Table of Contents

Using DB2 pureXML and ODF Spreadsheets

Abstract

ODF (Open Document Format) is a file format for office documents. At a high level, the ODF specification requires five mandatory XML documents and other optional items. This article will describe how DB2 pureXML could be used to handle ODF spreadsheet documents. Other ODF documents such as word documents, presentations, formulas, etc. could also be handled by DB2 because the ODF specification refers mainly to an archive of XML documents. However, for the sake of this article, we are considering only spreadsheets because it is easier to co-relate the rows and columns of the spreadsheet document and that of a DB2 table. This article can even be further extended into an on-line document editing software with .ods as file format and DB2 pureXML as the database.

Introduction

The ODF specification and DB2 pureXML are introduced below.

The ODF specification

The Open Document Format specification was originally developed by Sun [http://www.sun.com/] and the standard was developed by OASIS Open Document Format for Office Applications (Open Document)

TC – OASIS ODF TC. This standard is based on XML format originally created and implemented by OpenOffice.org office suite. In addition to being a free and open OASIS standard, it is published (in one of its version 1.0 manifestations) as an ISO/IEC international standard, ISO/IEC 26300:2006 Open Document Format for Office Applications (OpenDocument) v1.0 [http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43485] .

DB2 pureXML

DB2 pureXML [http://www-01.ibm.com/software/data/db2/xml/] is IBM software for management of XML data. It eliminates much of the work typically involved in the management of XML data. From validation of XML documents, to generation of documents and support for querying XML data in the form of XQuery [http://www.w3.org/TR/xquery/] and traditional SQL, DB2 pureXML is a complete solution to manage XML data in enterprise-scale databases.

The ODF document

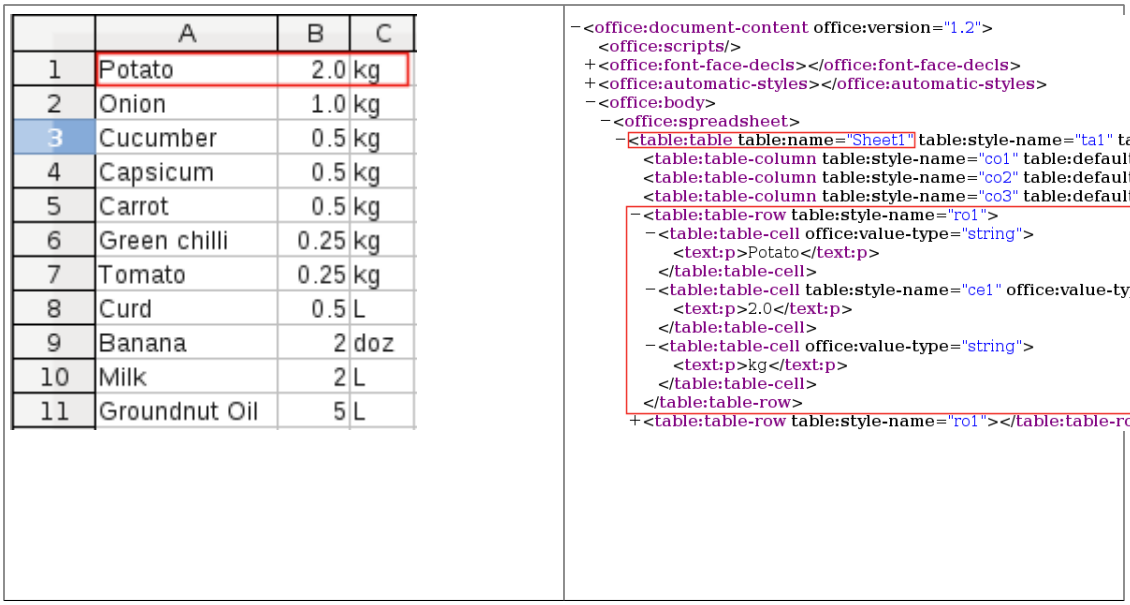
An ODF document is a zipped file of five mandatory XML files and other optional components. This can be seen by running your favorite un-zip utility to list the components. Here is the result after running the unzip command on an .ods file from command line to list the files. The XML files are highlighted below.

Here is a short description of XML files as seen above.

File name	Usage	Location in zipped file
manifest.xml	Information about the files contained in the package.	/META-INF
styles.xml	Styles used in the document content and automatic styles used in the styles themselves.	/ (root)
settings.xml	Application-specific settings, such as the window size or printer information.	/ (root)
content.xml	Document content and automatic styles used in the content.	/ (root)
meta.xml	Document meta information, such as the author or the time of the last save action.	/ (root)

Table 1: XML files of an ODF document

For this article, we are interested in content.xml file which will hold the contents of the document. The document used for this article is a simple shopping list named as Shop.ods. The below table shows the way the first row was entered in the spreadsheet document using OpenOffice and as seen by viewing the corresponding content.xml file in Firefox browser.



	A	B	C
1	Potato	2.0	kg
2	Onion	1.0	kg
3	Cucumber	0.5	kg
4	Capsicum	0.5	kg
5	Carrot	0.5	kg
6	Green chilli	0.25	kg
7	Tomato	0.25	kg
8	Curd	0.5	L
9	Banana	2	doz
10	Milk	2	L
11	Groundnut Oil	5	L

Table 2: Row in a ODF spreadsheet and as saved in content.xml file

A bit of hacking

Now that we know a bit about an ODF document, let us try to create one without actually running a standard software dealing with ODF documents. As noted above, the content.xml file holds the data entered while creating the spreadsheet. So, if this file is changed and zipped along with the rest of the files in the original archive (that is, the .ods file), we will be having a “legal” ODF spreadsheet. To test this, follow the steps below.

1. Open the content.xml file in a regular text editor.

2. In Table 2, the element <table:table-row> has been highlighted. Type in the contents of the highlighted section immediately as a sibling of another <table:table-row> element.

3. In the pasted element, make a change in any of the table:table-cell/text:p element say, something like this.

1. Save the edited content.xml file in the same folder where the Shop.ods file was unzipped into replacing the older one.

2. Select all the contents (that is, the ones mentioned in the manifest.xml file) of the unzipped folder and zip them. Name the zipped file as NewShop.ods.

3. Now, open NewShop.ods in OpenOffice or IBM Lotus Symphony and confirm that the hacking is successful. Note that, OpenOffice successfully opens a file with a .zip extension, so long it is a valid ODF document whereas, Symphony requires the extension to be .ods.

It is now clear that, to “read” an existing ODF spreadsheet in its simplest form, we have to navigate the path of the content.xml XML document. Similarly, to “modify” an existing ODF spreadsheet, we should generate the content.xml XML document correctly. One can, of course, extend this to create and/or modify other XML documents in the .ods archive. But, for the sake of this article, we will concentrate only on the creation of content.xml XML document.

Scheme of this article

With this knowledge of XML documents in an ODF document archive, we will now proceed to demonstrate the usage of DB2 pureXML with ODF documents. We consider the scheme below.

1. We consider an ODF spreadsheet of a simple shopping list where the item name, unit of measure (UOM) and the units required are provided.

2. The content.xml document of this ODF spreadsheet is imported into a DB2 table with an XML column.

3. For the sake of this article we will use the LOAD statement in DB2 to load the content.xml document and ignore any other methods of inserting XML documents into the DB2 table.

4. We will refer to another traditional table which stores the price list.

5. Using the table for price list and the table with XML column for content.xml, we will generate another content.xml XML document that will hold the items with their corresponding prices.

Inserting XML document into DB2 table

We will consider a table to demonstrate DB2 pureXML capabilities. The first table will be keyed via an ID column to differentiate multiple documents. The second column of the table will hold the content.xml file of the ODF spreadsheet document in a XML column. The DDL for this table is reproduced here :

Example 1. DB2 table to hold content.xml file of the ODF spreadsheet

```
-- DOCCONTENTXML table to hold the content.xml file of a ODF document.
-- The table is keyed on DOCID a running sequence number.
CREATE TABLE ODF.DOCCONTENTXML (
       DOCID INTEGER NOT NULL,
       DOCCONTENT XML NOT NULL
) ;
```

IMPORT statements

Inserting the XML document into the table can be done using a simple INSERT SQL statement or using IMPORT statement. However, since we have a considerably sized XML document, we will use IMPORT statement from the command line. To use IMPORT statement, first, save the content.xml in a folder as say, ODSFolder and create a flat file with a single record. Save the flat file as load.txt. This record will have two values separated by a comma as shown below.

Example 2. Contents of flat file to be used for loading DOCCONTENTXML table

```
1,<XDS FIL='content.xml'/>
```

Example 3. IMPORT statement to load content.xml into DOCCONTENTXML table

```
IMPORT FROM load.txt OF DEL XML FROM ODSFolder INSERT INTO ODF.DOCCONTENTXML;
```

Alternatively, IBM Data Studio Developer can be used to load into XML column as it offers a convenient GUI.

Converting the content.xml into row-column format

Getting the cell values from the XML file

To get to the cell values of the spreadsheet, we will start with the required XPath expression. Then, we will develop a working query around the XPath expression to get the results.

The XPath expression

Going back to Table 2, the left column shows the first row of the spreadsheet highlighted. The right column is a part of the generated content.xml file and some of the elements collapsed. The relevant part are put in red boxes. The first box shows the table:table element with an attribute table:name whose value is “Sheet1”. It is the name of the sheet in the complete spreadsheet. This element is repeated as many times as there are sheets in the spreadsheet with the table:name attribute set to the names as set by the user. The second box is the representation of the highlighted row in the left column. Thus, to get to, say, column B of the first row, the path to be navigated (or, the XPath expression) from the root of XML document would be as below.

Example 4. XPath expression to get first row, second column value

```
/office:document-content/office:body/office:spreadsheet/table:table[1]/table:table
```

In other words, we start from the office:document element to office:body and then to office:spreadsheet to look for the first table:table element. The first element represents the first sheet of the spreadsheet document. Then, to get to the second column of first row we navigate to table:row[1]/table:cell[2] and finally to text:p to get the actual text of the cell.

With Firefox Mozilla browser, the above XPath expression can be easily tested using the add-on Xpather (see Resources below). We follow the steps below :

1. Open the content.xml in Firefox Mozilla browser. Right-click on the page and select 'Show in XPather'.

1. The XPather dialog box opens up. Enter the XPath expression in the dialog box and click 'Eval'.

The required query

To wrap the XPath expression into a query, we first note the XML namespaces used. The content.xml file has the namespaces defined as shown below :

Prefix	Namespace

office	“urn:oasis:names:tc:opendocument:xmlns:office:1
table	“urn:oasis:names:tc:opendocument:xmlns:table:1.
text	“urn:oasis:names:tc:opendocument:xmlns:text:1.0

Table 7: Prefix and namespace URI used in content.xml document

The query shown below will use the XML namespaces and XPath expression (that we obtained in Section 6.2) to extract the cell values from the loaded content.xml file. The text in bold-blue shows the XPath expression used to extract cell values.

Example 5. Query to extract cell values from content.xml into tabular format

```
SELECT
X.*
FROM
XMLTABLE(
 XMLNAMESPACES('urn:oasis:names:tc:opendocument:xmlns:office:1.0' AS "office",
                  'urn:oasis:names:tc:opendocument:xmlns:table:1.0' AS "table",
                  'urn:oasis:names:tc:opendocument:xmlns:text:1.0' AS "text"),
   'db2-fn:sqlquery("SELECT
                     DOCCONTENT
                     FROM
                     ODF.DOCCONTENTXML
                     WHERE DOCID = 1")
                    /office:document-content/office:body/office:spreadsheet/table:
    COLUMNS
    "ITEMNAME" CHARACTER (20)  PATH 'table:table-cell[1]/text:p',
    "QUANTITY" DECIMAL   (4,2) PATH 'table:table-cell[2]/text:p',
    "UOM"      CHARACTER (5)   PATH 'table:table-cell[3]/text:p'
 ) AS X;
```

To see the content.xml document in traditional rows and columns format, we create a view using the query above.

Example 6. View on the content.xml document

```
CREATE VIEW ODF.V_CONTENT AS
(
 SELECT
 X.*
 FROM
 XMLTABLE(
   XMLNAMESPACES('urn:oasis:names:tc:opendocument:xmlns:office:1.0' AS "office",
                 'urn:oasis:names:tc:opendocument:xmlns:table:1.0'  AS "table",
                 'urn:oasis:names:tc:opendocument:xmlns:text:1.0' AS "text"),
   'db2-fn:sqlquery("SELECT
                     DOCCONTENT
                     FROM
                     ODF.DOCCONTENTXML
                     WHERE
                     DOCID = 1"
    )/office:document-content/office:body/office:spreadsheet/table:table[1]/table:
 COLUMNS
 "ITEMNAME" CHARACTER (20)  PATH 'table:table-cell[1]/text:p',
 "QUANTITY" DECIMAL   (4,2) PATH 'table:table-cell[2]/text:p',
 "UOM"      CHARACTER (5)   PATH 'table:table-cell[3]/text:p'
) AS X
);
```

A simple SELECT query on this view shows that we have successfully 'converted' the content.xml document in the ODF archive for spreadsheet into a row-column format.

Example 7. Result of SELECT on the view V_CONTENT.

```
SELECT * FROM ODF.V_CONTENT ;

ITEMNAME             QUANTITY UOM
-------------------- -------- ----
Potato                   2.00 kg
Onion                    1.00 kg
Cucumber                 0.50 kg
Capsicum                 0.50 kg
Carrot                   0.50 kg
Green chilli             0.25 kg
Tomato                   0.25 kg
Curd                     0.50 L
Banana                   2.00 doz
Milk                     2.00 L
Groundnut Oil            5.00 L

11 record(s) selected.
```

Shopping prices

We will now build the content.xml for an ODF spreadsheet that will hold the shopping list – discussed above – with an additional column for cost of items in the shopping list. We will also provide an additional row for the grand total of the items in shopping list.

Building the list of shopping prices

We create a simple table that holds the unit price of shopping items with the DDL shown below.

Example 8. DDL for table to hold the shopping prices

```
--Table to hold the unit price of shopping items
CREATE TABLE ODF.SHOPPER
(ITEMNAME CHARACTER(50),
 UNIT_PRICE DECIMAL (4,2));
```

This table is inserted with values such that all the item names are available in the SHOPPER table. To obtain the list of item names and the cost of items, we use the following query. Note that, this query uses the view V_CONTENT. We could have used the base table directly. Usage of the view makes the query 'compact' and easier to follow.

Example 9. Query to obtain list and cost of items with grand total

```
SELECT
COALESCE(P.ITEMNAME,' TOTAL')
ITEMNAME,
SUM(P.QUANTITY) QTY,
COALESCE(P.UNIT_PRICE, 0) UNIT_PRICE,
SUM(P.COST) COST
FROM
(SELECT
 A.ITEMNAME,
 A.QUANTITY,
 B.UNIT_PRICE,
 A.QUANTITY*B.UNIT_PRICE COST
 FROM
 ODF.V_CONTENT A,
 ODF.SHOPPER B
 WHERE
 UPPER(A.ITEMNAME) = B.ITEMNAME
) P
GROUP BY
ROLLUP(P.ITEMNAME,P.QUANTITY,P.UNIT_PRICE)
HAVING (P.ITEMNAME IS NULL OR
        (P.ITEMNAME   IS NOT NULL AND
         P.QUANTITY   IS NOT NULL AND
         P.UNIT_PRICE IS NOT NULL AND
         SUM(P.COST)  IS NOT NULL AND
         SUM(P.QUANTITY) IS NOT NULL
        )
)
ORDER BY COALESCE(P.ITEMNAME,' ') DESC
```

The result of the above query is shown below.

Example 10. Result of query for obtaining list and cost of items

```
ITEMNAME             QTY    UNIT_PRICE COST
-------------------- ------ ---------- --------
Tomato                 0.25       6.00     1.50
Potato                 2.00       5.50    11.00
Onion                  1.00       4.00     4.00
Milk                   2.00      40.00    80.00
Groundnut Oil          5.00      30.00   150.00
Green chilli           0.25       8.00     2.00
Curd                   0.50      20.00    10.00
Cucumber               0.50      10.00     5.00
Carrot                 0.50      15.00     7.50
Capsicum               0.50      20.00    10.00
Banana                 2.00       6.00    12.00
TOTAL                 14.50       0.00   293.00
```

View with list and cost of items

Now that we have the result of computation of cost of items, we will take this into a view. This is purely for convenience. We can proceed without using this view.

Example 11. View with list and cost of items in shopping list

```
CREATE VIEW ODF.V_SHOP_COST AS
(SELECT
 COALESCE(P.ITEMNAME,' TOTAL') ITEMNAME,
 SUM(P.QUANTITY)               QTY,
 COALESCE(P.UNIT_PRICE, 0)     UNIT_PRICE,
 SUM(P.COST)                   COST
 FROM
 (SELECT
  A.ITEMNAME,
  A.QUANTITY,
  B.UNIT_PRICE,
  A.QUANTITY*B.UNIT_PRICE COST
  FROM
  ODF.V_CONTENT A,
  ODF.SHOPPER   B
  WHERE
  UPPER(A.ITEMNAME) = B.ITEMNAME
 ) P
 GROUP BY
 ROLLUP(P.ITEMNAME,P.QUANTITY,P.UNIT_PRICE)
 HAVING (P.ITEMNAME IS NULL OR
         (P.ITEMNAME      IS NOT NULL AND
          P.QUANTITY      IS NOT NULL AND
          P.UNIT_PRICE    IS NOT NULL AND
          SUM(P.COST)     IS NOT NULL AND
          SUM(P.QUANTITY) IS NOT NULL
         )
 )
);
```

XQuery Update to build content.xml document

At this point, we have 'shredded' the shopping list (from the content.xml of ODF spreadsheet), joined with the relational table having the unit prices of items and generated a relational output of the cost of items in shopping list with the grand total. Our aim is now to convert this relational output into the format of content.xml so that the result is a complete ODF spreadsheet.

One way of building the content.xml is to start from the scratch and we build the whole of the document. This approach would be quite tedious. A better way would be to generate the relevant part in the content.xml part and then insert into an otherwise empty content.xml document. This approach will also allow us to make use of XQuery Update facility in DB2 pureXML.

A quick side note on XQuery Update

XQuery is a query and functional programming language designed to query XML documents. It became a W3C Candidate Recommendation on 23rd January, 2007. XQuery Update facility is an extension to XQuery that allows update (insert, modify and delete) of the XML documents. It became a W3C Candidate Recommendation on 14th March, 2008.

Generating the content.xml for the shopping prices

To start with, we will create a dummy content.xml that is complete and ODF compliant in all respects except it would resemble an empty spreadsheet. Next, we will generate XML sequences that is correct for a row and it's columns of a spreadsheet. Then, we will collect all of these XML sequences and insert into the dummy content.xml document.

To create the dummy document, we simply open the spreadsheet for shopping list, delete all rows and columns and save it with another name. Then, we unzip this document and refer the content.xml document. Here is a screenshot of how this document would look like.

Insert this dummy document into the ODF.DOCCONTENTXML table with DOCID as zero using the IMPORT method described before. Our aim now is to update this document such that, <table:table-row> elements are added as children of <table:table> and as siblings of <table:table-column>.

Building the <table:table-row> elements

In our example, each row of spreadsheet would be having four columns. Therefore, each row element – as required in content.xml – would have four children for the number of columns. A sample of <table:table-row> is shown below. The XML namespace declarations have been omitted for the sake of brevity.

Example 12. A sample <table:table-row> element

```
<table:table-row table:style-name="ro1">
  <table:table-cell office:value-type="string"><text:p>Tomato</text:p></table:tabl
  <table:table-cell office:value-type="string"><text:p>.25</text:p></table:table-c
  <table:table-cell office:value-type="string"><text:p>6.00</text:p></table:table-
  <table:table-cell office:value-type="string"><text:p>1.5000</text:p></table:tabl
</table:table-row>
```

To build the <table:table-row> elements, then, we need to generate <table:table-cell> for each column of the row in the relational table and then aggregate them to a single <table:table-row> element. We will use the XMLAGG function to build the elements and store in a table the generated elements so that it is easier to use while doing the update of XML document. The table to hold the generated elements is defined as shown below.

Example 13. Table to hold generated <table:table-row> elements

```
CREATE TABLE ODF.SHOP_COST_XML (SHOP_COST XML);
```

Example 14. Insert generated <table:table-row> elements with <dummy> parent

```
INSERT INTO ODF.SHOP_COST_XML (SHOP_COST)
(SELECT
 XMLDOCUMENT(
  XMLELEMENT(NAME "dummy",
             XMLNAMESPACES('urn:oasis:names:tc:opendocument:xmlns:office:1.0' AS "
                           'urn:oasis:names:tc:opendocument:xmlns:table:1.0'  AS "
                           'urn:oasis:names:tc:opendocument:xmlns:text:1.0'   AS "
   XMLAGG(
    XMLELEMENT(NAME "table:table-row",
               XMLATTRIBUTES('ro1' AS "table:style-name"),
               XMLELEMENT(NAME "table:table-cell",
                          XMLATTRIBUTES('string' AS "office:value-type"),
                          XMLELEMENT(NAME "text:p", STRIP(O.ITEMNAME)
               )
    ),
    XMLELEMENT(NAME "table:table-cell",
               XMLATTRIBUTES('string' AS "office:value-type"),
               XMLELEMENT(NAME "text:p", O.QTY )
    ),
    XMLELEMENT(NAME "table:table-cell",
               XMLATTRIBUTES('string' AS "office:value-type"),
               XMLELEMENT(NAME "text:p", O.UNIT_PRICE )
    ),
    XMLELEMENT(NAME "table:table-cell",
               XMLATTRIBUTES('string' AS "office:value-type"),
               XMLELEMENT(NAME "text:p", O.COST)
    )
   )
   ORDER BY O.ITEMNAME DESC
  )
 )
)
FROM
ODF.V_SHOP_COST O
);
```

Finally, we are now ready to update our dummy XML document. The query to do the update is shown below. Note that, this query is not inserting the resulting into any table. It is left to the reader to route the output to a table column, file, message queue, etc.

Example 15. Updated content.xml document

```
VALUES XMLQUERY(
 'declare namespace office="urn:oasis:names:tc:opendocument:xmlns:office:1.0" ;
  declare namespace table="urn:oasis:names:tc:opendocument:xmlns:table:1.0" ;
  transform
  copy $dummy := db2-fn:sqlquery("SELECT DOCCONTENT FROM ODF.DOCCONTENTXML WHERE D
       $rows  := db2-fn:sqlquery("SELECT * FROM ODF.SHOP_COST_XML")
  modify do
    insert $rows/dummy/* as last into $dummy/office:document-content/office:body/o
  return $dummy '
);
```

To test that our update has worked successfully run, we route the output of query above to a file called content.xml. Replace the content.xml file in the unzipped archive of our original shopping list spreadsheet with the one created now. Zip the files back and save it with name as ShopPrices.ods and open with OpenOffice. Here is the screenshot.

And, we are done !

Next steps

Further enhancements may involve manipulating other XML documents in a ODF compliant archive. For example, the meta.xml document holds information about the document itself. This information can be accessed and set by going to File -> Properties... dialog box. For example, for the document in this article (Shop.ods), this is how the dialog box looks like.

The corresponding meta.xml document (partial snapshot) is shown below.

Thus, while generating the spreadsheet, one can write queries to generate the meta.xml document with correct values for creator, creation time-stamp, etc.

Resources

Important resources :

1. DB2 Express-C [http://www-01.ibm.com/software/data/db2/express/download.html] – Free for download, develop, deploy and distribute.
2. DB2 v9 pureXML [http://www.ibm.com/db2/9] – All information about DB2 V9 pureXML
3. DB2 v9 Product Manuals [http://www.ibm.com/support/docview.wss?rs=71&uid=swg27009552] – Especially, refer the ones for XQuery and XML.
4. IBM Data Studio [http://www-01.ibm.com/software/data/studio/] – Govern, design, develop and deploy databases and data driven applications.

5. ODF Specification [http://www.oasis-open.org/committees/office/] – Link to the complete specification

6. OpenOffice.org XML essentials [http://books.evc-cit.info/book.php] – Nice explanation of XML data formats.

7. Thinking XML : The open office file format [http://www.ibm.com/developerworks/xml/library/x-think15/] – An earlier developerWorks article discussing the file formats of OpenOffice.

8. Manage ODF and Microsoft Office 2007 documents with DB2 9 pureXML [http://www.ibm.com/developerworks/db2/library/techarticle/dm-0705gruber/] – An earlier developerWorks article on using ODF and Microsoft Office documents with DB2 9 pureXML.

9. XPather add-on [https://addons.mozilla.org/addon/1192] – Nifty tool for Firefox Mozilla users to quickly check an XPath expression.

10. Update XML in DB2 9.5 [http://www.ibm.com/developerworks/db2/library/techarticle/dm-0710nicola/%5C] – Nice article on developerWorks explaining update of XML documents.

Biography

Nagesh Subrahmanyam is a Software Developer in IBM Global Services, India.....