isolation and Durability (ACID) properties of the transactions and consistency of the database.
- ***Resource Manager*** manages recently uploaded data and prevents direct accesses to the database. It also interacts with query processor to provide node retrieval over the database as well as log manager and recovery manager that are the components of recovery management system in NextDB.

The detailed architecture, access rules, design and implementation of transaction management in NextDB is further discussed in Chapter 5.

For a database system, maintaining the integrity of data is crucial. Data must be protected against any type of system failure and a recovery system has to be provided to support the resilience of data. In order to provide this function, we design a recovery system in NextDB. Among different recovery techniques, NextDB uses *redo-logging*, a simple and efficient strategy for logging and recovering data from failure. This strategy mainly depends on a *log* that keeps a history of changes in the database.

We design a log manager and a recovery manager in the NextDB working together with the transaction manager for supporting the goal of data integrity in the database. We deploy a *checkpoint* technique to determine the time when a log file is copied to disk. If a system failure occurs, the recovery manager deals with the resource manager to obtain the log file from disk, examines it and uses it to reconstruct the database

to a consistent state.

2.2.4 Visualizing XML data

A user interface is a mediator between user and the database system. It supports users to communicate with the database interactively, to make changes conveniently and to get the required data easily. In order to provide this function, a visual user interface, named *DB Manager* is implemented for NextDB [WRNL03]. The implementation of DB manager contains two parts: *data management interface* and *interactive query interface*.

- ***Data Management Interface*** is intended for the users who want to do database administrations and data modification. This interface allow users to do a new database creation, a new XML document and DTD/Schema creation, and a user query file that can be repeatedly used to submit a query or a group of queries to the database.
- ***Interactive Query Interface*** allows users for viewing DTD and XML data, importing and exporting XML document to and from the NextDB database, and interactive querying and updating of existing data.

For detailed discussion of architecture, design and the implementation of the NextDB visual user interface, reader is referred to Chapter 6.

2.3 Summary

In this chapter, we presented the overall architecture of the NextDB, and a brief introduction to its database components. NextDB is composed of database components including storage structure, indexing scheme, query processor, transaction management system and the visual user interface that are tailored to the needs of XML data management. These components are comprehensively built and integrated into database to support efficient storage and retrieval of XML data. This chapter provides the introductory description of each component, its sub-components and corresponding

functionalities in order to provide fundamental understanding to the readers on the entire database management system. The subsequent chapters elaborate the detailed architecture, design and implementation of each database components.

Chapter 3

The Native XML Storage System

Within a few decades, the Web has grown incredibly to become the main information interchange among various organizations. Many applications have been producing and consuming semistructured data that contains irregularities and evolves rapidly making the use of predefined rigid schemas infeasible. XML is emerging as a dominant standard for representing and exchanging semistructured data among applications over the Web. With the tremendous growth of XML data, an efficient storage system is required to manage them.

Several XML storage solutions [AMS99, CACS94, CPHK01, FK99, JKG⁺99, KM00b, LK00, Ray01b] proposed in recent years are based on file systems and conventional databases, such as *relational* (RDB) and *object-oriented* (OODB). The main reason might be that these conventional databases are matured enough to handle large volumes of data and provide robust data management features. In practice, they have limitations in dealing with the rigid schema of XML structures. These databases require an additional transformation process to map XML data into their formats and vice-versa. This process is complex and requires more space when the document structure is deep and nested. In addition, they have to convert XML queries into appropriate patterns that are understood by underlying query engines. It is time-consuming and degrading the performance in data retrieval.

Due to the above deficiencies and limitations, a new data model that is specifically designed for XML is emerging recently. This system is able to store XML in its native hierarchical structure and eliminates the transformation processes. It provides

faster query processing and efficient document navigation. Currently proposed solutions [JAKC⁺02, KM00a, QWG⁺96, Sch01] are designed to provide such strengths. However, they have not addressed how to efficiently store large collection of documents that contain data with similar hierarchical structure. They omitted common structural properties of elements, and as a result, the storage consumes additional space to maintain redundant structural information and requires more I/O accesses when all elements or contents (values) with the same path expression from multiple documents are retrieved.

In this chapter, we propose ENAXS, a novel approach of managing XML documents in native storage structure that addresses the above issue of performance degradation in existing native database management systems. ENAXS deals with the scenario that all XML nodes from multiple complex documents are collectively organized according to the similarity in structures, and clustered together into groups of physical records to provide efficient query processing and optimal storage utilization. In addition, multiple indexing schemes including path index, schema tree and value index are integrated into ENAXS to resolve various types of XML queries and to boost query processing performance.

3.1 Related Work

Since XML is developing very fast and emerging as a *de facto* standard for information exchange. The issue of managing XML documents becomes popular to address in database community. Some researchers have highlighted various issues, limitations and opportunities of handling XML data, and proposed appropriate solutions to address it, but no particular solution covers all aspects, and there still need much rooms to be considered. At a ground-level, the solutions can be simply classified into three categories: traditional file systems, conventional database systems, and native XML database systems [WNL03a].

3.1.1 File-Based Systems

File based system is the simplest and the most traditional storage system. It facilitates a store repository for either rigidly-structured or flexibly-structured XML documents. These documents can be stored as a file regardless of its structure. In a flat file system, a parser is always required to enable access to the structure of XML documents and to evaluate queries [ACM93]. Since the entire XML document is stored in a single file, the data can be readily accessed by a text editor or other XML tools. It is easily accessed from any programming language and is readily accessible by all XML parsers. An XML flat file system provides hierarchical access to data through the directory structure of a file system, and then through the tags (elements) structure in XML documents.

Usually, this approach is simple and capable of managing large quantities of static data in the file-based format by organizing each XML document per file. For specialized applications, this type of storage is useful to consider as an alternative format that allows additional features such as binary encoding and specialized character set to be able to speed up parsing efficiency with specialized parsers.

In addition, Significant strength of the flat file approach are of simplicity, ease of use, smallness in size and less resource utilization. Flat files are easy to build, maintain and access by various tools. A significant drawback is that this type of storage does not support streamlined indexes that, in turn, produces inefficiency in query processing and data retrieval. It is not scalable enough to handle a large volume of data and difficult to implement data management features such as security, data locking, multi-user support, transaction management and recovery.

3.1.2 Conventional Database Systems

This approach makes use of conventional database systems, such as RDBMS, OODBMS and ORDBMS, as a repository for XML data and employs a *mapping* tool to transform XML into structured data format that can be maintained in underlying data models, and vice-versa. Such mapping techniques inherit strengths and advantages of conventional databases that are ready-to-use for efficient data processing. Based on

the underlying data model, the storage techniques can be organized into two groups: *table-based* and *object-based systems*.

Table-Based Systems

Although there still have a lot of mismatches between the distinct characteristics of XML data and flat structure of relational model, the interests in the use of table-based systems are still significantly increasing due to existing attributes of those systems such as scalability, robustness, reliability and efficient data management features. In addition, matured indexing techniques and highly optimized relational query processors are also prominent resources that can be re-used in XML data processing [BFRS02]. The major obstruction of storing XML data into table-based systems is that the schema is often proprietary in these systems, and input data is typically flatten into tabular forms and normalized into many relations which make direct importing of XML data infeasible [FMST01].

To overcome this limitation, researchers from academic and commercial sectors have been putting their efforts in this area over the past years, and as a result, a variety of mapping techniques and transformation tools, named *middlewares*, are widely available and applicable to various relational databases. We categorize such storage strategies into three main groups according to their data mapping approaches.

Schema-Dependent Approach: Both Document Type Definitions (DTDs) and XML Schema represent models for describing the structure and semantics of XML documents, and defining constraints that can be used to validate data in a single or multiple documents. Without DTDs or Schema, an XML document will never reach its full potential since a well-formed tagged document is typically not very useful without some agreement among inter-operation applications by means of what the tags represent [BBC⁺98]. Since DTDs originated from the SGML specification that was designed for a more document-centric model, it did not have the specification for the use of complex datatyping definitions. On the other hand, XML Schema, a new and more powerful alternative to DTDs, provides better support of modular design and more systematic datatyping capability for elements and attributes.

Early mapping techniques [FTS00, JKG⁺99, ZMLR01] had mostly concentrated on the use of DTDs to transform XML into tabular form of data. DTD is mandatory in these transformations and it is converted to and maintained in tuples of relational tables to identify the schema of input XML documents [JKG⁺99]. One of the common problems encountered in this approach is how to deal with compound or complex DTDs that are difficult to generate relational schemas in order to capture such complexity. In general, DTD may consist of nested specification of elements, regular expressions and entities and entity references that make the data fragmented and the transformation sophisticated. In addition, there could be a situation where multiple DTDs are representing a single XML document, imposing ambiguity in transformation.

To simplify arbitrary DTD while keeping the effectiveness of queries over documents that are conforming to DTD, different types of transformations and inlining techniques were proposed [JKG⁺99, LC01]. Those techniques simplify the complexity of binary operators in DTD by converting a nested definition into a flat representation and reducing many unary operators to a single unary operator while grouping sub-elements having the same element name. At the same time, inlining techniques eliminate the fragmentation of elements while sharing repeatedly represented element nodes across multiple documents and preserving semantic constraints of DTDs.

An alternative technique to inlining is *grouping* where elements and attributes containing in DTD are classified into different groups [ZMLR01]. These groups are later converted into two relational tables; *item* and *attribute* tables, by referencing its properties in the DTD. The individual member of each group then becomes a tuple in an associated table, and an additional table, called *nesting table*, is also created for representing hierarchical and set-containment relationships between items and attributes. These three tables compose a relational schema for later mapping of input XML documents which are conformed to the DTD.

Most techniques in this approach typically propose a pair of mappings: from external XML source to relational tables where data can be manipulated or queried with the support of the structured query language, and vice-versa. The manipulated re-

sults are transformed back to XML format that can be published over the Internet or exchanged among user applications. Other middlewares [BCFR02, FMS01, FMST01, SSB⁺00] are also DTD-oriented which are similarly designed to extract data from relational database and model into XML format for publishing purpose. Those techniques mostly transform the necessary columns of relational tables into predefined XML format in which main columns become the elements and the primary key, and foreign key becomes attributes of related elements.

Since DTD describes the structure of document, it is not completely intuitive as it does not imply the data type of the XML contents. All contents are simply expressed as texts and the applications themselves must change the data types for mathematical calculations. In order to fulfil this requirement, W3C announced XML schema specifications [Gro01]. XML-Schema aims to be an eventual replacement of DTD to provide a richer grammar for prescribing the structure of elements by comprising more expressive and usable features to be used by a wider variety of applications. It brings many novel mechanisms such as inclusion and derivation, inheritance for attributes and elements, user defined data types, XML-like syntax for describing documents, and types of elements through XML-data [Kle01, LC00]. The announcement of XML schema draws database researchers' attention towards the XML schema-driven approach.

The general process of this approach is to produce the mapping framework to transform XML schema into relational database schema and vice-versa. This transformation basically involves the mapping of attributes, attribute groups, simple and complex types, constraints and inheritance types [VV01]. The techniques are those based on this concept to bridge the gap between XML schema and relational schema [LMCC01, LMCC02, VVP⁺01].

Similar to DTD, there could be multiple XML schemas, which are representing a single document, that may in turn produce different relational schemas and fragmented relational tables in transformation. In order to address this problem, the statistics-based technique of classifying elements to type names is proposed [BFRS02]. This approach analyzes and collects statistics information from input XML schema

and produces an appropriate fixed-mapping technique that intends to optimize the data fragmentation.

The aforementioned schema-dependent mapping approaches are more conventional and straightforward as it employs the DTD or schema as a guideline for transformation. A significant limitation is that DTD or schema is mandatory in this case and documents without metadata, such as well-formed XML documents, cannot work with it. In addition, this approach typically produces fixed mapping techniques that are not flexible enough to cope with arbitrary input XML documents. A commercial database [LK00] provides a more flexible approach by allowing the developer to specify customized storage mapping. However, it still needs the developer to be completely proficient in both XML and RDBMS. In practice, it might be difficult to determine the best mapping for a complex application [BFRS02]. An alternative approach to overcome this difficulty is schema-free transformation that is independent of the existence of metadata of the document.

Schema-Independent Approach: This approach is more general and relevant to the mapping of metadata-less well-formed documents that comply with rules laid out in the XML specification. There are fundamentally two steps towards this transformation: Modelling input XML documents to accumulate data structure and generating an appropriate relational schema. A useful technique for modelling structure of XML documents is *validation* and *structure construction* through XML parser such as DOM [Groa] or SAX [Meg]. DOM is a commonly-used tree based object model that organizes XML elements into a series of DOM object nodes while SAX uses event-driven approach of producing a sequence of events to provide an opportunity to create own custom object model. Metacat [BJBH01], MXM [AYS02], and XTables [FKS⁺02] work in such way of schema transformation to derive appropriate relational schema which is normally maintained under multiple tables. The advantage of this approach is significant as it is more generic and works with both schema-aware and schema-less XML documents but some degree of user's intervention is still necessary in most solutions to specify pre-defined rules and heuristics for schema generation and doc-

ument shredding. On the other hand, Adaptable and Adjustable Mapping (AAM) technique was proposed by Wang *et al.* [WLD02], and more flexible and adaptable mapping concepts was proposed by Tatarinov [TVB⁺02]. However, these techniques are general and still have limitations to deal with specific applications.

A more innovative solution used in STORED (Semistructured TO Relational Data) is the use of an extended data mining technique that analyzes and discovers the schema from input XML data, identifies highly supported tree patterns for storage, and constructs an aggressive mapping technique [AMS99]. This technique is more efficient in terms of schema optimization. However, it is more suitable for static XML documents that structure is rarely altered. For those particular applications and frequently changed documents, user defined mapping approaches have been proposed.

User-Defined Schema-Dependent Approach: X-Rays [KKR00], Mars [DT03] and some commercial relational and object-relational databases systems such as Oracle [LK00], MS-SQL [Ray01b], and IBM DB2 XML Extender use this approach to provide flexible transformation concept that allows user to define its own mapping of XML data to relational tables by making use of special-purpose queries or a declarative interface. In such approach, user has to specify target relational schema that captures the structure of XML documents and decomposes document trees into tabular forms. Although fundamental mapping concept that used in these solutions is the same, mapping strategies are different like transforming elements and attributes using XSLT [Gro99b] queries [LK00], parsing document into DOM trees and writing XPath expressions to specify the mapping technique [Ray01b], and providing a declarative interface that makes user easy to identify XML documents and specify relational schema with encoding mapping information ¹.

The advantage of this approach is that users are able to customize a mapping strategy to fit a particular application or requirements that they are going to focus. However, there is the limitation that user's effort is necessary to learn both XML and relational concepts, additional query languages and programming to write data-

¹<http://www-4.ibm.com/software/data/db2/extenders/>

loader programs. Since all imported data are simply stored in either relation objects or CLOB (Character Large Object), they can support complex querying and full-text search over it.

User-defined schema approach is more flexible than fixed mapping because user can define alternative mapping schemes and choose the one that is most relevant to the application and works best with input documents. In addition, user may capture all structural information using foreign-primary keys relationships which is substantially useful in relational architecture. However, it also imposes a drawback that requires user to be completely proficient in quite distinct technologies: XML and RDBMS. It might be difficult, even for an expert, to decide what is a good mapping and how efficient it is in terms of performance and adequacy.

As discussed details in above sections, maintaining XML in relational databases has the advantages that one may reuse robust and proven data maintenance features of relational architecture to interoperate and consolidate with the existing relational data. However, there is a significant limitation that the representation of the data is limited to relations. In addition, there still exist a number of mismatches between what the relation and XML can represent since XML has more inherent flexibility than the relation. Mapping ordered XML documents might impose several issues and modifying XML schema may cost a significant effort in updating relational schema. For data retrieval, all XML queries need to be converted to equivalent SQL queries that can be executed in relational engine. Moreover, imported complex documents might be dispersed over multiple tables that demand expensive join operations to be executed while searching and retrieving the required data. As an alternative to overcome this situation, a mapping approach to object-based database systems is introduced.

Object-Based Systems

In its nature, the structure of XML is tree-like; it can be easily mocked up into a hierarchical-based model. In addition, object-based data models support rich type systems and complex objects that can be structured as labelled-directed graphs.

Object-based approaches, such as [CACS94, CPHK01] and Excelon² exploit this opportunity to model XML into a tree of objects that can be easily managed by object-oriented database systems.

The main advantage of this approach is that a mapping technique can be derived from the efficient use of inheritance which is a key concept in both object-oriented paradigms and XML architecture. Elements and attributes in an XML tree, in this approach, can be converted into objects, classes and set-valued attributes that are linked by direct pointers to form an object tree in OODB model.

Similar to table-based solutions, this approach also introduces a middleware layer to perform mapping of XML data into objects, and vice-versa. In addition, there are some XML structures which are not easily mapped to object structure, such as mixed-contents and its alternatives. Such structures result in large database schemas and sparsely populated databases. Moreover, the object-model-based representation requires large amount of system resources and is not able to scale to the level of RDBMS. The current object-oriented database products are not mature enough to handle large volume of data and query engines are not adequate to process complex queries on large databases which demonstrate object-based solution is not a good choice for XML storage.

3.1.3 XML-Based Native Systems

Alternatively, XML documents can be stored in a special-purposed state-of-the-art database system which we called an XML-based native database system. This database is specifically designed for XML. It aims to overcome the shortages of traditional databases while achieving performance improvement in data storage and retrieval. It should eliminate the mapping layer of middleware and additional query transformation process. It also intends to simplify data importing from XML documents and to improve storage allocation and utilization. Another objective includes optimizing data fragmentation while cutting down scanning time for document navigation and data retrieval. These special databases can be classified into two groups based on

²<http://www.odi.com/excelon>

their approaches.

Pure Native Approach

Native databases under this approach are particularly designed for XML data management and are tailored to cope with features and distinct characteristics of XML. Both physical and logical storage structures of these databases are modelled accordingly to deal with the hierarchical nature of XML and customized to provide XML specific tasks such as XML query processing, traversing, order management, etc. Since each database has its own proprietary technique and unique architecture, it is the best way to approach individually.

Natix - Native repository system, called *Natix*, uses a hybrid model of file-based and table-based storage system in which a certain level of detail of the data is stored in its structured part of database whilst the rest is in the flat object part [KM00a]. Instead of keeping each element node in an individual record, Natix divides XML tree into subtrees, and stores each in a single record that is less than a disk page in size. Large trees are decomposed using a splitting algorithm and distributed over multiple pages linked each other using proxy objects. Rearranging proxy objects, along with its respective subtrees, reconstructs the original tree that can be used to traverse along for data retrieval.

Tamino - Tamino from Software AG is designed to manage hierarchical structure of XML in a set-oriented manner using X-Nodes [Sch01]. In the Tamino database, multiple documents are grouped into *collections* by the definition of an open content model, and a common schema can be defined for each document type in a collection. For each input document, Tamino compares and selects one matched document type and assigns appropriate schema for query processing and retrieval.

Adapted Approach

The physical storage for this type of databases are adapted from the existing storage engines, and logical models are designed in order to fit XML data.

Lore - As an underlying storage, the Lightweight Object Repository System (Lore) adopts an object-based data model *OEM*, which is a simple, schema-less and self-describing model providing object nesting and identity [MAG⁺97]. Documents in this database are represented as a label-directed graph and elements are organized as objects linked by labels. Each object has a unique object identifier in this storage and DataGuides [GW97] is used as a database schema to formulate user's queries and enable query optimization.

TIMBER - Similarly, TIMBER organizes XML documents as a collection of ordered-labelled trees manipulated by the use of bulk algebra and stores in *Shore* which is a hybrid data model of merging object-oriented database and file system [JAKC⁺02]. TIMBER also adds additional components such as an algebra-based query engine and various indexes (tag, value and term) to the storage to accelerate query processing in the database.

DB4XML - It is a main memory-resident XML database engine that maps element nodes in DOM [Groa] tree into objects that can be stored in the database. The Patricia Trie-based path index and T-tree-based value index are embedded in DB4XML to provide faster data retrieval. Database components such as concurrency control and recovery management are also implemented to manage transactions and to deal with system failure in the database.

The significant advantage of this approach is that XML documents can be stored in the database in its native form without needing additional middleware or mapping technique. Updating and deleting are straightforward and can be performed easily and efficiently as data is in native structure. Document navigation, traversing and data retrieval can avoid expensive join operations among distributed tables, and queries using XML query language can be resolved in its own query engine, rather than converting to equivalent SQL statements which degrade performance. In addition, new XML features like namespace and XSLT can be embedded into database. However, current state-of-the-art native databases do not address the issue of how multiple XML documents with similar hierarchical structure can be collectively or-

ganized in the database to provide optimal storage utilization as well as faster query processing and data retrieval. We resolve this issue in ENAXS.

3.1.4 Indexing Schemes for XML Data

Indexes are parts of the storage system in XML to speed up the query processing. *Lore* supports multiple indexing schemes such as *Value Index (V-Index)*, *Text Index (T-Index)*, *Link Index (L-index)*, and *Path Index (P-Index)* [MAG⁺97]. The V-index and T-index in *Lore* identify objects that have specific values while L-index and P-index are used to efficiently traverse the database graph. *Data Guide* is a summary of all paths that represents a structural summary to navigate the semistructured graph [GW97].

Another XML storage, *ToXin*, provides two types of index structure: *Value Index* and *Path Index*, that resolve users queries with regular path expressions [Riz01, RM01]. A different indexing approach proposed in *Index Fabric* applies a prefix encoding mechanism that encodes each path of a XML element along with value as a string [CNM⁺01]. The resultant encoded strings together with values are indexed as keywords in the well-known string index, Patricia Tries [Don88]. On the other hand, *APEX* proposes an adaptive path index that is a structural summary to deal with the regular path expressions by employing graph and hash structures [CMS02]. *APEX* applies sequential pattern mining algorithm to summarize paths that appear frequently in queries. Thus, it shorten query processing time significantly.

3.2 ENAXS

In the real world scenario, the practice of XML data exchange among various business organizations rely on bilateral agreement between involving parties that predefine common schema or structure of exchanged data. Based on mutually agreed schema, XML data is prepared, transferred, collected and processed in order to make accessible by internal applications. It makes sense that both parties understand the format prior to exchange data between them, and prepare parsing and execution tools in advance for further processing. This scenario makes it obvious that multiple XML documents

could have similar structures or even conform to a common DTD or XML schema. ENAXS brings this issues into storage design consideration and takes the advantage of maintaining such data together as a group of nodes, to contribute improvement in storage and efficiency in data retrieval.

ENAXS focuses on the scenario that all nodes are collectively clustered according to their similarities in path expressions, and stored together in physical blocks. Thus, the necessary space can be significantly optimized and the user queries can be resolved with less data loading and scanning effort. Another aim is to provide supporting indexing schemes such as path index, value index and schema tree in order to enable direct access to the desired nodes without requiring exhaustive tree traversal, and resolving regular path queries that are very common in XML.

3.2.1 Storage Structure

ENAXS aims at providing storage structure that facilitates efficient storage, updating and retrieval of XML data as well as supports optimal storage utilization and faster query execution. At the design consideration stage, we mainly focus on the following design goals:

- To define optimal block (page) size to support faster retrieval
- To design well-structured physical record formats that accommodate XML nodes
- To intuitively flatten XML logical tree into physical storage
- To facilitate direct accessing to a node in physical storage whenever necessary
- To handle both long and short scalar values effectively
- To resolve various types of user queries including regular path and predicate over values
- To provide efficient updating to a node or a group of nodes
- To design a node clustering approach to accelerate data storing and retrieval

In ENAXS, an XML document is logically viewed as a tree composed of many nodes linking by edges. All tags from the original document are treated as element nodes while contents are named as value nodes. Root and element nodes shape the structure of the tree while value nodes maintain the majority of the information in the document. Before going deeper into the detailed structure, we define some of the terms widely used in the ENAXS architecture.

Node Let n be a node and $\mathcal{N}$ be a domain of all nodes. A node n represents a tag t or a value v . Hereafter for any node n , $n \in \mathcal{N}$, and consequently for any set of nodes N , $N \subseteq \mathcal{N}$.

Node Set A set of nodes V comprising three types of tree nodes: the root node n_0 , the set of internal nodes N_I and the set of external (or leaf) nodes N_E . Hence, for any given node in the document tree, n , $n \in V = \{n_0\} \cup N_I \cup N_E$.

Let $\mathcal{N}_I$ represents a domain of internal nodes, thus $\mathcal{N}_I \subseteq \mathcal{N}$. The function $parent(n)$ returns a singleton set containing the node representing the enclosing parent element for an internal node $n \in \mathcal{N}_I$. Let $\mathcal{N}_E$ represents a domain of external nodes; thus, $\mathcal{N}_E \subseteq \mathcal{N}$. A root node n_0 represents the first, outermost lexical element; thus, $parent(n_0) = \emptyset$. The domain of root nodes is denoted $\mathcal{N}_0 \subseteq \mathcal{N}$. Thus, $\mathcal{N} = \mathcal{N}_I \cup \mathcal{N}_E \cup \mathcal{N}_0$.

Internal Nodes An internal node $n \in N_I$ is an ordered triple $n = \langle t, A, C \rangle$ where $t \in \mathcal{T}$ is the markup tag of a corresponding XML element ε , A is a set of attribute-value pairs of the form $\langle a_i, v_i \rangle$ ($i \geq 0$) representing the attributes of ε , and C is an ordered collection of child nodes of n such that $C \in (N_I \cup N_E) \setminus \{n\}$. Members of C represent the subelements and parsed character data portions of ε in the order in which they appear in the document. The functions $label(n)$, $attributes(n)$ and $children(n)$ defined in ENAXS, return a tag, an attribute set and a child respectively of an internal node n . The function $id(n)$ returns a unique identifier associated with node n .

A node with one and only one internal child node is defined as a *complex node*,

i.e., $n \mid \forall k \in \text{child}(n), k \in \mathcal{N}_I$. A node that has both types of child nodes (internal and external) is called a *mixed node*. The function $\text{value}(n)$ returns the character data string v for the external node.

External Nodes An external or a leaf node in the document tree contains a value or a character data that appears within a pair of markup tags. The parent of such a node is the element delimited by the corresponding tags. An external node $n \in N_E$ is characterized by a string $v \in \mathcal{C}$ representing a block of character data without any intervening markup.

Node Repository A group of internal node sets which is defined as, $\mathcal{N}_I = \bigcup_{i=1}^k N_I^i$ where N_I^i is a set of internal nodes for XML document D_i .

Value Repository A group of external node sets which is defined as $\mathcal{N}_E = \bigcup_{i=1}^k N_E^i$ where N_E^i is a set of external nodes for XML document D_i .

XML Document An XML document D is represented by an ordered, rooted, labelled tree given as $D = \langle V, E \rangle$ where V is the set of nodes and E is the set of labelled edges.

XML Database An XML database is a collection of XML documents. Thus, for a given set of XML documents $D_1, D_2, \dots, D_k$, an XML database DB is defined as $DB = \bigcup_{i=1}^k (\text{root}(D_i) \cup \mathcal{N}_I(D_i) \cup \mathcal{N}_E(D_i))$, where a set of internal nodes is defined as, $\mathcal{N}_I = \bigcup_{i=1}^k N_I^i$ and external nodes as $\mathcal{N}_E = \bigcup_{i=1}^k N_E^i$ for the document D_i .

In ENAXS, each XML document is assigned a document identifier (DID), which is unique within the entire storage, and each internal node is assigned a node identifier (NID), that is unique within a document. Thus, a pair $\langle \text{DocumentID}, \text{NodeID} \rangle$ can be used to identify a specific node in a particular document within the storage. For a single document, two nodes can be said the same only if their identifiers are identical. For multiple documents, two nodes are the same when both pair of DID and NID are

Figure 3.1: The General Storage Architecture of ENAXS.

identical.

ENAXS is a path-based storage structure in which internal and external nodes are respectively stored in each repository so that node operations such as inserting, updating, and deletion, can be performed simultaneously and independently at both ends in each repository. As shown in Figure 3.1, ENAXS is composed of two main structures: *node repository* and *value repository*.

In ENAXS, all external nodes are stored in the *Value Repository* while all root nodes and internal nodes are stored in the *Node Repository*. This type of storage structure has the advantage that the traversal along XML tree can be performed in the node repository in parallel to the value searching in the value repository.

Value Repository

Basically, value repository is a collection of external nodes in which each node is allocated in each physical record that is the basic building cell of the repository. The detailed format of an external node record is illustrated in Figure 3.2.

The external node record is composed of four fields: Document Identifier(DID), Parent node Identifier(PID), Parent node block Pointer(PPtr) and value.

- *DID* - It represents an unique identifier of a particular document that the external node belongs to. DID is useful to identify a unique document in the database.
- *PID* - It is a parent node ID that is associated with the parent node of the

Document ID	Parent Node ID	Parent Node Block Pointer	Value
-------------	----------------	---------------------------	-------

Figure 3.2: External Node Record Format

external node. Usually, all parent nodes are internal nodes.

- *PPtr* - It is a parent node block pointer that is referencing to the block where the parent node of the external node is stored. Using the triple $\langle \text{DocumentID}, \text{NodeID}, \text{ParentNodeBlockPointer} \rangle$ allows traversing bottom-up along the XML tree and locating a unique internal node in the Node Repository.
- *Value* - It is a scalar value or content of the external node. It can be of any length with any data type.

The size of an external node record may vary with respect to the length of its value, in this case, the number of records fitted in a block is indeterminate a priori. Thus, a tag is prefixed to each record to support easy identification of the record size and fast accessibility to consecutive records in the same block.

Typically, the size of a record is limited by the size of a physical block. For the case of a node with long content, we employ the *splitting* approach to make the record into parts; the first part is allocated in the current block and the rest is populated into successive blocks. A pointer to a new block is appended to the first block. Thus, documents with long contents like novels, articles and plays can be accommodated and retrieved efficiently. In ENAXS, we define the following operators to support node operations in the value repository.

Node Insertion Inserting an external node into the value repository is straightforward. It only appends a new record of the node to the last block of the value repository. If the record cannot fit in a block, the split procedure is applied to span the record over multiple blocks.

Node Updating Updating is not simple as insertion, and *reallocation procedure* is needed to be executed for rearranging records within a block. If the updated record

Figure 3.3: The Internal Node Record Format.

is shorter than the old record, we replace the old one with the updated one, and reallocate all subsequent records within a block to move up. If the updated one is longer, reallocation procedure pushes all subsequent records down to get enough space to allocate updated one. This may need to acquire a new block to accommodate exceeded record(s) after reallocation is completed.

Node Deletion Similarly, deletion of a node releases the space occupied by the record, and again all subsequent records in a block are reallocated to move up. The space available at the end of the block can now be used for future allocation of records.

Node Repository

Node repository is a collection of all internal nodes that are collectively organized and clustered according to their similarity in paths. Each node is referenced by a pointer in the path index, so that, a particular node can be directly reached by the use of associated path expression as a search key in the path index. Similar to the value repository, each node in the node repository is represented as a node record which consists of three parts. The detailed format of an internal node record is depicted in Figure 3.3.

A node record is composed of three main parts: *Node Info*, *Direct Block* and *Indirect Block*.

Node Info As the name implied, *node info* maintains the information of the international node such as DID, NID, Node Order, PID, PPtr, Attribute Flag and Value Pointer.

- *DID* - It is a unique identifier of a document that the internal node belongs to.
- *NID* - It is a unique identifier of the internal node.
- *Node Order* - It reflects the position that the internal node appears among its siblings in the document. This field is necessary to evaluate ordered access queries, where the order information is exploited.
- *PID* - It refers to parent node's identifier of the internal node. It represents the parent-child relationship of the node in the tree. Each internal node has a parent node except root. Parent ID is set to a negative value to indicate that the node is the root.
- *PPtr* - It is a reference to the block where parent node of the internal node resides. Combination of this pointer and PID can determine the exact physical location where the parent node is allocated.
- *Direct Pointer* - It is a reference pointing to a direct block.
- *Attribute Flag* - An attribute node is usually composed of two parts: *name* and *value*. ENAXS treats an attribute name and attribute value as an internal node and external node respectively. Attribute flag is used to distinguish whether it is an element node or attribute.
- *Value Pointer* - It points to the block where the scalar value of the internal node is stored. This pointer always references to a particular block of the value repository. For all internal node records without having values, the pointers are usually set to NULL.

A referral, named *direct pointer*, is added to address to the direct block where CID and CPtrs of the internal node are stored. Each pointer there is referencing back to the node repository where a physical record of each child node is allocated.

- *CIDs* - It maintains identifiers of all child nodes of the internal node. CIDs reserve the tree structure of the original XML document by maintaining the parent-child relationship among nodes.
- *CPtrs* - It is a group of references to the block(s) where the child nodes are stored. Same as the parent node pointer, all child nodes can be accessed from the repository by using the pair of CIDs and CPtrs.
- *Indirect Pointer* - It is a reference pointing to an indirect block where additional child nodes are allocated.

The detailed format of *Node Info* is depicted in Figure 3.3(a) and the entire internal node structure including direct and indirect blocks is shown in Figure 3.3(b).

Direct Block and Indirect Block A direct block contains a specific number of CIDs and CPtrs that belong to all child nodes of the internal node. ENAXS provides a system parameter, called *node set threshold* denoted by α . This parameter defines the maximum number of child nodes that can be allocated in a direct block of the node repository. For an internal node having more number of child nodes than α , the additional ones are then allocated in an *indirect block* that is linked by *indirect pointer*. This mechanism helps the database administrator to perform database tuning to optimize storage space and speeds up the data lookup during retrieval.

The value of α is initially set to 10 by default, meaning that, the first ten child nodes can be stored in the direct block, and the remaining nodes, if any, are allocated in the indirect block. ENAXS provides the flexibility that this value can be fine-tuned by database administrator to achieve the optimal storage utilization and data retrieval.

To perform data operations and processing in internal node records, the following operators are defined in ENAXS.

Node Insertion Inserting a new node into the node repository involves two activities: allocating a new record and inserting a CID and PPtr to its parent node record.

- If the direct pointer of its parent node record is NULL, the reference to the new node is stored in a direct block, and the direct pointer is set to reference the block where the child pointers are allocated.
- If the number of existing child nodes belong to its parent node is less than the value of α , a new pointer pointing to the new record is added to the direct block.
- If the number of existing child nodes is equal to or exceed α , a new child record is allocated in the indirect block and a pointer is appended in the indirect block.

Node Updating Updating an internal node involves modifying a particular field of a node record. The combination of insertion and deletion procedures may update a series of nodes. It may need to update an index if an internal node name is modified.

Node Deletion Deleting an internal node requires an update to both the value repository and node repository, including direct blocks as well as indirect blocks, if any. When an internal node is deleted, the whole subtree rooted from that node is also deleted. The deletion uses a bottom-up approach in which all external nodes are removed first from the value repository, and then internal nodes of the subtree, its CIDs and child pointers are also removed from the node repository accordingly. Finally, the pointer to the root node of the subtree is deleted from its parent node record by simply setting the direct pointer to NULL. The indirect blocks are released if there is no node in it.

The design of locating child nodes into direct and indirect blocks in ENAXS reduces the storage requirement since a block may retain multiple child nodes from multiple internal nodes. It significantly eliminates the storage overhead and cuts down expensive I/O accesses. In addition, the basic database operations such as insertion, updating and deletion can be done easily and efficiently without spending much execution time. The value of α is flexible for customization, so that an administrator can monitor and tune the database accordingly to the structure of input XML document to reach optimal storage utilization and query execution.

Figure 3.4: The Index Structure of ENAXS

3.2.2 Index Structure

Indexes are the crucial components of ENAXS. With the support of indexes, ENAXS can speed up document navigation and query execution by eliminating exhaustive path traversals along the document. The index structure integrated in ENAXS is comprised of three sub-components: *path index*, *node group* and *schema tree* as illustrated in Figure 3.4.

Path Index

Typically, indexes should be resided in memory and frequently accessed by query processor during query execution. This concept draws our attention to consider two important factors in designing indexes: *size of index* and *processing time*. The size of index is mainly contributed by two parts: the size of index keys and the size of references. In path index, index keys are the possible path expressions in the documents. The deeper the hierarchy of XML document tree, the longer the length of strings to represent paths. Such index generates a big structure in size and induces storage overhead that makes keeping the entire index structure in memory infeasible. An optimization technique is necessary to minimize the size of index.

Instead of indexing long path expressions, we employ a hash algorithm to generate equivalent hash values for those paths. The resulting values are then treated as index keys accordingly in the path index. We implement the path index using the B^+ -

Figure 3.5: Extended Node Groups Record Format

tree structure, a balanced tree in which leave nodes of the tree are block pointers referencing to the corresponding node groups. For a given search path, the path index generates an equivalent hash value, looks up in the index tree, and returns a pointer referencing to a particular node group whereby all nodes being accessible via a given path can be reached. Since all index keys are numerical hash values, simple arithmetic comparison can be applied to check the exact-matching between search and index keys, and number of levels in B^+ -tree can be significantly reduced.

Node Group

We organize all references that are pointing to nodes in the node repository according to the similarity in their path expressions and cluster them into multiple groups. Each group is associated with a *hash key* that is representing a specific path, and each group contains references to all internal nodes reachable via that path. *Node Groups* is a collection of such groups representing all possible paths in the entire storage. Typically, a hash value is unique in most cases and representing a unique path, but in some situations, the hash function may generate a same value for different paths, making a group representing more than a path. For this case, ENAXS allows forming a large common group, namely *extended node group*, to represent those paths. Figure 3.5 depicts the general format of an extended node group record.

A typical node group record contains a header and a body. The header in the node group record maintains the information regarding how many member groups being in the node group. Normally, there is one group in a record but it may contain more. In this case, *group path expression* can be used to identify which set of references represents which path. When a query is executed, the query processor searches for the header and identifies the correct node group with the correct path expression as given in the query.

There may be multiple node groups involved in the body of a record. *Group of node references* is a set of node pointers referencing to the node repository, and *pointer to next block* links to another block if the size of references exceeds a block size. Inserting, updating and deleting a node become easier in ENAXS and navigation over multiple documents is minimized since all related nodes are grouped accordingly. The following algorithm describes the insertion of a node into the node groups.

Algorithm: Insert(node)

```

1  path ← path expression of node
2  sig ← hash(path)
3  Search for sig in path index
4    If not found, then
5      Store node to Node Repository in a new block
6      Allocate a new block new_block in node group file
7      Create a new node group record, new_group, and
        Add the reference to new_block into new_group
8      Insert sig and reference to new_group into path index
9    else
10     Get the pointer to existing group old_group
11     Store node into block pointed by the last pointer in old_group
12     if block is full
13       Create new_block
14       Store node into new_block

```

```
15         Append new_block pointer to old_group
16     end if
17 end if
18 end
```

When a node is inserted into the node group, the hash value is generated as per its path expression, and validated against the index to verify whether there is any node group with the same path in the node group. If a group exists, the new node is inserted to corresponding block in the node repository that is referenced by the last block pointer of respective node group record. If the block is fully utilized, a new block is allocated in node repository, and the node is inserted to the new block. The reference to this new block is then appended to the respective node group record. If the group does not exist, the node is stored into a new block, a reference is then added to a newly created node group record, and then the node group record is associated with the new hash key that is to be stored as an index key of the path index.

Since the node groups can be seen as a part of storage structure, at the same time, a portion of index structure, it substantially reduces storage overhead and accelerates query execution as tree traversal can be done in the node repository while predicate value is scanned in the value repository.

Schema Tree

The schema tree is a tree representation of common structure for a set of XML documents. The main function of this tree is to lookup all possible full paths for a given path with regular path expressions. In general, it can be implemented using one of the following approaches:

1. *Top-Down approach*

This approach traverses the schema tree in the top-down fashion. For a given path, matching starts from the root and traverses along the paths. This approach is straight forward and easy to implement but takes longer time to determine descendents of a node and traversing all possible paths.

Figure 3.6: The Sample DTD and an XML Document.

2. Bottom-Up approach

This approach traverses the schema tree starting from the bottom to the root. This approach narrows down possibilities of traversal and cuts down the decision making time in each lookup since each node only has an ancestor. The drawback is that locating a leaf node in the tree may take longer time.

3. Hybrid approach

This approach is a combination of the previous approaches. It traverses from both root and leaf nodes of the schema tree simultaneously. This approach has the strengths and weakness of both approaches.

We implement *schema tree* using bottom-up approach. It is constructed by the use of either DTD or XML schema, and provides the ancestor-descendent relationship of a pair of nodes in order to deal with regular-path queries. Each node in the schema tree is assigned a unique identifier so that two nodes having the same node name but from different paths can be distinguished. For a given regular path expression, the query processor traverses along the schema tree and produces all possible full-paths that match to a given expression. We implement the schema tree using an *inverted list*. Figure 3.6 shows a sample DTD and a sample XML document, and Figure 3.7 depicts the scheme tree constructed from the sample DTD and its equivalent representation of inverted list.

For a given regular path, the last tag of the path is used to lookup in the index file of the inverted list to locate where is its parent node in the posting file. The

Figure 3.7: A Sample Schema Tree and Its Inverted Representation

resulting node is then used as an index entry to *inverted list* again to go next level up. This procedure continues until the root node is reached, and all possible full-paths can be obtained. The query processor uses the resulting paths as input and resolves the query. This approach magnifies the capabilities of schema tree that is not only providing efficient exploring of structure but also enhancing the query processing because the query can be answered preliminarily by identifying whether a given tag exists in the schema tree or not, without the need to search in repositories.

3.2.3 Enhancing Index Capability with Value Index

We also integrate a value index in NextDB to support queries with predicate over values. The value index is built over all external nodes in the value repository and provides an information-retrieval (IR) style keyword search for a word or a group of words. For a given value for a search, the value index locates value records containing a specific word(s), and returns a set of (DID,PID) that can be used to identify document and parent node it belongs to. The value index is created using keywords from external nodes by applying IR tools such as stopwords remover, and stemmer. A combination of value index, schema tree and path index in ENAXS can efficiently resolve regular

path queries with predicates over values.

3.3 Performance Evaluation

In this section, we will present performance evaluation results focusing on the space and I/O cost for storing and querying XML data with ENAXS. The measurement was taken on two aspects: Storage space utilization and query processing cost. The first experiment concerns the efficiency of storage space utilization, and the second experiment seeks to evaluate the I/O cost for querying data with the support of indexing schemes.

3.3.1 Experimental Environment

ENAXS was developed using the Java 2 Platform Standard Edition version 1.4 Software Developer Kit (SDK), and runs on the Java Run Time environment (J2RE 1.4.1). The evaluations were performed on a Pentium-III 800MHz machine with 512MB RAM under Windows 2000 operating system using 20GB hard-disk. Initially, we use IBM 4J DOM Parser [xml02] as a parser in ENAXS. However, we encountered various issues while parsing a large volume of data. DOM specification provides a very rich and intuitive structure for housing the XML data, but it is quite resource-intensive since an entire XML document is typically loaded into main memory as a document tree. For the case of very large XML document such as XMark data set, there is limitation on out of memory capacity.

On the other hand, SAX parser uses event based approach where parser calls handler functions only when certain events such as start and end of the document, finding a text node, find child elements, and hitting a malformed element take place. With the support of SAX parser, a large document can be parsed into ENAXS without having resource issues.

3.3.2 Experimental Data

Three types of data sets, Shakespeare's Plays [sha02], KJV Bible [kjb02] and XMark data [xma02] are used for performance evaluation in which Shakespeare and KJV are

Data Set	Size	Files	Elements	Attributes	Depth
Shakespeare	7.9M	37	327K	0	7
KJV Bible	4.9M	1	1	32K	5
XMark	115M	1	1,665K	381K	12

Table 3.1: XML Data Sets for Performance Evaluation

the real-world data while XMark is a syntactic data set. The details of all data sets are summarized in Table 3.1.

From the above table, we may summarize that Shakespeare’s Play data set is about 7.9MB in size organized in 37 documents containing deep and nested node hierarchy. All documents have similar hierarchical structure and each document is organized in an individual file. Each file contains different size of data, and a common external DTD is defined to conform the structure of all documents. KJV Bible file is a small data set of 4.9MB in size organized in a single document. The structure of the document is relatively flat and an external DTD is used to present the structure.

On the other hand, XMark is a syntactic data set that is generated by a data generator in various data size of 10MB, 20MB, 30MB...110MB, etc. In terms of structural form, XMark data is of deeper and more nested in hierarchy than Shakespeare’s Play, and stored in a single file like KJV Bible. To conduct thorough experiments with various data sizes, we generate multiple XMark data files, and evaluate ENAXS to find out how the repository reacts when the original data size is changed.

3.3.3 Storage Space Utilization

To evaluate ENAXS to see how it works with various settings, we change both block sizes BS , and the threshold node set α into different values accordingly. We monitor the storage on how the sizes of data files including node repository, value repository, node groups, and index file are changed accordingly. We do not take into account the size of schema tree as it is small in all data sets and it is not affected by changes in data size, block size, and threshold node set value.

Figure 3.8 and 3.9 illustrate the growing trend of data files and repositories sizes while loading Shakespeare’s Plays XML data files. Figure 3.8 (a) and (b) show the

Figure 3.8: Size of Data Files and Repositories for Shakespeare's Play (1)

Figure 3.9: Size of Data Files and Repositories for Shakespeare's Play (2)

size of data files and repositories in the situation when page size alters from 4KB and 8KB respectively and α is fixed to 25. Figure 3.9 (a) and (b) show the situations when page size is the same as 4KB and 8KB respectively and α is altered to 10.

Figure 3.10 shows the outcomes of XMark data set tested with various size of input documents. As shown in the results, the size of the value repository is almost the same as the original size while the node repository increases almost 3 times to the original size. Figures 3.10(a) and (b) show the situation when α is set to 10 and 25 respectively while fixing BS to 8KB. The index file and node group file occupy a small space according to the original file.

Figure 3.10: Size of Repositories for Various XMark Data Sets

Index File

Normally, the path index occupies a small space. For Shakespeare's Plays data set, it always occupies 3 blocks no matter how the page size changes since it contains 57 distinct paths. For the KJV, the index size is only 1 block since there are only 5 distinct paths in the data set. XMark data set with 115MB data in size occupies 30 blocks of 8KB pages as it contains 548 distinct paths in the document.

From the experimental results, we also found that the number of blocks required to hold index structure does not significantly vary with the changes in block size. The curve for the size of the path index is relatively flat even though the original document size is significantly increased.

Node Group File

As shown in the figures, the size of the node group increases linearly when the data set grows. After the block size is set to a greater value, we found that the number of blocks occupied by node group file decreases. The Table 3.2 shows that 59 blocks is necessary to allocate Shakespeare's Plays data with $BS=4KB$ while it goes down to 57 with $BS=8KB$. Similarly, KJV goes down from 33 to 23 while XMark decreases from 560 to 549.

Data Set	Block Size	Node Group Size	# of Blocks	# of Groups
Shakespeare	4 KB	241.68 KB	59	57
Shakespeare	8 KB	466.96 KB	57	57
KJV	4 KB	95.85 KB	33	23
KJV	8 KB	171.42 KB	23	23
XMark	4 KB	2,334.66 KB	560	549
XMark	8 KB	4,921.71 KB	549	549

Table 3.2: Node Group Size for Shakespeare, KJV, and XMark Data Sets

Node Repository

Node repository occupies most of the space in the database. Figure 3.8 and 3.9 shows that the size of the node repository increases proportionally with the size of the Shakespeare's data set. The experiments with $BS=4KB$ and $\alpha=25$ show that the node repository is approximately 2.5 times of the original data size. The ratio changes to nearly 3 times when BS is changed to 8KB as shown in Figure 3.8(b). In Figure 3.9(a), when BS is set to 4KB and α is 10, the node repository is around 3 times of the size of the data set. When BS alters to 8KB, the node repository is also roughly 3 times of the original data set size.

By comparing (a) and (b) in Figure 3.8 and Figure 3.9, we observe that when block size is fixed, larger value of α supports smaller storage space utilization for the data sets that contain nodes with large number of childs. From Figure 3.8 and Figure 3.9, the results demonstrate that changing smaller block size may reduce the storage utilization as well.

Figure 3.11 and 3.12 show the amount of node info blocks, direct and indirect blocks in the node repositories for all data sets that we observe. From there, we analyze that the indirect blocks occupy large portion of the data, especially for the data set with deeper hierarchy such as XMark. After increasing the value of α , the total number of blocks required for direct and indirect blocks are significantly reduced. This indicates that the minimizing number of indirect blocks in the storage by adjusting α value has a greater impact to the size of the Node Repository.

Figure 3.11: Block Allocations for Shakespear's Play and KJV Data Sets

Figure 3.12: Block Allocations for XMark Data Set

Value Repository

From all experiments with various data sets, the results illustrate that the size of the value repository is of almost the same as the original data set since the majority of input data are scalar values without any markup. In most cases, the two curves for size of value repository and original data set nearly overlap with each other.

3.3.4 Data Parsing and Storing Time

Regarding to data parsing and storing time in ENAXS, we observe that it is linearly increasing with respect to the growth of data size as shown in Figure 3.13. With the experiments over various sizes of the XMark data set, it demonstrates that the increasing cost of parsing and storing data is insignificant in ENAXS even though the growth of the original data size is fairly large.

We also perform experiments of data parsing and storing time over Shakespeare's play and KJV data sets and the results are depicted in Figure 3.14. From there, we can

Figure 3.13: Parsing and Storing Time for different Data Sizes of XMark

XML Documents	Index Size (KB)	Group Size (KB)	Node Group (KB)	Value Repository (KB)	Time (Second)
10 XML Documents (Shakespeare)	11	145	7293	2133	38.115
20 XML Documents (Shakespeare)	11	197	14233	4165	187.129
KJV Bible	2	21	6233	4605	35.021

Figure 3.14: Performance Comparison for Parsing XML Documents

analyze that data with many attributes and attribute values, like the KJV data set, it takes a bit longer time in ENAXS since additional time is needed to distinguish those attributes from element and value nodes.

3.4 Summary

This chapter covered the detailed description of a new storage structure that is specifically designed for maintaining multiple, large and complex XML documents in its native format to overcome the deficiencies of traditional databases and, to provide optimal storage utilization and efficient query processing. The storage is organized with two main structures: *node repository* for internal nodes and *value repository* for external nodes. We explored the detailed architecture and record structure of each repository in this chapter. In addition, we elaborated the structure of indexing scheme and its components: *path index* to enable direct access to nodes, *node group* to maintain a collection of nodes with a same path expression and *schema tree* to

transform regular path expression in the queries to simple path expressions.

We evaluated the various experiments of the performance with real data sets, syntactic data set and presented the results in terms of storage space utilization and I/O assess costs. The results through experiments have shown that the storage system enables to perform efficient storage, indexing and retrieval of XML data. The next chapter presents the design and implementation of NextDB's query processor that supports the execution of user queries and retrieval of XML data from the database.

Chapter 4

Query Processing

The ability to manage vast and complex repositories of data and searching for relevant information are fundamental attributes of a database management system. A DBMS provides a robust and scalable storage structure to manage data efficiently. At the same time, it is necessary to offer a relevant mechanism that supports effective data manipulation and retrieval. A query processor plays this critical role by resolving user queries and facilitating data lookup and retrieval.

Querying XML data over a large-scale database demands sophisticated data manipulation mechanisms and retrieval methods. Unlike traditional data models, XML data is semistructured; it does not exhibit a well-formed and consistent structure that makes querying data difficult. Although information may be of similar nature, there may be inconsistencies in structure and typing across different instances. In addition, the data are organized as trees; this is different from flat-structured data model and renders most of the querying and data manipulation methods widely-used in well-known relational database systems inapplicable.

The adoption of more representative native storage models and efficient indexing schemes lead to the motivation of designing an entirely different query processing mechanism that provides a set of data operators for querying an XML database, and manipulating, retrieving and restructuring XML data. The main objective in the development of XML query processing, in essence, is to accelerate the retrieval of data of a hierarchical nature. Additionally, there is a need to improve data representation capabilities and data interchange facilities. As a consequence of this innovative perspec-

tive, several XML query languages including XML-QL [DFF⁺98], YATL [CDSS98], Quilt [RCF00] and Lorel [AQM⁺97], XML-GL [CCD⁺99], XQL [RLS98] have been proposed over the past years.

Since the emergence of XML is relatively young, there still imposes new interesting challenges that motivate database researchers to learn and adopt more in certain areas. With increasing interests in native XML databases, new query processing engines that are designed to be embedded in native databases such as Lore [QWG⁺96], Natix [FHK⁺02] and Timber [JAKC⁺02] are introduced. The architectures of these engines are fundamentally similar and they support the necessary operations that are relevant to XML querying. However, the method of query execution, optimization, and data representation are different.

In this chapter, we present an overview of data models and a functional classification of existing XML query languages and query processing engines. Then, we elaborate the details of query operators and its functionalities followed by the implementation of the NextDB query processor - a crucial component of NextDB to facilitate complex XML query executions and data navigation. We describe more on how each data operator works, how a user query is answered, and how the three query components are incorporated into to resolve a user's query. Finally, we conclude the chapter with a performance evaluation of the proposed query processor.

4.1 Related Work

There have been many proposals that extend the capacities of existing query languages with path expressions, or that introduce new query languages that are specifically designed for XML. As an early work on semistructured data storage, Lorel adopts the patterns of a strong OQL query engine and includes additional features such as the extensive use of coercion and powerful path expressions that are relevant to XML [AQM⁺97]. Query processing with Lorel relies on a DataGuide [GW97] that provides path traversal operators to resolve path and tree patterns. Similarly, XML-QL adopts strong SQL's construct with extended clauses to build the document resulting from the query [DFF⁺98].

Alternatively, XML-GL is more declarative as it is a graphical query language relying on a graphical representation of XML documents [CCD⁺99]. It allows users to visually display query and execution results in XML format. XQL provides a rich set of operations for XML data manipulation, and integrates query functionality into the Extensible Style Sheets (XSL) syntax [RLS98]. On the other hand, XQuery [Gro02] is a new W3C specification - an extension of Quilt [RCF00] and XPath [Gro99a] that offers more declarative, flexible and easily understood expressions.

In the context of data model, Lorel, XML-QL, XML-GL, XQuery and XSL use their own data models while XQL depends on the data modelling features of XML. Except for XML-QL, which is able to query the ordering of the underlying data model, all elements in the rest of the data models mentioned are either ordered or unordered. Lorel and XML-QL deliver powerful and flexible path expressions whilst XML-GL supports only arbitrary containment to traverse the graph. XSL and XQL define path expressions as relative and absolute locations, and XQuery adopts XPath's expression which is typical and widely used in XML related applications.

In terms of declarative characteristics, Lorel is similar to the calculus-based OQL language. However, XML-QL is closer to XML while XML-GL has Web-familiarized URL-like pattern and XQuery has a simple FLWOR expressions pattern. From the perspective of query expressiveness, Lorel, XQuery and XML-QL are powerful as it has the ability to manage abstract data types, type coercion and updates, universal qualifications, negation as well as aggregate management. On the other hand, XML-GL, XSL and XQL have less expressive query processing power.

From the other angle, most of the native databases provide query processing engine that can evaluate user queries in the form of expression using one of the XML query languages. Lore database has its own Lorel query language while Natix [MHKM04], Timber [AKYJ03] and Tamino [Sch01] use XQuery and XPath expressions. The implementation of the query engine, and query evaluation in each database are different as it has its own architecture. Lore designs various steps of query execution, namely, parsing, preprocessor, logical query plan generation, query optimization, physical query plan generation and finally execution of physical query plan. In Natix, there

involves two steps of execution: query compilation and query execution. The query compilation translates an input query into an optimized query execution plan and the query execution manipulates the output plan by the use of query execution engine. Natix assigns a minimal set of data operators that are managed under the Natix Physical Algebra.

XML queries in Timber are passed to the TAX algebraic operator tree by a query parser. The query optimizer then reorganizes this tree based on a set of rules and metadata information that are collected in the form of positional histograms and performs the required mapping from logical to physical operators. The resulting query plan tree is evaluated by the query evaluator, pipelined one operator at a time, by means of a set of calls to the Data Manager and Index Manager, which in turn call the underlying *Shore* storage. Our work differs in such a way that we mainly focus on the effective manipulation of trees and nodes. All data operators used in NextDB are tree-based and can work with XML data in the functional area of node grouping, matching, manipulation, tree traversing, etc.

4.2 NextDB Query Processor

In the design stage of the NextDB query processing engine, the objective is to deliver efficient services that address current obstacles and limitations of state-of-the-art XML query engines, notably:

- to provide a declarative and resourceful query engine that enables efficient evaluation and answering of users' queries.
- to furnish a set of function-worthy query operators that support the execution of both individual operand and group of operands.
- to resolve user queries with regular path expression and long path traversal.
- to allow information retrieval-style partial and full-text searches.
- to optimize query execution to boost the performance of data retrieval and to minimize resource utilization.

- to deliver a user-friendly query interface that eases users' communication with the database.

In order to address the objectives, we deliver the following query processing components that extend the capabilities of NextDB to effectively deal with semistructured XML data.

- We implement a NextDB query processing engine that is capable of resolving XML queries without the need of an additional mapping layer.
- We adopt the XQuery syntax; it is a W3C standard query language that allows declarative FLWOR expressions.
- We define a set of algebra-based query operators that enables query execution by the use of multiple operands.
- We develop a graphical query interface that is embedded in the NextDB DB Manager that allows users to communicate with the database.

4.2.1 Designing Query Processor

Since the introduction of the relational model, two key query language paradigms have evolved, namely, *algebraic* and *calculus-based*. Calculi differs from algebras in that they are non-procedural. This feature makes them particularly suitable for user level query languages. However, a declarative specification of a query may not at the same time represent the most efficient evaluation strategy. In this aspect, algebras that use the power of formalism to enable optimizations appear more attractive. The NextDB query engine is designed based on query algebra. The choice of the algebraic approach is due to its simplicity and mathematical rigor.

The algebra for data operators is defined as a pair $\langle \text{ObjectSet}, \text{OperatorSet} \rangle$ formally in group theory. The set of objects in the NextDB algebra is a universe of node sets. Hence, the objects of manipulation for the operators are node sets. Based on this, we introduce a compact set of data operators for the query processor.

Operator	Description
Extend	The set of nodes reachable from node set N via path p .
Select	The subset of node set N comprising nodes that satisfy the condition $\Theta \in \mathcal{C}$.
Product	The node set (virtual nodes) comprising all orderings among nodes in M and N .
Partition	A collection of node sets (partition) indexed by the node set $\Phi_p(N)$.
Match	The node set of all possible valid matchings between partitions N and M .
Break	The node set union of all cells in partition N .
Project	The node set union of cells in partition N indexed by nodes of partition M .
Union	Set union of node sets M and N .
Intersect	Set intersection of node sets M and N .
Difference	Set difference between node sets M and N .

Table 4.1: Summary of Data Operators

4.2.2 Data Operators in the Query Processor

In general, an operator is a tool that manipulates data. Data operators are the core of a query processor and usually manipulate data elements in the database in order to search and retrieve them. For XML data, operators focus on manipulating data that is organized as a node or a set of nodes in the database. For instance, the *Select* operator chooses a particular node or a set of nodes by certain conditions and then collect resulting nodes to form a node set that will be returned as a result. A summary of algebraic operators used in NextDB is shown in Tables 4.1.

Before explaining the functionality of each operator, we need to define some common symbols that are shared among data operators. The query algebra is defined formally in group theory as a pair $\langle S, \Omega \rangle$, where S is a set of objects and Ω is a set of operators. The set of objects S in the NextDB algebra is a universe of node sets $\mathcal{P}(\mathcal{N})$. Hence, the objects of manipulation for the operators in Ω are node sets. In Tables 4.1, $\mathcal{M}$ refers to a set of Markup. In the NextDB data model, it is a set of element nodes. Hence, $\mathcal{M}^n$ represents a path set, and for each path; it is formed by n element nodes. $\mathcal{C}$ represents a condition set. Since $\mathcal{P}(\mathcal{N})$ is used to represent a node set, we use $\mathcal{P}^2(\mathcal{N})$ to represent a set of node sets. For example, $\{\{n_1, n_2\}, \{n_1, n_4\}, \{n_2, n_4\}\}$ is a set of node sets.

Extend

An *extension* of a path with respect to a set of nodes is the set of reachable nodes via a given path for each node in a given node set. The concept of extension is a useful generalization of reachable nodes from a set of source nodes. It is made up by reachability function that manipulates two inputs: a node and a path. The output of the reachability function is a node set; for each node, it can be reached from a node by a given path. For the **extend** operator, both the source and the output is a node set. The path p used in the **extend** operator Φ can be either forward or reverse. If the path is empty $p = \epsilon$, it returns the source node set. That means that if $p = \epsilon$, then $\Phi_\epsilon(N_S) = N_S$. Most of the time, the **extend** operator is used to manipulate internal nodes. In order to reach external nodes, path p must end with a wild card label $\#$.

An *extension* $\Phi_p(N)$ of a tag path p with respect to a set of nodes N is the set of reachable nodes via p for each node in N ; i.e.,

$$\Phi_p(N)I\& = I\& \bigcup_{n \in N} \phi(n, p) \quad (4.1)$$

The concept of extension is a useful generalization of reachable nodes from a set of source nodes. It is explored more in detail below.

We introduce two symbols in the equation 4.1. First, $\phi(n, p)$ is the reachability function. It has two inputs: a node $n \in \{\mathcal{N}_0 \cup \mathcal{N}_I\}$ and a path p . The output of this reachability function is a node set N_R ; for each node $n_r \in N_R$, it can be reached from node n by path p . As the operator seldom deals with singular data entities, suppose the source is a node set N_S and for each node $n \in N_S$, we use the reachability function $\phi(n, p)$, then the result is many sets of node sets. We then use a **union** operator to combine all these node sets into one node set. Note that, for the extend operator Φ , the source is a node set $N \in \{\mathcal{N}_0 \cup \mathcal{N}_I\}$, the output is also a node set.

Select

The function of the **Select** Ψ operator performs what its name implies: It is used to do selection. The **select** operator $\Psi_\Theta(N)$ selects a subset N' from a set of nodes N such that the set of reachable nodes via a given path for each of the selected nodes

satisfies a *selection condition* Θ . In particular, a node $n \in N$ is included in N' if the condition Θ applied to the reachable nodes of n via path p evaluates to true. For each node $n \in N$, we refer to the set of reachable nodes $\phi(n, p)$ from n via p as an *evaluation set* and the path p as the *evaluation path*. In the NextDB algebra, we define two kinds of select operations: **existential select** and **universal select**.

In existential select, a node $n \in N$ is selected if the selection condition Θ is satisfied by one or more of its reachable nodes via path p in the node set $\phi(n, p)$. Unlike the existential version, in universal select, a node $n \in N$ is selected only if the selection condition Θ is satisfied by all its reachable nodes via path p in the node set $\phi(n, p)$.

Product

The cartesian product of two sets is a set of ordered tuples such that the first component of each tuple belongs to the first operand set, and the second component to the second operand set. In the NextDB data model, we create a virtual node $\mathcal{V}$ to represent a tuple. The **product** operator in the algebra takes two node sets as input, and generates a set of virtual nodes. For each virtual node v , it has two child nodes; one from the first operand set, and the other one from the second set. Note that the virtual node is one kind of internal nodes in NextDB. We formalize this concept in the following definition:

$$N_1 \times N_2 I\& = I\&\{v \in \mathcal{N}_I \mid \text{childnode}(v) = \{n_1, n_2\}, n_1 \in N_1, n_2 \in N_2\} \quad (4.2)$$

As mention above, a virtual node belongs to the internal node set, $\mathcal{V} \subset \mathcal{N}_I$. There are some special characteristics for a virtual node that is different from internal nodes:

1. Virtual node does not have attributes or tags;
2. Virtual node set is unordered.

A virtual node set is typically unordered since virtual nodes do not originate from any of the documents in the document database. It is not possible to define an ordering among them. However, the child node set of a virtual node is ordered. The order is determined by the relative order of the operand node sets. For example,

in the above equation, N_1 is followed by N_2 . This implies that product operator is noncommutative.

Join

The product operator creates a set of virtual nodes with all possible orderings among nodes in the operand node sets. In order to restrict the result to the required virtual nodes only, a selection may be performed after the product operation. We name a product-select operation by *join*. Hence, a join $\bowtie$ between two node sets N_1 and N_2 can be expressed as

$$N_1 \bowtie_{\Theta} N_2 I\& = I\&\Psi_{\Theta}(N_1 \times N_2) \quad (4.3)$$

Where Θ is the select condition, and only those virtual nodes that fit the select condition will be included in the result node set.

Partition

Suppose there is a node set N . If $N = \bigcup_{i \in I} N_i$, then $\{N_1, N_2, \dots, N_i\}$ is called a partition of node set N where I is an index set. For example, $N = \{n_0, n_1, n_2, n_3, n_4, n_5\}$, then $\{\{n_0, n_1\}, \{n_2, n_3\}, \{n_4, n_5\}\}$ is a partition of set N .

In the NextDB algebra, given a node set N and a path p , an extend of node set N by path p generates a node set N' . The index set I is the set of identity values of nodes in N' . For each node n in the node set N' , we select those nodes in node set N that can reach node n by path p , and group these selected nodes as a node set. Hence, for each node n in node set N' we generate a set of nodes. We name this set of nodes by the identity of node n and group these generated node sets together to form the result of the partition.

In formal terms, given a node set N and a path q , a partition of nodes (denoted $[N]_q$) with index set $\Phi_q(N)$ is a set of node sets $\{N_i\}_{i \in \Phi_q(N)}$ such that there exists reachable nodes from all nodes in N_i for each $i \in \Phi_q(N)$ via path q that are equal. That is,

$$[N]_q I\& = I\&\{N_i (i \in \Phi_q(N)) \mid N_i \subseteq N \wedge \forall n \in N_i, i \in \phi(n, q)\} \quad (4.4)$$

The path q , referred to as the *partition path*, may be an interior or exterior path. When the path q is an interior path, node set N will be partitioned based on the internal node identity i where $i \in \Phi_q(N)$. When the path q is external path, the node set N will be partitioned based on the scalar value of the node set $\Phi_q(N)$. A special case is when path q is null; it can be done by self-partition. The general definition of self partition is

$$[N]I\& = I\&\{\{n\} \mid n \in N\} \quad (4.5)$$

For example, $N = \{n_1, n_2, n_3\}$, $[N] = \{\{n_1\}, \{n_2\}, \{n_3\}\}$.

The resulting objects that the operator manipulates is a node set containing most of the characteristics of *set*, for instance, no duplicate nodes in the node set. The resulting nodes in the node set is ordered by lexical precedence. In order to manipulate the node set, it is necessary to add *set theoretic operators* in NextDB query processor. For this case, three basic operators, namely **set union**, **intersect** and **difference**, are introduced.

Union

The **union** ($\cup$) of two node sets N_1 and N_2 is a set of all nodes belonging to N_1 or N_2 or both. Node set union is associative and commutative.

Intersection

The **intersect** ($\cap$) of two node sets N_1 and N_2 is defined as the set of nodes that belong to both N_1 and N_2 . Like **union**, node set intersection is associative and commutative.

Difference

The **difference** $\setminus$ of two node sets N_1 and N_2 is a set of nodes that belong to N_1 but not to N_2 . Node set difference is neither associative nor commutative.

Statement

Statement is the primary building blocks of a query file. It is a command line command containing an operator with one or more operands. The typical format of the

Operator	Statement Format
Union $\cup$	$N := \text{Union } N \text{ with } N$
Intersect $\cap$	$N := \text{Intersect } N \text{ with } N$
Difference $\ominus$	$N := \text{Difference } N \text{ with } N$
Product $\times$	$N := \text{Product } N \text{ with } N$
Extend Φ_p	$N := \text{Extend } N \text{ by } p$
Partition $[]_p$	$N := \text{Partition } N \text{ by } p$
Select Ψ_1	$N := \text{Select } N \text{ where } p \text{ R Literal};$
Select Ψ_2	$N := \text{Select } N \text{ where } p \text{ R } N;$
Select Ψ_3	$N := \text{Select } N \text{ where } p \text{ R } P;$

Figure 4.1: The Statement Format for NextDB Algebraic Operators.

statement is as follows:

$$statement ::= NODESET ASSIGN expression SEMI$$

where *NODESET*, *ASSIGN* and *SEMI* are all tokens that can be recognized by the query processor. For an example statement “*book_Set := Extend NodeSet by pub.book;*”, *NODESET* refer to *book_set* which is a node set to store return value, and *ASSIGN* is the assign mark ‘:=’; expression referring to *Extend NodeSet by .pub.book*, and *SEMI* is semicolon that ends the statement. Figure 4.1 depicts a list of statements for all operators defined in NextDB query processor.

With the combined use of these operators, the query processor is capable of resolving various types of user queries. More elaboration on how a query is executed is covered in later sections. In the next section, each functional component of the the query processor and their implementation is discussed.

4.3 Implementation of Query Processor

Typically, a query processor accepts users’ query as input, retrieves data, manipulates it and returns the result back to user. XML differs from traditional flat data in that XML is in a tree-like structure. Thus, document navigation and result generation are common processes in resolving users’ queries. In order to address these requirements, we design NextDB’s query processor with three functional components.

The processor is mainly composed of a *parser*, a *tree builder* and a *result generator*, as shown in Figure 4.2.

Figure 4.2: Main Components of NextDB Query Processor.

4.3.1 Parser

A Parser is a process that validates and translates a user's query statement. Since NextDB uses XQuery, the grammar unit and lexical patterns are defined according to the XQuery specifications. The original query in the XQuery expression is first synthetically validated according to the grammar rules and then translated into the command file that contains a set of internal commands. The other functionality of the parser process involves assigning specific sequence of tokens to each command in order to be executed by the query processor in sequence. Figure 4.3 demonstrates how the parser works in NextDB query execution.

4.3.2 Tree Builder

Tree builder is a process that builds a query tree that is a tree representation of an input query statement. In this process, each valid query statement in the query command file is organized to construct a tree with respective operator and operands. They are finally link with each other to form a complete query tree to represent a query. Figure 4.7 illustrates a query tree equivalent to a user query shown in Figure 4.5.

Figure 4.3: The Parser.

Note that a *circle* in the query tree represents an operator and a *rectangle* stands for an operand that becomes an input when a corresponding operator is executed. The arrows in the query tree indicate the sequence of execution and operators are executed as per sequence navigated by arrows in the tree.

4.3.3 Result Generator

Result generator is also a process that traverses the query tree as directed by arrows, fetches an operator, and then executes it using operand or operands provided. The execution result of an operator becomes an operand for its successive operation. This process is repeated until the root is reached and the final result is obtained. The results are then converted into XML format for the user.

Figure 4.4: The Query Execution in NextDB.

4.3.4 Query Execution

NextDB's query processor is designed to resolve queries in XQuery's FLWOR expression that forms the skeleton of XQuery. The query execution in NextDB is efficiently supported by the storage structure and accelerated by the combination of the path index and the Value Index as well as the Schema Tree. Figure 4.4 elaborates how a user query is executed step-by-step in NextDB. Figure 4.5 depicts a sample XML

document for an understanding of how all the components work together in every part of the query execution process.

User query : "Get a pair of books and articles selected by author's name"

User query in XQuery Expression :

```
for $b in bib/pub
  for $c in bib
    where $b/book = $c/article
  return {$b/book, $c/article}
```

First of all, the query interpreter fetches the query in FLWOR expression from the user interface, and subsequently translates into the query command file that is understood by the underlying NextDB query processor. Figure 4.6 shows the resulting query command file for the input user query.

Figure 4.5: A Sample Document for a Bibliography Database.

Each query statement is then used to construct an equivalent query tree where the operator, necessary operands and query result are the nodes of the tree. Those trees are then linked by each other to form a complete query tree that is a tree-style representation of the input query. Figure 4.7 illustrates four command trees that represent four internal command statements, and a complete tree that is a combination of those statements.

```

/*-----
/Query Description:  Pair of books and articles by same author
/-----*/
//get book set
book_set := Extend n0 by .pub.book;

//get article set
article_set := Extend n0 by .article;

//pair book and article
book_article := Product book_set with article_set;

//select by author's name
select_books_article := Select book_article
                        where .book.author.# = .article.author.#;

```

Figure 4.6: A Sample Query Command File.

Figure 4.7: A Query Tree Built from the Sample Query.

As mentioned, a *circle* in the query tree represents an operator and a *rectangle* stands for an operand that is necessary in execution of the corresponding operator. The execution result of an operator then becomes an input operand for the successive

operator as sequentially navigated by the arrows. Finally, the execution results are generated to be in XML format that is ready to publish or exchange.

4.4 Performance Evaluation

The query processor described in this chapter is fully implemented and integrated into the NextDB, including the *Parser*, the *Tree Builder* and the *Result Generator*. The implementation uses Java 2 Platform Standard Edition version 1.4 software developer kit (SDK) running on Windows 2000 in Pentium-III 800MHz machine with 512MB RAM and 20GB hard-disk. Measurements on execution are taken over the number of I/Os scanned and read during the execution of a particular query. A comparison is done based on query performance with and without the support of the Path Index we have integrated with the NextDB storage. We also examine the crucial role of the index that accelerates query execution by means of minimizing tree traversal time along the paths.

4.4.1 Experimental Environment

Performance evaluation is done by executing different types of queries on the Shakespeare's Plays data set. Recall that Shakespeare's Play is a real world data set that contains 37 documents in total size of 7.9MB with the form of deep and nested node hierarchy. The document trees are made up of total 327K elements that are being organized in 7 distinct paths.

Throughout all experiments, we mainly focus on examining number of block I/Os being accessed while each query is executed. The reason is that, query execution time could differ depending on the speed of CPU, hard-disk and system bus. However, number of I/Os cost is independent of those factors. In addition, we select different types of queries from XMark benchmark [xma02] including exact match query, regular path query, full text search query and path traversal query for effective evaluation and justification. Table 4.2 shows the sample queries used in our experiments.

Query	Data Set	Query Path Expression
Q1	Shakespeare	PLAY.ACT.SCENE.SPEECH.SPEAKER
Q2	Shakespeare	*.SPEAKER
Q3	Shakespeare	PLAY.ACT.SCENE.TITLE.#
Q4	Shakespeare	PLAY.ACT.SCENE.TITLE

Table 4.2: Queries Used in Performance Evaluation.

Experiment(1)

We begin with Q1, a simple exact match query, to return the names of all *speakers* with the *title* ‘The Winter’s Tale’.

```

for $b in PLAY/ACT/SCENE/SPEECH
  for $c in PLAY
    where $c/TITLE = ‘‘The Winter’s Tale’’
  return {$b/SPEAKER}

```

This query mainly examines the ability of the NextDB to handle simple string lookup with a particular full path expression in the situation when a predicate-value is provided.

Experiment(2)

As regular path expression is a crucial part of any query language for XML and semistructured databases, we run Q2 to test the execution of returning *speakers* under all possible paths in the database.

```

for $b in PLAY
  return {$b//SPEAKER}

```

This query is used to test how well NextDB query processor can resolve regular path expressions and manage to optimize traversals along the possible paths. It is also an another way of testing how the query processor coordinates and works with the Path Index and the Schema Tree introduced in Chapter 3.

Experiment(3)

Q3 is a type of full-text search query to extract all values of *titles* where *speaker* contains a keyword ‘*Hostress*’.

```
for $b in PLAY
  where some $p in $b/ACT/SCENE/SPEECH/SPEAKER satisfies
    contains($p, ‘‘Hostress’’)
  return $b/ACT/SCENE/TITLE
```

We select this kind of query to examine how the query processor efficiently reacts in such situation supporting of the Value Index that is integrated in the NextDB.

Experiment(4)

Q4 tests the ability of path traversal by extracting *title* elements along a specific path.

```
for $b in PLAY/ACT/SCENE
  return $b/TITLE
```

The different between Q3 and Q4 is that Q3 extracts values by making use of the Value Index while Q4 traverses along the paths by the support of the Path Index.

4.4.2 Experimental Results

For all queries, the physical page size in the storage unit is set to 8KB and the threshold value is set to 10 for the direct child pointers. The experimental results for the abovementioned queries are shown in Table 4.3. From this experiment, we conclude that executing Q1 with the support of Path Index is approximately 500 times faster than the execution without using Path Index. This shows that Path Index has greatly accelerated query processing by providing direct access to nodes in XML tree.

The regular path query Q2 is executed using the Path Index and the Schema Tree. The experimental result demonstrates that the number of I/Os cost is about the same as the number of block I/Os incurred in fetching the results, with little additional

	Query	Index	Node Group	Node Repository	Value Repository	Total	No. of Nodes
With Index	Q1	1	1	183	0	185	30972
	Q2	7	6	188	0	201	31067
	Q3	1	1	5	752	759	630
	Q4	1	1	5	0	7	748
Without Index	Q1	0	0	101350	0	101350	30972
	Q2	—	—	—	—	—	—
	Q3	0	0	4658	752	5410	630
	Q4	0	0	4658	0	4658	748

Table 4.3: Block I/Os for Query Evaluation

I/Os cost on tree traversal. The result shows that the use of the Schema Tree to convert regular path expression to full path expression is quite efficient and reduces the traversal of the XML tree and I/O accesses significantly.

In terms of tree navigation, Q3 and Q4 are similar and not significantly different as both queries traverse along the path ‘PLAY/ACT/SCENE/TITLE’. However, Q3 goes one step further by retrieving the contents of the `title` elements from the Value Repository and search the keyword ‘Hostess’ with the help of the Value Index. We use this query to examine the performance of the Value Repository as well as the Value Index. The result shows that Q3 costs more than Q4 due to the additional accesses to the Value Repository. In addition, comparing the cost with and without support of Path Index for evaluating Q3 and Q4 demonstrates significant reduction of block I/Os similar to that occurred in Q1. By using the Path Index, Q3 is about 7 times faster, and Q4 runs 665 times faster.

4.4.3 Experiment with Improved Indexing Schemes

From the previous section, we can see that most of processing time is spent on disk I/O accesses. This cannot really reflect the advantage of our indexing scheme. In this section, we evaluate the query performance of our index. Similar to the previous section, four kinds of queries are evaluated and compared with the pervious results.

Recall that Q1 obtains all speakers that appeared in speeches with a particular title. Q2 is a regular path query that extracts all `speakers` in all `plays`. Q3 and Q4 are similar. Q3 extracts the value of the `titles` of `scenes` containing a spe-

Query	Execution Time with Improved Index (sec)	Execution Time with Former Index (sec)
Q1	114.615	129.937
Q2	124.141	128.966
Q3	0.819	1.512
Q4	0.132	0.271

Table 4.4: Comparison of Query Execution Time.

cific keyword, whereas Q4 extracts the `title` elements, instead of their values. The experimental result is shown in Table 4.4.

As in the data loading performance, Q1 and Q2 require a lot of disk I/O accesses. Thus, the advantage of new index can not be demonstrated substantially. However, in Q3 and Q4, the new index is obviously faster when few I/O accesses are required. The query execution time is reduced to half compared to the previous implementation.

We can conclude that the new index structure is particularly useful and improve query performance significantly in two situations. For the case queries that use the index heavily, the new memory-based index outperforms the former disk-based index. Secondly, for the case of potentially large Path Index, memory-based index substantially reduce tree-traversing time.

4.5 Summary

In this chapter, we presented the design and implementation of an algebra-based query processor that supports the execution of user queries and retrieval of XML data from the database. We define a compact set of algebraic operators that are used to manipulate data during query processing and, elaborated their usability and how each operator works. We also covered the implementation of the query processor by exploring details of the three main query engine components and their functionalities in query execution. By the use of an example query, we also illustrated how it is executed in NextDB query engine. In the next chapter, the transaction management system, one of the crucial parts of NextDB data management system, will be presented.

Chapter 5

Transaction Management for XML Data

Interactions among transactions may cause inconsistency in the database even though individual transaction performs correctly and there is no system failure. Thus, the order among different transactions must be regulated using rules [MUW02]. For an enterprise-level data management system supporting multi-users, transaction management is an essential component that makes distributed computing possible. As NextDB is a client-server based multi-users system, the management of concurrent transactions is crucial in order to assure the database consistent.

Different from traditional flat data, an XML document can be visualized as a tree that is formed by the nesting structure of the nodes. This characteristic imposes various issues in serializing concurrent accesses to particular data in the transaction management system. The most challenging part for researchers is to design rules for accessing a particular node in an XML document tree as the parent-child relationship between nodes along a path should be taken into account. In addition, updating a node in the database is also another aspect as it solely impacts the child nodes as well as the parent nodes.

We elaborate more in order to provide readers' understanding. As an example, updating a root node simply affects the entire document, as each path expression to individual node is changed. Thus, other simultaneous read or write accesses to the remaining nodes should be properly serialized, otherwise, it may not return the correct results. This type of data operation may change the structure of the document.

Thus, all possible dependencies must be considered. We bring these issues into consideration when designing an effective transaction management system. This chapter presents techniques for controlling transactions and concurrent accesses to ensure the correctness of the database state, and permit recovery when a system failure occurs.

5.1 Related Work

For any database system, a concurrency control protocol is the heart of the transaction management system. Consistency issues in hierarchical database systems, and TREE protocols have been introduced since 1980 [SK80]. Based on those concepts, many variants have been introduced over the past years. Some of the well-known protocols include Dynamic Tree-Locking Protocol [CM86] and Dynamic Directed Acyclic Graph policy [Cha95], which is a generalized and extended protocol of Safe Locking Policies [Yan82]. With the emergence of XML, XML-specific lock-based protocols such as strict two-phase locking [HKM01], locking on directed acyclic graphs with DGLOCK [TKHJ02] and XPath-based locking approach [HT03] are introduced in recent years. However, there has been relatively little attention to concurrency control policy based on timestamps. Although the extension of timestamps and multiversion timestamps protocols to hierarchical databases have been proposed [Car83], the extension does not quite fit the XML scenario. NextDB's concurrency control protocol, *XStamps*, considers the operation semantics from the XML perspective, and offers a more flexible concurrency control mechanisms to ensure the granularity of database.

5.2 Serializing Concurrent Transactions in NextDB

Concurrency control is not a new concept as there exist many techniques in traditional databases. However, these approaches may not be easily ported to XML since the XML data model and the storage structure are different. XML has its peculiarity in its data access model. In NextDB, the concurrency control system ensures that the hierarchical structure of XML is preserved and the access of the document always logically starts from the root. To achieve a high degree serializability, a straightforward

ward approach is to lock the entire document when some operation is trying to access or modify the document. The most obvious disadvantage of this approach is that no other transaction can simultaneously access the document even if they are accessing a different portion of the data. Thus, such a phenomenon does not guarantee high degree of concurrency.

Currently, there is no standard method to update XML documents partly due to the lack of efficient concurrency control protocols that ensures the correctness and efficiency of concurrent operations. The current situation imposes a pressing need to develop more advanced transaction management systems with elegant concurrency control algorithms. To bridge the gap, we propose XStamps, a multiversion, times-tamp concurrency control protocol for XML.

5.3 Designing Data Model and Basic Operations

The concept of maintaining multiple versions of data to increase the degree of concurrency was introduced since 1970s. The fundamental theory of this protocol is based on the concept that each write operation on a data item x produces a new version of x to avoid the rejection of read operations that come at a later time. To ensure this, each transaction is assigned a unique timestamp, and an operation is carried out as per the timestamp schedule of the corresponding transaction.

This protocol was later formalized and extended to the Multiversion Timestamps Ordering Protocol (MTOP) [Ree78a]. In this protocol, an order scheduler translates operations on a data item with respective timestamp into operations on a particular version of the data item. Under MTOP, each transaction is assigned a unique timestamp upon initiation and multiple versions of each database item X , one for each time the X is written, are maintained. Each version V of X has a read-timestamp $RTS(V)$, (the greatest timestamp among transactions which read that version), and a write-timestamp $WTS(V)$, (the greatest timestamp among transactions which write that version). An attempt to read a data item X is always successful if there exists a version V' with $WTS(V') < TS(T)$ and no other version V'' such that $WTS(V') < WTS(V'') < TS(T)$. An attempt to write a data item X is suc-

cessful if the version V of X with $WTS(V)$ maximal, subject to $WTS(V) < TS(T)$, also has $RTS(V) \leq TS(T)$; otherwise, T is aborted and restarted with a new timestamp [Ree78a].

In order to adopt and tailor this approach in an XML environment, we need to consider a fundamental data model and basic transactional operations working with concurrency protocol before describing how the protocol works.

5.3.1 Data Model

The data model that XStamps is integrated into is the ENAXS; we have discussed the NextDB data model in chapter 3. In brief, ENAXS focuses on the scenario that all nodes from multiple XML documents are clustered according to its path expressions, and collectively organized together in physical blocks so that necessary space can be reduced and user queries can be resolved with less I/O accesses while the required data is scanned and loaded [WNL03c].

Different from traditional timestamps protocols in flat data models, the serialization in XStamps goes beyond a specific tuple as the ENAXS data model is hierarchically organized. Nodes in ENAXS are clustered and maintained under respective node groups, whereby, linking to physical records in the node repository. XStamps addresses that issue by maintaining the current state of each node to ensure granularities. In addition, XStamps keeps a series of dependencies along paths on consideration during execution of transactional operations over nodes since ENAXS preserves all ancestor-descendent relationships among nodes. Moreover, clustering node in the data model brings significant advantage as XStamps eliminates redundant administrations while updating a particular node over multiple documents.

5.3.2 Basic Transactional Operations

Transactional operations form a set of basic operations that are applied to database items. In the stage of design consideration to define operation set in XStamps, we pay close attention to the following design goals:

- *Completeness* - Any XML document that conforms to ENAXS can be built by applying a sequence of basic operations.
- *Soundness* - Applying a sequence of basic operations to a valid XML document will result in another valid XML document.
- *Unambiguity* - The effects of the basic operations and the necessary parameters should be clearly defined in order to support the manipulation of the data model.
- *Efficiency* - Any common and useful tasks can be expressed efficiently as a sequence of basic operations.

For our case, only two operations, `READ` and `WRITE` are not sufficient in order to meet the design goals. To cater to a wide coverage of transaction execution, we define four basic operations, namely `READ`, `UPDATE`, `INSERT` and `DELETE`, abbreviated as R , U , I , and D respectively. Execution of the first two operations does not affect the structure of XML document while the last two do.

In XStamps, read and traverse are treated as the same operations, and those are defined by a read operation R . An important point to note is that the effect of R is specified differently than a formal read operation. For a typical hierarchical structure, concurrency control algorithms generally define an operation on a granule, i.e., the effect of the operation is the entire granule, including all sub-granules. In our case, we may not always read the entire sub-tree in the data model when reading or traversing a node. Thus, the R operation performs only at a particular node. The purpose of reading a node is either for reading the content associated with it or for traversing back and forth along the path.

The U operation in XStamps is also slightly different from a formal update operation. As stated earlier, the U in XStamps normally updates the data but not the structure of the XML tree. In other words, the U operation always performs on the leaf nodes in the XML tree that contains attribute values or contents. The I operation appends a node to the XML tree conforming to the definition of DTD. Iterated I operations are allowed in XStamps to insert a branch or a group of nodes. The effect

of D operation is different from others as it could delete a node or an entire branch rooted by that node.

5.4 Designing XStamps Protocol

The XStamps protocol is designed based on MTOP by adding new features. The reason for choosing MTOP is that it has several advantages; it provides highly effective concurrency levels, deadlock-free operations (no or limited blocking) and efficient operations unless there is a conflict. In addition, timestamps-based protocols perform better in a low conflict environment as the number of rollbacks is less when the degree of conflicts is low. Unlike timestamps-based protocols that require transactions to commit in timestamps order, the XStamps is designed to make use of its new features to allow transactions to commit early.

5.4.1 Features of the XStamps

There are two features that make the XStamps protocol distinct from traditional timestamps protocols. The first feature is the classification of transactions. Upon initiation, a transaction is classified to be either a read or write transaction. The criterion is that it is a write transaction if a transaction contains at least one action that belongs to $\{I, U, D\}$; otherwise, it is a read. Read transactions operate on the read version of data while write transactions operate on the write version.

The second feature is the creation of two parameters: *safety coefficient* (SCO) and *safety threshold* (STH). In XStamps, a safety coefficient is assigned to each database item (node in XML) and a safety threshold is assigned to each transaction upon initiation. *SCO* indicates the certainty of how sure the latest write transaction is able to successfully complete without rollback, and *STH* indicates the strictness of the transaction. For the environment where a transaction needs the most up-to-date and absolutely correct data, *STH* should be a very high value, while it can be a lower value for environments where the return of slightly old data is permissible or the correctness of data is not important (this does not mean a high probability of getting incorrect data). The advantage is obvious; it allows early transaction commit

and a more flexible control.

With the introduction of *SCO* and *STH* parameters, an issue arose here as to how to decide a good value for *SCO*. We consider *SCO* as a function of a composite variable F , where F is a composition of factors carrying different weights. Factors are the system state information that determines whether an action can be physically unrealizable or not based on system information. Some of such information includes:

- The number of write operations in the write transaction.
- The total number of transactions in the database.
- Contention level that is decided by the:
 - size of database;
 - length of transaction in terms of number of operations;
 - average number of write operations in each transaction.

With this set of factors, an efficient estimation algorithm is necessary to estimate the next state of the system based on the current situation. Certainly, designing a good *SCO* value is not simple and straightforward. Instead, we set *SCO* to the lowest value permissible before the current write transaction is committed, and reset it to a maximum value when it completes successfully. In addition, a higher *STH* value is set for all transactions as default in order to provide access to the most updated data.

5.4.2 Data Access Rules in XStamps

We define the following access rules to guarantee the serializability of transactions and to achieve a high degree of concurrency.

Transaction Classification Rule. Upon entry to the transaction management system, a transaction is classified as either a read or a write transaction. A transaction is a write transaction if and only if it contains at least one non-read operation.

Data Node Access Rules. We define the following data node access rules for XStamps to work with XML data.

- For any read or write access to a node, we traverse from the root to that node; i.e., we consider all nodes along the path from the root to the destination node.
- We take into account the existing access control of each node along the path. For instance, the current version of a node could be read while its immediate or aggregated parent nodes are being read. However, any read or update is not allowed while one of its parent nodes is being updated.
- We assign an appropriate access control to each node along the path before a destination node is updated. For instance, we allow the reading of the current version of its immediate and aggregated parent nodes while the node is being read. We do not allow any read or write to its parents while the node is being updated.

These rules ensure the correctness and consistency of data while concurrent transactions are simultaneously performed. In order to make these rules work, we implement the following algorithm to guarantee data accesses.

1. If a transaction T issues a read request $R(N)$ on node N ,
 - a) $TS(T) \geq WTS(N)$
 - i) If Safety Coefficient of N is greater than or equal to the safety threshold of T , i.e. $SCO(N) \geq STH(T)$, and N is not marked as deleted, $R(N)$ is granted and $RTS(N)$ is set to $TS(T)$ if $TS(T) > RTS(T)$. If $SCO(N) \geq STH(T)$ but N is marked as deleted, operation is cancelled, transaction continues.
 - ii) If $SCO(N) < STH(T)$, transaction needs to wait until $SCO(N) \geq STH(T)$.

- b) $TS(T) < WTS(N)$
 - i) If $SCO(N) \geq STH(T)$ and N is marked as deleted, if old write timestamp when it is marked $WTS'(N) \leq TS(T)$, grant $R(N)$. If $SCO(N) \geq STH(T)$ and N is marked as inserted, $R(N)$ is cancelled and transaction continues.
 - ii) Find a latest version N' of N with no version N'' fall in between N and N' that has $TS(T) \geq WTS(N')$ and repeat $R(N)$ on this version. If N' can not be found, rollback.
- 2. If a transaction T issues a write request $I(N)$, $D(N)$ or $U(N)$,
 - a) $TS(T) \geq WTS(N)$ and $TS(T) \geq RTS(N)$
 - i) If request is $U(N)$
Create a new node N' based on N with the updated value, set $RTS(N')$ to 0 and $WTS(N')$ to $TS(T)$, update $SCO(N')$.
 - ii) If request is $I(N)$, construct a new node N' and set $RTS(N')$, $WTS(N')$, $SCO(N')$ accordingly, insert a new mark to N' and append N' as a child of N .
 - iii) If request is $D(N)$, mark node N as deleted if N does not have any mark, set $WTS(T)$ $TS(T)$ and update $SCO(N)$. Wait otherwise until mark disappears.
 - b) $TS(T) \geq RTS(N)$ but $TS(T) < WTS(N)$
 - i) If $SCO(N) \geq STH(T)$
Cancel the operation and transaction continues.
 - ii) If $SCO(N) < STH(T)$, wait until $SCO(N) \geq STH(T)$ or occurrence of other case.
 - c) $TS(T) < RTS(N)$
Rollback transaction.

As all access starts from the root and proceeds in a top-down fashion, we can easily derive from the access rules the most updated version of a node N that always have the greatest read and write timestamps among the subtree rooted by this node.

Figure 5.1: Transaction Management System in NextDB

To access a lower level node, the request must fulfill all the requirements to all nodes along the path to avoid inconsistency. For instance, it is impossible to access a version of a node that has been written by a future transaction.

5.5 Implementation of Transaction Management System

A transaction is a set of operations and it requests reads and writes of database elements. These requests are passed to the transaction management system and executed by accessing the required data from the database. To support this operation, we implement two components in the transaction management system: *transaction manager* and *resource manager*, as depicted in Figure 5.1.

5.5.1 Transaction Manager

The main function of the transaction manager in NextDB is to ensure that all transactions are executed correctly in the assigned sequential manner. It involves two activities: *Transaction analyzing* and *scheduling*. Upon the submission of a transaction, the *analyzer* identifies the type of the transaction and the required resources to execute the transaction. It invokes a process called *enlist* that communicates with the resource manager and loads the necessary resources into memory.

On the other hand, the *scheduler* assigns a unique ID to each transaction, creates execution threads, assigns each thread to a transaction and executes them according to the timestamps defined. It determines whether the transaction is granted, nulli-

fied, delayed or aborted based on the access rules by preserving the ACID properties of transactions and consistency of database. After a transaction is successfully executed, the transaction manager starts another process called *prepare* that links the resource manager to commit all changes permanently. After receiving an acknowledgement from resource manager, the transaction is accomplished successfully and another process, the *delist* is invoked to release all used resources.

5.5.2 Resource Manager

The resource manager is similar to a buffer manager between the transaction manager and the database. It manages recently uploaded data and prevents direct accesses to the database. It responds to requests from *enlist*, *prepare* and *delist*, and interacts with the query processor, the log manager and the recovery manager by providing the required data for their processing. As soon as it gets a request for data, the resource manager checks whether the required data is available in the buffer or not. If the data is ready, it immediately responds to the request. However, it may need to load data from the database when it is not ready. Similarly, when it receives a request for committing data, the resource manager immediately writes data on disk to store permanently in the database. Similarly, for a request from *delist*, the manager releases resources from the buffer used in the previous transactions and responds with an *acknowledgement* if the operation is successfully executed; otherwise, it returns *abort* if the operation does not complete successfully.

5.5.3 Intercommunicating with Recovery Management

For a database system, maintaining integrity of data is crucial. Data must be protected against any type of system failure and a recovery system has to be provided to support the resilience of data. In order to provide this function, we design a recovery system in NextDB. Among different recovery techniques, NextDB uses *redo-logging* a simple and efficient strategy for logging and recovering data from failure. This strategy mainly depends on a *log* that keeps a history of changes in the database. A log basically maintains a sequence of log records that can be used to reconstruct the

database when there is a system failure.

The principle of redo-logging is that all transactions have to follow a rule called *write-ahead logging rule*. According to the rule, before modifying any database element on disk, it is necessary for all log records related to this modification of the element, including the update and commit record, to be written to disk [MUW02]. When a system failure occurs, the recovery manager uses the redo log file to identify the committed transactions, write values of database elements done by these transactions to disk and abort all uncomplete transactions.

We design a log manager to communicate with transaction manager and the recovery manager. The log manager organizes log records related to the transactions issued by the transaction manager. It also communicates with the resource manager to maintain a log file in memory and writes to disk after a certain period of time. We deploy a *checkpoint* technique to determine the time when a log file is copied to disk. If a system failure occurs, the recovery manager deals with the resource manager to obtain the log file from disk, examines it and uses it to reconstruct the database to a consistent state. The complete transaction management system is shown in Figure 5.1.

5.6 Performance Evaluation

As mentioned, we implemented the NextDB transaction management system with two main components: Transaction manager (TM) and resource manager (RM). TM ensures all transactions are executed correctly in order while RM manages recently uploaded data and prevents direct accesses to the database. Under TM, we implemented the XStamps protocol, the TREE protocol, and two-phase locking protocol (2PL) for performance comparison.

5.6.1 Experimental Model

The system was developed using J2SDK1.4.1. To make the evaluation efficient, we developed a transaction simulator to generate various types of transactions using the following parameters:

- *Number of transactions* - the maximum number of transactions that will be executed simultaneously.
- *Average number of operations* - specify how many operations, an average, each transaction has. Read transactions have 50% more chances to have more number of operations than average.
- *Percentage of write transactions* - specify approximate percentage of write transactions in the system.
- *Simulated maximum I/O time* - simulate the time each operation takes to do the work.

5.6.2 Experimental Results

Figure 5.2 shows the throughput comparisons based on evaluation. We set the simulated I/O time to 10ms in all graphs. The Write% curve indicates the percentage of write transactions ranging from 100% to 0%. Since the result of TREE protocol is almost equal to 2PL, we hide its curve.

The figure indicates that three protocols perform differently at different contention levels. The 2PL and TREE protocols outperform XStamps at very high contention levels, while XStamps reveals a definite gain when the degree of contention is relatively low. In addition, XStamps demonstrates its strength in situations when the average number of operations between 50% to 60% of transactions are write operations regardless of the number of transactions in the system. The phenomenon is due to higher rollback rate in a high contention environment; XStamps allows higher concurrency in all situations.

In a lower contention situation, at some point, all protocols yield unexpected low throughput as the set of transactions generated contains more `DELETE` operations that performs over very high levels of hierarchy. This may cause later transactions to become invalid and are aborted.

Figure 5.2: Throughput comparison for 10, 50, 100 and 200 transactions

5.7 Summary

This chapter mainly presented the NextDB's transaction management system covering the design and implementation of the XStamps concurrency control protocol that is based on traditional multiversion, timestamp concurrency control protocol. XStamp separates the transactions into read and write transactions and creates read and write versions of data implicitly. A new concept in this protocol is the idea of a Safety Coefficient (SCO). SCO enables flexible control of the isolation level of transactions and allows transactions to early commit depending on the requirements. The evaluation results are also depicted in order to show the performance gain over other lock-based protocols. The design and functional components of the transaction management system are also conducted to provide the complete picture of the system.

The next chapter will present the NextDB's virtual user interface that enables users to communicate with the database system interactively.

Chapter 6

Visual Interface for XML Data

In a database management system, the user interface is the most important communication hub between external users and the internal database system. It is seamlessly integrated in the database system so that all operations can be performed in an intuitive manner. Nowadays, the success of any new system is judged less on processing speed and memory consumption but more on the quality of the user interface [Fis89]. Thus, usability and accessibility of interface becomes critical in a database. A good user interface exposes full functionality to the users to support their needs and it does not lead to confusion and frustration.

In this chapter, we mainly present design and implementation of *DB manager*, a visual interface for NextDB [WRNL03]. Since native XML databases are quite recent in the database community, there is still a lack of standard and references for user interface and issues must be considered in the design stage. This chapter presents the issues encountered when the interface is tailored to meet users' needs to provide intuitiveness, simplicity, and usability. The *DB manager* is an integrated interface composed of various components. It provides interactive communication between external users and internal database. It offers a rich variety of functions and operations, and provides usability and accessibility to users.

6.1 Related Work

There are very few works on visualizing XML databases. Lore provides a visual interface for viewing DataGuides and interactive querying and searching semistructured

Figure 6.1: Architecture of NextDB User Interface

databases [GW98, QWG⁺96]. That interface allows users to perform simple keyword searching and result viewing. X^2 is a graphical user interface for visualizing query and exploring complete and large answer sets of XML databases [MSB03]. It allows users to graphically submit a query to the database and obtains the query result in a compact form called the Complete Answer Aggregate (CAA). BBQ is a graphical user interface with strong support for interactive querying and browsing XML data [MP00]. It allows the querying of data with loose or incomplete schema and introduces visual result view to support query refinement. The common weakness of the above systems is the incompleteness. Those interfaces only provide querying XML data and browsing results. There is lack of support for exploring the entire database, editing, updating and deleting contents of databases. We address these issues in our visual user interface design.

6.2 NextDB User Interface

The NextDB user interface, DB Manager, bridges users and database to order to support interactive communication, to make changes conveniently and to get required data easily. The intuition behind the interface design is to hide the underlying physical storage structure and to give users an experience of working with XML documents. The design goals of the DB Manager include providing interactivity, usability and accessibility to both professional and casual users. Figure 6.1 depicts the system architecture of how DB manager works collaboratively with other database components in NextDB.

In the design stage, we mainly focus on the scenario that the user interface has to be as simple, interactive and well-functioned as possible to provide ease and efficiency to the users. Based on this scenario, we organize all database operations and data management functions into one integrated panel, the *main window*. For both users and database administrators, this window is the one and only one entrance to explore, navigate, and browse the contents of the databases. In addition, this panel equips user with complete tools to insert, update, modify and delete the contents.

We designed the working area in the main window with two parts; *Database Management Pane* and *Document Workspace*. The database management pane is an interface where users explore all databases and the contents such as XML documents, DTDs and query files. The interface supports users to browse all documents in each database, a corresponding DTD which is conforming to a set of documents and query files containing users' frequently-used query(s).

During the design stage of the *Document Workspace*, we deal with an issue that the interface must be simple and able to provide real XML browsing experience to the users, it means that, the users feel browsing and updating on an XML document regardless of how each document is fragmented and organized in the underlying physical storage. Since documents are represented by a hierarchical structure, the workspace must enable to represent the tree structure of documents graphically. In order to support such facility, we intuitively implement the document workspace as a graphical interface in which users can navigate each document and its elements in a native tree-style hierarchical format. That document tree allows users to insert a new element and content, update element, modify content, delete element and validate against predefined DTD.

We expand the capability of document workspace by integrating more functions such as visualizing and editing of DTD in tree-style editor, loading and viewing query from files, query submitting and result viewing. These functions can be done on the same space alternately according to the users' choice. This design simplifies the interface and provides integrated workspace to users to achieve ease and convenience during interacting with the database. Figure 6.2 shows the main user interface of

Figure 6.2: Main Interface of NextDB

NextDB.

All other user interfaces, except main window, that are provided in NextDB can be classified into two groups: *Data Management Interfaces* and *Interactive query interfaces*.

6.2.1 Data Management Interfaces

These interfaces allow users to perform data management operations such as creating, opening, importing and exporting database contents. Users can create a new or open an existing database, a DTD, an XML document, and a query file. In addition, users can also import and export XML document to and from the database.

Creating and Opening Database Contents

In NextDB, creating a new database is done on the initial screen. In this case, users have to provide a proper name for the database and select an appropriate DTD to

Figure 6.3: User Interface for Opening and Creating Database Contents

represent the schema for the database. Since the initial stage of NextDB is a schema-conscious database management system, a DTD or an XML schema is mandatory to be input in advance before importing any XML document into database. Similarly, the interactive interface allows user to create a new DTD, a new XML document, and a new query file. For users who want to access existing data, the interface supports the opening of existing database, DTD, XML document and query file. Figure 6.3 shows the initial screen for the creating and opening of database contents.

Importing and Exporting An XML Document

Typically, XML data is organized in the form of document. Before data is loaded into the database, these documents are necessary to validate and parse using a parser. The resulting parsed trees are then collectively organized and stored in the database. This process is called *importing XML document* in NextDB. Similarly, users may want to retrieve a document or a part of document from the database for the purpose

Figure 6.4: The Workflow of Importing and Exporting XML Data

of exporting or exchanging data with different organizations. This function can be done over exporting interface. Figure 6.4 illustrates the workflow of importing and exporting in NextDB.

We design both interfaces to facilitate users to import and export XML data interactively. In the importing, user provides an input XML document and the interface interacts with the user by displaying the number of elements that are loaded successfully into the database and the problems in parsing. In data exporting, the user is requested to select a document or a part of it in the hierarchical tree. DB Manager retrieves requested data from the storage and reformats it to be an XML document that is readily usable by the users. Figure 6.5 depicts the import and export interfaces in NextDB.

Editing DTD and XML Document

Some users prefer to create or edit a DTD or XML document in a text-based editor instead of using a tree-style graphic editor. We also take this issue into account in our design consideration. In order to address this, we have implemented a DTD editor and an XML editor. These editors facilitate the users to perform the following functions:

- create a new or viewing an existing DTD and XML document;
- visualize the structure of DTD and XML document;
- update and delete elements of a DTD or XML. document.

Figure 6.5: The Interfaces for Importing and Exporting

Figure 6.6 and 6.7 depict the interface of the DTD editor and Figure 6.8 shows the XML editor.

6.2.2 Query Interfaces

Querying and retrieving required data is the most frequently used operation in the database. When designing a query interface, we take note of the followings:

- It should be interactive with users.
- It must meet the specification defined in the XQuery language.
- It should communicate well with the query processor.
- It should be simple and it provide quick response to users.

Based on these specifications, we have implemented a text-based query interface, *query editor* with two main parts: *Query Preparation Pad* and *Result Viewing Pad*.

Figure 6.6: Text-Based DTD Editor

Creating, Opening and Executing Query

Creating a query can be done in query preparation pad. This pad allows users to input queries in XQuery language and validates and executes it by communicating with the query processor as shown in Figure 6.9. This pad also supports users to store their frequently used query or a group of queries in a file so that it can be used repeatedly. The upper panel in Figure 6.10 shows user query that is prepared in the query preparation pad.

Viewing Query Result

According to the XQuery specifications, users expect to see query results in XML format [Gro02]. We design the result viewing pad in order to meet this specification. After a query is executed, all fragmented results are organized and formatted in XML so that user easily understands the result. The lower panel of Figure 6.10 shows the query result in the resulting view pad.

We have implemented the entire user interface with numerous error handling procedures. For any particular function, the interface requests necessary input from

Figure 6.7: Graphics-Based DTD Editor

users, it demonstrates every processing stages, it shows processing status, results and errors, and ask users' confirmations for all important actions. It provides significant advantages in responding meaningful messages to all necessary user interventions, understandable messages for all types of error that can easily be recognized by users, and help for further debugging.

6.3 Usability Study Results

We conducted an initial usability study of the new interface with 15 XML users. This study focuses on three factors: the intuitiveness, the simplicity and ease of use, and the interactivity of the interface. The participants were students ranging from undergraduate to postgraduate in which 35% of participants are advanced users having high confidence in their XML knowledge and the rest were end-users who have fundamental knowledge of XML. We prepared a set of questions and asked for their

Figure 6.8: XML Editor

Figure 6.9: The Workflow of Querying Data in NextDB

suggestions and experience on exploring database over the integrated interface.

The overall results of the study are positive and showed that the interface is simple and easy to use, and that all menus and messages are understandable. 60% of the participants had a good impression on the interface and 30% is neutral but 10% suggested there are more rooms to improve. 75% of participants noted that they efficiently browsed and edited the database, DTD and XML documents over an integrated workspace and 25% said that they had some difficulties. Most users felt that the DTD and XML editors guided them intuitively to create a new document or update existing one. Almost all end-users commented that they could not efficiently

Figure 6.10: Query Interface

use the query editor as they did not know the XQuery expression very well. They suggested to provide more interactive query editor. On the other hand, all advanced users noted that the query editor is simple and functioned well but commented that there are more rooms for improvement.

After examining all users' comments and suggestions, we conclude that the interface is usable and efficient for most users but we still need to improve and simplify certain functions especially in query editor.

6.4 Summary

This chapter discussed issues in the stage of designing an visual interactive user interface for NextDB and its implementation. The NextDB's interface is an integrated one composed of various components such as database management pane, integrated document workspace, DTD editor, XML editor, importer, exporter and query editor.

It offers a rich variety of functions and operations and provides usability and accessibility to users. The results from the usability study are also conducted and it shows that the visual interface for NextDB is usable and efficient for most users. In future, we plan to improve the query editors interactivity with users. In the next chapter, we will present the architectural framework for native XML data management.

Chapter 7

Architectural Framework for Native XML Data Management

With the growth of XML, a scalable database management system becomes necessary to maintain those data efficiently. New databases that are specifically designed for storing XML data in its native format emerge as state-of-the-art solutions to overcome the deficiencies of conventional databases. NextDB is one of the native databases that has been designed to meet the special needs of XML. Timber [JAKC⁺02], Natix [KM00a], Lore [QWG⁺96], Tamino [Sch01] and DB4XML [SVC⁺02] are also native XML databases that have been introduced in recent years.

Unlike relational database systems, XML is new and all standardization activities are still in progress. The aforementioned state-of-the-art native database systems employ distinct data models, specific storage designs and different database architectures in order to provide efficient XML data management by their own way. Therefore, there is still lack of a general architectural framework that can be adopted and referenced by database designers when they design and develop new XML data management systems in future. This chapter proposes a new general architectural framework and an extended framework that are derived from our practical knowledge and experience on designing efficient XML storage system by preserving desirable features and possible constraints. At the end of introducing our framework, we will demonstrate the validity of the proposed framework with state-of-the-art native XML database systems.

The purpose of designing framework is not to propose a generic and comprehensive

architecture of native XML database system to manage all types of XML documents covering all possible database management features. Instead, we aim to provide a general concept on designing native XML database system that is derived from our practical experience and development. The discussion does not cover all aspects and issues arising from designing an entire database management system. Instead, the scope is limited to the architecture of native XML database system that preserves the hierarchical structure of XML documents [WNL03b].

7.1 Related Work

To the best of our knowledge, the introduction of architectural framework for native XML data management systems is relatively recent in the literature. The reason might be that XML technologies and standardizations are still emerging, and state-of-the-art native XML database systems are not mature enough to determine the complete reference architectural model. Unlike flat data in relational databases, XML imposes special characteristics and constraints for data management. Thus, the well-defined reference architecture used in relational systems are not completely relevant. A tailored architecture is necessary in order to meet the needs of XML characteristics. Our approach is similar to [Loc01]. However, we mainly focus on XML technologies. We start by identifying distinct characteristics of XML, desirable features and constraints for native XML data management system, and then propose a general as well as an extended architectural framework. Another article by Salminen and Tompa [ST01] identified and discussed requirements for XML document database systems. It does not focus on architecture. Instead, it primarily describes the desirable features of database components, the requirements of data definition and manipulation languages, document indexing, updating, user roles and access rights. In contrast, we view and analyze native XML data management system from architectural perspective.

7.2 Data Model and Design Methodology

Our motivation begins from the point that new database systems are independently introduced to address a common problem of XML data management, but there is still lack of a general framework to represent the general architecture of new generation database systems. Since data model and architectural design are playing an important role in the development of a new data management system, we need to take note on each topic.

7.2.1 The Importance of Architectural Design

In general, the architecture of a system is not only an overall framework of the system, it is also a description of the main system components, the interconnections among components, and interactions with another [Loc01]. In other words, the architecture of a system is a comprehensive framework that describes its form and structure, its components, and how they fit or tie together.

The architectural design comes first as a vital step in system development and plays a strategic role of identifying the desirable features and unavoidable constraints. In addition, architectural design is the means for identifying major problems, and for deciding possible solutions at a time when no major implementation is done. In practice, the analysis shows that many failures come from the lack of an explicit architectural design and implicit mistakes when the system is viewed from the architectural standpoint.

7.2.2 Importance of Data Model

A data model provides a collection of primitive state spaces and transition operators, and additional constructors that allow these to be grouped into more complex state spaces and transition procedures respectively. In formal terms, a data model can be viewed as a polymorphic type system in which operators and procedures correspond to the data operating functions [Loc01].

A data model is the core for a database system. A well-structured data model makes a database system capable and scalable. In practice, the distinct characteristics

and complex features of XML data create numerous challenges in designing an efficient underlying data model. Typically, XML data, which are organized as a document, can be represented as a *labelled directed graph* or *tree* in which all elements, attributes and values (contents) are linked using relations: *edges* or *labels*. Such a model is simple and sufficient to represent hierarchical structure of the document and to deal with low-end applications that use XML documents as an input. However, for a XML database management system designed for efficient data storage and retrieval for high-end and mission-critical applications, more scalable data model is fundamentally necessary.

Four different specifications from W3C: Infoset¹, XPath [Gro99a], DOM [Groa] and XML Query [Gro02] have already been defined for the structure of an XML document as a tree. Most of the current XML developments employ these data models as their reference. However, these specifications mainly focus on the logical model of data rather than viewing both logical and physical storage structure. The complete data model has not been developed yet in these specifications [ST01].

From this point, we can summarize that the data models defined in the W3C specifications are not completed; the architecture of current native XML database systems is still lacking of a general framework in terms of architectural design. We intend to bridge this gap by providing an architectural framework for XML data management.

7.3 Design of Architectural Framework

We define a two-step design methodology; the characteristics of XML data, desirable features, the constraints identified in the prior step, and the architectural design that best fits and preserves as many of the characteristics and desirable features as possible while minimizing the constraints determined in the subsequent step.

7.3.1 Desirable Features and Constraints

A problem in applying traditional database technologies to the management of persistent XML data lies in the special characteristics of such data including complex

¹<http://www.w3.org/TR/xml-infoset/>

units of information, consisting of formal and natural languages, and often including multimedia entities. The production and processing of such data in an organization may create a complicated set of variants, covering both data and metadata. In this section, we identify the dominant characteristics of XML, desirable features and constraints for designing data model.

Flexibility in Document

In general, XML documents can be classified into two main categories: *data-centric* and *document-centric*, based on their goals and concepts [NLB⁺02]. The documents falling into the first category are fairly structured containing fine-grained data. Each element, in such documents, contains meaningful information that can be used to be processed by applications. Such documents are normally created by specific programs for the primary purpose of data transport and exchange between different inter-applications. The sample documents subscribing to this category are sale orders, invoices, scientific data, etc. In such a document, the order of sibling elements is not important and significant.

In contrast, document-centric documents are less regular or irregularly structured containing large-grained data with mixed contents. Both the *implicit order*, which is the order of the elements within a document, and *explicit order*, which is expressed by attribute or tag in the document, are important in such documents [NLB⁺02]. They are usually designed for human consumption rather than information processing by the applications. User's manuals, novels and books are some example documents under that category.

In practice, the distinction between data-centric and document-centric documents is not clear. Due to the flexibility of XML, a data-centric document may contain large-grained, irregularly structured data. On the other hand, a document-centric document may consist of fine-grained, regularly structured data. In addition, the structures of these documents are unable to be predefined. In other words, the document might be fair or deeply nested.

According to the analysis on characteristic of XML documents, we can summarize

that the database system should have an efficient data model architecture which is able to capture both data-centric and document-centric documents, whether it is fairly or deeply nested, while preserving implicit and explicit order of elements within the documents.

Role of Schema

Tags in an XML document describe the structure of that document. In addition, XML provides document type definition (DTD) or XML schema together with the documents. A DTD is a pre-defined specification governing the structure of an XML document or documents. The DTD contains rules and constraints pertaining to elements and attributes. A current specification of XML Schema [Gro01] is an alternative solution to DTD. Like DTD, it provides a means of defining the structure, contents and semantics of XML documents. Besides, it also supports namespaces and data types of XML element. (Hereafter, we will use the term *XML schema* to represent both DTD and XML Schema).

When considering XML in the context of databases, it is tempting to treat a collection of documents as database instances, and associate XML schemas as database schemas. However, the meaning of XML schema, in this case, is different from well-understood definition of schema in database community. Firstly, XML documents may or may not be associated with a XML schema, i.e., schema is not mandatory in XML. Secondly, XML schema is primarily used to validate the document only when it is uploaded into the database or applications. As a result, it leads to the problem of additional elements or contents being inserted into the existing document that has already been allocated in the database. The challenges for the database designer is to determine how best to provide schema as part of the database instance rather than constraint mechanisms for input documents.

Another issue is that query processing in XML database should be accelerated by the use of XML schema if it is available. The appropriate schema should be assigned to the document in the database if it is unavailable. The reason is that schema plays a vital role in query execution because it lets the query processor know the structure

of documents in advance. It significantly reduces the time required to traverse along the document which is a frequent operation in XML query executions.

Analysis on the role of schema shows that the schema is necessary to be a part of the architecture of XML database systems. The desirable feature is that the data model should accommodate comprehensive schemas for both well-formed and valid documents so that it assures the reliability of data in the database and achieves faster query execution.

Data Retrieval Optimization

From a performance standpoint, the data organization and retrieval patterns determine the efficiency of data retrieval. Allocating an entire XML document as a whole into an arbitrary location of the physical storage is not a preferable storage mechanism since it does not provide efficiency when part of a document rather than an entire document is retrieved. For this case, the whole document is unnecessarily read and loaded from physical location to main memory in order to enable access to a small part of it. This operation is time consuming and makes the performance degraded.

There are two probable high and low-level approaches to address the problem: *structure transformation* and *segmentation*. As the name implies, the first high-level approach transforms the structure of input document into different forms of representation by using a certain criteria. In this case, the tree representation of a document is mapped into different logical structures in which related subtree or nodes are collectively organized in order to achieve faster data retrieval. This transformation demands a new layer between the logical and physical representations of a document.

The second approach focuses on the segmentation in which related data blocks or pages allocated in the low-level storage are clustered to form segments so that retrieving a group of related data can be performed without the need of massive I/O access. This feature is desirable in a data model in order to provide well-organized data structure and efficient retrieval pattern.

7.3.2 Framework for Data Management

Based on the analysis of the characteristics of XML, desirable features and possible constraints, we propose a general framework for a data model architecture as depicted in Figure 7.1.

Figure 7.1: The General Architectural Framework for XML Data Management

The proposed architecture of a database system can be viewed as complete in which both *logical* and *physical* layers are designed accordingly. Each layer is composed of different data models in which the representation of XML data, functions and data operators are also different.

Logical Layer

External Data Model. The external data model in this layer is very similar to the data models specified by W3C organization, such as the DOM data model. This

model views both XML schema and document as trees in which elements are treated as nodes in the tree connected by edges. Each node may contain its child nodes or text of any length or optionally an attribute/value list. A unique node identifier is assigned to each node so that one can be distinguished from another even when more than one node has the same node name. The parent-child relationship between a pair of nodes is also maintained. Thus, an entire tree can be traversed back and forth using the information. Validation of the document can be done through this model in order to preserve the persistence of data. The operators in this model perform query translation, view management and structural mapping.

Internal Data Model. Data in this model can be viewed in the form of transformed tree structure in which related nodes are collectively organized and logically clustered to form a group or groups according to a certain criteria. The properties of each node, such as node ID, its parent-child relationships etc., are still preserved in this model but the original structure of the document is transformed. The operators in this model are primarily tree and set operators as well as mapping functions.

Physical Layer

Physical Data Model. Physical records are the building cells of this model. Logical tree is transformed and allocated, in this model, into physical records. A record is of variable length but it is limited by the size of a physical block. Since schema is a desirable feature of the database system, a collection of XML schemas have to be accommodated in the different blocks. In addition, since long data such as text fields in the nodes are obstacles to clustering and indexing, these should be separated out and kept on other blocks so that the size of each record is smaller and the individual page could maintain primary objects of related nodes.

Segment. A segment is a collection of blocks in which related nodes are clustered in order to achieve faster data retrieval. In addition, buffer management is a frequent activity of this sub-layer.

Operating System File. File should form the lowest level in the architecture of a database system. All data are stored in files under the control of the operating system. The operating system tools can be utilized in this sub-layer to optimize the file structure.

The architectural framework proposed in this section highlights the harmonization between data models within or across layers; a vital function of a database system in order to provide data reliability, scalability and robustness.

7.4 Validation

It is impossible to validate the architectural framework the full generality. Instead, we restrict ourselves to an analysis of the architectures of state-of-the-art native XML database management systems. Among several research and commercial native databases, we have chosen four systems from the research area. The reason is that these systems describe the structure of their storage in detail.

7.4.1 Natix

Natix [KM00a] is a native storage system for XML document that maps, in the logical layer, an XML document into a tree in which all elements are mapped into nodes, attributes are mapped into child nodes, and PCDATA, CDATA nodes and comments are stored as leaf nodes. In the internal data model, the tree is transformed by partitioning into sub-trees and then stored in a single record which is less than a page in size. Figure 7.2 demonstrates tree partitioning in Natix.

Figure 7.2: Document Splitting in Natix

The resulting sub-trees in Natix is then materialized as a physical structure of large trees in the physical data model. Each record, in this layer, contains a pointer to the record containing the parent node of the root node of its subtree and the identifier of the document the contained subtree belongs. In Natix, nodes with long values (texts) are stored in separate blocks so that clustering over nodes with short values can be efficient. However, Natix does not concern much with the storage of schema.

The subtrees' records are maintained on data pages, which are based on a regular slotted page implementation that maintains a collection of variable-length records on a data page. These pages are finally managed by underlying operating system.

7.4.2 NextDB

NextDB [WNL04] is also an XML data management system implemented at the Nanyang Technological University. In NextDB, an XML document, in the logical layer, is represented as a DOM-like tree structure. Unlike Natix, NextDB transforms all input XML trees by organizing all nodes which have a unique structural property (i.e., same path expression) to form a node group in the internal data model.

Each node in a node group is then allocated in a variable-size record which is limited by the size of a page in the physical layer, and a node group forms a segment in which node records are collectively clustered. Detailed node record format can be seen in Figure 7.3. In NextDB, schemas that are associated with input documents are accommodated in separate block so that query processing and data retrieval can be accelerated by the use of database schema.

7.4.3 Db4XML

Db4XML [SVC⁺02] is a main-memory resident database that provides storage for XML document in native format. The input document in Db4XML is parsed, in the first step, to generate a DOM tree. Each element node in the tree is then mapped into an *element* object which is defined in logical data model.

These objects are allocated in variable-size records organized in physical storage.

Figure 7.3: Node Record Format in NextDB.

Figure 7.4 shows the record format of an element node.

Figure 7.4: The Record Format of an Element

Like NextDB, the schemas for input XML documents are maintained in the form of *ElementType* objects in Db4XML. The record format of an element in the schema is depicted in Figure 7.5.

Figure 7.5: The Record Format for Schema

7.4.4 TIMBER

TIMBER [JAKC⁺02] is a tree-structured native XML database which is based upon a bulk algebra for tree manipulation in order to store XML in its native format. Similar to other approaches, TIMBER takes XML document as input and produces a parse tree in the initial step. The resulting tree is transformed into internal data model,

and then stored as a collection of objects in the physical data model, called the *Shore* storage system.

In the physical storage, sub-elements are clustered to form a segment so that querying over sub-trees can be executed faster. The core of TIMBER does not depend on schema of XML data but TIMBER provides a metadata store in which schemas for XML documents are placed.

Thorough analysis on four native XML database systems shows that the different database system employ different storage formats and data access methods. However, the overall architecture of these systems are similar and covered under the framework which we proposed in the previous section. Findings from this analysis show that the proposed system can be used by database designers and researchers as their reference model for future XML database system developments.

7.5 Future Directions

The standardization of XML technologies are in progress and new specifications are still being introduced. This emergence imposes instability in architectural design and brings further demands for XML data management. The following characteristics and specifications should be considered as parts of a future architecture in XML database system.

- Typically, an XML document may contain XML components such as *prolog*, *comment*, *processing instruction*, *entity reference* etc. Current native database solutions have not clearly addressed how to manage them to preserve the contents of original documents.
- The specification of XML is extended with additional feature such as *Namespaces*. That should be accommodated in the database.
- An XML document may contain multimedia elements such as pictures, video and audio clips, animated icons, binary software application, etc. Future XML database systems should provide certain blocks to accommodate these objects and efficient accessing techniques.

- An XSL stylesheet² specifies the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary. The arisen issue is how to allocate stylesheet elements in the database.
- There is still a lack of development of a well-grounded data specification model (e.g., extensions to traditional E-R model) that can be used to capture the conceptual model of a database.

After addressing the issues to be considered for future architecture of XML data management system, we propose an extended framework as shown in Figure 7.6.

Figure 7.6: Future Architectural Framework for XML Data Management

The new architecture contains two new extended blocks for XSL elements and multimedia objects. They should be separately allocated from other data in order to

²<http://www.w3.org/Style/XSL/>

accelerate querying over data as well as multimedia objects. In this framework, we assume that *Namespaces* and other XML elements can be stored together with XML data.

7.6 Summary

The special characteristics of XML impose several problems and challenges when an XML data management system is designed. Different researchers employ different architectures when they develop native XML database systems. As a result, there is no general framework or reference architectural model although they all address a common problem. In this chapter, we presented the characteristics of XML, the desirable features and possible constraints for developing new generation XML data management systems. Based on our practical knowledge and experience, we proposed a general architecture of an XML database system. The validation is performed by analyzing the architecture of current state-of-the-art native databases. The result shows the proposed framework generally covers the architecture of these database systems, and can be employed as a reference model in future developments. Finally, we identify more XML specifications and vital issues to be considered in future native XML data management and also proposed an extended framework. We did not say that the proposed framework is generic and comprehensive but we hope it may support future XML database researchers as a reference model.

Chapter 8

Conclusions

The use of the Internet in electronic business is growing exponentially in recent years, and organizations engaged in such business are quickly adopting XML as a platform independent protocol for exchanging business information. XML is the breakthrough technology and easy-to-adopt method of encoding information in plain text that makes the flow of data easy and possible. The emergence of XML as a standard discipline for data representation and exchange leverages the publication of electronic data by providing a simple syntax for data that is both human and machine readable.

With the significant growth of XML data distribution, an efficient data model and a scalable database management system is necessary to manage such data. This requirement is treated as a driven factor for database community to introduce several variances of data management models. Most solutions use traditional databases such as relational, object-oriented, or hierarchical database as the underlying database to store XML data by employing third-party middleware or capabilities built in the database itself. Such databases fall into the category, named *XML-Enabled* databases [NLB⁺02].

The alternative solutions originate from designing a new data management system that is specifically tailored to meet the needs of XML data. Unlike traditional approaches, data management systems are customized by preserving the distinct characteristics of XML and possible constraints. The underlying storage structure may differ from one to another but all database systems took possible issues into account during design consideration. Such databases that employ their own data model and

architecture can be categorized as *Native XML Databases*.

As an emerging technology with tremendous improvements, we find that XML is an interesting field of study and a significant area with lots of opportunities and challenges for researchers. Our research work is not only to provide native storage techniques for XML documents but also to implement a complete native XML database management system, namely NextDB. Our work will contribute significantly to improving the storage strategy in XML technology and the database research area.

This chapter concludes the thesis by summarizing the key issues addressed, highlighting the major contributions in these areas and considering how the achievements can be built upon through future work for overcoming its present limitations as well as widening the scope.

8.1 Summary of the Thesis

The major contributions of our work can be summarized as follows:

- We studied the problems of existing XML storage systems, and then designed and proposed a new native XML storage model, namely ENAXS, that is based on a novel approach which is specifically tailored for managing large and complex XML documents. ENAXS takes advantages over similarity in structure of multiple XML documents and focuses on the scenario that all nodes are clustered according to their paths and collectively stored together in physical blocks so that the necessary space can be significantly optimized and the user queries can be resolved with less data loading and scanning. ENAXS is a fundamental building block of NextDB, a native XML data management system.
- We proposed a set of indexing schemes that are integrated into NextDB. The path index is implemented to support direct access to the desired nodes without requiring exhaustive tree traversal, and the schema tree is embedded in the index structure in order to deal with regular path queries. The value index is also added for efficient execution of queries with predicate over values.
- The third contribution of this thesis is the design and implementation of a query

processor to facilitate users for data retrieval. The query processor provides a set of data operators for querying the database, and manipulating, retrieving and restructuring XML data. In addition, it accelerates data retrieval in hierarchical nature and further explores to improves data representation capabilities and data interchange facilities.

- The forth contribution is the design and implementation of a transaction management system that is also a crucial component of an enterprise-level database that makes distributed computing possible. As the heart of transaction management, a new concurrency control protocol, namely XStamps, is designed based on Multiversion Timestamps Ordering Protocol (MTOP), and additional features are added to ensure the correctness and efficiency of concurrent operations in NextDB. XStamps maintains serializability among nodes along paths by preserving ancestor-descendent relationships. Aggressive access rules to regulate order among different transactions are also defined to assure database consistency. Other functional components such as transaction manager, resource manager and log manager are also integrated to provide a complete set of transaction management.
- A database system without a user interface is meaningless as users are not able to communicate with the internal database for data loading and retrieval. To address this issue, we implemented the DB Manager, an interactive user interface to provide intuitiveness, simplicity and usability. DB Manager supports communicating users with database interactively, making changes conveniently, and retrieving required data easily. In addition, it offers a rich variety of functions, and provides usability and accessibility to make users convenient.
- The integration of proposed database components into NextDB is also one of our contributions enable all components working together in synchronization. With the support of those components, NextDB provides a complete and scalable data management solution for XML data.
- The last contribution is the introduction of a general architectural framework

for XML data management that can be adopted and referenced by database designers in the future development of a new XML database management system. This framework is derived from our practical knowledge and experience on designing NextDB by preserving desirable features and possible constraints. An extended version of the architectural framework is also proposed to cover a wide area of XML features and upcoming standardizations.

8.2 Future Work

We initially focused on introducing a new storage repository and indexing scheme for XML data. However, we achieved significant progress in the research work and we later extended our goal to cover the design and implementation of a complete native XML database management system that is new and one of the most popular research areas for database researchers. In order to cope with the new scope of work, we have implemented a query processing engine and implemented new database features such as transaction management system, concurrency control and visual user interface for the database. As is the case with work in any emerging field, the thesis has a broad scope and each of the areas we have considered can be examined in greater details. Future work comprises either extending (through refinement or otherwise) previously taken approaches or broadening the scope further in new, unexplored directions.

- Despite the potential benefits and applications of the DB Manager and NextDB native database system, there are still several improvements that can be made to enhance the functionality and practicality of these modules. Some of these enhancements could lead to industrial applications of the software. Currently, the DB manager and the fundamental database engine are integrated together into a single module. This implementation does not allow modification on the implementation of one of the modules without adversely affecting the other. Thus, NextDB needs an enhancement to be partitioned into client portions and a server portion. The partitioning serves to help distribute the execution of the client program over several machines.

- To provide the efficiency of query execution, there is a need to enhance existing NextDB query processor by applying an effective query optimization techniques in order to boost query performance and to provide faster data retrieval.
- Some improvements to the user interface are also needed to make database functions more user-friendly and interactive. From the database perspective, NextDB provides plenty of enhancements that would increase the potential usefulness of the database system. Consequently, all of these implementations pose a need to further advance of the DB Manager.
- The current design of the database is schema-conscious, meaning that, a DTD or a XML schema that is conforming to a set of XML documents is mandatory to import XML data into NextDB. It is not a disadvantage but may seem inflexible for storing well-formed XML data coming from heterogeneous resources. Thus, an additional improvement is needed to be a schema-independent data management system.
- The recovery manager in NextDB assures resilience and integrity of database, and protects against any type of system failure. It still has much rooms to improve. The future work includes the integration and testing of this module into NextDB to improve the scalability and robustness.

Appendix A

List of Publications

The work in this thesis resulted in six papers (five accepted, one under review) over the past three years. The following list is the summary of our contributions in this thesis:

Journal, Conference and Workshop Papers:

- Khin Myo Win, Wee-Keong Ng, and Ee-Peng Lim. A Survey of XML Data Storage and Indexing Techniques. Proc. 1st Int. Conf. on Computer Applications, Yangon, Myanmar, January 2003.
- Khin Myo Win, Wee-Keong Ng, and Ee-Peng Lim. ENAXS: Efficient Native XML Storage System. Proc. 5th Asia Pacific Web Conference, X'ian, China, April 2003. LNCS 2642, pp. 59-70, 2003.
- Khin Myo Win, Emerald Rosasillfiani, Wee-Keong Ng, and Ee-Peng Lim. A Visual Interface for Native XML Database. Proc. 14th Int. Workshop on Database and Expert Systems Applications (DEXA'03) September 2003 Prague, Czech Republic.
<http://csdl.computer.org/comp/proceedings/dexa/2003/1993/00/19930571abs.htm>
- Khin Myo Win, Wee-Keong Ng, and Ee-Peng Lim. An Architectural Framework for Native XML Data Management. Proc. Int. Cyberworlds Conference, Singapore, December 2003.
<http://csdl.computer.org/comp/proceedings/cw/2003/1922/00/19220302abs.htm>

- Khin Myo Win, Wee-Keong Ng, Qincal Liu and Ee-Peng Lim. XStamps: A Multiversion Timestamps Concurrency Control Protocol for XML Data. Proc. 4th Int. Conf. on Information, Communications and Signal Processing, Singapore, December 2003.
- Khin Myo Win, Wee-Keong Ng, and Ee-Peng Lim. NextDB: Native XML Database Management System. Journal of Database Management System. (Submitted for publication).

References

- [ACM93] S. Abiteboul, S. Cluet, and T. Milo. Querying and Updating the File. In *Proc. Int. Conf. on Very Large Data Bases*, pages 73–84, Dublin, Ireland, 1993.
- [AKYJ03] S. Al-Khalifa, C. Yu, and H. Jagadish. Querying Structured Text in an XML Database. In *Proc. Int. Conf. on SIGMOD*, pages 4–15, San Diego, USA, June 2003.
- [AMS99] D. Alin, F. Mary, and D. Suciu. Storing Semistructured Data with STORED. *SIGMOD Record*, pages 431–442, 1999.
- [AQM⁺97] S. Abiteboul, D. Quass, J. MchHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Journal on Digital Libraries*, 1(1):68–88, April 1997.
- [AYS02] S. Amer-Yahia and D. Srivastava. A Mapping Schema and Interface for XML Stores. In *Proc. 4th ACM CIKM Int. Workshop on Web Information and Data Management (WIDM)*, pages 23–30, Virginia, USA, November 2002.
- [BBC⁺98] J. Bosak, T. Bray, D. Connolly, E. Maler, G. Nicol, C. M. Sperberg-McQueen, L. Wood, and J. Clark. W3C XML Specification DTD, 1998. <http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm>.
- [BCFR02] M. Benedikt, C. Y. Chan, W. Fan, and R. Rastogi. DTD-Directed Publishing with Attribute Translation Grammars. In *Proc. 28th Int. Conf.*

- on Very Large DataBases (VLDB)*, pages 838–849, Hong Kong, China, 2002.
- [BFRS02] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML Schema to Relations: A Cost-Based Approach to XML Schema. In *18th Data Engineering Conference*, pages 64–75, San Jose, California, February 2002.
- [BJBH01] C. Berkley, M. Jones, J. Bojilova, and D. Higgins. Metacat: A Schema Independent XML Database System. In *13th Scientific and Statistical Database Management(SSDBM) Conference*, pages 171–179, Fairfax, VA, USA, July 2001.
- [Bos] J. Bosak. XML Java and the Future of the Web.
<http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm>.
- [Bou] R. Bourret. XML Database Product. <http://www.rpbourret.com>.
- [CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *ACM SIGMOD Conference*, pages 313–324, Minneapolis, Minnesota, May 1994.
- [Car83] M. Carrey. Granularity Hierarchies in Concurrency Control. *Journal of Association for Computing Machinery*, 83(3):156–165, 1983.
- [CCD⁺99] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: A Graphical Language for Querying and Restructuring WWW Data. In *Proc. 8th Int. Conf. on WWW*, pages 1171–1187, Toronto, Canada, May 1999.
- [CDF⁺94] M. J. Carey, D. J. DeWitt, M. J. Frankl, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwillig. Shoring Up Persistent Applications. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 177–188, 1994.

REFERENCES

- [CDSS98] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your Mediators Need Data Conversion. In *Proc. Int. Conf. on Management of Data (ACM SIGMOD)*, Washington, USA, May 1998.
- [Cha95] V. Chaudhri. *Transaction Synchronization in Knowledge Bases: Concepts, Realization and Quantitative Evaluation*. Phd thesis, University of Toronto, 1995.
- [CM86] A. Croker and D. Maier. A Dynamic Tree-Locking Protocol. In *Proc. Int. Conf. on Data Engineering*, pages 49–56, Los Angeles, USA, February 1986.
- [CMS02] C. W. Chung, J. K. Min, and K. Shim. APEX: An Adaptive Path Index for XML Data. *ACM SIGMOD*, 4(6):505–516, June 2002.
- [CNM⁺01] B. F. Cooper, S. Neal, J. F. Michael, R. H. Gisli, and S. Moshe. A Fast Index for Semistructured Data. In *Proc. 27th Int. Conf. on Very Large Data Bases*, pages 341–350, Roma, Italy, 2001.
- [CPHK01] T. S. Chung, S. Park, S. Y. Han, and H. J. Kim. Extracting Object-Oriented Database Schemas from XML DTDs Using Inheritance. In *2nd EC-Web Conference*, pages 49–59, Munich, Germanys, September 2001.
- [DFF⁺98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. In *Query Language Workshop (QL 98)*, pages 201–212, December 1998.
- [Don88] K. Donald. *The Art of Computer Programming*. Addison Wisley, 1988.
- [DT03] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *Proc. 26th Int. Conf. On Very Large DataBases (VLDB)*, Berlin, Germany, 2003.
- [FHK⁺02] T. Fiebig, S. Helmer, C. C. Kanne, G. Moerkotte, J. Neumann, and R. Schiele and T. Westmann. Natix: A Technology Overview. In *Web and Database Related Workshops*, pages 12–33, Erfurt, Germany, 2002.

REFERENCES

- [Fis89] G. Fischer. Human-Computer Interaction Software: Lessons Learned. pages 44–52, January 1989.
- [FK99] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDBMS. In *Bullettin of the Technical Committee on Data Engineering*, 22(3):27–34, September 1999.
- [FKS⁺02] J. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita, and C. Wei. XTABLES: Bridging Relational Technology and XML. *IBM Systems Journal*, 41(4), 2002.
- [FM01a] T. Fiebig and G. Moerkotte. Algebraic XML Construction and its Optimization in Natix. *WWW Journal* 4(3), pages 167–187, 2001.
- [FM01b] T. Fiebig and G. Moerkotte. Evaluating Queries on Structure with Extended Access Support Relations. In *Proc. 3rd. Int. Workshop on WebDB 2000*, pages 41–46, Texas, USA, May 2001.
- [FMS01] M. Fernandez, A. Morishima, and D. Suciu. Efficient Evaluation of XML Middleware Queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 103–114, Santa Barbara, USA, May 2001.
- [FMST01] M. Fernandez, A. Morishima, D. Suciu, and W. C. Tan. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.
- [FOJ⁺98] B. Frank, D. Olivia, D. Jon, G. Joe, H. Dave, H. Paul, J. Trevor, J. Peter, H. K. Adrin, H. K. Kathy, M. Craig, and M. Stephen. *Learning XML*. Wrox Press Ltd., 1998.
- [FTS00] M. Fernandez, W. C. Tan, and D. Suciu. Silkroute: Trading between Relations and XML. In *9th WWW Conference*, pages 723–746, Amsterdam, The Netharlands, 2000.
- [Groa] W3C Working Group. Document Object Model. www.w3.org/TR/.

REFERENCES

- [Grob] W3C Working Group. Extensible Markup Language (xml).
<http://www.w3.org/xml/>.
- [Groc] W3C Working Group. HTML.
<http://www.w3.org/MarkUp/>.
- [Gro99a] W3C Working Group. XML Path Language (XPath), 1999.
<http://www.w3.org/TR/xpath>.
- [Gro99b] W3C Working Group. XSL Transformations (XSLT), 1999.
<http://www.w3.org/TR/xslt>.
- [Gro01] W3C Working Group. XML Schema: Formal Specification, 2001.
www.w3.org/TR/2001/WD-xmlschema-formal-20010320/.
- [Gro02] W3C Working Group. An XML Query Language (XQuery) Version 1.0, W3C Recommendation, April 2002. <http://www.w3.org/TR/xquery>.
- [GW97] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. 23rd Int. Conf. on Very Large DataBases (VLDB)*, pages 436–445, Athens, Greece, 1997.
- [GW98] R. Goldman and J. Widom. Interactive Query and Search in Semistructured Databases. In *International Workshop on the Web and Databases*, pages 52–62, 1998.
- [HKM01] S. Helmer, C. Kanne, and G. Moerkotte. Anatomy of a Native XML Base Management System. Technical report, University of Mannheim, 2001.
- [HKM04] S. Helmer, C. Kanne, and G. Moerkotte. Evaluating Lock-based Protocols for Cooperation on XML Documents. *SIGMOD Record*, 33(1):58–63, March 2004.
- [HT03] C. E. Hye and K. Tatsunori. XPath-based Concurrency Control for XML Data. In *Proc. on 14th Data Engineering Workshop*, Japan, 2003.

REFERENCES

- [JAKC⁺02] H. V. Jagadish, S. Al-Khalifa, A. Chapman, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *VLDB Journal*, 11(4):274–291, 2002.
- [JKG⁺99] S. Jayavel, T. Kristin, H. Gang, Z. Chun, D. David, and N. Jeffrey. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. 25th Int. Conf. on Very Large DataBases (VLDB)*, pages 302–314, Edinburgh, Scotland, 1999.
- [kjb02] KJV Bible Data set, 2002. <http://www.ibiblio.org/xml/examples/>.
- [KKR00] G. Kappel, E. Kapsammer, and W. Retschitzegger. X-Ray: Towards Integrating XML and Relational Database Systems. In *9th Conceptual Modeling Conference*, pages 339–353, Salt Lake, Utah, USA, October 2000.
- [Kle01] M. Klein. XML, RDF and Relative. *IEEE journal*, 16(2):26–28, April 2001.
- [KM00a] C. C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *16th Data Engineering Conference*, page 198, San Diego, CA, February 2000.
- [KM00b] M. Klettke and H. Meyer. XML and Object Relational Database Systems Enhancing Structural Mapping Based on Statistics. In *Int. Workshop on the Web and Database (WebDB)*, pages 151–170, Dallas, 2000.
- [LC00] D. Lee and W. W. Chu. Comparative Analysis of Six XML Schema Languages. *SIGMOD Record*, 29(3):76–87, 2000.
- [LC01] D. Lee and W. W. Chu. CPI: Constraints-Preserving Inlining Algorithm for Mapping XML DTD to Relational Schema. *Journal of Data and Knowledge Engineering*, 39(1):3–25, October 2001.
- [LK00] K. Loney and G. Koch. *Oracle 8i : The Complete Reference*. McGrawHill, 2000.

- [LMCC01] D. Lee, M. Mani, F. Chiu, and W. W. Chu. Nesting-based Relational-to-XML Schema Translation. In *Proc. Int. Workshop on the Web and Databases (WebDB)*, pages 61–66, Santa Barbara, USA, May 2001.
- [LMCC02] D. Lee, M. Mani, F. Chiu, and W. W. Chu. NeT & CoT: translating relational schemas to XML schemas using semantic constraints. In *Proc. 11th. Int. Conf. on Information and Knowledge Management*, Virginia, USA, 2002.
- [Loc01] P. C. Lockemann. Database System Architecture: A Study in Factor-Driven Software System Design. In *LNCS 2068: Proc. 13th CAiSE Conference, Interlaken, Switzerland*, pages 13–35, Springer Verlag, June 2001.
- [MAG⁺97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3):54–66, September 1997.
- [Meg] D. Megginson. Simple API for XML (SAX). <http://sax.sourceforge.net/>.
- [MHKM04] N. May, S. Helmer, C. Kanne, and G. Moerkotte. XQuery Processing in Natix with an Emphasis on Join Ordering. In *Proc. Int. Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, pages 49–54, Paris, France, June 2004.
- [MP00] K. D. Munroe and Y. Papakonstantiou. BBQ: A Visual Interface for Integrated Browsing and Querying of XML. In *Proc. 5th IFIP 2.6 Working Conf. on Visual Database Systems*, pages 277–296, Fukuoka, Japan, May 2000.
- [MSB03] H. Meuss, K. U. Schulz, and F. Bry. Visual Querying and Exploration of Large Answers in XML Databases with X^2 : A Demonstration. In *Proc. 19th Int. Conf. on Data Engineering (ICDE)*, pages 777–779, Bangalore, India, March 2003.

REFERENCES

- [MUW02] H. G. Molina, J. D. Ullman, and J. Widom. *Database Systems The Complete Book*. Prentice Hall, 2002.
- [NLB⁺02] U. Nambiar, Z. Lacroix, S. Bressan, L. L. Mong, and G. L. Ying. Efficient XML Data Management: An Analysis. In *Proc. Int. Conf. on EC-Web*, pages 87–98, Franch, September 2002.
- [NLH⁺98] W. K. Ng, E. P. Lim, C. T. Haung, S. Bhowmick, and F. Q. Qin. Web Warehousing: An Algebra for the World Wide Web. In *Proc. 3rd IEEE Int. Conf. on Advances in Digital Libraries*, pages 228–237, Santa Barbara, California, April 1998.
- [Pet97] B. Peter. Semistructured Data. In *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database System*, pages 117–121, Tucson, Arizona, United State, 1997.
- [PMW95] Y. Papakonstantinou, H. G. Molina, and J. Widom. Object Exchange Across Heterogeneous Information Source. In *Proc. 11th Int. Conf. on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.
- [QWG⁺96] D. Quass, J. Widom, R. Goldman, K. Haas, and et.al. LORE: A Lightweight Object REpository for Semistructured Data. *ACM SIGMOD*, 25(2):549–549, June 1996.
- [Ray01a] Erik T. Ray. *Learning XML*. O'Reilly and Associates, 2001.
- [Ray01b] M. Rays. Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Bulid Loosely-Coupled Systems. In *17th IEEE Data Engineering Conference*, pages 465–472, Heidelberg, Germany, April 2001.
- [RCF00] J. Robie, D. Chamberlin, and D. Florescu. Quilt: An XML Query Language. In *Proc. Int. Conf. on XML Europe 2000*, Paris, France, June 2000.
- [Ree78a] D. P. Reed. Naming and Synchronization in a Decentralized Computer System, September 1978.

- [Ree78b] D. R. Reed. *Naming and Synchronization in A Decentralized Computer System*. PhD thesis, Laboratory for Computer Science, MIT, MIT/LCS/TR-205, 1978.
- [Riz01] F. Rizzolo. ToXin: An Indexing Scheme for XML Data. Master's thesis, Department of Computer Science, University of Toronto, Canada, January 2001.
- [RLS98] J. Robie, J. Lapp, and D. Schach. XML Query Language(XQL). In *Query Language Workshop*, Cambridge, Mass, December 1998.
- [RM01] F. Rizzolo and A. Mendelzon. Indexing XML Data with ToXin. In *Proc. 4th Int. Workshop on the Web and Database (in Conjunction with ACM SIGMOD)*, pages 49–54, Santa Barbara, CA, May 2001.
- [Sch01] H. Schoning. Tamino: A DBMS Designed for XML. In *17th Data Engineering Conference*, pages 149–154, Heidelberg, Germany, April 2001.
- [sha02] Shakespeares' Data set, 2002.
<http://sunsite.unc.edu/pub/sun-info/standards/xml/eg>.
- [SK80] A. Silberschatz and Z. Kedem. Consistency in Hierarchical Database Systems. *Journal of Association for Computing Machinery*, 27(1):72–80, jan 1980.
- [SPS00] C. Stefano, F. Piero, and P. Stefano. XML: Current Developments and Future Challenges for the Database Community. In *Proc. 7th Int. Conf. on EDBT*, pages 1–17, Konstanz, Germany, March 2000.
- [SSB⁺00] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirshesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proc. 26th Int. Conf. on Very Large DataBases (VLDB)*, pages 65–76, Cairo, Egypt, September 2000.

- [ST01] A. Salminen and F. W. Tompa. Requirements for XML Document Database Systems. In *Proc. on ACM Symposium on Document Engineering*, pages 85–94, New York, November 2001.
- [Suc98a] D. Suci. An Overview of Semistructured Data. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 29(4):28–38, December 1998.
- [Suc98b] D. Suci. Semistructured Data and XML. In *Proc. Int. Conf. on Foundation of Data Organization*, pages 1–12, Kobe, Japan, November 1998.
- [SVC⁺02] S. Sipani, K. Verma, S. Chandrasekaran, X. Zeng, J. Zhu, D. Che, and K. Wong. Designing an XML Database Engine: API and Performance. In *Proc. 40th Annual Southeast ACM Conf.*, Raleigh, USA, April 2002.
- [TH] B. Travis and M. Hahn. HTML SGML PDF XML: What is the Difference.
<http://www.zvon.org/>.
- [TKHJ02] G. Torsten, B. Klemens, and S. Hans-Jorg. XMLTM: Efficient Transaction Management for XML Documents. In *Proc. 11th Int. Conf. on Information and Knowledge Management*, pages 142–152, VA, USA, 2002.
- [TVB⁺02] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. ACM SIGMOD*, pages 204–215, Madison, Wisconsin, USA, June 2002.
- [VV01] I. Varlamis and M. Vazirgiannis. Bridging XML-Schema and relational databases. A system for generating and manipulating relational databases using valid XML documents. In *Proc. ACM Symposium on Document Engineering*, pages 105–114, Atlanta, USA, November 2001.

REFERENCES

- [VVP⁺01] I. Varlamis, M. Vazirgiannis, P. Poulos, G. Akrivas, and S. Ioannou. X-Database. A middleware for collaborative video annotation, storage and retrieval. In *Proc. 8th Panhellenic Conference*, Cyprus, 2001.
- [WLD02] X. L. Wang, J. F. Luan, and Y. S. Dong. An Adaptable and Adjustable Mapping from XML Data to Tables in RDB. In *Proc. 1st VLDB Workshop on EEXTT and CAiSE*, pages 117–130, Hong Kong, China, 2002.
- [WNL03a] K. M. Win, W. K. Ng, and E. P. Lim. A Survey of XML Data Storage and Indexing Techniques. In *1st ICCA Conference*, pages 27–36, Yangon, Myanmar, January 2003.
- [WNL03b] K. M. Win, W. K. Ng, and E. P. Lim. An Architectural Framework for Native XML Data Management. In *2nd Cyberworlds Conference*, pages 302–309, Singapore, December 2003.
- [WNL03c] K. M. Win, W. K. Ng, and E. P. Lim. ENAXS: Efficient Native XML Storage System. In *5th APWeb Conference*, pages 59–70, Xian, China, April 2003.
- [WNL04] K. M. Win, W. K. Ng, and E. P. Lim. NextDB: Native XML Database Management System. *Journal of Database Management (submitted for publication)*, 2004.
- [WNLL03] K. M. Win, W. K. Ng, Q. Liu, and E. P. Lim. XStamps: A Timestamp, Multiversion Concurrency Control Protocol for XML Data. In *4th ICICS Conference*, page 90, Singapore, December 2003.
- [WRNL03] K. M. Win, E. Rosasillfiani, W. K. Ng, and E. P. Lim. A Visual Interface for Native XML Database. In *DEXA Workshop*, pages 571–575, Prague, Czech Republic, September 2003.
- [xma02] XMark Data set, 2002. <http://www.monetdb.cwi.nl/xml/>.
- [xml02] IBM XML 4J Parser, 2002. <http://www.alphaworks.ibm.com/tech/xml4j>.

REFERENCES

- [Yan82] M. Yannakakis. A Theory of Safe Locking Policies in Database Systems. *Journal of Association for Computing Machinery*, 29(3):718–740, 1982.
- [Zak02] R. H. Zakon. Hobbes Internet Timeline, 2002.
<http://www.zakon.org/robert/internet/>.
- [Zis00] A. Zisman. An Overview of XML. *Computing and Control Engineering Journal*, 11(4):431–442, August 2000.
- [ZMLR01] X. Zhang, G. Mitchell, W. C. Lee, and E. A. Rundensteiner. Synchronizing Internal Relational Storage with External XML Documents. In *11th Data Engineering Workshop*, pages 111–118, Heidelberg, Germany, April 2001.