Convert XML into relational data

In this tutorial we will show you some methods of converting XML documents into rows in relational tables.

Matthias Nicola, Pav Kumar-Chatterjee

Network Management - In this tutorial we will show you some methods of converting XML documents into rows in relational tables, the work is still known as shredded or split documents. XML data.

The method of converting XML documents into rows in relational data tables is known as *shred* or *decomposing* XML documents. One of the main reasons for *shred* is that existing SQL applications still need access to data in relational format. For example, legacy applications, packaged business applications, or reporting software do not always work with XML. So sometimes you will find it quite useful to *shred* all or some of the data values ??of a specified XML document into columns and rows in relational data tables.

This tutorial will introduce you to:

- The advantages and disadvantages of shredding and other shredding methods.
- How to shred XML data into relational tables using the INSERT statement contains the XMLTABLE function.
- How to use XML Schema annotations to map and shred XML documents into relational tables.

Advantages and disadvantages of shredding

The concept of shredding is illustrated in Figure 1. In this example, XML documents with *customer name* , *address* , and *phone information* are mapped into two relational tables. Documents may contain multiple phone components because the relationship between the customer and their phone number is a 1-n relationship. Therefore, the phone numbers will be shredded into a separate table. Each iterative component, such as phone, will lead to an additional table in the relational target schema. Suppose customer information can contain multiple email addresses, multiple accounts, recent orders list, multiple products in each order and other repeat items. Then the number of tables required in the relational target schema can increase very quickly. However shredding XML into a large number of tables can lead to complex business logic objects and make application development difficult or error-prone. Querying data has been shrunk or *reassembling* the original documents that may require complex *joins* .

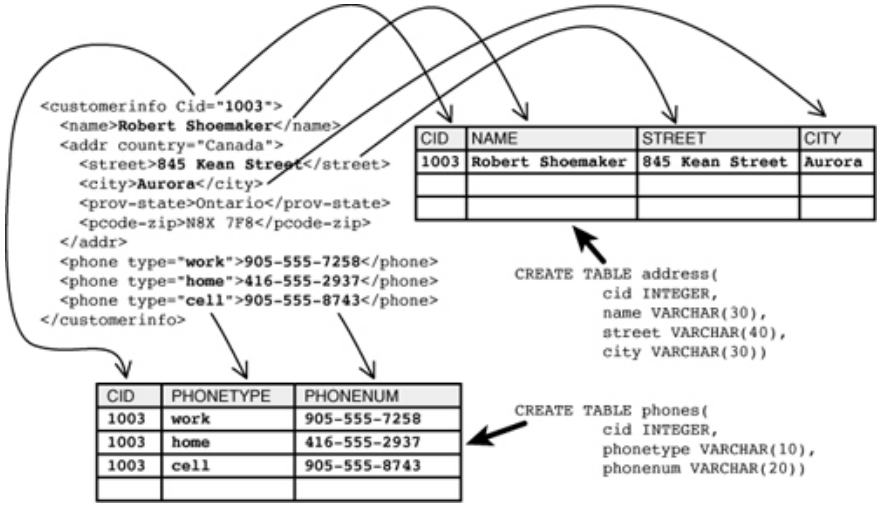

Figure 1: Shredding (*shred*) an XML document

Depending on the complexity and variability, purpose of XML documents, shredding (*shred*) may or may not be a mandatory option. Table 1 summarizes the advantages and disadvantages of shredding (*shred*) XML data into relational tables.

Is the appropriate choice when .

An inappropriate choice when .

- The specified XML data is providing for an existing relational database.
- Your XML data is complex, stacked, and difficult to map into a relational schema.
- XML documents do not represent business logic objects that need to be maintained.
- Mapping your XML format into a relational schema leads to a large number of tables.
- Your main purpose is to allow existing relational applications to access XML data
- Your XML Schema is volatile
- You agree with your relational schema and prefer to use it if possible.
- Your goal is to manage XML documents as intact enterprise objects.
- The structure of your XML data is easily mapped into relational data tables.
- You often need to rebuild documents that are shredded or partially in them.
- Your XML format is relatively stable and has little change.
- Use XML data in a high-speed database for your application.
- You rarely need to rebuild shredded documents.

- Querying or upgrading data with SQL is more important than inserting.

Table 1. When shredding is a good choice

In many XML application scenarios, the structure and usage of XML data does not adapt to shredding. The reason for this is because DB2 supports XML columns that can allow you to index and query data without the need for conversion. Sometimes you will see your application requests that can best respond to the *partial shredding* or *hybrid XML storage* . Here:

- *Partial shredding* means that only a small set of components or attributes from each incoming XML document are shredded into relational tables. This is quite useful if the relational application does not require all data values ??from each XML document. In cases where the whole shredding of each document is difficult and requires a complex relational schema, *partial shredding* can simplify mapping to a relational schema significantly. .
- *Hybrid XML storage* means while inserting an XML document into an XML column, the selected element or extracted attribute values ??and the reserved component are stored in relational columns.

If you want to shred XML documents, in whole or in part, DB2 provides you with a rich set of features to implement some or all of the following:

- Perform data value transformations before inserting into relational columns.
- Shredding the same attribute or element value into multiple columns of the same or different tables.
- Shred many different elements or attributes into the same column in a table.
- Specify conditions that govern whether certain components are or are not chopped. For example, shredding the address of a customer document only when the country is Canada.
- Validate XML documents with an XML Schema during shredding.
- Save the full XML document with hashed data.

DB2 9 for z / OS and DB2 9.x for Linux, UNIX, and Windows support two shredding methods:

- SQL *INSERT* statements use the *XMLTABLE* function. This function will navigate to an input document and generate one or more relational rows used to insert the relational table.
- Separate with an annotated XML Schema. Because XML Schema defines the structure of XML documents, annotations can be added to the schema to define how the components and attributes are mapped to relational tables.

Table 2 and Table 3 will introduce the advantages and disadvantages of *XMLTABLE* method and annotated schema method.

Advantages of XMLTABLE method

Disadvantages of XMLTABLE method

- Allows you to shred data even without XML Schema.
- Each target table you want to shred into, you need an *INSERT statement*
- Do not ask you to understand the XML Schema language or understand the annotations for decomposition.
- You may have to combine multiple *INSERT statements* in a stored procedure.
- Often easier to use than annotated schemas because they are based on SQL and XPath.
- You can use familiar functions and expressions of XPath, XQuery, or SQL to extract and adjust data values.
- Often requires less workload during XML Schema development.
- The shredding process may need data from multiple relational sources and XML if needed, such as values ??from DB2 strings or lookup data from other relational tables.
- Can provide better performance than annotation schema separation method.
- There is no user interface support for executing *INSERT statements* and necessary *XMLTABLE* functions. You need to know about XPath and SQL / XML.

Table 2. XMLTABLE method

Advantages of the Disadvantage method of the method

- Mapping from XML into relational tables can be defined with a GUI in IBM Data Studio Developer.
- Do not allow shredding without XML Schema.
- If you shred complex XML data into a large number of tables, less effort is required in coding than the XMLTABLE method.
- You may have to manually copy the annotations when you start using a new version of XML Schema.
- Provide a large number of detailed diagnostic information if some of the documents are shredded.
- Although there is a user interface, you still need to know the XML Schema language.
- An annotation of an XML Schema can be complicated if a schema itself is complex.

Table 3: Methods for separating annotated schemas

Shredding with XMLTABLE function

XMLTABLE function is an SQL function that uses *XQuery* expressions to create relational rows from an XML input document. In this section we describe how to use the *XMLTABLE* function in a statement to perform shredding. We use the shredding script in Figure 1 as an example.

The first step we need to do is to create a target relational table, if not yet. With the scenario in Figure 1, the target tables are defined as below:

```
CREATE TABLE address (cid INTEGER, name VARCHAR (30),
street VARCHAR (40), city VARCHAR (30))
CREATE TABLE phones (cid INTEGER, phonetype VARCHAR (10),
phonenum VARCHAR (20))
```

Based on the definition of the target tables you can build *INSERT statements* to shred incoming XML documents. *INSERT statements* must be in the form *INSERT INTO . SELECT . FROM . XMLTABLE,* as shown in Figure 2. Each *XMLTABLE* function includes a parameter marker *("?"),* Through which An application can pass through shredded XML documents. SQL rules require parameter markers to be converted to the appropriate data type. The *SELECT* clause selects columns created by the *XMLTABLE* function to insert into the *address* and *phones* tables accordingly.

```
INSERT INTO address (cid, name, street, city)
SELECT x.custid, x.custname, x.str, x.place
FROM XMLTABLE ('$ i / customerinfo' PASSING CAST (? AS XML) AS "i"
COLUMNS
custid INTEGER PATH '@Cid',
custname VARCHAR (30) PATH 'name',
str VARCHAR (40) PATH 'addr / street',
place VARCHAR (30) PATH 'addr / city') AS x;

INSERT INTO phones (cid, phonetype, phonenum)
SELECT x.custid, x.ptype, x.number
FROM XMLTABLE ('$ i / customerinfo / phone'
PASSING CAST (? AS XML) AS "i"
COLUMNS
custid INTEGER PATH './@Cid',
number VARCHAR (15) PATH '.',
ptype VARCHAR (10) PATH './@type') AS x;
```

Figure 2: Insert the XML element and attribute values ??into relational columns

To populate the two target tables as illustrated in Figure 1, both *INSERT statements* must be executed with the same input XML document. One method implemented here is that the application plays both statements in a

transaction and binds the same XML document to parameter markers for both statements. This method works well but can be optimized, since the same XML document is sent from the client to the server and separated at the DB2 server twice, each time for an *INSERT statement* . This process can be avoided by combining both *INSERT statements* in a stored procedure. The application only creates a stored procedure call and *passes* the input document once, and does not care about the number of statements in the stored procedure.

Alternatively, *INSERT statements* in Figure 2 can read a set of input documents from XML columns. Suppose the documents are *loaded* into the XML column info in the customer table. You then need to change a line in each *INSERT statement* in Figure 2 to read the input document from the *customer* table:

```
FROM customer, XMLTABLE ('$ i / customerinfo' PASSING info AS "i"
```

Loading the input documents into the table can be quite convenient if you have to shred many documents. The *LOAD* utility will render in parallel the XML parsing process, reducing the time it takes to transfer documents to the database. When documents are stored in an XML column in a parsed format, the *XMLTABLE* function can shred documents without parsing XML.

The *INSERT statement* can be enriched with Xquery, SQL or join functions to accommodate the shredding process with specific requirements. Figure 3 gives you an example. The *SELECT* clause contains the *RTRIM* function that *removes* the blank spaces from the *x.ptype* column. The row creation expression of the *XMLTABLE* function contains attributes used to prevent shredded family numbers in the target table. The expression to create columns for phone numbers uses the *XQuery normalize-space* function, which separates the white space at the beginning and the tail and replaces each internal string of white characters with a blank character. The statement also executes a join action for the *areacodes* lookup table so that phone numbers are inserted into the *phones* table only if its area code is listed in the *areacodes* table.

```
INSERT INTO phones (cid, phonetype, phonenum)
SELECT x.custid, RTRIM (x.ptype) , x.number
FROM areacodes a ,
XMLTABLE ('$ i / customerinfo / phone [@type! = "Home"]'
PASSING CAST (? AS XML) AS "i"
COLUMNS
custid INTEGER PATH './@Cid',
number VARCHAR (15) PATH 'normalize-space (.)',
ptype VARCHAR (10) PATH './@type') AS x
WHERE SUBSTR (x.number, 1.3) = a.code;
```

Figure 3: Using functions and joins to adjust the hashing process

Hybrid XML Storage

In many situations, the complexity of the XML document structure can make shredding difficult, inefficient, and annoying. In addition to the performance disadvantage of shredding, dispersing the value of XML documents in a large number of tables can make it difficult for application developers to understand and query data. material. To improve XML insertion performance and reduce the number of tables in the database, you can save XML documents as **Hybrid XML Storage** . This method will extract the values ??or attributes of the selected XML components and store them in relational columns next to the XML document.

The example in the previous section used two tables, *address* and *phones* , as the target tables for shredding customer documents. However, you can only use one table that contains customer *cid*, *name*, and *city* values ??in relational columns and full XML documents with *phone* repeating components, and other information in a column. XML. If so, you can define the table as below:

```
CREATE TABLE hybrid (cid INTEGER NOT NULL PRIMARY KEY,

name VARCHAR (30), city VARCHAR (25), info XML)
```

Figure 4 shows the *INSERT statement* to populate this table. *XMLTABLE* function uses XML document as an input data through a parameter marker. The column definition in the *XMLTABLE* function will *generate* four columns corresponding to the definition of the hybrid target table. The row creation expression in the *XMLTABLE* function *$ i* will create a full input document. This expression is the input for column creation expressions in the *COLUMNS* clause of the *XMLTABLE* function. Distinctive, column expression *'.'* will return the full input document and create the XML doc column to insert the *info* column in the target table.

```
INSERT INTO hybrid (cid, name, city, info)

SELECT x.custid, x.custname, x.city, x.doc

FROM XMLTABLE ('$ i' PASSING CAST (? AS XML) AS "i"

COLUMNS

custid INTEGER PATH 'customerinfo / @ Cid',

custname VARCHAR (30) PATH 'customerinfo / name',

city ??VARCHAR (25) PATH 'customerinfo / addr / city',

doc XML PATH '.'  ) AS x;
```

Figure 4: Saving a hybrid XML document

Currently you will not be able to define *constraints* in BD2 to enforce integrity between columns and relational values ??in an XML document in the same row. You can, however, define event traps (*INSERT* and *UPDATE triggers*) on the table to populate the relational columns automatically whenever a document is inserted or updated.

It is useful to test *INSERT statements* in the DB2 Command Line Processor (CLP). For this purpose, you can replace parameter markers with a regular XML document as shown in Figure 5. This regular document is a string that is distinguished by parentheses and converted to a data type. XML with *XMLPARSE* function. Alternatively, you can read the input document from the file system using one of the UDPs, which is illustrated in Figure 6.

```
INSERT INTO hybrid (cid, name, city, info)
SELECT x.custid, x.custname, x.city, x.doc
FROM XMLTABLE ('$ i' PASSING
XMLPARSE (document
'
Kathy Smith

25 EastCreek
Markham
Ontario
N9C 3T6

905-555-7258
') AS "i"
COLUMNS
custid INTEGER PATH 'customerinfo / @ Cid',
custname VARCHAR (30) PATH 'customerinfo / name',
city ??VARCHAR (25) PATH 'customerinfo / addr / city',
doc XML PATH '.'  ) AS x;
```

Figure 5: Hybrid insert statement with a regular XML document

```
INSERT INTO hybrid (cid, name, city, info)
SELECT x.custid, x.custname, x.city, x.doc
FROM XMLTABLE ('$ i' PASSING
XMLPARSE (document
blobFromFile ('/ xml / mydata / cust0037.xml')) AS "i"
COLUMNS
custid INTEGER PATH 'customerinfo / @ Cid',
custname VARCHAR (30) PATH 'customerinfo / name',
```

```
city ??VARCHAR (25) PATH 'customerinfo / addr / city',
doc XML PATH '.'  ) AS x;
```

Figure 6: Hybrid insert statement with a "FromFile" UDF

The logic inserted in Figures 4 and 5 and Figure 6 is exactly the same. The only difference is how the input document is provided: through the parameter marker, as the string is usually distinguished by parentheses or through a UDP reading the document from the file system.

Relational views on XML data

You can create relational views over XML data using *XMLTABLE* expressions. This allows you to provide relational or hybrid view applications to XML data without having to save data in relational or hybrid formats. This method is useful if you want to avoid the complexity of converting a large amount of XML data into relational format. The basic *SELECT . FROM . XMLTABLE structure* used in the *INSERT statement* in the previous section can also be used in *CREATE VIEW* statements.

For an example, suppose you want to create a relational view for components of XML documents in the *customer* table to display identifiers, names, streets, and cities (*identifier, name, street).* , and *city)* of customers. Figure 7 shows the corresponding view definition plus an SQL query for the view.

```
CREATE VIEW custview (id, name, street, city)
AS
SELECT x.custid, x.custname, x.str, x.place
FROM customer,
XMLTABLE ('$ i / customerinfo' PASSING info AS "i"
COLUMNS
custid INTEGER PATH '@Cid',
custname VARCHAR (30) PATH 'name',
str VARCHAR (40) PATH 'addr / street',
place VARCHAR (30) PATH 'addr / city') AS x;

SELECT id, name FROM custview WHERE city = 'Aurora';

ID NAME
-------------------------------------
1003 Robert Shoemaker
```

```
1 record (s) ???c ch?n.
```

Figure 7: Creating a view for XML data

The query on the view in Figure 7 includes an SQL attribute for the *city* column in the view. The values ??in the *city* column come from an XML element in the XML column below. You can speed up this query by creating an XML index on / customerinfo / addr / city for the *info* column of the *customer* table. DB2 9 for z / OS and DB2 9.7 for Linux, UNIX, and Windows can convert the relational attribute *city ??=* ' *Aurora* ' to an XML attribute in the XML column below to be able to use the *XML index* . However, this is not possible in DB2 9.1 and DB2 9.5 for Linux, UNIX, and Windows. In the previous sections of BD2, group the XML column in the view definition and write the search condition as an XML attribute, see in the query below. Otherwise it will not be able to use the *XML index* .

```
SELECT id, name

FROM custview

WHERE XMLEXISTS ('$ INFO / customerinfo / addr [city = "Aurora"]')
```

(*Also*)