Características objeto-relacional y de soporte de documentos XML de Oracle Database

Juan Francisco Adame Lorite

Bases de datos en la Distribución

Julio 2003

Introducción

Este documento trata de las características y funcionalidades XML del producto *software* comercial Oracle Database en su versión 9i Release 2. Estas características no son sólo el almacenamiento de este tipo de datos como cadenas de caracteres sino el modelado de estos tipos, su clasificación e indexado en el sistema de modelado de datos del producto, permitiendo peticiones y manipulación avanzada de los mismos.

La naturaleza de estructura no plana de los documentos XML obliga para su modelado hacer uso de las características objeto-relacionales de la base de datos. Por eso, la primera parte de este documento habla del modelo objeto-relacional de Oracle.

La segunda parte del documento se centra en el propósito del mismo: documentos XML en la base de datos Oracle.

Modelo objeto-relacional de Oracle Database 9i

Según la forma del modelado de los datos podemos dividir los *software* de bases de datos en tres tipos: relacionales, orientadas a objetos y objeto-relacionales.

Las bases de datos relacionales son una tecnología muy madura y eficiente, pero su principal defecto es la planaridad de su modelo. Esto ha dificultado el modelado de los tipos de datos objetos que carecen de esta planaridad obligando a establecer capas de transformación del modelo orientado a objetos de la aplicación con el modelo plano relacional de la base de datos. Esto exigía una doble labor de diseño en ambos planos y el añadido del *wrapper* entre los dos modelos.

Las bases de datos orientadas a objetos aparecen para cubrir esta problemática. Al ser el modelado de los elementos de estos *software* también orientado a objetos, un único diseño era necesario y el *wrapper* desaparece o se simplifica enormemente. Sin embargo, el nuevo modelo orientado a objetos era mucho más ineficiente y lento.

La última opción en aparecer fue la objeto-relacional. Este modelo combina las ventajas de los dos anteriores: la base de datos es relacional por lo que conserva su rapidez y eficiencia, pero permite hacer uso de nuevos elementos que modelan los objetos a esta base de datos relacional, con lo que el analista y diseñador ve un modelo orientado a objetos. Oracle Database 8i en adelante pertenece a este último tipo.

Los objetos en Oracle 9i se denominan tipos abstractos de datos y poseen funcionalidades de herencia y métodos implementados en SQL o Java. La problemática de la planaridad se resuelve con las tablas anidadas y los *arrays* variables. Las relaciones entre objetos se establecen mediante el tipo de referencia REF. El almacenamiento de conjuntos de datos no estructurado se realiza con los tipos de objetos grandes. Y la base de datos relacional se puede modelar como objetos mediante las vistas de objeto.

Tipos abstractos de datos (clases y objetos)

Las clases son los tipos abstractos de datos. Los tipos abstractos poseen atributos, relaciones y métodos. Los atributos son tipos que establecen el estado del objeto, las relaciones son referencias a otros objetos existentes al margen de este y los métodos son las acciones que el objeto puede realizar.

Oracle permite modelar los objetos en filas o columnas. Un objeto modelado en filas es una tabla en que cada fila es del tipo del objeto y sus atributos nativos son modelados como columnas. Un objeto modelado en columnas es una columna dentro de una tabla u objeto del tipo objeto y que se modela expandiendo sus atributos como nuevas columnas de la tabla o atributos del objeto respectivamente.

En el ejemplo siguiente, el objeto purchase_order se modelará con su atributo id seguido de los atributos de person: name y phone, y de una referencia a la tabla anidada lineitems. El cuerpo del método get_value se define posteriormente en otra operación como se indica más adelante.

```
CREATE TYPE person AS OBJECT (
     name VARCHAR2(30),
     phone VARCHAR2(20) );
CREATE TYPE lineitem AS OBJECT (
     item_name VARCHAR2(30),
     quantity NUMBER,
     unit_price NUMBER(12,2) );
CREATE TYPE lineitem_table AS TABLE OF lineitem;
CREATE TYPE purchase_order AS OBJECT (
    id NUMBER,
    contact person,
    lineitems lineitem_table,
    MEMBER FUNCTION
         get_value RETURN NUMBER );
CREATE TABLE orders OF purchase_order
    NESTED TABLE lineitems STORE AS lineitems_table;
```

Toda clase o colección tiene un constructor creado automáticamente para crear una instancia (objeto) de una clase utilizado en la inserción de elementos. Este constructor tiene como parámetros los atributos que definen la clase por el mismo orden en que fueron definidos. Por ejemplo para insertar un nuevo objeto de tipo purchase_order en la base de datos:

```
INSERT INTO orders VALUES (
    45,
    person ('Juan','555'),
    lineitem_table (
         lineitem('Piedras',4,3.5),
         lineitem('Churros',12,0.4)
    )
);
```

Para referenciar a los atributos, métodos y referencias se hace uso de la nomenclatura habitual del punto entre objeto y atributo. Por ejemplo para hacer una petición de elementos:

```
SELECT * FROM orders o WHERE o.contact.name='Juan' AND
    o.contact.phone='555';
```

O bien:

```
SELECT VALUE(o) FROM orders WHERE
    VALUE(o.contact)=person('Juan','555');
```

Ambos devuelven el mismo valor, salvo que la primera en forma de tabla siendo cada columna un atributo del objeto resultado de la consulta. Y en el segundo devuelve el objeto con la forma de constructor explicada en el ejemplo anterior.

La creación de índices en objetos es igual a la de tablas relacionales identificando el elemento de indexación mediante la referencia al atributo índice. Por ejemplo, para indexar el ejemplo anterior por el nombre del contacto:

```
CREATE INDEX NombreContacto ON orders(contact.name);
```

Los métodos se declaran en la declaración del objeto pero se implementan posteriormente asignando una función o método externo al del objeto. Su implementación puede realizarse en SQL o Java.

Herencia de clases

Oracle permite la herencia simple de clases a través del modificador UNDER. En el ejemplo del apartado anterior, añadimos la fecha a los pedidos:

```
CREATE TYPE date_order UNDER orders (
    date DATE
) NOT FINAL;
```

La cláusula FINAL indica si se puede heredar la clase por un subtipo (NOT FINAL) o si es una clase final y por lo tanto no se puede extender (FINAL). En este ejemplo si es heredable.

Otra cláusula es INSTANTIABLE que indica cuando la clase es abstracta y no se puede instanciar (NOT INSTANTIABLE) y cuando no lo es y se pueden tomar objetos de la clase (INSTANCIABLE).

La cláusula TREAT es el operador de *casting* de un tipo de objeto a subtipos del mismo. Si la conversión no es posible se devuelve NULL. El siguiente ejemplo busca en la tabla de date_orders pero devuelve los elementos respuesta como purchase_orders.

```
SELECT VALUE(TREAT(o AS orders)) FROM date_orders WHERE
    VALUE(o.contact)=person('Juan','555');
```

El operador de predicados para saber si un objeto es instanciable a una clase es IS OF.

Tablas anidadas y arrays variables

Los *arrays* variables y las tablas anidadas permiten modelar las relaciones de uno a varios que son muy comunes en los modelos orientados a objetos. Estos tipos se denominan colecciones, porque representan conjuntos de datos de un mismo tipo.

Cuando creamos una tabla anidada además de indicar que atributo es, debemos indicar que tabla externa será la que almacene los datos. En el ejemplo del apartado de los tipos abstractos es lineitems_table:

```
CREATE TABLE orders OF purchase_order
    NESTED TABLE lineitems STORE AS lineitems_table;
```

Los *arrays* variables son de tamaño fijo por lo que no es necesario guardarlos en una tabla externa y se almacenan en la propia tabla (u objeto). Haciendo uso del ejemplo anterior pero con un *array* variable de 10 elementos en vez de una tabla anidada:

```
CREATE TYPE lineitem_array AS VARRAY(10) OF lineitem;
CREATE TYPE purchase_order AS OBJECT (
    id NUMBER,
    contact person,
    lineitems lineitem_array,
    MEMBER FUNCTION
         get_value RETURN NUMBER );
CREATE TABLE orders OF purchase_order;
```

Hay dos formas de acceder a los elementos de las colecciones, la primera es recibiendo la colección como un solo elemento en la forma de su constructor. La segunda es devolver en cada fila de la respuesta a la consulta un elemento de la colección mediante la función TABLE.

Un ejemplo del primer tipo sería el siguiente, en el que se devuelve un valor de tipo colección que engloba a todos los valores de la colección:

```
SELECT e.id, e.lineitems FROM orders e;

id       lineitems
--     ---------
4        lineitem_array(lineitem(…),lineitem(…),…)
```

Un ejemplo del segundo tipo es el siguiente en el que se devuelve una fila por cada valor de la colección:

```
SELECT e.id, p.* FROM orders e, TABLE(orders.lineitems) p;

id       lineitems
--     ---------
4        lineitem('chorizo',4,3.5)
4        lineitem('jamón',3,7.6)
4        …
```

Igualmente para la inserción o modificación de colecciones podemos hacer uso del constructor de la colección para introducirle el conjunto de valores por completo modificándose todo el *array* variable o tabla anidada de una vez:

```
INSERT INTO orders VALUES (
    1,
    person('Juan','555'),
    lineitem_array(
        lineitem('chorizo',4,3.5),
        lineitem('jamón',3,7.6)
    )
);
```

O en el caso de una tabla anidada (no es posible en un *array* variable) acceder a la tabla directamente haciendo uso de la cláusula TABLE, lo que nos permitiría acceder de forma selectiva a los elementos:

```
UPDATE
    TABLE (SELECT e.lineitems FROM orders e WHERE e.id=4) P
    SET VALUE(P)=lineitem('pan',1,0.5)
    WHERE P.item_name='pan';
```

Objetos grandes

Los objetos grandes (Large objects) son los nuevos tipos de datos nativos que Oracle soporta para almacenar cantidades de datos muy grandes, hasta 4 Gb. Existen cuatro tipos:

- CLOB: LOB de caracteres ASCII o código ASCII extendido.
- BLOB: LOB de contenido binario.
- BFILE: No es un LOB propiamente dicho, es un puntero a un elemento en el sistema de ficheros. Su tamaño máximo dependerá de las características del sistema de ficheros. El borrado del tipo BFILE no implica el borrado del fichero sino de la referencia.
- NCLOB: LOB de caracteres multibyte.

Estos tipos permiten almacenar y manipular tipos de datos que requieren un gran volumen de almacenamiento, como el contenido multimedia por ejemplo.

El acceso a este tipo de datos es similar al del resto de los tipos nativos, pero el sistema permite algunas bibliotecas adicionales para la manipulación especial de estos datos, DBMS_LOB o las de manipulación de cadenas. Su inicialización puede ser a NULL o bien haciendo uso de las funciones EMPTY_BLOB, EMPTY_CLOB y BFILENAME.

Tipos referencia

Para modelar las asociaciones de objetos se utilizan los tipos referencia. Se puede considerar que son punteros a objetos (objetos modelados en filas), aunque esta referencia no se realiza mediante punteros a memoria, sino mediante identificadores únicos de objetos, *oid*. Los tipos referencia modelan las relaciones del tipo muchos a uno.

Los tipos REF se declaran indicando a que clase apuntan y es posible indicar a que tabla de objetos (modelados en filas) de esa clase referencian. Indicar esta tabla con la cláusula SCOPE IS restringe los elementos que puede apuntar la referencia a sólo los de esa tabla, pero facilita e incrementa la velocidad de acceso a los elementos mediante estas referencias.

En el siguiente ejemplo el atributo contact no contiene un objeto person, sino una referencia (o un valor NULL si no referencia a ningún elemento) a un objeto de tipo person en la tabla persons. La gestión de los datos de la tabla persons es independiente, es más el

borrado de un elemento en la tabla persons no conlleva la anulación o modificación de la referencia en orders, que referencia a un elemento no existente. Esta última situación se puede comprobar con el predicado IS DANGLING.

```
CREATE TABLE persons OF person;
CREATE TYPE purchase_order AS OBJECT (
    id NUMBER,
    contact REF person SCOPE IS persons,
    lineitems lineitem_array,
    MEMBER FUNCTION
         get_value RETURN NUMBER );
CREATE TABLE orders OF purchase_order;
```

Los tipos referencia permiten navegar a través de la estructura de objetos de la misma manera que si fuesen un atributo del objeto, mediante el operador '.'.

```
SELECT o.contact.name FROM orders o;
```

Oracle posee dos funciones para los tipos referencia: REF que devuelve el identificador de objeto dado la instancia de un objeto y DEREF que dado el identificador de un objeto devuelve la instancia del objeto, es el opuesto de la función REF. Con lo que el siguiente comando devuelve la instancia del objeto:

```
SELECT DEREF(REF(o)) FROM orders o;
```

Vistas de objeto

Para convertir el modelo relacional en el modelo orientado a objetos sin necesidad de modificar los datos ni su estructura (metadatos) es posible generar vistas de los datos relacionales de tal manera que estos puedan ser vistos como objetos.

```
CREATE TABLE emp_table (
    empnum NUMBER (5),
    ename VARCHAR2 (20),
    salary NUMBER (9, 2),
    job VARCHAR2 (20)
);
CREATE TYPE employee_t AS OBJECT(
    empno NUMBER (5),
    ename VARCHAR2 (20),
    salary NUMBER (9, 2),
    job VARCHAR2 (20)
);
CREATE VIEW emp_view1 OF employee_t
    WITH OBJECT IDENTIFIER (empno) AS
    SELECT e.empnum, e.ename, e.salary, e.job
    FROM emp_table e
    WHERE job = 'Developer';
```

Métodos

Existen tres tipos de métodos a definir en Oracle: métodos propiamente dichos, métodos de comparación y constructores.

Los métodos representan la interacción de los objetos con el resto del medio modelado mediante objetos. Los métodos, su firma, se indican en la definición del objeto, pero se implementan posteriormente mediante el comando CREATE TYPE BODY. El ejemplo siguiente muestra una clase que representa un quebrado y su método de simplificación:

```
CREATE TYPE Rational AS OBJECT (
    num INTEGER,
    den INTEGER,
    MEMBER PROCEDURE normalize,
    ...
);
CREATE TYPE BODY Rational AS
    MEMBER PROCEDURE normalize IS
        g INTEGER;
    BEGIN
        g := gcd(SELF.num, SELF.den);
        g := gcd(num, den); -- equivalent to previous line
        num := num / g;
        den := den / g;
    END normalize;
    ...
END;
```

Los métodos de comparación son métodos que permiten establecer mapeado u orden de elementos para las comparaciones de objetos. Estas funciones son muy útiles para funciones de agrupación del estilo DISTINCT, GROUP BY y ORDER BY.

Los métodos de mapeado convierten un objeto complejo en un tipo de dato representable en un sólo eje, un escalar, para poder mapear los objetos en la recta. Un ejemplo de método de mapeado es el siguiente:

```
CREATE TYPE Rectangle_typ AS OBJECT (
    len NUMBER,
    wid NUMBER,
    MAP MEMBER FUNCTION area RETURN NUMBER,
    ...
);
CREATE TYPE BODY Rectangle_typ AS
    MAP MEMBER FUNCTION area RETURN NUMBER IS
    BEGIN
        RETURN len * wid;
    END area;
    ...
END;
```

Los métodos de orden comparan dos objetos del mismo tipo para establecer un orden entre objetos. Un ejemplo de método de orden es el siguiente:

```
CREATE TYPE Customer_typ AS OBJECT (
    id NUMBER,
    name VARCHAR2(20),
    addr VARCHAR2(30),
    ORDER MEMBER FUNCTION match (c Customer_typ) RETURN
        INTEGER
);
CREATE TYPE BODY Customer_typ AS
    ORDER MEMBER FUNCTION match (c Customer_typ) RETURN
        INTEGER IS
    BEGIN
        IF id < c.id THEN
             RETURN -1; -- any negative number will do
        ELSIF id > c.id THEN
             RETURN 1; -- any positive number will do
        ELSE
             RETURN 0;
        END IF;
    END;
END;
```

Los métodos constructores, que son implícitamente creados por la base de datos, tienen como objetivo crear instancias de objetos a partir del estado definido por sus atributos. El constructor implícito creado por la base de datos tiene como nombre el mismo que la clase y como parámetros sus atributos en el mismo orden en que se definieron en la clase. Aún así, es posible definir constructores propios en que se soliciten otros atributos y el proceso de construcción no sea simplemente la asignación de valores a sus atributos.

```
CREATE TYPE Customer_typ AS OBJECT (
    id NUMBER,
    name VARCHAR2(20),
    phone VARCHAR2(30),
);
cust = Customer_typ(103, "Ravi", "1-800-555-1212");
```

Aunque todos los ejemplos de este apartado están realizados en SQL, es posible implementar estos métodos en Java. Una vez implementada la clase Java y compilada, se carga en la base de datos, a partir de este momento la clase y sus métodos están disponibles en Oracle. A continuación se define el tipo abstracto de la manera explicada hasta ahora y se indica que el método lo implementa una clase Java y cómo se mapean los tipos de Java a Oracle. La firma del método en Oracle va seguida del comando EXTERNAL NAME con la firma del método en Java de donde se infiere el mapeado de tipos. El siguiente ejemplo muestra un tipo abstracto con métodos, métodos estáticos que hacen labor de constructores, y un método de definición de orden entre objetos:

```
CREATE TYPE person_t AS OBJECT
    EXTERNAL NAME 'Person' LANGUAGE JAVA
    USING SQLData (
        ss_no NUMBER (9) EXTERNAL NAME 'socialSecurityNo',
        name varchar(100) EXTERNAL NAME 'name',
        address full_address EXTERNAL NAME 'addrs',
        birth_date date EXTERNAL NAME 'birthDate',
    MEMBER FUNCTION age () RETURN NUMBER EXTERNAL NAME 'age
        () return int',
    MEMBER FUNCTION address RETURN full_address EXTERNAL NAME
        'get_address () return long_address',
    STATIC create RETURN person_t EXTERNAL NAME 'create ()
        return Person',
    STATIC create (name VARCHAR(100), addrs full_address,
        bDate DATE) RETURN person_t EXTERNAL NAME 'create
        (java.lang.String, Long_address,
        oracle.sql.date) return Person',
    ORDER FUNCTION compare (in_person person_t) RETURN NUMBER
        EXTERNAL NAME 'isSame (Person) return int'
)
```

Documentos XML en Oracle Database

XML es el formato de codificación/estructuración para el intercambio de datos más común hoy en día. Sin embargo el modelado de este tipo de datos en bases de datos relacionales era complicado por el mismo motivo que el modelado de objetos: los documentos XML no tienen una estructura plana con lo que su representación en tablas se hace complicada.

La opción hasta hace poco era el almacenar los documentos en objetos grandes de tipo CLOB o NCLOB y analizar/transformar estos elementos tomando este flujo de caracteres y haciéndolos pasar por un parseador XML habitual. No se aprovechaba ninguna de las características de la base de datos para mejorar la eficiencia de búsquedas y consultas, salvo por alguna característica de indexado avanzada de texto (Oracle Text).

Con la llegada del modelo objeto-relacional el problema de la planaridad se resuelve y las mismas facilidades que se daban para el modelado de objetos se ofrecen ahora para el modelado de los objetos XML en la estructura de la base de datos. Esta integración permite hacer uso de la eficiencia de las consultas y flexibilidad de recorrido de estructuras del modelo relacional en estos documentos.

XMLType

Este mapeado de la estructura XML a la estructura de la base de datos en Oracle se realiza con el tipo XMLType, que es un tipo abstracto. El tipo XMLType se almacena en un tipo CLOB aunque puede asociarse a un Schema XML para la definición de su estructura lo que obliga que cualquier documento sea validado con este esquema. En este segundo caso el esquema del documento se modela en la estructura objeto-relacional de la base de datos.

La ventaja de hacerlo de la primera manera es que todo tipo de documentos XML pueden almacenarse en ese elemento XMLType. La segunda obliga a que el elemento sea válido frente al esquema asociado, aunque su mapeado en la estructura objeto-relacional permite tratar el documento de manera más eficiente y flexible.

Mapeado de XMLType dado un esquema XML

Los elementos del esquema XML se mapean como objetos en los que cada elemento anidado de tipo simple es representado por un atributo de un tipo nativo lo más acorde posible con el tipo del esquema: si es un número con NUMBER, si es texto con VARCHAR,… Aún así es posible forzar la representación del elemento a un tipo de Oracle mediante el atributo SQLType utilizado en el elemento del esquema.

Cuando un elemento contiene un elemento complejo, este es modelado con un objeto y el elemento padre establece una referencia a él con tipos referencia. Es posible forzar que el mapeado de los tipos complejos se realice en CLOB, NCLOB o VARCHAR (sin ser

representados en el modelo objeto-relacional) mediante el atributo SQLType (=CLOB) utilizado en el elemento del esquema.

Cuando la ocurrencia de un elemento, bien simple o complejo, es mayor que uno el elemento es representado en el objeto padre con un *array* variable si el número de ocurrencias máximas es finito o con un tabla anidada si es infinito.

El mapeado se entiende mucho mejor con el siguiente ejemplo:

```
declare doc varchar2(3000) :=
  '<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" maxOccurs="unbounded">
            <complexType>
              <sequence>
                <element name = "Street" type = "string"/>
                <element name = "City" type = "string"/>
              </sequence>
            </complexType>
          </element>
        </sequence>
      </complexType>
    </schema>';
```

Se mapearía de la siguiente manera:

```
CREATE TYPE Addr_t AS OBJECT (
     street VARCHAR2(4000),
     city VARCHAR2(4000) );
CREATE TYPE Addr_t_table AS TABLE OF Addr_t;
CREATE TYPE Employee AS OBJECT (
    Name VARCHAR2(4000),
    Age NUMBER,
    Addr_table Addr_t_table );
```

Crear tables/columnas XMLType

Para crear columnas o tablas XMLType sólo tenemos que hacerlo como lo hicimos al definir columnas o tablas de objetos. Si no validamos el tipo XMLType con un esquema, los ejemplos serían los siguientes (el primero es una tabla con una columna tipo XMLType y el segundo es una tabla de XMLType):

```
CREATE TABLE warehouses(
    warehouse_id NUMBER(3),
    warehouse_spec XMLTYPE,
    warehouse_name VARCHAR2(35),
    location_id NUMBER(4));
CREATE TABLE po_xtab of XMLTYPE;
```

Para definir una columna o tabla XMLType asociada a un esquema debemos registrar primero el esquema en la base de datos. Esto se realiza mediante la biblioteca DBMS_XMLSCHEMA que posee dos funciones: registerSchema para registrar el esquema y deleteSchema para eliminar el registro.

```
DBMS_XMLSCHEMA.registerSchema(
    'http://www.oracle.com/PO.xsd', doc);
DBMS_XMLSCHEMA.deleteSchema('http://www.oracle.com/PO.xsd',
    DBMS_XMLSCHEMA.DELETE_CASCADE_FORCE);
```

Una vez registrado el esquema podemos crear columnas y tablas XMLType asociadas haciendo uso del comando XMLSCHEMA en la definición:

```
CREATE TABLE po_tab(
    id NUMBER,
    po sys.XMLType
)
XMLTYPE COLUMN po
XMLSCHEMA "http://www.oracle.com/PO.xsd"
ELEMENT "PurchaseOrder";

CREATE TABLE po_tab OF XMLTYPE
    XMLSCHEMA "http://www.oracle.com/PO.xsd"
    ELEMENT "PurchaseOrder";
CREATE TABLE po_tab OF XMLTYPE
    ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";
```

Insertar documentos XML

Si tratamos a los tipos XMLType como objetos podemos utilizar el constructor de dichos objetos para instanciar nuevos elementos XMLType, tomando como parámetro la cadena que representa al documento XML:

```
INSERT INTO warehouses VALUES
    ( 100, XMLType(
         '<Warehouse whNo="100">
          <Building>Owned</Building>
          </Warehouse>'),
    'Tower Records', 1003);

UPDATE warehouses SET warehouse_spec = XMLType
    ('<Warehouse whono="200">
    <Building>Leased</Building>
    </Warehouse>');
```

También podemos hacer uso de los comandos SQLX para generar el documento XML en vez del constructor de XMLType como veremos en el apartado de SQLX.

Consultar documentos XML

Es posible recuperar un documento en forma de CLOB, VARCHAR o NUMBER mediante los métodos de XMLType: getClobVal, getStringVal, getNumberVal. Con estas funciones simplemente obtenemos el documento XML convertido en un tipo nativo.

Las consultas no sólo son de recuperación de documentos completos, es posible recuperar partes del documento y efectuar predicados de selección en partes del documento. Estas partes se basan en la estructura DOM de XML y se señalan haciendo uso de XPath. Las funciones incluidas con este propósito son extract y existsNode: el primero devuelve el nodo del documento XML (de la estructura DOM) solicitado y el segundo devuelve verdadero (1) cuando existe el nodo solicitado. En el siguiente ejemplo se buscan los documentos que posean el elemento PNAME dentro de PO situado en el nodo raíz del documento y que PNAME contenga un nodo de texto con el valor "po_2":

```
SELECT e.poDoc.getClobval() AS poXML
    FROM po_xml_tab e
    WHERE e.poDoc.existsNode('/PO[PNAME = "po_2"]') = 1;

POXML
--------------------
<?xml version="1.0"?>
  <PO pono="2">
    <PNAME>Po_2</PNAME>
    <CUSTNAME>Nance</CUSTNAME>
    <SHIPADDR>
      <STREET>2 Avocet Drive</STREET>
      <CITY>Redwood Shores</CITY>
      <STATE>CA</STATE>
    </SHIPADDR>
</PO>
```

En el siguiente ejemplo extraemos el nodo Docks dentro del nodo warehouse del nodo raíz del documento XML que se encuentra en la columna warehouse_spec de la tabla warehouses siempre que este no sea nulo y lo convertimos en una cadena (si no sería un XMLType):

```
SELECT warehouse_name,
   extract(warehouse_spec, '/Warehouse/Docks').getStringVal()
   AS "Number of Docks"
   FROM warehouses
   WHERE warehouse_spec IS NOT NULL;

WAREHOUSE_NAME          Number of Docks
--------------------     --------------------
Southlake, Texas        <Docks>2</Docks>
San Francisco           <Docks>1</Docks>
New Jersey                     <Docks/>
Seattle, Washington     <Docks>3</Docks>
```

El comando extract siempre devuelve el nodo en un tipo XMLType, si queremos recuperar el valor del nodo de texto de ese nodo podemos utilizar getNumberVal o getStringVal sobre el elemento XMLType devuelto. O bien podemos utilizar extractValue que tiene una sintaxis idéntica a extract pero que devuelve el valor del nodo de texto y no el elemento XMLType. Estas funciones sólo son válidas para nodos que tengan un solo y único nodo de texto.

El comando updateXML permite actualizar el valor de algunos nodos señalados del documento XML, para evitar de esa manera modificar todo el documento cuando sólo varía parte. Sus parámetros son parejas de rutas XPath y valores, donde la ruta señala el nodo a modificar y los valores sustituirán a los antiguos de ese nodo:

```
UPDATE po_xml_tab
    SET poDoc = UPDATEXML(poDoc,
        '/PO/CUSTNAME/text()', 'John');
```

El comando XMLTransform toma como parámetros dos instancias de XMLtype siendo la primera el documento origen y la segunda un documento XSLT de transformaciones XML y devuelve el documento resultante de la transformación XML. Un sinónimo de este comando es el método XMLTransform de la clase XMLType.

Es posible validar documentos XML frente a esquemas XML mediante el comando XMLisValid y el método de XMLType isSchemaValidated. Ambos devuelven verdadero (1) si el documento se valida correctamente. El siguiente ejemplo valida cada documento de la columna xmlcol de la tabla po_tab con el esquema ipo.xsd devolviendo 1 (verdadero) cuando la validación es correcta y 0 (falso) cuando no:

```
SELECT x.xmlcol.isSchemaValid(
    'http://www.example.com/schemas/ipo.xsd',
    'purchaseOrder')
    FROM po_tab x;
```

Indexar elementos XMLType

Para acelerar las consultas en los tipos XMLType podemos utilizar índices basados en función para acelerar las funciones extract o existsNode en rutas XPath definidas, como por ejemplo:

```
CREATE INDEX city_index ON po_xml_tab
    (poDoc.extract('//PONO/text()').getNumberVal());
CREATE BITMAP INDEX po_index ON po_xml_tab
    (poDoc.existsNode('//SHIPADDR'));
```

Para los documentos XMLType mapeados el índice a utilizar para acelerar las consultas con parámetros o rutas XPath es ctxsys.ctxxpath:

```
CREATE INDEX xml_idx ON xml_tab(col_xml) indextype is ctxsys.CTXXPATH;
```

Sin embargo si quisiéramos más flexibilidad y funcionalidad de un índice para los documentos XML sin esquema deberíamos utilizar Oracle Text que se utiliza en columnas tipo CLOB o VARCHAR y que en la última versión de Oracle se ha extendido para utilizar XMLType. Para crear un índice de Oracle Text en una columna XMLType poDoc en una tabla po_xml_tab:

```
CREATE INDEX po_text_index ON
    po_xml_tab(poDoc) INDEXTYPE IS ctxsys.context;
```

El tipo de índice utilizado es ctxsys.context que es apropiado para predicados y consultas de Oracle Text como CONTAINS, SCORE, INPATH, HASPATH. Si queremos acelerar las consultas con las funciones extract y existsNode debemos utilizar el índice ctxsys.ctxxpath.

Cuando trabajemos con las opciones de secciones de Oracle Text debemos definir cómo se crean estas secciones. Las secciones son partes del documento a las que referenciar en las búsquedas, como son por ejemplo los nodos de cada documento XML. Las secciones permitir restringir las búsquedas a ciertas partes del documento.

El predicado de búsqueda de Oracle Text es CONTAINS que toma 2 parámetros: el primero es la columna donde efectuar la búsqueda y el segundo es el predicado a cumplir por el documento. Por cada fila devuelta CONTAINS devuelve el porcentaje entre 0 y 100 de relevancia del documento.

```
SELECT id FROM my_table
    WHERE CONTAINS (my_column, 'receipts') > 0
```

Si el documento esta dividido por secciones (en el modo de secciones de Oracle Text AUTO_SECTION_GROUP las secciones se crean automáticamente con las etiquetas (y se les referencia: *etiqueta*) y atributos XML (y se les referencia: *etiqueta@atributo*)) nos es posible restringir la búsqueda a una sección en particular. En el siguiente ejemplo se busca receipts en el atributo title de la etiqueta report:

```
SELECT id FROM my_table
    WHERE CONTAINS (my_column,
       'receipts WITHIN report@title') > 0
```

El operador INPATH es muy parecido a WITHIN pero ha sido incluido para dar soporte a documentos XML. Su sintaxis es igual pero la ruta de sección ya no es del tipo *etiqueta* o *etiqueta@atributo*, si no que se utiliza XPath para referirse a cada nodo del documento. El operador HASPATH busca que documentos tienen la ruta especificada en XPath como parámetro.

```
SELECT id FROM library_catalog
    WHERE CONTAINS(text,'box INPATH(order/item)') > 0;
SELECT id FROM library_catalog
    WHERE CONTAINS(text,'HASPATH(order/item)') > 0;
```

SQLX, generar XML de los datos relacionales

Al igual que en el modelado objeto-relacional, en el que no es necesario convertir los datos en el modelo plano relacional al modelo objeto-relacional para trabajar con ellos en este último modelo, es posible mediante comandos y vistas representar datos del modelo relacional u objeto-relacional como documentos XML sin necesidad de modificarlos.

Oracle soporta cinco comandos del estándar SQLX (SQL to XML, SQL/XML) para la representación de datos relacionales con XML: XMLElement, XMLForest, XMLConcat, XMLAttributes y XMLAgg. También soporta XMLColAttVal como comando SQLX propio, pero aún no aceptado en el estándar. Estos comandos permiten representar datos como un documento XML definiendo nosotros la estructura de ese documento.

Oracle, además, soporta las funciones SYS_XMLGEN. SYS_XMLAGG y XMLSEQUENCE y XMLFormat con el mismo propósito que las anteriores pero sin ser parte del estándar SQLX o de su propuesta.

Nos es posible también crear vistas del tipo XMLType para representar tablas y vistas relacionales como documentos XML de forma transparente para la consulta, como si de una consulta a un XMLType se tratase.

Funciones SQLX

XMLElement es una función que devuelve un tipo XMLType dados como parámetros el nombre del elemento XML, una serie de atributos y el contenido del nodo. El XMLType devuelto es un nodo con el nombre del primer parámetro, los atributos del segundo y el contenido de los últimos parámetros. El contenido puede ser un valor o un nuevo elemento XMLType para poder formar la estructura anidada de los documentos XML.

Los atributos se definen mediante la función XMLAttributes que toman como método el listado de atributos a asignar al elemento XML. Si no se especifica la cláusula AS en cada atributo se toma como nombre de atributo el inferido de la estructura relacional, si se utiliza AS se toma el indicado.

Se puede ver su funcionalidad en el siguiente ejemplo:

```
SELECT XMLELEMENT("Emp",
    XMLATTRIBUTES (e.id,e.birth AS "birthDate"),
    XMLELEMENT("name", e.fname ||' '|| e.lname),
    XMLELEMENT ( "hiredate", e.hire)) AS "result"
    FROM employees e
    WHERE employee_id > 200 ;

result
-----------------
<Emp ID="1001" birthDate="1951-07-13"/>
   <name>John Smith</name>
   <hiredate>2000-05-24</hiredate>
</Emp>
<Emp ID="1206" birthDate="1951-03-09"/>
   <name>Mary Martin</name>
   <hiredate>1996-02-01</hiredate>
</Emp>
```

La función XMLForest crea un árbol XML de los parámetros que toma. Un árbol XML son nodos situados a la misma altura, es decir nodos que partirían del mismo nodo raíz, salvo que no definimos este nodo raíz. Cuando el parámetro se acompaña de la cláusula AS se utiliza éste como nombre de elemento XML, cuando no se infiere de la estructura de los datos. El siguiente es un ejemplo de XMLElement para crear el nodo raíz y de XMLForest para crear los elementos de esto nodo:

```
SELECT XMLELEMENT("Emp",
    XMLATTRIBUTES ( e.fname ||' '|| e.lname AS "name" ),
    XMLForest ( e.hire, e.dept AS "department")) AS "result"
    FROM employees e;

result
-----------------
<Emp name="John Smith">
   <HIRE>2000-05-24</HIRE>
   <department>Accounting</department>
</Emp>
<Emp name="Mary Martin">
   <HIRE>1996-02-01</HIRE>
   <department>Shipping</department>
</Emp>
```

La función XMLConcat, dados como parámetros una secuencia de elementos XMLType o datos tipo XMLType, los concatena uno tras otro en el orden en que aparecen como parámetros. Mientras que en XMLForest los parámetros son datos relacionales, en XMLConcat son tipos XMLType:

```
SELECT XMLConcat (
    XMLElement ("first", e.fname),
    XMLElement ("last", e.lname))
    AS "result"
    FROM employees e ;

result
-----------------
<first>Mary</first>
<last>Martin</last>

<first>John</first>
<last>Smith</last>
```

La función XMLAgg es una función de agregado que produce un bosque de elementos XML dada una colección de elementos. Se usa normalmente con consultas con cláusulas de agrupación como GROUP BY, como se puede ver en el siguiente ejemplo:

```
SELECT XMLELEMENT( "Department",
    XMLATTRIBUTES ( e.dept AS "name" ),
    XMLAGG (XMLELEMENT ("emp", e.lname))) AS "dept_list"
    FROM employees e
    GROUP BY dept;

result
-----------------
<Department name="Accounting">
   <emp>Yates</emp>
   <emp>Smith</emp>
</Department>
<Department name="Shipping">
   <emp>Oppenheimer</emp>
   <emp>Martin</emp>
</Department>
```

La función XMLColAttVal crea un árbol de XML donde cada elemento es de tipo *column* y posee un atributo tipo *name* con el nombre del elemento, especificado por AS en los parámetros o inferido de los datos. El siguiente es un ejemplo de uso de XMLColAttVal:

```
SELECT XMLELEMENT("Emp",XMLATTRIBUTES(e.fname ||' '||e.lname AS "name" ),
    XMLCOLATTVAL ( e.hire, e.dept AS "department")) AS "result"
    FROM employees e;

result
-----------------
<Emp name="John Smith">
    <column name="HIRE">2000-05-24</column>
    <column name="department">Accounting</column>
</Emp>
<Emp name="Mary Martin">
    <column name="HIRE">1996-02-01</column>
    <column name="department">Shipping</column>
</Emp>
<Emp name="Samantha Stevens">
    <column name="HIRE">1992-11-15</column>
    <column name="department">Standards</column>
</Emp>
```

SYS_XMLGEN. SYS_XMLAGG y XMLSEQUENCE y XMLFormat

La función SYS_XMLAGG engloba todos los documentos XML o fragmentos de una expresión en un solo documento XML. La etiqueta que engloba es por defecto ROWSET, pero puede ser definida con XMLFormat. El ejemplo de SYS_XMLAGG se muestra junto con el ejemplo de XMLSEQUENCE.

XMLSEQUENCE devuelve una secuencia (*array* variable) de XMLType dado un XMLType. Es decir, toma los nodos hijo directos del XMLType y devuelve un nodo XMLType por cada uno de ellos en un objeto XMLSequenceType. Supongamos el siguiente documento XML:

```
<EMPLOYEES>
  <EMP>
    <EMPNO>112</EMPNO>
    <EMPNAME>Joe</EMPNAME>
    <SALARY>50000</SALARY>
  </EMP>
  <EMP>
    <EMPNO>217</EMPNO>
    <EMPNAME>Jane</EMPNAME>
    <SALARY>60000</SALARY>
  </EMP>
  <EMP>
    <EMPNO>412</EMPNO>7
    <EMPNAME>Jack</EMPNAME>
    <SALARY>40000</SALARY>
  </EMP>
</EMPLOYEES>
```

El siguiente comando tendría el resultado:

```
SELECT SYS_XMLAGG(value(e), xmlformat('EMPLOYEES'))
    FROM TABLE(XMLSequence(Extract(doc, '/EMPLOYEES/EMP'))) e
    WHERE EXTRACTVALUE(value(e), '/EMP/SALARY') >= 50000;

<EMPLOYEES>
  <EMP>
    <EMPNO>112</EMPNO>
    <EMPNAME>Joe</EMPNAME>
    <SALARY>50000</SALARY>
  </EMP>
  <EMP>
    <EMPNO>217</EMPNO>
    <EMPNAME>Jane</EMPNAME>
    <SALARY>60000</SALARY>
  </EMP>
</EMPLOYEES>
```

La función SYS_XMLGEN toma un tipo nativo, un tipo abstracto o un tipo XMLType y genera con el un documento XML. Si es un tipo nativo forma una etiqueta con el valor dentro, si es un tipo abstracto mapea los atributos del tipo abstracto a un documento XML y si es un XMLType engloba a este elemento en otro elemento de nombre por defecto ROW. Es posible indicar el nombre de la etiqueta principal del documento XML generado mediante la función XMLFormat. El siguiente es un ejemplo de uso:

```
SELECT SYS_XMLGEN(email).getStringVal()
    FROM employees
    WHERE employee_id = 205;

SYS_XMLGEN(EMAIL).GETSTRINGVAL()
--------------------------------------------------------------
<EMAIL>SHIGGENS</EMAIL>
```

El objeto XMLFormat es un parámetro de SYS_XMLGEN y SYS_XMLAGG. Este objeto define las características del documento generado por estas dos funciones mediante sus atributos. Si queremos cambiar el formato del documento XML generado tan solo tendremos que darle el valor adecuado al correspondiente atributo de XMLFormat. Las más importantes son:

- enclTag. Es el nombre de la etiqueta que engloba el documento.
- schemaType. Indica si el documento está validado por un esquema o no, sus valores válidos son NO_SCHEMA y USE_GIVEN_SCHEMA.
- schemaName. Nombre del esquema.
- targetNameSpace. Namespace del documento.
- dburl. URL donde encontrar la definición de los esquemas, si no se declara se consideran relativos al documento.

Vistas XMLType

Las vistas XMLType nos permiten tomar elementos relacionales u objeto-relacionales de la base de datos y sin modificar ni los datos ni su estructura poder mostrarlos como si documentos XML fuesen.

Mediante la creación de vistas habitual creamos una vista indicando que es de tipo XMLType (OF XMLTYPE), la cláusula OBJECT ID indica que empno será el identificador único de cada elemento y que el tipo XMLType se almacenará en la columna sys_nc_rowinfo$. Y creamos la vista mediante las funciones SQLX y de Oracle vistas en el apartado anterior:

```
CREATE TABLE employees (
    empno number(4),
    fname varchar2(20),
    lname varchar2(20),
    hire date,
    salary number(6));
CREATE OR REPLACE VIEW Emp_view OF XMLTYPE
    WITH OBJECT ID (EXTRACT(
        sys_nc_rowinfo$,'/Emp/@empno').getnumberval())
    AS SELECT XMLELEMENT(
        "Emp",
        XMLAttributes(empno),
        XMLForest(e.fname ||' '|| e.lname AS "name",
        e.hire AS "hiredate")) AS "result"
    FROM employees e
    WHERE salary > 20000;
```

Si ahora insertamos datos en la tabla relacional y los consultamos mediante la vista:

```
INSERT INTO employees
    VALUES (2100, 'John', 'Smith', Date'2000-05-24', 30000);
INSERT INTO employees
    VALUES (2200, 'Mary', 'Martin', Date'1996-02-01', 30000);

SELECT * FROM Emp_view;

SYS_NC_ROWINFO$
------------------------------------------------------------
<Emp empno="2100">
  <name>John Smith</name>
  <hiredate>2000-05-24</hiredate>
</Emp>
<Emp empno="2200">
  <name>Mary Martin</name>
  <hiredate>1996-02-01</hiredate>
</Emp>
```

También es posible crear vistas XMLType mapeando los datos relacionales mediante un esquema y no con el comando SELECT de la definición de la vista. El esquema define el mapeado de cada elemento a la columna de datos mediante el atributo xdb:SQLName en el elemento del esquema, de tal manera que el elemento contendrá el valor de la columna indicada en ese atributo. A continuación podemos crear la vista de la siguiente manera:

```
CREATE OR REPLACE TYPE dept_t AS OBJECT (
    DEPTNO NUMBER(2),
    DNAME VARCHAR2(14),
    LOC VARCHAR2(13));
CREATE OR REPLACE TYPE emp_t AS OBJECT (
    EMPNO NUMBER(4),
    ENAME VARCHAR2(10),
    JOB VARCHAR2(9),
    MGR NUMBER(4),
    HIREDATE DATE,
    SAL NUMBER(7,2),
    COMM NUMBER(7,2),
    DEPT DEPT_T);
```

```
CREATE OR REPLACE VIEW emp_xml OF XMLTYPE
   XMLSCHEMA "http://www.oracle.com/emp.xsd"
   ELEMENT "Employee"
   WITH OBJECT ID (
       ExtractValue(sys_nc_rowinfo$, '/Employee/EmployeeId'))
   AS SELECT emp_t(
       e.empno, e.ename, e.job, e.mgr, e.hiredate,
       e.sal, e.comm, dept_t(d.deptno, d.dname, d.loc))
   FROM emp e, dept d
   WHERE e.deptno = d.deptno;
```

El esquema es el siguiente y en el podemos comprobar como cada columna de las tablas emp y dept es mapeada a un elemento del documento XML. La estructura del documento XML también es definida por el esquema:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.oracle.com/emp.xsd"
  version="1.0"
  xmlns:xdb="http://xmlns.oracle.com/xdb"
  elementFormDefault="qualified">
<element name = "Employee" xdb:SQLType="EMP_T" xdb:SQLSchema="SCOTT">
  <complexType>
    <sequence>
      <element name = "EmployeeId" type = "positiveInteger"
           xdb:SQLName="EMPNO" xdb:SQLType="NUMBER"/>
      <element name = "Name" type = "string" xdb:SQLName="ENAME"
           xdb:SQLType="VARCHAR2"/>
      <element name = "Job" type = "string" xdb:SQLName="JOB"
           xdb:SQLType="VARCHAR2"/>
      <element name = "Manager" type = "positiveInteger" xdb:SQLName="MGR"
           xdb:SQLType="NUMBER"/>
      <element name = "HireDate" type = "date" xdb:SQLName="HIREDATE"
           xdb:SQLType="DATE"/>
      <element name = "Salary" type = "positiveInteger" xdb:SQLName="SAL"
           xdb:SQLType="NUMBER"/>
      <element name = "Commission" type = "positiveInteger"
           xdb:SQLName="COMM" xdb:SQLType="NUMBER"/>
      <element name = "Dept" xdb:SQLName="DEPT" xdb:SQLType="DEPT_T"
           xdb:SQLSchema="SCOTT">
        <complexType>
          <sequence>
            <element name = "DeptNo" type = "positiveInteger"
                xdb:SQLName="DEPTNO" xdb:SQLType="NUMBER"/>
            <element name = "DeptName" type = "string" xdb:SQLName="DNAME"
                xdb:SQLType="VARCHAR2"/>
            <element name = "Location" type = "string" xdb:SQLName="LOC"
                xdb:SQLType="VARCHAR2"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
</schema>
```

La consulta tendría como respuesta el siguiente documento XML:

```
<Employee xmlns="http://www.oracle.com/emp.xsd"
    xmlns:xsi="http://www.oracle.com/emp.xsd http://www.oracle.com/emp.xsd">
   <EmployeeId>2100</EmployeeId>
   <Name>John</Name>
   <Manager>Mary</Manager>
   <Hiredate>12-Jan-01</Hiredate>
   <Salary>123003</Salary>
   <Dept>
      <Deptno>2000</Deptno>
      <DeptName>Sports</DeptName>
      <Location>San Francisco</Location>
   </Dept>
</Employee>
```

Bibliografía y documentación

- Oracle 9i XML Database Developer's Guide - Oracle XML DB, Release 2 (9.2). Marzo 2002.
- Oracle 9i: The complete reference. Kevin Loney, George Koch. McGraw-Hill/Osborne.
- SQL/XML is Making Good Progress. Andrew Eisenberg, Jim Melton. www.sqlx.org.
- Oracle 9i Application Developer's Guide - Object-Relational Features, Release 2 (9.2). Marzo 2002.
- XPath tutorial y XLab en www.zvon.org.
- (ISO-ANSI Working Draft) XML-Related Specifications (SQL/XML). Jim Melton. www.sqlx.org.
- Building Oracle XML Applications. Steve Muench. O'Reilly. Primera edición de September 2000.
- Oracle 9i XML Developer's Kits Guide – XDK, Release 2 (9.2). Marzo 2002.
- Oracle 9i Java Stored Procedures Developer's Guide, Release 2 (9.2). Marzo 2002.
- Oracle9i Java Developer's Guide, Release 2 (9.2). Marzo 2002.