Replacing DB2 XML Extender with integrated IBM DB2 for i XML capabilities

Learn how to use DB2 integrated XML functionality to replace solutions based on DB2 XML Extender

Nick Lawrence
Kent Milligan
Yi Yuan

IBM Systems and Technology Group

February 2013

IBM

Table of contents

Abstract

This white paper explores using the new integrated XML features in IBM DB2 for i 7.1 as a replacement for the XML-related functions and data types provided by the priced DB2 XML Extender option, which is part of DB2 Extenders for IBM i licensed product (5761DE1 and 5770DE1). The paper reviews the differences between the DB2 XML Extender and the integrated XML support. A fictional company's application is used as a mechanism to compare the integrated XML functionality with the capabilities provided by XML Extender. The application-based comparison can provide programmers a much better understanding of the integrated XML support.

Introduction

IBM® DB2® for i 7.1 provides integrated support for XML, allowing application developers to more easily store and process XML data. In addition to the XML support available in the initial release of DB2 for i 7.1, significant enhancements have been made available since the initial release; customers who need to make the most of the XML support should apply the DB2 Group PTF SF99701 Level 14. IBM encourages customers to regularly install the most recent version of the DB2 Group PTF to avoid problem rediscovery and to be able to use new technologies. A link to a list of the most recent IBM i technology updates is included in the references section.

The built-in support includes a new XML built-in data type. Also available are built-in functions and procedures for composing XML documents from relational data, XML schema validation, shredding XML data into relational tables, style sheet transformation, and XML query capabilities. Additionally, advanced text searches can be performed against XML documents stored in DB2 for i databases using the IBM OmniFind® Text Search Server product (5733-OMF).

In prior releases, applications that needed to integrate XML with DB2 for i had to use the DB2 XML Extender support. The DB2 XML Extender support is part of the IBM DB2 Extenders™ licensed program product, which was first made available in the V5R1 release. The DB2 Extenders product was a chargeable feature that had to be purchased separately from IBM. In contrast, the new XML support included in DB2 for i 7.1 is integrated into the base IBM i operating system, requiring no additional charge. In addition, the integrated XML support provides a wider range of functionality and is more consistent with rest of the DB2 product family, the World Wide Web Consortium (W3C) XML standards, and the SQL/XML standard (ISO 9075-14 2011). Customers are encouraged to modify their applications to use the new support delivered in DB2 for i 7.1.

This white paper provides a good description of the new function for those customers who need to replace XML Extender with the built-in XML data type and built-in functions. This paper includes an overview of the XML data type as a replacement for the user-defined data types offered by the XML Extender product. The overview is followed by an example scenario (involving a fictional company) that is used to illustrate how common use cases for XML might be implemented using the built-in capabilities, as opposed to using XML Extender. Together, these comparisons can depict what considerations are necessary when moving from a solution that employs XML Extender to a solution based on the built-in XML data type.

Overview of the XML data type

It is necessary to first understand some important differences between the user-defined XML types provided by the DB2 XML Extender and the built-in XML data type available in DB2 for i 7.1.

User-defined XML types in DB2 XML Extender

XML Extender defines three user-defined types (UDTs) for XML: XMLVARCHAR, XMLCLOB, and XMLFILE. Using these types, developers can define whether the XML value is stored in DB2 as a VARCHAR, CLOB, or in an external IFS stream file (known to XML Extender by an IFS path name). A table with user-defined XML typed columns might have been created, as shown in Listing 1.

```
CREATE TABLE Extender_sample (
    col1 DB2XML.XMLVARCHAR,
    col2 DB2XML.XMLCLOB,
    col3 DB2XML.XMLFILE
)
```

Listing 1: Column with XML Extender UDTs

Each of the user-defined types is based on character data types; meaning that values of these types contain serialized XML documents (or in the case of XMLFILE, a reference to a file with a serialized XML document).

The term serialized XML document means that the XML is represented as text data that can be transmitted between applications. XML parsers are able to convert serialized documents into whatever representation is optimal for the needs of the application. Applications can choose to store and process XML data using the most appropriate data structure for the type of processing being performed, and exchange XML data with other applications as serialized text data.

When constructing a serialized XML document, an important concept to be aware of is that the author has a variety of choices for representing the same logical XML ideas. For example, if an element has no content, it can be written as either <element></element> or <element/>. Special characters provide another example; authors have several ways to inform an XML parser that special characters (such as **<**) are ordinary characters and not part of the XML markup.

As XML Extender defines the user-defined XML data types based on the serialized document, XML documents are stored and retrieved exactly the way they are authored when received by DB2 for i. No parsing or validation of the XML document is performed, and DB2 for i can handle the XML data as simple character data, rather than a complex data type.

A user-defined type that represents serialized XML values is a disadvantage because a UDT offers no built-in understanding of the structure and relationships defined within the XML document. This is a topic that is discussed in more detail throughout this paper.

Built-in XML type

In contrast to a user-defined type, the built-in XML type that exists in DB2 for i 7.1 describes the structure and data within an XML document in terms of an XML Data Model (XDM). DB2 for i defines the XDM to be consistent with the industry-standard XPath 2.0 data model. The XPath 2.0 data model is an abstract

representation of XML that is defined by the W3C. Using the XML Data Model as the basis for describing XML values allows DB2 for i to keep the data model consistent between SQL and XML technologies.

Rather than a character string, the XDM describes an XML document in terms of a tree of nodes (document, element, attribute, text node, and so on). Listing 2 illustrates a sample XML document, and Figure 1 illustrates how this document is described using the XML Data Model.

```
<product xmlns="http://posample.org"
           pid="100-101-01">
 <description>
   <name>Snow Shovel, Deluxe 24</name>
   <details>A Deluxe Snow Shovel,
              24 inches wide, ergonomic curved handle with D-Grip
   </details>
   <price>19.99</price>
   <weight>2 kg</weight>
 </description>
</product>
```

Listing 2: XML document

In addition to nodes, the XDM also includes atomic values (xs:string, xs:integer, xs:double, xs:date, xs:dateTime, xs:time, and so on) which are used within built-in functions and procedures for processing XML. Nodes in an XML document can be converted to atomic values during XML operations. For example, the text node **19.99** inside the price element in Listing 2 may need to be converted to xs:double for arithmetic comparison. DB2 does not associate type annotations with each node, but built-in functions know how to convert the untyped data in the document to a specific atomic type when necessary.

For a complete explanation of the data model, refer to the “Resources” section for a link to the topic in the IBM i Information Center.

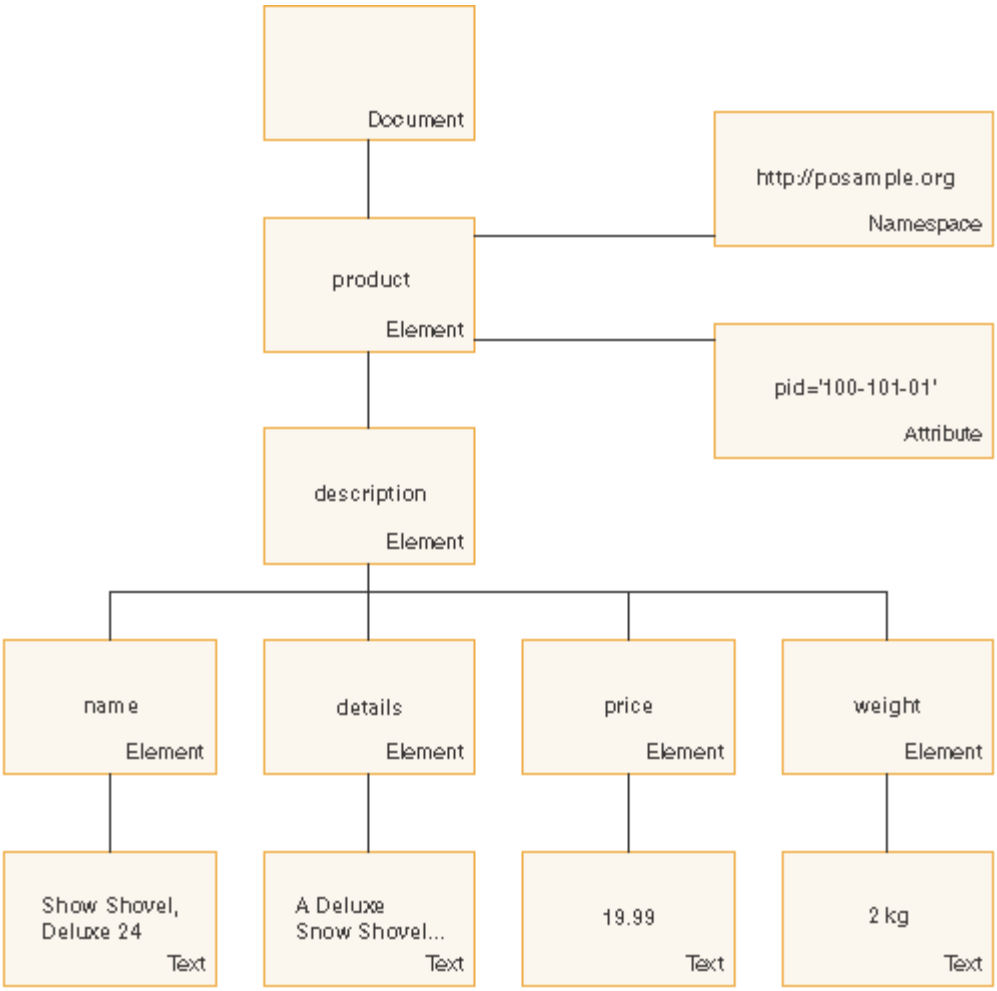

Figure 1: XML data model

It is important to understand that in SQL, each XML value is an instance of the XDM. The idea that an XML value is described by a separate data model is a big difference from other SQL types. XML values can be further divided up into nodes, atomic values, and relationships, while most other SQL data types represent a single value that cannot be further divided.

Describing an XML value in terms of the XDM means that the built-in XML type is not a synonym for a serialized XML document that can be exchanged with an application as plain text. In other words, in DB2 for i, the XML data type is neither a character, binary, or graphic type nor a path name to a file on the file system that contains the XML document. A table with XML columns (as shown in Listing 3 might be created.

```
CREATE TABLE sample (col1 XML, col2 XML, col3 XML)
```

Listing 3: Column with built-in XML type

Unlike the CREATE TABLE statement in Listing 1 the built-in XML data type does not define the storage mechanism that DB2 should use for the XML data. Not being defined to be based on a character, graphic, or binary data type means that the built-in XML type does not guarantee that the XML data is physically

stored as a serialized XML document. What is guaranteed is that the XML value can be serialized to a character, graphic, or binary data type when necessary.

As the storage and representation of the XML type is not determined by the application, the XML data type does not have an associated maximum length attribute. Be that as it may, DB2 for i restricts the maximum length of a serialized XML value (in bytes) to 2 147 483 647. The maximum length of an XML schema document is (in bytes) 2 147 483 647.

Another effect of not describing the built-in XML data type in terms of serialized data is that the XML document is not preserved exactly the way it was originally authored. As an example, an element that is originally authored and stored as <element></element> appears as <element/> when retrieved from DB2 for i. In terms of the XML Data Model, there is no distinction between these authoring decisions, and therefore, DB2 for i does not need to retain this information. For more information on this topic, refer to the link in the “Resources” section on the differences between storage and retrieval of XML values.

DB2 for i provides built-in functions and procedures for creating and working with XML values. These are designed to be consistent with the XDM, and effectively encapsulate the storage and internal representation of XML data from the application’s logic.

Creating an XML value using the XMLPARSE function

With XML Extender, an instance of the XMLVARCHAR or XMLCLOB data type can be created from character data with a simple SQL CAST because both the source and target type are based on character data. Listing 4 shows how to construct an XMLCLOB value from a VARCHAR value.

```
CREATE VARIABLE serialized_doc VARCHAR(2000) CCSID 1208;
SET serialized_doc = ‘<doc> this is an XML document </doc>’;

CREATE VARIABLE xmlclob_value DB2XML.XMLCLOB;
SET xmlclob_value = CAST(serialized_doc as DB2XML.XMLCLOB);
```

Listing 4: Create an XMLCLOB value

It is important to understand that Listing 4 demonstrates a cast between two compatible SQL types that are each defined for character data, rather than a parse of a serialized XML document into something described by the XDM. No checks are performed during the cast to ensure that the XML document is syntactically correct. An application might encounter errors when processing values that are of these user-defined types, if the original XML document was not a well formed XML document.

In comparison, creating an instance of the built-in XML type from serialized data must be done with the XMLPARSE function. XMLPARSE converts a serialized XML document into an XML value that is described by the XDM. Listing 5 shows how to construct an XML value using a serialized XML document stored in the VARCHAR variable defined in by Listing 4.

```
CREATE VARIABLE xmlvalue XML;
SET xmlvalue =
  XMLPARSE(DOCUMENT serialized_doc);
```

Listing 5: Create an XML value

The DOCUMENT keyword is required for XMLPARSE, and tells DB2 that the resulting XML value must be a *well-formed XML document*, or in other words, the document must have a single root element. The result of the XMLPARSE function is an instance of the XML data type.

If the serialized document that is provided to XMLPARSE is not a syntactically legal XML document, XMLPARSE will fail with an error. This prevents problems later on where an application might encounter errors that result from the original XML document not being legal.

Serializing XML data

The process of converting XML data from the XDM into a character value that can be transmitted to an application is called *serializing*. In SQL, serializing is necessary when we need to convert an XML value to a character, graphic, or binary value (VARCHAR, CLOB, BLOB, and so on) that can be processed by the application.

With XML Extender, an instance of the XMLCLOB or XMLVARCHAR types can be converted to a CLOB or VARCHAR type by performing a simple cast of the user-defined XML type to the target SQL CLOB or VARCHAR type as illustrated in Listing 6.

```
SET serialized_doc = CAST(xmlclob_value AS CLOB(1G))
```

Listing 6: Cast of XMLCOB to VARCHAR

Listing 7 shows the result of the CAST expression.

```
<doc> this is an XML document </doc>
```

Listing 7: Result of CAST

With the new XML type, the XMLSERIALIZE function must be used to perform this XML to character conversion. An SQL CAST expression cannot be used to convert an XML value to a character value. Listing 8 shows an example of the XMLSERIALIZE function in action.

```
SET serialized_doc =
   XMLSERIALIZE(xmlvalue AS VARCHAR(2000) CCSID 1208
                  INCLUDING XMLDECLARATION)
```

Listing 8: Serialize XML data

The INCLUDING XMLDECLARATION clause is optional and tells DB2 to include the encoding declaration on the result string. The result shown in Listing 9 will be a VARCHAR(2000) value in the UTF-8 character set.

```
<?xml version="1.0" encoding="UTF-8"?>
<doc> this is an XML document </doc>
```

Listing 9: Serialized XML document

Implicit XMLPARSE and XMLSERIALIZE

Using an XML Extender type for XML values means that the SQL rules for implicit type conversions of UDTs apply. The built-in XML type provides similar functionality by implicitly parsing and serializing in certain situations.

Listing 10 provides an example of how implicit type conversion works during an insert when using XML Extender. The variable *serialized_doc* has a source type of VARCHAR, and can therefore be promoted to CLOB. The column *col1* is of type XMLCLOB, which has a source type of CLOB. Consequently, the insert of a CLOB into an XMLCLOB column in Listing 10 is valid, and easier to read than an explicit cast from VARCHAR to XMLCLOB.

The INSERT statement in Listing 10 has no XML awareness. This INSERT statement is an insert of a character value into a column with a user-defined type that has a compatible source type. There is no implicit parsing or validation of the XML value. DB2 for i will not prevent invalid XML data from being stored in the column. This can cause an application to encounter errors when processing the invalid XML later on.

```
CREATE TABLE extenders(col1 DB2XML.XMLCLOB);

DECLARE serialized_doc VARCHAR(2000) CCSID 37;

SET serialized_doc =
  '<?xml version="1.0" encoding="ibm-037"?><doc> my document </doc>';

INSERT INTO extenders (col1) VALUES(serialized_doc);
```

Listing 10: Implicit type conversion using XML Extender

Listing 11 shows the serialization scenario using XML Extender. In this example, the column *col1* is defined using the XML Extender XMLCLOB UDT. The XMLCLOB is being fetched into a CLOB variable. As both have the same source type, the fetch of an XMLCLOB value into a CLOB variable in Listing 11 is valid.

```
DECLARE serialized_doc CLOB(2G) CCSID 1208;
DECLARE col1_cursor CURSOR FOR SELECT col1 FROM extenders;
OPEN col1_cursor;
FETCH col1_cursor INTO serialized_doc;
```

Listing 11: Implicit conversion during fetch from an XMLCLOB column

The value of the variable serialized_doc, after the FETCH statement is shown in Listing 12. The result looks innocent enough, but it demonstrates an important problem. As XML Extender deals only with XML as strings of character data, the encoding declaration in the document was not updated and still indicates that the data is in CCSID 37, when in reality the CLOB variable has a CCSID of 1208. This causes an error to occur if the CCSID 1208 CLOB value is passed to an application and parsed as XML.

```
<?xml version="1.0" encoding="ibm-037"?><doc> my document </doc>
```

Listing 12: Value of serialized_doc after fetch

The built-in XML type provides the ability to parse and serialize data implicitly in situations similar to the implicit casting that is supported for the XML Extender UDTs. Using the built-in type's implicit parsing and serialization simplifies the SQL statement, guarantees that XML values are legal, and also helps to avoid the encoding issue that was demonstrated in Listing 12. For these reasons, the best practice is to avoid explicitly coding XMLPARSE and XMLSERIALIZE in SQL statements where XML data is exchanged with an application.

DB2 for i performs an implicit XMLPARSE when a character, graphic, or binary type is assigned to an XML column in an INSERT, UPDATE, or MERGE statement. Listing 13 shows a valid insert operation of a VARCHAR value into an XML column. DB2 for i implicitly performs the XMLPARSE function on the XML

value stored in the serialized_doc value. Similar to the explicit XMLPARSE function, if the serialized value is not a legal XML document, an SQL error occurs at the time the INSERT statement is ran.

```
CREATE TABLE sample (col1 XML);

DECLARE serialized_doc VARCHAR(2000) CCSID 37;
SET serialized_doc =
  '<?xml version="1.0" encoding="ibm-037"?><doc> my document </doc>';
INSERT INTO sample (col1)
   VALUES(serialized_doc);
```

Listing 13: Implicit XMLPARSE during an SQL INSERT

When a query returns an XML value and a non-XML data type is needed, an XMLSERIALIZE is implicitly performed. Implicit serialization is often preferred when XML data is being retrieved by the application.

Listing 14 shows an example of an implicit XMLSERIALIZE. Assuming that *sample* is the table created in Listing 13, XML data is being retrieved into a non-XML data type (CLOB) during the fetch from a cursor col1_cursor. An implicit XMLSERIALIZE will be performed in this case. Scenarios where the XML value is fetched into a non-XML variable are a common occurrence due to the fact that host languages such as C, C++, RPG, COBOL, and Java™ do not support a native XML type.

```
DECLARE result CLOB CCSID 1208;
DECLARE col1_cursor CURSOR FOR SELECT col1 FROM sample;
OPEN col1_cursor;
FETCH col1_cursor INTO result;
```

Listing 14: Implicit XMLSERIALIZE

The value of the result variable after fetch in Listing 14 is shown in Listing 15. The serialization process has serialized the XML data (using UTF-8 as a character set), and updated the encoding declaration of the XML document to the correct CCSID (UTF-8). In Listing 11, XML Extender could not update the encoding declaration, because the user-defined type was based on character data.

```
<?xml version="1.0" encoding="UTF-8"?>
<doc> my document </doc>
```

Listing 15: Results from implicit XMLSERIALIZE

The maintenance of the encoding declaration is a major reason why implicit serialization is often preferred when using the built-in XML type. After the XML value has been serialized, any CCSID translation that is performed on the value might cause the encoding declaration to no longer be the same as the actual encoding. Implicit serialization allows the application to decide the type and character set encoding that it will obtain for XML data, and avoids CCSID translations of the serialized value.

There is one more circumstance where DB2 for i can implicitly parse an XML value. When DB2 for i receives an XML document through the host variable or parameter marker, and the DB2 target type is XML, it implicitly invokes the XMLPARSE function.

The SQL statement in Listing 16 is a valid way to provide DB2 for i with XML data. DB2 for i interprets the typed parameter marker as a promise to provide XML data. Providing a character, graphic, or binary value at runtime causes DB2 for i to interpret the value as a serialized XML document, causing DB2 for i to perform an implicit XMLPARSE invocation.

```
SET xml_value = CAST(? AS XML)
```

Listing 16: Implicit XMLPARSE for parameter markers

Supporting the typed parameter marker syntax for the built-in XML type allows the application to determine what data type and encoding will be used when supplying the serialized document.

Using typed parameter markers is also possible with user-defined types, including the user-defined types provided by XML Extender. However, similar to the problems with serialization, XML Extender would have issues if the document being supplied to DB2 had a different encoding specified in the document than the character set associated with the parameter marker's XML Extender type.

Handling of boundary white space when parsing XML data

To improve the readability of XML by users, XML documents often contain text nodes that consist only of white space characters (spaces, carriage returns, line feeds and tabs). These text nodes are also referred to as boundary white space because they are normally used to visually separate the end of one element from the beginning of the next, or to emphasize a new nesting level within the document. For example, the serialized XML in Listing 2 has carriage return characters and indenting spaces for each new level of nesting, and after the end of each element. While these white space text nodes can be represented in the XML Data Model, they are seldom interesting to an application. For instance, the boundary white space in Listing 2 was not included in the diagram of the XML Data Model (shown in Figure 1), and the omission is unlikely to be a problem.

When XML data is stored in DB2 using one of the UDTs provided by XML Extender, there is no way for DB2 to detect boundary white space. Thus, the boundary white space is always preserved, even though it is normally irrelevant and wastes storage. The built-in XML data type is more flexible, and by default, stores XML values with this insignificant white space data removed. Although the existence or non-existence of boundary white space is not usually significant to an application, it is important to understand the white space options that are available when parsing XML data, especially if the boundary white space needs to be preserved.

When parsing serialized data into an XML value using the XMLPARSE function, the default behavior is to remove (or strip) boundary white space. Although stripping white space is the default behavior, it is possible for clarity to explicitly specify that boundary white space should be stripped by adding the STRIP WHITESPACE option to the XMLPARSE function invocation, as shown in Listing 17.

```
VALUES XMLSERIALIZE(
 XMLPARSE(DOCUMENT
 '<product>
   <description>
    <details>A Deluxe Snow Shovel,
              24 inches wide
    </details>
   </description>
  </product>'
 STRIP WHITESPACE )
AS VARCHAR(2000))
```

Listing 17: XMLPARSE with STRIP WHITESPACE option

The results of the XMLPARSE are shown in Listing 18. It is essential to understand that only the text nodes that contain all whitespace characters are removed. The white space characters that are contained within a text node that has significant characters (such as the text node within the details element) are preserved.

```
<product><description><details>A Deluxe Snow Shovel,
             24 inches wide
   </details></description></product>
```

Listing 18: XML document with white space stripped

If the boundary white space is relevant, the PRESERVE WHITESPACE option can be used to prevent XMLPARSE from removing these text nodes, as shown in Listing 19.

```
VALUES XMLSERIALIZE(
 XMLPARSE(DOCUMENT
 '<product>
   <description>
    <details>A Deluxe Snow Shovel,
              24 inches wide
    </details>
   </description>
  </product>'
PRESERVE WHITESPACE )
AS VARCHAR(2000))
```

Listing 19: XMLPARSE with PRESERVE WHITESPACE option

When serialized XML data is implicitly parsed into an XML value, the handling of boundary white space is controlled by the CURRENT IMPLICIT XMLPARSE OPTION special register. The initial value of this special register is STRIP WHITESPACE. Listing 20 shows an SQL statement that changes the value of the special register so that boundary white space is preserved.

```
SET CURRENT IMPLICIT XMLPARSE OPTION = PRESERVE WHITESPACE
```

Listing 20: Set the IMPLICIT XMLPARSE OPTION special register

Often, if boundary white space is significant, it is significant only for a smaller portion of an XML document. In these cases, the easiest solution might be to override the white space option that is used for an implicit or explicit XMLPARSE operation. This is accomplished by providing an **xml:space** attribute on the element where the white space option needs to be overridden. The **xml:space** attribute is defined by the W3C XML standard which means that it can be understood by other compliant XML parsers, if the document is transmitted to an application. In Listing 21, the text nodes for the boundary white space

before and after the start of the **b** element are preserved, regardless of the white space option used for the implicit or explicit XMLPARSE operation.

```
<doc> <a xml:space="preserve"> <b> <c>c</c>b </b></a> </doc>
```

Listing 21: XML document with xml:space attribute

XML as an SQL parameter of an external routine

The XML Extender user-defined types can be used for parameter definitions of an external user-defined function or stored procedure. External functions and procedures are written in a host language such as C, C++, COBOL, or RPG. As the host languages do not understand user-defined types, the values are passed to the application using the same format as the source type for the user-defined type. Listing 22 contains an SQL CREATE PROCEDURE statement which registers the specified RPG service program (UR_LIB/UR_SRVPGM) as an external stored procedure. The input parameter is defined with the XMLVARCHAR type.

```
CREATE PROCEDURE writexml(IN DB2XML.XMLVARCHAR)
LANGUAGE RPGLE
PARAMETER STYLE SQL
NO SQL
EXTERNAL NAME 'UR_LIB/UR_SRVPGM(WRITEXML)'
```

Listing 22: SQL CREATE PROCEDURE statement (external RPG) using XML Extender data type

Listing 23 shows the source code for the RPG procedure that is used for the external procedure defined in Listing 22. This code performs the minor task of storing the input data in a stream file. Note that the type of the XML input parameter *myDocument* is a varying character array. A character array is the RPG equivalent of the SQL VARCHAR type, which is the source data type that XMLVARCHAR user-defined type is based on.

```
     H NOMAIN
      * Include IFS file apis
      /COPY QSYSINC/QRPGLESRC,IFS

      ********************************************
      * Procedure prototype
      ********************************************
     D WRITEXML        PR
     D                               3000A   varying
     D                                  5I 0
     D                                  5A
     D                                517A   varying
     D                                128A   varying
     D                               1000A   varying


      ******************************************
      *Procedure Implementation
      ******************************************
     P WRITEXML        B                   EXPORT
     D                 PI
     D  myDocument                   3000A   varying
     D  myDocNull                       5I 0
     D  oSQLState                       5A
     D  qualName                      517A   varying
     D  specificName                  128A   varying
     D  oMessage                     1000A   varying

      ************************************************************
      *Local variables
      ************************************************************
     D outputFile      S              28A    varying
     D                                       INZ('/home/ntl/out_xml.xml')

     D
     D fd              S              10I 0
     D rc              S              10I 0

      /FREE
        // open file, mode 384 is octal 600 (user +rw)
        EVAL fd = open(outputFile: O_CREAT+O_WRONLY: 384 ) ;
        // write data
        EVAL rc = write(fd: %addr(myDocument)+2:%len(myDocument));

        // close file
        EVAL rc = close(fd);
        // indicate all OK
        EVAL oSQLSTATE = '00000';
        // program complete
       EVAL *INLR = *ON;
       RETURN;
      /END-FREE
     P                 E
```

Listing 23: RPG procedure source code using an XML input parameter

Despite the commonality between the VARCHAR and XMLVARCHAR types within the external routine, the stored procedure invocation must pass an XMLVARCHAR type for the input parameter. The user-defined type's casting function can be used to convert a VARCHAR value into an XMLVARCHAR value.

Listing 24 shows a typical call that uses a casting function in this manner. This casting only changes the data type for the purpose of SQL data type analysis; casting between types does not verify that the VARCHAR value is a syntactically valid XML document.

```
CALL writexml( DB2XML.XMLVARCHAR('<doc> This is My XML Document </doc>') )
```

Listing 24: SQL procedure call using XML Extender types

When using the new built-in XML type, the CREATE PROCEDURE statement must explicitly specify how the XML data will be passed into the external procedure. This is accomplished by using the XML cast syntax. In Listing 25, the XML input parameter definition includes the AS clause to indicate to DB2 for i that the external procedure expects the XML data to be provided in the format of an SQL VARCHAR string with a length of 3000 characters and CCSID value of 37. DB2 will serialize the XML document to this data type before providing the value to the stored procedure. The source code for the RPG procedure can remain the same as the original code shown in Listing 24.

```
CREATE PROCEDURE writexml_xml(IN XML AS VARCHAR(3000) CCSID 37)
LANGUAGE RPGLE
PARAMETER STYLE SQL
NO SQL
EXTERNAL NAME 'UR_LIB/UR_SRVPGM(WRITEXML)'
```

Listing 25: Creating a procedure using the built-in XML type and XML cast syntax

The input parameter must be an XML value on the stored procedure invocation. If the parameter is a different SQL type that contains serialized XML data, it is necessary to use the XMLPARSE function to convert the data to an XML value. Listing 26 shows how an XML value created by the XMLPARSE function can be included in a procedure call.

```
CALL xmltest.writexml_xml(
  XMLPARSE(DOCUMENT '<doc> This is My new XML document </doc>' )
)
```

Listing 26: Procedure invocation with a parameter using the built-in XML type

With the XML Extender support, the XMLCLOB data type can be passed as a locator for improved performance. The built-in XML data type also supports locators. Listing 27 shows a trivial C function to return a CLOB locator that references a CLOB value containing serialized XML data.

```
#include <stdlib.h>
#include <string.h>
#include <sql.h>
#include <sqludf.h>

#define MY_XML_DOC "<example> this is an example </example>"

int build_xml_doc(
           udf_locator  * lob_output,
           short *         lob_output_null,
           char  *          sqlstate,
           char  *          funcname,
           char  *          specname,
           char  *          msgtext)
{
int rc;
short ccsid = 37;
long bytes_appended;

// create lob and append output
rc = sqludf_create_locator_with_ccsid(SQL_TYP_CLOB,
                                                  ccsid,
                                                  &lob_output);
sqludf_append(lob_output,
              MY_XML_DOC,
              strlen(MY_XML_DOC),
              &bytes_appended);

// SQLSTATE is OK
memcpy(sqlstate, "00000", 5);
// result is not null
*lob_output_null = 0;

return 0;
}
```

Listing 27: C program to return XML data as a locator

Listing 28 demonstrates how an SQL CREATE FUNCTION statement can be written using XML Extender user-defined XMLCLOB type. The result type of the SQL function is a locator for an XMLCLOB value.

```
CREATE FUNCTION this_returns_xmlclob()
RETURNS DB2XML.XMLCLOB AS LOCATOR
LANGUAGE C
PARAMETER STYLE SQL
NO SQL
EXTERNAL NAME 'UR_LIB/UR_SRVPGM(build_xml_doc)'
```

Listing 28: External function that returns an XMLCLOB locator

Listing 29 shows how to create the function so that it returns XML data using a locator. DB2 for i parses the data referenced by the locator into an instance of the XML type when the locator is returned to the invoker. The source code in Listing 27 does not need to change.

```
CREATE FUNCTION this_returns_xml()
RETURNS XML AS LOCATOR
LANGUAGE C
PARAMETER STYLE SQL
NO SQL
EXTERNAL NAME 'UR_LIB/UR_SRVPGM(build_xml_doc)'
```

Listing 29: External function that returns an XML locator

XML host variables

Host languages that support embedded SQL are not aware of the user-defined types defined by XML Extender, but they are able to process the data in the application by declaring a host variable that matches the source data type of the UDT. For example, a host variable that needs to store an XMLCLOB value should be declared in the host language with an SQL CLOB data type.

The XML data type is not based on a source type and no IBM i host languages support a built-in XML data type. To resolve this difficulty, the DB2 for i SQL precompilers have been enhanced to support the new SQL XML data types which specify what data type should be used for XML values within the host language.

Listing 30 is an RPG program that demonstrates the differences between host variables that represent the user-defined types from XML Extender and host variables that represent the built-in XML type available starting with the IBM i 7.1 release. This example assumes that a stored procedure named example_using_extender has been created with an XMLCLOB input parameter and a procedure named example_using_xml has been created with an input parameter declared with built-in XML data type.

As CLOB is the source type for the XMLCLOB UDT, the host variable ExtendersXV is declared with the CLOB data type. The CLOB data type for the ExtendersXV variable is defined using the keyword, SQLTYPE(CLOB:3000). This keyword directs the SQL precompiler to create a data structure that includes the LEN (length) and DATA subfields that RPG statements can refer to.

In order to satisfy the data type requirements of the SQL language, a casting function is sometimes needed when the host variable is used in embedded SQL statements. The construction of a XMLCLOB value from a CLOB value can be seen when calling the example_using_extender procedure. As discussed earlier, this casting is necessary because the stored procedure requires an XMLCLOB value to be passed rather than a CLOB value.

Host variables that represent built-in XML values are declared so that the SQL type of the host variable is the built-in XML data type. The type declaration must include an SQL character or binary type that will be used as the format for representing the serialized XML data within the host variable. When the host variable is used in an SQL statement, DB2 for i will parse or serialize the data to and from an instance of the built-in XML type.

In the example program, the variable BuiltinXV is declared using the keyword SQLTYPE(XML_CLOB:3000). This declaration tells the SQL precompiler to create a character large object (CLOB) data structure where RPG stores and manipulates the serialized XML data. Although RPG will process the XML document as serialized data in a CLOB structure, the data type of the host variable when referenced on SQL statements will be XML. When the XML_CLOB data is provided as a host variable on

a call to the example_using_xml stored procedure, DB2 for i will parse the data in the builtinXV variable into an XML value.

As the data type of the BuiltinXV variable in SQL is XML, parsing the serialized data into an instance of the XML data type happens implicitly when the host variable's data is used in an SQL statement. Parsing BuiltinXV with an explicit invocation of the XMLPARSE function is not valid because the SQL data type of the host variable is XML, rather than a character or binary type.

Although not shown in this example, an automatic serialize is also possible; if an XML_CLOB host variable is used as the target of a FETCH, SELECT INTO, or VALUES INTO, DB2 for i will serialize the XML document into the host variable for use by the application.

```
H MAIN(PGM1) DFTACTGRP(*NO) ACTGRP(*NEW)
 ****************************************************
 * Procedure Prototype
 ****************************************************
DPGM1             PR                  EXTPGM('PGM1')
 ****************************************************
 *Serialized XML Document
 ****************************************************
D Serialized_XML  S           3000A   varying
D                                     INZ('<DOC/>')
 ****************************************************
 * Host variables
 ****************************************************
 *Extenders Version for XMLCLOB
D ExtendersXV     S                   SQLTYPE(CLOB:3000)
C/EXEC SQL
C+ DECLARE :ExtendersXV VARIABLE CCSID 37
C/END-EXEC

 *Built-in Version of XML Type
D BuiltinXV       S                   SQLTYPE(XML_CLOB:3000)
C/EXEC SQL
C+ DECLARE :BuiltinXV VARIABLE CCSID 37
C/END-EXEC

 ***********************************************
 *Procedure implementation
 ***********************************************
PPGM1             B
D                 PI
 /FREE
  EVAL ExtendersXV_Len = %len(Serialized_XML);
  EVAL ExtendersXV_Data = Serialized_XML;
  EXEC SQL
         CALL example_using_extender(
                        DB2XML.XMLCLOB(:ExtendersXV)
                        );

  EVAL BuiltinXV_Len = %len(Serialized_XML);
  EVAL BuiltinXV_Data = Serialized_XML;
  EXEC SQL CALL example_using_xml(:BuiltinXV);
 /END-FREE
P                 E
```

Listing 30: RPG program with XML host variables

The program in Listing 30 also illustrates another important concept. The RPG program uses the SQL DECLARE VARIABLE statement to indicate to SQL that the host variable, BuiltinXV, will store data using the CCSID 37 encoding scheme. An XML document that has been associated with a CCSID by the application is said to have external encoding. In contrast, XML documents that include an encoding declaration such as the document in Listing 9 are said to have internal encoding. When an XML document is retrieved from Internet, the sender usually constructs the XML document using internal encoding, rather than negotiating the encoding with the intended receiver.

The encoding of an XML document is a common source of problems. If both internal and external encodings are supplied, both encodings must match in order to parse the XML value. If there is an encoding conflict, a parse error will occur when the serialized XML document is parsed. The user-defined types available in XML Extender are based on character types and do not make it easy to resolve issues when the internal encoding declaration is not known in advance.

With the built-in XML data type, binary data types can be used to avoid encoding issues when transferring data to or from a host language. A binary data type does not have an associated CCSID; therefore the encoding of the XML data is entirely determined from the encoding declaration within the XML document. Listing 31 shows a C example where a host variable has been declared to be of type XML as a BLOB. The host variable is used in the call of the example_using_xml stored procedure that expects an XML value to be passed in as a parameter.

```
void call_sql_example(
                  const char * const input_document,
                  unsigned long input_document_length)
{
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS XML AS BLOB(5000) var1;
EXEC SQL END DECLARE SECTION;

var1.length = input_document_length;
memcpy(var1.data, input_document, input_document_length);

EXEC SQL CALL example_using_xml(:var1);

}
```

Listing 31: C function with XML_BLOB host variable

XML values in JDBC and SQL CLI

Similar to other user-defined types, the XML Extender user-defined types are not included in the JDBC standard, or by the SQL call level interface (SQL CLI). These interfaces are forced to deal with the XML Extender types by using the data type that matches the source type of the XML Extender user-defined type. For example java.sql.CLOB might be used to contain an XMLCLOB in JDBC, and SQL_CLOB might be used as the SQL type for XMLCLOB when using SQL CLI. Both JDBC and SQL CLI have been enhanced in DB2 for i 7.1 to support the built-in XML data type.

JDBC 4.0 provides a java.sql.SQLXML class as a host data type. This class represents serialized XML data. DB2 for i will parse and serialize the XML values as needed when the data is exchanged. An important consideration when using JDBC 4.0 is that it requires Java version 1.6 or later.

Listing 32 shows how an XML column can be updated by assigning an SQLXML object to a parameter marker.

```
    Connection IBMi;
    IBMi = DriverManager.getConnection("jdbc:as400:your_url_here");

    SQLXML doc = IBMi.createSQLXML();
    // Assume a table named dept has been created with
    // the following statement:
    // CREATE TABLE dept (id CHAR(8), deptdoc XML)
    PreparedStatement updateStmt =
        IBMi.prepareStatement(
                          "UPDATE xmltest2.dept SET deptdoc = ? " +
                          "WHERE ID = '0001'"
                         );

    doc.setString("<deptdoc> new department data </deptdoc> ");
    updateStmt.setSQLXML(1, doc);

    updateStmt.executeUpdate();
```

Listing 32: Using the JDBC SQLXML type to update a column

When using SQL CLI on DB2 for i 7.1, applications can retrieve and store XML data using a new SQL_XML data type. This data type corresponds to the XML built-in data type, and can be bound to a supported C type such as SQL_BLOB or SQL_CLOB.

The example in Listing 33 shows how to update XML data in an XML column using the SQL_C_BINARY type. The SQLBindParameter API associates the parameter marker with the built-in XML type (SQL_XML) and binds it to xmlBuffer; a variable that contains binary data (SQL_C_BINARY). Similar to the other interfaces, DB2 will parse the XML data in xmlBuffer when the statement is ran.

```
char xmlBuffer[10240];
integer length;
// Assume a table named dept has been created with the following statement:
// CREATE TABLE dept (id CHAR(8), deptdoc XML)
// xmlBuffer contains an internally encoded XML document that is to replace
// the existing XML document
length = strlen (xmlBuffer);
SQLPrepare (hStmt, "UPDATE dept SET deptdoc = ? WHERE id = ‘001’", SQL_NTS);
SQLBindParameter (hStmt, 1, SQL_PARAM_INPUT,
                    SQL_C_BINARY, SQL_XML, 0, 0,
                    xmlBuffer, 10240, &length);
SQLExecute (hStmt);
```

Listing 33: Update an XML column using SQL CLI

The SQL/XML Programmer’s guide contains more information and examples about using these interfaces to process XML. Several links have been included in the “Resources” section for further reading.

Processing external XML files

XML Extender provides the XMLFILE UDT that can be used as a reference for stream files that contain XML data. User-defined functions (UDFs) are provided for accessing and storing XML data in stream files.

The built-in XML type contains an XML value, described by the XDM and never refers to a value outside of DB2. DB2 for i provides a GET_XML_FILE function and file reference variables for working with XML data that is stored in stream files. Serialized XML data may be loaded into DB2 and parsed, so that it can be stored in a DB2 column or referenced within a query. Similarly, XML data stored in DB2 can be serialized and stored in a stream file.

This section discusses several approaches for working with XML data and stream files.

GET_XML_FILE

XML Extender provides the db2xml.XMLCLOBFromFile user-defined function. This function can be used to retrieve an XMLCLOB value when given a file's IFS path. Listing 34 demonstrates this function in action.

```
CREATE TABLE Extender_example (col1 DB2XML.XMLCLOB);
INSERT INTO Extender_example (col1)
  VALUES(db2xml.XMLCLOBFromFile('/home/ntl/myfile.xml'));
```

Listing 34: XMLCLOBFromFile example

DB2 for i 7.1 includes a built-in GET_XML_FILE function that provides similar capabilities. The GET_XML_FILE function will read the file specified by the argument and convert the data to UTF-8. If the file does not contain an XML declaration, one will be added. The GET_XML_FILE function returns a BLOB locator, which can be used to construct an XML value with XMLPARSE.

An important consideration when using the GET_XML_FILE function is that, because it returns a BLOB locator, it must be used under commitment control. The locator will be freed when a commit or rollback is performed. Listing 35 demonstrates a possible usage of the GET_XML_FILE function.

```
CREATE TABLE example (col1 XML);
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
INSERT INTO example (col1)
  VALUES( XMLPARSE(DOCUMENT GET_XML_FILE('/home/ntl/myfile.xml')));
COMMIT;
```

Listing 35: GET_XML_FILE example

As some interfaces require a serialized XML document as a BLOB, a BLOB type is returned by the GET_XML_FILE function rather than the XML data type. An implicit or explicit XMLPARSE must be performed to obtain an instance of an XML type.

The BLOB result type returned from the GET_XML_FILE function can be an advantage over a CLOB result because it avoids character set conversion that is typically associated with a CLOB data type. As discussed earlier, when working with XML documents as a serialized character value, a common problem is to accidently convert the document to a different character set, which invalidates the encoding declaration. Using a binary type helps avoid such issues.

XML file reference variables

Another way to accomplish the task of reading XML data from a file is by using file reference variables. File reference variables can be used by embedded SQL in host languages (C++, C, RPG, COBOL), and provide a handle to serialized XML data stored in a stream file. A file reference variable represents (rather than contains) the file, just as a large object (LOB) locator represents (rather than

contains) the LOB value. File reference variables can be used for both reading XML from files and writing XML data into files within a database query.

Reading XML from a file

Listing 36 contains C code that shows how to declare a file-reference variable for a stream file (/doc/input_file.xml), and how to insert the contents of the file into an XML column my_xml_col of table my_schema.my_table.

```
const char xml_file_name[255] = "/doc/input_file.xml";

EXEC SQL BEGIN DECLARE SECTION;
 SQL TYPE IS XML AS BLOB_FILE my_file_ref;
EXEC SQL END DECLARE SECTION;

memset(&my_file_ref, 0x0, sizeof(my_file_ref));
strcpy(my_file_ref.name, xml_file_name);
my_file_ref.name_length = strlen(xml_file_name);
my_file_ref.file_options = SQL_FILE_READ;

EXEC SQL INSERT INTO my_schema.my_table(my_xml_col)
 VALUES(:my_file_ref);
```

Listing 36: XML file reference for read

The variable my_file_ref is defined to have an SQL type of XML AS BLOB_FILE. This means that DB2 will construct XML values from a BLOB_FILE host variable. A CLOB_FILE type could also be used, however, when the XML document specifies its own encoding, it is generally a good idea to use a binary data type so that CCSID conversions and encoding mismatches are easier to avoid.

The precompiler creates a data structure for the BLOB_FILE variable, as shown in Listing 36.

```
static _Packed struct {
  unsigned long name_length;
  unsigned long data_length;
  unsigned long file_options;
  char name[255];
} my_xml_file;
```

Listing 37: Generated structure for the XML file locator

Within the structure, the name component is the IFS path to the file, and name_length is the length of this file name.

The data_length component is an output parameter. When the file reference variable is used for reading data, DB2 returns the length of the file (in bytes) in the data_length component. If the file reference variable is used for writing data, data_length will be set to the length of the new data that was written to the file.

The file_options component determines how the referenced file is to be used. The C precompiler generates the constants for this variable that are described in the following points. As in this example, the file reference variable is being used to read data from the file, the file option is set to SQL_FILE_READ.

- Constants for read operations
  - SQL_FILE_READ (2) – The file can be opened, read, and closed
- Constants for write operations
  - SQL_FILE_CREATE (8) – The file will be created. An error will be returned if the file already exists when the SQL statement is ran.
  - SQL_FILE_OVERWRITE (16) – The file will be created if it does not exist, or overwritten with the new data if it already exists.
  - SQL_FILE_APPEND (32) – This option has the output appended to the file if it exists, otherwise a new file is created.

Writing XML into a file

XML Extender offers the db2xml.XMLFileFromCLOB and db2xml.XMLFileFromVarchar functions to store XML data in a file and return an XMLFILE type as a reference to the data. In Listing 38, the db2xml.XMLFileFromClob function is used to write out the serialized XML document (as a CLOB) to a stream file with the IFS path /home/ntl/out1.xml.

```
CREATE TABLE Extender_example (col1 DB2XML.XMLCLOB);

INSERT INTO Extender_example(col1)
   VALUES('<doc> this is my document </doc>');

SELECT
db2xml.XMLFileFromCLOB(cast(col1 as CLOB(2g)), '/home/ntl/out1.xml')
FROM Extender_example;
```

Listing 38: db2xml.XMLFileFromCLOB Example

In DB2 for i 7.1, file reference host variables can be used to write serialized XML data into a file. This process is very similar to the task of reading XML data from an XML file with a file reference host variable.

Listing 39 contains the RPG code that selects the XML document stored in my_xml_col column from the table, myschema.my_table, where the column named pk is equal to one. The serialized data is stored in a file named, /home/ntl/out2.xml.

```
D outputFile      S             28A   varying
D                                     INZ('/home/ntl/out2.xml')

D my_xmlfile      S                   SQLTYPE(XML_BLOB_FILE)

 /FREE
  EVAL my_xmlfile_NAME = outputFile;
  EVAL my_xmlfile_NL   = %len(outputFile);
  EVAL my_xmlfile_FO   = SQFOVR;

  EXEC SQL SELECT my.my_xml_col INTO :my_xmlfile
           FROM my_schema.mytable mt WHERE pk = 1;

 /END-FREE
```

Listing 39: XML file reference for write

The host variable, my_xmlfile, is declared with the keyword SQLTYPE(XML_BLOB_FILE), which will cause the RPG precompiler to create a data structure for the variable that is shown in Listing 40.

```
D MY_XMLFILE        DS
D MY_XMLFILE_NL               10U
D MY_XMLFILE_DL               10U
D MY_XMLFILE_FO               10U
D MY_XMLFILE_NAME             255A
```

Listing 40: Structure generated by RPG precompiler

In the data structure, MY_XMLFILE_NAME and MY_XMLFILE_NL correspond to name, and name_length shown in Listing 37. MY_XMLFILE_DL corresponds to data_length in Listing 37 and will be set to the number of bytes written to the file after the SELECT statement has been ran. MY_XMLFILE_FO corresponds to file_options in Listing 40.

Similar to the SQL precompiler for C, the RPG SQL precompiler can generate constants that can be used for the file options variable. The generated constants and their values are shown in the following list. Each constant corresponds to the constant that has the same value mentioned in the list. For this example, MY_XML_FO is set to SQFOVR so that the output file is overwritten if the file already exists.

- SQFRD (2)
- SQFCRT (8)
- SQFOVR (16)
- SQFAPP (32)

As the application handles this XML value as a BLOB_FILE reference, the file will be created as a binary file and the XML value will be implicitly serialized to UTF-8. The encoding declaration of UTF-8 will be included in the document.

Working with binary files helps to avoid encoding mismatches and CCSID conversions, but it can make it more difficult to process these files as text. Text editors and other tools need to convert the data in the file from the file's CCSID to the CCSID required for processing or display, and these tools might not be able to interpret the XML encoding declaration. If a binary file creates problems for these non-XML aware tools, an XML_CLOB_FILE type can be used instead of XML_BLOB_FILE. Using a CLOB file reference sets the CCSID of the result file to match the data, rather than setting the result file CCSID to 65535 for binary data. Additional care must be taken with this approach to ensure that the CCSID of the file, the encoding of the data in the file, and the encoding declaration for the XML data stay synchronized.

Scenario overview

The following fictional scenario is used throughout the rest of this white paper to contrast the DB2 XML Extender and integrated XML support.

The automobile parts manufacturer, ABC Corporation has been running transaction processing and data warehouse applications on the IBM i; they have been using the integrated relational database for many years with great success. To maintain a competitive advantage, XML data must be used to communicate customer and order information over a complex corporate infrastructure. Previously, the company was using XML Extender to perform XML related tasks; however, in comparison to the new functionality, XML Extender has limited capabilities and does not conform to an industry standard. As a result, the corporation now wants to use the new XML support in their environment.

Figure 2 shows an overview of the data flow used by ABC Corporation.

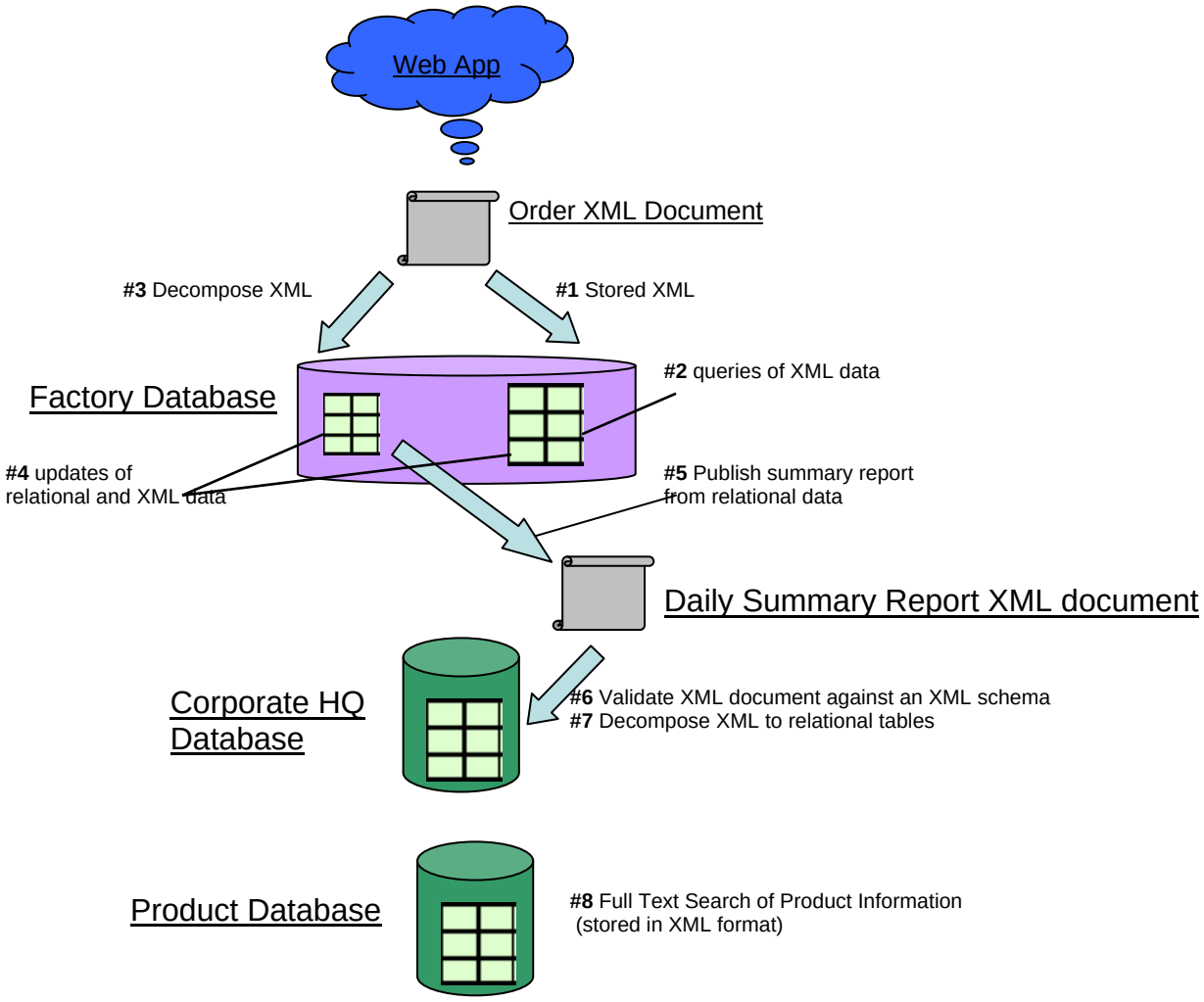

Figure 2: ABC Corporation overview

Each factory provides a web application to accept automobile part orders from customers. Part orders are received from the web application in an XML format, and processed throughout the organization.

XML processing steps

XML processing includes the following steps:

1. The original order (an XML document) is inserted into a DB2 table as XML for auditing purposes.
2. Queries are occasionally performed using the XML documents.
3. Applications prefer to use relational data, rather than XML data. Relational data integrates better with the company's existing data model, and offers improved scalability and performance benefits. For this reason, the original XML order documents are decomposed (shredded) into to a table.
4. In rare cases, XML order documents are updated with new information. For example, if a customer's name changes, all previous orders must be changed to reflect the new name (following a today for yesterday strategy). This is necessary to ensure that the information in the relational database is consistent with the XML documents.
5. At the end of each day, each factory uses its relational data to compose and send a summary report of new orders (in XML format) to the corporate headquarters database.
6. The corporate headquarters validates that the XML documents conform to corporate standards.
7. The validated XML documents are shredded into a relational database. Business intelligence applications can then integrate the relational data into a data warehouse.
8. The corporate headquarters maintains a DB2 column that contains a set of documents (in XML format) which describe the products that are in its inventory. The business needs to search this column for products that contain linguistic matches of keywords, where the search is scoped to specific elements within the document. The search also needs to incorporate values such as date ranges in the request.

Store XML documents in DB2

The first step is to store the order document (Listing 41) received from the Web application into a DB2 XML column. To simplify the example, the XML order documents will be loaded from an IFS stream file.

```
<?xml version="1.0"?>
<orders xmlns="http://www.abccompany.com">
    <submissionCode>123ABC</submissionCode>
    <submissionDate>2012-06-14</submissionDate>
    <order_infor>
        <part_name>Valve</part_name>
        <quantity>1000</quantity>
        <order_datetime>2012-06-14T08:20:00+03:00</order_datetime>
        <customer_name>First Automobile Works</customer_name>
    </order_infor>
    <order_infor>
        <part_name>Flywheel</part_name>
        <quantity>2000</quantity>
        <order_datetime>2012-06-14T13:20:00+03:00</order_datetime>
        <customer_name>Second Automobile Works</customer_name>
    </order_infor>
</orders>
```

Listing 41: XML order document

The original_orders table contains a column with the XML data type and is defined with the CREATE TABLE statement in Listing 42.

```
CREATE TABLE original_orders (
                               order_doc_id BIGINT
                                              GENERATED ALWAYS AS IDENTITY
                                                (START WITH 1
                                                 INCREMENT BY 1 NOCYCLE),
                               order_doc XML NOT NULL,
                               PRIMARY KEY (order_doc_id))
```

Listing 42: Table for original_orders document

Inserting the XML document into the table can be done with the GET_XML_FILE function, as shown in the following listing.

```
INSERT INTO original_orders (order_doc)
    VALUES(XMLPARSE(DOCUMENT
                      GET_XML_FILE('/Order/2012-06-14-123456.xml') ) )
```

Listing 43: Insert into original_orders

Query XML data

ABC Corporation needs to occasionally perform SQL queries that involve data contained in the XML documents archived in DB2. The XML Extender option provides scalar and table extraction functions to satisfy this requirement. The XMLTABLE function provides a much wider set of capabilities and can be used to replace the already existing function. This section compares the capabilities offered by the XML Extender functions with the capabilities provided by the XMLTABLE function in DB2 for i 7.1.

XML Extender extraction functions

XML Extender provides a set of scalar and table extraction functions for retrieving data from columns defined using one of the XML Extender user-defined types. The XML Extender functions are named for the SQL data type that was extracted from the document. The singular version of the name (for example, db2xml.extractVarchar) indicated the scalar function, and the plural version of the name (for example, db2xml.extractVarchars) represented the table function. For example, if the order_doc column is defined with the XMLCLOB UDT, a developer can extract the submission code as an SQL VARCHAR value with the following query.

```
SELECT
  order_doc_id,
  db2xml.extractVarchar(order_doc, '/orders/submissionCode')
FROM original_orders
```

Listing 44: db2xml.extractVarchar scalar function

The scalar function returned an error if more than one XML node matched the location path expression. Repeating elements are common in XML documents. For this reason, table functions that return result sets were more commonly used for extracting information.

Using XML Extender, the query in the following listing invokes the db2xml.extractVarchars table function to retrieve a result set that contains one row for each customer name.

```
SELECT  original_orders.order_doc_id,
         customer_rs.customer_name
FROM
 original_orders,
 TABLE(db2xml.extractVarchars(
                              order_doc,
                              '/orders/order_infor/customer_name'
                               )
        ) customer_rs(customer_name)
```

Listing 45: db2xml.extractVarchars table function

The result set (shown in the following table) contains one column for the order's document ID and another column for the extracted customer name as an SQL VARCHAR data type. The join between the original_orders table and the db2xml.extractVarchars table function produces two rows for the order XML document with the ID number 1.

ORDER_DOC_ID	CUSTOMER_NAME
1	First Automobile Works
1	Second Automobile Works

Table 1: Result set

These XML Extender extraction functions are useful, but there are a number of deficiencies associated with their usage:

- The path expressions are namespace unaware which can cause erroneous matches with elements and attributes in another namespace. The inability to specify a namespace in the path expression creates challenges for some XML environments where the same local name exists in multiple namespaces.
- XML Extender does not have any awareness of the atomic XML data types defined by the XML Data Model. Extracting SQL data from XML documents required that the data in the XML documents to be stored in an SQL character data format. As most applications expect XML data to use data types defined by the XDM, the inability to convert from a data type defined by the XDM to an SQL type was not preferred. This issue is discussed in detail later in this paper.
- Using the extraction functions to create a result set with multiple columns required multiple table references and correlations. This type of implementation can cause performance issues and is also difficult to code.

The XMLTABLE function provided in DB2 for i 7.1 provides similar functionality, but does not have the same weaknesses as the DB2 XML Extender table functions.

Using the XMLTABLE table function

The DB2 for i 7.1 Group PTF SF99701 Level 14 (or higher) includes a powerful XMLTABLE table function for queries that need to access XML data. This function returns a result set containing one or more columns with the specified portions of the XML document. Instead of location paths, the result set's values are determined using a very similar, but more expressive XPath 2.0 syntax. The function handles the conversion of the data in the document from the XML data type to the SQL type of the result set's column when necessary. XMLTABLE properly supports XML namespaces, and provides a rich set of expressions that can be used within XPath predicates.

The following query offers a comparison of the XMLTABLE function with the extractVarchars table function example in Listing 45. The following query uses the XMLTABLE function to return a result set with customer names from XML documents in the order_doc column.

```
SELECT  oo.order_doc_id,
         customer_rs.customer_name
FROM   original_orders oo,
        XMLTABLE(
          XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
          '/orders/order_infor/customer_name'
          PASSING oo.order_doc
          COLUMNS
           customer_name VARCHAR(255) PATH '.'
         ) customer_rs
```

Listing 46: XMLTABLE function used to return a VARCHAR column

The XMLTABLE keyword defines a table reference, meaning that the columns it defines can be referenced in the query the same way as the columns of a table function or table. For practical purposes, XMLTABLE is considered a special type of table function, rather than a completely unique SQL concept.

The XMLNAMESPACES declaration defines the namespace bindings used for the XPath expressions. In this example, *http://www.abccompany.com* is the default element namespace for all unqualified element names. Element and attribute names in path expressions will not match elements and attributes in the document unless both share this namespace Uniform Resource Identifier (URI) value.

The literal '/orders/order_infor/customer_name' is the row expression; it indicates that the result set will contain one row for each customer_name element. Each element returned from the row expression becomes the context item, or starting point for evaluation of the column path expressions.

The PASSING clause defines order_doc as the initial context, or relative starting point for the row XPath expression.

The COLUMNS clause defines the columns that will appear in the result set from XMLTABLE. The CUSTOMER_NAME column will be a VARCHAR(255) column. The value of this column is determined from the PATH expression ('.'). The period indicates that the current context item should be used to determine the customer name. In this example, the context item is the customer_name element (determined by the row expression) that is being used as a starting point for the evaluation of the column expressions.

Although its syntax is unique, the XMLTABLE function shares the same correlation rules as a table function. In Listing 46, the function invocation references oo.order_doc, a column in an object table declared previously (left-to-right) in the same FROM clause as XMLTABLE. This situation is called lateral correlation. Each row of original_orders is cross-joined with the rows returned by the XMLTABLE function for that row. Rows from original_orders for which the XMLTABLE function returns no rows will not appear in the final result set.

The results from the query in Listing 46 are shown in the following table. The cross-join between the original_orders table and XMLTABLE produces two rows for the order document with the ID number 1.

ORDER_DOC_ID	CUSTOMER_NAME
1	First Automobile Works
1	Second Automobile Works

Table 2: XMLTABLE query results

The XMLTABLE function is capable of much more powerful queries than that were ever possible with XML Extender. It supports much improved capabilities for using XPath expressions. XMLTABLE also has the ability to pass values from SQL as parameters and reference the equivalent XML values within XPath expressions. You can find more detailed discussion and tutorial for the XMLTABLE function's syntax and capabilities in the IBM i information center. Refer to the links provided in the "Resources" section at the end of the paper.

Using the XMLTABLE function to retrieve a scalar result

DB2 for i does not include a scalar counterpart to the XMLTABLE function. However, a scalar function is not necessary as the table function can be effectively used to return scalar values. At first glance, a query to extract submissionCode from the XML order document that was shown in Listing 41 might use a subquery as a column in the select list. In this approach, the subquery invokes the XMLTABLE function. This solution is shown in the following listing.

```
SELECT
 order_doc_id,
 (
  SELECT xt.submissionCode
  FROM
   XMLTABLE(
     XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
     'orders/submissionCode'
     PASSING original_orders.order_doc
     COLUMNS
     submissionCode VARCHAR(100) PATH '.'
   ) xt
 ) AS submissioncode
FROM original_orders
```

Listing 47: Using XMLTABLE in a subquery

The subquery returns a scalar VARCHAR value, given an XML document from the order_doc column in the original_orders table. If the subquery were to ever return more than one row, the query will fail when the attempt to return the additional row occurs. When the subquery does not return any rows, a null value is returned for the submissioncode result. The result set from the query is shown in the following table.

ORDER_DOC_ID	SUBMISSIONCODE
1	123ABC

Table 3: XMLTABLE result set

Replacing the scalar extract function call with a subquery is easy to do, but is not the most efficient solution, especially if many columns are being extracted from an XML document.

If a developer knows that the subquery will always return exactly one row, then the previous SQL query can be written using a cross join instead of a subquery. The cross join implementation is shown in the following listing.

```
SELECT  order_doc_id, xt.submissionCode
FROM
   original_orders CROSS JOIN
   XMLTABLE(
     XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
     '.'
     PASSING original_orders.order_doc
     COLUMNS
     submissionCode VARCHAR(100) PATH 'orders/submissionCode'
   ) xt
```

Listing 48: Using the XMLTABLE function in a cross join

This query was written in such a way that when the order_doc column contains a non-null value, the XMLTABLE function always returns exactly one row to use in the join. The row expression ('.') is a reference to the context item, which in this case is the root node of the XML document provided by the order_doc column. The table function can therefore result in at most one row per input document. The only way the row expression would not find a match for the root node (causing the table function to return no rows) is when the order_doc column value is null.

The submissionCode column's XPath expression (orders/submissionCode) contains the path to the value that needs to be returned. If the column's path expression results in no matches, the null value is assigned to the column. If the column's path expression results in more than one value, the query results in an error; SQL does not allow a row to have multiple values for a single column.

The result of the query in Listing 48 matches the query in Listing 47 and is shown in Table 3.

Suppose that the order_doc column had been created as null capable, and that the INSERT statement in Listing 49 has been run to insert a null value. The query in Listing 48 will not return a row for the newly inserted row in the order_doc table that contains the null value.

```
INSERT INTO original_orders (order_doc) VALUES(NULL)
```

Listing 49: Insert of null value into an XML column

If the value for the order_doc column is null, then the row expression will not result in a match for any node, and the result of the XMLTABLE function will return an empty table. The cross join will not include the rows from original_orders for which the XMLTABLE function resulted in an empty table, as those rows have nothing to join with. This issue can be resolved by changing the join operator from cross join to a left outer join. Unlike the cross join, the left outer join query includes the rows in original_orders, which do not have rows from the XMLTABLE function to join with, assigning null to the values of columns of those rows. A trivial join condition is used because there is no join column. The left outer join version of this query is shown in the following listing.

```
SELECT  order_doc_id, xt.submissionCode
FROM
   original_orders LEFT OUTER JOIN
   XMLTABLE(
     XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
     '.'
     PASSING original_orders.order_doc
     COLUMNS
     submissionDate VARCHAR(100) PATH 'orders/submissionCode'
   ) xt  ON (1 = 1)
```

Listing 50: Using the XMLTABLE function in a LEFT OUTER JOIN

The results from the query are shown in the following table. The dash character ('-') is used to represent a null value.

ORDER_DOC_ID	SUBMISSIONCODE
1	123ABC
2	-

Table 4: Results from the LEFT OUTER JOIN

Using either type of joins is an advantage over using a scalar function (or a subquery) in the select list. Multiple columns can be extracted from a single XMLTABLE invocation, which allows many path expressions to be combined into one XMLTABLE function call. For example, the following query uses XML Extender functions to extract and compare the values stored in an XML document. The three UDF invocations require evaluations of three location paths. The first two UDF invocations are used to extract submission date and code value, and the third UDF invocation is used to check if the value of the SubmissionDate element meets the specified selection criteria.

```
SELECT
   db2xml.extractDate(order_doc, '/orders/submissionDate')     sd,
   db2xml.extractVarchar(order_doc, '/orders/submissionCode') code
 FROM original_orders
 WHERE
  db2xml.extractDate(order_doc, '/orders/submissionDate') >
    DATE('2012-06-12')
```

Listing 51: Multiple scalar functions in XML Extender

A simpler solution is implemented with a single XMLTABLE invocation, as shown in Listing 52. This query can perform better, because fewer expressions need to be evaluated, and the XML document does not need to be evaluated by multiple user-defined functions.

```
SELECT xt.sd, xt.code FROM
 original_orders,
 XMLTABLE(
    XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
    '.'
    PASSING original_orders.order_doc
    COLUMNS
      sd   DATE         PATH 'orders/submissionDate',
      code VARCHAR(100)  PATH 'orders/submissionCode'
  ) XT
 WHERE xt.sd > DATE('2012-06-12')
```

Listing 52: Multiple column expressions in XMLTABLE

When extracting scalar values, evaluating multiple path expressions within a single table function offers a performance advantage over multiple function invocations, and makes the query more readable.

XML and SQL data type conversions

The XML Extender extraction functions assume that the data in the XML document is string data that represented an SQL data type. In order to extract an SQL value from an XML document, XML Extender simply extracts the string value using a location path, and converts the string result to the specified SQL type with the SQL Cast function.

The lexical format and properties of the XDM and SQL types are not always the same. For instance, the ISO SQL TIMESTAMP has a lexical format of **yyyy-mm-dd hh:mm:ss.uuuuuu**. However, the most similar XDM type xs:dateTime has a lexical format of **yyyy-mm-ddThh:mm:ss.sssssssssssszzzzzz**, where **zzzzzz** is an optional time zone component. In addition to lexical differences, the inclusion of a time zone component makes the properties of the xs:dateTime type fundamentally different than the SQL TIMESTAMP type.

As XML is used for exchanging information in platform- and application-independent environments, applications usually expect data in an XML document to conform to the XML standard, rather than the SQL representation. This makes the XML Extender approach inadequate for many environments, as it cannot handle the XDM types.

While the XML Extender functions expect XML documents to contain SQL data strings, the XMLTABLE function expects the XML document to contain XDM types and will convert a value of an XDM type to the appropriate SQL type whenever the value is returned to SQL. This allows XMLTABLE to work with XML documents that conform to the W3C standard, rather than documents built exclusively for consumption by SQL.

When using XMLTABLE, the required XDM type for the result of a column's PATH expression is determined by the mapping described in the following table. In this table, the SQL type represents the data type of the SQL column in the COLUMNS clause; the XDM type represents the required result type of the XPath expression. It is not necessary to explicitly specify the XDM type in the path expression because DB2 for i can determine this using the SQL type of the column and the mapping in Table 5. The result of the path expression will be implicitly casted to the appropriate XDM type, before converting the value to the requested SQL data type.

SQL data type	DB2 for i XDM type
SMALLINT INTEGER BIGINT	xs:integer
DECIMAL NUMERIC	xs:decimal
FLOAT DOUBLE DECFLOAT	xs:double
DATE	xs:date
TIME	xs:time
TIMESTAMP	xs:dateTime
CHAR VARCHAR GRAPHIC VARGRAPHIC CLOB DBCLOB	xs:string

Table 5: Supported SQL and XML type conversions

An example of XML to SQL conversion is shown in Listing 53. The order_datetime column specified in the COLUMNS clause on the XMLTABLE function has an SQL data type of TIMESTAMP. For this reason, XMLTABLE builds an xs:dateTime XML value using the result of the PATH expression. The xs:dateTime value is then converted into the SQL Timestamp format. The results of the query are shown in Table 6.

```
SELECT xt.order_datetime
  FROM
   original_orders,
   XMLTABLE(
     XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
     'orders/order_infor/order_datetime'
     PASSING original_orders.order_doc
     COLUMNS
      order_datetime TIMESTAMP PATH '.' ) xt
```

Listing 53: Query with timestamp result column

One thing to notice in the following table is that the timestamp values are different than the timestamp values in the XML document (refer to Listing 41). Because the SQL TIMESTAMP data type on DB2 for i does not support time zones, the XMLTABLE function normalizes the xs:dateTime value to the Coordinated Universal Time (UTC) and removes the time zone when the value is returned to the invoking SQL statement.

ORDER_DATETIME
2012-06-14 05:20:00.000000
2012-06-14 10:20:00.000000

Table 6: ORDER_DATETIME result column

Consistency with the ISO SQL/XML and W3C standards is a requirement for the ABC Corporation. So, they have chosen to always represent their timestamp values as xs:dateTime within their XML documents. This implementation ensures that applications that follow the W3C XML standard will now have no trouble in processing the date and timestamp values stored within their XML document.

Applications that need to extract values from an XML document using XML Extender have to store all timestamp data within XML documents as string data that represents SQL values with an ISO format. If ABC Corporation had stored timestamp values in this fashion, then these values would not be valid xs:dateTime values, and the built-in XML functions available in DB2 for i 7.1 would not be able to work with them as timestamps. The XMLTABLE function would not be able to return these values as an SQL TIMESTAMP, as the function expects to find data that can be recognized as an xs:dateTime value.

Listing 54 contains an example that is the XMLTABLE function equivalent of the example given by the XML Extender administrative guide for the db2xml.extractTimestamps function.

```
SELECT xt.*
FROM
 XMLTABLE('//data'
  PASSING
   XMLPARSE(DOCUMENT
     '<stuff>
        <data>2003-11-11-11.12.13.888888</data>
        <data>2003-12-22-11.12.13.888888</data>
      </stuff>')
  COLUMNS
   timestamp_col TIMESTAMP PATH '.'  ) xt
```

Listing 54: XMLTABLE example of an invalid xs:dateTime conversion

The query in Listing 54 will fail with an error. This error occurs because the timestamps in element 'data' are not valid xs:dateTime values. This is a common problem in the industry when working with XML applications that did not represent their data according to the XDM.
In these cases, the most common solution is to have the XMLTABLE function return the value as a VARCHAR value and then convert the value to a timestamp value on the SELECT list. This approach is shown in the following listing.

```
SELECT
 TIMESTAMP_FORMAT(xt.timestamp_as_varchar_col,
                  'YYYY-MM-DD-HH24.MI.SS.NNNNNN') AS out_col
FROM
 XMLTABLE('//data'
            PASSING
              XMLPARSE(DOCUMENT
               '<stuff>
                 <data>2003-11-11-11.12.13.888888</data>
                 <data>2003-12-22-11.12.13.888888</data>
                </stuff>'
              )
            COLUMNS
             timestamp_as_varchar_col VARCHAR(26) PATH '.'
 ) xt
```

Listing 55: Building a TIMESTAMP from a VARCHAR result to an SQL type

Due to the fact that the query in Listing 55 uses a VARCHAR type for the timestamp_as_varchar_col column, the XDM type for the path expression is required to be xs:string (the type mapped to VARCHAR by Table 5). The XMLTABLE function casts each data element to xs:string, and coverts the xs:string to an SQL VARCHAR with no issues, the VARCHAR is assigned to the timestamp_as_varchar_col column of the table function. The timestamp_as_varchar_col column becomes an input to the TIMESTAMP_FORMAT function, which creates the timestamp result that is stored in the column out_col.

The output of the query is shown in the following table.

OUT_COL
2003-11-11 11:12:13.888888
2003-12-22 11:12:13.888888

Table 7: TIMESTAMP result from a VARCHAR column

Return an xs:date, xs:time, or xs:dateTime value's local time

When an xs:date, xs:time, or xs:dateTime includes a time zone, the XMLTABLE function normalizes the value to UTC and removes the time zone before returning the value to SQL as a Date, Time, or Timestamp value. This processing causes a loss of information because after normalization, it is no longer possible to determine the original local date and time for the value.

Sometimes the local date and time is important to an application and normalizing values of these types to UTC causes a problem. For example, given a set of XML order documents similar to the one shown in Listing 41, a business intelligence application might need to determine on what dates, or at what time of day, the most purchases have been made. When customers are placing orders from many different time zones, normalizing the values to UTC is not helpful. It is the date and time of the purchase in the purchaser's local time zone that is interesting.

An additional problem is that some tools for working with XML do not support normalizing date, time, and timestamp values to UTC. These tools usually truncate the time zone component, which causes the result value to contain only the local time portion of the original value. In some cases, this makes it necessary to retrieve xs:date, xs:time, or xs:dateTime values with the time zone component removed when using the XMLTABLE function.

XPath built-in functions can be used to remove the time zone component from an xs:date, xs:time, or xs:dateTime value before returning the value to SQL. The fn:adjust-date-to-timezone, fn:adjust-time-to-timezone, and fn:adjust-dateTime-to-timezone built-in functions are provided to adjust the time zone component of an xs:date, xs:time, or xs:dateTime value respectively. When the new time zone is an empty sequence (written as a pair of empty parenthesis ('()') in XPath), the result of the function is the input value with the time zone removed. As the adjusted value does not have a time zone, it is not normalized to UTC when it is returned to SQL.

The following listing shows how the query in Listing 53 can be modified so that each row contains a timestamp that represents the local time of the original value. The xs:dateTime function is used to construct an xs:dateTime value using the current context node ('.'). The xs:dateTime is then passed into the fn:adjust-dateTime-to-timezone function. The second parameter to the adjust-dateTime-to-timezone function is an empty sequence, which causes the time zone component to be removed.

```
SELECT xt.order_datetime
  FROM original_orders,
   XMLTABLE(
     XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
     'orders/order_infor/order_datetime'
     PASSING original_orders.order_doc
     COLUMNS
      order_datetime TIMESTAMP
                      PATH   'fn:adjust-dateTime-to-timezone(
                                                              xs:dateTime(.),
                                                              ()
                                  )'
   ) xt
```

Listing 56: Return an xs:dateTime's local dateTime as a TIMESTAMP

The result of the query is shown in the following table. The difference between Table 8 and Table 6 is that the normalization to UTC did not occur for the output displayed in Table 8.

ORDER_DATETIME
2012-06-14 08:20:00.000000
2012-06-14 13:20:00.000000

Table 8: order_datetime results based on using the original value's local time

Decompose XML to a relational database table

Decomposition (also known as shredding) is the process of storing the values encapsulated within an XML document into columns of one or more relational tables.

ABC Corporation's business applications are dependent on data being available in a relational model. XML is typically used only in specific scenarios related to web interfaces. As a result, the order XML document needs to be decomposed into a relational database table. The target table for the decomposition is defined by the SQL CREATE TABLE statement displayed in Listing 57.

```
CREATE TABLE orders (
   order_doc_id      BIGINT,
   order_id           BIGINT,
   part_name          VARCHAR(1000),
   part_number        BIGINT,
   order_timestamp   TIMESTAMP,
   customer_name     VARCHAR(1000),
   PRIMARY KEY (order_doc_id, order_id))
```

Listing 57: Orders table definition

With the DB2 XML Extender, an XML document is decomposed into a relational table by using either the db2xml.dxxInsertXML or db2xml.dxxShredXML stored procedures. These procedures require the mapping between the XML document and target relational table to be defined in advance with a document access definition (DAD) file or XML collection file.

A sample DAD file is shown in Listing 58. This DAD file could have been used with XML Extender for decomposing the part_name and quantity elements from Listing 41 into the SQL columns part_name and part_number columns in the *orders* table defined in Listing 57.

```
<?xml version="1.0"?>
<DAD>
<validation>NO</validation>
<Xcollection>
<root_node>
<element_node name="orders">
  <RDB_node>
     <table name="EXAMPLE.ORDERS" />
  </RDB_node>
<element_node name="order_infor"
              multi_occurrence="YES">
<element_node name="part_name">
<text_node>
  <RDB_node>
     <table name="EXAMPLE.ORDERS" />
     <column name="PART_NAME" type="VARCHAR(1000)" />
  </RDB_node>
</text_node>
</element_node>
<element_node name="quantity">
<text_node>
  <RDB_node>
     <table name="EXAMPLE.ORDERS" />
     <column name="PART_NUMBER" type="BIGINT" />
  </RDB_node>
</text_node>
</element_node>
</element_node>
</element_node>
</root_node>
<Xcollection>
</DAD>
```

Listing 58: DAD file for decomposition

Using DAD files to decompose an XML document had a number of important limitations. The xs:dateTime value within the order_dateTime element of the Orders document cannot be directly mapped to the TIMESTAMP field of the ORDERS table using a DAD file because the XML Extender does not have an understanding of the XDM data types. Also, the XML Extender support is not able to easily include data values that are not stored in the XML document. A good example of where this feature is needed can be seen with the order_doc_id column on the original_orders table (created in Listing 42). This column was created as an identity column, so that DB2 would generate a unique document identifier for each XML document when the XML document was inserted into the original_orders table. The unique identifier needs to be included as a column on each of the tables that are the targets of the decomposition, so that the decomposed data can be linked back to the source XML document.

The limitations for decomposing XML data using XML Extender are no longer a problem with the integrated XML support in the IBM i 7.1 release. As the XMLTABLE function returns a result set, it can be used within an SQL INSERT statement as part of a query to decompose an XML document. This is particularly useful when we only need to decompose documents into a single target table. When multiple target tables are needed and an XML schema is available, annotating the XML schema and using the XML

decomposition stored procedures can be a better option. Schema annotations and decomposition are discussed later on.

After the order XML document has been stored into the original_orders table (refer to Listing 42), the statement in Listing 59 will insert the information from the order XML document into the *orders* table whose definition can be found in Listing 57.

```
INSERT INTO orders(order_doc_id, order_id,  part_name,  part_number,
                   order_timestamp, customer_name)
SELECT
 oo.order_doc_id,
 t.row_number,
 t.part_name,
 t.part_number,
 t.order_timestamp,
 t.customer_name
FROM original_orders oo,
 XMLTABLE(XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
   '/orders/order_infor'
   PASSING  oo.order_doc
   COLUMNS
    row_number       FOR ORDINALITY,
    part_name        VARCHAR (1000) PATH './part_name',
    part_number      BIGINT         PATH './quantity',
    order_timestamp  TIMESTAMP      PATH './order_datetime ',
    customer_name    VARCHAR (1000) PATH './customer_name'
  ) AS t
```

Listing 59: Decompose using the XMLTABLE function

The diagram in Figure 3 graphically represents this insertion process.

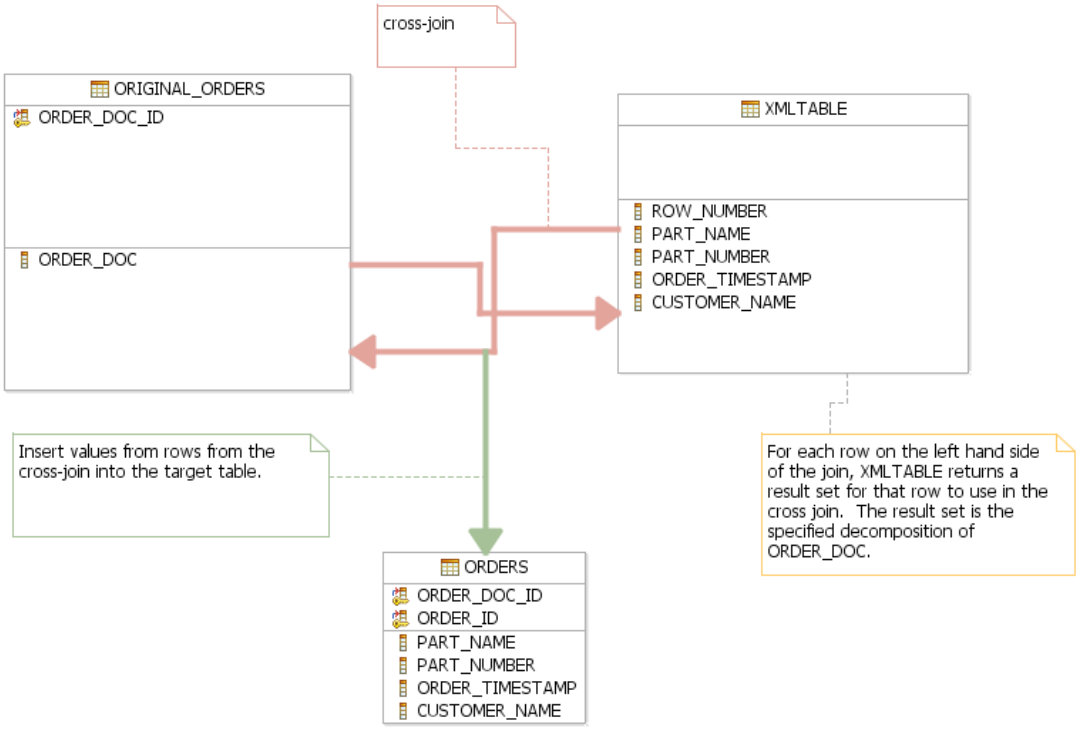

Figure 3: Graphical representation of the insertion process using the XMLTABLE function

The row_number column does not have an explicit data type or XPath expression associated with it. Instead, it specifies that the value returned should be the ordinality, or row number, for the row returned for this particular XML document. An ordinality column always has an SQL type of BIGINT. This value combined with the document ID associated with the original XML document can be used as a primary key for the target table.

Due to the fact that the XMLTABLE function handles the conversion of XML types to the specified SQL types of each column, dealing with xs:dateTime values are no longer a challenge. If a time zone is present on the xs:dateTime value for the order_dateTime element, the XMLTABLE function normalizes the time zone to UTC and removes the time zone. The dateTime value is then converted to an SQL timestamp and returned to SQL as the value for the order_timestamp column.

If the XML document stored in the order_doc column had been created so that the order_datetime element did not conform to the xs:dateTime type in the XML data model, then using a data type of VARCHAR for the order_timestamp column and casting the VARCHAR string to a TIMESTAMP type in SQL would allow the requested value to be inserted into the target table. This is the same approach that was shown in Table 6.

Table 9 contains a listing of the rows stored in the *orders* table after running the INSERT statement in Listing 59. These results assume that the original_orders table contains only a single order XML document (described in Listing 41), which has been assigned a document identifier of **1.**

ORDER_DOC_ID	ORDER_ID	PART_NAME	PART_NUMBER	ORDER_TIMESTAMP	CUSTOMER_NAME
1	1	Valve	1000	2012-06-14 05:20:00.000000	First Automobile Works
1	2	Flywheel	2000	2012-06-14 10:20:00.000000	Second Automobile Works

*Table 9: Rows in the **orders** table*

Update XML data

The XML data stored by ABC Corporation in their DB2 for i database is usually read-only and not updated. However, sometimes ABC Corporation does need to modify the XML documents stored in DB2. For example, if a customer changes their name, all of the order documents for that customer must be updated to the new name. This section describes how to update the relational and XML data so that the customer name, **Second Automobile Works**, is changed to **Ultimate Automobile Works**.

With the XML Extender product, simple changes to an element or attribute's value can be made to XML documents using the db2xml.update UDF. Listing 60 contains an SQL statement that invokes the db2xml.update function to change all customer names in a document to **Ultimate Automobile Works** .

```
UPDATE original_orders oo SET oo.order_doc =
           db2xml.update(oo.order_doc,
                 '/orders/order_infor/customer_name',
                 "Ultimate Automobile Works")
```

Listing 60: Update function in XML Extender

The capabilities of the db2xml.update function are very limited. Conditional changes are inadequate because only attributes and literals can be referenced in predicates. For example, it is not possible to use the update function to change the contents of the customer_name element to **Ultimate Automobile Works** only in the cases where the current content is **Second Automobile Works**, because customer_name is an element (rather than an attribute). In addition, changes to the structure of the document are not possible with this function, and there is no way to add or remove elements or attributes. Another problem is that the function has no understanding of XML namespaces.

More complex changes to XML documents can be made with the XML Extender support using an Extensible Stylesheet Language Transformation (XSLT) template and either the db2xml.XSLTransformToCLOB or db2xml.XSLTransformToFile UDFs.

The XML support in the DB2 for i 7.1 release does not provide an update built-in function, but it does include XSLT support. The XSLTRANSFORM built-in function can be used with an XSLT 1.10 template to modify an XML document. XSLT uses XPath expressions to find and transform information in an XML document.

A prerequisite to using XSLTRANSFORM on IBM i is that additional products and options must be installed to use it.

- XML Toolkit for IBM System i5® (5733-XT2 options 1 and 6)
- International Components for Unicode (5770-SS1 option 39)

The 5733-XT2 product is a priced licensed program; 5770-SS1 option 39 is a *no additional charge* option of 5770-SS1.

One thing to keep in mind is that XSLT 1.10 supports only XPath 1.0 expressions, while XMLTABLE supports a subset of the XPath 2.0 standard. This makes the XPath expressions that are used in style sheet transformations less versatile than the ones that can be used with the XMLTABLE function. While not as versatile, XSLT 1.10 offers the same level of support that is available in XML Extender and simple modifications to XML documents should not be a problem. A link to the XSLT 1.10 standard is included in the references.

Listing 61 shows an XSLT stylesheet that can be used to conditionally change an element's content within an XML document. The stylesheet changes the content of customer_name elements to **Ultimate Automobile Works** when the current element content is **Second Automobile Works**. The content of other customer_name elements are left unchanged.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
     xmlns:abc="http://www.abccompany.com"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="node()|@*">
   <xsl:copy>
     <xsl:apply-templates/>
   </xsl:copy>
  </xsl:template>

  <xsl:template match=
    "/abc:orders/abc:order_infor/abc:customer_name/
      text()[.=  'Second Automobile Works']">
   <xsl:text>Ultimate Automobile Works</xsl:text>
 </xsl:template>

</xsl:stylesheet>
```

Listing 61: XSLT stylesheet

The stylesheet in Listing 61 defines two templates for matches. Nodes in the XML document will be processed by at most one template. In cases where a node is matched by multiple templates, the default priority is used to determine the template that can be used. In this example, the second template has a higher priority because it specifies a specific element name and node test.

The first template is applied to all nodes that are not processed by the higher priority second template. In this example, the first template results in all of the elements except the customer_name element being copied into the output document. The second template is applied for text nodes under the customer_name element that are equal to **Second Automobile Works.** The second template replaces the text node with a text node that contains **Ultimate Automobile Works**.

Using the stylesheet approach, updates to the XML document and relational data can be done with a couple of SQL statements. An example of this approach is shown in Listing 62. Notice how the WHERE clause (in bold format) on the first UDATE SQL statement uses relational data to limit the number of XSL transformations to the XML documents that need them. An XSL transform is an expensive operation, and it is preferred to only apply the template to documents that actually need to be changed.

For simplicity, this example assumes that the stylesheet in Listing 61 is stored in the global variable xsl_stylesheet.

```
-- update affected XML documents
-------------------------------------
UPDATE original_orders oo SET oo.order_doc = XSLTRANSFORM (oo.order_doc
                 USING XMLPARSE(DOCUMENT xsl_stylesheet)AS CLOB(2G) )
WHERE oo.order_doc_id IN (
    SELECT order_doc_id FROM   orders
    WHERE customer_name = 'Second Automobile Works');

-- update relational data
-------------------------------------
UPDATE orders SET customer_name =    'Ultimate Automobile Works'
              WHERE customer_name = 'Second Automobile Works';
```

Listing 62: SQL script to update XML and relational data

After the transform and update operations have completed, the order document in the original_orders table will have the data value shown in the following listing. The second update statement in Listing 62 updates the relational data stored in the *orders* table.

```
<?xml version="1.0" encoding="UTF-8"?>
<orders xmlns="http://www.abccompany.com">
   <submissionCode>123ABC</submissionCode>
   <submissionDate>2012-06-14</submissionDate>
    <order_infor>
       <part_name>Valve</part_name>
       <quantity>1000</quantity>
       <order_datetime>2012-06-14T08:20:00+03:00</order_datetime>
       <customer_name>First Automobile Works</customer_name>
    </order_infor>
    <order_infor>
       <part_name>Flywheel</part_name>
       <quantity>2000</quantity>
       <order_datetime>2012-06-14T13:20:00+03:00</order_datetime>
       <customer_name>Ultimate Automobile Works</customer_name>
    </order_infor>
</orders>
```

Listing 63: Orders document after update

This XSLT-based solution provided with DB2 for i 7.1 is not as elegant as the db2xml.update function seen in the XML Extender product; however, the db2xml.update function does not have the capability to perform this modification as it lacks the ability to express the predicate and namespace conditions. It is a relatively trivial task to adapt the stylesheet presented here to handle the simpler modifications that the db2xml.update function might have handled in the past.

Compose XML documents from relational tables

ABC Corporation performs most of its transaction processing using relational tables. For example, customer account information is maintained in relational tables within the database, and the information is not part of an XML document. One important function of the application is to compose the XML summary report documents to capture the first time a customer places an order. These XML summary reports will be sent from each factory to the corporate server. This processing requires data stored in DB2 tables for new customers to be converted into an XML summary report. The table for storing customer data is defined by the CREATE TABLE statement, as shown in Listing 64.

```
CREATE TABLE CUSTOMERS (
      cust_id         BIGINT GENERATED ALWAYS AS IDENTITY,
      cust_name      VARCHAR(1000),
      add_timestamp TIMESTAMP,
      cust_phone     VARCHAR(50),
      cust_address   VARCHAR(200),
     PRIMARY KEY (cust_id),
     UNIQUE (cust_name))
```

Listing 64: Customer table definition

The examples that follow assume that two rows have been inserted into the *customers* table using the INSERT statement in Listing 65.

```
INSERT INTO customers(cust_name, add_timestamp, cust_phone, cust_address)
VALUES('First Automobile Works',
       '2012-06-14-05.20.00.000000', '86-10-12345678', 'Dongfeng St.' ),
      ('Ultimate Automobile Works',
       '2012-06-15-05.30.00.000000',  '99-10-12345678', 'Main St.' )
```

Listing 65: Insert rows in customers

Customer data needs to be combined with the part_name, part_number, order_timestamp columns from the *orders* table (created in Listing 57) to produce the XML document shown in Listing 66.

```
<?xml version="1.0" encoding="UTF-8"?>
<daily_summary xmlns="http://www.abccompany.com">
 <new_customer_report>
  <name>First Automobile Works</name>
  <phone>86-10-12345678</phone>
  <address>Dongfeng St.</address>
  <added_datetime>2012-06-14T05:20:00.000000</added_datetime>
  <orders>
   <order_infor>
    <part_name>Valve</part_name>
    <quantity>1000</quantity>
    <order_datetime>2012-06-14T05:20:00.000000</order_datetime>
   </order_infor>
  </orders>
 </new_customer_report>
 <new_customer_report>
  <name>Ultimate Automobile Works</name>
  <phone>99-10-12345678</phone>
  <address>Main St.</address>
  <added_datetime>2012-06-15T05:30:00.000000</added_datetime>
  <orders>
   <order_infor>
    <part_name>Flywheel</part_name>
    <quantity>2000</quantity>
    <order_datetime>2012-06-14T10:20:00.000000</order_datetime>
   </order_infor>
  </orders>
 </new_customer_report>
</daily_summary>
```

Listing 66: Daily summary XML document

A graphical representation of the process is shown in Figure 4.

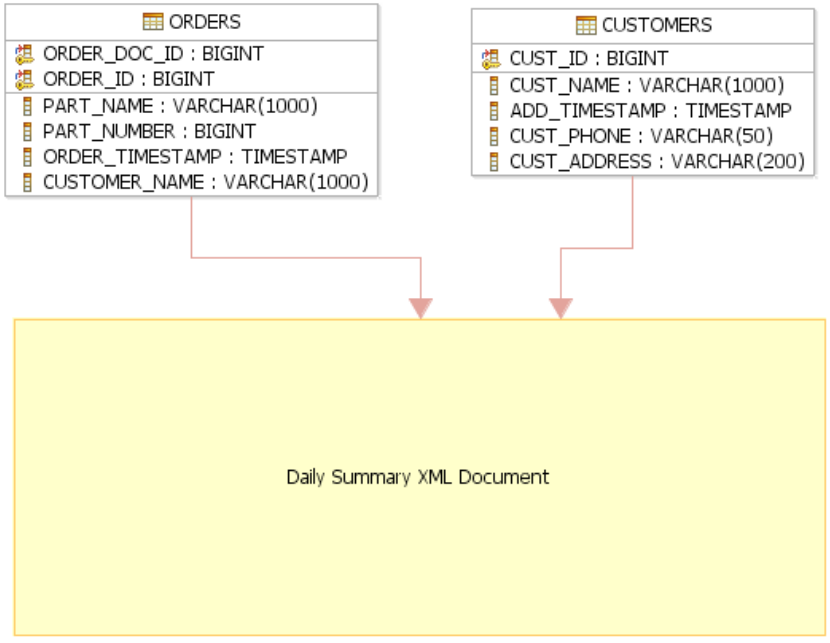

Figure 4: Composition process

In XML Extender, DAD files are used for composing XML. The db2xml.dxxRetrieveXMLClob procedure composes an XML document from an enabled XML collection. The procedure accepts a collection name as a parameter; collection needs to have a DAD file associated with it, and this provides XML Extender with a mapping for creating the XML document. The db2xml.dxxGenXMLClob procedure accepts a DAD file and applies the mappings defined in it to generate one or more XML documents.

Although a DAD file might specify a mapping that results in many XML documents being generated, both the db2xml.dxxRetrieveXMLClob and db2xml.dxxGenXMLClob are able to return only the first XML document that is generated. In many practical use cases, only one XML document needs to be generated and the restriction is not a problem. On the other hand, if the generation and processing of many XML documents is required, this limitation can be a problem.

Listing 67 shows an example of a DAD file that can be used to produce the output shown in Listing 68 using the db2xml.dxxGenXMLClob procedure.

The XML Extender support does not make it easy to get the hierarchical relationships represented correctly. A unique table reference is required for each level of elements that have content (other than child elements) in them. In addition, support for namespaces is almost non-existent. With XML Extender, it is often necessary to do some extra programming using the SYSDUMMY1 table to get the structure of the document correct and to get the namespace declaration included.

XML Extender is not able to handle very many variations on the SQL query. The order of the columns in the select list must appear exactly as they are presented in Listing 67. The columns are expected to be specified in a top-down order by the hierarchy of the XML document structure. In addition, the columns and their ordering in the order by clause contribute to the determination of the document structure. A more complete definition of how the SQL_Stmt element works can be found in the XML Extender Administration and Programming guide. A link to this document is in the “Resources” section. Due to the fact that the XML Extender support is dependent on the way the SQL statement is written, it is difficult to write more complex queries.

```
<?xml version="1.0" encoding="UTF-8"?>
<DAD><validation>NO</validation>
<Xcollection>
<SQL_stmt>
SELECT doc_namespace, cust_name, add_timestamp, cust_phone, cust_address,
part_name,  part_number, order_timestamp
FROM TABLE (SELECT 'http://www.abccompany.com' AS DOC_NAMESPACE
              FROM sysibm.sysdummy1) AS D,
     TABLE (SELECT cust_name, add_timestamp, cust_phone,
                   cust_address  FROM customers) AS C,
     TABLE (SELECT db2xml.generate_unique() as id,
                   customer_name, part_name, part_number, order_timestamp
              FROM orders) AS O
WHERE (c.cust_name = o.customer_name) AND
       C.add_timestamp > TIMESTAMP('2012-06-14 00:00:00')
ORDER BY doc_namespace, cust_name, part_name;
</SQL_stmt>
<prolog>?xml version="1.0"?</prolog>
<root_node>
 <element_node name="daily_summary">
  <attribute_node name="xmlns">
   <column name="DOC_NAMESPACE" />
  </attribute_node>
  <element_node name="new_customer_report" multi_occurrence="YES">
  <element_node name="name">
   <text_node>   <column name="CUST_NAME" />
   </text_node>
  </element_node>
  <element_node name="phone">
   <text_node>   <column name="CUST_PHONE" />
   </text_node>
  </element_node>
  <element_node name="address">
   <text_node>   <column name="CUST_ADDRESS" />
   </text_node>
  </element_node>
  <element_node name="added_datetime">
   <text_node>   <column name="ADD_TIMESTAMP" />
   </text_node>
  </element_node>
  <element_node name="order_infor" multi_occurrence="YES">
   <element_node name="part_name">
    <text_node>   <column name="PART_NAME" />
    </text_node>
   </element_node>
   <element_node name="quantity">
    <text_node>   <column name="PART_NUMBER" />
    </text_node>
   </element_node>
   <element_node name="order_datetime">
    <text_node>  <column name="ORDER_TIMESTAMP" />
    </text_node>
   </element_node>
  </element_node>
 </element_node>
</element_node>
</root_node></Xcollection></DAD>
```

Listing 67: Sample DAD file for composition

Listing 68 contains the result of the daily summary XML document that is generated by the XML Extender product. This XML document is similar to the desired result (shown in Listing 66), but there is one important difference. In Listing 68, the timestamps are in the SQL lexical format, rather than being converted to the XDM xs:dateTime format. This will be a problem for XML aware applications that will expect dates and times to conform to the XML standards.

```
<?xml version="1.0"?>
<daily_summary xmlns="http://www.abccompany.com">
  <new_customer_report>
    <name>First Automobile Works</name>
    <phone>86-10-12345678</phone>
    <address>Dongfeng St.</address>
    <added_datetime>2012-06-14-05.20.00.000000</added_datetime>
    <order_infor>
      <part_name>Valve</part_name>
      <quantity>1000</quantity>
      <order_datetime>2012-06-14-05.20.00.000000</order_datetime>
    </order_infor>
  </new_customer_report>
  <new_customer_report>
    <name>Ultimate Automobile Works</name>
    <phone>99-10-12345678</phone>
    <address>Main St.</address>
    <added_datetime>2012-06-15-05.30.00.000000</added_datetime>
    <order_infor>
      <part_name>Flywheel</part_name>
      <quantity>2000</quantity>
      <order_datetime>2012-06-14-10.20.00.000000</order_datetime>
    </order_infor>
  </new_customer_report>
</daily_summary>
```

Listing 68: Output from db2xml.dxxGenXMLClob

Designing, coding, and maintaining a DAD file is an expensive process. DAD files are unique to the XML Extender option and are not based on an industry standard. Significant time can be spent attempting to master and maintain DAD files that are used by only a minority of database developers.

In contrast, SQL/XML publishing functions are based on an industry standard. An industry-standard function greatly improves platform independence, and ensures the existence of improved technical support and supporting documentation.

SQL XML publishing functions

With DB2 for i 7.1, the task of publishing relational data as an XML document is entirely accomplished using SQL queries. XML publishing functions and namespace declarations are used within the query to create XML nodes and to convert relational data types into XML data types. When compared to the XML Extender option, the XML publishing functions provide the same advantage of improved data control that is mentioned throughout this article.

DB2 for i provides the following set of scalar publishing functions for constructing each XML node type.

- XMLATTRIBUTES
- XMLCOMMENT
- XMLDOCUMENT
- XMLELEMENT
- XMLPI
- XMLTEXT

An additional set of functions allow the creation of siblings within the XML document, rather than a parent/child (nesting) relationship.

- XMLCONCAT
- XMLAGG

The XMLCONCAT scalar publishing function concatenates multiple XML values into a single value. The XMLAGG function is an aggregate function that aggregates values from multiple rows into a single XML value.

Although these functions provide enough functionality to construct an XML document, several additional publishing functions are included for convenience. The results of an invocation of one of these functions can be obtained by using other publishing functions, but the following functions provide a simpler way of coding common scenarios.

- XMLFOREST
- XMLROW
- XMLGROUP

The XMLFOREST scalar function is particularly useful as it performs the task of both the XMLCONCAT and XMLELEMENT publishing functions in a single call. This function is frequently used to create several new elements at the same level within an XML document without requiring multiple calls to XMLCONCAT and XMLELEMENT. Listing 69 shows a query where XMLFOREST is used to create two child elements, nested within a parent element *root*.

```
SELECT  XMLELEMENT(NAME "root",
                     XMLFOREST(1 AS "element1", 2 AS "element2")
                     )
FROM SYSIBM.SYSDUMMY1
```

Listing 69: Query using XMLFOREST

The XML document that results from the query in Listing 69 is shown in Listing 70.

```
<root>
  <element1>1</element1>
  <element2>2</element2>
</root>
```

Listing 70: Results of a query using XMLFOREST

The XMLROW scalar function converts one or more values into the contents of new child elements (or optionally attributes) of a new element. This function can be used to avoid the need for directly using the XMLELEMENT and XMLCONCAT functions to construct an XML document for each row in a result set. An example of the XMLROW expression is shown in Listing 71.

```
SELECT XMLROW(1 AS "a", 2 AS "b", 3 AS "c"
                OPTION ROW "root_element")
FROM SYSIBM.SYSDUMMY1
```

Listing 71: Query using XMLROW

A single row results from this query, which contains the XML document is shown in Listing 72.

```
<root_element>
  <a>1</a>
  <b>2</b>
  <c>3</c>
</root_element>
```

Listing 72: Result of a query using the XMLROW function

The XMLGROUP aggregate function is an aggregate version of XMLROW. This function essentially combines the capabilities of the XMLROW and XMLAGG functions. A sample query using XMLGROUP is shown in Listing 73.

```
SELECT XMLGROUP(rs.col1         AS "col1",
                   rs.col2         AS "col2",
                   rs.col3         AS "col3"

                   ORDER BY rs.col1 DESC
                   OPTION ROW "row_element" ROOT "result_set"
                  )

FROM TABLE(VALUES
                     (1, 'Row_1-col_2', 'Row_1-col_3'),
                     (2, 'Row_2-col_2', 'Row_2-col_3')
            ) rs(col1, col2, col3)
```

Listing 73: Query using the XMLGROUP function

The resulting document from the query in Listing 73 is shown in Listing 74. As the SELECT statement does not contain a GROUP BY clause, the aggregation results in only a single row. The XMLAGG and XMLGROUP functions include an optional ORDER BY clause. In this example, the ORDER BY clause is

used to ensure that the row_element elements are ordered (descending) by rs.col1 in the resulting document.

```
<result_set>
  <row_element>
    <col1>2</col1>
    <col2>Row_2-col_2</col2>
    <col3>Row_2-col_3</col3>
  </row_element>
  <row_element>
    <col1>1</col1>
    <col2>Row_1-col_2</col2>
    <col3>Row_1-col_3</col3>
   </row_element>
</result_set>
```

Listing 74: Result set from the XMLGROUP function

Namespace declarations

The XMLNAMESPACES declaration can be specified as an argument of the XMLELEMENT and XMLFOREST publishing functions. This declaration is used to define the in-scope namespaces for the input values of the XMLELEMENT or XMLFOREST function. When using SQL/XML publishing functions to construct an XML element or attribute, if the name of the element or attribute is qualified with a namespace prefix, then the prefix must be *in-scope*. In other words, the prefix must be mapped to a URI by using an XMLNAMESPACES declaration. The scope of namespace mappings defined by the XMLNAMESPACES declaration includes the XMLELEMENT or XMLFOREST function for which the declaration is an argument, and any nested XML publishing functions. The XMLNAMESPACES declaration can also define the default namespace URI that is used when constructing elements that are not qualified with a namespace prefix.

The SQL query shown in Listing 75 generates the summary document shown in Listing 66. The XMLNAMESPACES declaration is used so that all of the constructed elements are defined in the namespace, *http://www.abccompany.com*.

```
WITH
order_infor AS (
 SELECT
         XMLAGG(
          XMLELEMENT(NAME "order_infor",
            XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
            XMLFOREST(p.part_name           AS "part_name",
                      p.part_number           AS "quantity",
                      p.order_timestamp    AS "order_datetime"
                     )
            )
         ORDER BY p.part_name
        ) -- XMLAGG
      AS order_infor_xml,
      customer_name AS cust_name
  FROM orders p
  GROUP BY p.customer_name
),

new_customer_report AS (
 SELECT
    XMLELEMENT(
       NAME "new_customer_report",
       XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
       XMLFOREST(
                    cr.cust_name      AS "name",
                    cr.cust_phone     AS "phone",
                    cr.cust_address   AS "address",
                    cr.add_timestamp  AS "added_datetime",
                    order_infor.order_infor_xml AS "orders"
                   )
     )  AS new_customer_report_xml,
    cr.cust_name
  FROM
   customers cr INNER JOIN  order_infor
   ON (cr.cust_name = order_infor.cust_name)
   WHERE cr.add_timestamp > TIMESTAMP('2012-06-14 00:00:00')
)

-- document root
SELECT
 XMLDOCUMENT(
  XMLELEMENT(
    NAME "daily_summary",
    XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
    XMLAGG(new_customer_report.new_customer_report_xml
              ORDER BY new_customer_report.cust_name)
   )
 ) – XMLDOCUMENT
 AS DAILY_SUMMARY
FROM new_customer_report
```

Listing 75: SQL query using XML publishing functions

Query design

These SQL queries that construct XML values can be complicated. A trick to writing XML publishing queries is to code common table expressions for repeating elements starting from the inside (or bottom) of the XML document and working outward (or upward) towards the root node of the document.

For example, to create the query in Listing 75, a developer could have started by determining how to build the list of *order_infor* elements (grouped by each customer), using the data in the *orders* table (Listing 57). The next step is to move up a level and build the new_customer_report elements, which involves a join between the previous results and rows in the *customers* table (Listing 64). Finally, the root element, daily_summary, is constructed around the aggregation of those rows.

Representation of XML values obtained from SQL

The result of the query in Listing 75 is an XML document that stores its data in the appropriate XML type that corresponds to the original SQL type. (In other words, SQL timestamps are casted to xs:dateTime values). This is likely what the ABC Corporation wants because this encoding allows the XML values to be handled by XML-aware applications. However, as mentioned earlier, the composition functions included with XML Extender will not do this conversion automatically; as shown in Listing 68.

Representing the value as an xs:dateTime type as opposed to an xs:string type (which happens to be an SQL Timestamp) is usually best practice. However, if an SQL representation is necessary, then a simple cast of the timestamp to a character value can help to achieve the required result.

Listing 76 contains the modified query with the cast of the timestamp value highlighted in bold format. This causes the added_datetime element to be built with string data, rather than an xs:dateTime type. The output is the same document that was produced by the db2xml.dxxGenXMLClob procedure in Listing 68.

```
WITH order_infor AS (
 SELECT XMLAGG(
          XMLELEMENT(NAME "order_infor",
            XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
            XMLFOREST(p.part_name           AS "part_name",
                        p.part_number         AS "quantity",
                        p.order_timestamp     AS "order_datetime"
                       )
            )
         ORDER BY p.part_name
        ) -- XMLAGG
      AS order_infor_xml,
      customer_name AS cust_name
  FROM orders p
  GROUP BY p.customer_name
),

new_customer_report as (
 select
    XMLELEMENT(
       NAME "new_customer_report",
       XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
       XMLFOREST(
                    cr.cust_name      AS "name",
                    cr.cust_phone     AS "phone",
                    cr.cust_address   AS "address",
                    CAST(cr.add_timestamp AS VARCHAR(26)) AS "added_datetime",
                    order_infor.order_infor_xml AS "orders"
                   )
     )  AS new_customer_report_xml,
    cr.cust_name
  FROM
   customers cr INNER JOIN  order_infor
   ON (cr.cust_name = order_infor.cust_name)
   WHERE cr.add_timestamp > TIMESTAMP('2012-06-14 00:00:00')
)
-- document root
select
 XMLDOCUMENT(
  XMLELEMENT(
    NAME "daily_summary",
    XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
    XMLAGG(new_customer_report.new_customer_report_xml
             ORDER BY new_customer_report.cust_name)
   )
 ) – XMLDOCUMENT
 AS DAILY_SUMMARY
FROM new_customer_report
```

Listing 76: Modified query to represent timestamps as strings of SQL ISO timestamps

Validation of XML documents

ABC Corporation prefers to validate that the summary documents received by the corporate headquarters conform to an agreed upon XML schema. Using an XML schema ensures that both the factory and headquarters can validate that the summary XML documents follow the agreed upon model.

XML documents can be validated using a schema. There are two commonly used industry-standard schema definitions for XML:

- Document type definition (DTD)
- W3C XML Schema Definition (XSD)

To simplify this paper, the terms DTD file and XSD file are used to describe an instance of a DTD or XSD, however an instance of a DTD or XSD need not exist in a file as such, as it might exist in a database row, or as a stream of bytes sent between applications.

The XSD language is a successor to the DTD language and is both more powerful and more extensible. Thus, it is recommended that XSD files be used instead of DTD files. Many tools, even ones that do not offer IBM i features (such as IBM Rational Application Developer), are available for developing an XML schema; these tools can be used to create XSD files from existing DTD files or from XML documents. The XSD language is an industry standard and therefore XSD design tools do not need to have any IBM i or DB2 awareness to be useful.

XML Extender offers several ways to validate an XML document. XML documents can be automatically validated during insert by registering a DTD file and binding it to a DAD file for an XML collection. XML Extender also provides the db2xml.svalidate and db2xml.dvalidate UDFs for explicitly validating XML documents against the XSD files or DTD files respectively.

The integrated DB2 for i XML functionality only supports validation of XML documents using XSD files. DTD files need to be converted to XSD files in order to use the schema to validate XML documents.

An XSD file can be created or edited with any text editor. However, most developers might want to use a toolset that is capable of designing and editing XSD files. Finding the tools to do this is not a problem because XSD is an industry standard; creating an XSD file does not require an IBM i toolset. For the example scenario, an XSD file for the summary document in Listing 66 was generated from an XML document using IBM Rational Application Developer. The resulting XSD file was modified so that the elements have the correct type annotations, and finally the XSD file was copied into a directory in IFS. This XSD file is shown in the following listing.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://www.abccompany.com"
            xmlns:Q1="http://www.abccompany.com"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="daily_summary">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded"
                     ref="Q1:new_customer_report"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="orders">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Q1:order_infor"
                     maxOccurs="unbounded" minOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="order_infor">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Q1:part_name"
                     maxOccurs="1" minOccurs="1"/>
        <xsd:element ref="Q1:quantity" maxOccurs="1" minOccurs="1" />
        <xsd:element ref="Q1:order_datetime"
                     maxOccurs="1" minOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="address" type="xsd:string"/>
  <xsd:element name="new_customer_report">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Q1:name" maxOccurs="1" minOccurs="1"/>
        <xsd:element ref="Q1:phone" maxOccurs="1" minOccurs="1"/>
        <xsd:element ref="Q1:address" maxOccurs="1" minOccurs="1"/>
        <xsd:element ref="Q1:added_datetime"
                     maxOccurs="1" minOccurs="1"/>
        <xsd:element ref="Q1:orders"
                     maxOccurs="1" minOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="part_name" type="xsd:string"/>
  <xsd:element name="added_datetime" type="xsd:dateTime" />
  <xsd:element name="name" type="xsd:string"/>
  <xsd:element name="order_datetime" type="xsd:dateTime"/>
  <xsd:element name="quantity" type="xsd:integer"/>
  <xsd:element name="phone" type="xsd:string"/>

</xsd:schema>
```

Listing 77: XSD for daily summary XML document

An XSD file is often difficult to read and understand. Using tools to generate the XSD file can make this worse because the result of automatically generating an XSD file is typically a schema that defines the structural constraints on a set of sample data, rather than a well thought out model that describes the

relationships and constraints within the data. The complexity of the schema design and generation is an important reason why tools are often also used to design and diagram XML schemas before registering them with DB2 for i. Figure 5 shows a diagram of the schema definition in Listing 77.

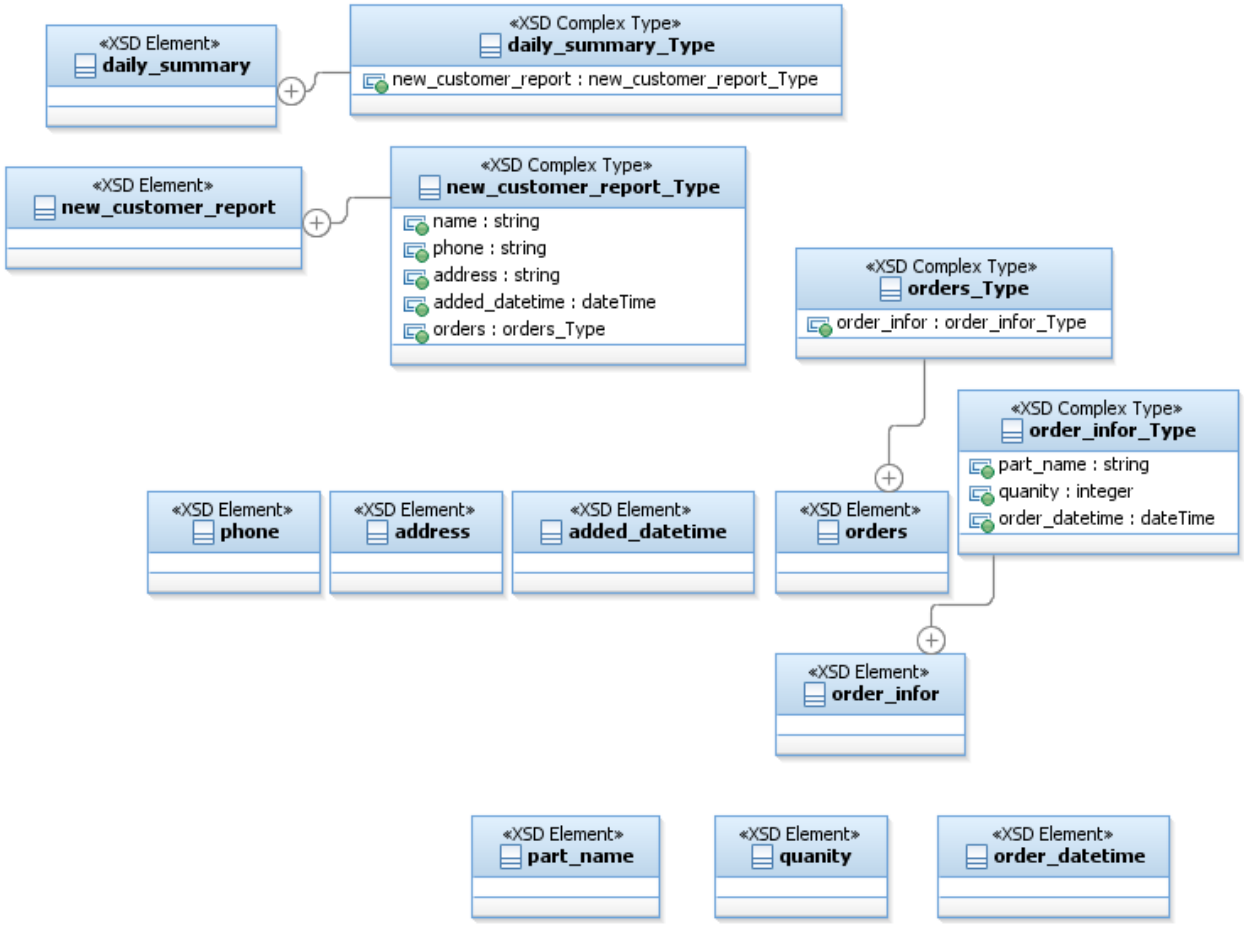

Figure 5: Diagram of daily summary XSD

Validating an XML schema with XML Extender

The XML Extender provides the db2xml.svalidate function for performing schema-based validation. In Listing 78, the db2xml.svalidate function validates an XML document stored in a stream file with a file path of /Summary/daily_summary.xml. Elements that are in the *http://www.abccompany.com* namespace are validated by the XSD found in the daily_summary.xsd file in the schemas directory.

```
VALUES db2xml.SValidate('/Summary/daily_summary.xml',     '',
                        'http://www.abccompany.com/schemas/daily_summary.xsd'   )
```

Listing 78: XML Extender db2xml.svalidate function

The result of the function is a boolean integer value indicating whether the XML document was successfully validated. A value of one indicates success and a value of zero indicates failure.

There are a number of different variations on the db2xml.svalidate function that allow different ways of specifying which schemas can be considered in the validation process, and whether the schemas are in a DB2 column or an IFS stream file. For example, if both the XML document and schema are stored in columns of a DB2 table, the db2xml.svalidate function can be invoked as shown in Listing 79.

```
SELECT db2xml.SValidate(doc, schema) FROM my_table
```

Listing 79: Alternate db2xml.svalidate invocation

Registering an XML schema with DB2 for i

DB2 for i 7.1 introduces a set of catalogs that contain information about XML schemas called the XML schema repository (XSR). On IBM i, an XML schema contains a binary representation of one or more XSD files. The binary representation is used to validate XML documents; DB2 for i will only resolve XSD files that are included in this representation. Each XML schema is represented by catalog entries and a named object that is called an XSR object. An XSR object is an IBM i object and is housed within an SQL schema (library).

Registering an XML schema and adding XSD files

Before an XML schema can be used to validate an XML document in DB2 for i, the XML schema must be registered with the database. When an XML schema is registered with DB2 for i, the DB2 engine creates an XSR object with the specified SQL identifier.

The XSR_REGISTER stored procedure registers the XML schema in DB2 (creating the XSR object in the process), and adds the first XSD file to the XML schema. The first XSD file is referred to as the primary XSD for the XML schema.

Listing 80 shows a call to the XSR_REGISTER procedure to create an XML schema for the daily summary schema presented in Listing 77.

```
CALL XSR_REGISTER(
           'sql_schema',
           'daily_summary_xsr',
           'daily_summary.xsd',
           GET_XML_FILE('/schemas/daily_summary.xsd'),
           NULL)
```

Listing 80: Call the XSR_REGISTER procedure

The first two parameters of the procedure call in Listing 80 specify that the XSR object will be created in the SQL schema named sql_schema and will have the name daily_summary_xsr. The combination of the schema and object name of the XSR object is also referred to as the SQL identifier or relational ID of the XML schema.

The third parameter defines the location URI for the XSD file. The location is an XML concept, and therefore, the SQL naming rules do not apply to this parameter. On DB2 for i, an XSD file is physically located in the XML schema, however, XML documents and XSD files are usually constructed to be used in many environments, including those environments which locate to the XSD file using a path name or web address. DB2 for i will make use of the location URI in several situations in order to

provide behavior that is more consistent with other XML solutions. Consequently, although this parameter can be a NULL value, proving a value is usually advised. The location is discussed in more detail later in this paper.

The fourth parameter of the XSR_REGISTER procedure is for passing in the contents of the XSD file. This data needs to be passed as a BLOB value. In this example, it has been loaded from a stream file using the GET_XML_FILE function.

The fifth parameter of the XSR_REGISTER procedure provides a way to add additional user-defined information (properties) about the XML schema definition, such as a version number. These properties can be retrieved from the QSYS2.XSROBJECTCOMPONENTS catalog. This parameter is allowed to be the null value if there is no user-defined information.

After the XSR_REGISTER procedure call, the XML schema is registered, but cannot be used for validation until all XSD files associated with the XML schema have been added to the repository and the XSR_COMPLETE stored procedure has been invoked. This example needs only one XSD file, but it is fairly common for an XSR to have many XSD files contained within it. The primary XSD might need to import or include types and elements from other XSD files. If this is necessary, the XSR_ADDSCHEMADOC procedure can be used to add additional XSD files. XSD files added by the XSR_ADDSCHEMADOC procedure are called secondary XSD files. The XSR_ADDSCHEMADOC procedure accepts the same parameters as the XSR_REGISTER procedure.

A secondary XSD file must be directly or indirectly connected to the primary XSD file. In other words, the primary XSD file must directly or indirectly import or include all secondary XSD files. If an XSD file is not connected, or if a required XSD file has not been added, an error will occur when the XSR_COMPLETE procedure is called to complete the XML schema.

Assigning a target namespace and location

A target namespace is associated with each XSD file. This is the namespace (or lack of a namespace) that contains the elements defined by the XSD file. During XML validation, an element is validated using the definitions in an XSD file that has a target namespace that matches the element's namespace. The target namespace is defined within the XSD file itself and is not specified as a parameter when adding the XSD file to an XML schema. The XSD file in Listing 77 defines the target namespace of the daily_summary.xsd XSD file to be http://www.abccompany.com.

The target namespace of the primary XSD file determines the primary target namespace of the XML schema. While the target namespace defines the scope of what is defined by the XSD file, it does not provide any information to identify the location of the XSD file within the XML schema. This location information is needed by processes that refer to the XSD file.

The location is provided as a parameter when the XSD file is added to the XML schema using either the XSR_REGISTER or XSR_ADDSCHEMADOC stored procedures. The location is essentially the name that is used to refer to the XSD file. It needs to be set correctly in order to successfully complete the schema registration.

The location is needed during XML schema registration for importing and including types and elements from another XSD file. An XSD file can specify that elements and types defined by a different XSD file should be imported by using the schema location. This example uses only one XSD

file, but if daily_summary.xsd referenced types defined in a second XSD file named daily_summary_types.xsd, an import element (similar to the one in Listing 81) would exist in the daily_summary.xsd file.

```
<import schemaLocation="daily_summary_types.xsd"/>
```

Listing 81: Import element

In order for DB2 to import a secondary XSD file, the imported XSD file must be added to the XML schema using the XSR_REGISTER or XSR_ADDSCHEMADOC procedures. In addition, the schema location that is assigned to the imported XSD file must match the schemaLocation attribute referenced on the import.

The location can also play an important role during XML document validation. The location of the XSD file that is provided on the XSR_REGISTER stored procedure call becomes the primary schema location; the primary location and primary target namespace can be used during validation of an XML document to identify which XML schema to use. Validation is discussed in more detail later in this paper.

The location URI that is supplied when an XSD file is added to an XML schema must match the location that will be provided when importing the XSD file and must also match the location that is used to identify the XML schema when validating an XML document.

Completing the schema registration using the XSR_COMPLETE stored procedure

After all XSD files have been added to the XML schema, the XSR_COMPLETE procedure must be invoked as shown in Listing 82. The procedure compiles the XSD files in the schema into a binary format that can be used later by DB2 for validation. After this procedure is called, it is no longer possible to modify or add additional XSD files to the XML schema without dropping and re-creating the associated XSR object.

```
CALL XSR_COMPLETE(
  'sql_schema',
  'daily_summary_xsr',
  NULL,
  0)
```

Listing 82: XSR_COMPLETE procedure call

The first two parameters of the XSR_COMPLETE procedure call specify which XSR object is being completed. The third parameter allows some additional user-defined information (properties) to be associated with the XSR object, such as a version number. This parameter can be NULL. These properties can be retrieved from the QSYS2.XSROBJECTS catalog. The final parameter of the XSR_COMPLETE procedure indicates whether the schema contains decomposition annotations. A value of 0 indicates that the schema does not contain annotations. Decomposition annotations are discussed later in this paper.

Validating XML documents with built-in functions

The built-in XMLVALIDATE function corresponds to the db2xml.svalidate user-defined function provided by XML Extender. The db2xml.svalidate function simply returns a value flag of 1 or 0 to indicate whether the XML document is valid according to the XML schema. The XMLVALIDATE built-in function takes a

different approach and returns an XML value for the validated document whenever the document is valid. An error is returned if the document is not valid.

An advantage of returning a validated XML document instead of a boolean flag is that the validated XML document includes default values that have been defined by the schema for missing elements and attributes in the XML document. One example of where this might be useful is if an XML schema evolves to include new elements or attributes. The new schema definition can provide a default value for each new element and attribute if the element or attribute is missing. The XML value that is returned from the XMLVALIDATE function contains the new elements and attributes. Application logic does not have to include special code to deal with missing data in documents that were created according to older versions of an XML schema.

The XMLVALIDATE function provides the optional ACCORDING TO XMLSCHEMA clause that can be used to explicitly specify the XML schema that can be used for validation. An example of how to use this clause to identify an XML schema, by providing the XSR name that was registered in Listing 80 is shown in Listing 83. The data in the /Summary/daily_summary.xml file is assumed to appear similar to what is shown in Listing 66.

```
VALUES
 XMLVALIDATE(
   XMLPARSE(DOCUMENT
              GET_XML_FILE('/Summary/daily_summary.xml ')
   )
   ACCORDING TO XMLSCHEMA ID
   sql_schema.daily_summary_xsr
 )
```

Listing 83: XMLVALIDATE with ACCORDING TO

Most database applications use the name of the XSR object to reference the XML schema; however, there might be cases where it is more convenient to identify an XML schema using the target namespace and location of the primary XSD. For example, suppose that ABC Corporation has many applications and each application shares the same XML schema. The XML schema might be registered with DB2 once for the entire database (possibly by the first application that uses the XML schema). Different systems and databases will have different applications and therefore possibly different SQL identifiers for the same XML schema. Although the name of the XSR object for the XML schema cannot always be determined in advance, the namespace URI and location can be used to identify the XML schema.

Listing 84 shows how to specify these identifiers when using XMLVALIDATE. The URI and location must identify exactly one XML schema or an error occurs when the function is evaluated.

```
VALUES XMLVALIDATE(
   XMLPARSE(DOCUMENT GET_XML_FILE('/Summary/daily_summary.xml '))
   ACCORDING TO XMLSCHEMA URI 'http://www.abccompany.com'
                              LOCATION 'daily_summary.xsd' )
```

Listing 84: XMLVALIDATE according to Schema URI and location hint

When specifying the XML schema URI and XML schema LOCATION, it is necessary to supply only the necessary information to uniquely identify a single schema. For example, if there is only one XML schema for the URI *http://www.abccompany.com*, it is not necessary to also specify the LOCATION, as the URI provides DB2 for i with sufficient information to find the XML schema. Best practice would dictate specifying both values because this reduces the possibility of errors due to duplicate XML schemas later

on. It is possible an XML schema may need to co-exist with an updated version that uses the same namespace with a different location.

An XML schema can define many global elements that might be the root of an XML document. If it is necessary to validate that a specific element is the root element of the document being validated, the ELEMENT clause should be used within the XMLVALIDATE function. Listing 85 includes a validation example which verifies that the daily_summary element in namespace *http://www.abccompany.com* is the root element of the input XML document.

```
VALUES XMLVALIDATE(
   XMLPARSE(DOCUMENT GET_XML_FILE('/Summary/daily_summary.xml ')   )
   ACCORDING TO XMLSCHEMA ID    sql_schema.daily_summary_xsr
                                NAMESPACE 'http://www.abccompany.com'
                                ELEMENT "daily_summary"
)
```

Listing 85: XMLVALIDATE with a valid element clause

When the XMLVALIDATE function is used to validate an XML document without specifying the ACCORDING TO XMLSCHMEA clause, DB2 for i examines the XML document to determine the XML schema. This kind of validation is called implicit validation.

When using implicit validation, the XML document indicates which XML schema should be used for validation by providing the target namespace and a location hint of the primary XSD file for the XML schema. Depending on whether the elements are defined to be in a namespace, either the xsi:schemaLocation or xsi:noNamespaceSchemaLocation attribute can be used to accomplish this. Listing 86 shows how the root element of the daily summary (introduced in Listing 66) can be written to provide a schema location.

```
<daily_summary
             xmlns="http://www.abccompany.com"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation=”http://www.abccompany.com daily_summary.xsd”>
….
```

Listing 86: Root Element that specifies a XSD for validation

For the example in Listing 86, the xsi:schemaLocation indicates that the target namespace of http://www.abccompany.com has an XSD file that can be found using the hint daily_summary.xsd file. The target namespace and schema location must match the primary target namespace and location of an XML schema that can be used for the validation. An error occurs if the XML schema cannot be located, or if more than one XML schema matches the search.

It is possible to include more than one namespace in an XML document, and each namespace can have a xsi:schemaLocation hint defined for it. In other environments, the validation code might have to locate an XSD file for each namespace using the provided location hint. DB2 for i requires the information it needs to be contained inside the XML schema; therefore, only the target namespace and location of the primary XSD is used to identify the XML schema, regardless of how many namespace and location pairs are defined within the XML document.

In some XML documents, the elements might not exist in any namespace. In that case, the location hint is supplied to the validation code using the xsi:noNamespaceSchemaLocation attribute.

The location hint is called a hint because it is designed so that the application requesting the validation can override the location hint, and provide an explicit location for the schema. An XML schema that is specified by the ACCORDING TO clause overrides the schema location hint in the XML document (if there is a hint).

Assume that the daily_summary.xml document (shown in Listing 66) has been modified so that the xsi:schemaLocation attribute is defined for the root element (as shown in Listing 86) and stored in the IFS stream file /Summary/daily_summary.xml, the validated XML document can be obtained by invoking the XMLVALIDATE function shown in in the following listing.

```
VALUES XMLVALIDATE(
   XMLPARSE(DOCUMENT GET_XML_FILE('/Summary/daily_summary.xml') )
 )
```

Listing 87: XMLVALIDATE function

The DB2 validation processing finds the XML schema to use for validation by using the target namespaces and schema location hints defined by the document.

Although it offers great flexibility, implicit validation significantly decreases performance because the XML document must be examined one more time before validation. Allowing an XML document to define how it should be validated is also an issue in cases where the document is not from a trusted source. Customers are advised to use the ACCORDING TO clause to explicitly specify which XML schema should be used for validation whenever possible.

Due to the fact that an error is returned for invalid documents and the XML document is returned for valid documents, the XMLVALIDATE function can be used within an INSERT or UPDATE statement to ensure that only valid XML data is assigned to an XML column. The example in Listing 88 invokes the XMLVALIDATE function from an INSERT statement. In this example, the xml_doc column is defined with the XML data type in the table named, daily_summary_table.

```
INSERT INTO daily_summary_table(xml_doc)
VALUES
 XMLVALIDATE(
   XMLPARSE(DOCUMENT GET_XML_FILE('/Summary/daily_summary.xml ') )
   ACCORDING TO XMLSCHEMA ID sql_schema.daily_summary_xsr
 )
```

Listing 88: Using XMLVALIDATE on an INSERT statement

Annotated decomposition

The applications used by the ABC Corporation's headquarters are based on a relational model. It is a requirement that the data contained within the XML document for the daily summary (Listing 66) be decomposed (shredded) into multiple relational tables. With XML Extender, this can be accomplished by defining the mapping in a DAD file and using the db2xml.dxxShredXml stored procedure. The structure of a DAD file for this task closely resembles Listing 58 with the exception that multiple tables must be referenced within the DAD. This approach for decomposition is not easy to implement, and suffers from similar problems as decomposing an XML document into a single table.

A better decomposition solution can be implemented in DB2 for i 7.1 by updating the XML schema that was shown in Listing 77 to include a mapping of the XML values to the columns in the target tables. After the addition of the relational mapping specifications, the XML schema document is known as an annotated XML schema. The annotated XML schema and XML document can be passed into the XDBDECOMPXML stored procedure to decompose the document.

Assume that the target tables, summary_customers and summary_orders, in schema sql_schema are created with the SQL statements in Listing 89. Two tables are necessary, because of the potential one-to-many relationship between first-time customer elements and the orders element.

```
CREATE TABLE sql_schema.summary_customers (
      cust_name      VARCHAR(1000),
      add_timestamp TIMESTAMP,
      cust_phone     VARCHAR(50),
      cust_address   VARCHAR(200));


CREATE TABLE sql_schema.summary_orders (
   order_id          BIGINT GENERATED ALWAYS AS IDENTITY,
   cust_name         VARCHAR(1000),
   part_name         VARCHAR(1000),
   quantity          BIGINT,
   order_timestamp  TIMESTAMP);
```

Listing 89: Table definitions for summary_customers and summary_orders tables

The decomposition process of an XML document is graphically represented in Figure 6.

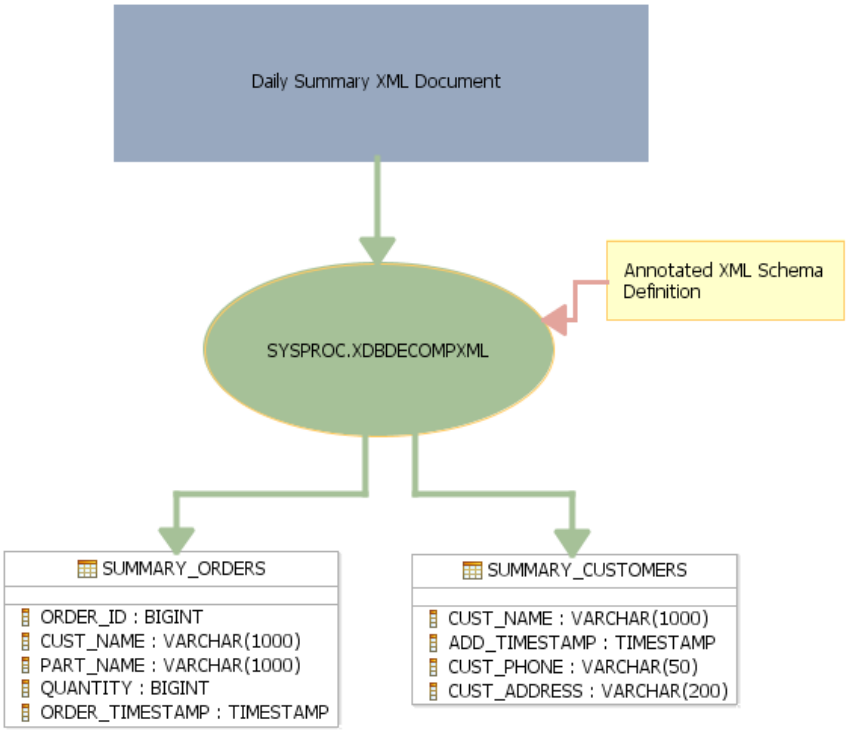

Figure 6: Decomposition process in DB2 for i 7.1

XML schema annotations

The W3C XML schema definition language allows XML schemas to contain annotations that are not used to validate the XML document. Instead, these annotations make extra information available to XML processing and validating tools. Annotations can be included as attributes of XSD components or as an annotation element as the first child of an XSD component.

Listing 90 shows a trivial XML schema that contains annotations. The definition of the element named root contains an attribute, annotation_ns:attrib, that does not exist in the XML schema namespace. This attribute is treated as an annotation, as its meaning is not defined by the XSD language. The definition of the root2 element shows a different approach. The xsd:annotation component indicates the existence of an annotation for the root2 element. The xsd:appinfo component indicates that the enclosed information will be used by an application. Using an xsd:annotation element is more verbose than the attribute approach, however, in some cases an xsd:annotation element is required because the annotation might need to include a complex data structure.

How the information in the annotations is used is not defined by the W3C standard. Instead, annotation usage is determined by the tool that is using the XML schema to process an XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://www.a_target_namespace.com"
            xmlns:annotation_ns="non_xsd_namespace"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!-- Define an element and include an annotation           as an attribute -->
<xsd:element name="root" annotation_ns:attrib="Data for tools" />
<!-- Define an element and include annotations
          as a child of the XSD element component -->
<xsd:element name="root2">
<xsd:annotation>
  <xsd:appinfo>
             <annotation_ns:myTool>
                  Data for tools
              </annotation_ns:myTool>
  </xsd:appinfo>
</xsd:annotation>
</xsd:element>
</xsd:schema>
```

Listing 90: Annotations in an XML schema

DB2 for i decomposition annotations

DB2 for i recognizes a number of XML schema annotations that are designed to be used for decomposing an XML document into relational tables.

Listing 91 shows the XSD file from Listing 77, with the addition of decomposition annotations. The decomposition annotations contain the information needed by the XDBDECOMPXML procedure to decompose an XML document into relational tables. This example XSD file in Listing 91 makes use of the most common annotations, which are described in detail in this paper.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://www.abccompany.com"
            xmlns:Q1="http://www.abccompany.com"
            xmlns:sql="http://www.ibm.com/xmlns/prod/db2/xdb1"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="daily_summary">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" ref="Q1:new_customer_report"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="orders">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="1" ref="Q1:order_infor"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="order_infor">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="1" minOccurs="1" ref="Q1:part_name"/>
        <xsd:element maxOccurs="1" minOccurs="1" ref="Q1:quantity"/>
        <xsd:element maxOccurs="1" minOccurs="1" ref="Q1:order_datetime"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="address"
                  sql:column="CUST_ADDRESS"
                  sql:locationPath=
         "/Q1:daily_summary/Q1:new_customer_report/Q1:address"
                  sql:rowSet="CUSTOMERS_ROWSET"
                  type="xsd:string"/>
  <xsd:element name="new_customer_report">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="1" minOccurs="1" ref="Q1:name"/>
        <xsd:element maxOccurs="1" minOccurs="1" ref="Q1:phone"/>
        <xsd:element maxOccurs="1" minOccurs="1" ref="Q1:address"/>
        <xsd:element maxOccurs="1" minOccurs="1" ref="Q1:added_datetime"/>
        <xsd:element maxOccurs="1" minOccurs="1" ref="Q1:orders"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="part_name"
                 sql:column="PART_NAME"
                 sql:locationPath=
"/Q1:daily_summary/Q1:new_customer_report/Q1:orders/Q1:order_infor/Q1:part_name"
                  sql:rowSet="ORDERS_ROWSET"
                  type="xsd:string"/>
  <xsd:element name="added_datetime"
                 sql:column="ADD_TIMESTAMP"
                 sql:locationPath=
"/Q1:daily_summary/Q1:new_customer_report/Q1:added_datetime"
                 sql:rowSet="CUSTOMERS_ROWSET"
                 type="xsd:dateTime"/>
```

```
<xsd:element name="name" type="xsd:string">
    <xsd:annotation>
      <xsd:appinfo xml:space="preserve">
        <sql:rowSetMapping sql:locationPath=
           "/Q1:daily_summary/Q1:new_customer_report/Q1:name">
          <sql:rowSet>CUSTOMERS_ROWSET</sql:rowSet>
          <sql:column>CUST_NAME</sql:column>
        </sql:rowSetMapping>
        <sql:rowSetMapping sql:locationPath=
          "/Q1:daily_summary/Q1:new_customer_report/Q1:name">
          <sql:rowSet>ORDERS_ROWSET</sql:rowSet>
          <sql:column>CUST_NAME</sql:column>
        </sql:rowSetMapping>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:element>
  <xsd:element name="order_datetime"
                 sql:column="ORDER_TIMESTAMP"
                 sql:locationPath=
"/Q1:daily_summary/Q1:new_customer_report/Q1:orders/Q1:order_infor/Q1:order_datet
ime"
                 sql:rowSet="ORDERS_ROWSET" type="xsd:dateTime"/>
  <xsd:element name="quantity" sql:column="QUANTITY"
                sql:locationPath=
"/Q1:daily_summary/Q1:new_customer_report/Q1:orders/Q1:order_infor/Q1:quantity"
              sql:rowSet="ORDERS_ROWSET" type="xsd:integer"/>
  <xsd:element name="phone" sql:column="CUST_PHONE"
             sql:locationPath=
                         "/Q1:daily_summary/Q1:new_customer_report/Q1:phone"
             sql:rowSet="CUSTOMERS_ROWSET"  type="xsd:string"/>
    <xsd:annotation>
    <xsd:appinfo xml:space="preserve">
      <sql:table>
        <sql:SQLSchema>SQL_SCHEMA</sql:SQLSchema>
        <sql:name>SUMMARY_ORDERS</sql:name>
       <sql:rowSet>ORDERS_ROWSET</sql:rowSet>
      </sql:table>
      <sql:table>
        <sql:SQLSchema>SQL_SCHEMA</sql:SQLSchema>
        <sql:name>SUMMARY_CUSTOMERS</sql:name>
        <sql:rowSet>CUSTOMERS_ROWSET</sql:rowSet>
      </sql:table>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:schema>
```

Listing 91: XML schema with annotations

All DB2 for i decomposition annotations are qualified by the namespace URI *http://www.ibm.com/xmlns/prod/db2/xdb1*. In the example XML schema found in Listing 91, this namespace is mapped to the prefix **sql**. However, other prefixes are frequently used. The decomposition annotation examples contained in the IBM i Information Center use **db2-xdb** as the prefix. The decision on which prefix value to use is entirely up to the programmer building the annotated XML schema. DB2 for i processes annotations using the associated namespace URI rather than the prefix.

Two annotations that are the critical features of the decomposition process are the sql:rowSet and sql:column annotations. These annotations are provided for each element or attribute to specify a mapping of the associated data value to a target table and column. These two annotations must be specified in order for the decomposition process to successfully complete.

The sql:table annotation defines the SQL table for the rowset. This annotation is optional and can be omitted. If this annotation is omitted, the name of the rowset is used as the table name, and the default schema is used as the table's schema. The default schema can be set with the sql:defaultSQLSchema annotation.

The sql:locationPath annotation is often used to describe the mapping of elements and attributes conditionally, depending on the location of the element or attribute in the XML document. The reason why this is useful requires a bit of explanation for how XSD files can be written to reuse an element's structure.

In an XSD file, an element can be declared globally, as a child of the xs:schema component, rather than locally as part of another element component's content. The global element can then be referenced from within other element components in the XML schema. This provides a way to reuse the definition of the global element, similar to how a global data structure might be reused within a program. A well-designed XML schema usually will not use this approach, and will instead, use real data types to facilitate reuse of structural ideas; however, automated tools for creating XML schemas most of the times cannot derive meaningful type information from an XML document. Thus, when a tool is used to generate an XML schema from a set of XML documents, the elements are usually declared globally, and then referenced at the locations in which they can appear in the XML document. Listing 77 was generated with automated tools, and so, it uses global elements and references. Listing 92 shows how the new_customer_report element was declared in Listing 77. Child elements of the new_customer_report element are declared by referring to global elements, rather than duplicating the declarations of these elements.

```
<xsd:element name="new_customer_report">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="1"
                   minOccurs="1" ref="Q1:name"/>
      <xsd:element  maxOccurs="1"
                   minOccurs="1" ref="Q1:phone"/>
      <xsd:element maxOccurs="1"
                   minOccurs="1" ref="Q1:address"/>
      <xsd:element maxOccurs="1"
                   minOccurs="1" ref="Q1:added_datetime"/>
      <xsd:element maxOccurs="1"
                   minOccurs="1" ref="Q1:orders"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Listing 92: new_customer_report element declaration

Child elements of the new_customer_report element in Listing 92 are declared by referring to global elements, rather than duplicating the declarations of these elements. Decomposition annotations cannot be included on any of the child elements of the new_customer_report element because they all reference global elements. However, decomposition annotations for these child elements can appear on the global element that is being referenced. This limitation creates some complexity because the global element might appear at multiple locations in the XML document, and each location can be decomposed differently.

This problem is resolved by using the sql:locationPath annotation. The annotation is used to describe the mapping for a specific element at a specific location. Listing 93 shows how the *address* global element has been annotated with a decomposition annotation that maps the *address* child element of the

new_customer_report element to the CUST_ADDRESS column. If the global address element was referenced by other child elements, then additional annotations can be added to the global address element to map address values to other columns.

The syntax for the sql:locationPath is based on a very small subset of the XPath syntax and is described in the IBM i Information Center. Refer to the link included in the “Resources” section at the end of this paper.

```
<xsd:element
         name="address"
         sql:column="CUST_ADDRESS"
         sql:locationPath="/Q1:daily_summary/Q1:new_customer_report/Q1:address"
         sql:rowSet="CUSTOMERS_ROWSET"
         type="xsd:string"/>
```

Listing 93: Declaration of the address global element

There are a number of other optional annotations that are not included in this example. These annotations allow for conditional shredding, transformation of the data that is inserted, and ordering of the inserts into the target tables. These optional annotations provide a super-set of the functionality offered by the XML Extender decomposition support. Refer to the “Resources” section for a more detailed documentation on supported annotations.

While the XSD language is an industry standard, the decomposition annotations are specific to IBM and DB2 for i. The result is that tools for generating these annotations on DB2 for i are difficult to find. Customers should consider that the annotations used by DB2 for i 7.1 are consistent with the annotations used by DB2 for Linux, UNIX, and Windows(LUW), and this allows annotated XML schemas that were built with tools designed for DB2 for LUW to also be used for IBM i. At the time of this writing, IBM Data Studio does not officially support XML decomposition on IBM i, but does support generating a subset of the decomposition annotations for use with DB2 for Linux, UNIX and Windows.

Registering XML schemas for decomposition

Registering an annotated schema with DB2 works almost exactly the same as registering a non-annotated schema. If the XSR object already exists, you must first remove it with the XSR_REMOVE procedure as shown in Listing 94.

```
CALL XSR_REMOVE('sql_schema', 'daily_summary_xsr')
```

Listing 94: XSR_REMOVE

The new annotated XSD file must be registered using the XSR_REGISTER stored procedure as shown in Listing 95. This procedure call works the same way as the previous XSR_REGISTER call in Listing 80.

```
CALL XSR_REGISTER('sql_schema', 'daily_summary_xsr',
                  'daily_summary.xsd',
                   GET_XML_FILE('/home/ntl/schemas/daily_summary_annotated.xsd'),
                   NULL)
```

Listing 95: XSR_REGISTER with annotations

The key difference is that when the XSR_COMPLETE procedure is invoked, the fourth parameter must indicate that decomposition annotations exist, by supplying a one instead of a zero value. This invocation type is shown in Listing 96.

```
CALL XSR_COMPLETE('sql_schema', 'daily_summary_xsr',NULL, 1)
```

Listing 96: XSR_COMPLETE with annotations for decomposition

The XML document shown in Listing 66 can now be decomposed using the annotated schema by using the XDBDECOMPXML stored procedure call shown in Listing 97.

```
CALL XDBDECOMPXML('sql_schema', 'daily_summary_xsr',
                  GET_XML_FILE('/home/ntl/daily_summary.xml'),
                  NULL)
```

Listing 97: XDBDECOMPXML procedure

The first two parameters on the XDBDECOMPXML procedure specify the XSR object, and the third supplies the XML document to be shredded. The third parameter expects the XML document to be passed as a BLOB value rather than an instance of the XML type.

The fourth parameter in this invocation example is NULL. This parameter can be used to supply a variable-length character string that can be referenced during the decomposition as the document ID. XML documents frequently do not contain an identifier that will be unique across all documents to be shredded. Therefore, when decomposing, it is often necessary to provide a unique identifier for the document which is being decomposed, so that the identifier can be used as a key value in the target tables.

After the XDBDECOMPXML procedure call successfully completes, the summary_orders and summary_customers tables contain the data shown in Table 10 and Table 11.

ORDER_ID	CUST_NAME	PART_NAME	QUANTITY	ORDER_TIMESTAMP
1	First Automobile Works	Valve	1000	2012-06-14 05:20:00.000000
2	Ultimate Automobile Works	Flywheel	2000	2012-06-14 10:20:00.000000

Table 10: Content of the summary_orders table

CUST_NAME	ADD_TIMESTAMP	CUST_PHONE	CUST_ADDRESS
First Automobile Works	2012-06-14 05:20:00.000000	86-10-12345678	Dongfeng St.
Ultimate Automobile Works	2012-06-15 05:30:00.000000	99-10-12345678	Main St.

Table 11: Content of the summary_customers table

Annotated decomposition with SQL dates and times

Although dates and times in an XML document should ideally be represented using the XML Data Model, the annotated decomposition feature can also be used to decompose XML documents where these values are stored as XML strings that contain an SQL data type in character format. This capability makes the decomposition process easier for XML documents that were originally designed to be decomposed by using the XML Extender support.

The following listing shows a sample document with an SQL timestamp value stored as an xs:string. Customers using XML Extender to decompose their XML documents had to store their timestamp values in this format.

```
<?xml version="1.0" encoding="UTF-8"?>
<SQL_TimeStamp>2012-01-01 00:00:00.000000</SQL_TimeStamp>
```

Listing 98: Sample XML document with SQL timestamps

Listing 99 includes the definition of the timestamp_decomp table which contains a timestamp column that can be used for storing the timestamp value embedded within in the SQL_TimeStamp element.

```
CREATE TABLE example.timestamp_decomp(SQL_TimeStampCol TIMESTAMP)
```

Listing 99: Decomposition target table with timestamp column

Listing 100 shows an annotated XSD that has been defined to decompose the XML document in Listing 98 into the timestamp_decomp table. In order for the XML document to be valid according to this XML schema, the type of the SQL_TimeStamp element must be defined with the xs:string because the data is not a valid xs:dateTime value. During the decomposition process, the XDBDECOMPXML procedure will first convert the xs:string value to an SQL character type and the SQL character string will eventually be cast to an SQL Timestamp value during the insert into the target table.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:sql="http://www.ibm.com/xmlns/prod/db2/xdb1"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SQL_TimeStamp"
               sql:column="SQL_TIMESTAMPCOL"
               sql:locationPath="/SQL_TimeStamp"
               sql:rowSet="EXAMPLE_AXSD_TIMESTAMP_DECOMP_0"
               type="xsd:string"/>
    <xsd:annotation>
    <xsd:appinfo xml:space="preserve">
      <sql:table>
        <sql:SQLSchema>EXAMPLE</sql:SQLSchema>
        <sql:name>TIMESTAMP_DECOMP</sql:name>
        <sql:rowSet>EXAMPLE_AXSD_TIMESTAMP_DECOMP_0</sql:rowSet>
      </sql:table>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:schema>
```

Listing 100: Annotated XSD with SQL timestamps

Annotated decomposition of values that have time zone components

Unlike the XMLTABLE function, the XDBDECOMPXML procedure cannot normalize an xs:date, xs:time, or xs:dateTime value to UTC when the value contains a time zone. The stored procedure can only be used to truncate the time zone. This capability allows the value to be shredded based on the value's local time zone.

An additional truncation annotation can be used in the XSD file to instruct the XDBDECOMPXML procedure to remove the time zone from the xs:date, xs:time or xs:dateTime value before converting the value to an SQL DATE, TIME, or TIMESTAMP value. Listing 101 shows an XML document that contains a timestamp with a time zone offset.

```
<?xml version="1.0" encoding="UTF-8"?>
<XML_TimeStamp>2012-01-01T00:00:00.000000+03:00</XML_TimeStamp>
```

Listing 101: XML document with xs:dateTime that has a time zone

The annotated XSD file shown in Listing 102 can be used to perform the decomposition of the XML document shown in Listing 101. A value of **1** for the truncate option indicates that the time zone will be removed from the xs:dateTime. If the truncate option is not provided, it defaults to a value of **0** which means that the XDBDECOMPXML procedure will fail with an error if a time zone component is part of the value.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:sql="http://www.ibm.com/xmlns/prod/db2/xdb1"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="XML_TimeStamp"
               sql:column="SQL_TIMESTAMPCOL"
               sql:locationPath="/XML_TimeStamp"
               sql:rowSet="EXAMPLE_AXSD_TIMESTAMP_DECOMP_0"
               sql:truncate="1"
               type="xsd:dateTime"/>
    <xsd:annotation>
    <xsd:appinfo xml:space="preserve">
      <sql:table>
        <sql:SQLSchema>EXAMPLE</sql:SQLSchema>
        <sql:name>TIMESTAMP_DECOMP</sql:name>
        <sql:rowSet>EXAMPLE_AXSD_TIMESTAMP_DECOMP_0</sql:rowSet>
      </sql:table>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:schema>
```

Listing 102: Annotated XSD with truncate option

Table 12 contains the result of the decomposition.

SQL_TIMESTAMPCOL
2012-01-01 00:00:00.000000

Table 12: Truncated timestamp

Full text search

Full text search is an important function in many XML environments. Assume that the ABC Corporation has created a products table with a product column that is used to store product descriptions using an XML format. A sample XML document with a description of the flywheel product is found in Listing 103.

```
<product>
  <name>flywheel</name>
  <description>mechanical device used for storing
                 rotational energy
  </description>
<date_added>2012-06-02</date_added>
</product>
```

Listing 103: Sample XML product description

The XML Extender product does not itself contain full text search capabilities for XML documents. However, these capabilities are available through the DB2 Text Extender options (5770-DE1 options 1 and 3). The DB2 Text Extender search support requires a model file to be set up in advance. The model file indicated which parts of the XML document should be indexed. A search can be performed on keywords within the indexed sections of the XML documents using the *Contains* user-defined function.

A sample query that uses the db2tx.contains function is provided in Listing 104. The query assumes that *handle* is a column handle defined for the product column, and that a model file has been created with a model called *mymodel* defined to include the */product/description* section of the XML document.

```
SELECT * FROM products
WHERE db2tx.Contains(handle,
                  'model mymodel sections (/product/description)  “energy”' ) = 1
```

Listing 104: Text Extender query search of XML document

Full text search for the integrated XML data type is supported by the OmniFind Text Search server for DB2 for IBM i (5733-OMF) product. OmniFind is a *no additional charge* licensed product. The OmniFind full text search support does not require determining in advance which specific sections of the XML document must be indexed and searched. The functions used for searching and ranking documents are built-in functions, allowing for better integration with the SQL language than the user-defined function provided by the Text Extender product. A link to a white paper that explains how to use OmniFind for XML searches is included in the “Resources” section of this paper.

A sample search using the built-in CONTAINS function and xmlxp search syntax is found in Listing 105. The query will search for products that were added after 1 June 2012 and contain variations on the phrase **stores energy** in the description.

When including xs:date and xs:dateTime values in a search, an important consideration is that OmniFind Text Search for DB2 for i stores only the local time in the index (in other words, the time zone component is truncated). Adding a time zone component to an xs:date or xs:dateTime that is used in the search criteria is not supported.

```
SELECT *
FROM products WHERE
 CONTAINS(product,
   ' @xmlxp:''/product[date_added > xs:date("2012-06-01")]/
                  description[. contains("stores energy")]’’') = 1
```

Listing 105: Full text search with date comparison

Recommendations:

The following section documents general recommendations and guidelines to consider when using the built-in XML type.

Comparing decomposition with XMLTABLE with annotated XML schema

There are two methods for performing XML decomposition using the integrated DB2 for i 7.1 XML support. One approach is to use the XMLTABLE function and the other is to use annotated XML schemas with the XDBDECOMPXML stored procedure. Deciding which approach to use is an important step in building your XML solution. The method of decomposition that was used with the XML Extender support is not usually a deciding factor in choosing which approach to use.

If no XML schema exists for the XML documents that need to be decomposed and creating a schema is impractical, then the annotated schema method cannot be performed. In this situation, the XMLTABLE function is the only option.

The XMLTABLE function also tends to be the preferred solution when the structure of the XML documents and decomposition requirements frequently change. The XMLTABLE function can dynamically decompose XML document without requiring pre-registration of a mapping XSD file. The XMLTABLE invocation can be easily changed whenever the decomposition mappings change. With annotated decomposition, it is necessary to define the mapping relationship of XML to DB2 tables in advance and maintain the annotations. If the mapping relationship needs to be updated, the registered XSR must be removed, updated, and registered again.

The XMLTABLE function is better suited for manipulations on xs:date, xs:time, and xs:dateTime values. The function can be used to return these values by normalizing them to UTC or truncating the time zone. The XDBDECOMPXML procedure can only return the values with the time zone truncated.

The XMLTABLE function does not require manipulation of an XML schema, but does require detailed knowledge of the XPath syntax. The XPath syntax might be easier for some developers to understand, but there also are very few tools capable of assisting a developer in building an SQL statement that includes an XMLTABLE reference for decomposition.

A major advantage of the annotated XML decomposition is that it supports shredding into more than one table. The XMLTABLE function requires a unique INSERT statement and function invocation for each target table.

Another consideration is the designer's familiarity with the XML technology and tools that are available. The annotated schema method, although similar to the mapping approach used by XML Extender, requires a detailed knowledge of the XML schema language and decomposition annotations. This can be a large learning curve, unless a proper XML toolset is available for creating and working with XML schemas and annotations.

Improving query performance using side tables

Users of XML Extender routinely use side tables for improving query performance. Side tables are additional tables that are created by the DB2 XML Extender product to improve performance of searches

over the elements and attributes in XML documents that were stored in a column. A DAD file defines how an element or attribute value is mapped to a specific column of a side table. As XML documents are inserted, data for specified elements and attributes are copied into the columns of the side table.

When the XML Extender support is used to enable a column for XML, the product created side tables with the columns corresponding to the elements and attributes defined in the DAD. The XML Extender support also ensures that the data in the side tables stays consistent with the data in the XML column. This is accomplished by creating triggers on the table to synchronize the XML data with side tables when insert, update, and delete operations are performed.

The relational data can then be used in a query, instead of processing the XML document, as long as the available relational data is sufficient to answer the query. This provides a way of optimizing location path expressions. In addition, users can build DB2 indexes over the columns in the side tables to further improve the performance of some queries.

The built-in XML type does not provide a concept of automatically creating and maintaining a side table for an XML column.

One suggestion is to manually create tables or materialized query tables (MQTs) to store a relational version of the XML data that is searched most frequently. These tables can be used to store decomposed XML values. Data within the tables can be maintained by creating triggers or by a REFRESH TABLE statement (if an MQT is used).

When a query involves XML data, the corresponding relational criterion can be added to the SQL query, improving performance. This process is essentially equivalent to the side tables that are available with XML Extender, and offers a manual way of using relational indexes to retrieve rows from an SQL table when the selection criteria involves data stored in XML documents.

An MQT is not automatically used by the optimizer in this scenario. The advantage of using an MQT as compared to a regular DB2 table is that the MQT has the SQL query associated with it that was used to initially populate its data. The contents of the table can be completely reloaded using the REFRESH TABLE statement. In addition, it is easier to recognize the dependency that the MQT has on the table that contains the XML document when exploring the database with tools.

The DB2 optimizer is fully aware of the relational aspects of an SQL query. When side tables are involved in a query, the optimizer considers indexes that are built over the side tables when building an access plan. One thing that the DB2 optimizer cannot do is understand the mapping of the data in the side tables to the XML data in the column. It is up to the application developer to reference data stored in side tables when creating a query. The developer must also determine which data is shredded into the side tables.

The first step is determining the elements and attributes that should be duplicated in a side table and how many side tables should be created. These considerations are similar to the recommendations used for the XML Extender product. However, because the XMLTABLE function supports more complex predicates and multiple result columns, additional possibilities exist.

An important step when designing the mapping between an XML document and a DB2 table is to determine whether or not an element or attribute can occur multiple times. If an element or attribute can occur multiple times then that value needs to be stored in its own side table. For example, if ABC Corporation wants to improve queries over the XML order documents (Listing 63) in the original_orders table (Listing 42).

More specifically, they would like to optimize the XPath expression, **/orders/order_infor/customer_name** by storing the value of the customer_name element in a relational format. The result of the expression will need its own side table because the relationship between the XML document and the customer_name element is one to many, meaning the customer_name element can occur multiple times in the document.

The XMLTABLE function offers a new and important trick for determining which side tables to create. The secret is to look at the row expressions on XMLTABLE invocations, and especially focus on the predicates in the row expressions.

For example, assume ABC Corporation often runs the queries similar to the one in Listing 106. These queries try to find XML documents in the original_orders table (Listing 42) to retrieve order information for orders placed after a certain date.

```
SELECT xt.*
FROM original_orders oo,
    XMLTABLE(XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
    '/orders/order_infor[xs:dateTime(order_datetime) >
                xs:dateTime("2012-06-14T12:10:00+03:00")]'
PASSING
  oo.order_doc
COLUMNS
  "customer_name"    VARCHAR(255)  PATH 'customer_name',
  "part_name"         VARCHAR(255) PATH 'part_name',
  "quantity"          BIGINT       PATH 'quantity'
) XT
```

Listing 106: XMLTABLE query with timestamp search

The most complete solution for optimizing this query is to decompose the contents of the customer_name, part_name, quantity, and order_datetime elements into a DB2 table. This is shown in Figure 7.

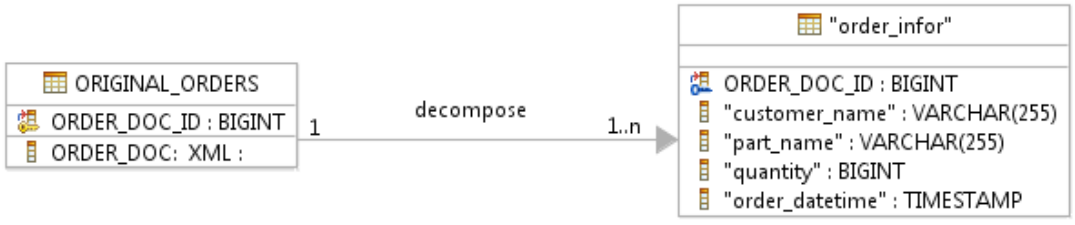

Figure 7: Side table for original orders

The one-to-many relationship between the XML document and the order_infor element necessitates a side table that contains one row for each order_infor element. As there can be only one customer_name, part_name, quantity, and order_datetime element contained within each order_infor element, additional side tables for each individual element will not be required in this example. The order_datetime element is included in the decomposition only to enable filtering on the order's timestamp.

Using the data in the side table, the query can be satisfied by using a traditional SQL query, as shown in Listing 107. As the order_datetime column has been normalized to UTC, the query must also use a timestamp value that has been normalized for UTC.

```
SELECT "customer_name", "part_name", "quantity"
FROM “order_infor” side_table
WHERE side_table."order_datetime" > TIMESTAMP('2012-06-14 09:10:00')
```

Listing 107: Query using only a side table

In real-world scenarios, it is not always practical to decompose all the necessary information from the XML document in advance. More realistic is to decompose the portions of the XML documents that are useful for the widest range of queries. The example that follows will decompose only the order_datetime elements from the order_doc XML document. A graphic representation of this decomposition is shown in Figure 8.

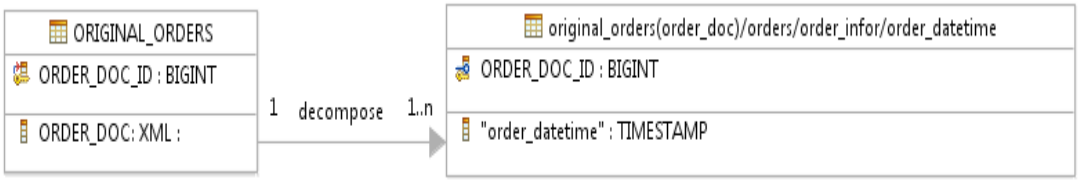

Figure 8: Diagram of a side table for order_datetime elements

Decomposing the order_datetime elements does not provide sufficient information to satisfy the sample query in Listing 106 using only relational tables, but it does make it possible to reduce the number of XML documents that must be analyzed as part of the query. Eliminating irrelevant documents using a relational query improves performance, because it is not necessary to process those XML documents that are not relevant to the query.

Creating a side table for the timestamp value of the order_datetime element involves creating an MQT and loading it with data. The reason for using an MQT for a side table as opposed to a regular table is more aesthetic than functional. The DB2 optimizer is not going to rewrite the SQL query to use the MQT; however, using an MQT creates a dependency in the database between the table with the XML column and the MQT. This dependency can make it easier to locate, manage, and use the side table in the future. Listing 108 shows the MQT definition.

```
CREATE TABLE
"original_orders(order_doc)/orders/order_infor/order_datetime" as (
 SELECT oo.order_doc_id, xt."order_datetime"
 FROM original_orders oo,
  XMLTABLE(XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
     '/orders/order_infor/order_datetime'
     PASSING oo.order_doc
     COLUMNS
       "order_datetime" TIMESTAMP path '.'
  ) XT
)
DATA INITIALLY IMMEDIATE
REFRESH DEFERRED
MAINTAINED BY USER
DISABLE QUERY OPTIMIZATION
```

Listing 108: Side table for order_datetime as an MQT

The materialized query table is defined so that it is immediately loaded with data when it is created. The path-based table name is used so that it is easier to see the XPath expression that is stored in the side table. The MQTs SELECT statement returns one row for each order_datetime element in the document. Each row has two columns, a document ID, and an SQL timestamp version of the xs:dateTime value stored in the order_datetime element. The document ID is used to link the rows in the MQT back to the XML document that generated them.

The next step is to create some triggers to keep the data in the table in synchronization with the XML documents. These triggers are not all that complicated. They insert, update, or delete the XML document's result rows from the associated XPath expression into the side table. You can find example trigger definitions in Listing 109.

```
CREATE TRIGGER “order_datetime->INS"
AFTER INSERT ON original_orders
REFERENCING NEW TABLE AS NEWT
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
INSERT INTO
"original_orders(order_doc)/orders/order_infor/order_datetime"
   (order_doc_id, "order_datetime")
SELECT nt.order_doc_id, xt."order_datetime"
 FROM newt nt,
  XMLTABLE(XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
     '/orders/order_infor/order_datetime'
     PASSING nt.order_doc
     COLUMNS "order_datetime" TIMESTAMP path '.') XT;
END;

CREATE TRIGGER "order_datetime->UPD"
AFTER UPDATE OF ORDER_DOC, ORDER_DOC_ID ON original_orders
REFERENCING NEW TABLE AS NEWT OLD TABLE AS oldt
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
DELETE FROM
 "original_orders(order_doc)/orders/order_infor/order_datetime"
  where order_doc_id in (select  oldt.order_doc_id from oldt);
INSERT INTO
 "original_orders(order_doc)/orders/order_infor/order_datetime"
(order_doc_id, "order_datetime")
SELECT nt.order_doc_id, xt."order_datetime"
 FROM newt nt,
  XMLTABLE(XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
     '/orders/order_infor/order_datetime'
     PASSING nt.order_doc
     COLUMNS "order_datetime" TIMESTAMP path '.'  ) XT;
END;

CREATE TRIGGER "order_datetime->DEL"
AFTER DELETE ON original_orders
REFERENCING OLD TABLE AS oldt FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
DELETE FROM
 "original_orders(order_doc)/orders/order_infor/order_datetime"
 WHERE order_doc_id IN (SELECT  oldt.order_doc_id FROM oldt);
END;
```

Listing 109: Same trigger definitions

When dealing with relational versions of XML data, the query must consider any conversions that occurred when converting XML data types to SQL data types. In this example, the TIMESTAMP value in the order_datetime column was converted to a timestamp in the UTC time zone by the XMLTABLE function. As a result, the SQL comparisons against this column must use timestamp values that correspond to a time zone of UTC rather than the xs:dateTime's local date and time that was used in Listing 106. This is not a huge consideration, but it is one more detail to keep in mind. The query in Listing 110 uses the side table to produce the same results as the query in Listing 106.

```
WITH eligible_rows AS (
SELECT * FROM   original_orders canidate_rows
   WHERE canidate_rows.order_doc_id IN
    (SELECT order_doc_id
     FROM "original_orders(order_doc)/orders/order_infor/order_datetime" AS rd
     WHERE rd."order_datetime" > '2012-06-14 09:10:00'
    ))
SELECT xt.*
FROM eligible_rows,
   XMLTABLE( XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
    '/orders/order_infor[xs:dateTime(order_datetime) >
                         xs:dateTime("2012-06-14T12:10:00+03:00")]'
PASSING   eligible_rows.order_doc
COLUMNS   "customer_name"    VARCHAR(255) PATH 'customer_name',
          "part_name"        VARCHAR(255) PATH 'part_name',
          "quantity"         BIGINT       PATH 'quantity') XT
```

Listing 110: SQL/XML query using a side table

The eligible_rows common table expression is evaluated first. The eligible_rows expression results in only the rows from original_orders where the XML document (order_doc) has at least one order_datetime element containing an xs:dateTime value that is greater than the value 2012-06-14T12:10:00+03:00. The rows that do not appear in the results of the eligible_rows expression have an XML value for order_doc that will result in an empty table being returned from the XMLTABLE function. It is therefore only necessary to provide the rows from the eligible_rows expression to the XMLTABLE function because it is known that the rows that have been removed will never appear in the result of a cross-join between the original_orders table and the XMLTABLE function.

The matching documents produced by the eligible_rows expression are cross-joined with the main query that involves the XMLTABLE function. For the common table expression to be useful, the number of rows produced by the eligible_rows expression must be much less than the number of rows that exist in the original_orders table. If this is not true, then the number of XML documents to be analyzed has not been reduced, and the performance improvements will not be achieved.

The query in Listing 110 returns the results displayed in Table 13.

customer_name	part_name	quantity
Ultimate Automobile Works	Flywheel	2000

Table 13: Query results

It is very important to understand that the eligible_rows common table expression returns all XML documents that will return at least one row when provided to the XMLTABLE function, thus preventing the XML documents that are irrelevant from being passed into an expensive table function. The table

expression does not filter the rows that result from the XMLTABLE function. This is the reason that the predicate needs to appear on the row expression for the XMLTABLE function. This idea is best illustrated by looking at the query in Listing 111. This query is exactly the same as the previous query in Listing 110, except that the row expression does not include the xs:dateTime comparison in a predicate.

```
WITH eligible_rows AS (
SELECT * FROM   original_orders canidate_rows
   WHERE canidate_rows.order_doc_id IN
    (SELECT order_doc_id
     FROM
      "original_orders(order_doc)/orders/order_infor/order_datetime"
      AS rd
     WHERE rd."order_datetime" > '2012-06-14 09:10:00'
    ))
SELECT xt.*
FROM eligible_rows,
   XMLTABLE(XMLNAMESPACES(DEFAULT 'http://www.abccompany.com'),
    '/orders/order_infor '
PASSING   eligible_rows.order_doc
COLUMNS  "customer_name"    VARCHAR(255) PATH 'customer_name',
         "part_name"        VARCHAR(255) PATH 'part_name',
         "quantity"         BIGINT       PATH 'quantity') XT
```

Listing 111: Query omitting the row expression predicate

The results are shown in the following table. An extra row appears in the results because the query only filtered based on XML documents and not the rows that result from the XMLTABLE function.

customer_name	part_name	quantity
First Automobile Works	Valve	1000
Ultimate Automobile Works	Flywheel	2000

Table 14: Result with row expression predicate omitted

The extra complexity of a query that uses side tables is worth the effort in some scenarios. Performance is significantly improved by using the common table expression because all of the documents that are known to result in an empty table are removed without the need for the XMLTABLE function to evaluate the XML document.

Summary

This paper has covered a wide range of XML-related functions. In earlier releases, DB2 for i could only interpret XML as a serialized character string. Any understanding or interpretation of the XML data itself was left up to licensed products and applications, such as DB2 XML Extender.

With the availability of DB2 for i 7.1, the integrated relational database has a deeper understanding of XML data in terms of the XML Data Model. The database now includes a built-in type capable of representing the values and relationships defined in the XML document. The data type is well-defined and does not permit corruption of the XML value by using non-XML operations. Additionally, built-in functions and procedures have been added for parsing, serialization, validation, decomposition, composition, query, transformation, and full text search.

This paper provided a general comparison between the built-in XML data type and the user-defined types previously offered by XML Extender. Examples were presented for using the built-in XML type to perform real-world XML related tasks that might have been previously handled with the XML Extender support. Also presented was the idea that the built-in XML support in DB2 for i 7.1 adds improvements for working with industry-standard XML data types, such as xs:dateTime, xs:date, and xs:time. Finally, the paper introduced some best practices for choosing a decomposition method, and also for improving query performance of queries over XML data by creating side tables.

Customers now have a strong motivation to move away from XML Extender in favor of the built-in XML type, built-in functions, and procedures. The SQL/XML functionality available for DB2 for i 7.1 is an improvement over non-integrated solutions because it:

- Simplifies publishing relational data as XML data
- Simplifies decomposing XML data into a relational model
- Simplifies the coding required to query XML data
- Encourages platform-independent applications
- Ensures the existence of greater technical support and educational resources

Resources

The following websites provide useful references to supplement the information contained in this paper:

- IBM Systems on PartnerWorld
  ibm.com/partnerworld/systems

- Virtual Loaner Program
  ibm.com/systems/vlp

- DB2 for i developerWorks forum
  http://www.ibm.com/developerworks/forums/forum.jspa?forumID=292

- DB2 for i Technology Updates wiki
  ibm.com/developerworks/ibmi/techupdates/db2

- DB2 for i SQL CLI Reference
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/topic/cli/rzadpkickoff.htm

- DB2 for i SQL Reference
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Fdb2%2Frbafzprintthis.htm

DB2 XML Extender resources:

- The Ins and Outs of XML and DB2 for i5/OS
  ibm.com/redbooks/abstracts/sg247258.html

- DB2 XML Extender Hints and Tips for the IBM eServer iSeries Server
  ibm.com/redbooks/abstracts/redp0135.html?Open

Integrated XML resources:

- Using RPG to exploit IBM DB2 XML support:
  ibm.com/developerworks/ibmi/library/i-using-rpg/index.html

- XML Meets DB2 for i - Getting Started With the XML Data Type Using DB2 for i
  http://www.ibmsystemsmag.com/ibmi/developer/general/xml_db2_part1/

- Using XML With DB2 for i - How to Use XML Data Type With DB2 for i
  http://www.ibmsystemsmag.com/ibmi/developer/general/xml_db2_part2/

- Integrating XML with DB2 for i 7.1
  http://www.iprodeveloper.com/article/databasesql/integrating-xml-with-db2-for-i-71-65081

- Introducing XML in SQL on DB2 for i
  http://www.mcpressonline.com/sql/now-introducing-xml-in-sql-on-db2-for-ibm-i.html

SQL XML documentation:

- IBM i SQL XML Programmers Guide
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzasp%2Frzaspkickoff.htm

- IBM i XML Data Model
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzasp%2Frzasp_datamodel.htm

- Differences in an XML document after storage and retrieval
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzasp%2Frbafyxml3852.htm

- Application programming language support for XML
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzasp%2Frbafyxml2265.htm

- XMLTABLE SQL Reference
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Fdb2%2Frbafzscaxmltable.htm

- XMLTABLE Tutorial
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzasp%2Frbafyxmltexample.htm

- Stylesheet transformation with XSLTRANSFORM
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Fdb2%2Frbafzscaxsltransform.htm

- XML Publishing Functions
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzasp%2Frbafyxml3909.htm

- XML Schema Validation
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Fdb2%2Frbafzscaxmlvalidate.htm

XML schema annotated decomposition documentation:

- Decomposition Overview
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzasp%2Frbafyxml2319.htm

- Annotations
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzasp%2Frbafyxml2725.htm

IBM OmniFind Text Search Server for DB2 for i resources:

- Rev Up XML Searches with IBM OmniFind
  http://www.iprodeveloper.com/article/databasesql/rev-up-xml-searches-with-ibm-omnifind-64984

- OmniFind white paper
  http://www.ibm.com/partnerworld/wps/servlet/ContentHandler/whitepaper/i/omnifind/search

About the authors

Nick Lawrence is an Advisory Software Engineer working on DB2 for i in IBM Rochester. He has been involved with DB2 for i since 1999. His most recent responsibilities have been in the area of full text search, SQL/XML, and XMLTABLE. You can reach Nick at ntl@us.ibm.com.

Yi Yuan is a Software Developer in DB2 team in China System and Technology Lab (CSTL). Yi has been working on XML new features of DB2 for i since 2009. Before that, Yi worked on development of OmniFind Text Search Server for DB2 for i. You can reach Yi at cdlyuany@cn.ibm.com.

Kent Milligan is a Senior Certified DB2 for i Consultant on the ISV Enablement team for IBM i in Rochester, Minnesota. After graduating from the University of Iowa, Kent spent the first eight years of his IBM career as a member of the AS/400 and DB2 development group in Rochester. He speaks and writes regularly about relational database topics and DB2 for i. You can reach Kent at kmill@us.ibm.com.

Trademarks and special notices

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.

Some information addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is

presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Photographs shown are of engineering prototypes. Changes may be incorporated in production models.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.