THE UNIVERSITY OF CALGARY

Fragmenting XML Documents in Distributed XML Database Systems

by

Ying Qi Chen

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

NOVEMBER, 2003

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Fragmenting XML Documents in Distributed XML Database Systems" submitted by Ying Qi Chen in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]

Supervisor, Kenneth Barker, Department of Computer Science

[illegible]

Reda Alhajj, Department of Computer Science

[illegible]

External Examiner, Douglas Demetrick, Faculty of Medicine

Nov 28/03

Date

Abstract

Although Extensible Markup Language (XML) is becoming a standard for document processing and interchange on the Internet, it does have several shortcomings. One challenging problem is how to determine that the design of a XML document is “good”. In addition, since the size of a XML document is usually huge, the inherent redundancy becomes a prominent problem. Very little work has been proposed for designing a XML document that minimizes redundancy and has good structure. Therefore, a set of rules for XML document design is very desirable.

Further, the growing popularity of XML will lead to large repositories of XML data. Hence, we need more sophisticated ways to manage XML data. XML database systems are designed for this purpose. Relational database systems have been thoroughly investigated so they can be distributed. However, limited work has been done on a *distributed XML database system (DXDB).* Many mechanisms for conversions between relational databases and XML have been proposed so it is natural to consider applying relational techniques to XML document design. The question is “how?” and to what extent?

A DXDB, similar to a *distributed XML database system* (DDBS), must distribute data over different sites. No work has appeared in the literature on fragmenting XML documents in a distributed XML database environment. Based on our design model for XML documents, fragmenting a XML document becomes possible in a distributed XML database environment. Algorithms are proposed to fragment XML documents horizontally.

Acknowledgements

I would like to gratefully acknowledge the enthusiastic supervision of Dr. Ken Barker during the construction of this thesis. His brilliant insights, professional editing skills, and knowledge of the domain enabled me to craft this solid bit of research.

I thank Dr. Reda Alhajj for the discussions on the chapter of background and his proofreading on other chapters.

I am deeply indebted to my wife, Ying Wang and my daughter, Jia Rui for their understanding, endless patience and encouragement when it was most required.

I owe thanks to my colleagues in the lab who offered support in various ways during my research and writing of this thesis. Thanks to: Wendy, Quang, Jun, Angela, Bo, Steve, Chun Yan, and Yi Min.

I am also grateful to Ms. Lorraine Storey for her support during my graduate study.

I wish to thank my best friends in junior high school, Qing Yang, Ya Qin, and Wen Xia for helping me get through the difficult times, and for all the emotional support, entertainment, and caring they provided.

Lastly, and most importantly, I must thank my grandparents: Liang Hong and Xiu Ying, my parents: Yao Han and Bai Wei, my wife's parents: Zhe Xing and Xiu Yun, and my aunts: Wan Hua, Wan Fen, and Wan Ru. They raised me, supported me, taught me, and loved me. To them I dedicate this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction and Preview

1.1 Motivation

The Internet has expanded remarkably and vibrantly since the last decade of the 20th century. Undoubtedly the expansion of the Internet will accelerate due to the maturity of technologies such as the Extensible Markup Language (XML). Database systems are well known for their consistent database definition, construction, and manipulation of data used for electronic commerce. However, the heterogeneity of database systems makes data exchange and integration among different systems very challenging. Previous hard-coded HTML (Hyper Text Markup Language) approaches do not scale well to meet the future needs of the web [Bosa01]. XML helps overcome these problems and is rapidly emerging as a popular data format on the web.

XML documents are self-describing, which means that the relation between a document's content and its structure can be found within each XML document (see Figure 1.1). Similar to HTML, tags are also used in XML. However, tags in XML do not define the "formatting" of the content. Instead, tags are semantically related to the content enclosed within the open and close tags. Furthermore, XML documents can be validated by predefined schemas or DTDs (Document Type Definition) to standardize the format and content of XML documents.

HTML

```
<html>
  <body>
     <p>
        <h2>John Smith</h2>
        <h3>Dept. of IT</h3>
     </p>
  </body>
</html>
```

XML

```
<?xml version=”1.0” encoding =”UTF-8”?>
<employee>
     <name>John Smith</name>
     <department>Dept. of IT</department>
</employee>
```

Figure 1.1 HTML and XML Containing the Same Data

The growing popularity of XML will lead to large repositories of XML data. It is natural to consider using file systems to store XML documents. Unfortunately, apart from storing XML files, file systems do not provide any additional functionality to manage XML documents. Therefore we need more sophisticated ways to manage XML data. Currently, there are two main directions being investigated to build XML databases: XML enabled and Native XML databases. These are specified in detail in Chapter 2.

The Internet provides a computing environment that is heavily “networked”. Data distribution and integration have been studied intensively during the past three decades. The main reasons for having distributed computer systems are:

- Market forces: the use of distributed technologies is a must for almost all large and middle-size companies.
- Lower cost: A number of PC computers are cheaper and more powerful than one mainframe systems serving hundreds of terminals [Koss00].
- Increased scalability: adding a new network node becomes easier when responding to extensibility needs of the company.

- Increased availability: by replicating data over several sites, data is closer to the end user and more resistant to system failures.
- Improved performance: since each site only handles a portion of the database, contention for CPU and I/O is not as severe for centralized databases and localization reduces remote access delays that are usually involved in wide area networks.

A *distributed database system (DDB)* is an information system composed of a networked collection of multiple databases that are logically interrelated. A *distributed database management system (DDBM)* is a software facility that permits the management of the DDB and makes the distribution transparent to the users [ÖV99]. Data fragmentation is a challenging problem with which a DDB must contend and it is a part of the DDB design problem.

Relational database systems have been thoroughly investigated so they can be distributed. Since XML is more suitable for web data, we need to adopt XML databases for data storage. However, this does not mean that we are going to replace traditional databases with XML databases for all applications. We argue that XML databases provide robust storage and manipulation of XML documents. This thesis investigates one aspect at the construction of a *distributed XML database system (DXDB)*.

A DXDB, similar to a DDBS, must distribute data over different sites. However, no work has appeared in the literature on fragmenting XML documents in a distributed XML database environment. Furthermore, very little work has been proposed for designing a XML document that minimizes redundancy and has good structure.

In this thesis, an approach is proposed where relational techniques are applied to design a distributed XML document. An example of a company that has branches across different cities is used to demonstrate the approach. Clearly a distributed solution is required. Each site must manage a portion of the whole XML document according to the user's application needs. Since normalizing relational designs eliminates data redundancy efficiently, producing a normalized XML document should also eliminate redundancy. Further, to respect the nested structure of the XML document and enhance the performance when answering queries, denormalization is investigated based on the model proposed in Chapter 3. This thesis explores a new way to manage XML documents and benefits from both relational design theories and XML specifications. This is one of the first items, to the author's knowledge that studies designing and fragmenting XML documents in a distributed computing environment.

The next section describes the issues in designing a XML document. Section 1.3 discusses the requirements for XML databases. Section 1.4 presents the fundamental research issues of this thesis. Section 1.5 addresses an overview of the contributions and describes the organization of the rest of the thesis.

1.2 Design Issues In XML documents

Although XML is becoming a standard for document processing and interchange on the Internet, it does have several shortcomings. One challenging problem is how to determine that the design of a XML document is "good". XML is "free-form", i.e. a XML document is valid as long as it complies with the XML syntax. W3C (World Wide Web Consortium) set up the "well-formedness" criteria for XML documents, which state the conditions that need to be adhered to when creating XML documents. These are rules to define and control how the documents are created. However this recommendation does not specify the structure for XML document.

A well-defined database system is based on a well-defined data model. The relational data model is based on the mathematics of set theory. The strength of the relational approach to data management comes from the formal foundation provided by the theory of relations. Further, there are many design models available when designing a database, such as the Entity-Relationship (ER) and Object Models.

In contrast, the data model for XML is simple and flexible. A XML document is no more than a tree structure. In addition, this "tree" is ill-defined due to the lack of fundamental theory and design rules. The hierarchical data structure of XML cannot model a many to many relationship well. Further, redundancy is a substantial problem for XML documents. Unfortunately, we cannot change the nature of XML. Thus, to overcome these drawbacks of XML, we must constrain them by applying some design rules. Many mechanisms for conversions between relational databases and XML have been proposed so it is natural to consider applying relational techniques to XML document design. The question is "how?" and to what extent?

1.3 Requirements for XML Database Systems

As more XML data is used by different types of applications, there is a need to effectively manage the XML documents as a database. Salminen [Saml01] defines a *XML database* as "a collection of XML documents and their parts, maintained by a system having capabilities to manage and control the collection itself and the information represented by that collection". It is not just a repository of semi-structured data. Managing persistent XML data requires the ability to deal with data independence, integration, access rights, versions, views, integrity, redundancy, consistency, recovery, and the enforcement of standards.

1.3.1 The Data Model

Researchers in the database community have actively investigating XML [Vian01]. However, the need to integrate the management of structured documents with the management of other types of data makes the underlying data model very challenging.

The XML data model is often represented with a labeled tree [Chan+02] or directed graph [Kaus+02]. There are four different specifications proposed by W3C: the Infoset model [Cowa+01], the Xpath data model [Clar+99], the DOM model [LeHo+00], and the XQuery1.0 and XPath 2.0 Data Model [Fern+01]. Among these four models, only the XQuery 1.0 and Xpath 2.0 Data Model supports inter-document and intra-document links. An XML database should be built on a model that supports collections of inter-related documents, document fragments, and other related forms of data.

Data in relational database systems is managed through a three-level architecture separating the conceptual model from an internal model and a set of external models. Data independence relies on the principle that the conceptual model is isolated from the physical storage of the data. All applications must access the data through the external model. By applying these rules to an XML database the conceptual model incorporates not only all the objects and relationships to be modeled in the enterprise, but also all the document components available to any XML application.

Another issue for the data model of a XML database is *Document equivalence* [Raym+96]. For instance, before inserting a document into the database, we may want to know if the same document exists already. This issue is important for archiving, version management, metadata management, and query optimization. The XML 1.0 specification Infoset, Xpath, and DOM models do not define equality of documents or entities. The equality of node values is clarified in the Xquery 1.0 and XPath 2.0 model through an equality operator. However it does not cover all data in a XML document.

1.3.2 Data Definition

The XML specification defines nineteen primitive data types for an attribute and/or an atomic element. Like a conventional database, a XML database should have its own Data Definition Language (DDL) and Data Manipulation Language (DML) to define and manipulate each kind of data. To date, there is no generally accepted DDL or DML.

An XML database system should support collections of different document types (e.g., DTDs, XML Schema, XSLT, *etc.*) as well. Due to the diversity of user's needs and the continually evolving international and industry-level standards, the emerging XML database system must cope with this evolution.

1.3.3 Data Manipulation

In a XML database, the DML is used to compose queries upon data in the database, including entities, URIs, tags, comments, processing instructions, schemas and other metadata. However, currently available XML query languages, such as Lorel [Gold+99], XML-QL [XML-QL], XQL [Robi99], and XQuery [Cham+01] do not support access to entities, entity references, or notations.

The DDL for a XML database should allow for the definition of views that hide the physical structure of XML documents. Constructing a new document from the fragments of a collection of existing documents should also be supported in a XML database.

1.3 Data Distribution Issues in a Distributed Computer System

Özsu and Valduriez [ÖV99] argue "the design of a distributed computer system includes making decisions on the placement of data and programs across the sites of a computer network". Data distribution is a core issue in designing a DDBM.

There are three orthogonal dimensions along which the organization of distributed systems can be investigated: the level of sharing, the behavior of access pattern, and the level of knowledge on access pattern behavior. Figure 1.2 illustrates the alternatives along these dimensions.

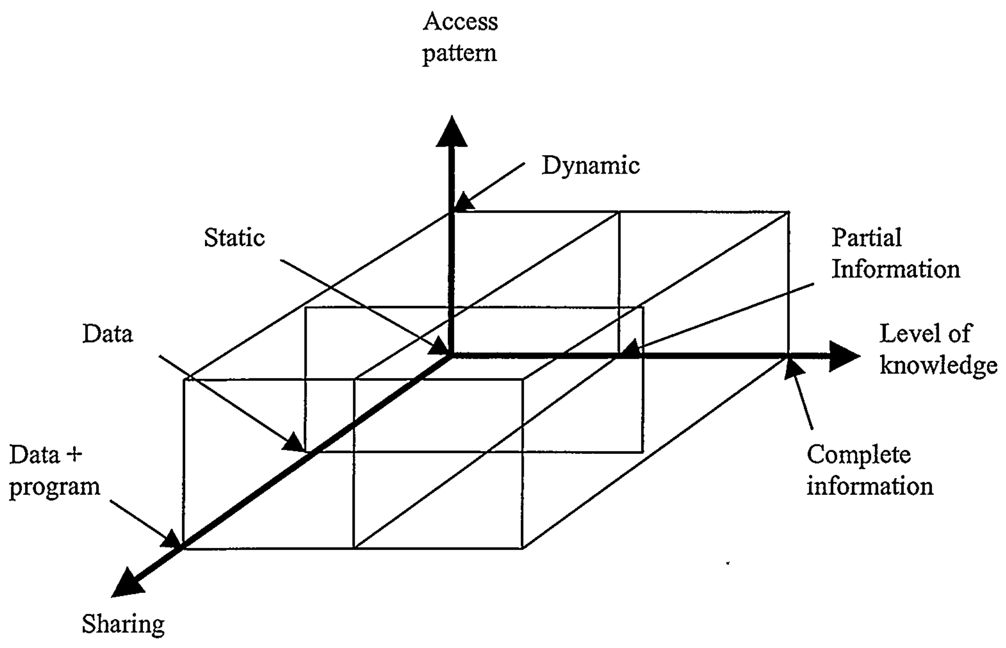


Figure 1.2 Framework of Distribution ([ÖV99])①

① Used with permission

The level of sharing has three options. First, there is *no sharing* when each site executes and accesses its own applications and data, and there is no communication among applications or data access to other sites. The *data sharing* level occurs when all programs are replicated at all sites but data files are not. Finally, in *data-plus-program sharing*, both data and programs may be shared, which means that an application at a given site can request a service from another program at a second site that may have to access data located at a third site.

In a distributed database system, a relation may be partitioned horizontally, vertically, or both into small fragments for distribution across sites. Since a distributed XML database system is a distributed computer system as well, it is natural and necessary to consider fragmenting the XML documents. Several algorithms [ÖV99] [NAVA95] have been proposed to fragment relations in the distributed database systems. However, to date this issue has never been considered for a DXDB.

1.4 Preview: Fundamental Research Issues

The challenges in document fragmentation in DXDBs are associated with the complexity of the distributed computing environment, the nature of XML, and the immature architectures for XML databases. These factors raise a number of issues. This section presents the key thesis elements and provides an overview of the key research issues in document fragmentation for DXDBs.

1.4.1 Thesis Overview

This thesis broadly addresses the document design problem in XML. Specifically, the document fragmentation problem in Distributed XML Database Systems is discussed. Issues of applying relational techniques to XML document design are identified. In particular, attention to horizontal fragmentation of XML documents is studied. Different architectural models for XML databases are discussed in the thesis. Further, based on our relational design models for XML documents, the procedure for designing a XML document is provided. Templates are provided to convert the design result to XML

Schema documents. Finally, algorithms to fragment a XML document horizontally are given.

The thesis uses the example specified in Chapter 4 for their fragmentation in a distributed database system. It includes four related entities each including data for employees, projects, subsidy, and assignment, respectively.

1.4.2 Key Issues

Although much research on XML technologies has appeared, none have addressed the issues of XML document design and fragmentation. This thesis initiates research on these two questions and addresses the following issues:

- Adapts relational techniques to XML document design.
- Proposes the relational design framework for XML documents.
- Identifies document fragmentation issues in distributed XML database systems.
- Provides algorithms to fragment XML documents horizontally.

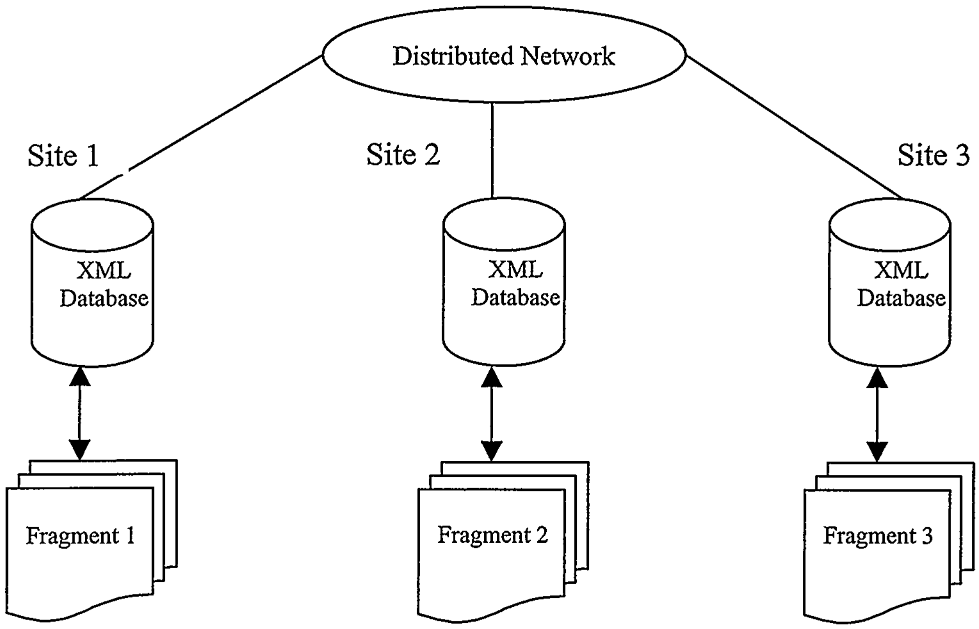


Figure 1.3 Fragmentation of a XML Document in a DXDB

Architectural Models for XML Databases

Several architectural models have been proposed for XML databases. Most are commercial products that store and manage XML data. Currently, there are two main categories: XML-enabled databases and Native XML databases. This section reviews currently available commercial XML storage systems.

XML-enabled databases hold data in some format other than XML. An interface is provided so that XML can be presented to an application even though the data is stored in some other format. An XML-enabled database could be a relational database, object-relational database, or an object-oriented database. Some object-relational mapping tools are also designed to work with XML.

Oracle XML SQL Utility (XSU) allows users to store XML by mapping it to an object-relational model. Mapping rules are embedded in the database model. The relational result is a collection of nested tables. XPath expressions can be nested within SQL queries. Oracle 9i supports XML-enabled databases. By providing SQL features implemented at the engine level, it allows user to view relational data as XML and XML data as relational data. A new data type: XMLType is added, which is a predefined object type that can store an XML document. A number of operators have been added to SQL to enable the conversion between XML and relational data. XMLType data can be stored in either of two ways: with object-relational storage or as a CLOB (Character Large Object). CLOB storage can round-trip XML documents exactly, while object-relational storage round-trips them at the level of the DOM. CLOB storage uses text indexes, while object-relational storage uses BTree indexes. Further, data stored with the object-relational mapping is directly available to non-XML applications, while data stored with CLOB storage can only be accessed by XML-aware applications.

Microsoft's SQL Server [Conr01] stores XML documents in three steps. First, a sp_xml_preparedocument stored procedure is invoked on source XML document to produce a DOM representation of the XML document stored in the memory. Second, mapping between relational tables and XML paths to atomic elements of DOM data is stored in relational tables. Finally, sp_xml_removedocument stored procedure removes the DOM from memory. Microsoft SQL Server 2000 supports XML in three ways: the FOR XML clause in SELECT statements, XPath queries that use annotated XML-Data Reduced schemas, and the OpenXML function in stored procedures. SELECT statements and XPath queries can be submitted via HTTP, either directly or in a template file. SQL Server has another tool called "Annotated XML-Data Reduced schemas", also known as mapping schemas. These schemas contain extra attributes that map elements and attributes to tables and columns. These specify an object-relational mapping between the XML document and the database, and are used to query the database using a subset of XPath. A tool exists to construct mapping schemas graphically.

Additionally there are a number of XML-enabled databases available currently such as ACCESS 2002 [Rice02], Cache, DB2 [Mala02], eXtremeDB [Mcob], among others. However, XML and SQL do not match on a number of levels. They use different data types and the free nested structure of XML documents does not mesh with a relational database's rigid table structures. Further, when an XML document is stored in a relational table, information can be lost, such as the element ordering and the distinction between attributes and elements. Hence, we consider using a native XML database.

Native XML databases allow XML data to be stored directly, which means populating a new database with the XML data. Native XML databases are likely to perform better than XML-enabled databases since ideally there is no need to convert the data. The data conversion in an enabled database is always more significant and time consuming than with a native database.

Sonic's eXtensible Information Server (XIS) [Sonic] uses XPath as its query language, which does not allow for joins or sorted query results. XIS uses an XML-based language for updates. It also offers Java- and COM-based programming APIs, database triggers, and XSLT transformations. Getting an XML database initially filled with data is especially easy because XIS can import data from any OLE DB or ODBC data source. XIS has separate index types for text and numeric data. Unfortunately, XML validation happens only once, when files are imported so later updates cause a mismatch in the DTD but XIS does not complain.

The Ipedo XML Database [Ipedo] is notable for supporting both XPath and XQuery. XQuery is more computationally complete than XPath, with a full programming language for writing queries, plus support for sorting, filtering, grouping, and joins. It also accepts XML Schema and DTD files to define database structures.

Adopting native XML database presents challenges too because relational data becomes hard (or impossible) to access and the software is immature in terms of data-integrity features, programmability, concurrency, and standardization.

Both XML-enabled and Native XML databases have critical drawbacks that make them uncompetitive with relational databases on many perspectives. We observe that a major problem with a XML document is the lack of design rules. To overcome this, this thesis applies the relational techniques to designing XML documents. A new architectural model for XML databases is proposed to deal with this type of XML document. A XML document designed in the relational framework combines good features from both XML and relational data.

Data Fragmentation in a XML Database

In the case of a distributed XML database system, XML documents must be fragmented and allocated across different sites to improve system performance and reliability. Designers may present the same data in different structures in a XML document. Thus, given an arbitrary XML document, it is impossible to find generic criteria to fragment it due to the flexibility of the nested structure of a XML document.

This thesis argues that, for a data-centric XML document, we may design it using techniques found in the relational model. XQuery1.0 and XPath 2.0 Data Model now supports joins across documents making our relational framework for XML more suitable to the design of data-centric XML documents. Since the data within a XML document in this design is presented in a relational way, the management system may adopt mature techniques and algorithms used in relational databases to handle the data.

Furthermore, we modify existing algorithms that partition relational tables to fragment XML documents in the relational framework. In this thesis, horizontally fragmenting a XML document is discussed. A treatment of data fragmentation for XML is the topic of Chapter 4.

1.5 Contributions and Structure of Thesis

In this thesis, the problems posed by the above issues and the solutions to address them are analyzed. First, we note that a set of rules for XML document design is very desirable. A good XML system is built based on solid design principles. Document design is the fundamental part for the design of a XML system. However, except for the *well-formedness* specified in [XML1.0], there is no criteria to evaluate what constitutes a good design for a XML document. In addition, since the size of a XML document is usually huge, the inherent redundancy becomes a prominent problem. This thesis provides a very generic design for XML documents by applying relational design techniques. Both bottom-up and top-down designs for XML documents are proposed. Second, the main drawbacks of current XML database products are identified. These drawbacks are primarily due to the ill-defined nature of XML documents. Third, procedures for designing a XML document in the relational framework are proposed. Normal forms and a relational framework for XML documents are defined. Finally, based on our design model for XML documents, fragmenting a XML document becomes possible in a distributed XML database environment. Algorithms are provided to fragment XML documents horizontally. Further this thesis considers the nested structure of XML and the feature of XML applications resulting in a combination of relational techniques and XML technologies. Algorithms for fragmenting XML documents with nested elements are presented. The fragmentation examples used in this thesis have been coded using XML, XPath, and VbScript. It must be noted that the implementation of this thesis provides an abstract framework that can be effectively utilized by any distributed XML applications.

The balance of this dissertation is organized as follows: Chapter 2 provides the necessary background and related work in the area of XML and distributed design. Chapter 3 gives the formal model of the relational framework of XML. Based on this model, the procedure for designing a XML document is provided. Chapter 4 addresses the data fragmentation issues in XML and presents algorithms to fragment XML documents conforming to the relational framework. Chapter 5 presents algorithms for fragmenting XML documents with elements nested to a certain level.

Finally, in Chapter 6, a summary of our work and its contributions is provided. A discussion about XML document design and data fragmentation issue in XML by applying relational techniques is addressed. Chapter 6 concludes by providing an outline of future research directions.

Chapter 2

Background and Related Work

This chapter introduces the reader to the necessary background and related research work in XML technologies. Section 2.1 provides an introduction to XML (extensible Markup Language). In Section 2.2, we discuss data fragmentation in distributed database systems. Section 2.3 presents various XML databases. Section 2.4 concludes this chapter with a summary.

2.1 XML (eXtensible Markup Language)

2.1.1 What is XML?

EXtensible Markup Language (XML) is a markup language capable of describing data. However, unlike HTML, its tags are not predefined, which means you must define your own. XML is "self describing" but can use a DTD (Document Type Definition) or XML schema to formally specify valid data.

XML is not a replacement for HTML as it has different design goals. XML is designed to describe data and to flexibly represent it. In contrast, HTML is designed to display data on a web browser. Thus HTML displays information, while XML describes information.

The tags used to markup HTML documents and the structure of HTML documents are predefined. The authors of HTML documents can only validly use tags that are predefined. XML allows authors to define their own tags and their own document structure.

XML documents must be *well-formed.* This means that they do not have to be created using predefined structures, but must comply with XML *constraint*s. These constraints require that elements, which are named content containers, properly nest within each other and use markup syntax correctly. Well-formed XML elements are defined by their use, not by a rigid structural definition, allowing authors to create elements in response to their individual needs. This flexibility offers authors greater control over document processing and design than in traditional SGML environments, in which structure must be formally defined in a DTD before any documents can be written.

XML frees Web authors from the predefined tags that characterize the fixed nature of HTML. HTML cannot be expanded or altered so its description power is constrained. For example, authors can describe documents in XML using their own names, such as ESSAY, SECTION, PARAGRAPH, NOTE, and IMPORTANT. After writing the document, the author can change an instance of the PARAGRAPH element to a different tag such as TAKENOTICE to signify that that instance differs from and contrasts with the rest. The flexibility of XML allows authors to describe documents as they see fit. Authors can publish XML documents with XSL, CSS, or DSSSL style sheets, which provide Web browsers and conversion tools with styling information for each element so documents are expressed in desirable ways.

The Web will likely utilize XML to structure and describe web data, while HTML will be used to format and display the data. In the future XML is likely to be used for data transmission and manipulation over the web.

XML's potential impact is significant, for Web servers and applications encoding their data in XML quickly makes their information available in a simple and usable format and such information providers can interoperate easily.

It is also worthwhile to note that XML differentiates between information content and information rendering (using XSL eXtensible Stylesheet Language), indeed reducing the effort in extracting usable data, as opposed to HTML, where one would have to follow laborious and error-prone methods such as screen scrapping to extract useful information. Further XML is an evolving standard and is actively pursued and promoted by key industrial players.

2.1.2 Different ways to use XML

Computer and database systems often contain data in incompatible formats. A critical costly challenge is to exchange data between heterogeneous systems over the Internet. Converting the data to XML can greatly reduce this complexity and create data that can be read by different types of applications.

XML can be used to store data in flat files or databases. Applications can then be written to store and retrieve information from these repositories as needed.

2.1.3 XML Syntax

Before exploring XML deeply, it is essential to understand a basic XML document. This section briefly introduces the syntax of XML.

2.1.3.1 Element

Elements are used to model structured data and are encoded using start and end tags. Elements typically make up the majority of the content of an XML document. Elements can have children, which can themselves be elements or may be processing instructions, such as comments, CDATA sections, or characters. Elements begin with an open tag and end with a close tag as follows:

```
<tagname>content</tagname>
```

The children of an element are enclosed between the open and close tags of their parent. For example, an element called 'parentelement' with a single child element called 'child' that contains no children itself is presented as follows:

```
<parentelement>
        <child>Hello!</child>
</parentelement>
```

2.1.3.2 Tag names

XML does not have a fixed vocabulary of predefined element names but allows element vocabularies to be invented. When designing new vocabularies, element names that convey some semantic meaning to human readers should be used. While the majority of XML documents will be generated and consumed by machines, having element names that mean something aids in human readability during the development phase; especially when debugging.

Element names in XML are case sensitive and must begin with a letter or an underscore (_). The initial character can be followed by any number of letters, digits, periods (.), hyphens (-), underscores (_), or colons (:). Element names beginning with "xml" are reserved by the XML specification for future use. The following elements all have invalid names:

```
<xmlfoo />
<XMLfoo />
<XmlFoo />
```

2.1.3.3 Namespaces and Namespace Declarations

To distinguish between element names in different XML vocabularies, the Namespaces in the XML Recommendation [XML1.0] provides rules for defining and using namespaces to disambiguate names in XML documents.

The possibility of name collision is quite high because XML was designed to allow the creation of new vocabularies. This makes it difficult for software (and humans) to know how to unambiguously process a given XML document. Consider an example: Software developer A decides to model a person in XML and needs to store the name, age, and height for the person, and to use element names of ***name***, ***age***, and ***height*** for those data items. An example instance of such a person would look like this:

```
<Person>
        <name>Mary</name>
        <age>32</age>
        <height>64</height>
</Person>
```

Software developer B decides to model a person in XML and needs to store the same data items and settles on the same element names. However, when describing the same person as in the previous example, B's instance looks like this:

```
<code xml:space='preserve'>
<Person>
        <name>Mary</name>
        <age>20</age>
        <height>162</height>
</Person>
</code>
```

The element names of A and B's instance are the same, but in the case of ***age*** and ***height*** the values contained between the open and close tags are different. The reason is that A used base 10 for age and gave height in inches, whereas B used base 16 for age and gave height in centimeters. XML namespaces provide a way to unambiguously distinguish between different XML vocabularies.

Using namespaces in XML involves associating element names with a Uniform Resource Identifier (URI). This URI serves as a unique string and forms the namespace name. The namespace name acts as a scope for all elements that are associated with the namespace. An element is associated with a namespace through a combination of a namespace declaration and a namespace prefix. The namespace declaration defines the prefix that represents the namespace URI. The prefix is then pre-pended to the element name. Namespace declarations take the form 'xmlns: prefix="URI"' and appear inside the element start tag. For example, the following XML shows a ***Person*** element in the 'http://www.cpsc.ucalgary.ca' namespace.

- <r:Person
  xmlns:r='http:// www.cpsc.ucalgary.ca/~yingqi/xml'></r:Person>

The prefix in this example is 'r'. Note that the closing tag must use the same name, including the prefix, as the opening tag. Note also that the prefix used is arbitrary. The name of namespace-qualified elements is made up of two parts: the prefix and the local name. In this example, the prefix is 'r' and the local name is 'Person'. This (prefix:local name) construction is known as a Qualified Name (Qname). Each part of a QName is a non-colonized name. Colons are used to separate the namespace prefix from the local name so element names containing colons must be avoided.

It is possible to omit the prefix and the colon but still associate an element with a namespace using a default namespace declaration. This takes the form of 'xmlns="URI"'. For example, <Person xmlns='http://www.cpsc.ucalgary.ca'/>, defines an element that is semantically equivalent to the previous example.

2.1.3.4 Scope of namespace declarations

Namespace declarations come into scope at the defining element and apply to all descendants unless overridden by a namespace declaration on a descendant. All namespace declarations have a scope (that is a set of elements to which they apply). A namespace declaration is in scope for the element it is declared at and all of that element's descendants.

2.1.3.5 Attributes

Elements can be annotated with name/value pairs known as attributes. Attributes are typically used to encode metadata; that is, they provide extra information about the content of the element on which they appear. The attributes for a given element are serialized inside the start tag for that element. Attributes appear as name/value pairs separated by an equal sign (=).

Attribute names have the same construction rules as element names. Attribute values are textual in nature and must appear either in single (') or double quotes ("). Attribute values may not contain the literals less-than (<) or ampersand (&) characters.

Attributes respect namespaces in the same way as the elements. The only difference between elements and attributes with respect to namespaces is that the default namespace declaration does not apply to attributes. This means that attributes without prefixes are always in 'no namespace' even if a default namespace declaration is in scope.

The following example illustrates namespace-qualified attributes:

```
<Person xmlns='http://www.ucalgary.ca/~yingqi/xml'
          xmlns:b='http:// www.ucalgary.ca/~yingqi/xml/base'
          xmlns:u='http:// www.ucalgary.ca/~yingqi/xml/units' >
           <name>Mary</name>
           <age b:base='10'>32</age>
           <height u:units='inches'>64</height>
</Person>
```

The following example illustrates attributes in ‘no namespace’:

```
<Person xmlns='http:// www.ucalgary.ca/~yingqi/xml'>
           <name>Martin</name>
           <age base='10'>32</age>
           <height units='inches'>64</height>
</Person>
```

2.1.3.6 Text

Certain character literals are illegal inside element and attribute content. XML provides several standard character entities for encoding these characters along with character references and CDATA sections.

Five character literals (<, >, &, ', and ") have certain limitations in terms of where they can legally appear in an XML document.

Certain characters cause problems when used as element content or inside attribute values. Specifically, the less-than character (<) cannot appear either as a child of an element or inside an attribute value as it is interpreted as the start of an element. The same restrictions apply to the ampersand character (&) although for different reasons. If the less-than (<) or ampersand (&) characters must be encoded as an element child or inside an attribute value then a character entity must be used. Entities begin with the ampersand character (&) and end with a semicolon (;). The name of the entity appears between the two. XML defines entities for the less-than character (<) and the ampersand character (&) as &lt and &amp, respectively.

The apostrophe (') and quote characters (") might also need to be encoded as entities when used in attribute values. If the delimiter for the attribute value is the apostrophe, the quote character (") is legal; but the apostrophe character (') is not, as it would signal the end of the attribute value. If an apostrophe is needed the character entity, &apos; should be used. For example,

```
<sayhello word='&amp;apos;Hi&amp;apos;' />
```

would result in the following being displayed or parsed:

```
<sayhello word='&apos;Hi&apos;' />
```

A fifth character reference (&gt) is also provided for the greater-than character (>). While such characters seldom need to be escaped, it provides a nice symmetry with the less-than character (<).

CDATA sections allow markup characters to appear as literals without being interpreted as markup. Using entities in place of less-than (<), greater-than (>), ampersand (&), apostrophe ('), and quote (") characters can become tedious and error-prone if a significant number of those characters appear in textual data. XML provides a construct called a CDATA section that allows such characters to appear as literally.

A CDATA section begins with the character sequence (<![CDATA[) and ends with the character sequence (]]>). Between the two character sequences a XML processor ignores all markup characters such as less-than (<), greater-than (>), and ampersand (&). The only markup an XML processor recognizes inside a CDATA section is the closing character sequence (]]>). The following XML shows a CDATA section containing literal less-than (<), greater-than (>), ampersand (&), apostrophe (')' and quote (") characters.

```
<sometext>
<![CDATA[ They're saying "x < y" &
 that "z > y" so I guess that means that
z > x ]]>
</sometext>
```

Comments are used to communicate information about the content of an XML document. They often contain documentation about the structure or content of the XML in which they are found. XML also supports comments that are used to provide information to humans about the actual XML content. They are not used to encode actual data.

Comments can appear as children of elements or of the document. They begin with <!-- and are terminated with -->. Textual data is serialized between the two constructs. For compatibility with SGML the character sequence -- cannot appear inside a comment. Other markup characters such as less-than (<), greater-than (>) and ampersand (&) can appear inside comments, but are not treated as markup. If the textual content of the comment ends with the hyphen (-) character, there must be some whitespace between the hyphen and the close comment character sequence (-->).

2.1.3.7 Processing instructions

Processing Instructions (PI) are information for the application. PIs allow documents to contain instructions for applications. They are ignored by the XML parser. Instead, the instructions are passed to the application using the parser because the purpose of processing instructions is to represent special instructions for the application.

Processing instructions are composed of two parts: the target or name of the processing instruction and the data or information. The target is preceded by the character sequence (<?). The target is followed by a whitespace and then the data portion of the processing

instruction. The data portion is textual and can contain whitespace. The processing instruction is terminated with the character sequence (?>).

Apart from the termination character sequence (?>) all markup is ignored, in processing instruction content. Processing instructions defined by organizations other than the W3C should not have targets that begin with the character sequence (xml). The following are all valid processing instructions:

```
<?display table-view?>
<?xml version="1.0" encoding="UTF-8" ?>
```

Processing instructions can appear as children of elements or as top-level constructs (children of the document) either before or after the document element.

Processing instructions are not specified by the Namespaces in the XML recommendation and the target portion of a processing instruction is not part of any in-scope namespace. This raises the possibility of collision among processing instruction targets between applications. However, namespace qualified elements can be used instead of processing instructions.

2.1.3.8 Version declaration

Version declaration, as a type of Processing Instruction, is information for the application. XML documents start with an **XML version declaration (XML declaration)**, which specifies the version of XML being used. Although the XML version declaration is optional, it is recommended by the W3C specification. The XML declaration is a processing instruction that notifies the processing agent that the following document has been marked up as an XML document. It appears as follows:

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

The version declaration can also contain other information such as an encoding or stand alone declaration.

Encoding declarations inform the processor of the kind of code the document uses (e.g. UFT8, which is the ASCII character set). All XML parsers support 8-bit and 16-bit Unicode encoding corresponding to ASCII. However, XML parsers may support a larger set. A list of encoding types is available [XML1.0].

```
<?xml version="1.0"? encoding="UTF-8"?>
```

Standalone declarations tell the processor whether the document can be read as a standalone document, or if it needs to look outside the document for the rules. Thus, if standalone is set to "yes", there will be no markup declarations in external DTDs. Setting it to "no" leaves the issue open. The document may or may not access external DTDs.

```
<?xml version="1.0"  encoding="UTF-8" standalone="yes"?>
```

2.1.4 Document Type Definition (DTD)

A DTD is a set of rules that defines what tags appear in a XML document; what attributes the tags may have; and what relationship the tags have with each other. When an XML document is processed, it is compared to the DTD specialization to ensure it conforms to the correct structure and all tags are used properly. This comparison process is called *validation* and is performed using a parser. Example for a DTD:

```
<!DOCTYPE course [
<!ENTITY COM "Computer Science">
<!ENTITY SOF "Software Engineering">

<!ELEMENT course (name+, department+, year+)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST name
        xml:lang NMTOKEN "EN"
        id ID #IMPLIED>
<!ELEMENT department (#PCDATA)>
<!ELEMENT year (#PCDATA)>
]>
```

Recall that a DTD is only needed to validate a XML document.

2.1.4.1 Internal DTDs

Internal DTD (markup declaration) are inserted within the doctype declaration. DTDs inserted in this way are used for the specific document in which it is found. This might be a suitable approach when a small number of tags in a single document occur:

```
<!DOCTYPE course [
<!ENTITY COM "Computer Science">
<!ENTITY SOF "Software Engineering">

<!ELEMENT course (name+, department+, year+)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST name
        xml:lang NMTOKEN "EN"
         id ID #IMPLIED>
<!ELEMENT department (#PCDATA)>
<!ELEMENT year (#PCDATA)>
]>
<course>
        <name id="471">Database Design</title>
        <department>&COM;</department>
        <year>2001</year>

        <name id="695">Web Application Design</title>
        <department>&SOF;</department>
        <year>2001</year>
</course>
```

2.1.4.2 External DTD

DTDs can be very complex so creating a DTD may require substantial expertise. DTDs are stored as ASCII text files with the extension '.dtd'. In the following example we assume, that the previous internal DTD is saved as a separate file (under the name course.dtd), and is therefore referred as an external definition (external DTD):

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

<!DOCTYPE course SYSTEM "course.dtd">

<course>

        <name id="471">Database Design</title>

        <department>&COM;</department>

        <year>2001</year>
        <name id="695">Web Application Design</title>
        <department>&SOF;</department>
        <year>2001</year>
</course>
```

2.1.5 XML Schema

Schemas, like *Data Type Definitions (DTD)*, define the elements that can appear in an XML document and the attributes that can be associated with those elements. The XML Schema language is also known as the XML Schema Definition (XSD).

Schemas define the document's structure - which elements are children of others, the order the child elements can appear, and the number of child elements. Schemas specify if an element is empty or if it must include text. They can also specify default values for attributes. Schemas are more powerful and flexible than DTDs and use XML syntax.

Independent developers can use a common schema to exchange XML data. A new application can use this agreed upon schema to verify the data it receives. Verifying an XML document against the schema is known as validating the document.

Schema standards are defined by the World Wide Web Consortium [W3C]. The W3C provides a comprehensive reference for XML schemas. An overview of XML Schemas is presented here.

2.1.5.1 The Schema element

The <schema> element is the root element of every XML schema. It may contain attributes as shown in Example *person.xsd.* The fragment on line 3 indicates that the elements and data types used in this schema (schema, element, complexType, sequence, string, boolean, *etc.*) come from the "http://www.w3.org/2001/XMLSchema" namespace. It also specifies that the elements and data types that come from the "http://www.w3.org/2001/XMLSchema" namespace should be prefixed with “xs:”

Example person.xsd

```
1       <?xml version="1.0"? encoding = "UTF-8">
2       <xs:schema
3               xmlns:xs="http://www.w3.org/2001/XMLSchema"
4               targetNamespace="http://www.cpsc.ucalgary.ca/~yingqi/xml"
5               xmlns="http:// www.cpsc.ucalgary.ca/~yingqi/xml "
6               elementFormDefault="qualified">
        ...
        ...
        </xs:schema>
```

Line 4 indicates that the elements defined by this schema come from the "http:// www.cpsc.ucalgary.ca/~yingqi/xml" namespace. The default namespace "http:// www.cpsc.ucalgary.ca/~yingqi/xml" is specified on line 5. Line 6 indicates that any elements used by the XML instance document that were declared in this schema must be namespace qualified.

The following example shows how a XML document refers to a XML Schema.

```
1       <?xml version="1.0"?>
2       <person xmlns=" http://www.cpsc.ucalgary.ca/~yingqi/xml "
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation=" http://www.cpsc.ucalgary.ca/~yingqi/xml sample.xsd">
5             <name>Mary</name>
6             <age>20</age>
7             <sex>female</sex>
8       </person>
```

From line 2 we know that the default namespace is "http://www.cpsc.ucalgary.ca/~yingqi/xml". This declaration tells the schema-validator that all the elements used in this XML document are declared in the "http://www.cpsc.ucalgary.ca/~yingqi/xml" namespace. Line 3 indicates the XML Schema Instance namespace so that you can use the attribute schemaLocation on line 4.

2.1.5.2 Simple types

A simple XML element can only contain text so it cannot have other elements or attributes. However, the text can be of many different types. It can be any of the types included in the XML Schema definition (boolean, string, date, *etc.*), or it can be a custom type.

To limit element content, restrictions (facets) can be applied to a data type, to ensure the data matches a defined pattern. XML Schema has a several built-in data types:

- string
- decimal
- integer
- boolean
- date
- time

Simple elements can have a default value or a fixed value set. In the following example the fixed value is "circle":

< element name="shape" type="string" fixed=" circle "/>

All attributes are declared as simple types that can have a default or fixed value.

Restrictions (facets) are used to control acceptable values for XML elements or attributes. The constraints include:

- enumeration: Defines a set of acceptable values.
- fractionDigits: Specifies the maximum number of decimal places allowed must be equal to or greater than zero.
- length: Specifies the exact number of characters or list items allowed but must be equal to or greater than zero.
- maxExclusive: Specifies the upper bounds for numeric values (the value must be less than this value).
- maxInclusive: Specifies the upper bounds for numeric values (the value must be less than or equal to this value).
- maxLength: Specifies the maximum number of characters or list items allowed, which must be equal to or greater than zero.

- minExclusive: Specifies the lower bounds for numeric values (the value must be greater than this value).
- minInclusive: Specifies the lower bounds for numeric values (the value must be greater than or equal to this value).
- pattern: Defines the exact sequence of characters that are acceptable.
- totalDigits: Specifies the exact number of digits allowed and must be greater than zero.
- whiteSpace: Specifies how white space (line feeds, tabs, spaces, and carriage returns) are handled.

2.1.5.3 Complex elements

Complex types permit element definitions with attributes or subelements in XML Schema. The four kinds of complex elements are:

a) empty elements
b) elements that contain only other elements
c) elements that contain only text
d) elements that contain both other elements and text

There are different ways to define complex elements in an XML Schema. A complex element can be declared directly by naming the element as follows:

```
<element name="employee">
    <complexType>
       <sequence>
          <element name="firstname" type="string"/>
          <element name="lastname" type="string"/>
       </sequence>
    </complexType>
</element>
```

This example defines a complex element ***employee*** with two subelements "firstname" and "lastname", which must appear in the order declared.

Another option is an element can have a type attribute that references the name of the complex type to use:

```
<element name="employee" type="person"/>
    <complexType name="person">
        < sequence>
            < element name="firstname" type="string"/>
            <element name="lastname" type="string"/>
        </sequence>
    </complexType>
```

We can also base a complex type element on an existing complex type and add additional elements.

2.2 Distributed Database Systems

Distributed database systems have been intensively studied. Since we adapt algorithms for fragmenting relations in a distributed database system to a distributed XML database system, this section briefly introduces distributed database systems and relevant fragmentation algorithms.

A distributed database (DDB) is defined as a collection of multiple, *logically interrelated* databases distributed over a *computer network* [ÖV99]. Özsu and Valduriez define a distributed database management system (DDBMS) as a software system that manages distributed databases and makes the distribution transparent to the users.

In a distributed database system, data is physically stored across several sites and each site is typically managed by DDBMS that is capable of running independent of the other sites. The computer network connecting these sites in a DDB is not a multiprocessor system having either the shared memory (tightly coupled) architecture or the shared disk (loosely coupled) architecture [EN]. A distributed database is a database rather than a file system managing files stored at each node of the network. In a DDB, distributed data should be logically related in terms of the relationship defined according to some structural formalism and the data access should be via a common interface. A distributed DBMS provides other functions including integration of heterogeneous data, query optimization and processing, concurrency control, and recovery [ÖV99]. Figure 2.1 depicts a distributed database environment.

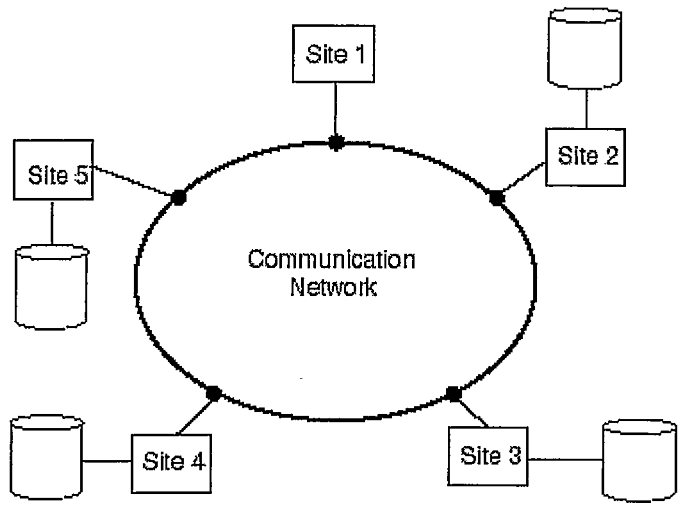


Figure 2.1 Distributed Database Environment [ÖV99]

2.2.1 Implementation alternatives

There are three typical architectural mechanisms for distributed DBMSs: client/server systems, peer-to-peer systems, and multi-database systems.

In the client/server systems [OH+96], the query sites correspond to clients while the data sites correspond to servers. For the client/server DBMSs, the server focuses on the data management. This means that all of query processing and optimization, transaction management and storage management are done at the server. At the client side, besides the application and the user interface, there is a *DBMS client* module responsible for managing the data that is cached to the client and (sometimes) managing the transaction locks that may have been cached as well [ÖV99]. A typical client/server functional distribution is given in Figure 2.2.

In contrast to the client/server system, a peer-to-peer system makes no distinction among the client machines and the server machines. Each site in the system performs the same functionality. In executing queries (transactions), the global query optimizer (global execution monitor) communicates directly with the local query processors (local recovery managers) where parts of the query (transaction) are executed [ÖV99]. Thus, the communication mechanism is more involved, leading to more complicated software structures.

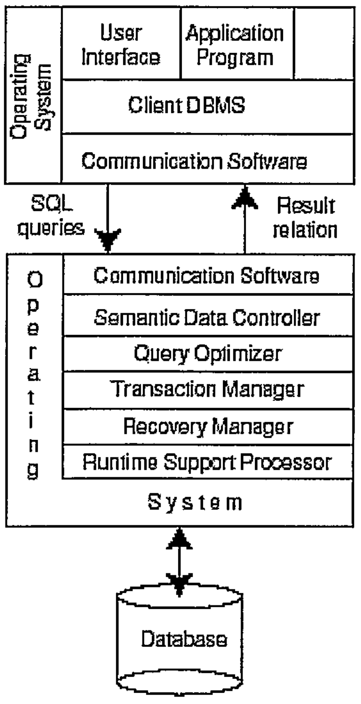


Figure 2.2 Client/Server Architecture [ÖV99]

In this thesis, among different distributed mechanisms, only peer-to-peer systems are considered.

2.2.2 Horizontal fragmentation in distributed database design

Many factors contribute to the optimum design of distributed systems such as logical data organization, the application location, characteristics with which applications access data, the network properties, and the computation systems available in each site of the network. These factors make the distribution design formulation complex. The information needed to carry out this distribution design can be split into four categories: data, application, network communication and the computing system information. The last two categories are entirely quantitative in nature and they are used in the allocation problem.

In many distributed systems there is no need for data fragmentation, since the minimal distribution unit is a file. However in distributed databases defining a suitable design granularity is necessary. The locality of accesses of applications drives re-defining the distribution unit to portions, instead of the relation as a whole. This process is known as fragmentation.

In distributed database design, the fragmentation of a relation consists of decomposing the relation into logical portions called fragments or partitions. Typically, fragmentation increases the concurrence level of the queries and enables the placement of data in close proximity to its place of use. There are two fundamental fragmentation strategies: horizontal and vertical [ÖV99]:

- Vertical Fragmentation divides a relation in a set of fragments by projecting it over its attributes.
- Horizontal Fragmentation partitions a relation along its tuples. It places each tuple of the relation in a different partition by using the predicates defined on the relation. There are two stages to horizontal partitioning: primary and derived. Primary horizontal fragmentation of a relation is performed by using predicates that are defined on that relation. Derived horizontal fragmentation partitions the relation that results from the predicates being defined on another relation.

In most cases it is not sufficient for only a horizontal or vertical fragmentation of a database schema to meet all of the requirements of user applications. At this point a horizontal fragmentation may be followed by a vertical one, or vice versa, producing a hybrid fragmentation. Although we do not consider hybrid fragmentation as a primitive type of fragmentation strategy, it is quite obvious that many real-life partitions may be hybrid.

For the primary and derived horizontal fragmentation, the objective is to determine the set of minterm predicates to be applied on the relation to generate a set of fragments. Each one of these fragments will contain a subset of tuples that satisfies a given minterm. The primary horizontal fragmentation of R (a relation) on the set of minterm predicates M is defined as:

$$F(R,M) = \{F_i \mid F_i = \sigma_{m_i}(R) \forall m_i \in M\} \qquad 1 \leq i \leq |M| \qquad (1)$$

Given a set $P = \{P_1, P_2, \ldots, P_m\}$ of simple predicates for relation R, the set of minterm predicate $M = \{m_1, m_2, \ldots, m_z\}$ is defined as:

$$M_i = \{m_j \mid m_j = \bigwedge_{p_k \in P} p^*_k\},\ 1 \leq k \leq m,\ 1 \leq j \leq z \qquad (2)$$

where $p^*_k = p_k$ or $p^*_k = \neg p_k$. Each simple predicate can occur in minterm predicate either in its natural or its negated form.

Given a relation $R(A_1, \ldots, A_n) \subseteq D_1 \times \ldots \times D_n$. A simple predicate is defined as

$$p = A_i\ \theta\ Value \qquad 1 \leq i \leq n \qquad (3)$$

where A_i is an attribute of R defined on D_i, $Value \in D_i$, $\theta \in \{=, \neq, <, \leq, >, \geq\}$ and p is a simple predicate.

The horizontal fragmentation problem of R can be stated as finding a set of fragments for all the applications defined on R. Additionally, any possible horizontal fragmentation of R must guarantee that it does not modify its original semantics. For this reason, the following conditions should be satisfied:

- Completeness: Each tuple t of the original relation is found in some of the generated fragments
  Completeness (R, M): $\forall t \exists F_i (t \in R \wedge F_i \in F(R,M) \wedge t \in F_i)$

- Disjointness: Each tuple t of the original relation is found at most in one of the generated fragments
  Disjoinness (R, M):
  $\forall t \exists F_i \neg \exists F_j (t \in R \wedge F_i \in F(R,M) \wedge t \in F_i \wedge F_j \in F(R,M) \wedge F_i \neq F_j \wedge t \in F_i \wedge t \notin F_j)$

- Reconstruction: an operator must be defined in such a way that it obtains the original relation when it is applied over the generated fragments.

$$R = \bigcup_{i=1}^{n} F_i$$

Once we have defined the horizontal fragmentation problem, it is necessary to specify the information required for its solution:

- Database information. The database information concerns the global conceptual schema. It is important to note how the database relations are connected to one another, especially with joins. In the relational model, these relationships are also depicted as relations.
- Application information. The fundamental information consists of the predicates used in user queries and their access frequencies in a specific period of time.

In addition to this information, it is also necessary to define a set of implications according to the semantics of the database. This is because the construction of predicates according to (2) can generate minterms contradictory to a set of implications. A full treatment of the design problem in distributed relational databases is provided by Özsu and Valduriez [ÖV99].

2.3 XML databases

Both industry and academic researchers have enthusiastically promoted XML as a data storage and retrieval medium. However, XML databases may not replace traditional databases, but they do present some very different possibilities. In any business application one has to exchange data between a database and other systems (other applications, another database, *etc.*). As the same data model is not used by everyone, it is necessary to find one that allows wider audience usage and yet manages to capture all the semantics entailed.

2.3.1 Types of XML databases

Database systems are built on the data model, DDL (Data Definition Language), and DML (Data Manipulation Language). However, XML databases were not initially built from these bases.

First efforts were aimed at exposing data in a relational database as XML. These systems are known as the *XML Enabled Databases*. Such databases should have a middleware that would enable publishing of relational data as XML documents, storing XML within relational databases, and querying stored data.

Alternatively, *Native XML Databases* are built on XML databases from scratch so they outperform XML Enabled Databases by fine-tuning the storage and indexing mechanisms to suite XML needs. Identifying the best approach depends on the types of applications (i.e. data-centric or document-centric applications).

2.3.2 XML enabled databases

Some mechanisms have been proposed for XML enabled databases. To handle XML data, these systems must address the following problems:

- Storing XML data in a relational database.
- Generating XML documents from data stored in a relational database.
- Executing XML queries over relational data.

The SilkRoute [Fern00, Fern01] system is a typical representative of XML Enabled Databases. SilkRoute uses its own *Relational to XML transformation language* (RXL) to define how the data in relational tables will be published as XML according to the XML Schema or DTD. The core components of SilkRoute are composed of *Query Composer*, *Translator*, and *XML Generator*.

The Translator uses RXL expression to:

- Generate a set of SQL queries that will extract the necessary data from the relational database. The task of RDBMS is to execute those queries and produce the answer in the form of the set of tuples, and to
- Extract the XML template that will be used to structure the relational data in the final XML output.

XML Generator fills the XML template with the received data to produce the final XML answer. Data is exported to XML in two steps. First, an XML view of the relational database is defined using RXL, a declarative query language. The resulting XML view is virtual. Secondly, some other applications formulate a query (e.g., an XML-QL query) over the virtual view. The RXL view query and the XML-QL query are then passed to the query composer. The composer computes the composition generates a new executable RXL query. The translator translates the executable query into one or more SQL queries, which is executed on the database server. The result is returned to the XML generator to produces XML documents, which are then returned to the application. The benefit is that only the result of that XML-QL query is materialized. Unfortunately, translating RXL queries into efficient SQL queries is still an open problem. Fernandez [Fren00] focuses on how to translate a user's XML-QL to RXL query and translating RXL query to SQL. The details of the XML generator are not specified.

Shanmugasundaram focuses on problems related to XML Enabled databases. In the middleware system XPERANTO [Shan+00, Shan01, Shan01', Shan99], the project generates an XML query interface for different object-relational database structure. This tool uses XML-QL. The user is only aware of the XML schema so only needs to write efficient XML queries. XPERANTO contains 4 major components. The XML Schema Generator is responsible for the automatic transformation process of the source schema into an XML schema based on the database catalog. The user still has to pre–define the XML views. Query Translation is the main component of the system and is in charge of translating the XML queries into queries according to the structure of the source. The XML Query Graph Model is used to create efficient queries. The XML Tagger's main goal is to convert the results of the SQL queries into something readable by the XML

Parser. The last component is the XML View Services, which is responsible for storing the XML-QL view definitions.

The advantages of XPERANTO are the automatic mapping and construction of the XML schema based on the object-relational source structure while preserving the names of the correspondent objects. The disadvantages of XPERANTO are that it defines more XML types than needed and it does not preserve relationships between tables/objects.

Shanmugasundaram [Shan01'] thoroughly analyzes the techniques for the XML publishing of relational data. Special attention is paid to ways to do a *computational pushdown*, which makes existing relational engines do as much work as possible to attain better performance. Relational engine extensions are proposed to support XML publishing.

The XML publishing task is separated into the three subtasks:

- Data extraction
- Data structuring
- Data tagging

These tasks can be performed in different ways. In transforming Relational to XML data, both hierarchical structure and tags have to be added to relational data. If done early in the process they are called *early structuring/tagging* and if done later in the process they are called *late structuring/tagging*. Structuring/tagging can be done *inside* or *outside* the relational engine.

Techniques for publishing and querying XML views over relational data are general enough to deal with any given relational schema. Nevertheless, the way data is stored in relational database greatly influences the performance of XML publishing. Therefore, when it comes to storing XML documents in relational database, offered solutions primarily differ in the performance of XML publishing of that data.

A method for storing XML documents in a set of binary tables is available [Schm00]. A binary table is created for every possible parent-child relation existing in the XML document.

Deutch [Deut99] presents the STORED that uses data mining techniques to discover common structure in semistructured XML documents. The discovered structure is used to develop the relational schema for storing structured parts of XML document.

2.3.3 Problems with XML enabled databases

There are many chanllenges facing XML enabled databases. First, if a relational database is used to store the XML data, transforming XML queries to efficient SQL queries is an open issue. Second, since the hierarchical XML data is distributed to different tables in the relational back end, complex joins are needed for some XML queries, which result in poor system performance. Third, updating may be very costly. Furthermore, there is the bi-directional format and structure conversion overhead.

Theoretically, XML enabled databases are supposed to handle entity usage, CDATA sections, comments, and Processing Instructions, *etc*. However, this is generally not done in practice.

When XML documents are stored in a database and then recreated through XML publishing is called *XML round tripping*. The ideal round trip is the case when the original and recreated XML documents are identical. This is not the case in many XML enabled systems due to the loss of some information such as order or white spaces. Even the basic representations of the original and recreated XML document often differ [Bour02]. Generally, storing XML data in XML enabled relational databases is appropriate for data-centric applications that do not care about exact round tripping.

2.3.4 Native XML Databases

Native XML databases are often considered a database being built "from scratch". As defined by the members of the XML:DB initiative [XML:DB], a native XML database is one that:

- Defines a (logical) model for an XML document -- as opposed to the data in that document -- and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order. Examples of such models are the XPath data model, the XML Infoset, and the models implied by the DOM and events in SAX 1.0.
- Must have XML documents as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.
- No required specific underlying physical storage model is assumed. For example, it can be built on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files.

Native XML databases differ from XML-enabled databases in three main ways:

- Native XML databases can preserve physical structure (entity usage, CDATA sections, *etc.*) as well as comments, PIs, DTDs, *etc.* while XML-enabled databases do not generally do this in practice.
- Native XML databases can store XML documents without knowing their schema or DTD, assuming one even exists. Although XML-enabled databases could generate schemas on the fly, this is impractical in practice, especially when dealing with schema-less documents.
- The only interface to the data in native XML databases is XML and related technologies, such as XPath, the DOM, or an XML-specific API, such as the XML:DB API [XMLAPI]. XML-enabled databases, on the other hand, offer direct access to the data, such as through ODBC.

Native XML databases are defined through the data model they support while XML enabled databases are identified by their implementation..

Native XML Database Management System (**NXDBMS**) should have the following properties:

- Designed from the bottom up exclusively to store and manage XML documents.
- The fundamental unit of logical storage is the XML document.
- The XML documents are logically stored (at the conceptual level) as XML.
- The XML hierarchical graph structure is preserved.
- The whole document is physically stored in its entirety in a "single" location.

In summary, a Native XML database system is a software system using XML DDL and XML DML to handle XML data.

There are two storage approaches for native XML databases [Bour02]:

- Text-Based storage of XML data and
- Model-Based storage

Text-Based storage stores XML documents "as-is"; that is, in a form of single textual unit. Text can be stored as a file in a file system, as a CLOB within a relational database, or using some other mechanism to store text.

Model-Based storage does not store XML document as text. Prior to storing any data on permanent media, XML documents are modeled i.e. transformed into an internal object model representing the source XML document. This model is then saved. The selected XML model must be rich enough to model all the elements of XML documents. DOM [W3CDOM] is one of the possible choices for modeling XML.

Tamino Version 3.1 [Tamino] adds new, more usable administration and schema-editing tools, a WebDAV server for easier XML file import and export, and a new Java API. It still supports its earlier Java API but now you can use JavaScript, Java Server Pages, and COM APIs. Tamino has both normal and full-text indexing features. It can be configured to either enforce schema structures or to permit the database to accept extra elements without complaint.

There is a built-in relational database (in addition to its native XML database engine), and it can create live mappings from XML databases to third-party relational databases. This means that relational data can be left where it is but be accessed through Tamino's XML APIs. This flexibility is key when integrating XML into an existing IT infrastructure and makes ODBC-type drivers unnecessary.

Tamino has some shortcomings. Its query language supports XPath with a few additions, so it suffers from XPath's limited syntax. To overcome this, an extension is added to XPath that allows for sorted results and an extension to Tamino's schema language to allow joins. These joins are static (you need to define them at database design time) and cannot be created on the fly in query code as with XQuery.

Tamino's storage engine is document-oriented so concurrency locking is done at the document level instead of the node level. In update-intensive situations, this design requires data be placed in many small XML documents to avoid a huge performance slowdown. Further, Tamino uses a non-XML–based update API that has too many separate but overlapping tools and lacks query-building helpers.

2.3.5 Problems with Native XML databases (NXDBs)

Chaudhri *et al.* [Chau+03] compare the performance of a XML-enabled database with the that of a native XML database. The result shows that the native XML database performs better than the XML-enabled database when handling XML documents with larger data sizes. This is because the native XML database engine uses the index key to access the XML data directly. However, the native XML database has some disadvantages. Both data and index size consumed by the native XML database is much larger than in the XML-enabled database.

The larger index in the native XML database is because more comprehensive indexing support is provided, such as full-text searching. Update is another weakness of the native XML database because that index files have to be updated as well. Further most products require a document be retrieved and changed using your favorite XML API, and then returned to the database. A few products have proprietary update languages that allow updates within the server, and a couple of open source NXDBs support XML: DB Xupdate for the same purpose. This is likely to be a problem until XQuery adds an update language. In the meantime DOM manipulation will remain the most common update method used with NXDB products. Furthermore, tag names have to be stored in the native XML database because as the XML document becomes larger, more disk space is required to store tag names.

In addition, indexing a XML document efficiently is an open problem. This is because of the complexity of XML documents' hierarchical structure

2.3.6 Distributed XML Databases

So far there is no formal definition for the distributed XML database (DXDB). However, based on the definition of distributed database system, A DXDB can be defined as a collection of multiple, *logically interrelated* XML databases distributed over a *computer network*. Although both XML-enabled databases and native XML databases are still not mature enough compared with relational databases, it is just a matter of time.

Much work has been done in the distribution and integration of relational database systems. Some concepts and mechanisms can also be introduced into DXDBs. The challenge is selecting that which should be chosen.

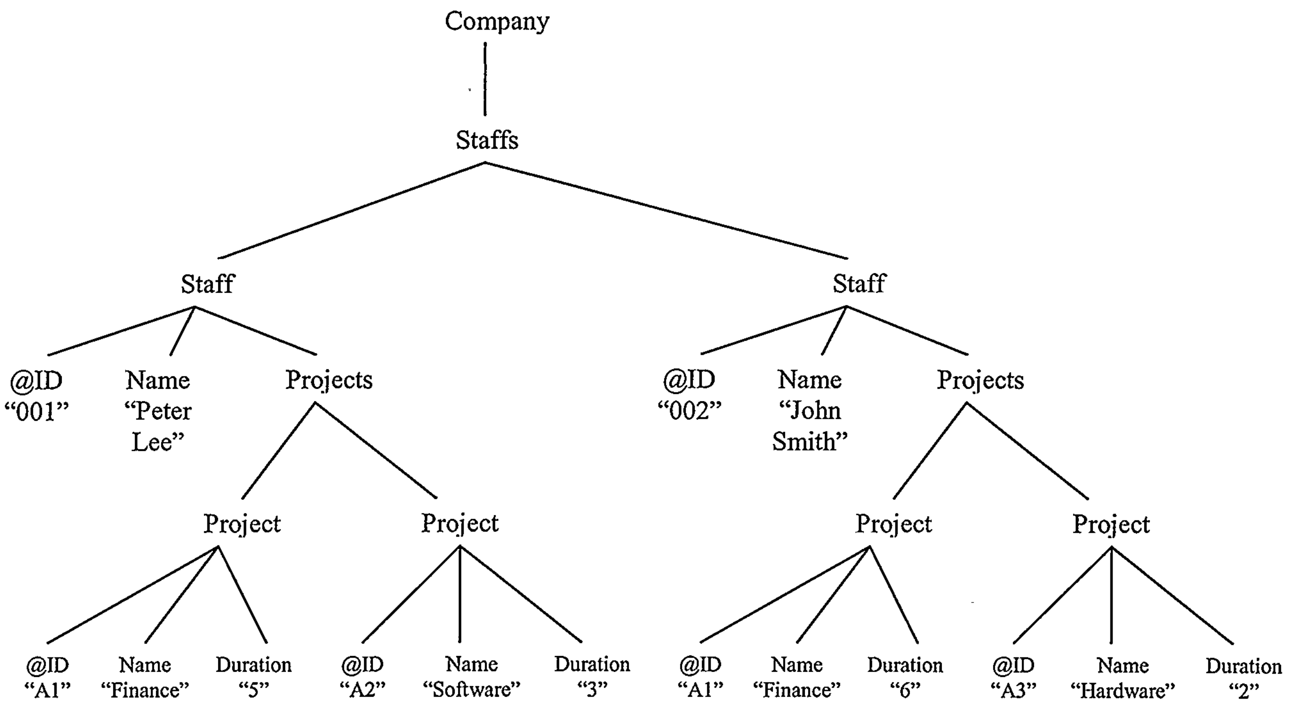
The semi-structured XML data is also too flexible, which increases the complexity of the XML database systems. To reduce the complexity, some constraints must be added to the XML documents.

2.4 Related Work

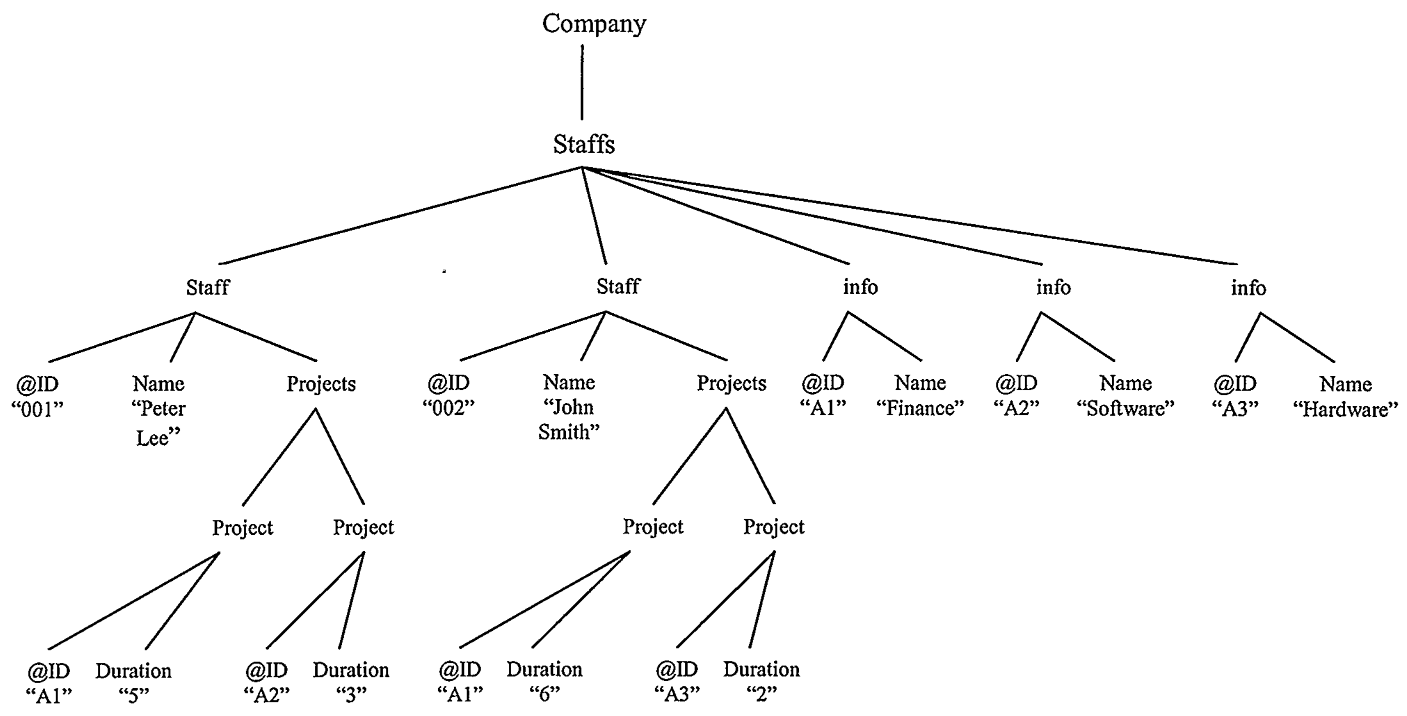
Some efforts have attempted to find criteria for good XML document design [Aren+03]. There are some recent projects related to normalizing XML design, querying fragmented XML documents, and distributing XML repositories [Aren+02] [DeHa+03] [Bose+03] [Brem+03] [Abit+03].

2.4.1 Normal Form for XML Documents

Arenas [Aren+02] uses tree tuples (paths) to determine functional dependencies (*FD*). Based on *FD*, *XNF* (XML Normal Form) is defined. An algorithm is given to decompose DTD. The goal of the algorithm is to find redundant elements first. It then decomposes those redundant elements and puts them under new created elements. Given the following example:



by running the decomposing algorithm, the result should be:



New elements - **info** are generated to hold **Name** of projects to eliminate redundancy. However, there are some restrictions of their approach. First, the algorithm only deals with DTDs. It is well know that there are many limitations of DTDs, such as very limited capability for specifying data types, no support for namespaces, and no explicit relationship. XML Schema is invented to overcome these limitations and believed to be the replacement of DTDs. Second, only the intrinsic properties of the data model are considered where query and update language are not taken into account. Finally, many-to-many relationship is not treated in the algorithm. In the above example, assuming that there is another element **Projects** under **Company,** which includes **Project** element where many staffs work on, the algorithm will not work well.

2.4.2 Distribution, Fragmentation, and Query of XML Documents

Issues raised by the distribution and replication of dynamic XML data are studied by Abiteboul *et al.* [Abit+03]. Dynamic XML documents consist of parts with explicit data and other parts that are given by embedded calls to web service. However, no horizontal and vertical fragmentation for XML documents are considered in their work.

Bremer and Gertz [Brem+03] use a rooted, node-labeled tree to model XML documents. They generally specify that what fragments of a XML document should be like. However, the fragmentation of XML documents is not presented.

Among many query languages for XML, XQuery [XQ01] has been widely recognized and adopted as the standard. Based on this standard, DeHaan *et al.* [DeHa+03] propose an approach to map a XQuery expression to a an equivalent SQL query using dynamic interval encoding of a collection of XML documents as relations. The approach tries to evaluate and generate XML queries using relational technology.

2.5 Summary

This chapter began with the discussion on the definition and syntax of XML. Section 2.2 discussed the various mechanisms of the distributed database systems. Further the discussion covered the data fragmentation in the DDB. Section 2.3 presented the different approaches to build the XML database. In addition, the problems of the existed XML databases are analyzed. Projects and issues related to XML document design, fragmentation and distribution are discussed in Section 2.4.

We observe that less research efforts and accomplishment have been made for the design and fragmentation problems of XML documents. This is the major motivation of this thesis. Chapter 3 presents a relational framework of XML as the platform for our work.

Chapter 3
A Relational Framework for XML

To date, there is very limited work on how to design a good XML document. In this chapter, we adapt relational concepts to describe a XML document and apply this framework to design a XML document in subsequent chapters. Since the use of attributes in XML is ambiguous and an attribute can always be replaced by an element, we will not consider using attributes in our XML document design. Further we assume that all XML documents come with a schema document where the XML schema is specified.

The relational framework of XML is a platform for designing XML documents rather than another specification of XML. The goal of this research is to explore the design rules of XML documents.

Section 3.1 defines the basic components of a XML document in the relational framework of XML. Keys and foreign keys are discussed in Section 3.2. Normal forms in XML are introduced in Section 3.3. In Section 3.4, design mechanisms for placing a XML document in $3NF_{xml}$ is discussed. This chapter concludes with a summary in Section 3.5

3.1 Table elements, Tuple elements, Field elements and Domain

The relational framework of XML represents the XML document as a collection of *table elements*. Informally, each table element resembles a table in the relational model. For example, the XML document in Figure 3.1 is considered to be in the relational framework of XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<company xmlns="http://www.cpsc.ucalgary.ca/~yingqi/xml"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceschemaLocation="Sample3-1.xsd">
        </company>
          <SUBSIDY>
             <role>
                <DEGREE>Ph.D</DEGREE>
                 <SUB>4000</SUB>
              </role>
             <role>
                 <DEGREE >MBA</DEGREE >
                 <SUB>3500</SUB>
              </role>
             <role>
                < DEGREE >MSc</DEGREE >
                 <SUB>2900</SUB>
              </role>
             <role>
                <DEGREE>BE</DEGREE>
                 <SUB>2300</SUB>
              </role>
           </SUBSIDY>
          <STAFF>
             <employee>
                 <SID>ID1</SID>
                  <SNAME>K. Barker </SNAME>
                 <DEGREE>Ph.D</DEGREE>
             </employee>

             <employee>
                 <SID>ID2</SID>
                  <SNAME>J. Wong</SNAME>
                 <DEGREE>MBA</DEGREE>
             </employee>
             <employee>
                 <SID>ID3</SID>
```

```
            <SNAME>D. Parker</SNAME>
            <DEGREE>MSc</DEGREE>
        </employee>
        <employee>
            <SID>ID4</SID>
             <SNAME>B. Unger</SNAME>
            <DEGREE>BE</DEGREE>
        </employee>
        <employee>
            <SID>ID5</SID>
             <SNAME>M. Shaw</SNAME>
            <DEGREE>MBA</DEGREE>
        </employee>
        <employee>
            <SID>ID6</SID>
             <SNAME>A. Sands</SNAME>
            <DEGREE>Ph.D</DEGREE>
        </employee>
        <employee>
            <SID>ID7</SID>
             <SNAME>C. Day</SNAME>
            <DEGREE>MSc</DEGREE>
        </employee>
        <employee>
            <SID>ID8</SID>
             <SNAME>F. Ada</SNAME>
            <DEGREE>MBA</DEGREE>
        </employee>
     </STAFF>
</company>
```

Sample3-1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.cpsc.ucalgary.ca/~yingqi/xml"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:r="http://www.cpsc.ucalgary.ca/~yingqi/xml"
          elementFormDefault="qualified">
<complexType name="SUBSIDYType">
  <sequence>
    <element name="role" type="r:roleType" maxoccurs="unbounded/>
  </sequence>
</complexType>

<complexType name="roleType" >
  <sequence>
    <element name="DEGREE" type="string"/>
    <element name="SUB" type="positiveInteger"/>
```

```
   </sequence>
  </complexType>

  <complexType name=”STAFFType”>
    <sequence>
      <element name=”employee” type=”r:employeeType” axoccurs=”unbounded/>
    </sequence>
  </complexType>

  <complexType name=”employeeType” >
    <sequence>
      <element name=”SID” type=”string”/>
     <element name=”SNAME” type=”string”/>
     <element name=”DEGREE” type=”string”/>
   </sequence>
  </complexType>

  <element name=”company”>
    <complexType>
      <sequence>
        <element name=”SUBSIDY” type=”r:SUBSIDYType” minOccurs=”0”/>
        <element name=”STAFF” type=”r:STAFFType” minOccurs=”0”/>
      </sequence>
    </complexType>
    <key name=”subsidyKeyP”>
        <selector xpath=”r:SUBSIDY/r:role”>
        <field xpath=”DEGREE”>
    </key >
    <key name=”staffKeyP”>
        <selector xpath=”r:STAFF/r:employee”>
        <field xpath=”SID”>
    </key >
    <keyref refer=”subsidyKeyP” name=”staff2pay”>
        <selector xpath=”r:STAFF/r:employee”/>
        <field xpath=”DEGREE”/>
    </keyref >
   </element>
  </schema>
```

Sample3-1.xsd

Figure 3.1 Sample XML Documents in the Relational Framework of XML

In Figure 3.1, element **SUBSIDY** and element **STAFF** are table elements. A **table element** (TBE) consists of **tuple elements**. Any tuple element within a table element has the same opening and closing tag name. A tuple element is made up of **field elements**. A field element must be a simple element whose domain is integer, string *etc.* according to [XS]. Field elements must occur in any tuple element within a table element in the same order and number. This rule is analogous to the relational model where each row in a table has the same number of attributes.

A **domain** D is a set of atomic values. **Atomic** means that each value in the domain is indivisible as far as the relational framework of XML is concerned. Since a field element consists of an atomic value, there is always a domain defined for it. The domain of a field element f is denoted as $dom(f)$.

A **table element structure** S, denoted by $S : T(F_1, F_2, ..., F_n)$, is made up of a table element name S and a list of field element $F_1, F_2, ..., F_n$ in its tuple elements having the tag name T. A table element structure is used to describe a table element. The **degree** of a table element is the number of field elements occurring in its table element structure.

An example of a table element structure for a table element of degree 2 (see Figure 3.1), which describes the salary of an employee, is as follows:

SUBSIDY: role(DEGREE, SUB)

For this table element structure, SUBSIDY is the name of the table element, which has tuple elements with tag *role* and field elements: DEGREE and SUB.

A **table element** e having the table element structure $S : T(F_1, F_2, ..., F_n)$, also denoted by $e(S)$, is a set of m tuple elements $e = \{ t_1, t_2, ..., t_m \}$, where T is the tag name of tuple elements and F_n is the tag name of field elements. Each tuple element t is an ordered list of n field elements $t = < f_1, f_2, ..., f_n >$, where each f_i, $1 \leq i \leq n$, is a field element with an atomic value or is a special **null** value (empty value between its opening and closing tags).

In the schema document, table element e with the table element structure $S : T(F_1, F_2, ..., F_n)$ is defined with the following template:

```
1    <complexType name="S">
2      <sequence>
3        <element name="T" type="r: T" maxoccurs="unbounded/>
4      </sequence>
5.   </complexType>
6    <complexType name="T" >
7      <sequence>
8        <element name="F1" type="DataType1"/>
9        <element name="F2" type="DataType2"/>
10            ...
11       <element name="Fn" type="DataTypen"/>
12    </sequence>
13   </complexType>
```

Figure 3.2 Template 1

DataType$_i$, $1 \leq i \leq n$, is the corresponding data type for F_i, which is either of a built-in simple type [XS] in XML Schema or of a extended built-in simple type on facets. In general, we append "Type" to "S" and "T". Hence, in the above template line 1 and line 3, "S" and "T" become "S Type" and "T Type" respectively. This naming convention is used through the balance of this thesis.

We also ensure that in one schema document, a complexType's name is unique. The "r:" is a prefix that specifies a name space. In this paper, we define "xmlns: r="http://www.cpsc.ucalgary.ca/~yingqi/xml"" at the beginning of each XML schema document.

3.2 Keys and Foreign keys

A table element is defined as a set of tuple elements. All elements of a set are distinct; hence, all tuple elements in a table element must be distinct. This means that no two tuple elements can have the same combination of values for all their field elements. The combination of field elements that makes a tuple element distinct from other tuple elements within a table element is called the *superkey* SK of that table element. A superkey can have redundant field elements. Therefore a more useful concept is that of a *key,* which has no redundancy. A **Primary key** K of a table element structure S is a superkey of S with the additional property that removing any field element F from K leaves a set of field elements K' that is not a superkey of S. Hence, a key is a minimal superkey from which we cannot remove any field elements and still have the uniqueness constraint hold. In Figure 3.1, field element *DEGREE* is defined as the key of table element **SUBSIDY** in Sample3-1.xsd at line 36. We note that once an element is defined as a key or part of the key in the schema document, then that element cannot be empty; otherwise the XML schema validator gives an error message.

Generally, a table element structure may have more than one key. In this case, each of the keys is called a **candidate key**. If a field element is a member of some candidate key in a table element structure, it is called **prime field element**; otherwise it is called **nonprime field element**. Typically we assign one of candidate keys to be the **primary key** of the table element. We use the convention of underlining the field elements that form the primary key of a table element structure as follows:

SUBSIDY: role (<u>DEGREE</u>, SUB)

STAFF: employee (<u>SID</u>, SNAME, DEGREE)

Note that when a table element structure has several candidate keys, choosing one of them to be the primary key is arbitrary. However, it is better to choose a primary key with a single field element or a group of field elements with small storage size.

Key constraints are specified on individual table elements. The foreign key constraint is specified between two table elements. Informally, the foreign key constraint states that a tuple element in one table element that refers to another table element must refer to an existing tuple element in the referred table element. For example, in Figure 3.1 line 44 of Sample3-1.xsd, field element *DEGREE* of tuple element *employee* in the table element **STAFF** is defined to refer to *DEGREE*, which is the key of tuple element *role* in table element **SUBSIDY**; hence, its value in every tuple element *employee* must match the *DEGREE* value of some tuple elements in the table element **SUBSIDY**. Table element **STAFF** is called **referencing table element** and table element **SUBSIDY** is the **referenced table element.** The primary key and foreign keys are defined and implemented in the XML schema using *key* and *keyref.*

Now we can present a template to describe two related table element structures in the schema document. The two given table element structures are:

$$\begin{tikzcd} S: T\,(\underline{F_1}, & F_2, \ldots, F_n) & & \text{Key: } F_1 \\ S`: T'\,(\underline{F_1`}, & F_2`, \ldots, F_n`) \arrow[ul] & & \text{Key: } F_1` \end{tikzcd}$$

There is a relationship between S and S`, i.e. $F_2$` in S` refer to F_1 in S. The template is shown in Figure 3.3.

```
<complexType name="SType">
  <sequence>
    <element name="T" type="r: TType" maxoccurs="unbounded/>
  </sequence>
</complexType>
<complexType name="TType" >
  <sequence>
    <element name="F1" type="DataType1"/>
    <element name="F2" type="DataType2"/>
       ...
    <element name="Fn" type="DataTypen"/>
 </sequence>
</complexType>
<complexType name="S'Type">
  <sequence>
    <element name="T" type="r: T'Type" maxoccurs="unbounded/>
```

```
17        </sequence>
18      </complexType>
19      <complexType name="T'Type" >
20        <sequence>
21          <element name="F1" type="DataType1"/>
22          <element name="F2" type="DataType2"/>
23               ...
24          <element name="Fn" type="DataTypen"/>
25       </sequence>
26      </complexType>
27      <element name="root">
28        <complexType>
29          <sequence>
30             <element name="S" type="r:SType" minOccurs="0"/>
31             <element name="S'" type="r:S'Type" minOccurs="0"/>
32           </sequence>
33        </complexType>
34        <key name="SKeyP">
35            <selector xpath="r:S/r:T">
36            <field xpath="F1">
37        </key >
38        <key name="S'KeyP">
39            <selector xpath="r:S'/r:T'">
40            <field xpath="F1'">
41        </key >
42        <keyref refer="SKeyP" name="S'2S">
43            <selector xpath="r:S'/r:T'"/>
44            <field xpath="F2'"/>
45        </keyref >
46       </element>
```

Figure 3.3 Template 2

On line 27 in Figure 3.3, "root" should be replaced with the XML document's name. For *key* and *keyref*, we adopt the naming convention as well, which makes name more meaningful. We append "KeyP" to the table element structure's name S (Line 34), which means "SkeyP" is the key's name for the table element S. Once we know the table element structure and foreign key relationship between table element structures, to define the corresponding XML schema document, we just use the template shown in Figure 3.3. Templates shown in Figure 3.2 and Figure 3.3 can be applied generally. We can apply *Template 1* to describe any individual table element structures and *Template 2* to describe any two related table element structures. Hence, by using *Template 1* and *Template 2*, we can describe table element structures within the same organization in XML Schema.

3.3 Normal Forms in XML

Compared with relational database technology, the main drawback of XML documents is redundancy. In the relational database design, redundancy is eliminated by normalizing tables. However, there is no normalization in producing an XML document, which considers all elements. If an XML document is very large and many elements contain one or many identical elements, the redundancy problem becomes obvious and affects the efficiency of queries as a result.

In the relational model, data redundancy is reduced or eliminated by designing the relational schema so it conforms to a normal form. However, data redundancy is a challenge problem for XML documents. There are few approaches to address how to design a XML document with minimal or no redundancy. In this section, we introduce two approaches to eliminate redundancy in XML documents by adapting the concept of normal forms from the relational model.

3.3.1 Definition of functional dependency in XML

A functional dependency is a constraint between two sets of field elements within a XML document in the relational framework. Suppose that table element structure has n elements $F_1, F_2, \ldots, F_n$; let us think of the whole XML document as being described by a single **universal** table element structure $S : T(F_1, F_2, \ldots, F_n)$ and $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$. We do not indicate that we actually store the XML document as a single universal table element; we use this concept only in developing the formal theory of data dependencies.

A **functional dependency** in XML, denoted by $A \rightarrow B$, between two sets of field element A and B that are subsets of $\mathcal{F}$ specifies a *constraint* on the possible tuple elements that can form a table element state e of S. The constraint is that, for any two tuple elements $t_1$and t_2 in e that have $t_1[A] = t_2[A]$, we must also have $t_1[B] = t_2[B]$. This means that the values of the B component of a tuple element in e depend on, or are *determined by,* the values of the A component; or alternatively, the values of the A component of a tuple element uniquely (or **functionally**) **determine** the values of the B component. We also say that there is a functional dependency from A to B or that B is **functionally dependent** on A. The abbreviation for functional dependency in XML is **FD_{xml}** or **$f.d_{xml}$.** The set of field element A is called the **left-hand side** of the FD_{xml}, and B is called the **right-hand side.**

Thus A functionally determines B in a table element structure S if and only if, whenever two tuple elements of $e(S)$ agree on their A-value, they must necessarily agree on their B-value. Notice the following:

- If A is a **candidate key** of S—this implies that $A \rightarrow B$ for any subset of field element B of S (because the key constraint implies that no two tuple element in any legal state $e(S)$ will have the same value of A).
- If $A \rightarrow B$ in S, this does not necessarily mean that $B \rightarrow A$ in S.

A functional dependency is a property of the **semantics** or **meaning** of the field element. The XML document designers will use their understanding of the semantics of the field elements of S—that is, how they relate to one another—to specify the functional dependencies that should hold on *all* table element states e of S. Whenever the semantics of two sets of field elements in S indicate that a functional dependency should hold, we specify the dependency as a constraint. The table element states $e(S)$ that satisfy the functional dependency constraints are called **legal extensions** (or **legal table element states**) of S, because they obey the functional dependency constraints. Hence, the main use of functional dependencies is to describe further a table element structure S by specifying constraints on its field elements that must hold *at all times.* Certain FD_{xml}s can be specified without referring to a specific table element, but as a property of those field elements. For example, {operation_licence_number} $\rightarrow$ SIN should hold for any adult in Canada. It is also possible that certain functional dependencies may cease to exist in the real world if the relationship changes.

A functional dependency is a *property of the table element structure S,* not of a particular legal table element state (extension) *e* of *S*. Hence, an FD_{xml} *cannot* be inferred automatically from a given table element extension *e* but must be defined explicitly by someone who knows the semantics of the field elements of *S*.

The inference rules for functional dependencies in XML are identical to those in the relational model (Chapter 14, [EN]). So these are not reiterated here.

3.3.2 Definition of First and Second Normal Form (1NF and 2NF) in XML

According to the definition, the tuple element can only have field elements with atomic or null values. Thus a XML document in the relational framework for XML is in **First normal form (1NF)** automatically, denoted as $1NF_{xml}$.

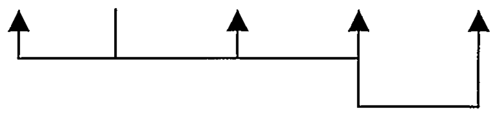
Second normal form (2NF) in XML, denoted by $2NF_{xml}$, is based on the concept of *full functional dependency.* A functional dependency $A \rightarrow B$ is a **full functional dependency** if removing any field element F from A indicates that the dependency does not hold any longer. In other words, for any field elements $F \in A$, $(A - \{F\})$ does not functionally determine B.

A TBE structure *TS* is in $2NF_{xml}$ if each nonprime field element in *TS* is *full functional dependent* on the primary key of *TS*. If the primary key of a table element structure contains only one field element, this table element is in $2NF_{xml}$ undoubtly. The TBE **STAFF** is in $2NF_{xml}$ because only "SID" is the primary key.

3.3.3 Definition of Third Normal Form (3NF) in XML

Third normal form (3NF) in XML, denoted by $3NF_{xml}$, is based on the concept of *transitive dependency.* A functional dependency $A \rightarrow B$ in a table element structure S is a **transitive dependency** if there is a set of field element Z that is neither a candidate key nor a subset of any key of S, and both $A \rightarrow Z$ and $Z \rightarrow B$ hold. Suppose we have the following example of a table element structure:

EMP: employee(ENAME, SIN, ADDRESS, DNO, DNAME)

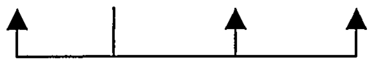


SIN → {ENAME, ADDRESS, DNO, DNAME}
DNO → DNAME

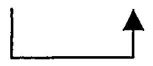
The dependency SIN → DNAME is transitive through DNO in the above example because both the dependencies SIN → DNO and DNO →DNAME hold *and* DNO is neither a key itself nor a subset of the key of EMP. Intuitively, we can see that the dependency of DNAME on DNO is undesirable in EMP since DNO is not a key of EMP.

To test if a table element is in $3NF_{xml}$, we have to find out if any nonkey field element is functionally determined by another nonkey field element (or by a set of nonkey field elements). Thus, there should be no transitive dependency of a nonkey field element on the primary key field element. We modify Example 3.1.3.2 into $3NF_{xml}$ as follows:

EMP: employee (ENAME, SIN, ADDRESS, DNO)



DEPT: department (DNO, DNAME)



SIN → {ENAME, ADDRESS, DNO}
DNO → DNAME

3.4 Design of XML Documents

3.4.1 Design a XML document in $3NF_{xml}$

To design a $3NF_{xml}$ XML document, there are two design patterns: bottom-up and top-down. For the bottom-up design, the design procedure is shown as the following steps:

1. System analysis.

2. Draw the ER diagram.
3. Map the ER diagram to the table element structures.
4. Check to guarantee that the table element structures are in $3NF_{xml}$.
5. Transform the table element structures into the XML schema using Template 1 and Template 2 (See Figure 3.3).

In contrast, a top-down design methodology would start with existing XML documents. Dealing with existing XML documents, we follow the steps shown below:

1. Treat simple elements in the XML documents as field elements in the relational framework of XML and group all these field elements into a universe table element structure.
2. Get the functional dependencies according to the schema document of the XML document or the designer of the original XML documents.
3. Break the universe table element structure into table element structures in $3NF_{xml}$.
4. Transform the table element structures into the XML schema using Template 1 and Template 2.
5. Re-generate the XML document according to the XML schema using the data in the original XML document.

Since there is almost no redundancy in 3NF, the resulting XML document has very little data repetition.

3.4.2 Transform a Relational Database Schema into a XML Schema Document

In the real world, we sometimes have to convert a database to a XML document. The challenge is how to convert the relational schemas to XML schemas. We introduce a novel and practical approach in this section.

Elmasri and Navathe [EN] describe a relational schema R, denoted by R (A_1, A_2, …, A_n), as composed of a relation name R and a list of attributes A_1, A_2, …, A_n. The only difference between the notation of a relational schema R (A_1, A_2, …, A_n), and a table element structure S: T (F_1, F_2, …, F_n) is that there is a tuple element named T in S. Naming the tuple element is arbitrary. We give an example to convert a relation schema to a table element structure in Figure 3.4.

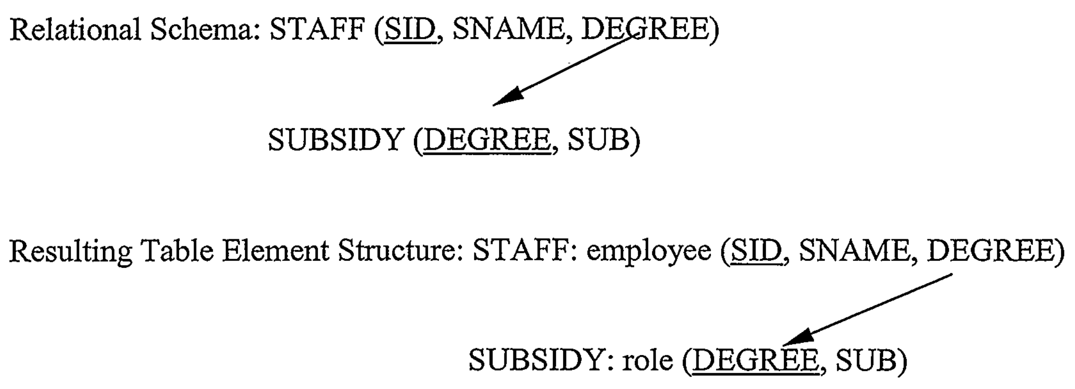

Relational Schema: STAFF (SID, SNAME, DEGREE)
SUBSIDY (DEGREE, SUB)
Resulting Table Element Structure: STAFF: employee (SID, SNAME, DEGREE)
SUBSIDY: role (DEGREE, SUB)

Figure 3.4 Conversion Between Relational Schemas and Table Element Structures

The next step is to find the referential constraints between tables. We then use Template 1 and Template 2 to convert the table structures to XML schema. The whole procedure is:

1. Convert relational schemas to table element structures.
2. Find referential constraints between the table element structures.
3. Using Template 1 and Template 2 converting the table element structure to XML schema.

Given the example in Figure 3.4, the resulting schema document is shown in Figure 3.1 Sample3-1.xsd.

3.5 Summary

This chapter introduces the relational framework of XML. The fundamental components in this framework are presented and defined in Section 3.1. It is worth noting that all documents in this framework are legitimate XML documents. The goal of this model is to constrain the XML documents into a good structure from which the applications benefit. The concept of keys and foreign keys are specifically defined in Section 3.2. Normal forms in XML are discussed in Section 3.3. Section 3.4 provides the procedures to design XML documents in the relational framework of XML.

Chapter 4
Fragmentation Algorithms for XML

4.1 Reasons for Fragmentation

In a distributed computing environment, both programs and data are placed across the sites of a computer network. Özsu and Valduriez [ÖV99] present some strategies for distributed database design. There are two fundamental fragmentation strategies: horizontal and vertical. Horizontal fragmentation partitions a relation into tuples. Each fragment possesses a subset of the whole relation. There are two ways to do horizontal partitioning: primary and derived. *Primary horizontal fragmentation* of a relation is achieved using predicates defined on that relation. *Derived horizontal fragmentation* is the fragmentation of a relation that results from the predicates being defined on another relation.

This thesis discusses the design strategies of distributed XML documents and the proposed algorithms in this design. We will adapt the algorithms proposed by Özsu and Valduriez [ÖV99] and apply them to partition XML documents.

4.2 Data Model

A XML document can be represented as a tree structure. Given a XML document Sample-1 (see Figure 4.1), its tree structure is shown in Figure 4.2.

```
<company>
  <project id = 0123>
    <title>Database Design</title>
     <employee id= 697231>
        <name> John Peterson</name>
        <jobTitle>Developer</jobTitle>
     </employee>

     <employee id= 697245>
        <name> Ricky  Bradley</name>
        <jobTitle>Programmer</jobTitle>
     </employee>
  </project>
  <project id = 0124>
    <title>Web Design</title>
     <employee id= 697231>
        <name> John Peterson</name>
        <jobTitle>Developer</jobTitle>
     </employee>
     <employee id= 697265>
        <name> Bill Smith</name>
        <jobTitle>Senior Programmer</jobTitle>
     </employee>
  </project>
</company>
```

Figure 4.1 XML Document Sample-1

Definition 1 (Notation for XML documents)

Let *xml* be an XML document. *Tree(xml)* is the tree structure of the elements defined by *xml. Subtrees(xml)* represents the set of sub-trees in *xml. Subtree(element)* denotes a sub-tree of *Tree(xml)* with the root node *element* and has at least one child node. *Node(element)* is a node of an element in *Tree(xml). Value(element)* is the value of an element. *Root(xml)* represents the root element of *xml. Parent(element)* is the parent of an element. *Children(element)* denotes all direct elements under an element. *Leaf(node)* is a node without any children nodes.

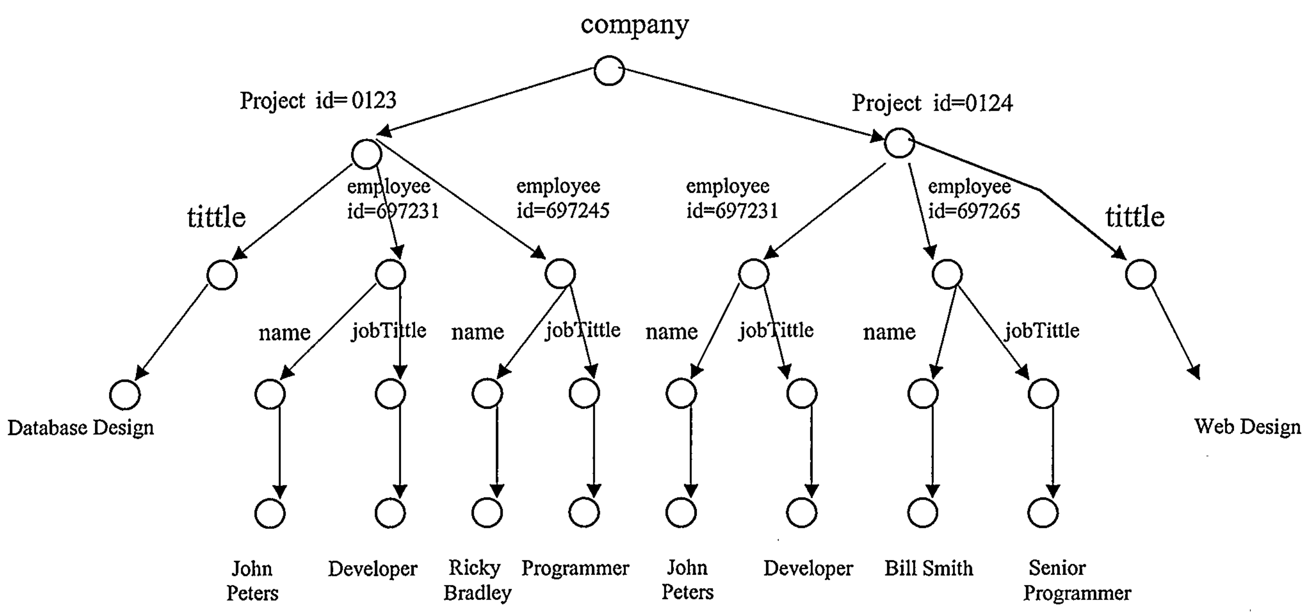


Figure 4.2 Tree Structure of Sample-1

Definition 2(Fragment of a XML document)

Let *xml* be a XML document, then $f \subseteq Subtrees(xml)$ is a fragment of *xml*. ∎

Definition 3(Fragmentation of a XML document)

Let *xml* be a XML document. A fragmentation $F = \{f_1, ..., f_n\}$ of *xml* is a partitioning of *Tree(xml)* into fragments f_1 *to* f_n, such that their union equals *Tree(xml)*. ∎

4.3 Fragmentation Strategy

4.3.1 Horizontal Fragmentation

In the partitioning of a XML document, horizontal fragmentation partitions a XML document in $3NF_{xml}$ along one or more of its table elements. Thus each fragment has a subset of the table elements of the document. The horizontal fragmentation of a XML document is performed using predicates that are defined on one field element of that XML document. There are two versions of horizontal fragmentation: primary and derived. *Primary horizontal fragmentation* of an element of a XML document is performed using predicates that are defined on that element. *Derived horizontal fragmentation,* on the other hand, is the partitioning of an element of a XML document that results from predicates being defined on another field element in the same XML document.

```
<company>
    <PROJ>
        <project>
           <PCODE>P001</PCODE>
           <PNAME>Consulting</PNAME>
           <BUDGET>160000</BUDGET>
           <CITY>Edmonton</CITY>
        </project>
        <project>
           <PCODE>P002</PCODE>
           <PNAME> Tech. Support </PNAME>
           <BUDGET>175000</BUDGET>
           <CITY>Calgary</CITY>
        </project>
        <project>
           <PCODE>P003</PCODE>
           <PNAME> Tele-Marketing</PNAME>
           <BUDGET>260000</BUDGET>
           <CITY>Calgary</CITY>
        </project>
        <project>
             <PCODE>P004</PCODE>
             <PNAME> Promotion </PNAME>
              <BUDGET>320000</BUDGET>
               <CITY>Red Deer</CITY>
        </project>
    </ PROJ>
    <SUBSIDY>
        <role>
            <DEGREE>Ph.D</DEGREE>
        <SUB>4000</SUB>
    </role>
    <role>
        < DEGREE >MBA</DEGREE >
        <SUB>3500</SUB>
    </role>
    <role>
        < DEGREE >MSc</DEGREE >
        <SUB>2900</SUB>
    </role>
    <role>
        <DEGREE>BE</DEGREE>
        <SUB>2300</SUB>
    </role>
</SUBSIDY>
<STAFF>
    <employee>
         <SID>ID1</SID>
         <SNAME>K. Barker </SNAME>
         <DEGREE>Ph.D</DEGREE>
    </employee>

    <employee>
         <SID>ID2</SID>
         <SNAME>J. Wong</SNAME>
         <DEGREE>MBA</DEGREE>
    </employee>
    <employee>
         <SID>ID3</SID>
        <SNAME>D. Parker</SNAME>
         <DEGREE>MSc</DEGREE>
    </employee>
    <employee>
         <SID>ID4</SID>
         <SNAME>B. Unger</SNAME>
         <DEGREE>BE</DEGREE>
    </employee>
    <employee>
         <SID>ID5</SID>
         <SNAME>M. Shaw</SNAME>
         <DEGREE>MBA</DEGREE>
    </employee>
    <employee>
         <SID>ID6</SID>
         <SNAME>A. Sands</SNAME>
         <DEGREE>Ph.D</DEGREE>
        </employee>
        <employee>
            <SID>ID7</SID>
            <SNAME>C. Day</SNAME>
            <DEGREE>MSc</DEGREE>
        </employee>
        <employee>
            <SID>ID8</SID>
            <SNAME>F. Ada</SNAME>
            <DEGREE>MBA</DEGREE>
        </employee>
    </STAFF>
    <WORK_ON>
        <asgn>
           <SID>ID1</SID>
           <PCODE>P001</PCODE>
           <DAYS>11</DAYS>
        </asgn>
        <asgn>
           <SID>ID2</SID>
           <PCODE>P001</PCODE>
           <DAYS>20</DAYS>
        </asgn>
        <asgn>
           <SID>ID2</SID>
           <PCODE>P002</PCODE>
           <DAYS>5</DAYS>
        </asgn>
        <asgn>
           <SID>ID3</SID>
           <PCODE>P003</PCODE>
           <DAYS>10</DAYS>
        </asgn>
        <asgn>
           <SID>ID3</SID>
           <PCODE>P004</PCODE>
           <DAYS>35</DAYS>
        </asgn>
        <asgn>
           <SID>ID4</SID>
           <PCODE>P002</PCODE>
           <DAYS>17</DAYS>
        </asgn>
        <asgn>
           <SID>ID5</SID>
           <PCODE>P002</PCODE>
```

```
            <DAYS>24</DAYS>
        </asgn>
        <asgn>
            <SID>ID6</SID>
            <PCODE>P004</PCODE>
            <DAYS>35</DAYS>
        </asgn>
        <asgn>
            <SID>ID7</SID>
            <PCODE>P003</PCODE>
            <DAYS>30</DAYS>
        </asgn>
        <asgn>
            <SID>ID8</SID>
            <PCODE>P003</PCODE>
            <DAYS>39</DAYS>
        </asgn>
    </WORK_ON>
</company>
```

Figure 4.3 Sample XML Document xml

In Figure 4.4 and Figure 4.5 *xml1* and *xml2* are two fragmentations of *xml* in Figure 4.3. *xml1* has the projects whose budgets are greater than or equal to 200000 and *xml2* has the projects whose budgets are less than 200000.

```
<company>
    <PROJ>
          <project>
            <PCODE>P003</PCODE>
            <PNAME>CAD/CAM</PNAME>
            <BUDGET>260000</BUDGET>
            <CITY>Calgary</CITY>
          </project>
          <project>
            <PCODE>P004</PCODE>
            <PNAME>Promotion</PNAME>
            <BUDGET>320000</BUDGET>
            <CITY>Red Deer</CITY>
          </project>
    </ PROJ>
    <SUBSIDY>
    ...
    </SUBSIDY>
  <STAFF>
  ...
  </STAFF>
<company>
```

Figure 4.4 Sample XML document xml1

```
<company>
    <PROJ>
          <project>
            <PCODE>P001</PCODE>
            < PNAME>Consulting</PNAME>
            <BUDGET>160000</BUDGET>
            <CITY>Edmonton</CITY>
        </project>
          <project>
            <PCODE>P002</PCODE>
            <PNAME>Tech. Support</PNAME>
            <BUDGET>175000</BUDGET>
            <CITY>Calgary</CITY>
          </project>
    </ PROJ>
    <SUBSIDY>
    ...
    </SUBSIDY>
  <STAFF>
  ...
  </STAFF>
<company>
```

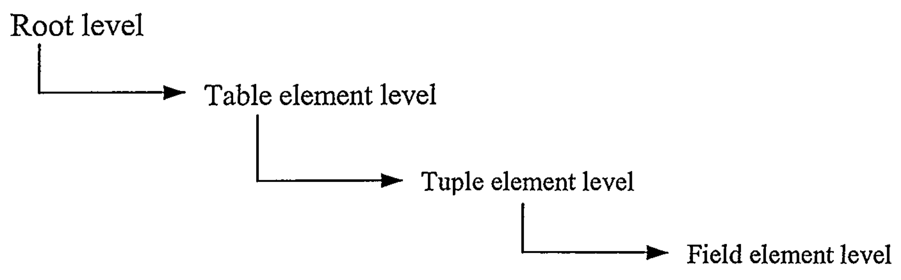
Figure 4.5 Sample XML Document xml2

4.3.2 Information Requirements of Horizontal Fragmentation

4.3.2.1 The Input Document

Currently, there are no recommended design rules for XML documents. As long as a XML document is "well formed" [XML1.0] and complies with the XML syntax, it is valid. However, a good design for XML documents is critical to the whole system as it will influence the performance of the system substantially in a distributed system.

Before presenting the algorithms for *Horizontal Fragmentation*, we first define the requirement for the input document of the algorithms presented in this chapter. Thus, the input XML document for Horizontal Fragmentation must be in $3NF_{xml}$ with the abstract structure as follows:



The algorithms fragment a XML document at the tuple element level. The result fragments are sets that are composed of some *Subtree(tuple element i)*, $1 \leq i \leq n$.

4.3.2.2 XML Schema Information

The schema information of a XML document describes the structure and constrains the contents of the XML documents. In this context it is important to note how the elements in a XML document are connected to one another. For example, from the schema document of a XML document, we know which is the key that identifies an element, the data type of an element, and the relationship between two elements. In Figure 4.7, directed *links* are drawn between elements that are related to each other by *key* and *keyref* defined in the schema document.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.cpsc.ucalgary.ca/~yingqi/xml"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:r="http://www.cpsc.ucalgary.ca/~yingqi/xml"
        elementFormDefault="qualified">
<complexType name="PROJType">
 <sequence>
  <element name="project" type="r:projectType" maxoccurs="unbounded"/>
 </sequence>
</complexType>

<complexType name="projectType" >
```

```
  <sequence>
    <element name="PCODE" type="string"/>
    <element name="PNAME" type="string"/>
    <element name="BUDGET" type="positiveInteger"/>
    <element name="CITY" type="string"/>
  </sequence>
</complexType>

<complexType name="SUBSIDYType">
  <sequence>
    <element name="role" type="r:roleType" maxoccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="roleType" >
  <sequence>
    <element name="DEGREE" type="string"/>
    <element name="SUB" type="positiveInteger"/>
  </sequence>
</complexType>

<complexType name="STAFFType">
  <sequence>
    <element name="employee" type="r:employeeType"
maxoccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="employeeType" >
  <sequence>
    <element name="SID" type="string"/>
    <element name="SNAME" type="string"/>
    <element name="DEGREE" type="string"/>
  </sequence>
</complexType>
<complexType name="WORK_ONType">
  <sequence>
    <element name="asgn" type="r:asgnType" maxoccurs="unbounded/>
  </sequence>
</complexType>
<complexType name="asgnType" >
  <sequence>
    <element name="SID" type="string"/>
    <element name="PCODE" type="string"/>
    <element name="DAYS" type="positiveInteger"/>
  </sequence>
```

```
</complexType>

<element name=”company”>
   <complexType>
      <sequence>
         <element name=”PROJ” type=”r:PROJType” minOccurs=”0”/>
         <element name=”SUBSIDY” type=”r:SUBSIDYType” minOccurs=”0”/>
         <element name=”STAFF” type=”r:STAFFType” minOccurs=”0”/>
         <element name=”WORK_ON” type=”r:WORK_ONType” minOccurs=”0”/>
      </sequence>
   </complexType>
   <key name=”projectKeyP”>
        <selector xpath=”r:PROJ/r:project”/>
        <field xpath=”PCODE”/>
   </key >
   <key name=”subsidyKeyP”>
        <selector xpath=”r:SUBSIDY/r:role”>
        <field xpath=”DEGREE”>
   </key >
   <key name=”employeeKeyP”>
        <selector xpath=”r:STAFF/r:employee”>
        <field xpath=”SID”>
   </key >
   <key name=”work_onKeyP”>
        <selector xpath=”r:WORK_ON/r:asgn”>
        <field xpath=”SID”>
        <field xpath=”PCODE”>
   </key >
   <keyref refer=”subsidyKeyP” name=”employee2subsidy”>
        <selector xpath=”r:STAFF/r:employee”/>
        <field xpath=”DEGREE”/>
   </keyref >
   <keyref refer=”employeeKeyP” name=”work_on2employee”>
        <selector xpath=”r:WORK_ON/r:asgn”/>
        <field xpath=”SID”/>
   </keyref >
   <keyref refer=”projectKeyP” name=”work_on2project”>
        <selector xpath=”r:WORK_ON/r:asgn”/>
        <field xpath=”PCODE”/>
   </keyref >
  </element>
</schema>
```

Figure 4.6 Sample XML Document xml’s Schema Document

Figure 4.7 shows the links among the XML document *xml* given in Figure 4.6. Note that the link direction shows a one-to-many relationship. For example, for each title there are multiple employees with that title; thus there is a link between **SUBSIDY** and **STAFF** elements. Similarly, there is another link between **PROJ** and **WORK_ON**.

The table element at the tail of a link is called the *owner* of the link and the table element at the head is called the *member* [CP83]. We define two functions: *owner* and *member*. Given a link, they return the owner or member elements of the link, respectively.

For example, given link L_1 of Figure 4.7, the owner and member functions have the following values:

$$owner(L_1) = \text{SUBSIDY}$$
$$member(L_1) = \text{STAFF}$$

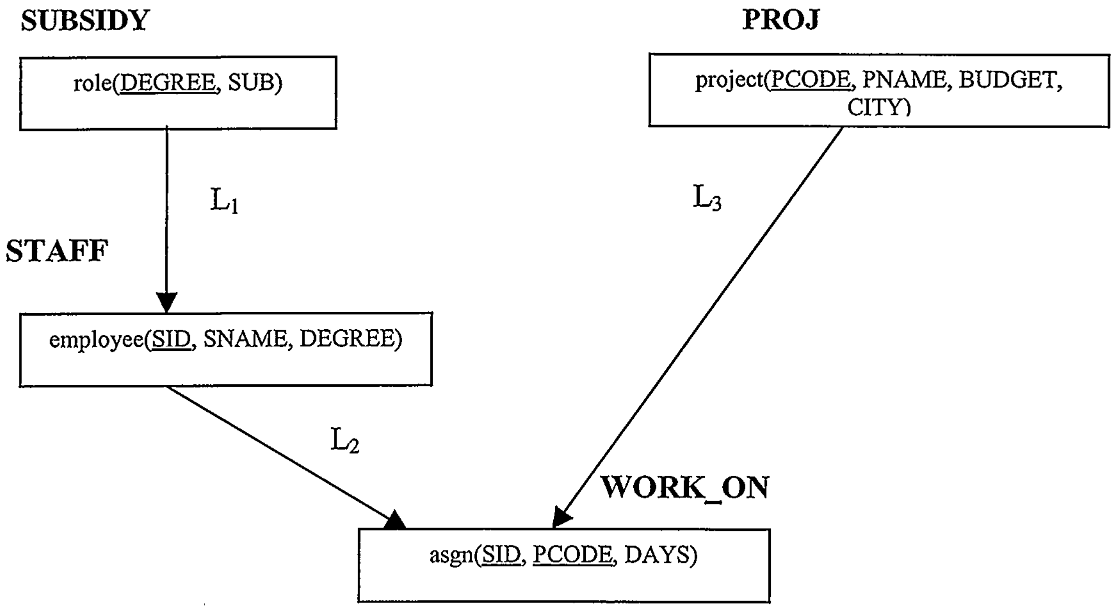


Figure 4.7 Expression of Relationships among Elements of xml

4.3.2.3 Application Information

To partition an XML document, we must know the predicates used in user queries. One should investigate at least the most active 20% of user queries, which account for 80% of the total data accesses [W82]. This "80/20 rule" may be used as a guideline in performing this analysis.

We are interested in determining simple predicates. Given a table element structure $S : T(f_1, f_2, ..., f_n)$, where $f_i \subseteq \{f_1, f_2, ..., f_n\}$ is a field element of an tuple element T_i in S defined over domain D_i, a simple predicate p_j defined on T_i has the form

$$p_j : f_i \ \theta \ Value$$

where $\theta \in \{ =, <, \neq, \lneq >, \gneq \}$ and $Value$ is chosen from the domain of f_i ($Value \in D_i$). Pr_i is used to denote the set of all simple predicates defined on S. The members of Pr_i are denoted by p_{ij} .

Example 1

Given the XML document instance *xml* of Figure 4.3,

PNAME = "Consulting"

is a simple predicate, as is

BUDGET ≤200000.

Simple predicates are combined into predicates that describe user queries in real applications, which are Boolean combinations of simple predicates. *Minterm predicates* [ÖV99] are the conjunction of simple predicates. Using minterm predicates in the design algorithms is sufficient as it is possible to transform a Boolean expression into conjunctive normal form.

Given a set $Pr_i = \{p_{i1}, p_{i2}, ..., p_{im}\}$ of simple predicates for table element S_i in a XML document, the set of minterm predicates $M_i = \{m_{i1}, m_{i2}, ..., m_{iz}\}$ is defined as

$$M_i = \{m_{ij} \mid m_{ij} = \bigwedge \ p^*_{ik}\}, 1 \leq k \leq m, 1 \leq j \leq z$$

$$p_{ik} \in Pr_i$$

where $p^*_{ik} = p_{ik}$ or $p^*_{ik} = \neg p_{ik}$. Each simple predicate can occur in minterm predicate either in its natural or its negated form.

For equality predicates, the reference to the negation of a predicate is meaningful

$$Value(element) = Value.$$

For inequality predicates, the negation should be treated as the complement. For example, the negation of the simple predicate

$$Value(element) \leq Value$$

is

$$Value(element) > Value$$

Unfortunately, there is the practical problem that the complement may be difficult to define so only simple equality predicates are considered here.

Example 2

Consider element **SUBSIDY** of Figure 4.3. The following are some of the possible simple predicates that can be defined on **SUBSIDY**.

p_1: DEGREE = "Ph.D"

p_2: DEGREE = "MBA"

p_3: DEGREE = "MSc"

p_4: DEGREE = "BE"

p_5: SUB $\leq$3000

p_6: SUB > 3000

The following are some of the minterm predicates that can be defined based on these simple predicates.

m_1 : DEGREE = "Ph.D" $\wedge$ SUB $\leq$3000

m_2 : DEGREE = "Ph.D" $\wedge$ SUB > 3000

m_3 : $\neg$ (DEGREE = "Ph.D") $\wedge$ SUB $\leq$3000

m_4 : $\neg$ (DEGREE = "Ph.D") $\wedge$ SUB > 3000

m_5 : DEGREE = "BE" $\wedge$ SUB $\leq$3000

m_6 : DEGREE = "BE" $\wedge$ SUB $\leq$3000

The above predicates are not all the minterm predicates that can be defined. Furthermore, some of these may be meaningless given the semantics of element **SUBSIDY**. Further m_3 can be written as

m_3 : DEGREE $\neq$ "Ph.D" $\wedge$ SUB $\leq$30000

In addition to analyzing the minterm predicate, we need to know the access frequency of user applications.

Access frequency: frequency with which user applications access data. If Q = {q_1, q_2,…,q_q} is a set of user queries, *acc(q_i)* indicates the access frequency of query q_i in a given period.

Minterm access frequencies can be determined from the query frequencies. The access frequency of a minterm is represented as *acc(m_i)*

4.3.2.4 Primary Horizontal Fragmentation in XML (PHF$_{xml}$)

A primary horizontal fragmentation is defined by a selection operation on the owner table elements of a XML document. Therefore, given a table element $E = \{t_1, t_2, ..., t_n\}$, where t_i, $1 \leq i \leq n$, is a tuple element under E, its horizontal fragments are given by

$$E_i = \sigma_{Fi}(E),\ 1 \leq i \leq z$$

where F_i is the selection formula used to obtain fragment E_i. Note that if F_i is in conjunctive normal form, it is a minterm predicate (m_i). The algorithm presented here demands that F_i be a minterm predicate.

Example 3

The decomposition of element PROJ into horizontal fragments $PROJ_1$ in Figure 4.4 and $PROJ_2$ in Figure 4.5 is defined as follows:

$$PROJ_1 = \sigma_{BUDGET \leq 200000}(PROJ)$$
$$PROJ_2 = \sigma_{BUDGET > 200000}(PROJ)$$

Since it is difficult to define the set of formulas if the domain of the elements participating in the selection formulas is continuous and infinite, we will consider the domain of the element(s) as limited according to the requirements of the application.

Example 4

Consider table element PROJ in *xml* of Figure 4.3. We can define the following horizontal fragments based on the project location. The resulting fragments are shown in Figure 4.8.

$$PROJ_1 = \sigma_{CITY=\text{"Edmonton"}}(PROJ)$$
$$PROJ_2 = \sigma_{CITY=\text{"Calgary"}}(PROJ)$$
$$PROJ_3 = \sigma_{CITY=\text{"Red Deer"}}(PROJ)$$

$PROJ_1$

```
  <company>
          <PROJ>
               <project>
                <PCODE>P001</PCODE>
                  <PNAME>Consulting</PNAME>
                 <BUDGET>160000</BUDGET>
                 <CITY>Edmonton</CITY>
             </project>
          </ PROJ>
      ...
  <company>
```

$PROJ_2$

```
  <company>
```

```
        <PROJ>
              <project>
                <PCODE>P002</PCODE>
                <PNAME>Tech. Support</PNAME>
                <BUDGET>175000</BUDGET>
                <CITY>Calgary</CITY>
              </project>
              <project>
                <PCODE>P003</PCODE>
                 <PNAME>Tele-Marketing</PNAME>
                 <BUDGET>260000</BUDGET>
                 <CITY>Calgary</CITY>
              </project>
        </ PROJ>
   ...
   <company>
PROJ3
   <company>
        <PROJ>
              <project>
                 <PCODE>P004</PCODE>
                 <PNAME>Promotion</PNAME>
                 <BUDGET>320000</BUDGET>
                 <CITY>Red Deer</CITY>
              </project>
        </ PROJ>
   ...
   <company>
```

Figure 4.8 Primary Horizontal Fragmentation of Sample XML Document xml

A horizontal fragment X_i of a XML document *xmldoc* consists of all the tuple elements, which satisfy a minterm predicate m_i defined on the table element being partitioned, and other table elements are not partitioned. Hence, given a set of minterm predicates M, there are as many horizontal fragments of XML document *xml* as there are minterm predicates. This set of horizontal fragments is also referred to as the set of *minterm fragments*. Therefore, the first step of any fragmentation algorithm is to determine a set of simple predicates that will form the minterm predicates.

There are two important aspects of simple predicates: *completeness* and *minimality*. A set of simple predicates Pr is said to be complete if and only if there is an equal probability of access by every application to any tuple element belonging to any minterm fragment that is defined according to Pr.

Example 5

Consider the fragmentation of element PROJ given in Example 4. If the only application that accesses PROJ accesses the tuple elements according to the location, the set is complete since each tuple element of each fragment $PROJ_i$ (Example 4) has the same probability of being accessed. If there is a second application that accesses only those project elements where the budget is less than $200,000, then Pr is not complete. Some of the elements within each $PROJ_i$ have a higher probability of being accessed due to this second application. To make the set of predicates complete, the predicates (BUDGET $\leq$ 200000, BUDGET > 200000) must be added to Pr:

Pr = {CITY="Montreal", CITY ="New York", CITY="Paris", BUDGET $\leq$200000,
BUDGET > 200000}

The second desirable property of the set of predicates is minimality. It simple states that if a predicte influences how fragmentation is performed (i.e., causes a fragment f to be further fragmented into, say, f_i and f_j), there should be at least one application that accesses f_i and f_j differently. In other words, the simple predicate should be relevant in determining a fragmentation. If all the predicates of a set Pr are relevant, Pr is *minimal*.

Özsu and Valduriez [ÖV99] describe the COM_MIN algorithm to generate a complete and minimal set of predicates *Pr'* given a set of simple predicates *Pr*. We modify this algorithm and apply it to fragment an element in a XML document. The algorithm is called COM_MINXML, which is given in Algorithm 1. To simplify the presentation, we adopt the notation as follows:

Basic Rule: fundamental rule of completeness and minimality, which states that a table element or fragment is partitioned "into at least two parts which are accessed differently by at least one application."

f_i of Pr': fragment f_i defined according to a minterm predicate defined over the predicates of *Pr'*.

```
Algorithm 1 COM_MINXML
        input: E: a table element in a XML document; Pr: set of simple predicates
                  defined on E
        output: Pr': set of simple predicates
        declare
                F: set of minterm fragments
        begin
                find a $p_i$ ε Pr such that partitions E according to the Basic Rule

                $Pr' \leftarrow p_i$
                $Pr \leftarrow Pr - p_i$
                $F \leftarrow f_i$          { $f_i$ is the minterm fragment according to $p_i$}
                do
                  begin
                    find a $p_j$ ε Pr such that $p_j$ partitions some $f_k$ of Pr' according to
                    the Basic Rule
                    $Pr' \leftarrow P_j \cup Pr'$
                    $Pr \leftarrow Pr - p_j$
                    $F \leftarrow f_j \cup F$
                    if $\exists p_k$ ε Pr' which is nonrelevant then
                      begin
                      $Pr' \leftarrow Pr' - p_k$
                      $F \leftarrow F - f_k$
                      end-if
                  end-begin
                until Pr' is complete
        end. { COM_MINXML}
```

The algorithm starts by finding a predicate that is relevant and that partition the input table element into a XML document. The **do-until** loop iteratively adds predicates to this set, ensuring minimality at each step. Hence, the set *Pr'* is both complete and minimal when the algorithm terminates.

Step 2 in the primary horizontal fragmentation is to derive the set of minterm predicates that can be defined on the predicates in set *Pr'*. These minterm predicates determine the fragments that are used as candidates in the allocation step. Since the set of minterm predicates may be quite large (exponential in the number of simple predicates), we must reduce the number of minterm predicates that need to be considered in the fragmentation, which is shown in the next step.

Step 3 in the design process is to eliminate some of the minterm fragments that may be meaningless. This elimination is performed by identifying those minterms that might be contradictory to a set of implications *I*.

The algorithm for primary horizontal fragmentation is given in Algorithm 2. The input to the algorithm *PHORIZXML* is a table element *E* that is subject to primary horizontal fragmentation, and *Pr*, which is the set of simple predicates that have been determined according to the applications defined on the table element *E*.

Algorithm 2 *PHORIZXML*

```
input: $E$: a table element in a XML document; $Pr$: set of simple predicates
        defined on $E$
output: $M$: set of minterm fragments
begin
    $Pr' \leftarrow$ COM_MINXML ($E, Pr$)
    determine the set $M$ of minterm predicates
    determine the set $I$ of implications among $p_i \varepsilon Pr'$
    for each $m_i \varepsilon M$ do
        if $m_i$, is contradictory according to $I$ then
            $M \leftarrow M - m_i$
        end-if
    end-for
end. { PHORIZXML }
```

Example 6

Consider the design of the table element structures given in Figure 4.7. There are two possible table elements on which primary horizontal fragmentation can be performed, namely SUBSIDY and PROJ table elements.

Suppose there is only one application that accesses SUBSIDY. This application checks the salary information and determines a raise accordingly. Assume that employee records are managed in two sites, one handling the records with salaries less than or equal to \$3,000. Therefore, the query is issued at two sites.

The simple predicates that would be used to partition table element SUBSIDY are:

p_1: SUB ≤ 3000

p_2: SUB > 3000

giving the initial set of simple predicates $Pr = \{ p_1, p_2 \}$. After applying the COM_MINXML algorithm with $i = 1$ as initial value, we get $Pr' = \{p_1\}$. Pr' now is complete and minimal since p_2 would not partition f_1 (the minterm fragment formed with respect to p_1) according to *Basic Rule*. Then the minterm predicates as members of M are:

m_1: SUB ≤ 30000

m_2: $\neg$(SUB ≤ 30000) = SUB > 30000

According to M, we have two fragments $X_f = \{F_1, F_2\}$(see Figure 4.9).

F_1

```
<SUBSIDY>
  <role>
    <DEGREE >MSc</DEGREE>
    <SUB>2900</SUB>
  </role>
  <role>
    <DEGREE>BE</DEGREE>
    <SUB>2300</SUB>
   </role>
</SUBSIDY>
```

F_2

```
<SUBSIDY>
  <role>
    <DEGREE >Ph.D</DEGREE>
    <SUB>4000</SUB>
  </role>
  <role>
    <DEGREE>MBA</DEGREE>
    <SUB>3500</SUB>
   </role>
</SUBSIDY>
```

Figure 4.9 Horizontal Fragmentation of Table Element SUBSIDY

4.3.2.5 Derived Horizontal Fragmentation in XML (DHF_{xml})

A derived horizontal fragmentation is defined on a member table element of a link (see Figure 4.7) according to a selection operation carried out on its owner. We want to partition a member table element according to the fragmentation of its owner. This can be implemented by means of semi-join [XML-QL] that is similar to that of relational databases. The resulting fragments are defined only on the field elements of the member table elements. We use "$\ltimes$" to denote the semi-join operator.

Given a link L where $owner(L) = O$ and $member(L) = M$, the derived horizontal fragments of M are defined as

$$M_i = M \ltimes O_i,\ 1 \leq i \leq z$$

where z is the maximum number of fragments that will be defined on M, and $O_i = \sigma_{F_i}(O)$, where F_i is the formula according to which the primary horizontal fragments O_i is defined and $\ltimes$ is a semi-join operation.

Example 7

Consider L_1 in Figure 4.7, where $owner(L_1)$ = SUBSIDY and $member(L_1)$ = STAFF. **STAFF** can be arranged into two groups according to their subsidy: those having subsidy less than or equal to \$3,000, and those having more than \$3,000. The two fragments $STAFF_1$ and $STAFF_2$ are defined as follows:

$$STAFF_1 = \mathit{STAFF} \ltimes SUBSIDY_1$$

$$STAFF_2 = \mathit{STAFF} \ltimes SUBSIDY_2$$

where

$$SUBSIDY_1 = \sigma_{SUB \leq 3000}(SUBSIDY)$$
$$SUBSIDY_2 = \sigma_{SUB > 3000}(SUBSIDY)$$

The result of this fragmentation is shown in Figure 4.10

STAFF$_1$

```
<STAFF>
  <employee>
    <SID>ID3</SID>
    <SNAME>D. Parker</SNAME>
    <DEGREE>MSc</DEGREE>
  </employee>
  <employee>
    <SID>ID4</SID>
    <SNAME>B. Unger</SNAME>
    <DEGREE>BE</DEGREE>
  </employee>
  <employee>
    <SID>ID7</SID>
    <SNAME>R. Davis</SNAME>
    <DEGREE>MSc</DEGREE>
  </employee>
</STAFF>
```

STAFF$_2$

```
<STAFF>
  <employee>
    <SID>ID1</SID>
    <SNAME>K. Barker</SNAME>
    <DEGREE>Ph.D</DEGREE>
  </employee>
  <employee>
    <SID>ID2</SID>
    <SNAME>J. Wong</SNAME>
    <DEGREE>MBA</DEGREE>
  </employee>
  <employee>
    <SID>ID5</SID>
    <SNAME>M. Shaw</SNAME>
    <DEGREE>MBA</DEGREE>
  </employee>
  <employee>
    <SID>ID6</SID>
    <SNAME>A. Sand</SNAME>
    <DEGREE>Ph.D</DEGREE>
  </employee>
  <employee>
    <SID>ID8</SID>
    <SNAME>F. Ada</SNAME>
    <DEGREE>MBA</DEGREE>
  </employee>
</STAFF>
```

Figure 4.10 Horizontal Fragmentation of the Table Element SUBSIDY

4.4 Summary

This chapter presented the data fragmentation problem in distributed XML database systems. The formal data model to describe a XML document is provided in Section 4.2. The fragmentation of a XML document is defined. Section 4.3 discusses the fragment strategy for a XML document. Horizontal fragmentation for a XML document is introduced. Algorithms for primary and derived horizontal fragmentation in XML are presented. The goal of this chapter is to show how to adapt the algorithms for horizontal fragmentation in the distributed relational databases to XML.

Chapter 5

Extended Fragmentation Strategy for XML

The last chapter introduced XML document design in the relational framework for XML. No nested elements are allowed to occur under a tuple element. However, nested structure is one main feature of the XML documents. To respect this feature and enhance query performance, this chapter extends that model to potentially accommodate nested elements. Section 5.1 discusses the denormalization of table elements. Nested table elements are presented in Section 5.2. Horizontal fragmentation for nested table elements is provided in Section 5.3. Section 5.4 compares this chapter's fragmentation mechanism with the one in last chapter. Section 5.5 summarizes this chapter.

5.1 Denormalization of TBEs

In relational databases, normalization optimizes updates at the expense of retrievals when retrieving related records requires accessing different tables. This is why denormalizing a relational design from higher normal forms can enhance performance by reducing join operations between related tables. This is the primary motivation for denormalizing relations in a datawarehouse.

Normalization for XML documents is described in Chapter 3. However, denormalizing a XML document from third normal form may have advantages so we consider how this could occur. The first reason is that join operations between XML elements is under investigation but is not fully understood at this time. Second, update functions for XML documents are still a problematic part of XML. Most applications may only retrieve data stored in a XML document. Therefore, retrieval performance should be given priority. Third, typically a XML element is "fact-oriented", which means that XML designers intend to put all related information as different elements under a certain element representing an object in the real world. In this section, the denormalization of table elements is discussed. The fragmentation algorithms for denormalized TBEs and nested TBEs are provided.

The *denormalization* of TBEs is the inverse procedure of normalizing a table element resulting in a lower normal form. There are many ways to denormalize table elements. However, only merging two table elements in $3NF_{xml}$ having one-to-many relationship is considered in this thesis. The denormalization process is required when there is a need to perform primary or derived horizontal fragmentation on a table element.

In Figure 4.7, the link shows the relationship among the table elements in $3NF_{xml}$. The link L_1 indicates that there is a one-to-many relationship between **SUBSIDY** and **STAFF**. For each degree in **SUBSIDY** there are multiple staff members with that degree in **STAFF**. **SUBSIDY** can be placed into **STAFF** leading to a new structure for the table element **New_STAFF** shown in Figure 5.1.

After the denormalization process shown in Figure 5.1, the field element "SUB" of **SUBSIDY** becomes a field element under **STAFF**. The new table element **STAFF** is in $2NF_{xml}$. In Example 7, **SUBSIDY** is fragmented into two fragments according to the predicates: "SUB $\leq$3000" and "SUB > 3000". Derived fragmentation is then performed on **STAFF** by semi-joining the two fragments of **SUBSIDY** and **STAFF.** Since the field element "SUB" is under **STAFF** now, no semi-join is required. Predicates: "SUB $\leq$

3000" and "SUB > 3000" can be directly applied to fragmenting **STAFF**. The result is shown in Figure 5.2.

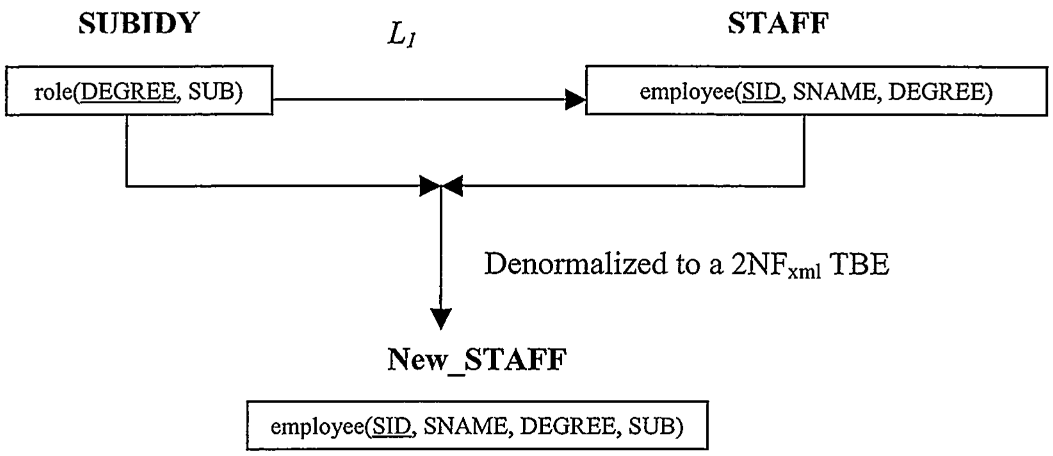

SUBIDY
L_1
STAFF
role(DEGREE, SUB)
employee(SID, SNAME, DEGREE)
Denormalized to a $2NF_{xml}$ TBE
New_STAFF
employee(SID, SNAME, DEGREE, SUB)

Figure 5.1 Denormalization of SUBSIDY and STAFF

There are two processes to denormalize two TBEs connected by a link. First, a new table element structure is created by merging two table element structures of the linked TBEs. Second, a new table element with data is generated by taking the two linked TBEs as input.

5.1.1 The Algorithm For Creating New TBE Structure

The template to convert table element structures to a XML schema document is presented in Chapter 3. Thus, once the table element structure(s) is provided, to get the XML schema is a trivial procedure. Given two linked TBEs, a new table element structure can be generated using the following algorithm:

Algorithm 3a *DenormTBESTR*

```
input:      $ES_1$: the *owner* table element structure of a *link* $L_i$;
        $ES_2$: the *member* table element structure of *link* $L_i$;
 output: $R$: a table element structure
begin
        $R \leftarrow ES_2$
        for each $f_j \varepsilon ES_1$ do    { $f_j$ is the tag name of a field element in $ES_1$}
            if $f_j$ is not the primary key then
                $ES_2 \leftarrow f_j \cup ES_2$
                $ES_1 \leftarrow ES_1 - f_j$
            end-if
        end-for
    end. { DenormTBESTR }
```

STAFF$_1$

```
<STAFF>
  <employee>
    <SID>ID3</SID>
    <SNAME>D. Parker</SNAME>
    <DEGREE>MSc</DEGREE>
    <SUB>2900</SUB>
  </employee>
  <employee>
    <SID>ID4</SID>
    <SNAME>B. Unger</SNAME>
    <DEGREE>BE</DEGREE>
    <SUB>2300</SUB>
  </employee>
  <employee>
    <SID>ID7</SID>
    <SNAME>R. Davis</SNAME>
    <DEGREE>MSc</DEGREE>
    <SUB>2900</SUB>
  </employee>
</STAFF>
```

STAFF$_2$

```
<STAFF>
  <employee>
    <SID>ID1</SID>
    <SNAME>K. Barker</SNAME>
    <DEGREE>Ph.D</DEGREE>
    <SUB>4000</SUB>
  </employee>
  <employee>
    <SID>ID2</SID>
    <SNAME>J. Wong</SNAME>
    <DEGREE>MBA</DEGREE>
    <SUB>3500</SUB>
  </employee>
  <employee>
    <SID>ID5</SID>
    <SNAME>M. Shaw</SNAME>
    <DEGREE>MBA</DEGREE>
    <SUB>3500</SUB>
  </employee>
  <employee>
    <SID>ID6</SID>
    <SNAME>A. Sand</SNAME>
    <DEGREE>Ph.D</DEGREE>
    <SUB>4000</SUB>
  </employee>
  <employee>
    <SID>ID8</SID>
    <SNAME>F. Ada</SNAME>
    <DEGREE>MBA</DEGREE>
    <SUB>3500</SUB>
  </employee>
</STAFF>
```

Figure 5.2 Horizontal Fragmentation of TBE STAFF after denormalization

Figure 5.1 shows the process of denormalization for **STAFF** and **SUBSIDY**. **STAFF** is member of the link L_1 and **SUBSIDY** is the owner of the link L_1. Applying the *DenormTBESTR* algorithm by assigning the value "STAFF: employee(SID, SNAME, DEGREE)" to R. There are two field element names "DEGREE" and "SUB" in the TBE structure of **SUBSIDY**. Since "DEGREE" is the primary key, only "SUB" is appended to R resulting in the new table element structure as "STAFF:employee(SID, SNAME, DEGREE, SUB)".

5.1.2 The Algorithm For Merging Two Linked TBEs with Data

Another process of denormalizing two linked TBEs is to merge the two TBEs. A generic algorithm for this process is as follows:

Algorithm 3 *DenormTBE*

```
input: $E_1$: the owner table element of a link $L_i$;
       $E_2$: the member table element of link $L_i$;
       $k$: the primary key of $E_1$;
       $fk$: the foreign key of $E_2$ referencing $k$ of $E_1$
output: $E_2'$ : a table element in $2NF_{xml}$
begin
    $E_2' \leftarrow E_2$
    for each $t_i \in E_2'$ do          { $t_i$ is a tuple element of $E_2'$ }
      for each $t_j \in E_1$ do           { $t_j$ is a tuple element of $E_1$ }
        if $fk = k$ then
          for each $f_s \in t_j$ do        { $f_s$ is a field element of $t_j$ }
            if $f_s$ is not primary key or part of primary key then
              $t_i \leftarrow f_s \cup t_i$
              $t_j \leftarrow t_j - f_s$
            end-if
          end-for
        end-if
      end-for
    end-for
end. { DenormTBE }
```

5.1.3 The Algorithm For Fragmenting The Denormalized TBE

The algorithm for fragmenting a TBE in $2NF_{xml}$, which is denormalized by merging two table elements having one-to-many relationship, is as follows:

Algorithm 4 *HorizDenormTBE*

```
input: $E_1$: the owner table element of a link $L_i$ ;
       $E_2$: the member table element of link $L_i$;
       $M$: a set of minterm predicates defined on $E_1$
 output: $N$: a set of fragments
declare
       $E$: a table element
begin
       $E \leftarrow DenormTBE(E_1, E_2)$
       Determine $N$ by applying $M$ on $E$
end. { HorizDenormTBE }
```

Example 8

Let us consider merging two TBEs **STAFF** and **SUBSIDY** in the document ***xml*** given in Figure 4.3 using *DenormTBE*. E_1 and E_2 will be **SUBSIDY** and **STAFF**, respectively. By applying the *DenormTBE* algorithm, E_2' is assigned all elements in **STAFF**. The algorithm runs the outer loop with the first tuple element of E_2' , whose degree is "Ph.D". Next, go through the tuple elements in **SUBSIDY** to find the tuple element with the degree value of "Ph.D" and copy this tuple's field element "SUB" to the first tuple element of E_2' . Do the same to all tuple elements of E_2' . The result is:

New_STAFF

```
<STAFF>
  <employee>
     <SID>ID1</SID>
     <SNAME>K. Barker</SNAME>
     <DEGREE>Ph.D</DEGREE>
     <SUB>4000</SUB>
  </employee>
  <employee>
     <SID>ID2</SID>
     <SNAME>J. Wong</SNAME>
     <DEGREE>MBA</DEGREE>
     <SUB>3500</SUB>
  </employee>
  <employee>
     <SID>ID3</SID>
     <SNAME>D. Parker</SNAME>
     <DEGREE>MSc</DEGREE>
     <SUB>2900</SUB>
  </employee>
  <employee>
     <SID>ID4</SID>
     <SNAME>B. Unger</SNAME>
     <DEGREE>BE</DEGREE>
     <SUB>2300</SUB>
  </employee>
  <employee>
     <SID>ID5</SID>
     <SNAME>M. Shaw</SNAME>
     <DEGREE>MBA</DEGREE>
     <SUB>3500</SUB>
  </employee>
  <employee>
     <SID>ID6</SID>
     <SNAME>A. Sand</SNAME>
     <DEGREE>Ph.D</DEGREE>
     <SUB>4000</SUB>
  </employee>
  <employee>
     <SID>ID7</SID>
     <SNAME>R. Davis</SNAME>
     <DEGREE>MSc</DEGREE>
     <SUB>2900</SUB>
  </employee>
  <employee>
     <SID>ID8</SID>
     <SNAME>F. Ada</SNAME>
     <DEGREE>MBA</DEGREE>
     <SUB>3500</SUB>
  </employee>
</STAFF>
```

After getting **New_STAFF**, the fragmentation can be performed on it according to the set of minterm predicates: "SUB $\leq$3000" and "SUB > 3000" based on the *HorizDenormTBE* algorithm. The result is shown in Figure 5.2.

5.2 Nested TBEs

In Chapter 3, the formal model and table element (TBE) definition were given. Under a tuple element of a TBE, only field elements are allowed to occur. In other words, no nested structure is permitted under a tuple element. However, a significant feature of XML is its nested structure. A XML document with a nested structure is easy to read for designers and reflects the hierarchical structure well. However, the XML documents are designed to be read by applications (programs) other than human beings. Hence, while designing a XML document, there is a trade-off between nesting and flat structures. The challenge is to make a XML document nest properly. In this section, the TBE model is extended to form a new data structure-nested TBE.

A *nested tuple element* (NTPE) is a tuple element having tuple elements under it. A *nested TBE* (NTBE) is a table element having nested tuple elements. A **nested table element structure** NS, denoted by $NS: NT(F_1, F_2, ..., T_m(f_1, f_2, ..., f_k), ..., F_n)$, is composed of a nested table element name NS, a list of field element $F_1, F_2, ..., F_n$, and some tuple element T_m consisting of field element $f_1, f_2, ..., f_k$ in its nested tuple elements having the tag name NT. The abstract structure for a XML document having NTBEs is:

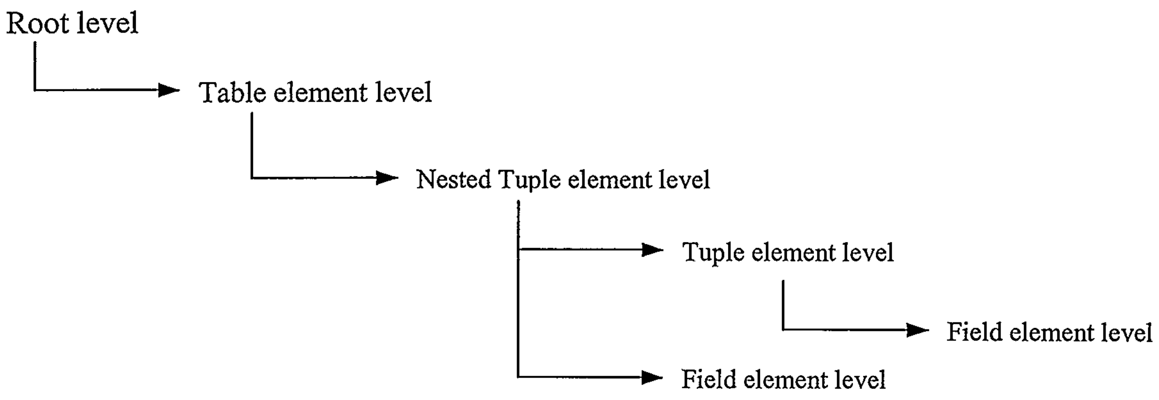


Figure 5.3 shows an example of a nested table element. In the NTBE **STAFF,** a tuple element "SUBSIDY" occurs under the tuple element "employee". Its structure is represented as *STAFF : employee(SID, SNAME, SUBSIDY(DEGREE, SUB))*.

```
<STAFF>
  <employee>
    <SID>ID1</SID>
    <SNAME>K. Barker</SNAME>
    <SUBSIDY>
      <DEGREE>Ph.D</DEGREE>
```

```
        <SUB>4000</SUB>
    </SUBSIDY>
</employee>
<employee>
    <SID>ID2</SID>
    <SNAME>J. Wong</SNAME>
    <SUBSIDY>
        <DEGREE>MBA</DEGREE>
        <SUB>3500</SUB>
    </SUBSIDY>
</employee>
<employee>
    <SID>ID3</SID>
    <SNAME>D. Parker</SNAME>
    <SUBSIDY>
        <DEGREE>MSc</DEGREE>
        <SUB>2900</SUB>
    </SUBSIDY>
</employee>
<employee>
    <SID>ID4</SID>
    <SNAME>B. Unger</SNAME>
    <SUBSIDY>
        <DEGREE>BE</DEGREE>
        <SUB>2300</SUB>
    </SUBSIDY>
</employee>
<employee>
    <SID>ID5</SID>
    <SNAME>M. Shaw</SNAME>
    <SUBSIDY>
        <DEGREE>MBA</DEGREE>
        <SUB>3500</SUB>
    </SUBSIDY>
</employee>
<employee>
    <SID>ID6</SID>
    <SNAME>A. Sand</SNAME>
    <SUBSIDY>
        <DEGREE>Ph.D</DEGREE>
        <SUB>4000</SUB>
    </SUBSIDY>
</employee>
<employee>
    <SID>ID7</SID>
    <SNAME>R. Davis</SNAME>
    <SUBSIDY>
        <DEGREE>MSc</DEGREE>
        <SUB>2900</SUB>
    </SUBSIDY>
</employee>
```

```
    <employee>
      <SID>ID8</SID>
      <SNAME>F. Ada</SNAME>
      <SUBSIDY>
        <DEGREE>MBA</DEGREE>
        <SUB>3500</SUB>
      </SUBSIDY>
    </employee>
  </STAFF>
```

Figure 5.3 An Example of a Nested Table Element

Given a NTBE having the structure $NS : NT(F_1, F_2, ..., T_m(f_1, f_2, ..., f_k), ..., F_n)$, converting this structure to the XML schema is accomplished with the following template:

Template 3

```
<complexType name="S">
  <sequence>
    <element name="T" type="r: T" maxoccurs="unbounded/>
  </sequence>
</complexType>
<complexType name="T" >
  <sequence>
    <element name="F1" type="DataType1"/>
    <element name="F2" type="DataType2"/>
        ...
   <element name="Tm" type="r: NT"/>
        ...
    <element name="Fn" type="DataTypen"/>
 </sequence>
</complexType>
<complexType name="NT" >
 <sequence>
    <element name="f1" type="DataType1"/>
    <element name="f2" type="DataType2"/>
        ...
    <element name="fn" type="DataTypen"/>
  </sequence>
</complexType>
```

5.3 Horizontal Fragmentation of Nested TBEs

Given the link L_1 in Figure 4.7, there are two options to denormalize the table elements **SUBSIDY** and **STAFF**. One option is discussed in Section 4.4.1. Another choice is to replace the field element "DEGREE" in **SUBSIDY** with the related tuple element in **STAFF** resulting in a new **STAFF**, which is a nested TBE as shown in Figure 5.3. To make it more meaningful, the tuple element's tag "role" in **SUBSIDY** is changed to its TBE's tag name "SUBSIDY" in the resulting **New_STAFF.**

5.3.1 The Algorithm for Generating the Structure of a Denormalized NTBE

As in the process of denormalizing two linked TBEs into a TBE in $2NF_{xml}$, merging two linked TBEs into a NTBE consists of two processes as well. One is to get the structure of the NTBE and the other is to merge the TBEs with their data. The algorithm to get the structure of a NTBE is as follows:

Algorithm 5a *DenormNTBESTR*

```
input: $ES_1$: $T_1$ ($k$, $F_2$, ..., $F_n$), the owner table element structure of a link $L_i$;
       $ES_2$: $T_2$ ($F_1$, $fk$, ..., $F_n$), the member table element structure of a link
            $L_i$;
       $k$: the primary key of $ES_1$ ;
       $fk$: the foreign key of $ES_2$ referencing $k$ of $ES_1$
output: $R$: the structure of a NTBE
begin
       $fk \leftarrow$ String ("$ES_1$ ($k$, $F_2$, ..., $F_n$)")        (String() is a string function)
       $R \leftarrow$ String("$ES_2$: $T_2$ ($F_1$, $fk$, ..., $F_n$)")
       end. { DenormNTBESTR }
```

Figure 5.4 shows the process of denormalizing **STAFF** and **SUBSIDY** into a NTBE. **STAFF** is the member of the link L_1 and **SUBSIDY** is the owner of the link L_1. Apply the

DenormNTBESTR algorithm with replacing the foreign key "DEGREE" of **STAFF** with the value "SUBSIDY(DEGREE, SUB)". Next, assign the value "STAFF: employee(SID, SNAME, SUBSIDY(DEGREE, SUB))" to *R*. *R* is a new NTBE structure generated by merging **STAFF** and **SUBSIDY**.

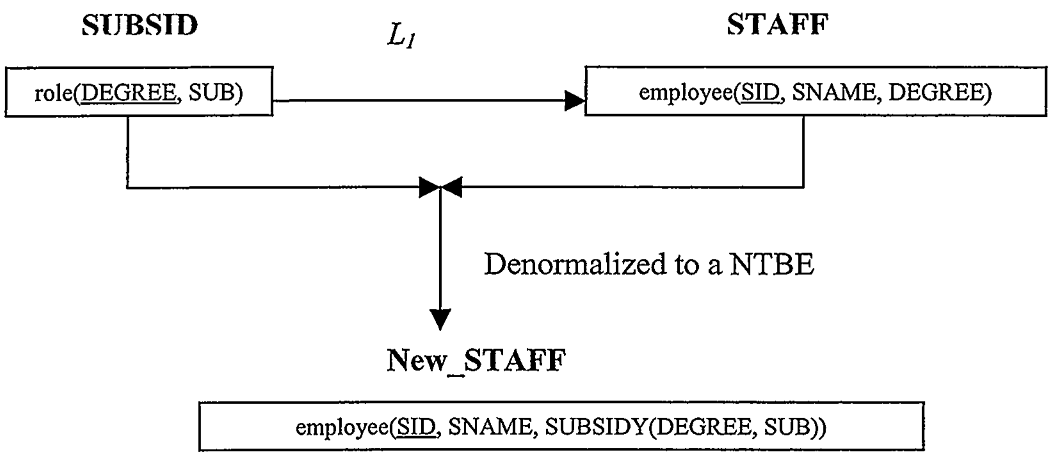

SUBSID
L_l
STAFF
role(DEGREE, SUB)
employee(SID, SNAME, DEGREE)
Denormalized to a NTBE
New_STAFF
employee(SID, SNAME, SUBSIDY(DEGREE, SUB))

Figure 5.4 Denormalizing Two Linked TBEs into a NTBE

5.3.2 The Algorithm for Merging Two Linked TBEs with Data

The other process of denormalizing two linked TBEs into a NTBE is to merge the data of the two TBEs. A generic algorithm for this process is as follows:

Algorithm 5 *DenormNTBE*

```
input: $E_1$: the owner table element of a link $L_i$;
       $E_2$: the member table element of link $L_i$;
       $k$: the primary key of $E_1$;
       $fk$: the foreign key of $E_2$ referencing $k$ of $E_1$
output: $E_2'$ : a nested table element
begin
    $E_2' \leftarrow E_2$
    for each $t_i \varepsilon E_2'$ do                 { $t_i$ is a tuple element of $E_2'$ }
        for each $t_j \varepsilon E_1$ do              { $t_j$ is a tuple element of $E_1$}
            if $fk = k$ then
                replace $fk$ with $t_j$
                change $t_j$'s tag name to the tag name of $E_1$
                end-if
            end-for
        end-for
    end. { DenormNTBE }
```

Example 9

The new nested table element **New_STAFF** (Figure 5.3) is generated after applying *DenormNTBE* on **SUBSIDY** and **STAFF**. First, E_2' is assigned with all elements in **STAFF**. The algorithm then runs the outer loop with the first tuple element of E_2', whose degree is "Ph.D"; next, go through the tuple elements in **SUBSIDY** to find the tuple element with the degree value of "Ph.D"; replace the field element "DEGREE" under the first tuple element of E_2' with the found tuple element in **SUBSIDY** and change its tag name "role" to "SUBSIDY". Repeat this procedure for the other tuple elements of E_2'. The result is shown in Figure 5.3,

The generic algorithm for fragmenting a NTBE denormalized using *DenormNTBE* is presented as follows:

```
Algorithm 6 HorizDenormNTBE
    input: $E_1$: the owner table element of a link $L_i$;
           $E_2$: the member table element of link $L_i$; $M$: a set of minterm
               predicates defined on $E_1$
    output: $N$: a set of fragments
    declare
           $E$: a nested table element
    begin
           $E \leftarrow DenormNTBE(E_1, E_2)$
           Determine $N$ by applying $M$ on $E$
    end. { HorizDenormNTBE }
```

In Example 7 (see Chapter 4), first, **SUBSIDY** is fragmented into two fragments according to the predicates: "SUB ≤3000" and "SUB > 3000". By then semi-joining **STAFF** with **SUBSIDY**, we get two staff groups according to their subsidy: those having subsidy less than or equal to \$3,000, and those having more than \$3,000. Now **SUBSIDY** becomes an element nested in **STAFF**. Thus predicates: "SUB ≤3000" and "SUB >

3000" can be directly applied to fragmenting **NEW_STAFF** by applying the *HorizDenormNTBE* algorithm. The result is shown in Figure 5.5.

STAFF$_1$

```
<STAFF>
  <employee>
     <SID>ID3</SID>
     <SNAME>D. Parker</SNAME>
     <SUBSIDY>
        <DEGREE>MSc</DEGREE>
        <SUB>2900</SUB>
     </SUBSIDY>
  </employee>
  <employee>
     <SID>ID4</SID>
     <SNAME>B. Unger</SNAME>
     <SUBSIDY>
        <DEGREE>BE</DEGREE>
        <SUB>2300</SUB>
     </SUBSIDY>
  </employee>
  <employee>
     <SID>ID7</SID>
     <SNAME>R. Davis</SNAME>
     <SUBSIDY>
        <DEGREE>MSc</DEGREE>
        <SUB>2900</SUB>
     </SUBSIDY>
  </employee>
</STAFF>
```

STAFF$_2$

```
<STAFF>
  <employee>
     <SID>ID1</SID>
     <SNAME>K. Barker</SNAME>
     <SUBSIDY>
        <DEGREE>Ph.D</DEGREE>
        <SUB>4000</SUB>
     </SUBSIDY>
  </employee>
  <employee>
     <SID>ID2</SID>
     <SNAME>J. Wong</SNAME>
     <SUBSIDY>
        <DEGREE>MBA</DEGREE>
        <SUB>3500</SUB>
     </SUBSIDY>
  </employee>
  <employee>
     <SID>ID5</SID>
     <SNAME>M. Shaw</SNAME>
     <SUBSIDY>
        <DEGREE>MBA</DEGREE>
        <SUB>3500</SUB>
     </SUBSIDY>
  </employee>
  <employee>
     <SID>ID6</SID>
     <SNAME>A. Sand</SNAME>
     <SUBSIDY>
        <DEGREE>Ph.D</DEGREE>
        <SUB>4000</SUB>
     </SUBSIDY>
  </employee>
  <employee>
     <SID>ID8</SID>
     <SNAME>F. Ada</SNAME>
     <SUBSIDY>
        <DEGREE>MBA</DEGREE>
        <SUB>3500</SUB>
     </SUBSIDY>
  </employee>
</STAFF>
```

Figure 5.5 Horizontal Fragmentation of NTBE STAFF

5.4 Primary and Derived Horizontal Fragmentation vs. Fragmentation on the Denormalized TBE and NTBE

In Section 4.3.2.4 and Section 4.3.2.5, primary and derived horizontal fragmentation are discussed with the assumption that all TBEs are in $3NF_{xml}$. In this case, primary horizontal fragmentation is performed on the *owner* of a link where derived horizontal fragmentation is performed on the *member* of the link. The result of PHF_{xml} determines the output of DHF_{xml}.

In terms of fragmentation on denormalized TBE and NTBE, there is no primary and derived horizontal fragmentation needed. Since the two linked TBEs are merged to either one TBE in $2NF_{xml}$ or a NTBE, fragmenting the denormalized TBE or NTBE is the same as performing primary and derived horizontal fragmentation at the same time.

5.5 Summary

In this chapter, both denormalization and the extended model for table elements – NTBE are considered. The denormalization procedure and the algorithm to create a new TBE structure are given in Section 5.1. The algorithm to merge two linked TBEs is presented. Section 5.1.3 gives the algorithm for fragmenting a denormalized TBE. NTBE is defined in Section 5.2. Algorithms for generating the structure of a NTBE and the NTBE with data are provided in Section 5.3. Finally, the fragmentation algorithm for a NTBE is discussed. The next chapter presents the conclusion and the summary of various concepts discussed in this dissertation and sets some future research directions.

Chapter 6

Conclusions and Future Work

This chapter summarizes the contributions of this dissertation and presents directions for future work.

6.1 Summary of Contributions

This thesis identifies the following problems in designing and fragmenting XML documents in XML database systems. A mechanism to adapt relational techniques to XML document design is presented.

- Absence of a mechanism to evaluate and design XML documents.

  The current available XML specifications lack a mechanism to evaluate what constitutes a good design for XML documents. No rules to follow when designing XML documents results in numerous ill-designed documents with severe redundancy. This will inevitable affect the performance of queries executed on those XML documents. A mechanism to present design rules for XML documents can reduce redundancy and enhance query performance.

- Absence of a mechanism to adapt relational techniques to XML.

  Though there have been numerous proposals and mechanisms for conversions between relational databases and XML, limited work has been done to adapt relational techniques to XML. This is due to the heterogeneity of the data models (relational databases and XML). A mechanism to adapt relational techniques to XML is very imperative when addressing data and transaction management problems in XML databases.

- Absence of a mechanism to fragment XML data in a distributed environment.

  Limited attention has been paid to fragment XML data in the past. XML is invented for data exchange and applications on the web. The Internet is the largest distributed system in the world. Therefore a mechanism to fragment XML data is desirable when considering the distributed environment for XML, especially for a distributed XML database.

This research analyzed the above problems and/or challenges and addressed them by adapting relational design techniques to construct XML documents. Further through the proposed design models, data fragmentation for XML documents becomes possible. The implementation provides a framework that can be utilized to solve data fragmentation problems in a DXDB.

- **Establish Design Rules for XML:** Driving force is initiated to set up design rules for XML documents. This thesis identifies some design issues for XML documents, especially for XML databases. The redundancy and query execution problems are discussed. Design rules for a relational model are used to evaluate and design XML documents. The design issues for XML documents are discussed in Chapter 2.

- **Relational Framework for XML:** Considering a XML database as the underlying system, table, tuple, and field elements are defined and used to construct a XML document. Constraints for the document are specified in the XML Schema documents. Since this mechanism makes XML data relational, some mature relational techniques can be applied to handle these data, such as data and transaction management. Design procedures are provided for XML documents. Normal forms are defined for XML, which then can be used to reduce redundancy of XML documents. Further, within this relational framework, fragmentation of XML documents can be realized. It is worth to note that our implementation treats native XML documents directly with no conversions between relational tables and XML.

- **Fragmentation of XML documents:** No work has been done with data fragmentation problems in a distributed XML database. A XML document in the relational framework can be fragmented by the relevant algorithms proposed in this dissertation. Horizontal fragmentation for XML documents is considered. Beyond the relational framework, a table element with nested tuple elements is discussed. This is desirable when derived horizontal fragmentation is required.

6.2 Future Directions

There are several interesting directions in which the work presented in this dissertation can proceed. The future work suggested here is based on this work coupled with directions to address the general problem of XML document design with respect to distributed XML database systems.

Attributes, PIs, and Comments: This thesis focuses on data-centric XML documents and some other features of XML, such as attributes, PIs, and comments are not considered. Although these entities will not substantially change the design model, we may facilitate the design by adding them to the XML document.

Nesting Levels for TBE: For the nested table element, only one level nesting is allowed under a nested tuple element. If there are too many nesting levels, redundancy becomes a prominent problem. Since the architecture model for XML databases and XML are all under investigation, it is hard to decide nesting levels for XML documents.

Vertical Fragmentation: This thesis considers horizontal fragmentation for XML documents. However, vertical fragmentation is also valid for XML documents. The challenge would be if we need that and how it will influence the performance when executing queries.

Bibliography

[Abit+03] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, T. Milo, "Dynamic XML Documents with Distribution and Replication", *SIGMOD 2003*.

[Abit+03] M. Arenas and L. Libkin, "A Normal Form for XML Documents", *PODS'02*, pp.85-96.

[Abit+03] M. Arenas and L. Libkin, "An Information-Theoretic Approach to Normal Forms for Relational and XML Data", *PODS'03*.

[Bert+99] E. Bertino, S. Castano, E. Ferrari, M. Mesiti, "Controlled Access and dissemination of XML documents", *Proc. 2nd International Workshop on Web Information and Data Management,* 1999, pp.22-27

[Bosa01] Jon Bosak, "The birth of XML: A personal Recollection", 2001, http://java.sun.com/xml/birth_of_xml.html.

[Bour02] Ronald Bourret, "XML and Databases". http://www.rpbourret.com/xml /XMLAndDatabases.htm

[Boye01] J. Boyer, "Canonical XML Version 1.0", *W3C Recommendation*, March 15, 2001, http://www.w3.org/TR/2001/REC-xml-c14n-20010315.

[Brem+03] J. Bremer and M. Gertz, "On Distributing XML Repositories", *International Workshop on the Web and Databases*, June, 2003, pp.73-78.

[Chau+03] Akmal B. Chaudhri, Awais Rashid, Roberto Zicari, "XML Data Management: Native XML and XML-Enabled Database Systems". *Pearson Education, Inc.*

[Clar+99] J. Clark, S. DeRose, "XML Path Language (Xpath) version 1.0", *W3C Recommendation,* November 16, 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[Conr01] Andrew Conrad, "A Survey of Microsoft SQL Server 2000 XML Features". http://msdn.microsoft.com/liberary/en-us/dnexxml/html/xml07162001.asp.

[Cowa+01] R. Cowan, J. Tobin, "XML Information Set", *W3C Proposed Recommendation,* August 10, 2001. http://www.w3.org/TR/2001/PR-xml-infoset-20010810

[Cham+01] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, M. Stefanescu, "XQuery 1.0: An XML Query Language", *W3C Working Draft*, June 07 2001. **http://www.w3.org/TR/2001/WD-xquery-20010607**.

[Chan+02] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. "Tree Pattern Aggregation for Scalable XML Data Dissemination". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, 2002.

[CP83] S. Ceri and G. Pelagatti. "Correctness of Query Execution Strategies in Distributed Databases". *ACM Trans. Database Syst.* (December 1983), 8(4):577-607.

[Deut99] Alin Deutch, Mary Fernandez, Dan Suciu, "Storing Semistructured Data with STORED," Proc. ACM-SIGMOD Conf., 1999, pp. 431-442.

[EN] Ramez Elmasri and Shamkant B. Navathe, "Foundamentals of Database Systems (3rd edition)". *Addison-Wesley, 2000.*

[Fern00] Mary F. Fernandez, Wang Chiew Tan, Dan Suciu, "SilkRoute: trading between relations and XML," *WWW9 / Computer Networks*, vol. 33, no 1-6, June 2000, pp. 723-745.

[Fern01] Mary F. Fernandez et al. "Publishing Relational Data in XML: the SilkRoute Approach," *IEEE Data Engineering Bulletin*, vol. 24, no. 2, June 2001, pp. 12-19.

[Fern+01] M. Fernandez, J. March, "Xquery 1.0 and Xpath 2.0 Data Model", *W3C Working Draft,* June 7, 2001. http://www.w3.org/TR/2001/WD-query-datamodel-20010607.

[Gold+99] R. Goldman, J. McHugh, J. Widow, " Form semistructured data to XML: Migrating the Lore model and query Language", *Proc. International Workshop on the Web and Databases*, 1999, pp.25-30

[Ipedo] Ipedo XML Database 3.0. http://www.ipedo.com/html/products_xml_dat.html

[Kaus+02] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, Ehud Gudes, "Exploiting Local Similarity for Indexing Paths in Graph-Structured Data", 18th International Conference on Data Engineering, 2002

[Koss00] Donald Kossmann, "The State of the Art in Distributed Query Processing", *ACM Computing Surveys,* vol.32, no.4, 2000, pp. 422-469.

[Kudo+00] M. Kudo, S. Hada, "XML Document Security based on Provisional Authorization", *Proc. 7th ACM Conference on Computer and Communications Security,* Nov. 2000, pp.87-96.

[LeHo+00] A. Le Hors, P. Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne, "Document Object Model (DOM) Level 2 Core Specification Version 1.0", *W3C Recommendation,* November 13, 2000. http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.

[Mcob] McObject LLC. **http://www.mcobject.com/extremedb.htm#xml.**

[NAVA95] Shamkant B. Navathe, " A Mixed Fragmentation Methodology For Initial Distributed Database Design", *Journal of Computer and Software Engineering,* Volume 3, Number 4, pages 395-426, 1995.

[OH+96] R. Orfali, D. Harkey, and J. Edwards, "Essential Client/Server Survival Guide, 2/e", *Wiley, 1996.*

[ÖV99] M.T. Özsu, P. Valduriez, "Principles of Distributed Database Systems, 2/e", *Prentice Hall, 1999.*

[Rice02] Frank C. Rice, "Exploring XML and Access 2002", *MSDN library*, July 2001. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnacc2k2/html/odc_acxmllnk.asp.

[Mala02] Susan Malaika, "Meet the experts: Susan Malaika on XML capabilities in DB2", *IBM Silicon Valley Lab,* December 2002. http://www7b.software.ibm.com/dmdd/library/techarticle/0212malaika/0212malaika.html.

[Raym+96] D.R. Raymond, F.W. Tompa, D. Wood, "From data representation to data model: meta-semantic issues in the evolution of SGML", *Computer Standard & Interfaces* 18, 1996, pp. 25-36

[Robi99] J. Robie, "XQL(XML Query Language)", August 1999. http://www.ibiblio.org/xql/xql-proposal.html.

[Salm01] Airi Salminen, Frank Wm. Tompa, "Requirements for XML Document Database Systems", *ACM Press, 2001,* pp. 85-94.

[Schm00] Albrecht Schmidt et al, "Efficient Relational Storage and Retrieval of XML Documents," *Procs. Int. Workshop on the Web and Databases (WebDB)*, 2000.

[Shan+00] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. J. Shekita and S. N. Subramanian. "XPERANTO: Publishing Object-Relational Data as XML". *Proc.of the Int. Workshop on Web and Databases (WebDB)*, pages 105–110, 2000.

[Shan00] Jayavel Shanmugasundaram et al, "Architecting a network query engine for producing partial results," In WebDB (Informal Proceedings) 2000, pages 17-22, 2000.

[Shan01] Jayavel Shanmugasundaram et al, "Querying XML Views of Relational Data," Proceedings of the27th VLDB, 2001, pp. 261-270.

[Shan01'] Jayavel Shanmugasundaram et al, "Efficiently Publishing Relational Data as XML Documents," The VLDB Journal vol. 10, no. 2-3, 2001, pp 133-154.

[Shan99] J. Shanmugasundaram et al., "Relational Databases for Querying XML Documents: Limitations and Opportunities," *Proc. of 25th Int'l Conf. Very Large Data Bases* (VLDB'99, Edinburgh, Scotland, UK), Morgan Kaufmann, 1999, pp. 302-314.

[Sonic] eXtensible Information Server 3.1, *Sonic Software Corp.* http://www.sonicsoftware.com/products/additional_software/extensible_information_server/index.ssp.

[Tamio] Tamino XML Server, http://www.softwareag.com/tamino/.

[Vian01] V.A. Vian, "Web odyssey: from Codd to XML" *Proc. 20th Symp. On Principles of Database Systems*, 2001, pp 1-15.

[XML1.0] Extensible Markup Language (XML) 1.0 (Second Edition), http://www.w3.org/TR/REC-xml

[XML:DB] XML:DB Initiative, "What is an XML database?", http://www.xmldb.org /faqs.html - faq-1

[XMLAPI] XML:DB Initiative, "Application Programming Interface for XML Databases", http://www.xmldb.org/xapi/index.html.

[XML-QL] A. Deutsch, M. Fernandez, D. Florescu, A.Y. Levy, D. Suciu, "XML-QL: a query language for XML", *Submission to W3C*, NOTE-xml-ql-19980819. http://www.w3.org/TR/NOTE-xml-ql.

[XQ01] S. Boag, D. Chamberlin, *et al.* "XQuery 1.0: An XML Query Language", *Technical Report, W3C*, 2001.

[XS] XML Schema Part 1: Structures. W3C note, http://www.w3.org/TR/xmlschema-1.

[W3C} World Wide Web Consortium (W3C), October 23, 2003. **http://www.w3.org**.

[W3CDOM] World Wide Web Consortium (W3C), *Document Object Model (DOM)*, http:// www.w3c.org/DOM.

[W82] G. Wiederhold, "Database Design (2nd edition)", *New York: McGrawHill,* 1982.