XML Schemas in Oracle XML DB

Ravi Murthy, Sandeepan Banerjee

Oracle Corporation
500 Oracle Pkwy
Redwood Shores, USA
{Ravi.Murthy, Sandeepan.Banerjee}@Oracle.com

Abstract

The W3C XML Schema language is becoming increasingly popular for expressing the data model for XML documents. It is a powerful language that incorporates both structural and datatype modeling features. There are many benefits to storing XML Schema compliant data in a database system, including better queryability, optimized updates and stronger validation. However, the fidelity of the XML documents cannot be sacrificed. Thus, the fundamental problem facing database implementers is: how can XML Schemas be mapped to relational (and object-relational) databases without losing schema semantics or data-fidelity? In this paper, we present the Oracle XML DB solution for a flexible mapping of XML Schemas to object-relational databases. It preserves *document fidelity*, including ordering, namespaces, comments, processing instructions etc., and handles all the XML Schema semantics including cyclic definitions, derivations (extension and restriction), and wildcards. We also discuss various query and update optimizations that involve rewriting XPath operations to directly operate on the underlying relational data.

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 from interchange to modelling and storage. 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 artefact of 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. Fidelity of storage to the XML original is critical for many cases including document exchange. For example, ordering of elements is highly relevant in many applications. However, the order of elements in an XML document may not be constrained by the XML Schema declaration (for example, using <choice> or <all> model groups within XML Schema allows elements to appear in any order). In a simple relational mapping, since many of these elements may be flattened into a single row of a table, the relative ordering of these elements is not tracked. Further, uniform queryability is desired over XML whether it be data-oriented or content-oriented. From these considerations, it has become clear that insofar as 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), the relational paradigm needs to be enhanced to efficiently handle XML. This is a core thrust of Oracle XML DB.

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 [1]. 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

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

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 a large number of scalar data types, and this base set of data types can be extended using techniques like restriction, composition, and extension to define more simple and 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.

Oracle has introduced a new datatype for handling XML data, called XMLType [2, 8]. This datatype can be used to define columns of tables and views, arguments to stored procedures and other places where a native datatype could be used. XMLType defines a rich set of XML operators to extract, transform and validate XML data. However, the datatype does not dictate the storage option used to store XML data. In fact, it is designed to accommodate a variety of storage choices, from completely unstructured to highly structured storage. This paper presents the XML Schema-based structured storage option for XMLType. The key benefits of this solution are:

- Full enforcement of XML Schema semantics
- Support for XML document fidelity
- Compact XML Storage by avoiding tag overheads
- Efficient queryability by rewriting XPath queries
- Efficient updates of portions of XML documents by rewriting update operations

Before we look at how XML data conforming to XML Schemas are stored in Oracle XML DB, we briefly recapitulate the ANSI SQL:1999 [3] style Object-Relational technology that is used as infrastructure for high-fidelity XML storage and retrieval in Oracle.

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 a user to treat object data relationally and relational data as objects [4]. 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 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 large object (LOB) types to handle the storage demands of documents or multimedia. Large objects are stored in a manner that optimizes space utilization and provides efficient access. 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 built 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. Table 1 lists some of the common constructs.

Table 1 : Comparison of XML Schema and SQL99

XML Schema Construct	SQL99 Construct
ComplexType	Object Type
Local ComplexType with maxOccurs = 1	Embedded Object Type
ComplexType with maxOccurs > 1	Collection Type
Derived ComplexType	Subtype
XML Schema scalar type	SQL primitive type

4. Oracle and XML Schema

An XML Schema document with additional attributes defined by Oracle XML DB 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 object-relational storage structures. Table 2 shows the default storage structures automatically created in Oracle XML DB for the sample 'purchase order' XML schema po.xsd.

Table 2 : Example XML Schema Mapping

XML Schema (po.xsd)

```
<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_COLL
);
```

XMLType Table & Nested table

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

The object type "Item_T" is created corresponding to the "Item" (local) complexType. An additional collection (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 "PurchaseOrder" (local) complexType. The simple types referenced in the XML Schema are mapped to appropriate SQL datatypes. E.g. The XML Schema primitive types *string*, *decimal* and *date* are mapped to SQL VARCHAR2, NUMBER and DATE types respectively. The constraints specified in the XML Schema such as the maximum length of a string element are preserved in the SQL attribute definitions. For example, "Company" is defined as an attribute of type VARCHAR2 with a maximum length of 100 characters. A registered schema can then be referred to within a CREATE TABLE statement. This results in the underlying SQL types being used to create appropriate 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 the values inserted into the underlying columns. In the case of collections stored in separate tables, one or more rows get inserted into these nested tables. Table 3 shows an instance document and the values in the top level and nested tables.

Table 3 : Example XML Instance Document

XML Instance Document

```
<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

4.1 Validation of XML Schemas

The XML schema document is used both for enforcing schema constraints (in terms of validating the input documents) and also as the source of mapping information that describes how the instances of the XML documents are stored in the database. When instance documents are inserted into a table conforming to an XML Schema, the instance is validated against the schema. In addition, the mapping information in the XML Schema is used to shred the document and store it in the appropriate table(s).

When a XML Schema is registered in Oracle, it is given a name (could be any arbitrary URL). The instance document refers to its XML schema using the xsi:schemaLocation attribute. The value of this special attribute consists of pairs of namespace URLs and schema URLs. The schema URL corresponding to the namespace of the root element identifies the XML schema that is then used for validation and shredding purposes.

4.2 Storage of Collections

When XML is stored in structured format, all simple elements and attributes appearing as direct children of the root element are stored as columns in the root row. Further, nested complex elements and attributes that can occur utmost once (maxOccurs = 1) are also stored as columns in the parent row. However, in the case of elements that can occur more than once (maxOccurs > 1), referred to as collections, there are multiple storage options supported by XML DB.

One option stores the entire collection in a single binary column – referred to as the in-lined storage of collections. The other option is to store collections in a separate nested table which contains one row per collection item. The rows of the nested table contain the key of the parent row. In addition users can specify if they want to preserve the original ordering of the items of a collection. If collection item order needs to be preserved, an additional system column called array_index is added to the nested table. This column of NUMBER datatype tracks the ordering of elements within a collection. When a new collection with N items is inserted, the array_index of these items are assigned values 1… N. 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.

The in-lined storage option has lesser overhead in terms of insert and retrieval processing (since there are fewer rows to insert). However, the major benefit of storing collections in separate nested tables is that you can create indexes on the nested tables to satisfy queries for collection items. Further, efficient piecewise updates of the collection items can be performed without having to overwrite the entire collection with a new collection.

Multiple levels of nesting are handled in Oracle 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 the maxOccurs is greater than 1, a corresponding 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.

One of the key advantages offered by Oracle's storage mechanism is the reduction is space usage when XML documents are stored in the database. The main source of space savings is the fact that tag names are not stored in the instance document. The tags are part of the metadata – as captured by the XML Schema and its mapping to the column names. These tags are regenerated when the XML document or fragment is subsequently retrieved.

4.3 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:

1. whitespaces between elements and between attributes
2. ordering of elements
3. comments within the XML document
4. processing instructions
5. namespaces declarations
6. element and attribute prefixes

In Oracle XML DB, Whitespace fidelity can be maintained by storing an entire XML document in a CLOB. Whitespace-fidelity is typically of less interest to applications than fidelity to the XML Document Object Model (DOM), the standard interface that allows

programs and scripts to dynamically access and update the content, structure and style of documents [5]

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, `SYS_XDBPD$`, 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
- Mixed content – text nodes that are intermixed with elements are stored in the system column.

4.4 'XDB' Attributes

XML DB defines a set of attributes in the xdb namespace *http://xmlns.oracle.com/xdb*. These attributes can be used within a XML Schema document to influence various aspects of XML storage and processing. For example, the xdb:SQLName is used to specify the name of the attribute in the corresponding object type. The table 4 lists the XDB attributes and their functions.

Table 4 : XDB Attributes

XDB Attribute Name	Function
SQLName	Specifies the name of attribute in corresponding object type. If this attribute is omitted, the name is generated by mangling the XML element or attribute name.
SQLType	Specifies name of object type created – applicable for complexType declarations only. If this attribute is omitted, the name of the SQL object type is generated from the name of the complexType or complex element.
SQLCollType	Specifies the name of the collection type – applicable for complexType declarations with maxOccurs > 1
storeVarrayAsTable	Specifies whether the corresponding varray is stored in-lined in the row or as multiple rows in a separate table.
maintainOrder	Specifies if ordering of elements within a collection is relevant – this attribute determines whether a varray type or nested table type is created.
maintainDOM	Specifies if the DOM fidelity in terms of comments, PIs, etc is relevant – this attribute determines if the system PD column is created.

4.5 Hybrid Storage Options

XML DB supports the 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" when the entire XML document is stored in a single LOB.

A novel aspect of 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>
```

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 these XML fragments as LOBs, the additional overheads of decomposition and re-composition are avoided.

4.6 Complex XML Schemas

This section discusses various interesting XML Schema constructs and their corresponding structured mappings.

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 by XML DB by introducing a REF (reference) attribute at the point of cycle completion. The REF value(s) point at XML fragments which are stored in the same or in different tables. Table 5 shows an example of a cyclic definition and the corresponding SQL type.

Table 5 : Cyclic Definition

XML Schema

```
<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>
```

SQL Types

```
type SECTION_T
(
  title varchar2(4000),
  body VARRAY OF VARCHAR2(4000),
  section VARRAY OF REF SECTION_T
);
```

Table 6: ComplexType Derivations

XML Schema

```
<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>
```

SQL Types

```
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)
);
```

ComplexType Derivations: 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. In Table 6, the types *USAddress* and *IntlAddress* extend the base type *Address*. The SQL type *ADDR_T* corresponding to Address is extended by types *USADDR_T* and *INTLADDR_T*.

Wildcards:
An XML Schema could allow for arbitrary XML to appear at certain portions of the XML documents. This is achieved by the use of the <any> declaration. This is mapped to a CLOB column that permits (in general) any wellformed XML to be stored in that column. In addition, the <any> declaration could specify namespace constraints such as allowing only elements belonging to namespaces other than the schema's namespace (##other) or unqualified elements (##local) or provide a list of permitted namespace URLs. The namespace constraints are enforced by Oracle prior to storing the entire fragment within the CLOB column.

5. Queries

The XML data stored in a schema-based XMLType table or column can be queried using XPath operators. Specifically, the following operators are provided in Oracle XML DB. The *existsNode* operator tests for the presence of a node satisfying the given XPath. The *extract* operator retrieves the document fragment identified by the XPath. The *extractValue* operator returns the raw value of the leaf node 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 rewrite further enables B-Tree, 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, consider a query such as:

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

The above query is silently rewritten to the following:

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

In the above query, “company” is the column underlying the Company simple element. Similarly, extract() operators are also rewritten to directly retrieve data from the underlying columns. In the example below, the select list and the where clause expressions are rewritten into the underlying columns.

Original Query

```
SELECT extract(value(p),
  '/PurchaseOrder/PONum') FROM po_tab p
WHERE existsNode(value(p),
'/PurchaseOrder[Company=Oracle]');
```

Rewritten Query

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

Further, if a B-Tree index is created on the “company” column, the execution plan for the above query is as follows:
Index Lookup -> Rowid Access

The table below lists the flavors of XPath expressions that can be translated into equivalent underlying SQL queries.

Table 7 - XPath Rewrites

XPath Expression	Description

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]`	List indexes are rewritten to access the n'th item in a collection. An index on the ARRAY_INDEX column (if present) is used to quickly access the appropriate item.

5.1 Functional Indexes

B-Tree indexes can be created on arbitrary expressions including results of user defined functions. These are referred to as functional indexes. The query optimizer picks a functional index when the corresponding expression is used as a query predicate. For example, a user could choose to not shred an XML document. Instead it could be stored as a CLOB (using the appropriate schema annotations). Then, functional indexes can be built on specific fragments of XML data to allow index access paths for SQL queries. In the example below, `po_clob` is a CLOB column (in a table called po_tab2) storing purchase orders and a functional index is created on the “Company” element value as follows:

```
CREATE INDEX po_func_idx
ON po_tab2
(extract(po_clob,
‘/PurchaseOrder/Company’));
```

The functional index can be used to accelerate queries that lookup purchase orders based on the "Company" value.

5.2 Text Indexes

XML DB provides a rich full-text search capability that is also aware of XML sections, to help perform keyword searches over XML stored in CLOBs. In the case where keyword searches are needed over documents stored in structured format, temporary CLOBs are generated at index-creation time.

```
CREATE INDEX po_ctx_idx
ON po_tab2(po_clob)
INDEXTYPE IS ctxsys.context;
```

The above Oracle Text index can be used to satisfy queries that use the Contains() operator to perform keyword based searches. The query below searches for all documents that contain both the keywords Oracle and XML.

```
SELECT * FROM po_tab2
WHERE Contains(po_clob, 'Oracle AND XML');
```

The following query restricts the keyword search to a specific section of the XML documents using the INPATH query operator.

```
SELECT * FROM po_tab2
WHERE Contains(po_clob,
   'Oracle INPATH /PurchaseOrder/Item');
```

6. Updates

The XML data stored in XMLType tables can be updated by replacing the entire document with a new XML document – or updating only certain portions of it. Oracle XML DB provides an `updateXML` operator to update only portions of the document. This operator identifies the target node using an XPath expression and specifies its new value. For example, the statement below updates the PONum value of a specific purchase order.

```
UPDATE po_tab p
SET value(p) =
 updatexml(value(p),
           '/PurchaseOrder/PONum/text()',
           9999)
WHERE existsNode(value(p),
'/PurchaseOrder[Company=Oracle]');
```

The UPDATE statement is rewritten to directly update the underlying column of the table.

```
UPDATE po_tab p
  SET p.ponum = 9999
WHERE p.Company = 'Oracle';
```

Rewriting the update statement enables significant performance gains because of avoiding the need to construct a DOM object in memory. Further, the predicate used to identify the documents to be updated is also rewritten and can use relational indexes as access paths.

A similar mechanism is used to optimize manipulation of collection elements i.e. inserting or deleting elements within a collection. This is achieved by defining new SQL operators – `addXML` to insert a XML fragment into the collection identified by the given XPath and `deleteXML` that deletes the specified node. These operators are rewritten as appropriate SQL statements against the underlying nested tables. For example, the statement below appends a new `Item` element.

```
UPDATE po_tab p SET value(p) =
addXML(value(p),
'/PurchaseOrder/LineItems',
'<Item>
   <Part>10i Doc Set</Part>
   <Price>3450</Price>
 </Item>')
WHERE existsNode(value(p),
'/PurchaseOrder[Company=Oracle]');
```

The above statement is rewritten to directly insert a new row into the `item_tab` nested table.

7. XML Views

XML DB provides a mechanism to create XML views over relational and object-relational data. Additionally, a XML Schema can be attached to the view definition, thereby creating a schema-based XMLType view. The query comprising the view definition can either rely on the default mapping of object-relational data to XML as specified by the XML Schema – or use the standard SQL/XML operators to create ad-hoc XML from underlying data. The examples below show the two mechanisms for constructing XML Views.

Example: XML View over object-relational data
The table `po_obj_tab` is an object table of type `PurchaseOrder_T`. The view is created based on the previously registered XML Schema "po.xsd". Note that the mapping from object-relational to XML is specified within the XML Schema document. Hence, no further information is required during the view creation.

```
CREATE VIEW po_view of XMLTYPE
XMLSCHEMA "po.xsd" ELEMENT "PurchaseOrder"
AS
SELECT value(p) FROM po_obj_tab p;
```

Example: XML View over relational data using SQL/XML operators

In this example, `po_rel_tab` and `items_rel_tab` are relational tables for storing purchase orders. The SQL/XML operator `XMLElement` creates an element value in the output XML document. The `XMLForest` operator creates a set of child element values. Finally, `XMLAgg` is an aggregate operator over the inner select statement which returns a single (concatenated) value corresponding to all the `Item` values.

```
CREATE VIEW po_view of XMLTYPE
XMLSCHEMA "po.xsd" ELEMENT "PurchaseOrder"
AS
SELECT
   XMLElement("PurchaseOrder",
     XMLForest(p.ponum "PONum",
               p.company "Company"),
      (SELECT XMLAGG(
               XMLElement("Item",
                  XMLForest(i.part
                     "Part",
                     i.price,
                     "Price"))
       FROM items_rel_tab i
       WHERE i.po_id = p.id))
FROM po_rel_tab p;
```

7.1 XML Views for Data Integration

XML Views are also useful in integrating data from multiple, potentially disparate data sources including other databases, etc. This is achieved by accessing the external data using External Tables and/or Database Links [9]. The XML view integrates all the data from multiple databases, web sources and content servers into a single XML model.

Example: XML View over external data sources
This example shows a XML View created over two external tables accessed via database links db1 and db2.

```
CREATE VIEW catalogs of XMLTYPE
XMLSCHEMA "catalog.xsd" ELEMENT "Catalog"
AS
SELECT …
FROM listings@db1, inventory@db2
WHERE …;
```

7.2 Queries over XML Views

XML Views can be queries using the same set of operators as XMLType tables e.g. `extract`, `existsNode`, etc. The user queries are rewritten to operate directly over the underlying query expressions. This enables further optimizations in terms of index access methods, and avoids the need for costly in-memory DOM operations.

```
SELECT extract(value(p),
          '/PurchaseOrder/PONum')
FROM po_view p
WHERE existsNode(value(p),
'/PurchaseOrder[Company=Oracle]');
```

8. Future Directions

As XML Query [7] becomes standard, we will support the XML Query syntax over XML documents. Further, the XML Query syntax will work against distributed databases using standard connectivity mechanisms provided by Oracle.

For highly variable or semi-structured (typically schema-less) XML documents, different techniques are needed for efficient storage and retrieval. We are developing new storage and indexing mechanisms for such documents.

9. Conclusions

The SQL:1999 object-relational mechanisms provide an attractive, standard way of absorbing the XML Schemas in databases. We have used such mechanisms to absorb the XML Schema data model into the Oracle server, ensuring fidelity to the XML DOM, and providing new standard access methods for navigating and querying XML. This merging of the XML and SQL models creates an attractive ‘duality’ whereby applications can perform SQL operations on XML data, and XML operations on SQL data.

10. Acknowledgements

Over the years a number of individuals have made key contributions to the gradual development of the capabilities described in this paper. Vishu Krishnamurthy, Susan Kotsovolos, Muralidhar Krishnaprasad, Anand Manikutty and other members of their teams have developed the SQL object-relational functionality. Eric Sedlar, Nipun Agarwal and Mark Drake have helped build capabilities related to XML stored as content. Paul Dixon, Chung-Ho-Chen and their teams have developed full-text query support over XML.

11. References

[1] The W3C XML Schema Standard (Schema Working Group), see http://www.w3.org/XML/Schema.

[2] Oracle XML DB Developer's Guide, Oracle 9iR2. See http://otn.oracle.com/tech/xml/xmldb/

[3]The ANSI-ISO SQL:1999 Standard see, http://www.ncb.ernet.in/education/modules/dbms/sql99index.html.

[4] Vishu Krishnamurthy, Sandeepan Banerjee, Anil Nori: Bringing Object-Relational Technology to Mainstream. SIGMOD Conference 1999 : 513-514

[5]The W3C XML DOM Standard (DOM Working Group), see http://www.w3.org/DOM/

[6]The International Committee for Information Technology Standards H2.3 Task Group see, http://www.sqlx.org/.

[7]The W3C XML Query Group see, http://www.w3.org/XML/Query.

[8] Sandeepan Banerjee, Vishu Krishnamurthy, Muralidhar Krishnaprasad, Ravi Murthy : Oracle8i - The XML Enabled Data Management System, ICDE 2000.

[9] Oracle 9i Heterogeneous Connectivity Administrator's Guide.