DB2 and XML

1. pureXML Overview

The DB2 9 pureXML technology unlocks the latent potential of XML data by providing simple efficient access to it with the same levels of security, integrity, and resiliency taken for granted with relational data, thereby allowing you to seamlessly integrate XML and relational data. DB2 9 stores XML data in a hierarchical structure that naturally reflects the structure of XML. This structure along with new indexing techniques allows DB2 to efficiently manage this data and eliminate the complex and time-consuming parsing typically required for XML.

DB2 9 includes an optional add-on feature pack, called pureXML. For all editions of DB2 9, except DB2 9 Express C, you have to purchase pureXML services in addition to a base data server license. For this reason, and more, it's important to not only understand the technology that pureXML provides, but the alternative solutions available to justify its use.

2. Before pureXML: How XML is Traditionally Stored

Before pureXML, XML could be stored in an XML-only database, shredded into a relational format, stored in a relational database intact within a large object (LOB), or stored on a file system (unfortunately the most common format). These options, and their respective trade-offs, are shown below:

2.1 The XML-Only Database

There have been a number of niche corporations that offer XML-only databases (for example, Tamino). While XML databases store XML as XML, their weakness is the only strength: they can only do that store XML. In other words, they create a silo of information which ultimately requires more work and engineering to integrate with your relational systems. As more and more business move towards XML, challenges around integration with the relational data (which isn't going away, rather, it will be complimented by XML) arise. An XML-only database ultimately adds unforeseen costs to an environment because different programming models, administrative skills sets, servers, maintenance plans, and more are implicitly tagged to such a data storage solution.

What's more, end users will experience different service levels since there is no capability to unify a service-level across separate serves. Database administrators (DBAs) live and breathe by their service-level agreements (SLAs), and the separation of data servers based on the semantic structure of the data is an obstacle to sustainable SLA adherence among other efficiencies.

2.2 Storing XML in a File System

Perhaps the most common way to store XML is on a file system. The biggest problem associated with storing critical XML data on a file system is that file systems can't offer the Atomic, Consistent, Isolated, and Durable (ACID) properties of a database. Consider the following questions:

- How can file systems handle multiple changes that impact each other?
- What is the richness of the built-in recovery mechanism in case of a failure? (What about point-in-time recovery and recovery point objectives?)
- How do you handle complex relationships between your data artifacts in a heterogeneous environment?
- What type of facilities exists for access parallelism?
- Is there an abstracted data access language like XQuery or SQL?

- How is access to the same data handled?

Furthermore, storing your XML on a file system and then interacting with it from your application requires a lot of hand coding and is prone to errors. Quite simply, the decision to store critical XML data on a file system forces you to accept the same risks and issues associated with storing critical data in Excel spreadsheets which is a serious issue associated with data integrity and quality.

2.3 XML Within a LOB in a Relational Database

One option provided by most of today's relational database vendors is to take the XML data, leave it as XML, and place it in a LOB column within a relational table. This solves a number of issues, yet presents new ones too.

When the XML is placed in a LOB, you indeed unify XML and the relational data. And in most cases, you can leverage what XML was designed for: flexibility, because you are not tightly bound to a relational schema within the LOB container. The problem with storing XML data in a LOB is that you pay a massive performance price for this flexibility because you have to parse the XML when it is required for query.

It should be noted that this storage method may work well for large documents that are intended to be read as a whole (for example a book chapter). However, by and large, XML is moving to facilitate transactional processing as evident by the growing number of standards that have emerged in support for this.

For example, the FiXML protocol is a standard XML-based markup language for financial transactions. Other vertically aligned XML-based markup languages include:

- Justice XML Data Dictionary (JXDD) and Justice XML Registry and Repository (JXRR) and their corresponding RapSheet.xsd, DriverHistory.xsd, and ArrestWarrent.xsd XML Schema Definition (XSD) documents
- Air Quality System Schema (Asbestos Demolition & Removal Schema)
- Health Level 7 (HL7) for patient management, diagnosis, treatments, prescriptions, and so on
- Interactive Financial Exchange (IFX) standard for trades, banking, consumer transactions, and so on
- ACORD standard for policy management such as underwriting, indemnity, claims, and so on
- IXRetail standard for inventory, customer transaction, and employee management
- and 1000s more...

As companies store massive orders in XML, and want to retrieve specific order details, they'll be forced to parse all of the orders with this storage mechanism. For example, assume a financial trading house is sent a block of stock trades on the hour for real time input into a fraud detection system. To maintain the XML, the data is placed in a LOB. If a specific request came to take a closer look at trades by Mr. X., the database manager would have parse all the trades in a block to find Mr. X's trades. Imagine this overhead if Mr. X. made a single trade in the first hour and the block contained 100,000 trades. That's a lot of overheard to locate a single transaction.

It quickly becomes obvious that while LOB storage of XML unifies the data, it presents a new issue, namely performance. There are others, depending on a vendor's implementation of XML storage, but this is the main issue you should be aware of.

2.4 XML Shredded to a Table in a Relational Database

Another option provided by most of today's relational database vendors is to take XML data and shred it into a relational table. In this context, shred means that there is some mapping that takes the data from the XML format and places it into a relational table. Consider the following example:

In the previous example you can see a single XML document has been shredded into two different relational tables: DEPARTMENT and EMPLOYEE. This is typical since in an online transaction processing (OLTP) application, performance is key; this example illustrates 3rd Normal Form (no repeating elements).

In this storage mechanism XML data is unified with its relational counterpart since it's actually stored with the relational data. Additionally, there is no parsing overheard since data is retrieved as regular relational data via SQL requests.

From a performance perspective, shredding XML to relational data seems to solve the aforementioned issues; however, there are a number of disadvantages with this approach as well. XML is all about flexibility. It's the key value proposition of this technology. Application developers (big proponents of XML) like the flexibility to alter a schema without the long process that's typically well understood by DBAs. Consider this example, if a new business rule was set such that a Human Resources department now required employee's cellular phone numbers for business emergencies, what ill-effects would such a change present XML data whose storage mechanism was to shred it to a set of relational tables?

First, the mapping (not shown in the figure) between the XML data and the relational tables would have to be updated, this could cause application churn, testing requirements, and more. Of course to maintain 3rd Normal Form you may actually be forced to create a new table for phone numbers and instantiate referential integrity rules to link them together as shown below:

You can see in this example that the flexibility promise associated with XML is quickly eroded when shredding is used for storage because a change to the XML Schema requires a change to the underlying relational schema. In addition, a change to the XML Schema requires a change to the mapping tier between the XML document and the relational database.

Also consider that when shredded, access to the XML data is via SQL which means that the newly engineered language specific to querying XML data (XQuery) must be translated into SQL in this model which can cause ineffectiveness, loss of function, and more.

Essentially, the shredding model solves the performance issue (though there is a performance penalty to rebuild the shredded data as XML), but presents major flexibility issues which is perhaps the most significant value proposition of XML.

3. Creating Tables with XML Columns

Once a database has been enabled for pureXML, you create tables that store XML data in these columns in the same manner in which you create tables with only relational data. One of the great flexibility benefits of pureXML is that it doesn't require supporting relational columns to store XML. You can choose to create a table that has a single, or multiple XML columns, a table that has a relational column with one or more XML columns, and so on.
Note: There are some features in DB2 that do not support pureXML. For example, the Database Partitioning Feature (DPF) and the Storage Optimization Feature (for deep row compression) do not support pureXML columns.
For example, any of the following CREATE TABLE statements are valid:
CREATE TABLE xml1 (id INTEGER NOT NULL,

```
customer XML, CONSTRAINT RESTRICTID PRIMARY KEY(ID));

CREATE TABLE xml2 (customer XML, order XML);
```

You can create tables with pureXML columns using either the DB2 CLP or the Control Center as shown below:

The XML data type integration extends beyond the DB2 tools and into popular integrated development environments (IDEs) like IBM Rational Application Developer, Microsoft Visual Studio 2005, Quest TOAD, and more:

This means that you can work with pureXML columns from whatever tool or facility you feel most comfortable. To follow the examples in this section, create the following table in the working example's MYXML database:

```
CREATE TABLE commuters (id INTEGER NOT NULL,
 customer XML, CONSTRAINT RESTRICTID PRIMARY KEY(ID));
```

This table will be used to store commuter information (based in XML) alongside an ID which represents the Web site's customer that is building the commuter community (the name of the commuters is purposely not entered into the database). Commuter information is retrieved from various customers that want to carpool to work and their information is received via a Web-form and passed to DB2 via XML – a typical Web application.

4. Inserting Data into XML Columns

Once you have a pureXML column defined in a table, you can work with it, for the most part, as you would work with a any table. This means that you may want to start by populating the pureXML column with an XML document or fragment. Consider the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<customerinfo xmlns="http://tempuri.org/XMLSchema.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://tempuri.org/XMLSchema.xsd
C:\Temp\QUESTExample\Customer.xsd">
        <CanadianAddress>
                <Address>434 Rory Road</Address>
                <City>Toronto</City>
                <Province>Ontario</Province>
                <PostalCode>ML51C7</PostalCode>
                <Country>Canada</Country>
        </CanadianAddress>
        <CanadianAddress>
                <Address>124 Seaboard Gate</Address>
                <City>Whitby</City>
                <Province>Ontario</Province>
                <PostalCode>L1N9C3</PostalCode>
                <Country>Canada</Country>
        </CanadianAddress>
</customerinfo>
```

Note that because this is a well formed XML document, you should be able to see it in a XML-enabled browser, such as Internet Explorer:

You can use the SQL INSERT statement to add this data to the MYXML table in the same manner that you would add it to a table that only had relational data. The data inserted into the pureXML column must be a well-formed XML document (or fragment), as defined by the XML 1.0 specification.
For example, to insert the working example XML document using the CLP, issue the following command:

```
INSERT INTO commuters VALUES (1,
 '<?xml version="1.0" encoding="UTF-8"?>
  <customerinfo xmlns="http://tempuri.org/XMLSchema.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://tempuri.org/XMLSchema.xsd
  C:\Temp\QUESTExample\Customer.xsd">
 <CanadianAddress>
<Address>434 Rory Road</Address><City>Toronto</City>
<Province>Ontario</Province>
<PostalCode>ML51C7</PostalCode><Country>Canada</Country>
 </CanadianAddress>
 <CanadianAddress>
<Address>124 Seaboard Gate</Address><City>Whitby</City>
<Province>Ontario</Province>
<PostalCode>L1N9C3</PostalCode><Country>Canada</Country>
  </CanadianAddress>
</customerinfo>')
```

Note that this data in now in the pureXML column:

Since pureXML provides you with the utmost flexibility when it comes to storing XML, you don't have to just store full XML documents as in the previous example. In the same table you could store a fragment of that document. For example, to add the XML document in the working example (one full <CanadianAddress> block), you could enter the following command:

```
INSERT INTO commuters VALUES (2,
  '<CanadianAddress>
     <Address>434 Rory Road</Address><City>Toronto</City>
     <Province>Ontario</Province>
     <PostalCode>ML51C7</PostalCode>
     <Country>Canada</Country>
  </CanadianAddress>')
```

Now the COMMUTERS table has two rows with XML data, one with a full XML document and the other with an XML fragment:

The previous figure visualizes the XML data in the COMMUTERS table using the new DB2 9 Developer Workbench, another tool that lets you graphically work with your XML data.
DB2 must store well-formed XML documents and fragments, though you have the option to have the XML validated or not against XML Schema Definition (XSD) documents.
If you tried to insert a non-well-formed XML document or fragment into a pureXML column you would receive an error:

In the previous example note that the <Address> element isn't closed properly it uses <Address> to close it instead of the proper </Address> element. Because this element isn't closed properly,

as per the standard, this fragment isn't considered to be well-formed and therefore the insert operation fails.
Finally, you should be aware that there are multiple options that you can use when using the INSERT command to insert XML data into a table.
Consider the following example:

```
INSERT INTO COMMUTERS VALUES
  (
  3,
  XMLPARSE
    (DOCUMENT '<?xml version="1.0" encoding="UTF-8"?>
<customerinfo xmlns="http://tempuri.org/XMLSchema.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://tempuri.org/XMLSchema.xsd
C:\Temp\QUESTExample\Customer.xsd">
        <CanadianAddress>
                <Address>434 Rory Road</Address>
                <City>Toronto</City>
                <Province>Ontario</Province>
                <PostalCode>ML51C7</PostalCode>
                <Country>Canada</Country>
        </CanadianAddress>
        </customerinfo>' PRESERVE WHITESPACE)
  ),
  (
  4,
  XMLPARSE
    (DOCUMENT '<?xml version="1.0" encoding="UTF-8"?>
<customerinfo xmlns="http://tempuri.org/XMLSchema.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://tempuri.org/XMLSchema.xsd
C:\Temp\QUESTExample\Customer.xsd">
                <CanadianAddress>
                <Address>124 Seaboard Gate</Address>
                <City>Whitby</City>
                <Province>Ontario</Province>
                <PostalCode>L1N9C3</PostalCode>
                <Country>Canada</Country>
        </CanadianAddress>
</customerinfo>')
  );
```

The previous INSERT statement illustrates the flexibility you have when inserting XML data into pureXML columns. First note that two separate XML documents were inserted in a single INSERT statement. Also note the use of various options that affect XML insert activity into pureXML columns; some of which include DOCUMENT, PRESERVE WHITESPACE, and so on. Others include the ability to validate the XML data against a registered XML schema definition document (the XMLVALIDATE option) and much more.
Note in the previous example that the first XML document (who's ID = 3 - the first column's data value in the COMMUTER table) specifically requests that whitespace be preserved in the on-disk format via the PRESERVE WHITESPACE option. In contrast, the second XML document doesn't make such a request and therefore boundary whitespace will be stripped (the default) from this document.

5. Selecting Data from XML Columns

You can use XQuery, SQL, SQL/XML, or any combination there-of, to retrieve data stored in pureXML columns as illustrated below:

In practice, depending on the data you want to retrieve, you'll use different interfaces to solve different problems; however this ultimately depends on your skill set. If your enterprise is well versed in all the XML data access methods, you'll be able to leverage varying approaches to data retrieval which will yield benefits that relate to development time, performance, efficiency, and more.
The easiest (yet perhaps most inefficient) method to retrieve data from a pureXML column is to solely use only SQL. However, generally more value can be delivered to an application that uses SQL/XML, XQuery, or some combination there-of.

6. Updating and Deleting in XML Columns

To update or delete entire XML documents or fragment in a pureXML column you can use traditional INSERT and UPDATE SQL statements. For example, the COMMUTERS table at this point may contain the following data:

If you wanted to update an entire XML document in a pureXML column, you could enter an UPDATE statement similar to the following:

```
UPDATE commuters SET customer = XMLPARSE

(DOCUMENT (

   '<?xml version="1.0" encoding="UTF-8"?>

<customerinfo xmlns="http://tempuri.org/XMLSchema.xsd"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://tempuri.org/XMLSchema.xsd

C:\Temp\QUESTExample\Customer.xsd">

       <CanadianAddress>

              <Address>688 Freemont Drive</Address>

              <City>Belleville</City>

              <Province>Ontario</Province>

              <PostalCode>K8V8C3</PostalCode>

              <Country>Canada</Country>

       </CanadianAddress>

       </customerinfo>'))

WHERE id=4
```

If you were to run the same query in the previous figure you would now see that the previous XML document assigned to ID=4 has been updated and contains the XML document above.

You delete rows with XML documents in the same manner as regular rows using the DELETE command:

```
DELETE FROM commuters WHERE id=4
```

If you wanted to completely remove an XML document or fragment from a pureXML column you would use the UPDATE command and set the value to NULL:

```
UPDATE commuters SET customer = null WHERE id=3
```

At the time when DB2 9 became generally available, the XQuery standard really only defined what its name implies: query. There was no ANSI standard method for updating or deleting fragments of an XML document. As you've seen in the previous examples, you can manipulate data in existing pureXML columns, but only when you work on the entire document or fragment in a single operation. As XML proliferates into the mainstream, one should expect for more and more operations to occur transactionally at child levels in the hierarchy. For example, a commuter in the working example could move to a new location in the same city this would likely require the XML data be updated, but not all elements (you don't need to update <Country> or <Province>). If you considered these types of operations across a large number of documents (within a single column, or spread across multiple pureXML columns) the overhead of having to work with the entire document for fragment could be quite high.

Because there was no defined standard for updating parts of an XML document or fragment when DB2 9 became generally available, a decision was made to provide the DB2XMLFUNCTIONS.XMLUPDATE stored procedure to provide the capability to update portions of an XML document or fragment in the interim.

Although the source for this routine is provided for you, you are required to build and install this stored procedure (it's not considered part of the pureXML support, rather a work-around method until the XQuery standard evolves to support such activity).