Implementing XML Schema inside a ‘Relational’ Database

Sandeepan Banerjee
Oracle Server Technologies
500 Oracle Pkwy
Redwood Shores, CA 94065, USA
+ 1 650 506 7000
Sandeepan.Banerjee@Oracle.com

ABSTRACT

XML Schema has emerged as a promising data model that unites structured and unstructured content. The Oracle database has led the commercial database community in integrating support for XML Schema inside an enterprise data server. The foundation for this was laid with the absorption of the SQL:1999 'object-relational' type system in the database, which provided the necessary hierarchical abstractions necessary for representing XML. We look at how XML Schemas have been implemented in Oracle XML DB, what optimizations are available to cover the diverse use-cases for schema-based XML storage and retrieval, and how this technology contributes to richer data management.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages – *Data description languages (DDL), Data manipulation languages(DML), Database (persistent) programming languages, Query languages*

General Terms

Standardization, Languages, Management.

Keywords

XML Schema, SQL:1999, Relational, Object-Relational, DOM Fidelity, Query-Optimization, XPath, Indexing.

1. INTRODUCTION

Early adopters of XML exploited the standard’s core characteristics of self-description and ad-hoc extensibility for the flexible transportation of messages between applications. The second generation of XML standards such as XML Schema expanded the scope of XML technologies beyond data- or instruction-interchange. XML Schema is the first data model that can be used to represent both unstructured ‘documents’ and structured ‘data’.

Today, applications store data in a relational database and documents or web content in a file system. XML is used mostly as an artifact for transport, generated from a database or a file-system. As the volume of XML being transported grows, and developers consider the costs of constant regeneration of XML documents, there arises the question whether these storage methods can effectively accommodate XML content. From these considerations, it becomes clear that XML Schema is an important model for databases to absorb, so that the core capabilities of strong relational management can be extended to all kinds of data, and also so that both storage and generation of XML can be done with the efficiencies that accrue from understanding the structure of XML.

2. XML SCHEMA

The W3C Schema Working Group has published a specification of XML Schema to provide a means for defining the structure, content and semantics of XML documents. The XML Schema language is an improvement over DTDs in that it provides strong typing of the elements and attributes, uses XML syntax for its specification, can address content-models (mixed content, exact number of occurrences of elements, named group of elements), is extensible and self-documenting. Its type system is rich, defining 47 scalar data types, and this base set of data types can be extended using techniques like inheritance and extension to define more complex types. Sequences and collections are supported. URN-based Namespaces can be used to disambiguate names. XML Schemas can be designed to be variable -- supporting optional attributes, optional and repeated elements, and choices from alternatives of multiple elements.

The XML Schema type system is rich enough to address ‘structured’ relational data (i.e. where the structure of each item is regular, collections are homogeneous, and the terminal data-items consist of scalar values) as well as ‘unstructured’ documents (where the structure is flexible, and the document interleaves some data with regular structure and large portions of un-typed annotations or text which has irregular structure.) In addition, XML Schema can be used to specify semi-structured documents (in whom structure exists, but this structure is variable between instances.)

Storing and retrieving XML Schema-based documents in a ‘relational’ database presents a number of novel challenges.

- Preservation of the XML Document Object Model: The tuples in a relational system have to inherent ordering. However, the relative ordering of elements in an XML documents (say paragraphs in a chapter) can be an important part of the semantics of the document. In addition, constructs like namepsaces cannot be easily mapped to relational tables. These differences between what XML Schemas can represent and what basic relational models allow, can result in loss of fidelity as part of storing XML Schema-based documents in relational storage.
- Efficient access of mixed or variable content: Relational solutions have addressed indexing of structured data. However

efficient access of unstructured or variable content is an important issue.

- Application of constraints and semantic rules: XML Schemas can specify not only the structure but also semantics and business rules of certain kinds. It is important to be able to constrain Schema-based documents to all the semantics in the Schema, and not just the structural ones.
- Evolution of Schemas: Schemas allow for variability and extensibility, and can also change over time in an operational context. A system that supports XML Schemas should be able to handle schema evolution
- Global and Local elements: Element of an XML schema can be local or global. Global elements are children of the root schema element. Local elements are nested inside schema structure and not direct child of schema element.
- Efficient storage and materialization: XML document instances are relatively large for the amount of information they contain (due to the extra overhead of markup and conversion of all information into characters), and the Document Object Model relatively inefficient in terms of memory consumption. For such documents to be scalably stored and processed it is important to use the information latent in XML schemas for efficient storage and retrieval.

While a full discussion of the resolution of all the above issues relating to the 'impedance mismatch' between XML Schema and the relational model would exceed the scope of this paper, we will look at the significant aspects of absorbing XML Schemas into an extended-relational model.

For a number of years, the relational model of the SQL standard has been hybridizing to include complex structures and variability. This is often called object-relational technology, is captured in the SQL:1999 standard [1], and has served to converge the standard relational and XML data models. We look at the basic constructs of Oracle's SQL:1999 style object-relational implementation.

3. OBJECT-RELATIONAL TECHNOLOGY

Historically, applications have focused on accessing and modifying corporate data that is stored in tables composed of native SQL data types such as INTEGER, NUMBER, DATE, and CHAR. In Oracle, there is support not only for these native types, but also for new user-defined or system-generated 'object' data types that support composition, aggregation, encapsulation, inheritance, identity-based reference semantics and so on.

Oracle allows users to treat object data relationally and relational data as objects. For example, users can use SQL to query on object data in the same way that they access relational data. Users can access an object (using SQL DML for the query), the object types attributes and methods, with extended path expressions. They can also use SQL to perform explicit joins between objects in tables. In addition, Oracle lets users perform implicit joins between objects, by traversing or navigating references from one object to the other. Object types are indexable. Object types can be instantiated in identity-preserving 'object' tables, or used as datatypes of columns in relational tables. In addition, Object Views allow the synthesis of 'virtual' objects from data that continues to be stored in relational tables.

Oracle supports the single type inheritance model with substitutability of objects and references. View hierarchies can also be constructed. Oracle also supports collection types. Collections are SQL data types that contain multiple elements. Each element or value for a collection has the same substitutable data type. In Oracle, there are two collection types – Varrays and Nested Tables.

A Varray contains a variable number of ordered elements. Varray data types can be used as a column of a table or as an attribute of an object type.

Using Oracle SQL, a (named) table type can also be created. These can be used as Nested Tables to provide the semantics of an unordered collection. As with Varray, a Nested Table type can be used as a column of a table or as an attribute of an object type. Oracle supports multiple levels of nesting within collections, e.g. Nested Tables or Varrays embedded within a Nested Table or Varray.

Oracle provides the large object (LOB) types to handle the storage demands documents or multimedia. Large objects are stored in a manner that optimizes space utilization and provides efficient access. More specifically, large objects are composed of locators and the related binary or character data. The LOB locators are stored in-line with other table record columns. In case of internal LOBs (BLOB, CLOB, and NCLOB) the data can reside in a separate storage area. However, for external LOBs (BFILEs), the data is stored outside the database in operating system files. Full text keyword indexes (which can exploit any XML markup that exists) can be builds on CLOBs or BFILEs.

Object types can be evolved, including adding an attribute to a type, dropping an attribute from a type. Modifying the type of an attribute by increasing its length, precision, or scale, as well as adding or dropping a method to a type.

What is immediately apparent is the number of parallels that exist between XML Schema and the SQL:1999 object model. Simple and complex XML types can be captured as object types; the notion of inheritance and substitution groups can be mapped; ordered collections can be specified; mixed content achieved using LOBs – and so on. The object-relational infrastructure is used to harmoniously absorb XML Schemas in Oracle's XML DB implementation.

4. XML SCHEMAS IN ORACLE XML DB

An XML Schema document, with certain additional attributes defined by our XML DB implementation, is used to describe the storage mappings, in-memory representations and language bindings of XML documents that conform to the schema. The process of compiling the XML Schema (referred to as schema registration) creates the appropriate storage structures (in terms of SQL:1999 object-relational types and tables). The example below shows the default storage structures created for the XML schema po.xsd, which describes the canonical purchase-order with associated line items.

In the example shown in Table 1, the object type Item_T is created corresponding to the local Item complexType. An additional collection (an Oracle Varray) type Item_COLL is created because there can be more than one occurrence of Item (maxOccurs > 1). The object type PurchaseOrderType_T corresponds to the local PurchaseOrder complexType. The simple types referenced in the XML Schema are mapped to appropriate SQL datatypes. A registered schema can then be referred to within a CREATE TABLE statement. This results in the underlying SQL types being used to create the columns of the table. In addition, a second table is created to hold the collection of Items. A foreign key is used to associate the Item rows with the corresponding parent "PurchaseOrder" row.

When a XML document is inserted into the XMLType table, it is appropriately shredded and values inserted into the underlying columns. In case of collections stored in separate tables, one or more rows get inserted into these nested tables. Table 2 below shows an instance document and the values in the top level and nested tables.

Note the presence of array_index column in the nested table. This system column of NUMBER datatype tracks the ordering of elements within a collection. When new elements are inserted into the middle of existing collections, the array_index range is subdivided to compute new values. For example, an entry to be inserted between [1, …] and [2, …] is assigned array_index = 1.5 viz. (1+2)/2. This enables entries to be inserted and deleted within collections without affecting other entries. Multiple levels of nesting are handled in XML DB by either creating embedded object types or embedded collection types. If the maxOccurs of a nested complexType is 1, the corresponding object type is embedded within the parent object type. If maxOccurs > 1, a collection type is created and embedded within the parent object type. Further, these multiple levels of collections are stored in multiple tables with foreign keys associating rows with their parent row. Each nested table has an array_index column to track the ordering of elements within the specific collection.

Table 1: An XML Schema and Corresponding SQL Types

```
<schema
targetNamespace=http://www.oracle.com/PO.xsd
xmlns:po="http://www.oracle.com/PO.xsd"
elementFormDefault="qualified"
xmlns="http://www.w3.org/2001/XMLSchema">
 <complexType name="PurchaseOrderType">
  <sequence>
   <element name="PONum" type="decimal"/>
   <element name="Company">
    <simpleType>
     <restriction base="string">
      <maxLength value="100"/>
     </restriction>
    </simpleType>
   </element>
   <element name="Item" maxOccurs="1000">
    <complexType>
     <sequence>
      <element name="Part">
       <simpleType>
        <restriction base="string">
         <maxLength value="1000"/>
        </restriction>
       </simpleType>
      </element>
      <element name="Price" type="float"/>
     </sequence>
    </complexType>
   </element>
  </sequence>
 </complexType>

 <element name="PurchaseOrder"
type="po:PurchaseOrderType"/>
</schema>
```

SQL Object Types

```
TYPE "Item_T"
(
  part varchar2(1000),
  price number
);
```

```
TYPE "Item_COLL" AS
varray(1000) OF "Item_T";
```

```
TYPE "PurchaseOrderType_T"
(
  ponum number,
  company varchar2(100),
  item Item_varray_COLL
);
```

XMLType Table & Nested table

```
TABLE po_tab OF XMLTYPE
XMLSCHEMA " PO.xsd"
ELEMENT "PurchaseOrder"
VARRAY(xmldata.item)   STORE   AS
item_tab;
```

Table 2: Handling Collections in XML Schemas

```
<PurchaseOrder
 xmlns="http://www.oracle.com/PO.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://www.oracle.com/PO.xsd
http://www.oracle.com/PO.xsd">
      <PONum>1001</PONum>
      <Company>Oracle Corp</Company>
      <Item>
        <Part>9i Doc Set</Part>
        <Price>2550</Price>
      </Item>
      <Item>
        <Part>8i Doc Set</Part>
        <Price>350</Price>
      </Item>
</PurchaseOrder>
```

PO_TAB

Row ID	ponum	Company
1	1001	Oracle Corp

ITEM_TAB

Parent ROW ID	Array Index	part	price
1	1	9i Doc Set	2250
1	2	8i Doc Set	350

5. DOM FIDELITY

In general, any technique that involves shredding XML documents to relational storage loses the fidelity of the document in terms of one or more of the following aspects:

- whitespaces between elements and between attributes
- ordering of elements
- comments within the XML document
- processing instructions
- namespaces declarations
- element and attribute prefixes

Ordering of elements is highly relevant in many applications. However, the XML Schema may not constrain the order of elements (for example, using <choice> or <all> model groups). Since many of these elements may be flattened into a single row of a table, the relative ordering of these elements is not tracked.

Oracle XML DB supports fidelity of documents with respect to their DOM (Document Object Model) i.e. an application that uses the DOM API to traverse the XML document will find that the input document is identical to the output DOM. This corresponds to all the aspects listed above except (1).

To ensure DOM fidelity, XML DB adds a system binary attribute to each created object type. This attribute is referred to as the positional descriptor – which stores (in a binary encoded format) all pieces of information that cannot be stored in any of the other structured attributes. The encoded information includes:

- Ordering of elements
- Comments
- Processing Instructions
- Namespace declarations
- Prefix information

This information is carried in a hidden attribute and maintained for all DDL and DML operations. As a result, XML DB provides DOM fidelity, i.e. the XML DOM that is stored is the DOM that is retrieved with no loss of information.

6. HYBRID STORAGE MAPPINGS

Oracle XML DB supports a complete spectrum of storage mappings. At one end of the spectrum is "full shredding" – as shown in the first example. Every attribute and simple element value is stored in a separate column of some table. All collections are stored in a separate table from the parent table using a foreign key association. At the other end of the spectrum, XML DB also supports "packed storage" i.e. the entire XML document is stored in a single LOB.

A novel aspect of Oracle XML DB is that it also supports any intermediate mapping (semi-structured) of the XML Schema – by defining certain portions of the XML document to be "shredded" while storing other fragments in LOBs. This is referred to as the hybrid storage mapping. The hybrid storage is accomplished by specifying the XDB attribute `SQLType` within the corresponding <complexType> declaration. In the following example, the XML schema specifies that the `Addr` fragment is stored as a CLOB while the other elements and attributes are shredded.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.oracle.com/emp.xsd"
 xmlns:emp="http://www.oracle.com/emp.xsd"
 xmlns:xdb="http://xmlns.oracle.com/xdb">

<complexType name = "Employee">
 <sequence>
   <element name = "Name" type = "string"/>
   <element name = "Age" type = "decimal"/>
   <element name = "Addr" xdb:SQLType = "CLOB">
    <complexType >
     <sequence>
       <element name = "Street" type = "string"/>
       <element name = "City" type = "string"/>
     </sequence>
    </complexType>
   </element>
 </sequence>
</complexType>
</schema>
```

Table 3: Handling Cyclic Definitions	

<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="SectionT">
    <xs:sequence>
      <xs:element name="title"
type="xs:string"/>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="body"
type="xs:string"/>
        <xs:element name="section"
type="SectionT"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
	type SECTION_T
	(
	  title varchar2(4000),
	  body VARRAY OF VARCHAR2(4000),
	  section VARRAY OF REF SECTION_T
	);

Table 4: Extensions	

<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="Address">
    <xs:sequence>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
 	
  <xs:complexType name="USAddress">
   <xs:complexContent>
    <xs:extension base="Address">
      <xs:sequence>
        <xs:element name="zip" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>
 	
  <xs:complexType name="IntlAddress">
   <xs:complexContent>
    <xs:extension base="Address">
      <xs:sequence>
        <xs:element name="country"
type="xs:string"/>
      </xs:sequence>
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>
</xs:schema>
	type ADDR_T
	(
	  street varchar2(4000),
	  city varchar2(4000)
	);
	
	type USADDR_T
	under ADDR_T
	(
	 zip varchar2(4000)
	);
	
	type INTLADDR_T
	under ADDR_T
	(
	  country varchar2(4000)
	);

Table 5: XPath Expressions	
Simple XPath expressions: /PurchaseOrder/@PurchaseDate /PurchaseOrder/Company	Involves traversals using child and attribute axis. Rewritten as traversals over object type attributes, where the attributes are simple scalar or object types.
Collection traversal expressions: /PurchaseOrder/Item/Part	Involves traversal of collection expressions using child and attribute axes. Rewritten as joins with the appropriate nested tables.
Predicates: [Company="Oracle"]	Predicates in the XPath are rewritten into SQL predicates.
List indexes: lineitem[1]	Indexes are rewritten to access the n'th item in a collection.

The hybrid storage option is particularly useful when certain parts of the XML document are seldom queried and are mostly retrieved and stored in their entirety. By storing the XML fragments as LOBs, the additional overheads of decomposition and re-composition are avoided.

7. Complex XML Schemas

Some more complex XML Schema constructs and the corresponding structured mappings are discussed next.

XML Schemas can have *cyclic definitions*. A complexType can be defined directly or indirectly in terms of itself. Similarly, the definition of an element can contain a reference back to itself. Such cyclic definitions are supported in XML DB by introducing a REF(reference) attribute at the point of cycle completion. The REF value(s) point at XML fragments that could be stored in the same or different tables, as shown in Table 3.

Two other important constructs are *extension* and *restriction*. A complexType can be declared as a derivation of another global complexType. The derived complexType is mapped as a subtype of the object type corresponding to the parent complexType. In case of derivation by extension, the subtype has extra attributes corresponding to the newly added elements and attributes in the derived complexType. In case of derivation by restriction, the subtype is empty and the restriction semantics are enforced during schema validation. These are shown in Table 4.

8. XML Queries

The XML data stored in a schema-based XMLType table or column can be queried using XPath operators. Support for the XML Query is awaiting completion of the W3C standardization process for that standard, so we will confine our discussion to XPath. XML Query is not expected to be handled any differently.

Specifically, Oracle XML DB provides two operators:

- existsNode : tests for the presence of a node satisfying the given XPath
- extract : retrieves the document fragment identified by the XPath

One of the major benefits of structured storage in XML DB is that queries involving XPath over XML data are rewritten into SQL operators over the underlying columns, This then enables BTree, bitmap and other index access paths to be chosen by the query optimizer. Thus XPath operators can be evaluated against large collections of large XML documents without having to ever construct documents (DOM) in memory. For example a query such as:

```
SELECT * FROM po_tab p
WHEREexistsNode(value(p),
 '/PurchaseOrder[Company=Oracle]');
```

is rewritten to:

```
SELECT * FROM po_tab p
      WHERE p.company = 'Oracle';
```

Table 5 above lists the flavors of XPath expressions that can be translated into equivalent underlying SQL queries.

9. ACKNOWLEDGEMENTS

A number of individuals at Oracle -- Vishu Krishnamurthy, Ravi Murthy, Eric Sedlar, Susan Kotsovolos, Nipun Agarwal, Paul Dixon, Chung-Ho Chen, Subramanian Murlidhar, Mark Drake and many others in their groups have contributed to the development of this technology.

10. REFERENCES

[1] The SQL:1999 Standard, ISO/IEC 9075-n:1999, Published by INCITS, http://www.ncits.org.