close window

e-Newsletter Exclusive

Print

Using XML With DB2 for i

How to Use XML Data Type With DB2 for IBM i

March 2012 | by Nick Lawrence

In Part 1 of this article, I provided an overview of the extensible markup language (XML) and shared how XML and relational data can form a partnership. Having a native XML type in DB2 can provide an integrated solution for working with both XML and relational data in the same database.

In Part 2, I want to share the practical steps for how to Parse, Insert, Validate and Serialize XML using DB2 for IBM i, a new enhancement in 7.1.

Storing XML in a DB2 Column

Let’s look at an example of how to put an XML value in a DB2 column. Consider first the case where I simply want to store Order XML documents in an SQL column. This is ideal when I need to retrieve the entire XML document exactly as it was received, perhaps for auditing purposes. It would also be a necessity if shredding the XML document into a traditional relational format proved impractical.

Step 1 is to create a table that has a column with the XML type:

```
create table order_records (
      order_number bigint generated always as identity
                   (start with 1 increment by 1 nocycle),
      order_doc xml ,
      primary key (order_number));
```

I created a primary key using an identity column. The XML data type can’t be compared to any data type, including XML, therefore it can’t be a primary key. Armed with an understanding of XML documents, it’s easy to see why. A comparison between two XML values that may contain different structures and data types isn’t something that can be universally defined.

Step 2 is to create an XML value and insert it into the table. It’s possible to do this with a single insert, but I used a procedure and broke this process up into multiple steps to illustrate a few important details. I’ll demonstrate the easier solution later.

```
1 create or replace procedure load_xml_from_blob(in_blob blob)
2 language sql
3 begin
4 declare xmlvalue xml;
```

```
5 set xmlvalue = xmlparse(document in_blob);
6 insert into order_records (order_doc) values (xmlvalue);
7 end;
```

The serialized XML document that is the input to the procedure is represented as character data. Therefore, on line 1, the procedure accepts a binary large object (BLOB). Most developers would naturally want to use a character large object (CLOB) for this, but this approach can be problematic.

Serialized XML documents can be represented in many different character sets, and typically a processing instruction exists at the beginning of the document that tells the XML parser which encoding the document is using. In my example, the document encoding is declared as UTF-8. If I were to declare my input parameter as CLOB with a character set of 37 (EBCDIC English), then DB2 would of course convert the UTF-8 data to EBCDIC.

Unfortunately, the encoding declaration in the serialized document still says the document is UTF-8; the result is that the serialized XML document can no longer be correctly parsed into an XML model.

It is best practice, therefore, to either use a BLOB to avoid character set conversion, or keep the data in a single character set that's guaranteed to match the document's encoding declaration.

Line 5 creates an XML value. DB2 for i doesn't allow casting any data type, including BLOB and CLOB, to an XML type. The concept of casting an SQL character type to XML is ambiguous. DB2 can't know if the cast should build an XML document or construct some part of an XML document (such as a text node).

The XMLPARSE built-in function constructs a well-formed XML document from the serialized character data. The construction verifies that the relationships and data within the document are syntactically legal for XML. Keep in mind that XMLPARSE doesn't do schema validation, which is a step that can be performed after the XML document is constructed.

The XML data type contains a complete representation of the XML document's data and all relationships defined by the structure of the document. Because the XML type is native to DB2, the internal representation of XML data is not of concern for the database application, and the implementation details can be left up to DB2 for i. DB2 for i, in general, keeps XML data in a representation that can be very efficiently serialized for transmission. Other implementations often have difficultly achieving high performance goals for serialization, which is an important use case for DB2 for i customers.

Line 6 is a trivial insert of the xml value into the table.

Of course, it's easy to do the parse and insert in a single SQL statement.

```
create or replace procedure load_xml_from_blob(in_blob blob)
language sql
begin
 insert into order_records (order_doc) values (in_blob);
end;
```

In this example, I've even removed the XMLPARSE from the insert. This may seem like I'm inserting a

BLOB into an XML column – but this isn't the case. DB2 for i knows that an XML column must contain a well-formed XML document, and so it will do the XMLPARSE implicitly for me.

The same idea applies for parameter markers. Applications are going to be providing DB2 with serialized, well-formed XML documents – since those are all that are suitable for transmission. Because DB2 knows this, it can allow casting a parameter marker to XML. It's best to think of this process as an implicit parse, rather than a cast from character to XML.

This insert shows how to load an xml document in an IFS stream file called /home/ntl/VehicleOrder.xml and insert it into the table:

```
insert into order_records (order_doc) values
(get_xml_file('/home/ntl/VehicleOrder.xml'));
```

The get_xml_file function loads an IFS stream file's data, converts the data to UTF-8, adds an appropriate encoding declaration and returns the whole thing as a BLOB. Since the column is an XML type, DB2 is smart enough to XMLPARSE the BLOB into the XML data type for us before inserting the value.

It may seem strange for this function to return a BLOB, however certain commonly used procedures, such as schema registration (XSR_REGISTER), expect the XML schema (which is itself written in XML) to be provided as a BLOB. The BLOB return type offers a performance improvement in those cases.

Schema Validation

In many applications, the XML document being provided to DB2 can already be assumed to be valid according to an XML schema, and the performance overhead of validation can be avoided.

In development environments, or in situations where the source of an XML document is not trusted, formal validation might be required. Another scenario where validation is used is to complete an XML document by adding default values for elements and attributes that don't exist, but which have default values defined by the schema.

Before an XML document can be validated against an XML schema, the schema must be registered with DB2. XML documents and schemas don't reference a schema document using SQL rules, therefore DB2 needs to be made aware of the XML schema location hint and target namespace during the registration process.

Since schemas support reusable complex types, it's common to reuse types defined in other schema documents. When I created my XML schema for the example in Part 1, I created three schema document files, which I copied into an IFS directory (/home/ntl/vehicle_orders) on my IBM i.

Schema Location (File Name)	Target Namespace	Description
Customer.xsd	http://www.example.ibm.com/Customer	Defines the customer_t data type that will be used to define the Customer element.

		The CustomerName_t and GivenNames_t are also defined in this schema document.
Vehicle.xsd	http://www.example.ibm.com/Vehicle	Defines the Vehicle_t data type and sub-types that will be used to define the Vehicle element
VehicleOrder.xsd	http://www.example.ibm.com/VehicleOrder	Defines the structure of the Order Element, this needs to reference Customer.xsd & Vehicle.xsd to define the structure of the nested Vehicle and Customer elements, respectively.

In my example, each schema has its own target namespace. Looking at the sample document presented in Part 1, you can see that the elements are associated with the target namespace of the schema that defines their structure. Namespaces are therefore used to avoid name collisions when elements are defined in multiple schemas and to help the validate process determine which schema should be used for validation.

Schema location could have been any URI that tells validate how to find the schema document. In my example, it happens to be the file name for the schema file on my PC. When validating an element, the validation process will determine which schema to use by looking for a schema location that's assigned to a namespace that matches the element's namespace.

For example, in the sample document, element “vo:Order” is defined in the namespace “http://www.example.ibm.com/VehicleOrder”. (The reason for this is that the ‘vo’ prefix has been mapped to this namespace.) The xsi:schemaLocation attribute in the XML document says that elements in “http://www.example.ibm.com/VehicleOrder” should be validated using a schema that has a location hint of VehicleOrder.xsd. Schema validation will look for a matching definition of the Order element using the schema located by VehicleOrder.xsd. The trick is to register the schema using this location hint and target namespace, so that XMLVAIDATE can find the correct schema when it validates the document.

The first step for providing the XML schema to DB2 is to register VehicleOrder.xsd in the schema repository. The XSR_REGISTER stored procedure can be used to create an XSR SQL object to store XML schema documents and insert the initial schema document.

```
call xsr_register('XMLEXAMPLE',
                  'VEHICLE_ORDER_SCHEMA',
                  'VehicleOrder.xsd’,
      get_xml_file('/home/ntl/vehicle_orders/VehicleOrder.xsd'),
                  NULL);
```

Here, we specified an SQL name (XMLEXAMPLE.VEHICLE_ORDER_SCHEMA), a schema location and the schema document. DB2 creates the XSR object using the SQL name. The schema document is inserted into the XSR object using VehicleOrder.xsd as the location hint. The target namespace for the schema is defined in the schema document.

The XSR can't be used for validation until registration is complete. In this example, we still have two more schemas to add to the repository before we can complete the process.

The XSR_ADDSCHEMADOC Stored procedure can be used to add additional schema documents to the schema repository.

```
call xsr_addschemadoc('XMLEXAMPLE',
                      'VEHICLE_ORDER_SCHEMA',
                      'Vehicle.xsd',
                get_xml_file('/home/ntl/vehicle_orders/Vehicle.xsd'),
                      NULL);

call xsr_addschemadoc('XMLEXAMPLE',
                      'VEHICLE_ORDER_SCHEMA',
                      'Customer.xsd',
               get_xml_file('/home/ntl/vehicle_orders/Customer.xsd'),
                      NULL);
```

The parameters work the same way as the xsr_register procedure.

After all Schema documents have been added to the XSR, the XSR_COMPLETE stored procedure can be used to complete the registration process.

```
call xsr_complete('XMLEXAMPLE', 'VEHICLE_ORDER_SCHEMA', NULL, 0);
```

DB2 will now compile the schema documents into a binary format so that validation will perform better. If any errors or missing documents are discovered, an error will be signaled at this point.

The schema is now available for use in validating XML documents within DB2 using the XMLVALIDATE function.

In this example, the XML document defines VehicleOrder.xsd as the location where validate should look for the schema of anything defined in the "http://www.example.ibm.com/VehicleOrder namespace". XMLVALIDATE will determine this schema location from the XML document, search the registered schemas to find the matching schema, and use the schema to perform the validation, prior to inserting the value into a table.

```
Insert into order_records (order_doc) values(
xmlvalidate(
      xmlparse(document
            get_xml_file('/home/ntl/vehicle_orders/VehicleOrder.xml'))
));
```

Several other options can override which schema is used for validation and control how an XML document is validated, and I recommend reviewing the Info Center documentation for this function to get a more detailed description. See the References at the end of the article for more.

Serializing XML to Character

To transmit an XML document, we need to serialize it to a character or binary data type. XML isn't a character or binary type, and casting from XML to another type isn't allowed. The reason that the cast isn't allowed is that it's ambiguous whether such a cast should result in the serialized XML document or the "value" of the document. The "string value" of an XML document has a special meaning and refers to all of the child text values concatenated together.

Serializing the document is easily accomplished with the XMLSERIALIZE function.

```
select order_number,
xmlserialize    (
                    order_doc as clob(1M) ccsid 1208
                    including xmldeclaration
                )
from order_records;
```

The XML document is serialized to a large character object (CLOB) in UTF-8. The encoding declaration is added asserting that the data is UTF-8. Take care after serialization; if we cast the CLOB to some other CCSID the encoding declaration within the document will no longer be correct. Once serialized, DB2 has no idea this character data is really a serialized XML document, and it won't maintain the encoding declaration.

In certain scenarios, an implicit XMLSERIALIZE will be performed. For example, if an XML document is exchanged with an application's string or binary type, DB2 knows that the application is most likely expecting a serialized document and will serialize the XML value to the target type implicitly.

Helpful Capabilities

Now you should have a better understanding of XML and its integration with DB2 for i. Those developers interested in working with XML data will find the new capabilities very helpful. In this article, I've also mentioned several other XML functions that are worth looking into. Find a few of them in the "Other XML Operations" sidebar.