THE FLORIDA STATE UNIVERSITY

COLLEGE OF ART AND SCIENCE

XML AND DATABASES

By

JUNGKEE KIM

A Survey Paper submitted to the
Department of Computer Science
In partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Survey Paper Defended in Spring Semester, 2002

The members of the Committee approve the survey paper of Jungkee Kim defended on April, 2002.

Gregory Riccardi
Professor Directing Dissertation

Geoffrey C. Fox
Committee Member

Gordon Erlebacher
Committee Member

David Whalley
Committee Member

Larry Dennis
Outside Committee Member

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

XML is rapidly becoming a key technology for information representation and exchange on the Internet, where it increases the opportunity for the integration of the various data formats. The World Wide Web Consortium proposes several standards to define the structure of XML and extend the XML for addressing and linking. To adopt such benefits of XML, many academic institutions and business companies have made XML enabled databases on the current or new architectures. In this paper, I will present the XML standards and compare various technologies used for storing XML data in a repository. By highlighting their strengths and weaknesses, I seek ideas for combining those technologies with a collaborating system to leverage the integration of metadata.

CHAPTER 1

Introduction

Since the Extensible Markup Language (XML) was introduced as a standard document form, the usage of XML for the data exchange format between applications has dramatically increased. Before the appearance of XML, application dependent data formats were used for data exchange. However, the World Wide Web environment is so huge that application dependent data formats require a lot of coding effort for synchronizing data between different applications, due to the many different data format on the Web. Though Hyper Text Markup Language (HTML) protocol is used on the Web, it only describes how the data are shown through Web browsers with the fixed tag format. XML provides user-defined tags and it is simple to produce and flexible. With nested tags, XML can represent not only hierarchical tree structure, but also a graph structure using special attributes. Some applications may need an agreement on the XML document format, because such standardization would reduce the errors and increase the efficiency in exchanging data. Document Type Definition (DTD) and XML Schema are used for forcing the production of XML documents satisfying particular rules.

To extract key information through the Web, metadata can be used to designate the desired information. The Resource Description Framework (RDF) provides a description of metadata mainly relating to data on the Web. Through RDF, the semantic meaning of the Web resources can be revealed to machine-oriented systems, for example, search engines. The RDF Schema is defined to constrain the vocabulary of RDF documents. From the extensions of RDF and RDF Schema, a semantic markup language can be generated for the intelligent semantic descriptions of the Web documents. To support linking for XML documents, XLink, XPointer, and XPath were introduced by the World Wide Web Consortium.

A novel point of this paper is to survey technologies for storing XML documents in a repository. Many commercial database system companies have tried to integrate XML enabling functionalities on their existing architectures or devise new architectures for XML document storage. I focus on three typical relational database companies' support for XML repositories, and on other academic approaches with mapping XML documents to a relational database system or a semistructured repository. The commercial database systems provide some internal or external procedures and packages to shred XML documents into relational columns or objects. The data in the relational columns can generate XML documents through procedures and packages in the database system. Another way of storing the XML documents is using Large Object types in relational columns. Each Large Object type record keeps the whole XML document in the object column, without shredding. However, the functionalities on those types have not yet met the demand of XML users. Furthermore, those mappings are programmed manually, and some academic researches approach mapping on top of those XML enabling relational database systems. STORED [14] from AT&T Labs combined the relational mapping and semistructured techniques to generate mappings from XML documents to relational tables. The technology from University of Wisconsin-Madison [15] used the DTDs to produce the relational schemas with several different *Inlining* methods. Another XML storage technology is the semistructured data repository. Semistructured data research started before the emergence of XML, but the similarity of semistructured data and XML documents naturally allow the technology to apply to XML data. Lore from Stanford University [2] is a semistructured database system supporting an XML repository. DataGuide [18] is the core part for the description of the structure of the semistructured data.

CHAPTER 2

XML

The Extensible Markup Language (XML) is a standard document specification, a subset of Standard Generalized Markup Language (SGML) format [1]. Though SGML is powerful enough to have been used by the U.S. government and publishing companies for making documents, its implementation is considered to be difficult and complex. The Hypertext Markup Language (HTML) is another application of SGML, but HTML only presents the shape of the documents on the Web. The tags of HTML are fixed and the HTML has no mechanism for the data validation. Those features limit HTML to Web information manipulation and led to the advent of the XML.

XML consists of a *well-formed* structure, rooted in a prolog, one or more elements including balanced start- and end-tags with attributes, and miscellaneous optional features including comments, processing instructions, and white space. A typical example of the prolog as follow:

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

This shows that this is an XML document and the version number is “1.0.” The other *attributes*, “encoding” and “standalone,” are optional. The encoding denotes that this document used Unicode Transformation Format 8-bit (UTF-8) encoding. As XML is designed to support the International code, the encoding names can be any the parser supports, but the name is recommended to be registered with Internet Assigned Numbers Authority. The *standalone* document declaration is also an optional attribute and it indicates whether the XML document is affected by an external markup declaration like DTD and XML schema.

An *element* forms a root of a hierarchical tree structure for an XML document. Other elements can be added and should be nested within each other. If there is no content in an element, the empty-element tag, “<*tag-name*/>,” can be used instead of a pair of tags. Those tags are user-defined and that feature makes XML big different from HTML. Attributes are used to attach additional information of an element. They are located in start-tags or empty-element tags. Using attributes of type ID, IDREF or IDREFS [2], XML elements can be uniquely identified and form links. This linking mechanism allows XML to represent graph-structured information as well as tree-structured. In special CDATA sections, any markup data are interpreted as text data.

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

<!-- This is an example car. -->
<car id=”J544XD” state=”NY”>
  <company> Toyota </company>
  <model> Corolla </model>
  <type> DX </type>
  <year> 1996 </year>
  <color> white </color>
</car>
```

Figure 1. A car description in XML.

Figure 1 shows an example of the XML document. The first line stands for the prolog. The version attribute should be declared and currently the value should be “1.0.” The optional attributes *encoding* and *standalone* show that the code for this XML document is "UTF-8" without any schema ("standalone" is "yes"). The second line is empty - XML allows empty line for the good formatting. The third line is encoded as a comment, which does not present any meaning in the XML structure. The comment only provides optional information to readers. There is only one root element named "car" in the document and other elements are nested in the root element. The car element has two attributes. The *id* attribute identifies a specific car and the *state* attribute means the registered state of the car. The each element opens with start-tag "<*TAG-NAME*>" and closes with end-tag "</*TAG-NAME*>" and there is no overlapped tags. So, this XML example said to be well-formed.

CHAPTER 3

DTD and XML Schema

Document Type Definition (DTD) and XML Schema are ways to define the structure of XML documents. DTD has been used in SGML for over twenty years and XML is specified newly by the World Wide Web Consortium (W3C) [3]. The goal is to make rules to construct XML documents. For many purposes, user-defined tags alone don't provide a sufficiently rigorous structure for the XML information exchanges. By requiring the same DTD or XML Schema, two different applications can agree on a particular structure for an XML document. If a well-formed XML document satisfies a DTD or XML Schema, the document said to be *valid*.

The XML Schema specification reflects the demands of users, who have found DTD too limited. The schema has many improved features over DTD. Before defining XML Schema, there were several attempts to improve the functionality of the schema language for XML documents; some examples are Document Definition Markup Language (DDML), Document Content Description (DCD), Schema for Object-Oriented XML (SOX), and Microsoft's XML-Data for BizTalk. The W3C consortium activity for the new schema, XML Schema, considered those schemas in producing their design. The main differences between DTD and XML Schema will be presented in the XML Schema section.

DTD

DTD format is very different from XML. A DTD is usually included in the prolog part of an XML document using the "!DOCTYPE" tag. The DTD can be defined externally in a separate file, designated with a file name or a Uniform Resource Identifier (URI). The typical blocks of a DTD are *elements* and *attributes*. BNF syntax can be shown as follow:

```
<!ELEMENT> <element-name> <element-type>
<!ATTLIST> <attribute-name> <attribute-type> <attribute-option>
```

Figure 2 shows an example DTD for the car document of Figure 1. In the example, the "car" element is *non-terminal* and the other elements are *terminal*. The non-terminal element, "car," has five sub-elements: company, model, type, year, and color in that order. It is called a *sequence*, which restricts the order of sub-elements present. *Choice* is another group option for the sub-elements and it gives a list of alternatives for them. The vertical bar ("|") is used as the delimiter for choices, and the comma for sequences.

```
<!ELEMENT car(company, model, type?, year, color)>
<!ATTLIST car
          id CDATA #REQUIRED
          state CDATA #IMPLIED>
<!ELEMENT company (#PCDATA)>
<!ELEMENT model (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT color (#PCDATA)>
```

Figure 2. DTD for a car document

In the sequence of the example, all but *type* element will appear exactly once. The type element can be included optionally. This is indicated by the suffix, "?." Other allowed suffixes include "+," which means one or more elements can appear, and "*," which means zero or more can appear.

"#PCDATA" in terminal elements stands for parsed character data, which denotes text that has no markup. That is the only way to represent text in DTD, and this was one of motivations for the invention of XML Schema. The element content can also be *empty* or *any*. The *empty* element has no content but may have attributes. The *any* element has no restriction for that element.

The "car" element in the example has two attributes, which are declared in the DTD. The order of the attributes is not constrained. Both of the attributes have the same data type, character data (CDATA). Other attribute types like ID, IDREF, and IDREFS are also very useful and have key roles in representing a graph structures in XML format. The final term is an attribute description specifies whether this attribute is optional or required. The option for the attribute "id" is "#REQUIRED," and this attribute must be appeared in the every defined element. "#IMPLIED" in "state" means the attribute can be optional. Other option is "#FIXED" and this type attribute should have a default value. The fixed value cannot be changeable by the user.

XML Schema

While DTD is written in the syntax of Extended Backus Naur Form (EBNF), XML Schema uses XML document syntax. By supporting *namespaces*, XML Schema allows several sources of document definitions to be used in a single document. In DTD, a new DTD is needed to combine multiple DTDs. The XML Schema supports 44 datatypes including *string*, *decimal*, *time* and *date*, whereas DTD only provides 9 XML-related primitive types. Inheritance is another major feature of XML Schema, which is not present in DTD. This allows reusing existing structures by extending or restricting the base types.

Figure 3 includes an example of XML Schema, which reproduces the schema presented in Figure 2 in DTD format.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="car" type="CarType"/>

<xsd:complexType name="CarType">
  <xsd:sequence>
    <xsd:element name="company" type="xsd:string"/>
    <xsd:element name="model" type="xsd:string"/>
    <xsd:element name="type" type="xsd:string"
      minOccurs="0" maxOccurs="1"/>
    <xsd:element name="year" type="xsd:decimal"/>
    <xsd:element name="color" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string" use="required"/>
  <xsd:attribute name="state" type="xsd:string"/>
</xsd:complexType>

</xsd:schema>
```

Figure 3. XML Schema for a car document

The *car* schema has one *schema* element with sub-elements *element* and *complexType*. In the *schema* element, a namespace has been declared. The prefix “xsd:” associated with the namespace is used on each of the elements. The prefix name of a namespace can be an arbitrary value and different namespaces from different sources can be used. In the multiple namespaces, the different prefix names specify the meanings of elements and attributes, which are followed by the prefix. In this example, the association forces the elements and simple types to be identified with the XML Schema language. In XML

Schema, elements may have simple types or complex types. A simple type does not include elements. Many simple types, such as “xsd:string,” are defined in the XML Schema. A complex type may have nested elements and carry attributes optionally. The type of the *car* element is defined as a complex type, *CarType*, in the example. As in the DTD example, the elements of a complex type can be ordered with *sequence* tag. All the nested elements except the *year* element have a *string* type. This is declared in the *type* attribute. The *decimal* type in the *year* element is a number. If the year needs to be restricted to four-digit numbers for example, another simple type, *gYear*, already defined in XML Schema, and the declaration can be used instead as follows:

```
<xsd:element name="year" type="xsd:gYear"/>
```

If the *state* attribute has to be two capital letters, a new simple type can be defined as follows:

```
<xsd:simpleType name="StateType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

The equivalent of optional elements in DTD can be expressed in XML Schema using “minOccur” and “maxOccurs” attributes. Additionally to the three restrictions of DTD (*, +, and ?), XML Schema can designate any number of minimum and maximum occurrences – for example, between 15 and 30. Any given attribute, the attribute may appear at most once. The “use” attribute designates the attribute usage - one of *required*, *optional*, and *prohibited*.

The *fixed* attribute is used when the allowed value of an element or attribute is unique. For example, I may add another attribute *country* in the “car” element. The state name used in the attribute from United States and the fixed value the *country* could be “US.”

The definition of the attribute as follows:

```
<xsd:attribute name=”country” type=”xsd:string” fixed=”US”/>
```

CHAPTER 4

Metadata Support

There have been many efforts to extract useful information through the Web. Especially, the search engines devised various technologies to find the exact location of the desired information, but in practice many of them also show useless information. Moreover, they only reach less than one percent of the whole Web. That is because the scale of the Web is so huge, and keyword spamming is very widespread. Though some Web directory sites categorize the information manually, this is not for the machine-oriented system.

Metadata is "structured data about data." This could include catalogs of libraries, author lists of books, ranking of Web pages by frequency of reference, or the relations between indexes. Both human and machine generated information can be metadata.

RDF

The Resource Description Framework (RDF) is a W3C recommendation for a standard representation of metadata [5]. This framework is described in XML format. RDF has an innate function for machine-oriented data exchanging between applications because of its XML features. XML and RDF provide semantic interoperability in the current Web domain, but XML only describes the document structure. RDF emphasizes semantic meaning on the Web resources by adding a capability as a data model for knowledge representation.

The basic block of RDF consists of three object types - *resources*, *properties*, and *statements*. A *resource* is anything that can be written as a Uniform Resource Identifier (URI) in the RDF expression. It can be not only a Web page but also an XML element. Anything written in URI could be a resource. A *property* is a specific characteristic, attribute, or relation of the resource - for example, "owner." Each property has a specific meaning, which can be classified by a schema related to the name of the property. A *statement* is a combination of a *resource*, a *property*, and a *value*. Each part of a statement is also known as the *subject*, the *predicate*, and the *object*. The object can be another resource or a literal, which might be any string or XML. Figure 4 presents an example of RDF graph for the Web page of the University.

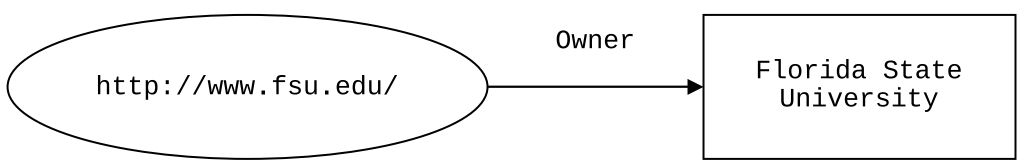

Figure 4. An example of RDF graph

In the figure, the oval shape node denotes a subject, the arc denotes a named property, and the rectangular shape is a node, which represents a literal. The graph represents the following statement:

"Florida State University is the owner of the resource http://www.fsu.edu/."

It also can be read as:

“http://www.fsu.edu/ has owner Florida State University.”

The statement can be written in the XML format:

```
<?xml version="1.0">
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-synfax-ns#"
  xmlns:s="http://description.org/schema/">
  <rdf:Description about="http://www.fsu.edu/">
    <s:owner>Florida State University</s:owner>
  </rdf:Description>
</rdf:RDF>
```

The RDF XML syntax has a root element, <*RDF*>, but this element is optional when the *description* is known to be RDF from the application content. In RDF element, the namespace attributes designate the location of the declarations of the RDF elements with the prefix “rdf:”, and the location of the schema declaration associating with the prefix “s:”. The namespace declaration can alternatively appear in a specific *description* element, or even in property elements. The *description* element has the subject; the child elements describe the properties and the objects. In this example, “<s:owner>” and “</s:owner>” - a pair of tags - show the property. The object is “Florida State University.”

As in XML Schema for XML document, RDF Schema provides a vocabulary constraint facility for RDF document. In RDF Schema, the classes of the resources are defined. The classes have the same role as in the object-oriented programming models. The classes have hierarchical structures and they are extended with subclass refinement. The terms such as “*Class*,” “*subPropertyOf*,” and “*subClassOf*” are used for the basic type system for RDF to define such classes. By using class concepts, the reusability of metadata can be increased because sharing schemas and adding subclasses to the existing schemas will produce sufficient mechanisms in many schema specifications.

Ontologies play a crucial role in the "Semantic Web" – a machine-understandable Web with intelligent services. They provide shared and precisely defined terms in a particular domain for communications between human users and application systems. DAML+OIL – combined terminology from old versions, DARPA Agent Markup Language (DAML) and Ontology Inference Layer (OIL) - is an ontology language submitted to W3C as a semantic markup language for Web resources [19]. It extends RDF and RDF schema. The object-oriented structure of domains in DAML+OIL consists of the terms "*Class*" and "*Property*." The usage of the term class is similar to that of RDF schema, but the classes of DAML+OIL are less restricted. For example, "*subClassOf*" class elements of DAML+OIL allow cyclic subclass-relations. A property is a binary relation, which defines the relation between two items. All expressions of description logic can be written in DAML+OIL terms. The DAML+OIL properties are of two types: "*objectProperty*" for object relations and "*datatypeProperty*" for datatype values. Those properties are ranged and multiple ranges can apply to a property *conjunctively*. For example, positive integer range and greater than 1 range will produce a property that has positive integers greater than 1. The restriction of domains in DAML+OIL is global, while the RDF Schema only has a single range and a local scope on domains. However, efficient tools for reasoning about Web resources and complete algorithms for the full DAML+OIL language are not provided yet [20].

From the viewpoint of semantic interoperability, RDF is better than plain XML. RDF vocabularies are simple enough to manipulate huge numbers of data. Meanwhile, XML often regards ordered elements as important and has complex structure. Those features make it difficult for XML to handle large amount of data. Additional data conversion is not necessary in RDF because RDF presents domain models naturally with defining objects and relations. Another benefit of RDF is independency of the XML. In an XML document, a schema change may cause invalidity for the query based on the old structure of the XML document. RDF presents a semantic tree that is parsed with only usable set of triples, and the data not to be interpreted are ignored [6].

XLink, XPointer, and XPath

As in hyperlinks of HTML documents, XML documents can be linked to other XML documents by using XML Linking Language (XLink) [7], XML Pointer Language (XPointer) [8], and XML Path Language (XPath) [9]. The XLink describes links between resources. The XPointer points the reference through URI. The XPath presents the location of specific parts of an XML document. Last version of the XPointer (September 2001) is a Candidate Recommendation of the World Wide Web Consortium. The XLink and the XPath is recommended by the W3C.

XPath

The main purpose of XPath is to designate parts of an XML document. XPath provides an extended addressing syntax that defines a compact notation for *node* location in the XML document tree. XPath does not use XML syntax but it is a string-based language. The Extensible Stylesheet Language Transformations (XSLT) and the XPointer use the functionality of the XPath.

XPointer

XPointer is used to identify specific fragments in XML documents via a URI. XPointer may select on the basis of XML ID attributes, or nodes in the hierarchical structure of an XML document related using XPath. An XPointer can also reference an arbitrary user designation on a specific *point* or *range* – doesn't have to be an XML node. The *range* can be specified with two *points*, as:

```
xpointer(id("start")/range-to(id("end")))
```

That XPointer locates the range between the start point for the element with ID "*start*" and the end point for the element with ID "*end*". The XPointer is complex enough to present most usages.

XLink

XLink is able to link not only documents but also resources, which include documents, audio, video, database data, and any addressable information or services. While HTML links need to edit the resource for additional links, XLink don't require any write permission to edit the source. The XLink can simply set the URI with the starting and ending point for the linking. The XLink also provides multidirectional links (*extended* links) as well as the unidirectional link (*simple* link) – the traditional link on the Web. The links can be stored externally (*extended* link) of the documents, they address with URI, and they can be inline. Traversal of "*A*" link usually replaces the document currently viewed. Traversal means "using or following a link for any purpose" [7]. The user may initiate traversal with clicking on the links, or the retrieving document may initiate it.

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

<doc xmlns:xlink="http://www.w3.org/1999/xlink">

<head>
<title> Animals </title>
<extendedlink xlink:type="extended">
  <loc xlink:type="locator"
       xlink:label="seaplace"
       xlink:href="#xpointer(//body/animal[1]/sentence[2]/place[1])"/>
  <loc xlink:type="locator"
       xlink:label="seareference"
       xlink:href="#sea"/>
  <arc xlink:type="arc"
       xlink:from="seaplace"
       xlink:to="seareference"
       xlink:show="new"
       xlink:actuate="onRequest"/>
</extendedlink>
</head>

<body>

<animal name="whale">
<sentence>Whales are mammals.</sentence>
<sentence>Whales live in the
  <place>sea</place>.</sentence>
</animal>

<animal name="horse">
<sentence>Horses are mammals.</sentence>
<sentence>Horses live in the
  <place>land</place>.</sentence>
</animal>

</body>

<tail>
<reference>
<places id="sea">
  Sea is the continuous body of salt water covering the earth.
</places>
<places id="land">
  Land is the part of the earth not covered by water.
</places>
</reference>
</tail>

</doc>
```

Figure 5. An example for the XML document with XLink expressions.

Figure 5 shows an example, which includes the features of XLink and XPointer. In the example, "**<doc xmlns:xlink = "http://www.w3.org/1999/xlink">**" denotes the XLink namespace definition with the URI. The *extendedlink* element is a kind of extended link, which has full XLink functionality such as arcs (*inbound* and *third-party*) and links with arbitrary resources. The other type for the link is the *simple* link, which has only two participating resources.

In the *extendedlink* element, three sub-elements are embedded: they are two *locator* elements (XLink type) and an *arc* element (XLink type). The *locator* type element designates remote resources, whose location is denoted with the locator attribute, "*href*". The arc type elements represent the link traversal, which is usually a pair of start (*from*) and end (*to*) resources. The *locator* labeled "*seaplace*" has the XPointer and the expression in the parentheses of this XPointer is also the XPath expression. This XPointer points to the first *place* element of the second *sentence* in the first *animal* element of the *body* element. The "*seareference*" *locator* links to the first "*places*" element, which has the attribute *id* named "*sea*."

The *arc* element in the example has two remote resources for the traverse and it is called a *third-party* arc. If the arc from local resource to remote resource, it is the *outbound* arc. Or, the *inbound* arc traverses from remote resource to local resource. The traversal attributes, "*from*" and "*to,*" are for the start and the end points of the link. The *show* and *actuate* attributes represent the behavior of the link. They designate the behavior of the ending resource of the arc. In the figure, the "new" value for the *show* attribute will open a new window when the traversal event has been requested. The *actuate* attribute sets "*onRequest*," and it constraints the traversal event. If the value of the *actuate* attribute changed to “*onLoad*,” the new window would be shown immediately on loading the starting resource.

CHAPTER 5

XML and Databases

An XML document is a text document used for information exchange between application programs, typically through the Web. XML does not force any internal structure on the computer. However, it is important how to store and process the XML document from the viewpoint of efficiency of data manipulation. XML logically has a tree structure with elements and attributes, and is used both in the role of a data transport format and as a document markup language. Bourret [10] classifies the XML documents as *data-centric* documents and *document-centric* documents. Data-centric documents are highly structured and have relatively small sized text elements. Document-centric documents have free-format text with some words marked-up. The performance of the XML document processing depends on the kind of XML document presentation: data-centric and document-centric documents have pros and cons relative to the different types of database.

Many attempts were made to leverage the current database technologies to represent XML documents. Mapping XML to relational database was one of first methods used to store XML documents in existing databases. Many XML repositories have proposed ways to map XML to relational databases, because the relational database dominates the current database market and many applications are already developed on relational databases. Not only legacy applications, but also data-centric XML documents obtain benefits from using the relational database. Object-relational storage is a natural fit for XML storage, because its logical structure is similar to that of XML documents. Another approach for the XML storage is from *semi-structured* data. Semi-structured data had been developed before XML standard emerged. The XML format unsurprisingly applied to the semi-structured database, because XML is a semi-structured data format. Lore [2] is a well-known example of semi-structured data system to present XML documents as semi-structure data.

Relational Database

Oracle Database System

Currently the relational database dominates the database market. Since XML has emerged as a new standard for the information exchange, many relational databases have been tried to combine their databases and XML technologies. One of initial answers from the commercial databases is the XML-enabled Oracle 8i from Oracle, which has the biggest market share in the current database business.

Mapping an XML document to a table or several tables is a primitive way of storing XML documents on the relational database. The elements, attributes, and names are mapped to the columns of tables in ways depending on the functionalities of data usage and mapping design. This method is useful in transferring data between relational databases, but it is not applicable to sophisticated XML formats.

Oracle [11] provides a natural manner using object-relational support. The elements with attributes define *object* types that encapsulate data. Sets of object types and references to object types can form a model of classes. A class maps into a table. In Figure 6, *CAR_TYPE* is defined as an object and a single object maps to a table in this example.

```
create table CARS
(
ID            VARCHAR2(7),
STATE         VARCHAR2(2),
COMPANY       VARCHAR2(16),
MODEL         VARCHAR2(16),
TYPE          VARCHAR2(3),
YEAR          NUMBER(4),
COLOR         VARCHAR2(16)
)
/

create or replace view NEWCARS as
select SYS.XMLTYPE.CREATEXML(‘<CAR/>’) “CAR”
from CARS;

create or replace trigger CAREXPLOSION
instead of insert on NEWCARS
for each row
declare
  CARID         VARCHAR2(7);
  STATE         VARCHAR(2);
  COMPANY       VARCHAR2(16);
  MODEL         VARCHAR2(16);
  CARTYPE       VARCHAR2(3);
  CARYEAR       NUMBER(4);
  COLOR         VARCHAR2(16);

  DOCUMENT      sys.XMLTYPE;
  ELEMENT       sys.XMLTYPE;

  NOT_A_CAR     exception;

  I             binary_integer;

begin

  DOCUMENT := :new.CAR;

  if (DOCUMENT.existsNode('/car') = 0) then
    raise NOT_A_CAR;
  end if;

  CARID := DOCUMENT.extract('/car/@id').getStringVal();
```

```
  STATE := DOCUMENT.extract('/car/@state').getStringVal();

  ELEMENT := DOCUMENT.extract('/car/company/text()');
  if (ELEMENT is not null) then
    COMPANY := ELEMENT.getStringVal();
  end if;

  ELEMENT := DOCUMENT.extract('/car/model/text()');
  if (ELEMENT is not null) then
    MODEL := ELEMENT.getStringVal();
  end if;

  ELEMENT := DOCUMENT.extract('/car/type/text()');
  if (ELEMENT is not null) then
    CARTYPE := ELEMENT.getStringVal();
  end if;

  ELEMENT := DOCUMENT.extract('/car/year/text()');
  if (ELEMENT is not null) then
    CARYEAR := ELEMENT.getNumberVal();
  end if;

  ELEMENT := DOCUMENT.extract('/car/color/text()');
  if (ELEMENT is not null) then
    COLOR := ELEMENT.getStringVal();
  end if;

  insert into CARS values (CARID, STATE, COMPANY, MODEL, CARTYPE,
CARYEAR, COLOR);

  exception

    when NOT_A_CAR then
      raise_application_error(-20000, 'Only car documents can be
stored in this column.');

end CAREXPLOSION;
/


create or replace type CAR_TYPE as object
(
“@id”             VARCHAR2(7),
“@state”          VARCHAR2(2),
“company”         VARCHAR2(16),
“model”           VARCHAR2(16),
“type”            VARCHAR2(3),
“year”            NUMBER(4),
“color”           VARCHAR2(16)
)
/

create or replace view CARDOCUMENTS as
select
sys_xmlgen(
CAR_TYPE(
```

```
  C.ID,
  C.STATE,
  C.COMPANY,
  C.MODEL,
  C.TYPE,
  C.YEAR,
  C.COLOR
),
sys.xmlgenformattype.createformat('car')
) car
from CARS C;


SQL> select C.car.getClobVal() "car" from CARDOCUMENTS C;

car
----------------------------------------------------------
<?xml version="1.0"?>
<car id="J544XD" state="NY">
 <company>Toyota </company>
 <model>Corolla </model>
 <type>DX </type>
 <year>1996</year>
 <color>white </color>
</car>
```

Figure 6. An example for the XML document with mapping to Oracle database.

Another approach for XML document storage is *Large Object* (LOB). A LOB column holds an XML instance and this type of storage is useful for document-centric documents. The XML instance in LOB columns can be indexed as in other texts. The New version of the Oracle database, *Oracle 9i* database system provides *XMLType* - an object data type on a *character large object* (CLOB) column storage. XMLType supports XPath in SQL queries to extract elements and attributes of XML instances as follow:

```
SELECT c.car.extract(‘/car/model/text()’).getStringVal() FROM cars c;
```

To improve performance, a key element of the XML document can be stored in another column and be indexed. Without separate column indexing, extracting query performance is not acceptable in the case of large number of rows. Updating elements or attributes in LOB is not possible. Only the entire XML documents update is allowed. There are additional packages and functions integrated in SQL which can be used in queries to wrap the data in columns and produce XML documents.

Figure 6 shows an example that maps between an XML document and a relational table using Oracle 9i. The first half of the code represents the mapping from an XML document to a relational table. The last half shows how to get an XML instance from a relational table. The *CARS* table stores the elements and the attributes of the XML example from chapter one. The *NEWCARS* view shows XML instances from a relational table, *CARS*. When an XML document inserts into a relational table, each element and attribute needs to be shredded. The *CAREXPLOSION* trigger substitutes the insert query to map the XML document into proper types of columns. From the relational table, an XML instance can be produced. The *CARDOCUMENT* view generates an XML instance from a table using SYS_XMLGEN function in the SQL query.

IBM DB2 Database System

IBM's *DB2* database system [12] also supports XML documents for storage and query. To store XML documents in *IBM DB2 XML Extender*, an XML repository, *XML column* and *XML collection* options are available.

In the XML column option, the XML document is saved as *XMLCLOB*, *XMLVARCHAR* or *XMLFILE* type without shredding. Some elements and attributes can be stored in *side tables* and indexed for performance improvement. The side tables and index must be in a *Data Access Definition* (DAD). An example DAD for side tables is given in Figure 7.

In the XML collection option, an XML document maps into a set of relational tables and the mapping mechanisms between DTD and tables are described in a DAD. This option is useful for frequent updates to part of data, extraction of only some part of data, and relating to other relational data.

```
…
<dad>
  <dtdid>dxx_install/dtd/getstart.dtd</dtdid>
  <validation>YES</validation>
  <Xcolumn>
    <table name=”car_table”> </table>
      <column name=”year” type=”decimal(4,0)”
       path=”/car/year”/>
    </table>
  </Xcolumn>
</dad>
```

Figure 7. An example of DAD side table definition

To publish XML documents from database tables, SQL queries with macros of a script language and stored procedures are used.

Microsoft SQL Server

Microsoft SQL Server [13] is another relational database system that generates and stores XML documents through relational data. XML enabling features are executed in middle tier applications, for example, *templates* and *XML views*. Templates are XML documents which include SQL queries executed against the database. XPath is supported in templates. An XML view of the relational data can be created by annotated schema using *XML Data Reduced* (XDR) schema. These annotations are used to specify a mapping between XML and relational tables.

To publish an XML document from a relational database, SQL Server includes SQL extensions to produce query results as XML documents. There are three different ways to serialize SQL query results into XML: *RAW*, *AUTO*, and *EXPLICIT* modes. In RAW mode, each row of the query result maps into a name of an XML element, *row*, and each non-NULL column of the query result maps to an XML attribute. The example query may produce as follow:

```
SELECT CarID, CarState FROM Cars FOR XML raw
```

```
<row CarID=”J544XD” CarState=”NY”/>
```

To produce query results with nested XML elements, SQL server provides AUTO mode. Each row maps to an XML element and each table alias is used for the element name. The order of the table names in the SELECT clause determines the nesting according to their appearance from left to right. The columns of the query results map to the XML attributes as in RAW mode. The example of an auto mode query may return the result as follow:

```
SELECT Dealers.DealerID, Dealers.DealerName, Customers.CustomerName
FROM Dealers, Customers
WHERE Dealers.DealerID = Customer.DealerID
FOR XML auto

<Dealers DealerID=”ROMANOTOYOTA” DealerName=”Peter Dan”>
  <Customers CustomerName=”Jungkee Kim”/>
  <Customers CustomerName=”Bryan Carpenter”/>
  …
</Dealers>
```

The EXPLICIT mode can produce any XML document from the relational database tables. The EXPLICIT mode query, which specifies the structure of the XML tree, produces a *universal table* which consists of *tag* and *parent*, column names, and row ordering. For, example the query may produce the universal table and an XML instance as follows:

```
SELECT 1 as Tag, NULL as Parent, Dealers.DealerID AS
[Dealers!1!DelaerID],
  NULL as [Customers!2!name!element]
FROM Dealers
UNION ALL
```

```
SELECT 2, 1, Dealers.DealerID, Customers.CustomerName
FROM Dealers INNER JOIN Customers ON Dealers.DealerID =
Customers.DealerID
ORDER BY [Dealers!1!DealerID]
For XML explicit

Tag Parent Dealers!1!DealerID Customers!2!name!element
1   0      ROMANOTOYOTA       NULL
2   1      ROMANOTOYOTA       Jungkee Kim
1   0      SYRACUSEFORD       NULL
2   1      SYRACUSEFORD       Bryan Carpenter

<Dealers DealerID=”ROMANOTOYOTA”>
  <Customers><name>Jungkee Kim</name></Customers>
</Dealers>
<Dealers DealerID=”SYRACUSEFORD”>
  <Customers><name>Bryan Carpenter</name></Customers>
</Dealers>
```

To produce a convenient relational view from an XML document, *OpenXML rowset provider* is provided, similar to the *extract* function of Oracle 9i.

STORED

STORED (Semistructured TO Relational Data) [14] is one of several initial proposals for storing and querying XML documents mapped to relational database system. STORED utilizes a combination of relational and semistructured techniques to manage XML documents. It tries to discover the most frequently occurring sub-trees from XML documents and maps the extracted sub-trees to relational tables. The remaining part of XML documents is stored in a semistructured *overflow* graph. For example, the XML document and the graph in Figure 8 will produce a relational table *car* with attributes, plate and driver. The second driver, Mary, might be stored in the overflow repository.

A declarative query language is defined and this language expresses the structure in both input and output of a query. Because the language is non-recursive, the elements cannot have arbitrary number of sub-elements in the same format. If a DTD is available, the overflow queries are expected to be executed faster than they would be without a DTD.

```
<cars>
  <car>
    <plate>J544XD</plate>
    <driver>Jake</driver>
  </car>
  <car>
    <plate>S7RDG</plate>
    <driver>Tom</driver>
    <driver>Mary</driver>
  </car>
  <car>
    <plate>53G123</plate>
    <driver>Bryan</driver>
  </car>
</cars>
```

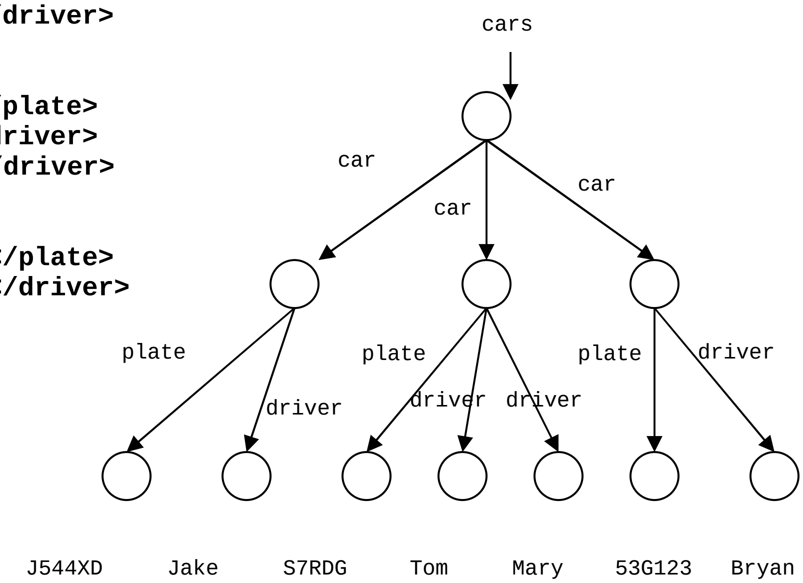

Figure 8. An Example of an XML document and a graph

When STORED generates a mapping, it requires various parameters: the maximum number of tables, the maximum number of attributes per table, the maximum disk space, the *collection size threshold*, and *minimum support*. The collection size threshold designates the boundary between *small sets* and *collections*. The minimum support parameter is required for the mining algorithm. The mining algorithm determines which elements of the XML document should be stored in relational tables and which elements should be saved in overflow. The mining processes measure which path prefixes and bodies are frequently occurring in XML documents and queries. From those numbers, the best relational mapping will be selected taking into account the number of matching values and the maximum disk space.

Since the queries and update are performed against the original XML documents rather than mapped tables or overflow, STORED should *rewrite* them into queries and updates over relational tables and overflow storage. The rewrite algorithm of the STORED system, *inversion rules*, converts the queries over XML documents to take table columns and add tree structure to the data layout.

Relational Mapping Technology from University of Wisconsin-Madison

Shanmugasundaram and others [15] suggested a different approach for mapping XML documents to relational database systems. They use a DTD to generate a relational schema unlike STORED, where using a DTD is optional. There is no consideration of the query workload. XML documents are parsed, matched to DTDs, and loaded to relational database tables. They used an IBM DB2 database for the relational tables. For querying the data, semistructured queries are translated to SQL queries and the results are converted to XML.

When generating relational schemas from DTDs, DTDs are simplified with a set of transformations that is more restricted than that of STORED. Transformations are performed for flattening, simplification, and grouping as in the following examples:

Flattening: $(e_1, e_2)^* \rightarrow e_1^*, e_2^*$
Simplification: $e_1^{**} \rightarrow e_1^*$
Grouping: …, a*, …, a*, … -> a*, …

Because of the simplification, the order of elements of the original XML document may be lost though additional fields for some elements were suggested for keeping the order. A DTD can be expressed as a graph. The simplified DTD graphs can be converted to relational schemas using one of three proposed methods: the *Basic Inlining Technique*, the *Shared Inlining Technique*, or the *Hybrids Inlining Technique*.

The Basic Inlining Technique creates separated relations for each elements of an XML document, because all elements of a DTD can be a root of an XML document. Each relation has an ID that acts as the key field. All sub-elements and attributes of the element in the relation are inlined, but there are two exceptions. Multiply occurring sub-elements – the nodes below "*" – and recursive referenced elements are not inlined. They have separate relations. The recursive element will also have the parent reference. The XML fragment in Figure 8 may produce the relations as following:

```
cars(carsID: integer)
cars.car(cars.car.carID: integer, cars.car.parentID: integer,
cars.car.plate: string)
cars.car.plate(cars.car.plate.plateID: integer,
cars.car.plate.parentID: integer, cars.car.plate: string)
cars.car.driver(cars.car.driver.driverID: integer,
cars.car.driver.parentID:integer, cars.car.driver: string)
```

The Shared Inlining Technique expresses each element as a relation and avoids the redundancy of the Basic Inlining Technique. The multiple relations for elements with several parents in Basic Inlining are stored in each relation in Shared Inlining. A new relation for those elements is created and shared. All the nodes with in-degree of one will be inlined into columns of the tables for their parent elements. The Shared Inlining for Figure 8 may produce the following relations:

```
cars(carsID: integer, cars.isRoot: boolean)
car(cars.car.carID: integer, cars.car.parentID: integer,
cars.car.parentCODE: integer, cars.car.plate: string, cars.car.plate:
string)
driver(driverID: integer, parentID:integer, cars.car.driver: string)
```

However, the Shared Inlining can require more join operations on some particular elements comparing to the Basic Inlining.

The Hybrid Inlining Technique is similar to the Shared Inlining Technique, but it has additional inlining which is not included in the Shared Inlining. The extra inlining elements in this technique are not for recursive or cyclic.

The performance for the Basic Inlining Technique is very poor. There are trade-offs for the Shared Inlining Technique and the Hybrid Inlining Technique.

Semistructured Database

Research on semistructured data initially looks for an efficient way to describe data with no fixed schema, for example, the information on the World-Wide Web. The relational or object-oriented database systems have schema, and all the instances in the systems should be created under the rules of the schema. Semistructured data have no explicit explanation of the structure, but they include direct descriptions of the data with a simple syntax. The Object Exchange Model (OEM) is a typical semistructured data model. It was designed for exchanging data between heterogeneous systems and originates from the Tsimmis [16] data integration project. An OEM object has four components: *label*, *oid*, *type*, and *value*. Label is a character string and oid is the identifier of the object. Type can be an atomic type or a complex type. If the object type is complex, the value is a set of oids. The value of an atomic type is one of base types – *integer*, *string*, *image*, *sound*, etc. Therefore OEM data form a graph in which the nodes are the objects and the labels are generally attached to edges, though the initial system was labeled on nodes.

Abiteboul and others [17] announced three approaches to develop a database management system for semistructured data: *building an application on top of existing relational or object database systems*, *using a low-level object server*, and *building the system from scratch*. The Lore (Lightweight Object Repository) system from Stanford

University [2] took the last approach and is a typical database system for semistructured data. The Lore system consists of the API, query processors, and data managers. A graphical user interfaces provides queries and views of data by the users. *Lorel* is the query language of Lore. The queries presented with Lorel are parsed, preprocessed, transformed, and optimized in query processors. There are several managers and tools for data handling. The OEM objects can be saved to or fetched from files though an object manager. The XML documents are mapped and kept in OEM forms internally. To describe the structures of the stored semistructured data, DataGuides [18] were introduced in Lore.

A DataGuide is a *concise* and *accurate* summary of the structure of a specified data graph. It describes every unique path of the original data graph only once. Every label path appearing in the DataGuide exists in the original graph to reserve the accuracy. The logical structure of a DataGuide is a directed acyclic graph. A *target set* is the set of all objects that a given label path in a graph reaches. A node in the input graph may appear more than once in the target set, because the node can be reached from several different edges. The target sets containing IDs are the *annotations* of the label paths that designate the nodes of the DataGuide. Goldman and Widom [18] assert that generating DataGuides over the input graphs is similar to the conversion from a non-deterministic finite automation (NFA) to a deterministic finite automation (DFA). They expect that the conversion would be finished within a reasonable time and they did not see any exponential time or space problem during their experiments. However, those annotations mean that multiple label paths may exist to reach the same object. In Figure 9, a data graph and two DataGuides of the graph are shown.

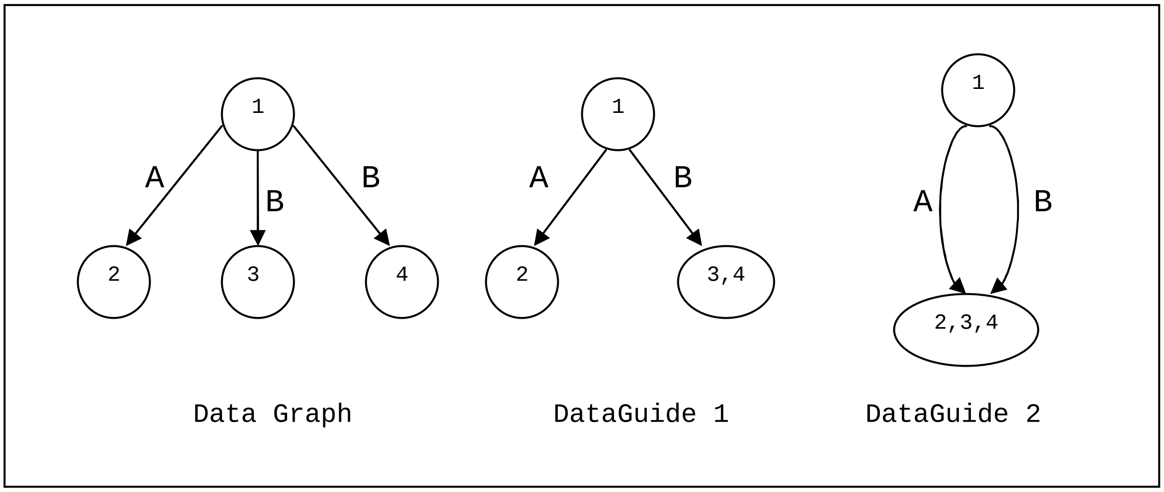

Figure 9. A data graph and two DataGuides

We can reach the node 2 with path *A* from the data graph and DataGuide 1, but the same node attains through both *A* and *B* paths in DataGuide 2. To avoid the confusion as in DataGuide 2, a class of DataGuides is defined and named *Strong DataGuide*. The main aspect of strong DataGuides is that *each set of label paths that share the same (singleton) target set in the DataGuides is the set of label paths that shares the same target set* in the data graph.

REFERENCES

[1] Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/2000/REC-xml-20001006, World Wide Consortium (W3C) Working Draft.

[2] Goldman, R., McHugh, J., & Widom, J. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language, In *Proceedings of the Workshop on the Web and Databases,* June 1999.

[3] XML Schema. http://www.w3.org/XML/Schema, World Wide Consortium (W3C).

[4] Radiya, A. & Dixit, V. The basics of using XML Schema to define elements: Get started using XML Schema instead of DTDs for defining the structure of XML documents. http://www-106.ibm.com/developerworks/library/xml-schema/, August 2000.

[5] Lassila, O., Swick, R.R., Resource Description Framework (RDF) Model and Syntax Specification, http://www.w3.org/TR/REC-rdf-syntax/, February 1999.

[6] Berners-Lee, T., Why RDF model is different from the XML model, http://www.w3.org/DesignIssues/RDF-XML.html, September 1998.

[7] XML Linking Language (XLink), Version 1.0, W3C recommendation, http://www.w3.org/TR/xlink/, 20 December 2000.

[8] XML Pointer Language (XPointer), W3C Working Draft, http://www.w3.org/TR/xptr, 8 January 2001.

[9] XML Path Language (XPath), Version 1.0, W3C recommendation, http://www.w3.org/TR/xpath, 16 November 1999.

[10] Bourret, R., XML and Databases, http://www.rpbourret.com/xml/, February 2002.

[11] Banerjee, S., Krishnamurthy, V., Krishnaprasad, M., and Murthy, R., Oracle8i – The XML Enabled Data Management System, In *International Conference on Data Engineering*, February 2000.

[12] Cheng, J. & Xu, J., XML and DB2, In *International Conference on Data Engineering*, February 2000.

[13] Rys, M., State-of-the-Art XML Support in RDBMS: Microsoft SQL Server's XML Features, Bulletin of the Technical Committee on Data Engineering, June 2001.

[14] Deutsch, A., Fernandez, M., & Suciu, D., Storing Semistructured Data with STORED, In SIGMOD Conference, June 1999.

[15] Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., & Naughton, J., Relational Databases for Querying XML Documents: Limitations and Opportunities, In *International Conference on Very Large Data Bases*, September 1999.

[16] Papakonstantinou, Y., Garcia-Molina, H., & Widom, J., Object Exchange Across Heterogeneous Information Sources, In *International Conference on Very Large Data Bases*, September 1997.

[17] Abiteboul, S., Buneman, P., & Suciu, D., Data on the Web: From Relations to Semistructured Data and XML, Morgan Kaufmann, 2000.

[18] Goldman, R. & Widom, J., DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, In *International Conference on Very Large Data Bases*, September 1997.

[19] DAML+OIL Web Ontology Language, Submission request to W3C, http://www.w3.org/Submission/2001/12/, December 2001.

[20] Horrocks, I., DAML+OIL: a Reason-able Web Ontology Language, In *Proceedings of EDBT*, March 2002.