

XML application migration from DB2 for z/OS V8 to DB2 9, Part 3: Sample DB2 for z/OS XML application migration scenario

Skill Level: Intermediate

[Hardeep Singh \(hardeep@us.ibm.com\)](mailto:hardeep@us.ibm.com)

Architect DB2 XML tooling, DB2 XML application migration
IBM

[John R. Shelton \(johnshel@us.ibm.com\)](mailto:johnshel@us.ibm.com)

Software engineer, DB2 XML Extender
IBM

17 Apr 2008

Step through the XML application migration process using an example scenario. Create a sample application using DB2® for z/OS®, Version 8 XML functionality, then migrate that application using DB2 9 for z/OS XML capabilities.

Note: *These three articles were originally written for workstation DB2 by Hardeep Singh. They were then modified to be applicable to DB2 for z/OS by John Shelton.*

Introduction

In this article, walk through an XML application migration process using an example scenario. The scenario goes through the steps of creating a sample application using DB2 for z/OS V8 functionality for XML. The application is then migrated using DB2 9 for z/OS XML capabilities. As much as possible the scenario is developed using scripts that can be run from the DB2 SPUFI or DSNTEP2 SQL processors. The rest of the code is a Java™ application.

Goals of the scenario

The scenario has been built keeping the migration process in mind. The step-by-step implementation of the scenario walks you through the following:

- Migrating tables with XML data.
- Migrating DTDs to XML schemas, then registering the XML schemas.
- Migrating XML collection document access definition (DAD) (shred) to annotated XML schema. Showing the use of annotated XML schema functions instead of XML Extender functions.
- Migrating XML Collection DAD (publish) to SQL/XML calls. Showing the use of SQL/XML publishing functions instead of DAD files.
- Migrating data. Showing the use of unload and load.
- Migrating XML indexes. Showing how to create XML indexes and how to use information in the DAD to create the XML expression for the index.
- Migrating SQL queries and application code. Showing the usage of SQL/XML functions like `XMLPARSE`, `DSN_XMLVALIDATE`, and `update` stored procedure instead of XML Extender `query`, `update`, and `insert` functions.

You also look at:

- Changes in relational schemas to take advantage of new XML storage
- Changes in application design to take advantage of new query language

Description of the scenario

The sample application is an online discount shop for members only. Users register to the site and then buy products from the Web site. They add the items to be purchased to the online shopping cart. On checkout, an electronic purchase order is created for the items. An invoice is generated and sent to the user's email account. The purchase order is used by shipping and handling to send the items to the customer. At any point in the process, the customer can check the status of the purchase order. Reports are generated for items purchased and customer lists.

Note: Not all the steps in the scenario are implemented. Only those relevant to the migration process are explained.

Create the scenario

All the scripts used in creating the scenario are attached to the article (see [Download](#) section). Create a migrate directory and save the scripts and XML document in the migrate directory.

You can then run the scripts from DB2 SPUFI or DSNTEP2 as indicated in the steps.

Note: After each step you can check to see if the data has been changed as expected.

Enable server for DB2 XML Extenders

```
Run job SDXXJCL(DXXGPREP)
```

Create the tables and indexes

The following tables are created:

- **Customer:** Customer name and address.
- **Phone:** Customer phone number information.
- **Product:** Product information extracted from the product XML document. This table has a column that also contains the intact XML document as a character large object (CLOB).
- **Color:** Color options for the product.
- **PurchaseOrder:** Purchase order information in an XMLClob column. It is supported by the following side tables.
- **Shipto:** Ship to address.
- **Item:** Lists all the items in the purchase order.

Listing 1. Create the tables and indexes

```
CREATE TABLE Product (Pid VARCHAR(10) NOT NULL PRIMARY KEY,
Name VARCHAR(128), Category VARCHAR(32), Price DECIMAL(30,2), Info CLOB);

CREATE TABLE Color (Pid VARCHAR(10) NOT NULL,
Name VARCHAR(64), CONSTRAINT Pidfk FOREIGN KEY (Pid)
REFERENCES Product(Pid) ON DELETE NO ACTION ON UPDATE NO ACTION);

CREATE TABLE Customer ( Cid BIGINT NOT NULL PRIMARY KEY,
Name varchar(64), Email varchar(128),Country varchar(2),
Street varchar(128),City varchar(64),State varchar(64), Zip varchar(12));

CREATE TABLE Phone ( PhNo varchar(12) not null primary key,
Cid BIGINT NOT NULL,PhoneType char(12),CONSTRAINT cidfk FOREIGN KEY (cid)
REFERENCES Customer(cid) ON DELETE NO ACTION ON UPDATE NO ACTION);

CREATE INDEX cididx ON customer (zip);
CREATE UNIQUE INDEX CustomerIDX1 ON Customer(Email);

CREATE TABLE PurchaseOrder ( POid BIGINT NOT NULL PRIMARY KEY,
Status VARCHAR(10) NOT NULL WITH DEFAULT 'New',
Info DB2XML.XMLCLOB NOT NULL, row_id rowid not null generated always);
```

Create the DTD and XML schemas

For this scenario, consider that the source of all inserted data is XML data that has a DTD or an XML schema associated with it. The first thing to do here is to create the XML schemas and DTDs:

- Products DTD
- Customer DTD
- Purchase Order DTD

Extract the DTDs from the zip file and register them to the database (see [Download](#) section).

From SPUFI or DSNTEP2, run the following:

Listing 2. Insert the DTDs

```
insert into db2xml.dtd_ref (dtdid, content, usage_count, author, creator,
updater) values ('customerdtd', db2xml.XMLCLOBFromFile(
' /migrate/customer.dtd'), 0, 'xml', 'xml', 'xml');

insert into db2xml.dtd_ref (dtdid, content, usage_count, author, creator,
updater) values ('productdtd', db2xml.XMLCLOBFromFile(
' /migrate/ product.dtd'), 0, 'xml', 'xml', 'xml');

insert into db2xml.dtd_ref (dtdid, content, usage_count, author, creator,
updater) values ('podtd', db2xml.XMLCLOBFromFile(
' /migrate/po.dtd'), 0, 'xml', 'xml', 'xml');

Commit;
```

Note: The root directory path /migrate for the DTD files needs to be substituted with the directory in which the files are extracted.

Create and enable mapping

To shred the incoming XML documents to the relational tables, you first need to create the DAD files. Once the DAD files are created, they can either be registered and enabled for shredding or passed into the XML Extender shred user-defined functions (UDFs) at run time.

Note: Different mapping schemas are used for shredding the XML data to the relational tables in order to cover more usage possibilities.

Customer DAD (XML collection - relational database (RDB) node mapping)

XML data for customer information is shredded to the customer and phone tables using an RDB formatted DAD (RDB node mapping enables decomposition of the

XML).

Extract the customer.dad file to the migrate directory.

Note: For this scenario, pass this DAD file to the DB2 XML Extender shred function at runtime.

PurchaseOrder DAD (XML column)

XML data representing a purchase order is stored intact in a DB2 XML Extender column. Part of the purchase order information is shredded into relational side tables for indirect indexing. The DAD format for this mapping is also an RDB node mapping that creates the side tables.

Extract the po.dad file to the migrate directory and then enable the purchaseorder.info column. The information in the DAD file is used, when the XML column is enabled, to create the `insert`, `update` and `delete` triggers that help in keeping the side tables in sync with the XML document. This means the XML documents are inserted using a SQL `insert` command and the shredding is done by the triggers.

```
dxxadm enable_column -a V81A purchaseorder info "/migrate/po.dad" -r poid
```

Invoice DAD (XML collection - SQL mapping)

In order to publish an XML invoice from the data in the relational table, create an invoice.dad file. The invoice.dad file maps information from the side tables associated with a given purchase order to create an XML formatted invoice. Since the DAD file is only needed for publishing, the format used for it is SQL mapping.

Extract the invoice.dad file to the migrate directory.

Insert/update XML data

Once all the mapping information has been created and set up, the next step is to populate the tables with data. Ignore how this information was initially gathered for this exercise, and just assume it to be there in XML files on the local file system.

Extract the XML files from the zip file into the migrate directory.

For populating the product and color table, the shredding is done in a Java application that uses a Document Object Model (DOM) parser. The shredded XML data is inserted into the product and color tables using SQL `insert` statements. The XML data is also stored as a CLOB in the database.

Note: The full source code for the product.java and the XMLParse.java file is in the attached zip file (see [Download](#) section).

Listing 3. Insert the XML data

```
XMLParse dom= new XMLParse(filename,true);
String sql=
"insert into PRODUCT values('"+dom.getValue("/product/@pid").trim()
+"','" +
dom.getValue("/product/description/name/text()").trim()
+"','" + dom.getValue("/product/description/category/text()").trim()
+"','"+dom.getValue("/product/description/price/text()")
+"','" + "?"");

PreparedStatement iStmt=con.prepareStatement(sql);
File inputfile= new File(filename);
long filesize=inputfile.length();
BufferedReader in = new BufferedReader(new FileReader(inputfile));
iStmt.setCharacterStream(1,in,(int)filesize);
int rc= iStmt.executeUpdate();
if(rc==1)
{
String pid=dom.getValue("/product/@pid");
dom.find("//color",true);
}
}
String insertsql= "insert into color (pid,name) values('"+
pid + "','" +dom.getValue("text()",i)+"')";
Statement stmt = con.createStatement();
int rescnt= stmt.executeUpdate(insertsql);
}
```

Copy the files to the migrate directory, compile and run it, passing the name of each product XML file as an argument.

On Unix System Services: Set CLASSPATH

```
export CLASSPATH=$CLASSPATH:/usr/lpp/db2810/jcc/classes/db2jcc.jar
```

Note: The above assumes either the sh or bash shells. Change as appropriate for csh, tsh, and others.

Run the utility.

```
"%DB2PATH%\java\jdk\jre\bin\java.exe" product product1.xml
"%DB2PATH%\java\jdk\jre\bin\java.exe" product product2.xml
"%DB2PATH%\java\jdk\jre\bin\java.exe" product product3.xml
"%DB2PATH%\java\jdk\jre\bin\java.exe" product product4.xml
```

For populating the customer and phone table, use the XML Extender utility dxxshred that takes in the customer.dad file to shred the input customer XML data to the customer and phone tables.

```
shred -a V81A /migrate/customer.dad /migrate/customer1.xml
shred -a V81A /migrate/customer.dad /migrate/customer2.xml
```

```
shred -a V81A /migrate/customer.dad /migrate/customer3.xml
```

For populating the PurchaseOrder and its associated side tables, since the purchaseorder.info is an DB2 XML Extender column that has been enabled with the po.dad file, a simple insert statement stores the XML data in the column and also creates the necessary records in the side tables.

```
INSERT INTO PurchaseOrder VALUES(2001,'',db2xml.XMLCLOBFromFile('/migrate/po.xml'));
Commit;
```

Update the purchase order

Once the order has been shipped, change the status of the purchase order to shipped using the db2xml.update UDF.

```
Update purchaseorder set status='shipped', info=db2xml.update(info,
'/purchaseOrder/@status','shipped') where poid=2001;
```

Generate reports

Generate an invoice based on the purchase order

Once the purchase order is created, an invoice can be generated and sent to the user. Use the XML Extender publish function and invoice.dad to create the invoice.

```
call db2xml.dxxgenxmlclob( db2xml.xmlclobfromfile(
'/migrate/invoice.dad'), 0,'',?,?,?,?);
```

Customer Information

To publish the customer information in an XML format, you can use the same RDB mapped customer.dad file that was used for shredding the customer document to the customer and phone tables. Use the DB2 XML Extender publishing UDF for collections.

```
call db2xml.dxxgenxmlclob( db2xml.xmlclobfromfile(
'/migrate/customer.dad'), 0,'',?,?,?,?);
```

Customer list

To get a mailing list of all the customers from the customer table, use SQL/XML to publish the information in an XML format.

Listing 4. Produce customer list in XML format

```

SELECT XML2CLOB(
XMLELEMENT ( NAME "customer",
XMLATTRIBUTES (e.cid AS "id"),
XMLELEMENT (NAME "name", e.name),
XMLELEMENT (NAME "email", e.email),
XMLELEMENT (NAME "zip", e.zip)
)
) FROM customer e;

```

Migrate the database objects to DB2 9

All the preceding steps could take place in DB2, Version 8, but the steps after this point use functionality that is available only in DB2 9. If you have not already done so, at this point you have to upgrade your database server to DB2 9. Follow the instructions in your DB2 9 installation guide to do this.

Create a migrate1 directory on your local HFS file system.

Note: There is some restructuring of the relational tables to get rid of redundant tables and columns. Also, some of the parsing that was done in the application layer moves to the server (using SQL/XML and XQuery).

Create the new tables with XML columns

The database schema for some of the tables has been simplified as you no longer need to shred out data for querying purposes.

Listing 5. Create new tables

```

CREATE TABLE Product(Pid VARCHAR(10) NOT NULL PRIMARY KEY, Info XML);
CREATE TABLE PurchaseOrder(POid BIGINT NOT NULL PRIMARY KEY, Info XML );

CREATE TABLE Customer ( Cid BIGINT NOT NULL PRIMARY KEY,
Name varchar(64), Email varchar(128),Country varchar(2),
Street varchar(128),City varchar(64),State varchar(64), Zip varchar(12));

CREATE TABLE Phone ( PhNo varchar(12) not null primary key,
Cid BIGINT NOT NULL,PhoneType char(12),CONSTRAINT cidfk FOREIGN KEY (cid)
REFERENCES Customer(cid) ON DELETE NO ACTION ON UPDATE NO ACTION);

CREATE INDEX cididx ON customer (zip);
CREATE UNIQUE INDEX CustomerIDX1 ON Customer(Email);

```

Create XML Indexes

Product table: The following indexes have been created for the product table in order to help queries that are searching for product items based on product ID, product category, product name, and product price range.

Note: The xmlpattern value for the XML index corresponds to the XPath that are used in the Java application to extract data, from the incoming XML purchase order

document, using the DOM parser.

Listing 6. Create XML indexes on Product table

```
create index product_idon product (info) generate key using
xmlpattern '/ product/@pid' as sql varchar hashed;
create index product_categoryon product (info) generate key using
xmlpattern '/ product/ category' as sql varchar hashed;
create index product_nameon product (info) generate key using
xmlpattern '/ product/ name' as sql varchar(64);
create index product_priceon product (info) generate key using
xmlpattern '/ product/ price' as sql varchar(12);
```

Purchase Order table: To create XML indexes on the purchase order table, look at the purchase order DAD file to see which paths were used to create the side tables.

1. Copy the DAD file from the XML_USAGE table to the local directory.
2. Look at the paths used to create the side tables for indirect indexes
3. Use these paths in the create index calls.

Listing 7. Create XML indexes on Purchase Order table

```
create unique index order_key on purchaseorder (info) generate key using
xmlpattern '/purchaseOrder/@poid' as sql double;

create index order_status on purchaseorder (info) generate key using
xmlpattern '/purchaseOrder/@status' as sql varchar HASHED;

create index order_item_id on purchaseorder (info) generate key using
xmlpattern '/purchaseOrder/items/item/@pid' as sql varchar HASHED;

create index order_item_shipdate on purchaseorder (info) generate key
using xmlpattern '/purchaseOrder/items/item/shipDate' as sql DATE;

create index order_item_price on purchaseorder (info) generate key
using xmlpattern '/purchaseOrder/items/item/USPrice' as sql VARCHAR(12);

create index order_item_qty on purchaseorder (info) generate key using
xmlpattern '/purchaseOrder/items/item/quantity' as sql DOUBLE;
```

Migrate DTDs and DAD mapping to XML schemas

Migrate DTD to XML schema

Copy the DTDs from the DTD_REF table in the migrate database to the migrate1 directory.

Listing 8. Copy DTDs from database to files

```

select db2xml.XMLFileFromCLOB(db2xml.clob(content), '/migrate1/customer.dtd')
from db2xml.dtd_ref where DTDID='customerdtd';

select db2xml.XMLFileFromCLOB(db2xml.clob(content), '/migrate1/po.dtd')
from db2xml.dtd_ref where DTDID='podtd';

select db2xml.XMLFileFromCLOB(db2xml.clob(content), '/migrate1/product.dtd')
from db2xml.dtd_ref where DTDID='productdtd';

```

Now convert the DTD to XML schema files. There are third party Web-based utilities for XML that can help you deal with conversions between DTDS and XML schemas.

Migrate customer.dad to an annotated XML schema customer.xsd

The next step is to take the customer XML schema that was created from the customer.DTD file and add annotations to the XML schema. These annotations contain the necessary information needed to map the data that needs to be shredded to the relational tables. Use the customer DAD to get the mapping information to be used in the annotations. You can also use the utility found in the Download section of the article "[From DAD to annotated XML schema decomposition](#)" (developerWorks, April 2006) to help you convert a DAD to an XML schema.

Extract the XML schema customer.xsd to the migrate1 directory. In the file, change the schema name from "hardeep" to your schema name.

Listing 9. XML schema customer.xsd

```

<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
  xmlns:sql="http://www.ibm.com/xmlns/prod/db2/xdb1">
<xs:annotation><xs:appinfo><sql:defaultSQLSchema>hardeep
</sql:defaultSQLSchema></xs:appinfo></xs:annotation>
...

```

Register the XML schemas to the XSR.

Listing 10. Register the XML schemas

```

CALL SYSPROC.XSR_REMOVE('SYSXSR', 'migrate.po');
CALL SYSPROC.XST_REGISTER('SYSXSR', 'migrate.po', 'http://migrate.db2',
  :content_host_var, :docproperty_host_var);
CALL SYSPROC.XSR_COMPLETE('SYSXSR', 'migrate.po', :schemaproperty_host_var, 0);

CALL SYSPROC.XSR_REMOVE('SYSXSR', 'migrate.product');
CALL SYSPROC.XST_REGISTER('SYSXSR', 'migrate.product', 'http://migrate.db2',
  :content_host_var, :docproperty_host_var);
CALL SYSPROC.XSR_COMPLETE('SYSXSR', 'migrate.product', :schemaproperty_host_var, 0);

CALL SYSPROC.XSR_REMOVE('SYSXSR', 'migrate.customer');
CALL SYSPROC.XST_REGISTER('SYSXSR', 'migrate.customer', 'http://migrate.db2',
  :content_host_var, :docproperty_host_var);
CALL SYSPROC.XSR_COMPLETE('SYSXSR', 'migrate.customer', :schemaproperty_host_var, 1);

```

Note: The annotated XML schema customer.xsd is also enabled for decomposition when it is registered.

Unload and load the intact XML data

Run the following job to unload the data:

Listing 11. Unload the XML data

```
//UNLOAD      JOB      .....
//*
//JOBLIB      DD      DISP=SHR,DSN=DB2A.SDSNLOAD
//*
//PRODUNLD   EXEC   DSNUPROC,UID='PRODUNLD',UTPROC=' ',SYSTEM='V81A'
//SYSREC     DD   DSN=JRS.PRODUNLD.SYSREC,DISP=OLD
//SYSPUNCH  DD   DSN=JRS.PRODUNLD.SYSPUNCH,DISP=OLD,
//SYSPRINT  DD   SYSOUT=*
//SYSIN     DD   *
      TEMPLATE LOBFV DSN 'JRS.PRODUNLD.RESUME'
                DSNTYPE(PDS) UNIT(SYSDA)
UNLOAD TABLESPACE JRS.PRODTS NOPAD
      FROM TABLE JRS.PRODUCT
      (PID      VARCHAR(128),
       INFO     CLOB TRUNCATE)
//*
//POUNLD     EXEC   DSNUPROC,UID='POUNLD',UTPROC=' ',SYSTEM='V81A'
//SYSREC     DD   DSN=JRS.POUNLD.SYSREC,DISP=OLD
//SYSPUNCH  DD   DSN=JRS.POUNLD.SYSPUNCH,DISP=OLD,
//SYSPRINT  DD   SYSOUT=*
//SYSIN     DD   *
      TEMPLATE LOBFV DSN 'JRS.POUNLD.RESUME'
                DSNTYPE(PDS) UNIT(SYSDA)
UNLOAD TABLESPACE JRS.POTS NOPAD
      FROM TABLE JRS.PURCHASEORDER
      (POID     VARCHAR(128),
       INFO     CLOB TRUNCATE)
/*
```

Load the XML data into the XML columns

Listing 12. Load the XML data

```
/LOAD        JOB      .....
//*
//JOBLIB      DD      DISP=SHR,DSN=DB2A.SDSNLOAD
//*
//LOADPROD   EXEC   DSNUPROC,UID='LOADPROD',UTPROC=' ',SYSTEM='V91A'
//SYSREC     DD   DSN=JRS.PRODUNLD.SYSREC,DISP=OLD
//SYSPRINT  DD   SYSOUT=*
//SYSIN     DD   *
LOAD DATA INDDN SYSREC  LOG NO RESUME YES
EBCDIC CCSID(00037,00000,00000)
INTO TABLE "JRS"."PRODUCT"
WHEN(00001:00002) = X'0009'
IGNOREFIELDS YES
```

```

( "PID"
  POSITION( 00003:00132) VARCHAR
, "INFO"
  POSITION( 00133:00389) XML PRESERVE WHITESPACE
)
/*
//LOADPO EXEC DSNUPROC,UID='POLOAD',UTPROC='',SYSTEM='V91A'
//SYSREC DD DSN=JRS.POUNLD.SYSREC,DISP=OLD
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
LOAD DATA INDDN SYSREC LOG NO RESUME YES
EBCDIC CCSID(00037,00000,00000)
INTO TABLE "JRS"."PURCHASEORDER"
WHEN(00001:00002) = X'0009'
IGNOREFIELDS YES
( "POID"
  POSITION( 00003:00132) VARCHAR
, "INFO"
  POSITION( 00133:00389) XML PRESERVE WHITESPACE
)
/*

```

In this scenario, you modified the table schema and completely dropped the side tables and some of the shredded columns. This has a big impact on the application queries. An alternate approach could be to use an annotated schema shred to decompose all new documents to existing side tables, and only change the column type without dropping the side tables. The advantage with this approach is that the application queries to the side tables do not change and much of the application logic based on them is preserved. The main drawback with this approach is that there is no inherent synchronization between the new XML column and the side tables. And since Native XML columns cannot have triggers defined on them, the synchronization code has to be added to the application logic.

Note: The step to migrate data for the tables containing the shredded data (meaning relational data only tables) is not described here since it has not changed from DB2, Version 8.

Insert or update XML data

In this step, replace the existing queries or code for inserting and updating XML data with the new XML functionality introduced in DB2 9.

Note: You can delete the existing records in the product table that you imported from the old database to test out the code. For the purchase order XML data, the insert statement does not change at all for calls from within an application.

```
INSERT INTO PurchaseOrder VALUES(?,?)
```

Note: If you were using SPUFI or DSNTEP2, the XMLPARSE would have to be explicitly given.

```
INSERT into PurchaseOrder VALUES (2001, XMLPARSE(DOCUMENT
db2xml.XMLCLOBFromFile('/migrate/po.xml') preserve whitespace));
```

Note: You can delete the existing records in the product table that you imported from the old database to test out the code.

For the purchase order XML data, the `insert` statement does not change at all for calls from within an application.

```
INSERT INTO PurchaseOrder VALUES(?,?)
```

Note: If you were using SPUFI or DSNTEP2, the XMLPARSE would have to be explicitly given.

```
INSERT into PurchaseOrder VALUES (2001, XMLPARSE(DOCUMENT
db2xml.XMLCLOBFromFile('/migrate/po.xml') preserve whitespace));
```

Note: In order to use the `db2xml.XMLCLOBFromFile` function, you need to enable the server for DB2 XML Extender. There is no comparable file function in DB2 9.

For shredding customer XML data, use annotated XML schema decompose stored procedure XDBDECOMPXML while inserting the data instead of the XML Extender shred functions.

```
CALL SYSPROC.XDBDECOMPXML('SYSXSR', 'migrate.customer', :xml doc_host_var,
:docid_host_var, 0, NULL, :doctor_host_var, NULL);
```

Update the purchase order: Since the status attribute is part of the XML document, use the update stored procedure to do sub-document updates. For more information on the update stored procedure, refer to "[Partial updates to XML documents in DB2 Viper](#)" (developerWorks, May 2006).

Listing 13. Update the XML document

```
Call DB2XMLFUNCTIONS.XMLUPDATE (
'<updates><update action="replace" col="1"
path="/purchaseOrder/@status">shipped</update></updates>',
'Select info from purchaseorder where poid=2001',
'update purchaseorder set info=? where poid=2001',?,?);
```

Generate reports

XML invoice based on the purchase order - This part changes completely since the side tables are no longer available. Instead of using the DAD, you use SQL/XML functions to generate the invoice from the purchaseorder XML document stored natively in the info column.

Note: The XMLTABLE function is expected to ship post DB2 9 GA.

Listing 14. Produce Purchase Order invoice

```

SELECT
  XMLELEMENT(NAME "invoice", poid,
    XMLELEMENT(NAME "customer",
      name,
      XMLELEMENT(NAME "address",
        street, state, zip)
    ),
    XMLELEMENT(NAME "items",
      (SELECT XMLAGG(XMLELEMENT(NAME "item", pid,
        productName, quantity, USPrice)
      )
    )
  )
FROM XMLTABLE('item'
  PASSING O.items
  COLUMNS "@pid" XML,
    "productName" XML,
    "quantity" XML,
    "USPrice" XML
  ) T(pid, productName, quantity, USPrice)
)
)
FROM PPP1 P,
  XMLTABLE('purchaseOrder[@poid="2001"]'
  PASSING P.INFO
  COLUMNS "@poid" XML,
    "name" XML PATH 'shipTo/name',
    "street" XML PATH 'shipTo/street',
    "state" XML PATH 'shipTo/state',
    "zip" XML PATH 'shipTo/zip',
    "items" XML
  ) O(poid, name, street, state, zip, items)

```

Note: For XML data stored intact in the db2xml.xmlclob column, the queries have to be rewritten completely.

Customer information

Event though the structure of the table has not changed, you are now using an annotated XML schema to shred the incoming XML. Unlike the RDB node DAD files, annotated XML schemas cannot publish the shredded document. So instead, you create a SQL/XML call to publish the customer information.

Listing 15. Produce information for a customer

```

SELECT XMLSERIALIZE ( CONTENT
  XMLELEMENT ( NAME "customer",
    XMLATTRIBUTES (e.cid AS "id"),
    XMLELEMENT (NAME "name", e.name),
    XMLELEMENT (NAME "email", e.email),
    XMLELEMENT (NAME "zip", e.zip)
  )
AS CLOB(7200)) AS CINFO FROM customer e where cid=1001

```

Customer list

To get a mailing list of all the customers from the customer table, since you are already using SQL/XML functions, not much changes. The only difference is that you no longer need to convert the XML to CLOB, since now the JDBC/CLI drivers can understand XML type.

Listing 16. Produce customer list

```
SELECT XMLELEMENT ( NAME "customer",
XMLATTRIBUTES (e.cid AS "id"),
XMLELEMENT (NAME "name", e.name),
XMLELEMENT (NAME "email", e.email),
XMLELEMENT (NAME "zip", e.zip)
)
FROM customer e;
```

Conclusion

DB2 9 for z/OS introduces a new powerful and yet simple environment for storing, indexing, querying, and publishing both relational and XML data. In order to take full advantage of this environment you need to migrate your tables, queries, and application code. This article walked you through a series of migration steps using an example scenario. The second article in this series also details the new XML features in DB2 9 and compares them to the existing ones in DB2 for z/OS, Version 8. The first article provides a Java stored procedure for doing sub-document updates. Since much of the migration process is manual, hopefully these articles help you in the migration process and also in making decisions on what course to take before and during the migration.

Acknowledgements

Thanks to Jason Cu, Manogari Simanjuntak and Li Chen for their help with converting the original article to z/OS.

Downloads

Description	Name	Size	Download method
Migrate scenario source code	Scenario_code.zip	9KB	HTTP

[Information about download methods](#)

Resources

Learn

- [DB2 for z/OS page area on developerWorks](#): Get the resources you need to advance your skills on DB2 for z/OS.
- ["XML application migration from DB2 for z/OS V8 to DB2 9, Part 1: Partial updates to XML documents in DB2 9 for z/OS"](#) (developerWorks, March 2008): Learn a method for performing partial updates to XML documents stored natively in DB2 9 for z/OS, using a stored procedure that's included as a download.
- ["XML application migration from DB2 for z/OS V8 to DB2 9, Part 2 : Comparing XML functionality in DB2 9 to DB2 V8"](#) (developerWorks, March 2008): In part two of a series, explore the XML functionality in DB2 9 for z/OS and understand the impact the new XML support has on migrating existing XML-based applications from DB2 for z/OS V8.
- [developerWorks Information Management zone](#): Learn more about Information Management. Find technical documentation, how-to articles, education, downloads, product information, and more.
- Stay current with [developerWorks technical events and webcasts](#).
- [Technology bookstore](#): Browse for books on these and other technical topics.

Get products and technologies

- Build your next development project with [IBM trial software](#), available for download directly from developerWorks.

Discuss

- [Participate in the discussion forum for this content](#).
- [DB2 for z/OS Exchange](#): Contribute to the DB2 for z/OS Exchange.
- Participate in [developerWorks blogs](#) and get involved in the developerWorks community.

About the authors

Hardeep Singh

Hardeep Singh is a member of the DB2 Native XML team. He is the architect for DB2 XML tooling and is responsible for XML Extender application migration to Viper. He has more than 21 years of industry experience.

John R. Shelton

John Shelton is a member of the DB2 XML Extender team