

Teradata Vantage™ - XML Data Type

Release 17.10

July 2021

Copyright and Trademarks

Copyright © 2013 - 2021 by Teradata. All Rights Reserved.

All copyrights and trademarks used in Teradata documentation are the property of their respective owners. For more information, see [Trademark Information](#).

Product Safety

Safety type	Description
	Indicates a situation which, if not avoided, could result in damage to property, such as to equipment or data, but not related to personal injury.
	Indicates a hazardous situation which, if not avoided, could result in minor or moderate personal injury.
	Indicates a hazardous situation which, if not avoided, could result in death or serious personal injury.

Third-Party Materials

Non-Teradata (i.e., third-party) sites, documents or communications ("Third-party Materials") may be accessed or accessible (e.g., linked or posted) in or in connection with a Teradata site, document or communication. Such Third-party Materials are provided for your convenience only and do not imply any endorsement of any third party by Teradata or any endorsement of Teradata by such third party. Teradata is not responsible for the accuracy of any content contained within such Third-party Materials, which are provided on an "AS IS" basis by Teradata. Such third party is solely and directly responsible for its sites, documents and communications and any harm they may cause you or others.

Warranty Disclaimer

Except as may be provided in a separate written agreement with Teradata or required by applicable law, the information available from the Teradata Documentation website or contained in Teradata information products is provided on an "as-is" basis, without warranty of any kind, either express or implied, including the implied warranties of merchantability, fitness for a particular purpose, or noninfringement.

The information available from the Teradata Documentation website or contained in Teradata information products may contain references or cross-references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation representative for those features, functions, products, or services available in your country.

The information available from the Teradata Documentation website or contained in Teradata information products may be changed or updated by Teradata at any time without notice. Teradata may also make changes in the products or services described in this information at any time without notice.

Machine-Assisted Translation

Certain materials on this website have been translated using machine-assisted translation software/tools. Machine-assisted translations of any materials into languages other than English are intended solely as a convenience to the non-English-reading users and are not legally binding. Anybody relying on such information does so at his or her own risk. No automated translation is perfect nor is it intended to replace human translators. Teradata does not make any promises, assurances, or guarantees as to the accuracy of the machine-assisted translations provided. Teradata accepts no responsibility and shall not be liable for any damage or issues that may result from using such translations. Users are reminded to use the English contents.

Feedback

To maintain the quality of our products and services, e-mail your comments on the accuracy, clarity, organization, and value of this document to: docs@teradata.com.

Any comments or materials (collectively referred to as "Feedback") sent to Teradata Corporation will be deemed nonconfidential. Without any payment or other obligation of any kind and without any restriction of any kind, Teradata and its affiliates are hereby free to (1) reproduce, distribute, provide access to, publish, transmit, publicly display, publicly perform, and create derivative works of, the Feedback, (2) use any ideas, concepts, know-how, and techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, and marketing products and services incorporating the Feedback, and (3) authorize others to do any or all of the above.

Contents

Chapter 1: Introduction to the Teradata XML Data Type	5
Changes and Additions	5
Teradata Support for XML	5
Standards Compatibility	5
Examples in this Document	6
Chapter 2: XML Data Type	9
XML Data Type Syntax	9
XML INLINE LENGTH Specification	10
New XML Type Instances	12
XML Type Usage	13
Restrictions for the XML Type	14
XML Type Transform	15
XML Type Ordering	16
XML Type Cast	17
External Representation	17
Migrating Data to the XML Type	17
Related Information	18
Chapter 3: XML Operations	19
Processing Large XML Documents	19
Creating and Altering Tables to Store XML Data	20
Storing XML Data	20
Retrieving XML Data	22
Creating a Join Index With an XML Type	23
XML Query Using XPath and XQuery	23
XSLT Transformation	24
XML Validation	24
Converting XML Data to Relational Data	25
XML Type Usage Examples	25
Chapter 4: Functions for XML Type and XQuery	34
CREATEXML	34
DataSize	36
XMLQUERY	38
XMLSERIALIZE	41
XMLTABLE	43
XMLDOCUMENT	47
XMLELEMENT	49

XMLFOREST	52
XMLCONCAT	55
XMLCOMMENT	56
XMLPI	57
XMLTEXT	58
XMLPARSE	59
XMLVALIDATE	62
XMLAGG	64
XMLSPLIT	66
Chapter 5: Methods on the XML Type	72
CREATESCHEMABASEDXML	72
CREATENONSCHEMABASEDXML	73
EXISTSNODE	74
ISCONTENT	75
ISDOCUMENT	76
ISSCHEMAVALID	76
ISSCHEMAVALIDATED	78
XMLEXTRACT	79
XSLTTRANSFORM	80
Chapter 6: Schema and Stylesheet Management	84
Schema Management	84
Resolving Schema Dependencies	84
Using Schemas	85
Stylesheet Management	86
Resolving Stylesheet Dependencies	86
Using Stylesheets	87
Related Information	88
Chapter 7: XML Shredding and Publishing	89
XML Shredding and Publishing	89
XML Shredding Based on a Schema	89
XML Shredding Based on a Stylesheet	105
XML Publishing	140
XMLPUBLISH	144
XMLPUBLISH_STREAM	146
Appendix A: Notation Conventions	149
Appendix B: External Representations for the XML Type	152
Appendix C: Encoding Names Supported by Teradata XML	163
Appendix D: Additional Information	164

Introduction to the Teradata XML Data Type

Teradata Vantage™ is our flagship analytic platform offering, which evolved from our industry-leading Teradata® Database. Until references in content are updated to reflect this change, the term Teradata Database is synonymous with Teradata Vantage.

Advanced SQL Engine is a core capability of Teradata Vantage, based on our best-in-class Teradata Database. Advanced SQL refers to the ability to run advanced analytic functions beyond that of standard SQL.

For information on data type mapping between Advanced SQL Engine and ML Engine, see *Teradata Vantage™ User Guide*, B700-4002.

Teradata Vantage™ - XML Data Type describes Teradata support for XML data, including the XML data type and the functions and methods available for processing, shredding, and publishing XML data.

Changes and Additions

Date	Description
July 2021	Minor edits.

Teradata Support for XML

Vantage provides the following support for storing and processing XML data:

- An XML data type that allows you to store XML content in a compact binary form that preserves the information set of the XML document
- Functions and methods on the XML type that support common XML operations like parsing, validation, transformation (XSLT) and Query (XPath and XQuery)
- The XQuery query language for querying and transforming XML content
- Stored procedures that allow you to publish the results of SQL queries in XML format
- Shredding functionality that allows you to extract values from XML documents and use them to update database tables

Standards Compatibility

Teradata XML conforms to the following XML standards.

Note:

In some cases, there are newer versions of the standards documents than the versions listed in the table; however, Teradata XML conforms only to the versions referenced by the links in the table.

Standard	Reference Documents	Compliance Notes
ANSI SQL/XML (SQL 2008)		
XML 1.1	https://www.w3.org/TR/2006/REC-xml11-20060816	
XSLT 1.0	https://www.w3.org/TR/1999/REC-xslt-19991116	
XPath 2.0	https://www.w3.org/TR/2007/REC-xpath20-20070123	
XQuery 1.0	https://www.w3.org/TR/2007/REC-xquery-20070123	<ul style="list-style-type: none"> Minimal conformance: https://www.w3.org/TR/2007/REC-xquery-20070123/#id-minimal-conformance
XML Schema 1.0	<ul style="list-style-type: none"> XML Schema Part 0: Primer https://www.w3.org/TR/xmlschema-0 XML Schema Part 1: Structures https://www.w3.org/TR/xmlschema-1 XML Schema Part 2: Datatypes https://www.w3.org/TR/xmlschema-2 	
XML Infoset	https://www.w3.org/TR/xml-infoset	
Fast Infoset	ITU-T Rec X.891 or ISO/IEC 24824-1 http://www.itu.int/ITU-T/asn1/xml/finf.htm	

Examples in this Document

The examples in this document reference the customer and customerText tables defined as follows:

```
CREATE TABLE customer (
  customerID INTEGER,
  customerName VARCHAR(256),
  customerXML XML )
PRIMARY INDEX (customerID);
```

```
CREATE TABLE customerText (
  customerID INTEGER,
  customerName VARCHAR(256),
```

```
customerXMLText CLOB )
PRIMARY INDEX (customerID);
```

The examples use a sample XML document in a file named Cust001.xml. The contents of this file are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Customer>
  <Name>John Hancock</Name>
  <Address>100 1st Street, San Francisco, CA 94118</Address>
  <Phone1>(858)555-1234</Phone1>
  <Phone2>(858)555-9876</Phone2>
  <Fax>(858)555-9999</Fax>
  <Email>John@somecompany.com</Email>
  <Order Number="NW-01-16366" Date="2012-02-28">
    <Contact>Mary Jane</Contact>
    <Phone>(987)654-3210</Phone>
    <ShipTo>Some company, 2467 Pioneer Road, San Francisco, CA - 94117</ShipTo>
    <SubTotal>434.99</SubTotal>
    <Tax>32.55</Tax>
    <Total>467.54</Total>
    <Item ID="001">
      <Quantity>10</Quantity>
      <PartNumber>F54709</PartNumber>
      <Description>Motorola S10-HD Bluetooth Stereo Headphones</Description>
      <UnitPrice>29.50</UnitPrice>
      <Price>295.00</Price>
    </Item>
    <Item ID="101">
      <Quantity>1</Quantity>
      <PartNumber>Z19743</PartNumber>
      <Description>Motorola Milestone XT800 Cell Phone</Description>
      <UnitPrice>139.99</UnitPrice>
      <Price>139.99</Price>
    </Item>
  </Order>
</Customer>
```

The sample XML document contains a Customers/Customer/Order/Item hierarchy. The examples use a schema file named customerschema.xsd that specifies the grammar for this XML document. The contents of customerschema.xsd are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Item">
```

```

<xs:complexType>
  <xs:sequence>
    <xs:element name="Quantity" type="xs:integer"/>
    <xs:element name="PartNumber" type="xs:string"/>
    <xs:element name="Description" type="xs:string"/>
    <xs:element name="UnitPrice" type="xs:float"/>
    <xs:element name="Price" type="xs:float"/>
  </xs:sequence>
  <xs:attribute name="ID" type="xs:string"/>
</xs:complexType>
</xs:element>

<xs:element name="Order">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Contact" type="xs:string"/>
      <xs:element name="Phone" type="xs:string"/>
      <xs:element name="ShipTo" type="xs:string"/>
      <xs:element name="SubTotal" type="xs:float"/>
      <xs:element name="Tax" type="xs:float"/>
      <xs:element name="Total" type="xs:float"/>
      <xs:element ref="Item" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="Number" type="xs:string"/>
    <xs:attribute name="Date" type="xs:date"/>
  </xs:complexType>
</xs:element>

<xs:element name="Customer">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Address" type="xs:string"/>
      <xs:element name="Phone1" type="xs:string"/>
      <xs:element name="Phone2" type="xs:string"/>
      <xs:element name="Fax" type="xs:string"/>
      <xs:element name="Email" type="xs:string"/>
      <xs:element ref="Order" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="ID" type="xs:string"/>
  </xs:complexType>
</xs:element>
</xs:schema>

```


XML Data Type

Vantage provides the XML data type for XML data. You can use it in the same way as other SQL data types supported by Teradata.

Using the XML type, you can store XML content in a compact binary form that preserves the information set of the XML document, including the hierarchy information and type information derived from XML validation. The document identity is preserved as opposed to XML shredding, which only extracts values out of the XML document.

The XML type stores XML values up to 2GB in size and supports XML processing functionality including the following:

- XSD validation
- XPath and XQuery-based queries
- XSLT transformations of XML content

While a common use of the XML type instance is to represent a single XML document, it can also represent a sequence of items where an item is an element, attribute or even an atomic value.

It can be used for XML values such as the following:

- Fragments of XML documents, such as nodes like elements, attributes, or text
- Values of XML schema predefined simple types, such as string or int

XML Data Type Syntax

```
{ XML | XMLTYPE }
  [ ( maxlength ) ]
  [ INLINE LENGTH integer ]
  [ attributes [...] ]
```

Syntax Elements

maxlength

A positive integer value followed by an optional multiplier.

The multiplier, if specified, is KkMmGg.

maxlength specifies the maximum length of the data type in bytes. You can define a maximum length on a per instance basis. When specified, the data type is used in a manner analogous to the VARBYTE or BLOB data types.

The length specified only covers the actual data length. The actual storage sizes include additional header information.

The value of *maxlength* cannot be less than 100 bytes or greater than 2097088000 bytes.

If *maxlength* is not specified, the default maximum length used is 2097088000 bytes.

INLINE LENGTH *integer*

A positive integer value which specifies the inline storage size. Data that is smaller than or equal to the inline storage size is stored inside the base row; otherwise, it is stored in a LOB subtable.

The inline length cannot be larger than *maxlength*.

attributes

The following data type attributes are supported for the XML type:

- NULL and NOT NULL
- TITLE
- NAMED
- DEFAULT NULL

Vantage does not support column storage or column constraint attributes for the XML type. For more information about the data type attributes, see *Teradata Vantage™ - Data Types and Literals*, B035-1143.

XML INLINE LENGTH Specification

You can use the optional INLINE LENGTH specification to specify the inline storage size. When the data is smaller than or equal to the inline storage size, it is stored inside the base row. Otherwise, the data is stored as a LOB (large object).

If the inline length is equal to the maximum length specified for the data type, the data type is treated as a non-LOB type. In this case, the performance may be better because there is no LOB overhead. You may see some performance improvement especially when the data type is used with UDFs.

You can choose a larger inline storage size when most of the table scans are on the XML column, therefore avoiding LOB table access which slows down performance.

Alternatively, you can choose a smaller inline storage size when the XML column is accessed only in a few of the table scans. This reduces the row size of the base table row which may improve table scan performance.

XML Minimum and Maximum Values for INLINE LENGTH

If the data type is not a LOB, then the minimum inline length must be equal to the maximum length specification for the data type.

If the data type is a LOB type (inline length is less than maximum length), the minimum inline length must be at least 100 bytes to accommodate the storage of the LOB OID (Object Identifier).

The following table shows the minimum INLINE LENGTH required for the XML data type.

Data Type	Non-LOB Type Minimum INLINE LENGTH	LOB Type Minimum INLINE LENGTH
XML	100 bytes	100 bytes

The maximum inline length that can be specified is 64000 bytes. In addition, the inline length cannot be larger than the maximum length specified for the XML type.

XML Default Values for INLINE LENGTH

If you do not specify an inline length and the maximum length of the data type is 64000 bytes (32000 UNICODE characters) or smaller, the default inline length is the same as the maximum length, and the data type is a non-LOB type.

If you do not specify an inline length and the maximum length of the data type is larger than 64000 bytes (32000 UNICODE characters), the data type is a LOB type with a default inline length as shown in the following table.

This table summarizes possible values for the inline length and the maximum length and whether the data type will be a LOB or non-LOB.

Data Type	Inline Storage	Maximum Length	LOB Type
XML	4046 bytes (default)	2097088000 bytes (default)	LOB
XML(<i>n</i>), where <i>n</i> ≤ 64000	<i>n</i> bytes (default)	<i>n</i> bytes	non-LOB
XML(<i>n</i>), where <i>n</i> > 64000	4046 bytes (default)	<i>n</i> bytes	LOB
XML(<i>n</i>) INLINE LENGTH <i>m</i>	<i>m</i> bytes	<i>n</i> bytes	<ul style="list-style-type: none"> If <i>n</i> = <i>m</i>, then non-LOB If <i>n</i> > <i>m</i>, then LOB If <i>n</i> < <i>m</i>, then error

Examples: Specifying the INLINE LENGTH for an XML Type

The following examples show XML type declarations with and without the INLINE LENGTH specification.

```
CREATE TABLE xmlTable1(id INTEGER,
/* non-LOB */      xml1 XML(64000));
```

```
CREATE TABLE xmlTable2(id INTEGER,
/* non-LOB */      xml1 XML(100),
/* LOB */          xml2 XML,
/* LOB */          xml3 XML INLINE LENGTH 30000);
```

The following is exactly the same as the example for xmlTable2, but with a different syntax.

```
CREATE TABLE xmlTable3(id INTEGER,
  /* non-LOB */      xml1 XML(100) INLINE LENGTH 100,
  /* LOB */          xml2 XML,
  /* LOB */          xml3 XML INLINE LENGTH 30000);
```

```
CREATE TABLE xmlTable4(id INTEGER,
  /* non-LOB */      xml1 XML(30000) INLINE LENGTH 30000,
  /* LOB */          xml2 XML INLINE LENGTH 100);
```

```
CREATE TABLE xmlTable5(id INTEGER,
  /* LOB */          xml1 XML(64000) INLINE LENGTH 100);
```

New XML Type Instances

You can construct XML type instances using the following:

- The NEW operator
- The CREATEXML function
- The XMLPARSE function

The following example uses the NEW operator to construct an XML type instance from the text representation of the XML document in the customerText.customerXMLText column.

If the XML is loaded in its text representation into a VARCHAR or CLOB column named customerXMLText in the customerText table, it can be used to construct XML type instances:

```
SELECT customerID, (NEW XML(customerXMLText)).XMLEXTRACT('/Customer/
Address', NULL)
FROM customerText;
```

Query result:

```
customerID  NEW XML(customerXMLText).XMLEXTRACT('/Customer/Address', Null)
-----
1  <Address>100 1st Street, San Francisco, CA 94118</Address>
```

The following example uses the CREATEXML function to construct an XML type instance from the text representation of the XML document in the customerText.customerXMLText column:

```
SELECT customerID, (CREATEXML(customerXMLText)).XMLEXTRACT('/Customer/
Address', NULL)
FROM customerText;
```

Query result:

```
customerID  CREATEXML(customerXMLText).XMLEXTRACT('/Customer/Address', Null)
-----
1  <Address>100 1st Street, San Francisco, CA 94118</Address>
```

The customerText.customerXMLText column can be of VARCHAR, CLOB, or BLOB type.

XML type instances are also generated by other operations such as casts and invocation of methods and functions that return results as XML type values.

XML Type Usage

You can use the XML type the same way as other SQL data types. For example, you can specify the XML type as follows:

- In table definitions
- As part of a structured UDT
- As parameters and return types for UDFs written in C, C++, or Java. This includes scalar and aggregate UDFs, table functions, and table operators.
- As IN, INOUT, and OUT parameters of stored procedures and external stored procedures written in C, C++, or Java.
- As parameters and return types for UDMs written in C or C++.

XML FNC functions and Java classes and methods are provided to enable a UDF, UDM, or external stored procedure to access and set the value of an XML parameter, or to get information about the XML type parameter. For information about these functions and methods, see *Teradata Vantage™ - SQL External Routine Programming*, B035-1147.

Example: Using the XML Type in a Table Definition

This statement creates a table with an XML type column, customerXML:

```
CREATE TABLE customer (
  customerID INTEGER,
  customerName VARCHAR(256),
  customerXML XML )
PRIMARY INDEX (customerID);
```

Example: XML Parameter in a Java UDF

```
REPLACE FUNCTION get_XML(X1 XML)
RETURNS INTEGER
LANGUAGE JAVA
```

```

NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.get_XML';

public static int get_XML(java.sql.SQLXML xml_var) throws SQLException

```

Alternatively, you can define the function as follows:

```

REPLACE FUNCTION get_XML(X1 XML)
RETURNS INTEGER
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.get_XML(java.sql.SQLXML)
returns int';

public static int get_XML(java.sql.SQLXML xml_var) throws SQLException

```

Example: XML Parameter in a Java External Stored Procedure

```

REPLACE PROCEDURE get_XML(IN X1 XML, INOUT A1 INTEGER)
LANGUAGE JAVA
NO SQL
PARAMETER STYLE JAVA
EXTERNAL NAME 'UDF_JAR:UserDefinedFunctions.get_XML';

public static void get_XML(java.sql.SQLXML X1, int[] A1) throws SQLException

```

Restrictions for the XML Type

- The XML type can accommodate values up to 2 GB in size. However, operations like XSLT and XQuery are only supported on documents that are smaller in size where the processing operation does not require more memory than specified by the XML_MemoryLimit DBS Control field.
- XML type columns cannot be used in the following situations:
 - Occur in queue tables
 - Be part of an index
 - Participate in joins

Note:

If the XML type is a non-LOB type, it can be part of a join index. However, it cannot be part of the primary index of the join index.

- You cannot use XML type columns in clauses that depend on ordering or comparison, such as ORDER BY, GROUP BY, or HAVING.
- You cannot use XML values in arithmetic expressions. XML values can be of a type, such as xs:int, that can be used in arithmetic computations. In this case, you can cast the XML value to the appropriate SQL type to perform the computation.
- XML type values are not comparable and should not be used in relational comparison operations (for example >, <, =). XML values can be explicitly cast to other scalar SQL types, and values of those types may be comparable. For example, if the value is known to be of XML type xs:int, it can be cast to SQL integer data type and then the integers can be compared.

Note:

Because XML values are not comparable, they are not included in checks for row duplication (for example, during an insert into a set table). This behavior is similar to CLOBs/BLOBs which also do not participate in row duplication checks.

- Although the external representation of XML values is of the character string type, string operations are not allowed directly on the XML values. XML can be serialized or cast to generate a string representation before you apply the string operation.

XML Type Transform

The XML type in Field, Record, and Indicator modes uses transforms. The XML type has the following predefined transform groups to convert objects to CLOB, BLOB, VARCHAR, and VARBYTE.

Transform Group	Complex Data Type	Primary Type	Default	Format
TD_XML_CLOB	XML	CLOB	Yes	Text format in CHARACTER SET UNICODE
TD_XML_BLOB	XML	BLOB	No	Text format in UTF-8
TD_XML_VARCHAR	XML	VARCHAR(32000)	No	Text format in CHARACTER SET UNICODE
TD_XML_VARBYTE	XML	VARBYTE(64000)	No	Text format in UTF-8

You can use the TRANSFORM option in the CREATE PROFILE/MODIFY PROFILE or CREATE USER/MODIFY USER statements to specify for a user the particular transform group that will be used for a given data type.

Use the SET TRANSFORM GROUP FOR TYPE statement to change the active transform group in the current session. You can use this statement multiple times for a data type to switch from one transform group to another within the session. If the logon user already has transform settings, the statement modifies the transform settings for the current session.

Note:

You cannot use CREATE TRANSFORM or REPLACE TRANSFORM to create new transforms for complex data types (CDTs). You can only create new transforms for structured and distinct user-defined types (UDTs).

Transform Group Macros

You can use the following macros to find the transform group for a UDT (or CDT), or the transform group settings for a user, profile, or current session.

Macro	Description
SYSUDTLIB.HelpCurrentUserTransforms	Lists the transform group settings of the current logon user.
SYSUDTLIB.HelpCurrentSessionTransforms	Lists the transform group settings of the current session.
SYSUDTLIB.HelpUserTransforms(<i>User</i>)	Lists the transform group settings for a specific user.
SYSUDTLIB.HelpCurrentUDTTransform(<i>UDT</i>)	Lists the transform group settings of the current session for the specified UDT.
SYSUDTLIB.HelpUDTTransform(<i>User</i> , <i>UDT</i>)	Lists the transform group for a UDT for a user.
SYSUDTLIB.HelpProfileTransforms(<i>Profile</i>)	Lists the transform group settings for a specific profile.
SYSUDTLIB.HelpProfileTransform(<i>Profile</i> , <i>UDT</i>)	Lists the transform group for a UDT for a profile.

For more information about these macros, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Related Information

- [XSLT Transformation](#)
- In *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144:
 - CREATE PROFILE
 - MODIFY PROFILE
 - CREATE USER
 - MODIFY USER
 - SET TRANSFORM GROUP FOR TYPE

XML Type Ordering

XML values are not comparable. Referencing an XML type column in any clause that depends on comparing values such as ORDER BY, GROUP BY, or DISTINCT results in an error.

XML Type Cast

When casting to XML type from other SQL types, an XML value of the nearest xml schema primitive type is created. For example, xs:string for VARCHAR/CLOB and xs:date for DATE.

When casting from XML type to other SQL types, the string value of the XML instance should be compatible with the target data type. For example, xs:date value being cast to SQL DATE type.

Casts to and from XML type are supported for the following data types.

<ul style="list-style-type: none"> • VARCHAR • CLOB • VARBYTE • BLOB • BYTEINT • SMALLINT • INTEGER • BIGINT • DECIMAL • FLOAT • NUMBER • DATE • TIME • TIME WITH TIMEZONE • TIMESTAMP • TIMESTAMP WITH TIMEZONE 	<ul style="list-style-type: none"> • INTERVAL YEAR • INTERVAL YEAR TO MONTH • INTERVAL MONTH • INTERVAL DAY • INTERVAL DAY TO HOUR • INTERVAL DAY TO MINUTE • INTERVAL DAY TO SECOND • INTERVAL HOUR • INTERVAL HOUR TO MINUTE • INTERVAL HOUR TO SECOND • INTERVAL MINUTE • INTERVAL MINUTE TO SECOND • INTERVAL SECOND
--	---

Note:

Casting to XML type from large object types (CLOB and BLOB) is subject to a limitation that only values up to 64KB in size can be cast successfully.

When casting from XML type to VARCHAR or CLOB, only the UNICODE character set is supported for the target data type.

When casting to XML type, you can specify the inline length so that the resulting XML data type can be a LOB type or a non-LOB type.

External Representation

For information about the external representations for the XML type, see [External Representations for the XML Type](#).

Migrating Data to the XML Type

To migrate XML data stored in VARCHAR or CLOB columns to equivalent schemas with XML type columns, do the following:

1. Verify the XML data is well-formed and conforms to rules of XML formatting.
2. Create new versions of the tables using XML type for columns that will hold the XML data.
3. Insert the XML text into the XML columns using the NEW XML operator or the CREATEXML function.

Related Information

- Data type attributes in *Teradata Vantage™ - Data Types and Literals*, B035-1143
- The NEW operator in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210
- The CREATEXML function: [CREATEXML](#)
- The XMLPARSE function: [XMLPARSE](#)
- The XMLEXTRACT method: [XMLEXTRACT](#)
- [Processing Large XML Documents](#)
- [Storing XML Data](#)
- Examples of XML type usage in [XML Type Usage Examples](#)

XML Operations

Processing Large XML Documents

The XML type can accommodate values up to 2 GB in size; however, operations like XSLT and XQuery are only supported on documents that are smaller in size. The methods that can be evaluated in one pass are supported on documents of all sizes. These methods allow for a streaming implementation in which the XML document is traversed in one direction without requiring significant memory resources for holding any state information. This includes methods for parsing and validation, and a constrained implementation of XMLEXTRACT which evaluates the query on subtrees of the large XML document tree, allowing for a streamed processing of the large document.

Operations such as XQuery and XSLT cannot be evaluated in a streamed manner. The documents on which they operate must be loaded into memory. For large XML documents, this can consume a significant amount of memory resources. For example, XSLT needs to load both the document and the stylesheet into memory and requires about 10 times as much memory as the document size. Similarly, the XQuery data model instance representing an XML document can occupy as much memory as 20 to 30 times the size of the document on disk. Therefore, these operations can be supported only on smaller documents that can fit within the memory constraints.

You can specify the maximum amount of memory allowed for such operations using the XML_MemoryLimit DBS Control field setting. An error results if the XML processing operation requires more memory than allowed by this field.

You can also use the XML_MemoryLimit setting in conjunction with Teradata Active System Management (ASM) to limit the number of concurrent invocations of large memory XML functions and methods. Using the Workload Designer portlet in Teradata Viewpoint, you can place filters, throttles, and classification criteria on XML functions and methods. This will control the amount of memory available to XML operations and prevent runaway queries from using up system resources and degrading performance.

Recommendation: If you expect document sizes to be large in relation to the XML_MemoryLimit setting, you should consider XML shredding as an alternate method for storing the XML content to query the data efficiently. Consider XML shredding if query performance is important, because performance for queries on larger documents is generally worse than for smaller documents.

Related Information

For details about ...	See...
the XML_MemoryLimit field and DBS Control	<i>Teradata Vantage™ - Database Utilities</i> , B035-1102.
the XML shredding process	XML Shredding Based on a Schema.

For details about ...	See...
Teradata ASM and workload management	<ul style="list-style-type: none"> • <i>Teradata Vantage™ - Application Programming Reference</i>, B035-1090. • <i>Teradata Vantage™ - Database Administration</i>, B035-1093.
using Viewpoint	<i>Teradata® Viewpoint User Guide</i> , B035-2206

Creating and Altering Tables to Store XML Data

You can create tables containing XML type columns or alter a table to add, drop, or rename XML type columns:

- You can specify the same CREATE TABLE or ALTER TABLE options that are permitted on the UDT types on the XML type.
- You can use the CREATE TABLE statement to create a table that contains one or more XML type columns.

Note:

You cannot use an XML type column in an index definition.

- You can use the ALTER TABLE statement to add, drop, or rename an XML type column.
- You can use ALTER TABLE to change the maximum length and inline storage length of an XML column subject to the following restrictions:
 - If the existing XML column type is a LOB type, you can only change the maximum length to a larger value.
 - If the existing XML column type is a non-LOB type, the newly changed data type must remain a non-LOB type, and the new maximum length and inline length values must be greater than the old values.

For more information about CREATE TABLE and ALTER TABLE, see *Teradata Vantage™ - SQL Data Definition Language Syntax and Examples*, B035-1144.

Storing XML Data

Vantage provides the following methods for storing XML data:

- Store XML data in tables containing columns defined with the XML data type
- Use XML shredding to extract values from XML documents and use the values to update tables in the database

This table compares the two methods for storing XML data.

XML Type	XML Shredding
<ul style="list-style-type: none"> • No requirement to map between hierarchical and relational models prior to storing the XML contents. • Preserves the information set of the XML document, including the hierarchy information and type information derived from XML validation. • Stores XML content in a compact binary form. 	<ul style="list-style-type: none"> • Requires you to map the XML document structure to the target tables prior to storing the XML contents. • Does not preserve the document identity. • Provides better query performance for large XML documents.

Procedure

These are the typical tasks you perform to store the text representation of XML documents in an XML column in a table:

1. Create a table defined with XML columns that will contain your XML documents.
2. Determine an appropriate method for loading the documents into the table.
Note that some load utilities, such as FastLoad and MultiLoad, do not support LOBs or XML. However, you can use these utilities to load XML data in the following cases:

- You can load XML data if it is less than 64 KB, and the target table defines the column as CHAR or VARCHAR.
- If you use a transform group that converts XML to/from VARCHAR or VARBYTE.

When loading XML data using FastLoad or MultiLoad using VARCHAR or VARBYTE transforms, the imported data must fit in the row. If it cannot be stored inline, the input row is put into the error table.

For documents larger than 64KB, use a utility that has LOB support:

- The MLOADX protocol can load LOB XML data using any transforms without the restriction of the inline length specified for the type.
- Teradata Parallel Transporter fully supports loading large XML documents.
- BTEQ only supports loading large XML documents in ASCII session character set which is not the default character set for XML.

3. Insert the XML text into the XML columns using the NEW XML operator or the CREATEXML function.

When loading your XML documents:

- Consider the encoding of the XML document if the values will be translated to the session character set. For example, if you load the document as VARCHAR, you can get errors if the document contains characters that cannot be represented in the session character set.
- Consider shredding parts of the XML document to columns of other SQL types that can be indexed which allows for better performance when retrieving the XML values.

Related Information

- For information about the XML shredding process, see [XML Shredding Based on a Schema](#).

- For information about transform groups for the XML type, see [XML Type Transform](#).
- For an example of using the NEW XML operator, see [Examples: Using CREATEXML to Create XML Type Instances](#).
- For information about the CREATEXML function, see [CREATEXML](#).
- For examples of loading XML documents, see:
 - [Example: Loading an XML Document into a Table](#)
 - [Example: Loading Large XML Documents](#)

Retrieving XML Data

You can retrieve XML data as an entire document or as query results.

XML values returned from the database might not be well-formed XML documents as defined by the XML 1.0 and XML 1.1 specifications. For example, XML queries can return sequences or atomic values that might not have a well-formed XML representation. Note that parsing errors can occur in these cases. You can either include these results into other XML documents through entity references or using string concatenation. You can also modify queries so that well-formed XML is returned.

XML values are always returned in non-Field Mode.

XML Value Encoding and the Encoding Declaration

This table shows what the encoding and encoding declaration will be based on the return data type of the XML value.

Return Type of XML Value	Encoding	Encoding Declaration
XML type (in record/indicator mode)	UTF-8	<ul style="list-style-type: none"> • If a document is retrieved in its entirety, and if it originally had an encoding declaration, the encoding declaration is preserved. • If there was no original encoding declaration, none is added (no encoding declaration is interpreted as UTF-8 by the parser according to the standard).
CLOB or VARCHAR (in field mode, or when XMLSERIALIZE is called with CLOB or VARCHAR target type)	The encoding of the document received will match the session character set in use.	<ul style="list-style-type: none"> • If a document is retrieved in its entirety, and if it originally had an encoding declaration, the encoding declaration is preserved. • If there was no original encoding declaration, none is added.
BLOB or VARBYTE (when XMLSERIALIZE is called with BLOB or VARBYTE target type)	The encoding of the XML value will be as specified by the user via the ENCODING clause in the XMLSERIALIZE	The encoding declaration is as specified by user via the ENCODING clause or none.

Return Type of XML Value	Encoding	Encoding Declaration
	function call. It is UTF-8 if an encoding is not specified.	

If the XML values are returned as XML, CLOB, or VARCHAR values, there is potential for mismatch between the actual encoding of the XML value and its encoding declaration. You can instruct the parser via parser-provided APIs to override the encoding declaration in the XML document based on external knowledge, such as knowing the session character set in use.

Document Size Limitation and Encoding

There is a 2 GB limit on the size of XML documents. If you request a document in an encoding that results in a document larger than 2 GB, an error is raised. For example, you may have a large document encoded in ASCII which is 1.5 GB and fits under the 2 GB limit. However, if you request a serialization with UTF-16 encoding, the size of such an encoding would be larger than the 2GB limit because UTF-16 is a double-byte encoding, and this would result in an error.

Creating a Join Index With an XML Type

When the inline length is equal to the maximum length, the XML type is treated as a non-LOB type. In this case, the non-LOB XML type can be part of a join index. However, it cannot be part of the primary index of the join index.

XML Query Using XPath and XQuery

The basic steps for querying XML content are:

1. Compose an XPath or XQuery query and use BTEQ to test it against XML documents in the database.
2. Depending on the intended use, you can use these methods and functions to evaluate the XML queries:
 - XMLEXTRACT method
 - EXISTSNODE method
 - XMLQUERY function
 - XMLTABLE function

Improving XML Query Performance

XML type columns cannot be part of an index. However, if the XML type is a non-LOB type, it can be part of a join index although it cannot be part of the primary index of the join index.

In addition, you can use the XMLEXTRACT method to extract paths or values of interest in the XML document structure to shred portions of the document into columns of other SQL data types that can be indexed. You can design your tables and queries to leverage this capability to restrict XML queries to rows

of interest, thereby improving performance. You can also use the AS_SHRED_BATCH stored procedure to shred XML documents.

Related Information

- The XMLEXTRACT method: [XMLEXTRACT](#)
- The EXISTSNODE method: [EXISTSNODE](#)
- The XMLQUERY function: [XMLQUERY](#)
- The XMLTABLE function: [XMLTABLE](#)
- For examples of XML query usage, see the following:
 - [Example: XPath Query](#)
 - [Example: XQuery Query](#)
- The AS_SHRED_BATCH stored procedure in [XML Shredding Based on a Schema](#)

XSLT Transformation

To transform an XML value using a stylesheet:

1. Design the stylesheet and test it using tools such as Stylus Studio, XML Spy, or Oxygen.
2. If the stylesheet contains includes, consolidate it using the Schema and Stylesheet Consolidation utility.
3. Insert the stylesheet into a database table.
4. Use the XSLTTRANSFORM method to transform the XML value. Join with the table containing the stylesheet to provide the stylesheet to the method.

You can download sample stylesheets and the Schema and Stylesheet Consolidation utility from [Teradata Downloads](#).

Related Information

- For information about consolidating stylesheets, see [Resolving Stylesheet Dependencies](#).
- The XSLTTRANSFORM method: [XSLTTRANSFORM](#)
- [Using Stylesheets](#)

For examples of transforming XML values, see the following:

- [Example: XSL Transform on XML Values](#)
- [Example: Using Stylesheets in XSLT Transformations](#)

XML Validation

To validate an XML document:

1. Design the schema and test it using tools such as Stylus Studio, XML Spy, or Oxygen. Sometimes schemas are also available from external sources, such as standards bodies like FpML or ACORD.

2. If the schema document contains includes or imports, consolidate it using the Schema and Stylesheet Consolidation utility.
3. Insert the schema into a database table.
4. Use methods on the XML type like `CREATESCHEMABASEDXML` or `ISSCHEMAVALID` to validate an XML value. Join with the table containing the schema to provide the schema to the method.

In addition, you can download sample schemas and the Schema and Stylesheet Consolidation utility from [Teradata Downloads](#).

Related Information

- For information about consolidating schemas, see [Resolving Schema Dependencies](#).
- [Using Schemas](#)
- The `CREATESCHEMABASEDXML` method: [CREATESCHEMABASEDXML](#)
- The `ISSCHEMAVALID` method: [ISSCHEMAVALID](#)

For examples of XML document validation, see the following:

- [Example: Storing a Schema in a Table](#)
- [Example: Validating an XML Document](#)

Converting XML Data to Relational Data

You can use the `XMLTABLE` function to convert an XML tree structure into zero or more rows:

1. Compose an XQuery query that will generate a sequence. Each item of this sequence will generate a corresponding row.
2. Determine the row format (column name, type of all the columns) and determine the path expression that will return the value of the column. The path expression should be relative to the items returned by the query in step (1).
3. Compose a query which calls the `XMLTABLE` function to turn the XML documents into rows.

Related Information

- The `XMLTABLE` function: [XMLTABLE](#)
- [Example: Converting XML Documents to Rows and Columns](#)

XML Type Usage Examples

The following sections use a running example to illustrate the use of the XML type. These examples reference the following:

- customer and customerText tables
- A sample XML document in a file named Cust001.xml

- A sample schema file named customerschema.xsd that specifies the grammar for the sample XML document

See [Examples in this Document](#) for descriptions of these tables and files.

Example: Loading an XML Document into a Table

These INSERT statements load an XML document into the customer and customerText tables:

```
INSERT INTO customer (1, 'John Hancock', CREATEXML('<?xml version="1.0"
encoding="UTF-8"?>
<Customer ID="C00-10101">
  <Name>John Hancock</Name>
  <Address>100 1st Street, San Francisco, CA 94118</Address>
  <Phone1>(858)555-1234</Phone1>
  <Phone2>(858)555-9876</Phone2>
  <Fax>(858)555-9999</Fax>
  <Email>John@somecompany.com</Email>
  <Order Number="NW-01-16366" Date="2012-02-28">
    <Contact>Mary Jane</Contact>
    <Phone>(987)654-3210</Phone>
    <ShipTo>Some company, 2467 Pioneer Road, San Francisco, CA - 94117</ShipTo>
    <SubTotal>434.99</SubTotal>
    <Tax>32.55</Tax>
    <Total>467.54</Total>
    <Item ID="001">
      <Quantity>10</Quantity>
      <PartNumber>F54709</PartNumber>
      <Description>Motorola S10-HD Bluetooth Stereo Headphones</Description>
      <UnitPrice>29.50</UnitPrice>
      <Price>295.00</Price>
    </Item>
    <Item ID="101">
      <Quantity>1</Quantity>
      <PartNumber>Z19743</PartNumber>
      <Description>Motorola Milestone XT800 Cell Phone</Description>
      <UnitPrice>139.99</UnitPrice>
      <Price>139.99</Price>
    </Item>
  </Order>
</Customer>')));
INSERT INTO customerText (1, 'John Hancock', '<?xml version="1.0"
encoding="UTF-8"?>
<Customer ID="C00-10101">
```

```

<Name>John Hancock</Name>
<Address>100 1st Street, San Francisco, CA 94118</Address>
<Phone1>(858)555-1234</Phone1>
<Phone2>(858)555-9876</Phone2>
<Fax>(858)555-9999</Fax>
<Email>John@somecompany.com</Email>
<Order Number="NW-01-16366" Date="2012-02-28">
  <Contact>Mary Jane</Contact>
  <Phone>(987)654-3210</Phone>
  <ShipTo>Some company, 2467 Pioneer Road, San Francisco, CA - 94117</ShipTo>
  <SubTotal>434.99</SubTotal>
  <Tax>32.55</Tax>
  <Total>467.54</Total>
  <Item ID="001">
    <Quantity>10</Quantity>
    <PartNumber>F54709</PartNumber>
    <Description>Motorola S10-HD Bluetooth Stereo Headphones</Description>
    <UnitPrice>29.50</UnitPrice>
    <Price>295.00</Price>
  </Item>
  <Item ID="101">
    <Quantity>1</Quantity>
    <PartNumber>Z19743</PartNumber>
    <Description>Motorola Milestone XT800 Cell Phone</Description>
    <UnitPrice>139.99</UnitPrice>
    <Price>139.99</Price>
  </Item>
</Order>
</Customer>');

```

Example: Loading Large XML Documents

The following shows how you can use the LOB support in BTEQ to load large documents.

In this example, you have a file called custdocs.txt with the following content:

```
Cust001.xml|1|John Hancock
```

Cust001.xml is an XML document on the client file system. You can use this .IMPORT statement to load the Cust001.xml document into the customer table:

```
.IMPORT VARTEXT '|' LOBCOLS=1 FILE='custdocs.txt'
USING (custdoc XML AS DEFERRED, custid VARCHAR(256), custname VARCHAR(256))
INSERT INTO customer(CAST(:custid AS INTEGER), :custname, :custdoc);
```

Example: Storing a Schema in a Table

After defining a schema, you can store it in a table before using it for validating XML values. In this example, the schema document is stored in the table schematab, which is defined as the following:

```
CREATE TABLE schematab (
  schemaid VARCHAR(32),
  schemaContent XML )
PRIMARY INDEX (schemaid);
```

The schema.txt file used in the .IMPORT statement below has the following format:

```
customerschema.xsd|customerschema.xsd
```

Use this statement to store the schema in the schematab table:

```
.IMPORT VARTEXT '|' LOBCOLS=1 FILE=schema.txt
USING (fxsd XML AS DEFERRED, xmlid varchar(32))
INSERT into schematab values(:xmlid, :fxsd);
```

Example: Validating an XML Document

In this query, the ISSCHEMAVALID method is used to validate the XML document in the customer table using the schema stored in the schematab table.

```
SELECT customerID, customerXML.ISSCHEMAVALID(schematab.schemaContent,
'Customer', '')
FROM customer, schematab
WHERE schematab.schemaid = 'customerschema.xsd';
```

The result of the query is:

customerID	customerXML.ISSCHEMAVALID(schemaContent, 'Customer', '')
1	1

The ISSCHEMAVALID method returned a value of 1 indicating that the XML document is valid based on the schema.

Example: XPath Query

You can use methods like EXISTSNode on the XML type to check for existence of nodes matching an XPath expression:

```
SELECT customerID
FROM customer
WHERE customerxml.EXISTSNode('/Customer/Name', '') = 1;
```

The query result is:

```
customerID
-----
1
```

Example: XQuery Query

You can use the XMLQUERY function to evaluate XQuery 1.0 queries against XML values. This example shows a query that is evaluated with the XML value as the context node:

```
SELECT XMLQUERY('for $i in /Customer/Order/Item
return $i/PartNumber' PASSING customerXML RETURNING CONTENT) as queryres
FROM customer WHERE customerID=1;
```

The result from the query is:

```
queryres
=====
<PartNumber>F54709</PartNumber><PartNumber>Z19743</PartNumber>
```

Example: XSL Transform on XML Values

This example transforms an XML value by calling its XSL transform method with the stylesheet passed as parameter by value:

```
SELECT customerXML.XSLTTRANSFORM(new XML('<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/Customer">
    <html>
    <title>Items Ordered</title>
    <body>
    <table border="1">
```

```

    <tr>
      <th>Item ID</th>
      <th>Description</th>
      <th>Part Number</th>
      <th>Qty</th>
      <th>Unit Price</th>
      <th>Total Price</th>
    </tr>
    <xsl:for-each select="Order/Item">
      <tr>
        <td><xsl:value-of select="@ID"/></td>
        <td><xsl:value-of select="Description"/></td>
        <td><xsl:value-of select="PartNumber"/></td>
        <td><xsl:value-of select="Quantity"/></td>
        <td><xsl:value-of select="UnitPrice"/></td>
        <td><xsl:value-of select="Price"/></td>
      </tr>
    </xsl:for-each>
  </table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>'), '') FROM customer WHERE customerID=1;

```

The result is an XML document, which is the result of transforming the source XML using the stylesheet.

Here are partial results from the query. The ellipsis (...) is not part of the query results. It indicates that the query returns additional results that are truncated in the example.

```

customerXML.XSLTTRANSFORM( NEW XML('<?xml version="1.0"?> <x ...
-----
<html><title>Items Ordered</title><body><table border="1"><tr><th>Item
ID</th><th>Description</th><th>Part Number</th><th>Qty</th><th>Unit Price</
th><th>Total Price</th></tr><tr><td>001</td><td>Motoro ...

```

Example: Using Stylesheets in XSLT Transformations

You can store stylesheets in tables and use them with functions and methods that perform XSLT transformations through joins. In this example, the xslt.txt file contains information for importing the stylesheet (the stylesheet file name and ID). The stylesheet to be stored in the styletab table is written in the Itemlist.xslt file.

```

CREATE TABLE styletab(
  stylesheetid INTEGER,

```

```
stylesheetcontent XML )
PRIMARY INDEX (stylesheetid);
```

The xslt.txt file contains:

```
Itemlist.xslt|1
```

The Itemlist.xslt file contains:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/Customer">
    <html>
      <title>Items Ordered</title>
      <body>
        <table border="1">
          <tr>
            <th>Item ID</th>
            <th>Description</th>
            <th>Part Number</th>
            <th>Qty</th>
            <th>Unit Price</th>
            <th>Total Price</th>
          </tr>
          <xsl:for-each select="Order/Item">
            <tr>
              <td><xsl:value-of select="@ID"/></td>
              <td><xsl:value-of select="Description"/></td>
              <td><xsl:value-of select="PartNumber"/></td>
              <td><xsl:value-of select="Quantity"/></td>
              <td><xsl:value-of select="UnitPrice"/></td>
              <td><xsl:value-of select="Price"/></td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Use this statement to store the stylesheet in the styletab table:

```
.IMPORT VARTEXT '|' LOBCOLS=1 FILE='xslt.txt'
USING (fxslt CLOB AS DEFERRED, schemaid VARCHAR(32))
INSERT INTO styletab values(:schemaid, new XML(:fxslt));
```

The XSLTTRANSFORM method can now reference the stylesheet by joining the styletab table:

```
SELECT customerXML.XSLTTRANSFORM(styletab.stylesheetcontent, '')
FROM customer, styletab
WHERE customerID=1 and styletab.stylesheetid = 1;
```

Here are partial results from the query. The ellipsis (...) is not part of the query results. It indicates that the query returns additional results that are truncated in the example.

```
customerXML.XSLTTRANSFORM(stylesheetcontent, '')
-----
<html><title>Items Ordered</title><body><table border="1"><tr><th>Item
ID</th><th>Description</th><th>Part Number</th><th>Qty</th><th>Unit Price</
th><th>Total Price</th></tr><tr><td>001</td><td>Motoro ...
```

Example: Converting XML Documents to Rows and Columns

The XMLTABLE function can convert one or more XML documents into rows and columns. It takes as input a row query expression which generates a sequence of nodes. Each sequence of nodes results in a row. A set of column query expressions determines how each of the column values is computed from the nodes.

This example finds all the "item" elements in the input XML documents and for each element found, a row is constructed with the column values constructed from the descendant elements and attributes of that element, such as ItemID or quantity.

```
SELECT X.*
FROM (SELECT * FROM customer WHERE customerID = 1) AS C,
XMLTable (
  '/Customer/Order/Item'
  PASSING C.customerXML
  COLUMNS
    "Seqno" FOR ORDINALITY,
    "CustomerName" VARCHAR(64) PATH '../..Name',
    "OrderNumber" VARCHAR(32) PATH '../@Number',
    "OrderDate" DATE PATH '../@Date',
    "ItemID" VARCHAR(12) PATH '@ID',
    "Quantity" INTEGER PATH 'Quantity',
    "UnitPrice" DECIMAL(9,2),
    "TotalPrice" DECIMAL(9,2) PATH 'Price'
```



```
) AS X ("Sequence #", "Customer Name", "Order #", "Order Date", "Item ID",
"Qty", "Unit Price", "Total Price");
```

The result of the query is:

Sequence#	Customer Name	Order#	Order Date	Item ID	Qty	Unit Price	Total Price

1	John						
Hancock	NW-01-16366	12/02/28	001	10	29.50	295.00	
2	John Hancock	NW-01-16366	12/02/28	101	1	139.99	139.99

Related Information

- The ISSCHEMAVALID method: [ISSCHEMAVALID](#)
- The EXISTSNODE method: [EXISTSNODE](#)
- The XMLQUERY function: [XMLQUERY](#)
- The XSLTTRANSFORM method: [XSLTTRANSFORM](#)
- The XMLTABLE function: [XMLTABLE](#)

Functions for XML Type and XQuery

This section describes functions that provide the following functionality:

Return information about an XML type instance:

- DataSize

Create XML nodes and sequences of XML nodes of different types:

- CREATEXML
- XMLDOCUMENT
- XMLELEMENT
- XMLFOREST
- XMLCONCAT
- XMLCOMMENT
- XMLPI
- XMLTEXT
- XMLAGG

XML processing operations such as parsing, validation and query:

- XMLPARSE
- XMLVALIDATE
- XMLQUERY
- XMLSERIALIZE
- XMLTABLE
- XMLSPLIT

The functions are ANSI SQL/XML (SQL 2008) compliant.

Note:

If the argument you pass to these functions is a large object argument, it is implicitly cast to XML type, and such arguments cannot be larger than 64KB. For example, you cannot construct an element using XMLELEMENT where the content of the element is specified by a large object that is greater than 64KB in size.

The examples in this section reference tables, files, and data defined in:

- [Examples in this Document](#)
- [XML Type Usage Examples](#)

CREATEXML

Creates an XML type value based on the specified text representation.

You can also use the NEW operator to construct an XML type instance.

Result Type

An XML type value that represents the character string specified by the input argument *XML_data*.

If *XML_data* is NULL, CREATEXML returns NULL.

CREATEXML Syntax

```
CREATEXML ( XML_data )
```

Syntax Elements

XML_data

The string representation of an XML value, which must have one of these data types:

- VARCHAR(*n*), where the maximum supported size (*n*) is 65536
- CLOB
- BLOB

Examples: Using CREATEXML to Create XML Type Instances

In the following query, an XML type instance is created from its text representation stored in the customerXMLText column. This column can be a VARCHAR or CLOB type.

```
SELECT customerID, CREATEXML(customerXMLText)
FROM customerText;
```

Here are partial results from the query. The ellipsis (...) is not part of the query results. It indicates that the query returns additional results that are truncated in the example.

```
customerID CREATEXML(customerXMLText)
-----
1 <?xml version="1.0" encoding="UTF-8" ?> <Customer>  <Name>John
Hancock</Name>  <Address>100 1st Street, San Francisco, CA 94118</
Address>  <Phone1>(858)555-1234</Phone1>  <Phone2>(858)555- ...
```

Once an XML type instance is created, you can invoke various XML processing operations on that instance.

```
SELECT customerID, CREATEXML(customerXMLText).XMLEXTRACT('/Customer/
Address', NULL)
FROM customerText;
```

Query result:

```
customerID  CREATEXML(customerXMLText).XMLEXTRACT('/Customer/Address', NULL)
-----
1  <Address>100 1st Street, San Francisco, CA 94118</Address>
```

```
SELECT
customerID, CREATEXML(customerXMLText).ISSCHEMAVALID(schematab.schemaContent,
'Customer', NULL)
FROM customerText, schematab
WHERE schematab.schemaid = 'customerschema.xsd';
```

Query result:

```
customerID  CREATEXML(customerXMLText).ISSCHEMAVALID(schemaContent, 'Customer',
NULL)
-----
1  1
```

Related Information

- The NEW operator in *Teradata Vantage™ - SQL Operators and User-Defined Functions*, B035-1210
- The XMLEXTRACT method: [XMLEXTRACT](#)
- The ISSCHEMAVALID method: [ISSCHEMAVALID](#)

DataSize

Returns the data length in bytes of any of the following Teradata variable maximum length complex data types:

- DATASET
- JSON
- ST_Geometry
- XML

Returns a BIGINT that is the size in bytes of the data object passed in to the function.

DataSize Syntax

```
[TD_SYSFNLIB.] DataSize (var_max_length_cdt)
```

Syntax Elements

var_max_length_cdt

A DATASET, JSON, ST_Geometry, or XML data type object.

DataSize Examples

The following examples use JSON data types.

```
SELECT TD_SYSFNLIB.DataSize (NEW JSON ('{"name" : "Mitzy", "age" : 3}'));

datasize( NEW JSON('{"name" : "Mitzy", "age" : 3}', LATIN))
-----
29
```

```
CREATE TABLE JSON_table (id INTEGER, jsn JSON INLINE LENGTH 1000);

INSERT INTO JSON_table VALUES (100, '{"name" : "Mitzy", "age" : 3}');
INSERT INTO JSON_table VALUES (200, '{"name" : "Rover", "age" : 5}');
INSERT INTO JSON_table VALUES (300, '{"name" : "Princess", "age" : 4.5}');
```

```
SELECT * FROM JSON_table ORDER BY id;
```

```
      id jsn
-----
    100 {"name" : "Mitzy", "age" : 3}
    200 {"name" : "Rover", "age" : 5}
    300 {"name" : "Princess", "age" : 4.5}
```

```
SELECT id, TD_SYSFNLIB.DataSize (jsn) FROM JSON_table ORDER BY id;
```

```
      id      datasize(jsn)
-----
    100              29
    200              29
    300              34
```

XMLQUERY

Evaluates an XQuery query over the XML value.

XMLQUERY Syntax

```
XMLQUERY (
  'xquery_expression'
  [ xml_query_argument ]
  [ RETURNING { CONTENT | SEQUENCE } ]
  [ { NULL | EMPTY } ON EMPTY ]
)
```

Syntax Elements

xml_query_argument

```
PASSING [ BY VALUE ]
        { XML_query_context_item | XML_query_variable_spec [, ...] }
```

XML_query_variable_spec

```
XML_query_variable AS variable_name
```

'xquery_expression'

The XQuery query string supplied as a character string literal.

RETURNING CONTENT

The return value is a document node with one or more child element nodes.

RETURNING SEQUENCE

The return value is a sequence. This is the default.

NULL ON EMPTY

A NULL is returned if the query result is an empty sequence.

EMPTY ON EMPTY

An empty sequence is returned if the query result is an empty sequence.

BY VALUE

XML query arguments are passed in by value. This is the default.

XML_query_context_item

A value expression representing an XML query context item.

You can pass in only one context item.

Examples

These queries illustrate the XMLQUERY syntax.

'(1,2,3)' is an XQuery query that returns a sequence of three integers (1, 2, and 3).

```
SELECT XMLQUERY('(1,2,3)');
```

The result of the query is:

```
XMLQUERY('(1,2,3)')
-----
1 2 3
```

The following query returns a sequence of Item elements. The XQuery query, the path expression /Customer/Order/Item, is evaluated on the XML document in the customerXML column. The PASSING clause passes the document to the query as the context item. The path expression /Customer/Order/Item is evaluated relative to the context item.

```
SELECT XMLQUERY('/Customer/Order/Item'
  PASSING BY VALUE customer.customerXML
  RETURNING CONTENT
  NULL ON EMPTY) FROM customer WHERE customerID = 1;
```

Here are partial results from the query. The ellipsis (...) is not part of the query results. It indicates that the query returns additional results that are truncated in the example.

```
XMLQUERY('/Customer/Order/Item', PASSING BY VALUE customerXML ...
-----
<Item ID="001">    <Quantity>10</Quantity>    <PartNumber>F54709</PartNumber>
<Description>Motorola S10-HD Bluetooth Stereo
Headphones</Description> <UnitPrice>29.50</UnitPrice> <Price...
```

This query is similar to the previous one except that the parameter is passed by name, and it is referenced in the query by its name (\$doc).

```
SELECT XMLQUERY('$doc/Customer/Order/Item'
  PASSING customerXML AS "doc") FROM customer WHERE customerID = 1;
```

The following is an identical query returning XML as a document node with multiple child Item elements. These results do not have a well-formed XML representation. If a well-formed XML document result is desired, wrap the query in a root element.

```
SELECT XMLQUERY('$i/Customer/Order/Item'
  PASSING customerXML AS i
  RETURNING CONTENT) FROM customer WHERE customerID = 1;
```

This query illustrates wrapping the sequence of Item elements in a root element:

```
SELECT XMLQUERY('<root>{$i/Customer/Order/Item}</root>'
  PASSING customerXML AS i
  RETURNING CONTENT) FROM customer WHERE customerID = 1;
```

Here are partial results from the query:

```
XMLQUERY('<root>{$i/Customer/Order/Item}</root>', PASSING BY ...'
-----
<root><Item ID"001">    <Quantity>10</Quantity>
<PartNumber>F54709</PartNumber>
    <Description>Motorola S10-HD Bluetooth Stereo Headphones</Description>
    <UnitPrice>29.50</UnitPrice>...
```

This query returns the names of all customers who ordered an item with a particular item id. The query takes multiple parameters: one context item parameter (the XML document) and one named parameter (the item id).

```
SELECT XMLQUERY('for $i in /Customer/Order/Item
  where $i/@ID = $itemId
  return $i/../../Name'
  PASSING BY VALUE customer.customerXML,
  '001' AS itemId
  RETURNING SEQUENCE
  EMPTY ON EMPTY ) FROM customer WHERE customerID = 1;
```

Here are partial results from the query:


```
XMLQUERY('for $i in /Customer/Order/Item where $i/@ID = $itemId ...
-----
<Name>John Hancock</Name>
```

This query takes multiple named parameters (\$i and \$doc):

```
SELECT XMLQUERY('for $i in $doc/Customer/Order/Item
  where $i/@ID = $itemId
  return $i/../../Name'
  PASSING BY VALUE customer.customerXML AS "doc",
  '001' AS itemId
  RETURNING SEQUENCE
  EMPTY ON EMPTY ) FROM customer WHERE customerId=1;
```

Here are partial results from the query:

```
XMLQUERY('for $i in $doc/Customer/Order/Item where $i/@ID = $itemId ...
-----
<Name>John Hancock</Name>
```

XMLSERIALIZE

Serializes an XML value to its string representation.

XMLSERIALIZE Syntax

```
XMLSERIALIZE (
  { DOCUMENT | CONTENT } XML_value_expr
  [ AS data_type ]
  [ ENCODING XML_encoding_name ]
  [ WITH [NO] BOM ]
  [ VERSION 'char_str_literal' ]
  [ { INCLUDING | EXCLUDING } XMLDECLARATION ]
  [ NO INDENT | INDENT [ SIZE = integer ] ]
)
```

Syntax Elements

DOCUMENT

The XML value is an XML document node.

CONTENT

The XML value is an XQuery document node.

XML_value_expr

An instance of the XML type.

data_type

VARCHAR, CLOB, VARBYTE, or BLOB. If *data_type* is VARCHAR or CLOB, you can specify the character set as LATIN or UNICODE.

XML_encoding_name

An identifier for encodings such as ISO-8859-1 that are supported by Vantage.

You can specify the XML encoding specification only if the data type requested is VARBYTE or BLOB.

See [Encoding Names Supported by Teradata XML](#).

WITH [NO] BOM

Whether or not a byte order mark should be added to the result.

The WITH BOM directive produces a byte order mark in these cases:

- Target type is VARBYTE or BLOB and the encoding is a Unicode encoding (UTF-8, UTF-16, or UTF-32/UCS4)
- Target type is VARCHAR or CLOB with the UNICODE CHARACTER SET and the client character set is UNICODE

In all other cases, the WITH BOM directive will not produce a byte order mark.

Specify the WITH BOM option only with UNICODE client character sets.

VERSION '*char_str_literal*'

The XML version to which the serialization conforms. Valid values:

- '1.0'
- '1.1'

If not specified or if an invalid value is specified, the default is '1.0'.

INCLUDING XMLDECLARATION or EXCLUDING XMLDECLARATION

Whether or not the XML declaration (<?xml..?>) occurs at the beginning of the serialization.

NO INDENT

Specifies no indentation.

INDENT [SIZE = *integer*]

Specifies indentation and the amount of indentation. The default size is 4.

Example: Using XMLSERIALIZE to Serialize an XML Value to a CLOB Value

```
SELECT XMLSERIALIZE(DOCUMENT customerXML AS CLOB CHARACTER SET UNICODE
  WITH BOM INCLUDING XMLDECLARATION)
FROM customer
WHERE customerID = 1;
```

Here are partial results from the query. The ellipsis (...) is not part of the query results. It indicates that the query returns additional results that are truncated in the example.

```
XMLSERIALIZE(DOCUMENT customerXML AS CLOB(2097088000) CHARACTER ...
-----
<?xml version="1.0" encoding="UTF-8" ?> <Customer>  <Name>John
Hancock</Name>  <Address>100 1st Street, San Francisco, CA 94118</
Address>  <Phone1>(858)555-1234</Phone1>  <Phone2>(858)555-9876</Phone> ...
```

Example: Using XMLSERIALIZE to Serialize an XML Value to a BLOB Value

When serializing to BLOB or VARBYTE, you can specify the desired encoding for the serialization. If any of the data in the document cannot be represented in the specified encoding, a character translation error will occur.

```
SELECT XMLSERIALIZE(DOCUMENT customerXML AS BLOB ENCODING "UTF-8"
  WITH BOM INCLUDING XMLDECLARATION)
FROM customer
WHERE customerID = 1;
```

XMLTABLE

The XMLTABLE function converts one or more XML documents into rows and columns. It takes as input a row query expression (*XML_tab_row_pattern*), which returns a sequence of items. The table function returns one row for each item in this sequence. A set of column query expressions (XML table column pattern) determines how each of the column values is computed.

XMLTABLE Syntax

```
XMLTABLE (
  [ XML_namespace_declaration , ]
  'XML_tab_row_pattern'
  [ XML_query_argument ]
  [ COLUMNS { column_name FOR ORDINALITY | regular_column_definition } [,...] ]
)
```

Note:

You can include only one default namespace declaration item. It can appear in any position in the list of comma-separated namespace declarations or at the end of the list.

Syntax Elements

XML_namespace_declaration

```
XMLNAMESPACES (
  { 'XML_namespace_URI' AS XML_namespace_prefix |
    DEFAULT 'XML_namespace_URI' |
    NO DEFAULT
  } [,...]
)
```

Note:

Namespaces declared in the prolog of the queries override any namespace declarations specified here.

XML_query_argument

```
PASSING [ BY VALUE ]
  { XML_query_context_item | XML_query_variable_spec [,...] }
```

regular_column_definition

```
column_name_data_type [ PATH 'char_str_literal' ] [ default_clause ]
```

Note:

The optional clauses can be in any order.

XML_query_variable_spec

```
XML_query_variable AS variable_name
```

XML_query_variable

An XML query variable in the format:

```
value_expression AS identifier
```

'XML_tab_row_pattern'

A valid XQuery expression supplied as a character string literal. One row is generated for each item in the result of this query.

COLUMNS

The columns in the table returned by the function.

If not specified, the function returns a single column, named COLUMN_VALUE, which contains XML typed values resulting from the evaluation of the row pattern query.

column_name

The name of a column in the output table.

FOR ORDINALITY

The column contains an integer that represents a sequence number for the rows generated, starting with 1 for the first row.

'XML_namespace_URI'

The URI (Uniform Resource Identifier) that identifies the XML namespace.

XML_namespace_prefix

The namespace prefix.

BY VALUE

Row pattern query arguments are passed in by value. This is the default.

XML_query_content_item

A value expression representing an XML query context item.

You can pass in only one context item.

column_name_data_type

The data type of a column in the output table.

PATH 'char_str_literal'

The XML table column pattern, a query that returns the column value. This query is evaluated relative to the item returned by the row query; that is, it uses the item returned by the row query as the context item. The resulting value is cast to the column data type.

default_clause

The default value for the column if the column query returns no results.

The default clause is not applicable for certain data types such as CLOB, BLOB, and UDTs.

Example: Using XMLTABLE to Convert XML Data to Relational Data

See [Example: Converting XML Documents to Rows and Columns](#).

Example: Using XMLTABLE Without the COLUMNS Clause

If you omit the COLUMNS clause, the items returned by the '*XML_tab_row_pattern*' query are returned as a column of XML data type.

```
SELECT X.*
FROM (SELECT * FROM customer WHERE customerID = 1) AS C,
     XMLTable (
       '/Customer/Order/Item'
       PASSING C.customerXML
     ) AS X ("ItemXML");
```

Here are partial results from the query. The ellipsis (...) is not part of the query results. It indicates that the query returns additional results that are truncated in the example.

```
ItemXML
-----
<Item ID="001"> <Quantity>10</Quantity> <PartNumber>F54709</
PartNumber> <Description>Motorola S10-HD Bluetooth Stereo
```

```
Headphones</Description> <UnitPrice>29.50</UnitPrice> <Price>
<Item ID="101"> <Quantity>1</Quantity> <PartNumber>Z19743</
PartNumber> <Description>Motorola Milestone XT800 Cell Phone</
Description> <UnitPrice>139.99</UnitPrice> <Price>139.99<
...
```

Example: Using XMLTABLE With the default_clause

This example illustrates the use of the *default_clause* in column definitions. The path for customerName is modified to one that does not match the document structure. The default value is returned for the customer name.

```
SELECT X.*
FROM (SELECT * FROM customer WHERE customerID = 1) AS C,
XMLTable (
  '/Customer/Order/Item'
  PASSING C.customerXML
  COLUMNS
    "Seqno" FOR ORDINALITY,
    "CustomerName" VARCHAR(64) PATH '../NonExistentNode' DEFAULT 'John Doe',
    "OrderNumber" VARCHAR(32) PATH '../@Number',
    "OrderDate" DATE PATH '../@Date',
    "ItemID" VARCHAR(12) PATH '@ID',
    "Quantity" INTEGER PATH 'Quantity',
    "UnitPrice" DECIMAL(9,2),
    "TotalPrice" DECIMAL(9,2) PATH 'Price'
  ) AS X ("Sequence #", "Customer Name", "Order #", "Order Date", "Item ID",
"Qty", "Unit Price", "Total Price");
```

The result of the query is:

Sequence #	Customer Name	Order #	Order Date	Item ID	Qty	Unit Price	Total Price
1	John Doe	NW-01-16366	12/02/28	001	10	29.50	295.00
2	John Doe	NW-01-16366	12/02/28	101	1	139.99	139.99

XMLDOCUMENT

Creates an XQuery document node from an XML type value.

Result Type

The return value is of XML type. The return value is based on the RETURNING clause. If the clause is not specified, the default is RETURNING SEQUENCE.

XMLDOCUMENT Syntax

```
XMLDOCUMENT (
  XML_value_expr
  [ RETURNING { CONTENT | SEQUENCE } ]
)
```

Syntax Elements

XML_value_expr

An instance of the XML type. If *XML_value_expr* is not of XML type, it is implicitly cast to XML type. If the cast is not supported, an error is raised. If *XML_value_expr* is NULL, the result is NULL.

RETURNING CONTENT

The return value is a document node with one or more child element nodes.

RETURNING SEQUENCE

The return value is a sequence (an ordered collection of items, each of which can either be a valid Xquery node or atomic value). This is the default.

Examples: Using XMLDOCUMENT to Create an XML Document Node From an XML Type Value

The following query returns an XML document node with a child element named 'Customer':

```
SELECT XMLDOCUMENT(new XML('<Customer/>'));
```

Query result:

```
XMLDOCUMENT( NEW XML('<Customer/>') RETURNING SEQUENCE )
-----
<Customer></Customer>
```

The following query returns an XML document node with two child elements named 'Name' and 'Address'. This is not a well-formed document according to the XML1.1 specification, but is allowed by the XQuery data model.


```
SELECT XMLDOCUMENT(XMLQUERY('(<Name/>, <Address/>)' ) RETURNING CONTENT);
```

Query result:

```
XMLDOCUMENT(XMLQUERY('(<Name/>, <Address/>)' ) RETURNING CONTENT)
-----
<Name></Name><Address></Address>
```

XMLELEMENT

Constructs an XML element node.

Result Type

An XML type value representing an XML element node with the specified:

- Name
- Namespace declarations
- Attributes
- Content

The return value is based on the RETURNING clause.

XMLELEMENT Syntax

```
XMLEMENT (
  NAME XML_element_name

  [ , XML_namespace_declaration ]

  [ , ( XML_attribute_spec [ , ... ] ) ]

  [ , XML_value_expr
    [ OPTION
      {
        { NULL | EMPTY | ABSENT | NIL } ON NULL |
        NIL ON NO CONTENT
      }
    ]
  ]

  [ RETURNING { CONTENT | SEQUENCE } ]
)
```

Note:

You can include only one default namespace declaration item. It can appear in any position in the list of comma-separated namespace declarations or at the end of the list.

Syntax Elements***XML_namespace_declaration***

```
XMLNAMESPACES (
  { 'XML_namespace_URI' AS XML_namespace_prefix |
    DEFAULT 'XML_namespace_URI' |
    NO DEFAULT
  } [, ...]
)
```

XML namespaces that are scoped when the query expressions (row and column) are evaluated.

Note:

Namespaces declared in the prolog of the queries override any namespace declarations specified here.

XML_attribute_spec

```
XML_attribute_value [ AS XML_attribute_name ]
```

XML_element_name

The name of the element.

XML_value_expr

An instance of the XML type representing the XML element content.

- *XML_value_expr* should be of XML type.
- The element to be constructed will have a name derived by applying the rules in the ANSI SQL/XML specification that results in an XML 1.1 QName.
- If XML element content is provided, you cannot have an attribute named "xsi:nil" (where xsi maps to xml schema instance namespace identified by the namespace URI "http://www.w3.org/2001/XMLSchema-instance").
- If XML element content is NULL, the result value is dictated by the XML content option. If XML content option is not specified, the default is EMPTY ON NULL. If XML element content is NULL and XML content option is:

- NULL ON NULL, NULL is returned
- ABSENT ON NULL, an empty XQuery sequence is returned
- NIL ON NULL, an attribute `xsi:nil=true` is added to the element being constructed (xsi being the prefix associated with xml schema instance schema)
- EMPTY ON NULL, an empty element is constructed according to the rest of the parameters (such as name, namespaces, or attributes) and returned
- If the XML element content is a sequence made up solely of nodes other than text and element nodes, and XML content option is NIL ON NO CONTENT, an attribute `xsi:nil=true` is added to the element being constructed.

OPTION**NULL ON NULL****EMPTY ON NULL****ABSENT ON NULL****NIL ON NULL****NIL ON NO CONTENT**

XML content options.

RETURNING CONTENT

The return value is a document node with one or more child element nodes.

RETURNING SEQUENCE

The return value is a sequence (an ordered collection of items, each of which can either be a valid Xquery node or atomic value). This is the default.

'XML_namespace_URI'

The URI (Uniform Resource Identifier) that identifies the XML namespace.

XML_namespace_prefix

The URI (Uniform Resource Identifier) that identifies the XML namespace.

XML_attribute_value

A value for an attribute of the element.

XML_attribute_name

A name for the attribute. The name cannot be 'xmlns' and no two attributes can have the same qualified name (for example, same namespace and local name).

Example: Using XMLEMENT to Construct an XML Element Node

This query returns an XML element that has a serialization of `<Item ID="001"/>`:

```
SELECT XMLEMENT(NAME Item, XMLATTRIBUTES('001' AS ID));
```

The result of the query is:

```
XMLEMENT(NAME Item, XMLATTRIBUTES('001' AS ID) RETURNING SEQUENCE)
-----
<Item ID="001"></Item>
```

This query returns an XML element that has a serialization of `<Item xmlns="http://example.teradata.com" ID="001"/>`:

```
SELECT XMLEMENT(NAME Item, XMLNAMESPACES(DEFAULT
'http://example.teradata.com'),
XMLATTRIBUTES('001' AS ID));
```

The result of the query is:

```
XMLEMENT(NAME Item, XMLNAMESPACES(DEFAULT 'http://example.teradata.com'),
XMLATTRIBUTES('001' AS ID) RETURNING SEQUENCE)
-----
<Item ID="001" xmlns="http://example.teradata.com"></Item>
```

XMLFOREST

Constructs a sequence of XML elements, which when serialized return a concatenation of the serializations of individual members of the sequence.

Result Type

The return value is of XML type. The return value is based on the RETURNING clause. If the clause is not specified, the default is RETURNING SEQUENCE.

XMLFOREST Syntax

```
XMLFOREST (
  [ XML_namespace_declaration , ]
```

```
[ forest_element_spec [, ...] ]

[ OPTION
  {
    { NULL | EMPTY | ABSENT | NIL } ON NULL |
    NIL ON NO CONTENT
  }
]

[ RETURNING { CONTENT | SEQUENCE } ]
)
```

Note:

You can include only one default namespace declaration item. It can appear in any position in the list of comma-separated namespace declarations or at the end of the list.

Syntax Elements***XML_namespace_declaration***

```
XMLNAMESPACES (
  { 'XML_namespace_URI' AS XML_namespace_prefix |
    DEFAULT 'XML_namespace_URI' |
    NO DEFAULT
  } [, ...]
)
```

XML namespaces that are scoped when the query expressions (row and column) are evaluated.

Note:

Namespaces declared in the prolog of the queries override any namespace declarations specified here.

forest_element_spec

```
forest_element_value [ AS forest_element_name ]
```

OPTION**NULL ON NULL****EMPTY ON NULL****ABSENT ON NULL****NIL ON NULL****NIL ON NO CONTENT**

XML content options.

RETURNING CONTENT

The return value is an XQuery document node with the forest elements as its children.

RETURNING SEQUENCE

The return value is a sequence with the forest elements as items. This is the default.

'XML_namespace_URI'

The URI (Uniform Resource Identifier) that identifies the XML namespace.

XML_namespace_prefix

The namespace prefix.

forest_element_value

A value for the forest element.

forest_element_name

A name for the forest element.

If *forest_element_name* is specified, the name is converted to an XML name. The resulting name is a QName. *forest_element_name* must be a NAME such as "ELEMENTNAME".

If *forest_element_name* is not specified, *forest_element_value* must be a column reference and the column name is converted to an XML name. The resulting name is a QName.

forest_element_value can be of any Teradata predefined data type or XML.

The effect of the XML content option is the same as for XMLELEMENT for each individual forest element constructed.

Example: Using XMLFOREST to Construct a Sequence of XML Elements

```
SELECT XMLFOREST('(858)555-1234' AS Phone1, '(858)555-9876' AS Phone2);
```

The result of the query is:

```
XMLFOREST(' (858)555-1234' AS Phone1, ' (858)555-9876' AS Phone2)
-----
<Phone1>(858)555-1234</Phone1><Phone2>(858)555-9876</Phone2>
```

Related Information

See [XMLELEMENT](#).

XMLCONCAT

Generates a forest of XML values given a list of values as parameters.

XMLCONCAT takes a number of expressions as parameters. Each of these expressions results in a sequence. An item is equivalent to a sequence containing just that item. A NULL is treated like an empty sequence.

The result is a sequence containing the items in all the sequences in order.

The semantics of XML concatenation are specified in pairs where if either of the pair resolves to a NULL, the result is the other value. If both are NULL, the result is a NULL. Therefore in the event that all *XML_value expr* arguments are NULL, the result is a NULL.

Result Type

The return value is of XML type. The return value is based on the RETURNING clause. If the clause is not specified, the default is RETURNING SEQUENCE.

XMLCONCAT Syntax

```
XMLCONCAT (
  XML_value_expr [, ...]
  [ RETURNING { CONTENT | SEQUENCE } ]
)
```

Syntax Elements

XML_value_expr

An instance of the XML type.

RETURNING CONTENT

The return value is an XQuery document node with the concatenated items as its children.

RETURNING SEQUENCE

The return value is a sequence with the concatenated items. This is the default.

Example: Using XMLCONCAT to Concatenate a List of XML Values

The following query returns the sequence (1,2,3,4,5,6).

```
SELECT XMLCONCAT(XMLQUERY('(1,2,3)'), XMLQUERY('(4,5,6)'));
```

Query result:

```
XMLCONCAT(XMLQUERY('(1,2,3)'),XMLQUERY('(4,5,6)') RETURNING SEQUENCE)
-----
1 2 3 4 5 6
```

XMLCOMMENT

Creates a comment node with the comment value given by *char_value_expr*.

Result Type

The return value is of XML type. The return value is based on the RETURNING clause. If the clause is not specified, the default is RETURNING SEQUENCE.

XMLCOMMENT Syntax

```
XMLCOMMENT (
  char_value_expr
  [ RETURNING { CONTENT | SEQUENCE } ]
)
```

Syntax Elements

char_value_expr

A character string.

The data type of *char_value_expr* can be CLOB, CHAR(*n*), or VARCHAR(*n*), where *n* is as close to 64K as permitted. If the type is CLOB, it cannot be larger than 64K.

If *char_value_expr* is NULL, the function returns NULL.

If the comment, derived by concatenating '<!--', the character string returned by *char_value_expr*, and '-->' is not a valid XML comment, an error is raised. For example, *char_value_expr* should not contain the substring '-->'.

RETURNING CONTENT

The return value is an XQuery document node with the comment node as its child.

RETURNING SEQUENCE

The return value is a sequence with the comment node as its only item. This is the default.

Example: Using XMLCOMMENT to Create a Comment Node

The following query returns a sequence with a single comment node whose serialization is `<!--The orders following this comment will be shipped the next day-->`.

```
SELECT XMLCOMMENT('The orders following this comment will be shipped the
next day');
```

Query result:

```
XMLCOMMENT('The orders following this comment will be shipped the next day')
-----
<!--The orders following this comment will be shipped the next day-->
```

XMLPI

Creates an XML processing instruction node with the name determined by *XML_PI_target*.

XMLPI creates an XML processing instruction node with the name specified by *XML_PI_target*.

char_value_expr is evaluated to a character string value using the rules for converting SQL data type values to XML values. A processing instruction node is constructed from the name and character string value so that its serialization is `<? name character-string-value ?>`. If this is not a well-formed XML 1.1 PI, an error is raised.

Result Type

The return value is of XML type. The return value is based on the RETURNING clause. If the clause is not specified, the default is RETURNING SEQUENCE.

XMLPI Syntax

```
XMLPI (
  NAME XML_PI_target
  [, char_value_expr ]
  [ RETURNING { CONTENT | SEQUENCE } ]
)
```

Syntax Elements

XML_PI_target

A valid XML 1.1 NCName.

It should not be the string 'xml' in any upper or lower case combination. For example, 'xml', 'XML', and 'Xml' are not valid.

The data type of *XML_PI_target* is VARCHAR.

char_value_expr

A string used to construct the processing instruction node.

The data type of *char_value_expr* is CLOB, CHAR(*n*), or VARCHAR(*n*), where *n* is as close to 64K as permitted. If the type is CLOB, it cannot be larger than 64K.

If *char_value_expr* is NULL, the function returns NULL.

If *char_value_expr* is omitted, it is treated as a zero-length string.

RETURNING CONTENT

The return value is an XQuery document node with the PI as its child.

RETURNING SEQUENCE

The return value is a sequence with the PI as its only item. This is the default.

Example: Using XMLPI to Create an XML Processing Instruction Node

The following query returns a sequence with a processing instruction whose serialization is `<? orderpi 'NextDayShipment'?>`

```
SELECT XMLPI(NAME orderpi, 'NextDayShipment');
```

Query result:

```
XMLPI( NAME orderpi , 'NextDayShipment' RETURNING SEQUENCE )
-----
<?orderpi NextDayShipment?>
```

XMLTEXT

Creates an XQuery text node with the node value given by *char_value_expr*.

Result Type

The return value is of XML type. The return value is based on the RETURNING clause. If the clause is not specified, the default is RETURNING SEQUENCE.

XMLTEXT Syntax

```
XMLTEXT (
  char_value_expr
  [ RETURNING { CONTENT | SEQUENCE } ]
)
```

Syntax Elements

char_value_expr

A string that specifies the node value.

The data type of *char_value_expr* is CLOB, CHAR(*n*), or VARCHAR(*n*), where *n* is as close to 64K as permitted. If the type is CLOB, it cannot be larger than 64K.

If *char_value_expr* is NULL, the function returns NULL.

RETURNING CONTENT

The return value is an XQuery document node with the text node as its child.

RETURNING SEQUENCE

The return value is a sequence with the text node as its only item. This is the default.

Example: Using XMLTEXT to Create an XML Text Node

The following query returns a sequence with the text node with the node value 'Special instructions for processing this order' as its only item.

```
SELECT XMLTEXT('Special instructions for processing this order');
```

Query result:

```
XMLTEXT('Special instructions for processing this order' RETURNING SEQUENCE)
-----
Special instructions for processing this order
```

XMLPARSE

Performs a nonvalidating parse on *string_value_expr* and returns an XML type instance representing the parsed value.

Result Type

The return value is of XML type. The return value depends on whether you specify DOCUMENT or CONTENT.

XMLPARSE Syntax

```
XMLPARSE (
  { DOCUMENT | CONTENT }
  string_value_expr
  { PRESERVE | STRIP }
  WHITESPACE
)
```

Syntax Elements

DOCUMENT

The function returns an XML document node.

CONTENT

The function returns an XQuery document node.

string_value_expr

The string value to be parsed.

The data type of *string_value_expr* is CLOB or VARCHAR.

If *string_value_expr* is NULL, the function returns NULL.

PRESERVE WHITESPACE

White space is preserved during parsing.

STRIP WHITESPACE

White space is removed during parsing.

Example: Using XMLPARSE With the CONTENT and STRIP WHITESPACE Clauses

```
SELECT XMLPARSE(CONTENT
  '<Customer/>'
  STRIP WHITESPACE);
```

The result of the query is:

```
XMLPARSE( CONTENT '<Customer/>' STRIP WHITESPACE )
-----
<Customer></Customer>
```

Example: Using XMLPARSE With the CONTENT and PRESERVE WHITESPACE Clauses

```
SELECT XMLPARSE(CONTENT
  '<Customer>
    <!--Customer record-->
  </Customer>'
  PRESERVE WHITESPACE);
```

The result of the query is:

```
XMLPARSE( CONTENT '<Customer>          <!--Customer record--> </Customer>'
  PRESERVE WHITESPACE )
-----
<Customer>          <!--Customer record-->  </Customer>
```

Example: Using XMLPARSE With the DOCUMENT and STRIP WHITESPACE Clauses

```
SELECT XMLPARSE(DOCUMENT
  '<Customer xml:space="preserve">
    <Address>Address with lots of space</Address>
    <Address>More space  </Address>
  </Customer>'
  STRIP WHITESPACE);
```

The result of the query is:

```
XMLPARSE( DOCUMENT '<Customer xml:space="preserve"> <Address> ...
-----
<Customer xml:space="preserve">   <Address>Address with lots of space</
Address>       <Address>More space  </Address>  </Customer>
```

Example: Using XMLPARSE With the DOCUMENT and PRESERVE WHITESPACE Clauses

```
SELECT XMLPARSE(DOCUMENT
  '<Customer xml:space="default">
    <Address>Address with lots of space</Address>
    <Address>More space  </Address>
  </Customer>'
  PRESERVE WHITESPACE);
```

The result of the query is:

```
XMLPARSE( DOCUMENT '<Customer xml:space="default"> <Address> ...
-----
<Customer xml:space="default">   <Address>Address with lots of space</
Address>       <Address>More space  </Address>  </Customer>
```

XMLVALIDATE

Validates the XML type instance using the specified XML schema.

XMLVALIDATE uses the XML schema argument to validate the XML type instance. You can optionally specify the actual element declaration to be used for validation. The validation is basically an evaluation of the XQuery expression:

```
validate strict {$seq}
```

where \$seq is a sequence containing the items to be validated (that is, the sequence returned by the *XML_value_expr*).

ANSI Compliance

ANSI SQL/XML supports an optional ACCORDING TO XMLSCHEMA clause that allows you to specify the schema to use for validation of the XML type instance. The schema is identified either through a URI reference or through the registered schema ID. If the clause is not specified, the schema to be used for validation is determined from the namespace of the value being validated (the registered schemas are searched for a schema with a matching target namespace and if one is found, it is used for validation).

Vantage does not support schema registration; therefore, the ACCORDING TO XMLSCHEMA clause is required and you must specify a schema.

Result Type

The result is a copy of the original XML type instance with type annotations added as a result of the validation. Type annotations are made on the internal representation of the XML and are not visible to the user.

XMLVALIDATE Syntax

```
XMLVALIDATE (
  { DOCUMENT | CONTENT | SEQUENCE }
  XML_value_expr
  ACCORDING TO XMLSCHEMA
  VALUE XML_value_expr
  [ NAMESPACE 'XML_URI' | NO NAMESPACE ]
  [ ELEMENT XML_valid_element_name ]
)
```

Syntax Elements

DOCUMENT

XML_value_expr is a sequence of one item, which is a document node with a single element node child.

CONTENT

XML_value_expr is a sequence of one item, which is a document node that can have multiple child elements.

SEQUENCE

XML_value_expr is a sequence.

XML_value_expr

The XML type instance to be validated.

If *XML_value_expr* is NULL, the function returns NULL.

If *XML_value_expr* is an empty sequence, the function returns an empty sequence.

If *XML_value_expr* is a sequence that contains even a single item that is an atomic value, attribute node, namespace node or text node, an error is raised.

ACCORDING TO XMLSCHEMA

The XML schema to use to validate the XML type instance.

VALUE XML_value_expr

An XML type instance representing the XML schema.

XML_valid_element_name

The element declaration to be used for validation.

'XML_URI'

The URI (Uniform Resource Identifier) that identifies the XML schema.

Example: Using XMLVALIDATE to Validate XML Values

This query validates the XML values in the customerXML column of the customer table using the schema in the schemacontent column of the schematab table for the row where the schemaid column has the value 'customerschema.xsd'. The resulting column are copies of the XML values with type annotations.

```
SELECT XMLVALIDATE(DOCUMENT customer.customerXML ACCORDING TO XMLSCHEMA
VALUE schematab.schemacontent)
FROM customer, schematab
WHERE schematab.schemaid = 'customerschema.xsd';
```

Here are partial results from the query. The ellipsis (...) is not part of the query results. It indicates that the query returns additional results that are truncated in the example.

```
XMLVALIDATE(DOCUMENT customerXML ACCORDING TO XMLSCHEMA VALUE...
-----
<Customer> <Name>John Hancock</Name> <Address>100 1st Street, San Francisco,
CA 94118</Address> <Phone1>(858)555-1234</Phone1> <Phone2>(858)555-9876</
Phone2> <Fax>(858)555-9999</Fax> <Email>John...
```

XMLAGG

Performs an aggregate of multiple rows to construct an XML value.

XMLAGG aggregates multiple rows to construct an XML value. During aggregation, for a given collection of rows processed by the function, *XML_value_expr* is evaluated for every row and the resulting values are ordered according to the ORDER BY clause and concatenated according to the XML value concatenation semantics (same as XMLCONCAT), after eliminating NULLs.

Result Type

The return value is of XML type. The return value is based on the RETURNING clause. If the clause is not specified, the default is RETURNING SEQUENCE.

XMLAGG Syntax

```
XMLAGG (
  XML_value_expr
  [ ORDER BY order_by_spec [, ...] ]
  [ RETURNING { CONTENT | SEQUENCE } ]
)
```

Syntax Elements

order_by_spec

```
sort_key [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

XML_value_expr

An instance of the XML type.

ORDER BY

The resulting values are ordered as follows:

- ASC - in ascending order
- DESC - in descending order
- NULLS FIRST - Nulls are ordered first
- NULLS LAST - Nulls are ordered last

RETURNING CONTENT

The return value is an XQuery document node.

RETURNING SEQUENCE

The return value is a sequence. This is the default.

sort_key

Example: Using XMLAGG to Aggregate Multiple Rows to Construct an XML Value

The following query constructs a Department element that has an attribute "name" and child elements "emp" (a list of employees) ordered on the last names of the employees. One department element is created for each WORKDEPT.

```
SELECT XMLSERIALIZE(DOCUMENT XMLDOCUMENT
  (XMLELEMENT(NAME "Department",
    XMLATTRIBUTES(E.WORKDEPT AS "name"),
    XMLAGG(XMLELEMENT ( NAME "emp", E.LASTNAME)
      ORDER BY E.LASTNAME)
    )) AS CLOB(200)) AS "dept_list"
FROM EMPLOYEE E
WHERE E.WORKDEPT IN ('C01', 'E21')
GROUP BY WORKDEPT;
```

XMLSPLIT

The XMLSPLIT table function takes a single XML document as input and returns multiple rows, each containing a smaller document split from the source document. XSLT based shredding or schema based shredding can then be applied to these smaller XML documents.

This function is useful in situations where an XML document needs to be processed in memory (for example, XSLT based shredding, XML Query, XSLT processing, and so on), and it is too large to be loaded into memory given the restrictions placed by the *XMLMemoryLimit* dbcontrol setting. XMLSPLIT requires XML documents to be passed in as parameters of CLOB data type; this is because XMLSPLIT is expected to be used as a pre-processing step to split XML documents into smaller documents before XML type instances are created for further processing. XMLSPLIT returns the split documents as CLOB data type.

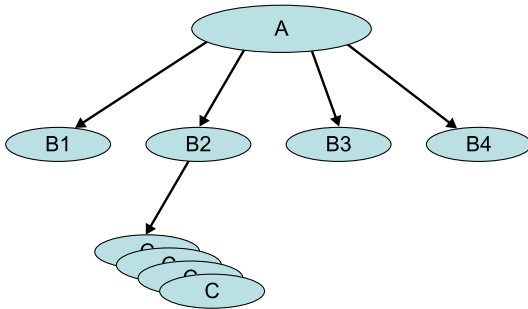
The XMLSPLIT function splits a source XML document into multiple documents, preserving specified parts of the document hierarchy. The semantics of this function depend on the fact that XML documents are large, typically because there is some repeating structure within it. For instance, a customer archive XML document might be large because it has a large number of "Customer" elements within it.

The output documents contain the same basic document structure as the source document up to and including the element specified by *splitPath*. The element identified by the *splitPath* is replicated in its entirety; even if the process exceeds the *splitSize* it does not split off a document at some descendant of the element identified by *splitPath*.

Any elements left over after the end of the last element matching the *splitPath* are returned as the last document. Any other paths that occur earlier in document order will be replicated in all the result documents only if they are specified by the *replicationList* parameter.

All ancestor elements (and their attributes) of the element identified by the *splitPath* are always replicated. In addition, you can choose which of the preceding elements (and its descendants) of the split element will

be replicated, by specifying them in the *replicationList* parameter. This parameter takes a comm- separated list of paths, the wild card character "*" or NULL. The wild card character indicates that all content that occurs earlier in the document, other than the parent element of the split element, will be included in all of the output documents.

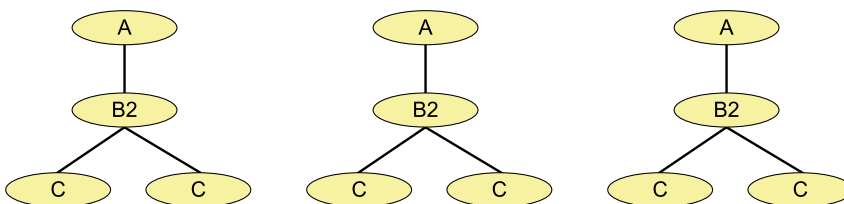


The following example rules can be applied to split a source document with the structure shown above.

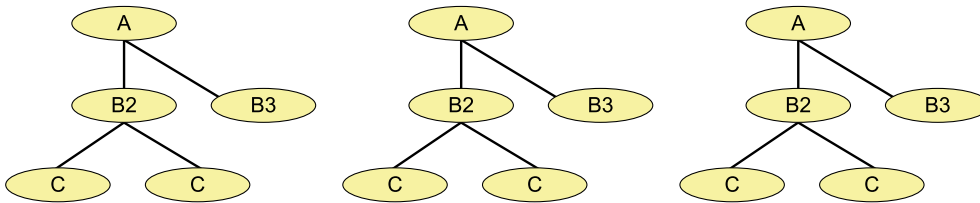
- If *splitPath* is */A/B3* (assuming B3 occurs multiple times), then the element A and its attributes will appear in all the resulting documents. In addition, you can choose to replicate the previous sibling element B1, by specifying the *replicationList* parameter as */A/B1*.
- You can choose that all elements that occur in document order before B3 (which includes elements A, B1 and all its descendants, and B2 and all its descendants) be replicated in each of the output document by specifying "" in the *replicationList* parameter.
- You can choose that a particular descendant of a previous sibling be replicated by specifying the path to it. In the document structure described above, suppose element B1 has two children, C1 and C2. You can choose that element C1 be replicated but not C2 by specifying the replication list as */A/B1/C1*. Any element identified like this, and all of its descendants, will be replicated.

For the figure above, if the *splitPath* is identified as /A/B2/C, the path /A/B2 will occur in each of the output documents with each of the output documents containing enough number of C elements to make the size of the output document conform to the *splitSize* specification. The output documents will contain complete C elements - the process will not stop at some descendant of the C element in one document and continue with it in the next output document.

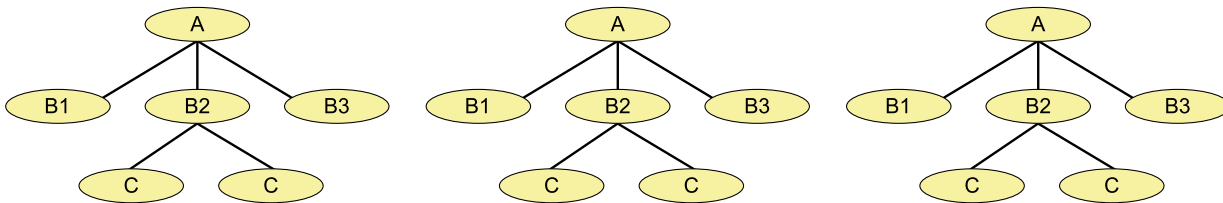
The diagram below shows the resulting documents if the *splitPath* is /A/B2/C.



If the *splitPath* is /A/B2/C and the *replicationList* is /A/B3, the output documents contain the following.



If the *splitPath* is */A/B2/C* and the *replicationList* is *"/A/B1, /A/B3"*, the output documents contain the following.



XMLSPLIT Syntax

```
XMLSPLIT (
  docID,
  sourceDoc,
  splitSize,
  splitPath,
  [ replicationList ]
)
```

Syntax Elements

docID

To correlate output documents to their source documents. INTEGER.

sourceDoc

The source document. CLOB.

splitSize

Size of the resulting document. INTEGER.

splitPath

The path to the recurring element in the document hierarchy where the source document is split. *splitPath* can use wild-card characters, such as *'*'* to indicate a hierarchy level down

a path at which the documents can be split; for example: /A/B/*. VARCHAR. CHARACTER SET UNICODE.

replicationList

[Optional] Comma-separated list of paths to elements that need to be replicated in all the output documents. VARCHAR. CHARACTER SET UNICODE.

Example: XMLSPLIT Usage Example

The example shows how to use XMLSPLIT.

Set up the staging table used to store the XML source document.

```
CREATE TABLE LargeXMLs(id INTEGER, lxml CLOB);

INSERT INTO LargeXMLs VALUES(1, '<?xml version="1.0" encoding="UTF-8"?>
<root>
<customers>
  <customer>
    <c_custkey>1805</c_custkey>
    <c_name>Customer#000001805</c_name>
    <c_address>ZERs4Cu5lQTYD</c_address>
    <c_nationkey>9</c_nationkey>
    <c_phone>19-679-706-1096</c_phone>
    <c_acctbal>-274.75</c_acctbal>
    <c_mktsegment>AUTOMOBILE</c_mktsegment>
    <c_comment>quickly unusual courts alongside of the furiously
      pending requests thrash careful even package</c_comment>
  </customer>
  <customer>
    <c_custkey>1806</c_custkey>
    <c_name>Customer#000001806</c_name>
    <c_address>BB6Vr7W rSIpWKp</c_address>
    <c_nationkey>9</c_nationkey>
    <c_phone>19-872-322-3433</c_phone>
    <c_acctbal>254.17</c_acctbal>
    <c_mktsegment>MACHINERY</c_mktsegment>
    <c_comment>ideas are blithely. ironic instructions wake quickly.
      quickly regular theodolites haggle blithely</c_comment>
  </customer>
  <customer>
    <c_custkey>1807</c_custkey>
    <c_name>Customer#000001805</c_name>
    <c_address>ZERs4Cu5lQTYD</c_address>
```

```

    <c_nationkey>9</c_nationkey>
    <c_phone>19-679-706-1096</c_phone>
    <c_acctbal>-274.75</c_acctbal>
    <c_mktsegment>AUTOMOBILE</c_mktsegment>
    <c_comment>quickly unusual courts alongside of the furiously
        pending requests thrash careful even package</c_comment>
  </customer>
</customers>
<dealers>
  <dealer>
    <d_id>12610</d_id>
    <d_name>VBIT</d_name>
    <d_address>HYD</d_address>
    <d_comment>AP Based dealer</d_comment>
  </dealer>
</dealers>
</root> ');

```

To split the XML document into smaller documents, run the following query with the XMLSPLIT function.

```
SELECT * FROM TABLE(XMLSPLIT(LargeXMLs.id, LargeXMLs.lxml, 400, '/root/customers/
customer', '')) AS xs;
```

Result: The XML source document is split into smaller documents. Note: The output has been formatted for readability.

XMLDoc-1.xml

```

-----
1 <?xml version="1.0" encoding="UTF-16"?>
<root>
  <customers>
    <customer>
      <c_custkey>1805</c_custkey>
      <c_name>Customer#000001805</c_name>
      <c_address>ZERs4Cu5lQTYD</c_address>
      <c_nationkey>9</c_nationkey>
      <c_phone>19-679-706-1096</c_phone>
      <c_acctbal>-274.75</c_acctbal>
    </customer>
  </customers>
</root>

```

XMLDoc-2.xml

```

-----
1 <?xml version="1.0" encoding="UTF-16"?>

```

```

<root>
  <customers>
    <customer>
      <c_custkey>1806</c_custkey>
      <c_name>Customer#000001806</c_name>
      <c_address>BB6Vr7W rSIpWKp</c_address>
      <c_nationkey>9</c_nationkey>
      <c_phone>19-872-322-3433</c_phone>
      <c_acctbal>254.17</c_acctbal>
    </customer>
  </customers>
</root>

```

XMLDoc-3.xml

```

-----
1 <?xml version="1.0" encoding="UTF-16"?>
<root>
  <customers>
    <customer>
      <c_custkey>1807</c_custkey>
      <c_name>Customer#000001805</c_name>
      <c_address>ZERs4Cu5lQTYD</c_address>
      <c_nationkey>9</c_nationkey>
      <c_phone>19-679-706-1096</c_phone>
      <c_acctbal>-274.75</c_acctbal>
    </customer>
  </customers>
</root>

```

XMLDoc-4.xml

```

-----
1 <?xml version="1.0" encoding="UTF-16"?>
<root>
  <dealers>
    <dealer>
      <d_id>12610</d_id>
      <d_name>VBIT</d_name>
      <d_address>HYD</d_address>
      <d_comment>AP Based dealer</d_comment>
    </dealer>
  </dealers>
</root>

```

Methods on the XML Type

Methods on the XML type support common XML operations like parsing, validation, transformation (XSLT) and query (XPath and XQuery).

Wherever string representations of XML values are required, you can use VARCHAR and CLOB typed values for small documents (<64K) and large documents (< 2GB).

IF you want to...	THEN use this method...
add type annotations to an XML value	CREATESCHEMABASEDXML
remove type annotations from XML values	CREATENONSCHEMABASEDXML
execute an XSLT transform on the XML value using a specified stylesheet	XSLTTRANSFORM
execute an XPath or XQuery query against the XML value	XMLEXTRACT
check if a query will return a node	EXISTSNODE
check if the XML value is a well-formed XML document	ISDOCUMENT
check if the XML value is a document node with one or more child elements	ISCONTENT
check if the XML value has been validated and has type annotations	ISSCHEMAVALIDATED
check if the XML value is valid according to a specified schema	ISSCHEMAVALID

Related Information:

[Examples in this Document](#)

[XML Type Usage Examples](#)

CREATESCHEMABASEDXML

Creates a schema-validated and type-annotated XML value.

Result Type

An XML type value representing the XML information set with type annotations. Type annotations are made on the internal representation of the XML and are not visible to the user.

CREATESCHEMABASEDXML Syntax

```
CREATESCHEMABASEDXML ( schema )
```


Syntax Elements

schema

The schema to use to validate the XML value.

The data type of the input argument *schema* must be XML type. If *schema* is NULL or is not a valid schema, an error is raised.

The *schema* should be stored as an XML type value in a table with a unique identifier associated with it. When invoking this method, you can join the schema table and specify the schema by its unique identifier in the WHERE clause.

If *schema* is composed of a single schema document (the schema document does not include or import another schema document), you can pass the schema document as an XML type value to this method.

If *schema* consists of more than one document, then do the following:

1. Use the Schema and Stylesheet Consolidation utility to consolidate the schema document.
2. Pass the consolidated schema document as an XML type value to this method.

Example: Using CREATESCHEMABASEDXML to Create a Schema-validated and Type-annotated XML Value

These statements perform a validating parse on the XML document in the customerXML column using the schema with schemaid customerschema.xsd. If the document is valid, the fast infoset representation of the document is updated with type information.

```
UPDATE customer
SET customerXML = customerXML.CREATESCHEMABASEDXML(
(SELECT schemacontent FROM schematab
WHERE schematab.schemaid='customerschema.xsd'))
WHERE customerID = 1;
```

Related Information

For information about consolidating a schema document, see [Resolving Stylesheet Dependencies](#).

CREATENONSCHEMABASEDXML

Removes type annotations from a schema-based XML value.

Result Type

An XML type value representing the XML information set without type annotations.

CREATENONSCHEMABASEDXML Syntax

```
CREATENONSCHEMABASEDXML ( )
```

Example: Using CREATENONSCHEMABASEDXML to Remove Type Annotations From a Schema-based XML Value

```
UPDATE customer
SET customerXML = customerXML.CREATENONSCHEMABASEDXML()
WHERE customerID = 1;
```

EXISTSNODE

Checks if a query returns a result.

Result Type

An INTEGER value as follows:

- 0 if the query returns NULL or an empty string, or if the result of evaluating the query on the XML instance is an empty sequence.
- 1 in all other cases.

If the input argument *query* is NULL, EXISTSNODE returns NULL.

EXISTSNODE Syntax

```
EXISTSNODE (
  [ filterExpr, ]
  query,
  nsmmap
)
```

Syntax Element***filterExpr***

An optional filter expression that allows for streamed evaluation of the query expression. *filterExpr* is required if the query is being evaluated over a large XML value.

filterExpr must be VARCHAR(*n*), where the maximum supported size (*n*) is 512.

query

The XPath or XQuery query string.

query must be VARCHAR(*n*), where the maximum supported size (*n*) is 8192.

nsmap

Namespace declarations.

nsmap must be VARCHAR(*n*), where the maximum supported size (*n*) is 1024.

Example: Using EXISTSNODE to Check If a Query Will Return a Result

This query returns all ids in the customer table for which the corresponding XML document in the column customerXML has an element named customer whose Name child element has the value "John Hancock".

```
SELECT customerID
FROM customer
WHERE customerXML.EXISTSNODE('/Customer[Name="John Hancock"]', '') = 1;
```

Query result:

```
customerID
-----
          1
```

ISCONTENT

Checks if an XML value is a document node with one or more child elements.

Result Type

An INTEGER value as follows:

- 1 (TRUE), if the XML value is a document node with one or more child elements
- 0 (FALSE)

ISCONTENT Syntax

```
ISCONTENT ( )
```

Example: Using ISCONTENT to Check if an XML Value is a Document Node With Child Elements

```
SELECT customerXML.ISCONTENT()
FROM customer
WHERE customerID = 1;
```

Query result:

```
customerXML.ISCONTENT()
-----
1
```

ISDOCUMENT

Checks if an XML value is a well-formed XML document.

Result Type

An INTEGER value as follows:

- 1 (TRUE), if the XML value is a well-formed XML document
- 0 (FALSE)

ISDOCUMENT Syntax

```
ISDOCUMENT ()
```

Example: Using ISDOCUMENT to Check if an XML Value is a Well-formed XML Document

```
SELECT customerXML.ISDOCUMENT()
FROM customer
WHERE customerID = 1;
```

Query result:

```
customerXML.ISDOCUMENT()
-----
1
```

ISSCHEMAVALID

Checks the XML instance for validity against an XML schema.

Result Type

An INTEGER value as follows:

- 0 (FALSE), if the XML value is not schema-validated
- 1 (TRUE)

If the input argument *schema* is NULL or is not a valid schema, ISSCHEMAVALID raises an error.

ISSCHEMAVALID Syntax

```
ISSCHEMAVALID (
  schema,
  elemDecl,
  ns
)
```

Syntax Elements

schema

The schema used for validating the XML instance.

schema must be an XML type.

The *schema* should be stored as an XML type value in a table with a unique identifier associated with it. When invoking this method, you can join the schema table and specify the schema by its unique identifier in the WHERE clause.

If *schema* is composed of a single schema document (the schema document does not include or import another schema document), you can pass the schema document as an XML type value to this method.

If *schema* consists of more than one document, then do the following:

1. Use the Schema and Stylesheet Consolidation utility to consolidate the schema document.
2. Pass the consolidated schema document as an XML type value to this method.

elemDecl

The element name whose declaration is used for validating the XML instance.

elemDecl must be VARCHAR(*n*), where the maximum supported size (*n*) is 256.

ns

The namespace of the element.

ns must be VARCHAR(*n*), where the maximum supported size (*n*) is 256.

Example: Using ISSCHEMAVALID to Check an XML Instance for Validity Against an XML Schema

```
SELECT customerID, customerXML.ISSCHEMAVALID(schematab.schemacontent,
'Customer', '')
FROM customer, schematab
WHERE schematab.schemaid = 'customerschema.xsd';
```

Query result:

customerID	customerXML.ISSCHEMAVALID(schemaContent, 'Customer', '')
-----	-----
1	1

Related Information

For information about consolidating a schema document, see [Resolving Stylesheet Dependencies](#).

For additional examples of using ISSCHEMAVALID to validate an XML document, see the following:

- [Example: Validating an XML Document](#).
- [Using Schemas](#).

ISSCHEMAVALIDATED

Checks if the XML value represents a validated, type-annotated value.

Result Type

An INTEGER value of 0 (FALSE) if the XML value is not schema-validated and type-annotated.

ISSCHEMAVALIDATED Syntax

```
ISSCHEMAVALIDATED ( )
```

Example: Using ISSCHEMAVALIDATED to Check if an XML Value is a Validated, Type-annotated Value

```
SELECT customerXML.ISSCHEMAVALIDATED()
FROM customer
WHERE customerID = 1;
```

Query result:

```
customerXML.ISSCHEMAVALIDATED()
-----
                                0
```

XMLEXTRACT

Evaluates a query against the XML instance.

Result Type

An XML type value representing the query result.

If the input argument *query* is NULL, XMLEXTRACT returns NULL.

XMLEXTRACT Syntax

```
XMLEXTRACT (
  [ filterExpr, ]
  query,
  nsmap
)
```

Syntax Elements

filterExpr

An optional filter expression that allows for streamed evaluation of the query expression.

filterExpr is required if the query is being evaluated over a large XML value.

filterExpr must be VARCHAR(*n*), where the maximum supported size (*n*) is 512.

query

The XPath or XQuery query string.

query must be VARCHAR(*n*), where the maximum supported size (*n*) is 8192.

nsmap

Namespace declarations.

nsmap must be VARCHAR(*n*), where the maximum supported size (*n*) is 1024.

You can declare namespaces in two ways:

- Declare the namespaces in the *query* string. For example:

```
SELECT CREATEXML('<a xmlns="http://td.com"><b>c</b><b>d</b></a>').XMLEXTRACT('declare namespace td = "http://td.com"; /td:a/td:b', NULL);
```

The namespace is declared at the beginning of the query: `declare namespace td = "http://td.com"`

- Specify the namespaces in the *nsmap* parameter as a string of *prefix=namespace_uri* declarations with a space separating each namespace declaration. For example:

```
SELECT CREATEXML('<a xmlns="http://td.com"><b>c</b><b>d</b></a>').XMLEXTRACT('/td:a/td:b', 'td=http://td.com');
```

Example: Using XMLEXTRACT to Evaluate a Query Against an XML Instance

In this example, the XPath expression (in bold) is extracting the Address elements in the XML documents for the customer named John Hancock.

```
SELECT customerXML.XMLEXTRACT('/Customer[Name="John Hancock"]/Address', '')
FROM customer
WHERE customerID = 1;
```

Query result:

```
customerXML.XMLEXTRACT('/Customer[Name="John Hancock"]/Address', '')
-----
<Address>100 1st Street, San Francisco, CA 94118</Address>
```

XSLTTRANSFORM

Executes an XSLT transformation of an XML value.

Result Type

An XML type value representing the transformation output.

XSLTTRANSFORM Syntax

```
XSLTTRANSFORM (
  xs/,
  parammap
)
```

Syntax Elements

xs/

The stylesheet as an XML value.

xs/ must be an XML type. If *xs/* is NULL, XSLTTRANSFORM returns NULL.

The XSLTTRANSFORM method accepts an XSLT stylesheet as a parameter and transforms the XML value by applying the stylesheet. The *xs/* stylesheet should be stored as an XML type value in a table with a unique identifier associated with it. When invoking this method, you can join the stylesheet table and specify the stylesheet by its unique identifier in the WHERE clause.

If the *xs/* stylesheet is composed of a single stylesheet document (the stylesheet document does not include another stylesheet document), you can pass the stylesheet document as an XML type value to this method.

If the *xs/* stylesheet consists of more than one document, then do the following:

1. Use the Schema and Stylesheet Consolidation utility to consolidate the stylesheet document.
2. Pass the consolidated stylesheet document as an XML type value to this method.

parammap

The parameters for stylesheet processing as semicolon separated *name=value* pairs. The semicolons in the values are escaped as ;;

parammap must be VARCHAR(*n*), where the maximum supported size (*n*) is 8192.

Example: Using XSLTTRANSFORM to Transform an XML Value

```
SELECT customerXML.XSLTTRANSFORM(new XML('<?xml version="1.0"
encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```

<xsl:template match="/Customer">
  <html>
  <title>Items Ordered</title>
  <body>
  <table border="1">
  <tr>
    <th>Item ID</th>
    <th>Description</th>
    <th>Part Number</th>
    <th>Qty</th>
    <th>Unit Price</th>
    <th>Total Price</th>
  </tr>
  <xsl:for-each select="Order/Item">
  <tr>
    <td><xsl:value-of select="@ID"/></td>
    <td><xsl:value-of select="Description"/></td>
    <td><xsl:value-of select="PartNumber"/></td>
    <td><xsl:value-of select="Quantity"/></td>
    <td><xsl:value-of select="UnitPrice"/></td>
    <td><xsl:value-of select="Price"/></td>
  </tr>
  </xsl:for-each>
  </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>'), '')
FROM customer
WHERE customerID = 1;

```

Here are partial results from the query. The ellipsis (...) is not part of the query results. It indicates that the query returns additional results that are truncated in the example.

```

customerXML.XSLTTRANSFORM( NEW XML('<?xml version="1.0" encoding...
-----
<html><title>Items Ordered</title><body><table border="1"><tr><th>Item ID</
th><th>Description</th><th>Part Number</th><th>Qty</th><th>Unit
Price</th><th>Total
Price</th></tr><tr><td>001</td><td>Motoro...

```

Related Information

For information about the Schema and Stylesheet Consolidation utility, see [Resolving Stylesheet Dependencies](#).

For examples of using XSLTTRANSFORM to transform XML values, see the following:

- [Example: XSL Transform on XML Values](#).
- [Example: Using Stylesheets in XSLT Transformations](#).
- [Using Stylesheets](#).

Schema and Stylesheet Management

This section provides recommendations for schema and stylesheet management.

Schema Management

Methods on the XML type such as `CREATESCHEMABASEDXML` and `ISSCHEMAVALID` take an XSD (XML Schema Definition) schema as a parameter and validate the XML type instance against it. Teradata recommends that you store the schemas as XML type values in tables with unique identifiers associated with them. When invoking a method that requires the schema as an input argument, you can join the schema table and specify the schema by its unique identifier in the `WHERE` clause.

Resolving Schema Dependencies

An XML schema consists of one or more XSD documents. Each of these schema documents can import or include other schema documents, resulting in a series of dependencies that must be resolved in order to obtain a complete XML schema. The XML type methods that require a schema expect a consolidated schema in which all of the dependencies have been resolved.

The consolidated schema document consists of an envelope which contains the XML representations of all the component schema documents. This consolidated schema document can then be used to construct the grammar for the intended schema.

You can use the Schema and Stylesheet Consolidation utility to consolidate your schema document before passing the schema to an XML method.

You can download sample schemas and the Schema and Stylesheet Consolidation utility from [Teradata Downloads](#).

Example: Invoking the ConsolidateSchema Utility

```
C:\Work\Schemas> ConsolidateSchema root.xsd conschema.xsd
Reading root.xsd
Retrieving dependency ..... Imp1.xsd
Retrieving dependency ..... incl.xsd
Retrieving dependency ..... Imp22.xsd
Writing consolidated schema ..... conschema.xsd
C:\Work\Schemas>
```

Example: Consolidated Schema Document

The following is a sample consolidated schema document:

```

<?xml version="1.0" encoding="UTF-8"?>
<ConsolidatedSchema
  schemaURI="http://www.example.com/schemauri" >
  <ComponentSchema locationURI="http://example.com/schemauri"
    targetNamespace="http://www.example.com/tns">
    <xs:schema xmlns:xs="..." xmlns:xsi="..."
      xmlns:tns="http://www.example.com/tns">
      ....
    </xs:schema>
  </ComponentSchema>
  <ComponentSchema locationURI="http://www.example.com/includedschema1.xsd"
    targetNamespace="http://www.example.com/tns">
    <xs:schema xmlns:xs="..." xmlns:xsi="..."
      xmlns:tns="http://www.example.com/tns">
      ....
    </xs:schema>
  </ComponentSchema>
  <ComponentSchema locationURI=" ../otherschema/importedschema1.xsd"
    targetNamespace="http://www.example.com/tns2">
    <xs:schema xmlns:xs="..." xmlns:xsi="..."
      xmlns:tns2="http://www.example.com/tns2">
      ....
    </xs:schema>
  </ComponentSchema>
</ConsolidatedSchema>

```

The ConsolidatedSchema element represents the envelope which will contain the individual schema documents that make up the schema. It identifies the schema in question through a schemaURI attribute, making it a complete self-identifying package.

The ComponentSchema element, which is a child of the ConsolidatedSchema element, will wrap each individual schema document that makes up the schema grammar. This element contains a locationURI attribute that is used to pick out the schema document when it is required during the processing of the root schema. For example, if a schema with a given location URI is imported in the root schema or one of the other imported/included schemas, the resolver will pick the ComponentSchema element with the appropriate locationURI attribute and take its xs:schema child element.

Using Schemas

The CREATESCHEMABASEDXML and ISSCHEMAVALID methods accept a schema as an argument. For XML schemas that are composed of a single schema document (the schema document does not include or import another schema document), you can pass the schema document as an XML type value to the method. For XML schemas that are made up of more than one document, you must pass the consolidated schema document as an XML type value to the method.

You can store the schema or consolidated schema document in a table and use it in a query that performs schema validation through a join as follows:

```
SELECT customerID, customerXML.ISSHEMAVALID(schematab.schemacontent,
'Customer', '')
FROM customer, schematab
WHERE schematab.schemaid = 'customerschema.xsd';
```

If you want to perform multiple validations per row, you can use subqueries of the following form:

```
SELECT x.xmlcol1.isschemavalid(s1.schemacontent, '', ''),
       x.xmlcol2.isschemavalid(s2.schemacontent, '', '')
FROM xmldata x,
     (select st.schemacontent
      from schematab st
      where st.schemaid = 'myFirstSchema.xsd') s1,
     (select st.schemacontent
      from schematab st
      where st.schemaid = 'mySecondSchema.xsd') s2;
```

Stylesheet Management

The XSLTTRANSFORM method on the XML type accepts an XSLT stylesheet as a parameter and transforms the XML value by applying the stylesheet. Teradata recommends that you store the stylesheets as XML type values in tables with unique identifiers associated with them. When invoking the XSLTTRANSFORM method, you can join the stylesheet table and specify the stylesheet by its unique identifier in the WHERE clause.

Resolving Stylesheet Dependencies

An XSLT stylesheet can include other stylesheets to reuse transformations. The XSLTTRANSFORM method expects either a stylesheet with no includes or a consolidated stylesheet, which is a stylesheet document resulting from resolving all the dependencies for the stylesheet and packaging those dependencies in a single XML envelope. This package is now a self-contained stylesheet definition which can be passed to methods and functions that need to perform XSLT transformations.

You can use the Schema and Stylesheet Consolidation utility to consolidate your stylesheet before passing the stylesheet to the XSLTTRANSFORM method.

You can download sample stylesheets and the Schema and Stylesheet Consolidation utility from [Teradata Downloads](#).

Example: Invoking the ConsolidateStylesheet Utility

```
C:\Work\Stylesheets> ConsolidateStylesheet root.xslt constyle.xslt
Reading root.xslt
Retrieving dependency ..... incl1.xslt
Retrieving dependency ..... incl11.xslt
Retrieving dependency ..... incl12.xslt
Writing consolidated stylesheet ..... constyle.xslt
C:\Work\Stylesheets>
```

Example: Consolidated Stylesheet Document

The following is a sample consolidated stylesheet document:

```
<?xml version="1.0" encoding="UTF-8"?>
<ConsolidatedStylesheet
  stylesheetURI="http://www.example.com/stylesheeturi" >
  <ComponentStylesheet locationURI="http://example.com/stylesheeturi"
    <xsl:stylesheet ...>
      ....
    </xsl:stylesheet>
  </ComponentStylesheet>
  <ComponentStylesheet locationURI="../xslt/includedstylesheet1.xsl"
    <xsl:stylesheet ...>
      ....
    </xsl:stylesheet>
  </ComponentStylesheet>
</ConsolidatedStylesheet>
```

The ConsolidatedStylesheet element is the document element for the consolidated stylesheet document. It is the envelope that contains the individual component stylesheets. Each component is wrapped in a ComponentStylesheet element which includes a locationURI attribute. The base stylesheet has a stylesheetURI attribute whose value matches the locationURI of one of the components. The stylesheet is compiled starting at this component, and stylesheet includes occurring in that stylesheet are resolved using the locationURI attribute on the ComponentStylesheet elements.

Using Stylesheets

The XSLTTRANSFORM method accepts a stylesheet as an argument along with a parameter map. For XSLT stylesheets that are composed of a single stylesheet document (the stylesheet document does not include another stylesheet document), you can pass the stylesheet document as an XML type value to the method. For XSLT stylesheets that are made up of more than one document, you must pass the consolidated stylesheet document as an XML type value to the method.

You can store the stylesheet or consolidated stylesheet document in a table and use it in a query that performs XSLT transformation through a join or subquery as follows:

```
SELECT x.xmlcol.xslttransform(s.stylesheetcontent, '')
FROM xmldata x, styletab s
WHERE s.stylesheetid = 'myAppStylesheet.xslt';
```

If you want to perform multiple transformations per row, you can use subqueries of the following form:

```
SELECT x.xmlcol1.xslttransform(s1.stylesheetcontent, ''),
       x.xmlcol2.xslttransform(s2.stylesheetcontent, '')
FROM xmldata x,
     (select st.stylesheetcontent
      from styletab st
      where st.stylesheetid = 'myFirstStylesheet.xslt') s1,
     (select st.stylesheetcontent
      from styletab st
      where st.stylesheetid = 'mySecondStylesheet.xslt') s2;
```

Related Information

- The CREATESCHEMABASEDXML method: [CREATESCHEMABASEDXML](#)
- The ISSCHEMAVALID method: [ISSCHEMAVALID](#)
- The XSLTTRANSFORM method: [XSLTTRANSFORM](#)

XML Shredding and Publishing

XML Shredding and Publishing

This section describes the XML shredding and publishing process and the stored procedures that provide this functionality:

- XML publishing stored procedures allow you to publish the results of an SQL query in XML format:
 - XMLPUBLISH_STREAM
 - XMLPUBLISH
- XML shredding stored procedures load XML documents into relational tables:
 - The AS_SHRED_BATCH stored procedure provides streamed shredding functionality with the mapping provided as an annotated schema.
 - The XSLT_SHRED and XSLT_SHRED_BATCH stored procedures provide shredding functionality with the mapping provided as an XSLT stylesheet. XSLT_SHRED is used for single XML document shredding and XSLT_SHRED_BATCH is used to shred multiple XML documents. In comparison to streamed shredding, the XSLT-based shredding procedures load the document being shredded into memory. While this requires more memory, it makes shredding more flexible.

Related Information

The schema based shredding examples in this section reference tables, files, and data defined in:

- [Examples in this Document](#)
- [XML Type Usage Examples](#)

The XSLT stylesheet based shredding examples in this section reference tables, files, and data defined in:

- [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#)

XML Shredding Based on a Schema

XML shredding is the process of extracting values from XML documents and using them to update tables in the database. This is done based on a user-defined mapping specification that maps the XML structure to the target tables.

For example, you can perform these typical tasks to shred XML documents:

1. Load the XML document to be shredded into an XML column of a staging table.
2. Define an SQL query that returns a result set with 2 columns:
 - An ID column
 - The XML column that contains the XML document to be shredded

3. Create an annotated schema that maps the XML document structure to the target tables.
4. Store the mapping so that it can be easily referenced, for example, in a schema repository table.
5. Call the AS_SHRED_BATCH stored procedure with the following arguments:
 - The SQL query you defined in step 2
 - The annotated schema you saved in step 4

Mapping XML Documents to Target Tables

For streamed shredding, you create W3C XML schema documents to define the mapping that describe the XML documents to be shredded. A schema document is an XML document that describes the syntax and structure of XML documents. You add annotations to the document to describe how different parts of the XML document map to target tables in the database.

Annotations in XML schema documents are comments that the XML shredding procedures interpret as instructions about what data items to extract from conforming XML documents and how to use these data items in updating the target tables. The annotations are ignored by applications that do not know how to interpret them.

Adding Schema Annotations

To add an annotation to an XML schema, you add an `xs:annotation` element with an `xs:appinfo` as its child element, and any application specific elements under it.

This is an example of a Context annotation that is used to explain the various elements that occur as part of the annotations:

```
<xs:annotation>
  <xs:appinfo>
    <context xmlFeatureVersion="1.0" xmlns="http://www.teradata.com/xml">
      <defaultDatabase>XMLTYPE_TEST</defaultDatabase>
      <defaultEncoding>ISO-8859-1</defaultEncoding>
      <rootElement ref="_customer_item_"/>
      <transaction>
        <operation type="insert">
          <table name="CUSTDTL">
            <column name="ID" ref="_customerID_item_" path="Customer/@ID">
              <sqltype name="char">
                <constraint name="length">9</constraint>
              </sqltype>
            </column>
            ...
            ...
          </table>
        </operation>
      </transaction>
    </context>
  </xs:appinfo>
</xs:annotation>
```

The Context annotation is the most important annotation in the schema for defining how a document should be shredded. You add it as the first annotation under the `xs:schema` element in the root schema document that describes the XML documents to be shredded. The following sections describe some of the key components of the context annotation.

defaultDatabase Element

Use the `defaultDatabase` element to identify the default database. If you reference a target table using an unqualified name, the table is assumed to reside in the default database. To refer to a table in a database other than the default database, the annotations should include the fully qualified name in the format:

```
databasename.tablename
```

defaultEncoding Element

Use the `defaultEncoding` element to identify the default encoding. The default encoding is the encoding to which text values extracted from the documents will be converted before they are used in table updates.

rootElement Element

Use the `rootElement` element to identify the element declaration in the schema that is the Document element of the XML documents to be shredded. Most schemas contain many global element declarations (elements that are declared at the top level). Any of these elements can be the Document element (the top level element in the document tree). By specifying a root element, you establish the XML tree structure. For instance, if you are interested in shredding Purchase Order documents, you can specify the `rootElement` to be the `PurchaseOrder` element.

Transaction and Operation Elements

Use the `Transaction` element to group together operations that are executed as a single transaction during shredding. Only one transaction is allowed. A transaction contains one or more `Operation` elements, each describing an insert, update, delete or upsert operation against a single table.

Each `Operation` element has a `type` attribute that describes the type of the operation. This attribute has one of the following values:

- "insert"
- "update"
- "delete"
- "upsert"

The `Operation` element contains a `Table` element that identifies the target table to be updated by the shredding process.

Table Element

The Table element is a child of the Operation element and identifies the target table that is updated by the shredding process.

The Table element has the following components:

- A name attribute that specifies a table name. If this is an unqualified name, the attribute refers to a table in the default database.
- A key attribute that identifies the primary key of the table. The value of this attribute can be a comma-separated list of column names.
- A number of Column elements as its child elements, which describe a row template that is used in the update operations

Column Element

The Column element is a child of the Table element. Each Column element has the following components:

- A name attribute that identifies the column name in the table
- An optional sqltype child element that describes the data type of the column
- An optional sqlexpr child element that specifies an SQL expression that generates the value for the column. Sqlexpr is required when the extracted value will be modified before shredding. Example:

```
<column name="Sal"><sqlexpr>case when Grade=10 then Sal_Temp*2 else
Sal_Temp*3 END</sqlexpr></column>
```

See complete example in the next section.

- Additional attributes, such as an optional attribute named transient, which if true indicates that the associated value will be used in the computation of a target table column (that is, the data item itself will not be inserted into the target table).

Transient is required when the extracted value will be used in sqlexpr. For transient columns, sqltype is mandatory since the column will not exist in the target table. Example:

```
<column name="Sal_Temp" transient="true" ref="_Sal_item_" path="Employee/
Sal"><sqltype name="Integer"/></column>
```

See complete example in the next section.

The td:item attribute appears on element or attribute declarations in the schema. Validating XML parsers will ignore this attribute, just as they will ignore the annotations that represent the mappings.

The td:item attribute is used to provide a unique identifier for each element or attribute that can occur in an XML document. This unique identifier is used in the ref attribute of the Column elements to identify the element or attribute from which the value of the column is sourced when shredding the XML document to target tables. Every ref attribute on a Column element must match a td:item attribute defined elsewhere in the schema.

Example: Using sqlexpr and transient Within a Column Element

```
CREATE TABLE TRANSIENT_SQLEXPR_DOC(Id INTEGER, XmlDoc XML);
```

```
INSERT INTO TRANSIENT_SQLEXPR_DOC VALUES(1, CREATEXML('
<Employees>
  <Employee>
    <Id>1000</Id>
    <Name>TIES</Name>
    <Grade>10</Grade>
    <Sal>40000</Sal>
  </Employee>
  <Employee>
    <Id>1001</Id>
    <Name>TIES</Name>
    <Grade>11</Grade>
    <Sal>60000</Sal>
  </Employee>
</Employees>'));
```

```
CREATE TABLE EMP(Id Integer, Name VARCHAR(100), Grade Integer,
Sal Decimal(18,4));
```

```
CALL TD_SYSXML.AS_SHRED_BATCH(
'sel id, xmldoc from TRANSIENT_SQLEXPR_DOC',
CREATEXML('<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:td="http://
www.teradata.com/xml">
  <xs:annotation>
    <xs:appinfo>
      <context xmlFeatureVersion="13.00.00.00" xmlns="http://
www.teradata.com/xml">
        <defaultDatabase>dr178882</defaultDatabase>
        <defaultEncoding>ISO-8859-1</defaultEncoding>
        <rootElement ref="_Employee_item_"/>
        <transaction>
          <operation type="insert">
            <table name="EMP">
              <column name="Id" ref="_Id_item_" path="Employee/Id"/>
              <column name="Name" ref="_Name_item_" path="Employee/Name"/>
              <column name="Grade" ref="_Grade_item_" path="Employee/Grade"/>
              <column name="Sal_Temp" transient="true" ref="_Sal_item_"
path="Employee/Sal"><sqltype name="Integer"/></column>
```

```

        <column name="Sal"><sqlexpr>case when Grade=10 then Sal_Temp*2
else Sal_Temp*3 END</sqlexpr></column>
    </table>
</operation>
</transaction>
</context>
</xs:appinfo>
</xs:annotation>
<xs:element name="Employees">
    <xs:complexType>
        <xs:sequence>
            <xs:element maxOccurs="unbounded" ref="Employee"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="Employee" td:item="_Employee_item_">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Id" td:item="_Id_item_" type="xs:int"/>
            <xs:element name="Name" td:item="_Name_item_" type="xs:string"/>
            <xs:element name="Grade" td:item="_Grade_item_" type="xs:int"/>
            <xs:element name="Sal" td:item="_Sal_item_" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>'), '', :res);

```

```
SEL * FROM EMP;
```

Query results:

Id	Name	Grade	Sal
1000	TIES	10	80000.0000
1001	TIES	11	180000.0000

retainedElements Element

XML shredding is performed in a streamed fashion in which the XML document is processed in one pass as a stream of characters. One consequence of this is that no state is maintained. For example, element values are discarded as the parser moves on to the next element. However, in some cases, some values occurring in a subtree early in the document might be of use throughout the document.

Consider an XML document that represents purchase orders for an entire day of sales for a store:

```

<purchaseorders>
  <meta>
    <store no="12345"/>
    <date>09/11/2011</date>
  <meta>
  <purchaseorder>
    <!-- content of the purchase order -->
  </purchaseorder>
  <purchaseorder>
    <!--content of the purchase order -->
  </purchaseorder>
  <!-- more purchaseorders -->
</purchaseorders>

```

The /purchaseorders/meta/date element identifies the date that applies to all the purchase orders in the document. In this case, the shredding processor must retain this element throughout the processing of the document. To use the minimum amount of memory, the mapping should indicate which elements need to be retained. For this purpose, use the retainedElements instruction as follows:

```

<retainedElements>
  <element ref="_meta_item" path="purchaseorders/meta"/>
  <!-- more retained element can be appended -->
</retainedElements>

```

Example: Annotated Schema

In this example, you store the annotated schema in a user table and then use it to shred an XML document that is valid according to the schema. You can use this schema to shred the sample XML document, Cust001.xml. The file containing the schema is named annotatedcustomerschema.xsd. The default database is XMLTYPE_TEST.

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:td="http://www.teradata.com/xml">
  <xs:annotation>
    <xs:appinfo>
      <context xmlFeatureVersion="1.0" xmlns="http://www.teradata.com/xml">
        <defaultDatabase>XMLTYPE_TEST</defaultDatabase>
        <defaultEncoding>ISO-8859-1</defaultEncoding>
        <rootElement ref="_customer_item"/>
        <transaction>
          <operation type="insert">
            <table name="CUSTDTL">

```

```

    <column name="ID" ref="_customerID_item_" path="Customer/@ID">
      <sqltype name="char">
        <constraint name="length">9</constraint>
      </sqltype>
    </column>
    <column name="NAME" ref="_name_item_" path="Customer/Name">
      <sqltype name="varchar"/>
    </column>
    <column name="ADDRESS" ref="_address_item_" path="Customer/Address">
      <sqltype name="varchar"/>
    </column>
    <column name="PHONE1" ref="_phone1_item_" path="Customer/Phone1">
      <sqltype name="varchar"/>
    </column>
    <column name="PHONE2" ref="_phone2_item_" path="Customer/Phone2">
      <sqltype name="varchar"/>
    </column>
    <column name="FAX" ref="_fax_item_" path="Customer/Fax">
      <sqltype name="varchar"/>
    </column>
    <column name="EMAIL" ref="_email_item_" path="Customer/Email">
      <sqltype name="varchar"/>
    </column>
  </table>
</operation>
<operation type="insert">
  <table name="ORDERDTL">
    <column name="CUSTOMER_ID" ref="_customerID_item_" path="Customer/@ID">
      <sqltype name="char">
        <constraint name="length">9</constraint>
      </sqltype>
    </column>
    <column name="ORDER_ID" ref="_orderNumber_item_" path="Customer/Order/@Number">
      <sqltype name="char">
        <constraint name="length">11</constraint>
      </sqltype>
    </column>
    <column name="ORDER_DATE" ref="_orderDate_item_" path="Customer/Order/@Date">
      <sqltype name="date"/>
    </column>
    <column name="CONTACT" ref="_contact_item_" path="Customer/Order/Contact">
      <sqltype name="varchar"/>
    </column>
    <column name="CONTACT_PHONE" ref="_phone_item_" path="Customer/Order/Phone">

```



```

        <sqltype name="varchar"/>
      </column>
      <column name="SHIP_TO_ADDRESS" ref="_shipTo_item_" path="Customer/Order/ShipTo">
        <sqltype name="varchar"/>
      </column>
      <column name="SUB_TOTAL" ref="_subTotal_item_" path="Customer/Order/SubTotal">
        <sqltype name="decimal"/>
      </column>
      <column name="TAX" ref="_tax_item_" path="Customer/Order/Tax">
        <sqltype name="decimal"/>
      </column>
      <column name="TOTAL" ref="_total_item_" path="Customer/Order/Total">
        <sqltype name="decimal"/>
      </column>
    </table>
  </operation>
  <operation type="insert">
    <table name="LINEITEMDTL">
      <column name="ORDER_ID" ref="_orderNumber_item_" path="Customer/Order/@Number">
        <sqltype name="char">
          <constraint name="length">11</constraint>
        </sqltype>
      </column>
      <column name="ITEM_ID" ref="_itemID_item_" path="Customer/Order/Item/@ID">
        <sqltype name="varchar"/>
      </column>
      <column name="QUANTITY" ref="_quantity_item_" path="Customer/Order/
Item/Quantity">
        <sqltype name="integer"/>
      </column>
      <column name="PART_NUMBER" ref="_partNumber_item_" path="Customer/Order/
Item/PartNumber">
        <sqltype name="char">
          <constraint name="length">6</constraint>
        </sqltype>
      </column>
      <column name="DESCRIPTION" ref="_description_item_" path="Customer/Order/
Item/Description">
        <sqltype name="varchar"/>
      </column>
      <column name="UNIT_PRICE" ref="_unitPrice_item_" path="Customer/Order/
Item/UnitPrice">
        <sqltype name="decimal"/>
      </column>

```

```

        <column name="PRICE" ref="_price_item_" path="Customer/Order/Item/Price">
            <sqltype name="decimal"/>
        </column>
    </table>
</operation>
</transaction>
</context>
</xs:appinfo>
</xs:annotation>
<xs:element name="Customer" td:item="_customer_item_">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Name" type="xs:string" td:item="_name_item_" />
            <xs:element name="Address" type="xs:string" td:item="_address_item_" />
            <xs:element name="Phone1" type="xs:string" td:item="_phone1_item_" />
            <xs:element name="Phone2" type="xs:string" td:item="_phone2_item_" />
            <xs:element name="Fax" type="xs:string" td:item="_fax_item_" />
            <xs:element name="Email" type="xs:string" td:item="_email_item_" />
            <xs:element ref="Order" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="ID" type="xs:string" td:item="_customerID_item_" />
    </xs:complexType>
</xs:element>
<xs:element name="Order" td:item="_order_item_">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Contact" type="xs:string" td:item="_contact_item_" />
            <xs:element name="Phone" type="xs:string" td:item="_phone_item_" />
            <xs:element name="ShipTo" type="xs:string" td:item="_shipTo_item_" />
            <xs:element name="SubTotal" type="xs:float" td:item="_subTotal_item_" />
            <xs:element name="Tax" type="xs:float" td:item="_tax_item_" />
            <xs:element name="Total" type="xs:float" td:item="_total_item_" />
            <xs:element ref="Item" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="Number" type="xs:string" td:item="_orderNumber_item_" />
        <xs:attribute name="Date" type="xs:date" td:item="_orderDate_item_" />
    </xs:complexType>
</xs:element>
<xs:element name="Item">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Quantity" type="xs:integer" td:item="_quantity_item_" />
            <xs:element name="PartNumber" type="xs:string" td:item="_partNumber_item_" />
            <xs:element name="Description" type="xs:string" td:item="_description_item_" />

```

```

        <xs:element name="UnitPrice" type="xs:float" td:item="_unitPrice_item_"/>
        <xs:element name="Price" type="xs:float" td:item="_price_item_"/>
    </xs:sequence>
    <xs:attribute name="ID" type="xs:string" td:item="_itemID_item_"/>
</xs:complexType>
</xs:element>
</xs:schema>

```

The tables referenced in the schema are defined as:

```

CREATE SET TABLE CUSTDTL ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT (
        ID                CHAR(9),
        NAME               VARCHAR(256),
        ADDRESS            VARCHAR(256),
        PHONE1             VARCHAR(16),
        PHONE2             VARCHAR(16),
        FAX                VARCHAR(16),
        EMAIL              VARCHAR(64)
    ) PRIMARY INDEX CUSTID_NUPI (ID);
CREATE SET TABLE ORDERDTL ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT (
        CUSTOMER_ID       CHAR(9),
        ORDER_ID          CHAR(11),
        ORDER_DATE        DATE,
        CONTACT            VARCHAR(256),
        CONTACT_PHONE      VARCHAR(16),
        SHIP_TO_ADDRESS    VARCHAR(256),
        SUB_TOTAL          DECIMAL(10,2),
        TAX                DECIMAL(10,2),
        TOTAL              DECIMAL(10,2)
    ) PRIMARY INDEX ORDERID_NUPI (ORDER_ID);
CREATE MULTISET TABLE LINEITEMDTL ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT (
        ORDER_ID          CHAR(11),
        ITEM_ID           VARCHAR(6),
        QUANTITY           INTEGER,
        PART_NUMBER        CHAR(6),

```

```

DESCRIPTION      VARCHAR(512),
UNIT_PRICE       DECIMAL(10,2),
PRICE            DECIMAL(10,2)
);

```

Example: Shredding an XML Document

This example shows how you can use the annotated schema defined in the previous example (annotatedcustomerschema.xsd) to shred the sample XML document, Cust001.xml.

The tables referenced in this example are defined as:

```

CREATE TABLE SHREDMAPPING (
  SCHEMA_ID VARCHAR(32),
  ANNOTATED_SCHEMA XML )
  PRIMARY INDEX (SCHEMA_ID);
CREATE TABLE CUSTOMER (
  customerID INTEGER,
  customerName VARCHAR(256),
  customerXML XML )
  PRIMARY INDEX (customerID);

```

1. Load the annotated schema into the shredmapping table:

```

.IMPORT VARTEXT '|' LOBCOLS=1 FILE=annotatedschema.txt
USING (asxsd XML AS DEFERRED, schemaid varchar(32))
INSERT into SHREDMAPPING values(:schemaid, :asxsd);

```

The contents of the annotatedschema.txt file are:

```
annotatedcustomerschema.xsd|annotatedcustomerschema.xsd
```

Here are partial contents of the shredmapping table after the INSERT operation:

SCHEMA_ID	ANNOTATED_SCHEMA
annotatedcustomerschema.xsd	<?xml version="1.0" encoding="UTF-8" ?> <xs:schema...

2. Load the XML document into the customer table:

```
.IMPORT VARTEXT '|' LOBCOLS=1 FILE='custdocs.txt'
USING (custdoc XML AS DEFERRED, custid VARCHAR(256), custname VARCHAR(256))
INSERT INTO CUSTOMER(CAST(:custid AS INTEGER), :custname, :custdoc);
```

The contents of the custdocs.txt file are:

```
Cust001.xml|1|John Hancock
```

3. Define an SQL query that returns a result set with 2 columns: an ID column and the XML column that contains the XML document to be shredded:

```
SELECT customerID, customerXML FROM CUSTOMER;
```

Here are partial results from the query. The ellipsis (...) is not part of the query results. It indicates that the query returns additional results that are truncated in the example.

```
customerID customerXML
-----
1 <?xml version="1.0" encoding="UTF-8" ?> <Customer> <Name>John...
```

4. Invoke the AS_SHRED_BATCH stored procedure with the following arguments:

- The SQL query you defined in step 3
- The annotated schema from step 1

The annotated schema now available in the SHREDMAPPING table as a result of step (1) needs to be made available to the shredding stored procedure. One way to do this is to write a wrapper stored procedure:

```
REPLACE PROCEDURE MY_SHREDBATCH_SP
(
  IN sourceDataQuery varchar(6000),
  IN annotatedSchemaID varchar(32),
  OUT resultCode varchar(128)
)
SPMAIN:BEGIN
  DECLARE annotatedSchema XML;
  DECLARE inputCursor CURSOR FOR SELECT ANNOTATED_SCHEMA from SHREDMAPPING
WHERE SCHEMA_ID = :annotatedSchemaID;
  OPEN inputCursor;
  FETCH inputCursor INTO annotatedSchema;
  IF (SQLSTATE <> '02000') THEN
  BEGIN
    CALL TD_SYSEXML.AS_SHRED_BATCH(:sourceDataQuery, :annotatedSchema,
NULL, :resultCode);
```

```

END;
ELSE
SET resultCode = -1;
END IF;
CLOSE inputCursor;
END SPMAIN;

```

This stored procedure takes two parameters:

- a. An SQL query that retrieves all the documents to be shredded
- b. An ID for the annotated schema (the key for the SHREDMAPPING table)

In the stored procedure, we first retrieve the annotated schema and then use it in the call to the AS_SHRED_BATCH stored procedure to shred the documents.

AS_SHRED_BATCH

This stored procedure is used for XML shredding (extracting values from XML documents to be used to populate the database).

The shredding procedure processes in a streaming fashion (the whole document is not loaded into memory all at once) each of the documents returned by evaluating the SQL query specified by *queryString*. The annotated schema is used to extract the values of interest from each of the documents and the target tables are updated using these values. During this process, if an *externalContext* is specified, any configuration values specified are used to override equivalent parameters in the annotated schema mapping.

Required Privileges

The AS_SHRED_BATCH stored procedure is created under the user TD_SYSEXML.

The user invoking this stored procedure must have the following privileges:

- EXECUTE PROCEDURE on TD_SYSEXML
- SELECT privilege on the staging table
- GRANT ALL (Insert/update/delete/upsert) on the target tables

AS_SHRED_BATCH Syntax

```

TD_SYSEXML.AS_SHRED_BATCH (
  queryString,
  annotatedSchema,
  externalContext,
  resultCode
)

```

Syntax Elements

queryString

An SQL query that returns 2 columns:

- An ID column
- A column that contains the XML document to be shredded as an XML data type

If *queryString* is NULL, the stored procedure returns a *resultCode* of 1 without performing any operations.

IN parameter of data type VARCHAR(20000), CHARACTER SET UNICODE.

annotatedSchema

An annotated schema that maps the XML document structure to the target tables that will be updated. If *annotatedSchema* is NULL or is not a valid mapping definition, an error is raised.

IN parameter of data type XML.

externalContext

Configuration values used to override equivalent parameters specified in the annotated schema mapping. This must be a comma separated list of *name=value* pairs and the names must match one of the configuration parameters in the annotated schema.

Four parameter values can be set through the *externalContext*:

- defaultDatabase
- defaultEncoding
- rootElement
- errorTable

For explanations of the defaultDatabase, defaultEncoding, and rootElement options, see the description of the corresponding elements in the discussion of [Adding Schema Annotations](#)

The errorTable option allows you to specify the fully qualified name of a table to which error information will be written if any of the documents fails shredding. Specifying this option activates a more permissive form of shredding in which failure of any individual document during shredding does not result in the abortion of the entire shredding procedure. Instead, the remaining valid documents in the batch are shredded, and the error information for the documents that failed shredding is written to an error table.

IN parameter of data type VARCHAR(512), CHARACTER SET UNICODE.

resultCode

A return code where:

- 0 = success
- 1 = failure

OUT parameter of data type INTEGER.

Error Handling

In batch mode shredding, by default, if an error occurs while an individual document is being processed, the batch is aborted and the stored procedure returns an error message indicating the type of error that occurred.

In some cases, the user may prefer to continue processing the remaining documents in the batch, and write the errors to a specified error table for later triage and resolution. This permissive mode of shredding can be invoked by passing in the `errorTable` option as part of the `externalContext` parameter to the batch shredding stored procedure.

The `externalContext` parameter is a comma separated list of *name=value* pairs. To invoke permissive error handling, users can specify `errorTable` as one of the names in the `externalContext`, with a table name as its value. The table name can be qualified with its database name. If not qualified, the table should reside under the user invoking the shredding stored procedure. While shredding a batch of documents, if any individual document fails to be shredded, its document ID and an error message will be written to the error table.

The error table should be a multiset table, and it should have the following two columns:

```
DOCID  VARCHAR(128),
ERRMSG VARCHAR(512),
```

Additional columns can be defined, but these columns should either be nullable or have a default value defined. Teradata recommends adding a column that will help with identifying the time when the error occurred. For example:

```
TS      TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

An error is raised if any of the following is true:

- The error table does not exist.
- The error table does not have the `docid` and `errmsg` columns.
- The user invoking the shredding procedure does not have insert privileges on the error table.

XML batch shredding is done in two phases. In the first phase, all the documents in the batch are shredded to a volatile table, and in the second phase, data is transferred from the volatile table to the target tables. During the first phase, if any document fails to shred for any reason, the document ID and the error message corresponding to the failure are written to the error table. This allows the stored procedure to continue shredding the remaining documents in the batch. If, however, an error occurs in the second phase (while moving data from the volatile table to the target tables), the shredding procedure will abort without writing to any of the target tables.

Related Information

See [Example: Shredding an XML Document](#).

XML Shredding Based on a Stylesheet

XML shredding is the process of extracting values from XML documents to populate tables in the database. One way to define the mapping from the XML document's tree structure to the database's tabular structure is to use an annotated schema. An alternative is to use an XSLT stylesheet-based mapping.

For efficient memory use, streamed shredding does not load the entire document into memory. XSLT-based shredding loads the document in memory, giving it more flexibility than one-pass processing over the document.

XSLT-based shredding is accomplished using shredding stored procedures:

- XSLT_SHRED - used for single XML document shredding
- XSLT_SHRED_BATCH - used to shred multiple XML documents

Perform these typical tasks to shred XML documents.

1. Load the XML document to be shredded into an XML column of a staging table.
The XML document must conform to W3C XML standards. The following is an example of such an XML document.

```
<<?xml version="1.0"?>
<Root>
  <predictixOfferMessage>
    <offer>
      <offerId>1000002</offerId>
    </offer>
    <mediaBlock>
      <mediaBlockId>90000000010002</mediaBlockId>
    </mediaBlock>
  </predictixOfferMessage>
</Root>
```

2. Define an SQL query that returns a result set with two columns:
 - An ID column
 - The XML column that contains the XML document to be shredded
3. Create an XSLT stylesheet that defines how the XML document is shred to the target tables.
4. Store the XSLT stylesheet so that it can be easily referenced; for example, store it in a stylesheet repository table.
5. Call the XSLT_SHRED_BATCH stored procedure with the following arguments:
 - The SQL query you defined in step 2

- The XSLT stylesheet you saved in step 4

The XML documents returned by the SQL query are shredded based on the mapping in the XSLT stylesheet. This results in the target tables being populated with data from these documents. NULL is inserted into target columns in some cases (for example, where the corresponding elements in the XML are empty or missing). NULL is stored if the given element path is missing in the input XML document.

XSLT_SHRED_BATCH

Use the XSLT_SHRED_BATCH stored procedure for XML shredding (extracting values from XML documents to be used to populate the database), and is based on a mapping provided in an XSLT stylesheet. This stored procedure is used for batch mode shredding, where a number of XML documents are shredded in a single call to the stored procedure.

Required Privileges

The XSLT_SHRED_BATCH stored procedure is created under the user TD_SYSEXML.

The user invoking this stored procedure must have the following privileges:

- EXECUTE PROCEDURE on TD_SYSEXML
- SELECT privilege on the staging table
- GRANT ALL (Insert/update/delete/upsert) on the target tables

XSLT_SHRED_BATCH Syntax

```
TD_SYSEXML.XSLT_SHRED_BATCH (
    queryString,
    xsltMapping,
    externalContext,
    resultCode
)
```

Syntax Elements

queryString

The SQL query string that is evaluated and returns a result set with two columns: an ID column and a column with the XML documents to be shredded.

IN parameter of data type VARCHAR(20000), CHARACTER SET UNICODE.

An error is reported if this parameter is NULL.

xsltMapping

An XSLT stylesheet provides the mapping from the XML elements to be shredded to the target table columns.

IN parameter of data type XML.

externalContext

A comma separated list of *name=value* pairs that parameterize the shredding operation. These values override configuration parameters, such as defaultDatabase, that are specified in the stylesheet mapping.

IN parameter of data type VARCHAR(512), CHARACTER SET UNICODE.

The *externalContext* parameter specifies configuration values used to override equivalent parameters specified in the XSLT stylesheet mapping. This must be a comma separated list of *name=value* pairs and the names must match one of the configuration parameters in the XSLT stylesheet.

Currently four parameter values can be set through the *externalContext*:

- defaultDatabase
- defaultEncoding
- rootElement
- errorTable

For explanations of the defaultDatabase, defaultEncoding, and rootElement options, see the description of the corresponding elements in the discussion of [Adding Schema Annotations](#).

The errorTable option allows you to specify the fully qualified name of a table to which error information will be written if any of the documents fails shredding. Specifying this option activates a more permissive form of shredding in which failure of any individual document during shredding does not result in the abortion of the entire shredding procedure. Instead, the remaining valid documents in the batch are shredded, and the error information for the documents that failed shredding is written to an error table.

resultCode

A return code where:

- 0 = success
- 1 = failure

OUT parameter of data type INTEGER.

Error Handling

In batch mode shredding, by default, if an error occurs while an individual document is being processed, the batch is aborted and the stored procedure returns an error message indicating the type of error that occurred.

In some cases, the user may prefer to continue processing the remaining documents in the batch, and write the errors to a specified error table for later triage and resolution. This permissive mode of shredding can be invoked by passing in the `errorTable` option as part of the `externalContext` parameter to the batch shredding stored procedure.

The `externalContext` parameter is a comma separated list of *name=value* pairs. To invoke permissive error handling, users can specify `errorTable` as one of the names in the `externalContext`, with a table name as its value. The table name can be qualified with its database name. If not qualified, the table should reside under the user invoking the shredding stored procedure. While shredding a batch of documents, if any individual document fails to be shredded, its document ID and an error message will be written to the error table.

The error table should be a multiset table, and it should have the following two columns:

```
DOCID  VARCHAR(128),
ERRMSG VARCHAR(512),
```

Additional columns can be defined, but these columns should either be nullable or have a default value defined. Teradata recommends adding a column that will help with identifying the time when the error occurred. For example:

```
TS      TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

An error is raised if any of the following is true:

- The error table does not exist.
- The error table does not have the `docid` and `errmsg` columns.
- The user invoking the shredding procedure does not have insert privileges on the error table.

XML batch shredding is done in two phases. In the first phase, all the documents in the batch are shredded to a volatile table, and in the second phase, data is transferred from the volatile table to the target tables. During the first phase, if any document fails to shred for any reason, the document ID and the error message corresponding to the failure are written to the error table. This allows the stored procedure to continue shredding the remaining documents in the batch. If, however, an error occurs in the second phase (while moving data from the volatile table to the target tables), the shredding procedure will abort without writing to any of the target tables.

XSLT_SHRED_BATCH Examples

Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples

Set up the users and tables used in subsequent `XSLT_SHRED_BATCH` and `XSLT_SHRED` examples.

- Create the users.
- Create the source tables to store the XML documents. A source table must have at least two columns: an ID column and an XML document column.

- Create target tables to store the XML values.

```
CREATE USER xsltuser AS PASSWORD = xsltuser PERM = 2000000*(HashAMP()+1);
GRANT ALL ON xsltuser TO xsltuser;
GRANT ALL ON TD_SYSEXML TO xsltuser;
GRANT EXECUTE PROCEDURE ON TD_SYSEXML TO xsltuser;
.logoff

.logon ie1510/xsltuser,xsltuser
```

```
CREATE SET TABLE Offer(
    offerid char(10),
    mediaBlockid varchar(64),
    datetimecol varchar(30)
);
```

```
CREATE TABLE Offer1(C_COL1 char(10), C_COL2 varchar(64));
```

```
CREATE TABLE Input_Docs(id INT, xmldoc XML);
```

```
CREATE SET TABLE TransientTbl(C_Id int, C_Name varchar(64), C_Sal Int);
CREATE SET TABLE TransientTbl_docs(id int, xmlcol xml);
```

```
CREATE TABLE DefaultValue
(
    datec DATE,
    timec TIME(6) ,
    timewzc TIME(6) WITH TIME ZONE,
    timestampc TIMESTAMP(6),
    timestampwzc TIMESTAMP(6) WITH TIME ZONE
);
```

```
CREATE TABLE DefaultValue2
(
    byteintc ByteInt,
    smallintc Smallint,
    intc Integer,
    floatc Float
);
```

```
CREATE SET TABLE DefaultValue3 ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL,
    CHECKSUM = DEFAULT,
```

```

DEFAULT MERGEBLOCKRATIO
(
  C_Id INTEGER,
  C_Byteintc BYTEINT,
  C_Smallintc SMALLINT,
  C_Intc INTEGER)
PRIMARY INDEX ( C_Id );

```

```

CREATE TABLE DefaultValue4
(
  C_Datec DATE,
  C_Timec TIME(6),
  C_Timewzc TIME(6) WITH TIME ZONE,
  C_TimeStampc TIMESTAMP(6),
  C_TimeStampwzc TIMESTAMP(6) WITH TIME ZONE
);

```

```

CREATE TABLE Customer
(
  c_custkey BIGINT NOT NULL,
  c_name VARCHAR(25) CHARACTER SET LATIN,
  c_address VARCHAR(40) CHARACTER SET LATIN,
  c_nationkey BIGINT,
  c_phone CHAR(15) CHARACTER SET LATIN,
  c_acctbal DECIMAL(20,2),
  c_mktsegment CHAR(10) CHARACTER SET LATIN,
  c_comment VARCHAR(117) CHARACTER SET LATIN
) UNIQUE PRIMARY INDEX PK_CUSTKEY (c_custkey);

```

```

CREATE TABLE Orders
(
  o_orderkey BIGINT NOT NULL,
  o_custkey BIGINT,
  o_orderstatus CHAR(1) CHARACTER SET LATIN,
  o_totalprice DECIMAL(20,2),
  o_orderdate DATE FORMAT 'YY/MM/DD',
  o_orderpriority CHAR(15) CHARACTER SET LATIN,
  o_clerk CHAR(15) CHARACTER SET LATIN,
  o_shippriority BIGINT,
  o_comment VARCHAR(79) CHARACTER SET LATIN
) UNIQUE PRIMARY INDEX PK_ORDERKEY ( o_orderkey );

```

```
CREATE TABLE Dealer
(
    d_id BIGINT NOT NULL,
    d_name CHAR(15) CHARACTER SET LATIN,
    d_address CHAR(15) CHARACTER SET LATIN,
    d_comment VARCHAR(79) CHARACTER SET LATIN
) UNIQUE PRIMARY INDEX PK_ORDERKEY (d_id);
```

Example: XSLT_SHRED_BATCH <xsl:copy-of select="">

This example shows how to use the <xsl:copy-of select=""> mapping definition. In <xsl:copy-of select=""> mappings the column and element names match.

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#) for details.

Populate the staging table used to store the XML source document.

```
DELETE Offer;
DELETE Input_Docs;

INSERT INTO input_docs VALUES(1,
    CREATEXML('<?xml version="1.0"?>
        <Root>
            <predictixOfferMessage>
                <Offer><offerid>1000001</offerid></Offer>
                <mediaBlock>
                    <mediaBlockid>90000000010001</mediaBlockid>
                    <parentofferid>1001</parentofferid>
                </mediaBlock>
            </predictixOfferMessage>
            <predictixOfferMessage>
                <Offer><offerid>1000002</offerid></Offer>
                <mediaBlock>
                    <mediaBlockid>90000000010002</mediaBlockid>
                    <parentofferid>1002</parentofferid>
                </mediaBlock>
            </predictixOfferMessage>
        </Root>'));
```

Call the XSLT_SHRED_BATCH stored procedure with the stylesheet mapping to shred the XML data stored in the staging input table. The following arguments are used for the call.

- The *queryString* is `SEL id, xmldoc FROM xsltuser.Input_Docs`.
- The *xsltMapping* argument is supplied by invoking the `CREATEXML` function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* is `NULL` for this example.
- The *resultCode* is returned in `:res`. A successful operation returns 0.

```
CALL TD_SYSXML.XSLT_SHRED_BATCH('SEL id, xmldoc FROM xsltuser.Input_Docs',
createxml('<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/Root">
<Transaction>
<Insert>
<Table>
<xsltuser.Offer>
<xsl:for-each select="predictixOfferMessage">
<Row>
<xsl:copy-of select="Offer/offerid"/>
<xsl:copy-of select="mediaBlock/mediaBlockid"/>
</Row>
</xsl:for-each>
</xsltuser.Offer>
</Table>
</Insert>
</Transaction>
</xsl:template>
</xsl:stylesheet>'), NULL,:res);
```

Result: To view the updated data in the target table, run: `SELECT offerid, mediaBlockid FROM Offer;`

offerid	mediaBlockid
1000001	900000000010001
1000002	900000000010002

Example: XSLT_SHRED_BATCH <xsl:value-of select="">

This example shows how to use the `<xsl:value-of select="">` mapping definition. In `<xsl:value-of select="">` mappings the column and element names are different.

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#) for details.

Populate the staging table used to store the XML source document.

```
DELETE Input_Docs;

INSERT INTO Input_Docs VALUES(1, CREATEXML('<?xml version="1.0"?>
<Root>
  <predictixOfferMessage>
    <Offer><offerid>1000003</offerid></Offer>
    <mediaBlock><mediaBlockid>90000000010003</mediaBlockid></mediaBlock>
  </predictixOfferMessage>
</Root>' ));
```

Call the XSLT_SHRED_BATCH stored procedure with the stylesheet mapping to shred the XML data stored in the staging input table. The following arguments are used for the call.

- The *queryString* is: `sel * from xsltuser.Input_Docs`
- The *xsltMapping* argument is supplied by invoking the CREATEXML function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* is NULL for this example.
- The *resultCode* is returned in `:res`. A successful operation returns 0.

```
CALL TD_SYSEXML.XSLT_SHRED_BATCH('sel * from xsltuser.Input_Docs', CREATEXML('<?
xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/Root">
<Transaction>
  <Insert>
    <Table>
      <xsltuser.Offer1>
        <xsl:for-each select="predictixOfferMessage">
          <Row>
            <C_COL1><xsl:value-of select="Offer/offerid"/></C_COL1>
            <C_COL2><xsl:value-of select="mediaBlock/mediaBlockid"/></C_COL2>
          </Row>
        </xsl:for-each>
      </xsltuser.Offer1>
    </Table>
  </Insert>
</Transaction>
</xsl:template>
</xsl:stylesheet>
'), NULL, :res);
```

Result: To view the updated data in the target table, run: `SELECT * FROM Offer1;`

C_COL1	C_COL2
-----	-----
1000003	900000000010003

Example: XSLT_SHRED_BATCH Combined <xsl:copy-of select=""> and <xsl:value-of select="">

These examples show how to use both the <xsl:copy-of select=""> and <xsl:value-of select=""> mapping definitions.

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#) for details.

Populate the staging table used to store the XML source document.

```
DELETE Offer;
DELETE Input_Docs;

INSERT INTO Input_Docs VALUES(1, CREATEXML('<?xml version="1.0"?>
<Root>
  <predictixOfferMessage>
    <Offer><offerid>1000001</offerid></Offer>
    <mediaBlock><mediaBlockid>900000000010001</mediaBlockid></mediaBlock>
  </predictixOfferMessage>
  <predictixOfferMessage>
    <Offer><offerid>1000003</offerid></Offer>
    <mediaBlock><mediaBlockid>900000000010003</mediaBlockid></mediaBlock>
  </predictixOfferMessage>
</Root>'));
```

Call the XSLT_SHRED_BATCH stored procedure with the stylesheet mapping to shred the XML data stored in the staging input table. The following arguments are used for the call.

- The *queryString* is: `sel * from xsltuser.Input_Docs`
- The *xsltMapping* argument is supplied by invoking the CREATEXML function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* is NULL for this example.
- The *resultCode* is returned in :res. A successful operation returns 0.

```
CALL TD_SYSXML.XSLT_SHRED_BATCH('sel * from xsltuser.Input_Docs', CREATEXML('<?
xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/Root">
```

```

<Transaction>
  <Insert>
    <Table>
      <xsltuser.Offer>
        <xsl:for-each select="predictixOfferMessage">
          <Row>
            <xsl:copy-of select="Offer/offerid"/>
            <mediaBlockid><xsl:value-of select="mediaBlock/mediaBlockid"/></mediaBlockid>
          </Row>
        </xsl:for-each>
      </xsltuser.Offer>
    </Table>
  </Insert>
</Transaction>
</xsl:template>
</xsl:stylesheet>'), NULL, :res);

```

Result: To view the updated data in the target table, run: `SELECT offerid, mediaBlockid FROM Offer;`

offerid	mediaBlockid
1000003	900000000010003
1000001	900000000010001

This is another example on how to use both the `<xsl:copy-of select="">` and `<xsl:value-of select="">` mapping definitions.

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#) for details.

Populate the staging table used to store the XML source document.

```

DELETE Customer;
DELETE Input_Docs;

INSERT INTO Input_Docs VALUES(1,
  CREATEXML('<?xml version="1.0" encoding="UTF-8"?>
<customers>
<customer>
  <c_custkey>1805</c_custkey>
  <c_name>Customer#000001805</c_name>
  <c_address>ZERs4Cu5lQTYD</c_address>

```

```

    <c_nationkey>9</c_nationkey>
    <c_phone>19-679-706-1096</c_phone>
    <c_acctbal>-274.75</c_acctbal>
    <c_mktsegment>AUTOMOBILE</c_mktsegment>
    <c_comment>quickly unusual courts alongside of the requests</c_comment>
  </customer>
<customer>
  <c_custkey>1806</c_custkey>
  <c_name>Customer#000001806</c_name>
  <c_address>BB6Vr7W rSIpWKp</c_address>
  <c_nationkey>9</c_nationkey>
  <c_phone>19-872-322-3433</c_phone>
  <c_acctbal>254.17</c_acctbal>
  <c_mktsegment>MACHINERY</c_mktsegment>
  <c_comment>ideas are blithely. ironic instructions wake quickly.</c_comment>
</customer>
</customers>')));

```

Call the XSLT_SHRED_BATCH stored procedure with the stylesheet mapping to shred the XML data stored in the staging input table. The following arguments are used for the call.

- The *queryString* is: `sel id, xmldoc from xsltuser.Input_Docs`
- The *xsltMapping* argument is supplied by invoking the CREATEXML function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* is NULL for this example.
- The *resultCode* is returned in `:res`. A successful operation returns 0.

```

CALL TD_SYSXML.XSLT_SHRED_BATCH('SEL id, xmldoc FROM xsltuser.Input_Docs',
CREATEXML('<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/customers">
  <Transaction>
    <Insert>
      <Table>
        <xsltuser.customer>
          <xsl:for-each select="customer">
            <Row>
              <c_custkey genexp="cast(? as integer)">
                <xsl:value-of select="c_custkey"/>
              </c_custkey>
              <xsl:copy-of select="c_name"/>
              <xsl:copy-of select="c_address"/>
              <c_nationkey genexp="cast(? as integer)">
                <xsl:value-of select="c_nationkey"/>

```

```

    </c_nationkey>
    <xsl:copy-of select="c_phone"/>
    <c_acctbal genexp="cast(? as numeric(15,2))">
      <xsl:value-of select="c_acctbal"/>
    </c_acctbal>
    <xsl:copy-of select="c_mktsegment"/>
    <xsl:copy-of select="c_comment"/>
  </Row>
</xsl:for-each>
</xsltuser.customer>
</Table>
</Insert>
</Transaction>
</xsl:template>
</xsl:stylesheet>'), NULL,:res);

```

Result: To view the updated data in the target table, run: `SELECT * FROM customer;`

c_custkey	c_name	c_address	c_nationkey	c_phone
1805	Customer#000001805	ZERs4Cu5lQTYD	9	19-679-706-1096
-274.75	AUTOMOBILE	quickly unusual courts		
1806	Customer#000001806	BB6Vr7W rSIpWKp	9	
19-872-322-3433	254.17	MACHINERY	ideas are blithely	

Example: XSLT_SHRED_BATCH with a Transient SQL Expression

This example shows how to use the `<xsl:value-of select="">` mapping definition with a transient column. The value in the transient column is used to compute the value of a column in the target table. Any valid SQL expression can be used to do this computation, including user-defined functions when highly customized computations are required. The associated data item itself will not be inserted into the target table.

The following syntax shows the `C_Name_Transient` column being used in the computation of the `C_Name` column.

Note:

All transient columns must have an `sqltype` attribute with a valid SQL data type because this column does not exist in the target table.

```

<Row>
  <C_Id><xsl:value-of select="/Id " /></C_Id>
  <C_Name_Transient sqltype="varchar(30)" transient="true">
    <xsl:value-of select=" ..../NameTransent " />
  </ C_Name_Transient >
  < C_Name sqlexpr="true"><![CDATA[ 'HYD_' || C_Name_Transient]]>
  < C_Name >
    .
    .
    .
</Row>

```

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#) for details.

Populate the staging table used to store the XML source document.

```

INSERT INTO TransientTbl_Docs values(1, CREATEXML('<?xml version="1.0"?>
<Customers>
  <Customer>
    <C_Id>100</C_Id>
    <C_Name>EMP_ABCD</C_Name>
    <C_Sal_Transient>1200</C_Sal_Transient>
  </Customer>
  <Customer>
    <C_Id>200</C_Id>
    <C_Name>EMP_XYZ</C_Name>
    <C_Sal_Transient>1300</C_Sal_Transient>
  </Customer>
</Customers>'));

```

Call the XSLT_SHRED_BATCH stored procedure with the stylesheet mapping to shred the XML data stored in the staging input table. The following arguments are used for the call.

- The *queryString* is `SEL * FROM xsltuser.TransientTbl_Docs`.
- The *xsltMapping* argument is supplied by invoking the CREATEXML function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* is NULL for this example.
- The *resultCode* is returned in `:res`. A successful operation returns 0.

```

CALL TD_SYSXML.XSLT_SHRED_BATCH('sel * from xsltuser.TransientTbl_Docs',
CREATEXML('<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

```

```

<xsl:template match="/Customers">
  <Transaction>
    <Insert>
      <Table>
        <xsltuser.TransientTbl>
          <xsl:for-each select="Customer">
            <Row>
              <xsl:copy-of select="C_Id"/>
              <xsl:copy-of select="C_Name"/>
              <C_Sal_Transient sqltype="Int" transient="true">
                <xsl:value-of select="C_Sal_Transient" />
              </C_Sal_Transient>
              <C_Sal sqlexpr="true">100 * C_Sal_Transient</C_Sal>
            </Row>
          </xsl:for-each>
        </xsltuser.TransientTbl>
      </Table>
    </Insert>
  </Transaction>
</xsl:template>
</xsl:stylesheet>'), NULL, :res);

```

Result: To view the updated data in the target table, run: `SEL * FROM TransientTbl;`

C_Id	C_Name	C_Sal
200	EMP_XYZ	130000
100	EMP_ABCD	120000

Example: XSLT_SHRED_BATCH defaultDatabase

A common database can be used for all the tables shredded with a given mapping. This example shows how to use the *defaultDatabase* option.

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#) for details.

Populate the staging table used to store the XML source document.

```

DELETE Input_Docs;
DELETE DefaultValue2;

INSERT INTO Input_Docs VALUES(1, NEW XML('<Root>

```

```

<AllTypes>
  <Numeric>
    <Byteintc>1</Byteintc>
    <Smallintc>100</Smallintc>
    <Floatc>100</Floatc>
    <Decimalc>100</Decimalc>
    <Numberc>1100</Numberc>
  </Numeric>
</AllTypes>
<AllTypes>
  <Numeric>
    <Byteintc>2</Byteintc>
    <Smallintc>200</Smallintc>
    <Intc>200</Intc>
    <Floatc>200</Floatc>
    <Decimalc>200</Decimalc>
    <Numberc>1200</Numberc>
  </Numeric>
</AllTypes></Root>');

```

Call the XSLT_SHRED_BATCH stored procedure with the stylesheet mapping to shred the XML data stored in the staging input table. The following arguments are used for the call.

- The *queryString* is `sel * from xsltuser.Input_Docs`.
- The *xsltMapping* argument is supplied by invoking the CREATEXML function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* is NULL for this example.
- The *resultCode* is returned in `:res`. A successful operation returns 0.

```

CALL TD_SYSXML.XSLT_SHRED_BATCH('sel * from xsltuser.Input_Docs', CREATEXML('<?
xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/Root">
<defaultDatabase>xsltuser</defaultDatabase>
<Transaction>
  <Insert>
    <Table>
      <dummyuser.DefaultValue2>
        <xsl:for-each select="AllTypes">
          <Row>
            <byteintc genexp="cast(? as byteint)">
              <xsl:value-of select="Numeric/Byteintc"/>
            </byteintc>
            <smallintc><xsl:value-of select="Numeric/Smallintc"/></smallintc>

```



```

    <intc default="120"><xsl:value-of select="Numeric/Intc"/></intc>
    <floatc><xsl:value-of select="Numeric/Floatc"/></floatc>
  </Row>
</xsl:for-each>
</dummyuser.DefaultValue2>
</Table>
</Insert>
</Transaction>
</xsl:template>
</xsl:stylesheet>'), ',:res);

```

Result: To view the updated data in the target table, run: `SELECT * FROM DefaultValue2;`

byteintc	smallintc	intc	floatc
1	100	120	1.00000000000000E 002
2	200	200	2.00000000000000E 002

Examples: XSLT_SHRED_BATCH Using a Context Parameter

These examples show how a context parameter can be passed to the shredding operation. Currently *defaultDatabase* is the only supported context parameter.

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#) for details.

Populate the staging table used to store the XML source document.

```

DELETE Input_Docs;
DELETE DefaultValue2;

INSERT INTO Input_Docs VALUES(1, NEW XML('
<Root>
<AllTypes>
  <Numeric>
    <Byteintc>1</Byteintc>
    <Smallintc>100</Smallintc>
    <Floatc>100</Floatc>
    <Decimalc>100</Decimalc>
    <Numberc>1100</Numberc>
  </Numeric>
</AllTypes>
<AllTypes>

```

```

<Numeric>
  <Byteintc>2</Byteintc>
  <Smallintc>200</Smallintc>
  <Intc>200</Intc>
  <Floatc>200</Floatc>
  <Decimalc>200</Decimalc>
  <Numberc>1200</Numberc>
</Numeric>
</AllTypes>
</Root>')));

```

Call the XSLT_SHRED_BATCH stored procedure with the stylesheet mapping to shred the XML data stored in the staging input table. The following arguments are used for the call.

- The *queryString* is: `sel * from xsltuser.Input_Docs`
- The *xsltMapping* argument is supplied by invoking the CREATEXML function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* sets the defaultDatabase value.
- The *resultCode* is returned in :res. A successful operation returns 0.

Take note of the setting for *<defaultDatabase>*.

```

CALL TD_SYSEXML.XSLT_SHRED_BATCH('sel * from xsltuser.Input_Docs',
CREATEXML('<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/Root">
<defaultDatabase>dummyuser2</defaultDatabase>
<Transaction>
  <Insert>
    <Table>
      <dummyuser.DefaultValue2>
        <xsl:for-each select="AllTypes">
          <Row>
            <byteintc genexp="cast(? as byteint)">
              <xsl:value-of select="Numeric/Byteintc"/>
            </byteintc>
            <smallintc ><xsl:value-of select="Numeric/Smallintc"/></smallintc>
            <intc default="120"><xsl:value-of select="Numeric/Intc"/></intc>
            <floatc><xsl:value-of select="Numeric/Floatc"/></floatc>
          </Row>
        </xsl:for-each>
      </dummyuser.DefaultValue2>
    </Table>
  </Insert>

```

```

</Transaction>
</xsl:template>
</xsl:stylesheet>'), 'defaultDatabase=xsltuser',:res);

```

Result: To view the updated data in the target table, run: `SELECT * FROM DefaultValue2;`

byteintc	smallintc	intc	floatc
1	100	120	1.00000000000000E 002
2	200	200	2.00000000000000E 002

The following is another example using the context parameter.

Populate the staging table used to store the XML source document.

```

DELETE Input_Docs;
DELETE Offer;

INSERT INTO Input_Docs VALUES(1,
  createxml('<?xml version="1.0"?>
    <Root>
      <predictixOfferMessage>
        <Offer><offerid>1000001</offerid></Offer>
        <mediaBlock>
          <mediaBlockid>90000000010001</mediaBlockid>
          <parentofferid>1001</parentofferid>
        </mediaBlock>
      </predictixOfferMessage>
      <predictixOfferMessage>
        <Offer><offerid>1000002</offerid></Offer>
        <mediaBlock>
          <mediaBlockid>90000000010002</mediaBlockid>
          <parentofferid>1002</parentofferid>
        </mediaBlock>
      </predictixOfferMessage>
    </Root>')));

```

Call the `XSLT_SHRED_BATCH` stored procedure with the stylesheet mapping to shred the XML data stored in the staging input table. The following arguments are used for the call.

- The *queryString* is: `SEL id, xmldoc FROM xsltuser.Input_Docs`
- The *xsltMapping* argument is supplied by invoking the `CREATEXML` function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* sets the defaultDatabase to `xsltuser`.
- The *resultCode* is returned in `:res`. A successful operation returns 0.

Take note of the setting for `<defaultDatabase>`.

```
CALL TD_SYSXML.XSLT_SHRED_BATCH('SEL id, xmldoc FROM xsltuser.Input_Docs',
createxml('<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/Root">
<defaultDatabase>TestUser</defaultDatabase>
<Transaction>
  <Insert>
    <Table>
      <Offer>
        <xsl:for-each select="predictixOfferMessage">
          <Row>
            <xsl:copy-of select="Offer/offerid"/>
            <xsl:copy-of select="mediaBlock/mediaBlockid"/>
          </Row>
        </xsl:for-each>
      </Offer>
    </Table>
  </Insert>
</Transaction>
</xsl:template>
</xsl:stylesheet>'), 'defaultDatabase=XSLTUSER',:res);
```

Result: To view the updated data in the target table, run: `SELECT offerid, mediaBlockid FROM Offer;`

offerid	mediaBlockid
1000001	900000000010001
1000002	900000000010002

Example: XSLT_SHRED_BATCH `<xsl:param>`

This example shows how to use the `<xsl:param>` parameter to define context level constants.

The following shows the syntax for `<xsl:param>`.

```
<xsl:param name="XML_TimeStamp" select="date:date-time()"></xsl:param>
```

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#) for details.

Populate the staging table used to store the XML source document.

```
DELETE Input_Docs;
DELETE Offer;

INSERT INTO Input_Docs(1, CREATEXML('<?xml version="1.0"?>
<Root>
  <predictixOfferMessage>
    <Offer><offerid>1000001</offerid></Offer>
    <mediaBlock><mediaBlockid>90000000010001</mediaBlockid></mediaBlock>
  </predictixOfferMessage>
  <predictixOfferMessage>
    <Offer><offerid>1000002</offerid></Offer>
    <mediaBlock><mediaBlockid>90000000010002</mediaBlockid></mediaBlock>
  </predictixOfferMessage>
</Root>' ));
```

Call the XSLT_SHRED_BATCH stored procedure with the stylesheet mapping to shred the XML data stored in the staging input table. The following arguments are used for the call.

- The *queryString* is: `sel * from xsltuser.Input_Docs`
- The *xsltMapping* argument is supplied by invoking the CREATEXML function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* sets the defaultDatabase to xsltuser.
- The *resultCode* is returned in `:res`. A successful operation returns 0.

This example shows the `<xsl:param>` being set to a timestamp and then being used as a value for one of the columns in the target table.

```
CALL TD_SYSEXML.XSLT_SHRED_BATCH('sel * from xsltuser.Input_Docs', CREATEXML('<?
xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:date="http://exslt.org/dates-and-times" extension-element-
prefixes="date" version="1.0">
<xsl:param name="XML_TimeStamp" select="date:date-time()"></xsl:param>
<xsl:template match="/Root">
  <Transaction>
    <Insert>
      <Table>
        <Offer>
          <xsl:for-each select="predictixOfferMessage">
            <Row>
              <offerid> <xsl:value-of select="Offer/offerid"/></offerid>
              <mediaBlockid>
```

```

        <xsl:value-of select="mediaBlock/mediaBlockid"/>
    </mediaBlockid>
    <datetimecol default="{ $XML_TimeStamp }">
        <xsl:value-of select="Offer/dummysElement"/>
    </datetimecol>
</Row>
</xsl:for-each>
</Offer>
</Table>
</Insert>
</Transaction>
</xsl:template>
</xsl:stylesheet>'), 'defaultDatabase=xsltuser', :res);

```

Result: To view the updated data in the target table, run: `SELECT * FROM Offer;`

offerid	mediaBlockid	datetimecol
1000001	90000000010001	2014-11-04T15:22:55-08:00
1000002	90000000010002	2014-11-04T15:22:55-08:00

Example: XSLT_SHRED_BATCH Default Value for an Absent or Empty Column Value

A column element with `<xsl:value-of>` can have attributes named `default`. If `default` is specified its value will be inserted for Absent/Empty elements. Otherwise Teradata NULL will returned.

The following shows the syntax for using the default value.

```

<Row>
    <C_Name default="Teradata Employee" >
        <xsl:value-of select=" ...../AbsentOREmptyElement" />
    </C_Name >
</Row>

```

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#) for details.

Populate the staging table used to store the XML source document.

```

DELETE Input_Docs;
DELETE DefaultValue;

INSERT INTO Input_Docs(1,NEW XML('

```

```

<Root>
  <AllTypes>
    <DateTime>
      <Datec>2000-09-09</Datec>
      <Timec>12:12:12</Timec>
      <Timewzc>12:12:12+00:00</Timewzc>
      <TimeStampc>2000-09-09T12:12:12</TimeStampc>
      <TimeStampwzc>2000-09-09T12:12:12+00:00</TimeStampwzc>
    </DateTime>
  </AllTypes>
  <AllTypes>
    <DateTime>
      <Datec>3000-09-09</Datec>
      <Timec>12:12:12</Timec>
      <Timewzc>12:12:12+00:00</Timewzc>
      <TimeStampc>3000-09-09T12:12:12</TimeStampc>
    </DateTime>
  </AllTypes>
</Root>');

```

Call the XSLT_SHRED_BATCH stored procedure with the stylesheet mapping to shred the XML data stored in the staging input table. The following arguments are used for the call.

- The *queryString* is: `SELECT id, xmldoc FROM xsltuser.Input_Docs`
- The *xsltMapping* argument is supplied by invoking the CREATEXML function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* is NULL for this example.
- The *resultCode* is returned in :res. A successful operation returns 0.

```

CALL TD_SYSXML.XSLT_SHRED_BATCH('SELECT id, xmldoc FROM xsltuser.Input_Docs',
  CREATEXML('<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:date="http://exslt.org/dates-and-times"
extension-element-prefixes="date" version="1.0" >
<xsl:param name="XML_TimeStamp" select="date:date-time()"/>
<xsl:template match="/Root">
  <Transaction>
    <Insert>
      <Table>
        <xsltuser.DefaultValue>
          <xsl:for-each select="AllTypes">
            <Row>
              <datec><xsl:value-of select="DateTime/Datec"/></datec>
              <timec><xsl:value-of select="DateTime/Timec"/></timec>

```

```

    <timewzc><xsl:value-of select="DateTime/Timewzc"/></timewzc>
    <timestampc>
      <xsl:value-of select="DateTime/TimeStampc"/>
    </timestampc>
    <timestampwzc default="{ $XML_TimeStamp }">
      <xsl:value-of select="DateTime/TimeStampwzc"/>
    </timestampwzc>
  </Row>
</xsl:for-each>
</xsltuser.DefaultValue>
</Table>
</Insert>
</Transaction>
</xsl:template>
</xsl:stylesheet>'), '',:res);

```

Result: To view the updated data in the target table, run: `SELECT * FROM DefaultValue;`

Note:

The last timestamp under the timestampwzc column is the default value.

datec	timec	timewzc	timestampc	timestampwzc
2000/09/09	04:12:12. 000000	12:12:12. 000000+00:00	2000-09-09 05:12: 12.000000	2000-09-09 12:12:12. 000000+00:00
3000/09/09	04:12:12. 000000	12:12:12. 000000+00:00	3000-09-09 05:12: 12.000000	2014-11-04 15:22:56. 000000-08:00

Example: XSLT_SHRED_BATCH Multiple Operations

Use XSLT_SHRED_BATCH in multiple operations: Insert, Update, Delete, and Upsert .

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#) for details.

Populate tables for use in the XSLT_SHRED_BATCH multiple operations example.

```

DELETE Customer;
DELETE Orders;
DELETE Dealer;
DELETE Input_Docs;

```



```

INSERT INTO Orders(o_orderkey) values (12610);

INSERT INTO Orders(o_orderkey) values (12611);

INSERT INTO Dealer(d_id) values (12610);

INSERT INTO Input_Docs VALUES(1, CREATEXML('<?xml version="1.0"
encoding="UTF-8"?>
<root>
<customers>
  <customer>
    <c_custkey>1805</c_custkey>
    <c_name>Customer#000001805</c_name>
    <c_address>ZERs4Cu5lQTYD</c_address>
    <c_nationkey>9</c_nationkey>
    <c_phone>19-679-706-1096</c_phone>
    <c_acctbal>-274.75</c_acctbal>
    <c_mktsegment>AUTOMOBILE</c_mktsegment>
    <c_comment>quickly unusual courts</c_comment>
  </customer>
  <customer>
    <c_custkey>1806</c_custkey>
    <c_name>Customer#000001806</c_name>
    <c_address>BB6Vr7W rSIpWKp</c_address>
    <c_nationkey>9</c_nationkey>
    <c_phone>19-872-322-3433</c_phone>
    <c_acctbal>254.17</c_acctbal>
    <c_mktsegment>MACHINERY</c_mktsegment>
    <c_comment>ideas are blithely</c_comment>
  </customer>
</customers>
<orders>
  <order>
    <o_orderkey>12610</o_orderkey>
    <o_custkey>4228</o_custkey>
    <o_orderstatus>F</o_orderstatus>
    <o_totalprice>9096.92</o_totalprice>
    <o_orderdate>1900-01-17T00:00:00.0Z</o_orderdate>
    <o_orderpriority>5/LOW</o_orderpriority>
    <o_clerk>Clerk#000000952</o_clerk>
    <o_shippriority>0</o_shippriority>
    <o_comment>ironic requests are furiously</o_comment>
  </order>
</order>

```

```

    <o_orderkey>12611</o_orderkey>
    <o_custkey>2050</o_custkey>
    <o_orderstatus>0</o_orderstatus>
    <o_totalprice>224970.76</o_totalprice>
    <o_orderdate>1900-01-13T00:00:00.0Z</o_orderdate>
    <o_orderpriority>3/MEDIUM</o_orderpriority>
    <o_clerk>Clerk#000000579</o_clerk>
    <o_shippriority>0</o_shippriority>
    <o_comment>fluffily ironic asympto</o_comment>
  </order>
  <order>
    <o_orderkey>12612</o_orderkey>
    <o_custkey>72163</o_custkey>
    <o_orderstatus>F</o_orderstatus>
    <o_totalprice>197724.39</o_totalprice>
    <o_orderdate>1900-01-08T00:00:00.0Z</o_orderdate>
    <o_orderpriority>5/LOW</o_orderpriority>
    <o_clerk>Clerk#000000381</o_clerk>
    <o_shippriority>0</o_shippriority>
    <o_comment>furious, regular deposits</o_comment>
  </order>
</orders>
<dealers>
  <dealer>
    <d_id>12610</d_id>
    <d_name>VBIT</d_name>
    <d_address>HYD</d_address>
    <d_comment>AP Based dealer</d_comment>
  </dealer>
  <dealer>
    <d_id>12611</d_id>
    <d_name>VBIT</d_name>
    <d_address>HYD</d_address>
    <d_comment>AP Based dealer</d_comment>
  </dealer>
</dealers>
</root>')));

```

Run XSLT_SHRED_BATCH with Insert, Update, Delete, and Upsert operations.

```

CALL TD_SYSXML.XSLT_SHRED_BATCH('SEL id, xmldoc FROM xsltuser.Input_Docs',
CREATEXML('<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/root">

```

```

<Transaction>
  <Insert>
    <Table>
      <xsltuser.customer pkey="c_custkey">
        <xsl:for-each select="customers/customer">
          <Row>
            <c_custkey>
              <xsl:value-of select="c_custkey"/>
            </c_custkey>
            <c_name>
              <xsl:value-of select="c_name"/>
            </c_name>
            <c_address>
              <xsl:value-of select="c_address"/>
            </c_address>
            <c_nationkey>
              <xsl:value-of select="c_nationkey"/>
            </c_nationkey>
            <c_phone>
              <xsl:value-of select="c_phone"/>
            </c_phone>
            <c_acctbal>
              <xsl:value-of select="c_acctbal"/>
            </c_acctbal>
            <c_mktsegment>
              <xsl:value-of select="c_mktsegment"/>
            </c_mktsegment>
            <c_comment>
              <xsl:value-of select="c_comment"/>
            </c_comment>
          </Row>
        </xsl:for-each>
      </xsltuser.customer>
    </Table>
  </Insert>
  <Update>
    <Table>
      <xsltuser.orders pkey="o_orderkey">
        <xsl:for-each select="orders/order">
          <Row>
            <o_orderkey>
              <xsl:value-of select="o_orderkey"/>
            </o_orderkey>
            <o_custkey>

```

```

        <xsl:value-of select="o_custkey"/>
    </o_custkey>
    <o_orderstatus>
        <xsl:value-of select="o_orderstatus"/>
    </o_orderstatus>
    <o_totalprice>
        <xsl:value-of select="o_totalprice"/>
    </o_totalprice>
    <o_orderdate genexp="cast(? as date)">
        <xsl:value-of
select="substring(o_orderdate,1,10)"/>
    </o_orderdate>
    <o_orderpriority>
        <xsl:value-of select="o_orderpriority"/>
    </o_orderpriority>
    <o_clerk>
        <xsl:value-of select="o_clerk"/>
    </o_clerk>
    <o_shippriority>
        <xsl:value-of select="o_shippriority"/>
    </o_shippriority>
    <o_comment>
        <xsl:value-of select="o_comment"/>
    </o_comment>
</Row>
</xsl:for-each>
</xsltuser.orders>
</Table>
</Update>

<Delete>
    <Table>
        <xsltuser.dealer pkey="d_id">
            <xsl:for-each select="dealers/dealer">
                <Row>
                    <d_id><xsl:value-of select="d_id"/></d_id>
                    <d_name>
                        <xsl:value-of select="d_name"/>
                    </d_name>
                </Row>
            </xsl:for-each>
        </xsltuser.dealer>
    </Table>
</Delete>

```

```

<Upsert>
  <Table>
    <xsltuser.dealer pkey="d_id">
      <xsl:for-each select="dealers/dealer">
        <Row>
          <d_id><xsl:value-of select="d_id"/></d_id>
          <d_name>
            <xsl:value-of select="d_name"/>
          </d_name>
          <d_address>
            <xsl:value-of select="d_address"/>
          </d_address>
          <d_comment>
            <xsl:value-of select="d_comment"/>
          </d_comment>
        </Row>
      </xsl:for-each>
    </xsltuser.dealer>
  </Table>
</Upsert>
</Transaction>
</xsl:template>
</xsl:stylesheet>'), NULL,:res);

```

Result: To view the updated data in the target table, run:

```
SEL c_custkey, c_name, c_phone, c_acctbal, c_mktsegment FROM Customer;
```

c_custkey	c_name	c_phone	c_acctbal	c_mktsegment
1805	Customer#000001805	19-679-706-1096	-274.75	AUTOMOBILE
1806	Customer#000001806	19-872-322-3433	254.17	MACHINERY

Result: To view the updated data in the target table, run:

```
SEL o_orderkey, o_custkey, o_orderstatus, o_totalprice, o_clerk
FROM Orders;
```

o_orderkey	o_custkey	o_orderstatus	o_totalprice	o_clerk
12611	2050	0		
224970.76	Clerk#000000579			
12610	4228	F	9096.92	Clerk#000000952

Result: To view the updated data in the target table, run: `SEL * FROM Dealer;`

d_id	d_name	d_address	d_comment
12611	VBIT	HYD	AP Based dealer
12610	VBIT	HYD	AP Based dealer

XSLT Shredding Mapping Definition

XSLT-based shredding is based on a mapping defined by XSLT stylesheet documents. The stylesheet defines how different parts of the XML document map to columns in target tables in a relational database. Shredding procedures interpret the stylesheet mappings as instructions on what data items to extract from the input XML documents and how to use those data items in updating the target tables.

Example XSLT Stylesheet

The following shows the syntax of the stylesheet mapping.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
  <Transaction>
    <OPERATION>          ///Insert/Update/Delete/Upsert
    <Table>
      <TABLE NAME>        ///Table name in which to store the shredded values
      <xsl:for-each select="PATH">    ///Sub-tree path to iterate the shredding
        <Row>
          <xsl:copy-of select="ELEMENT PATH"/> ///ELEMENT PATH in XML sub tree
          OR
          <COLUMN><xsl:value-of select=" ELEMENT PATH "/></COLUMN>
          ///COLUMN is a column name in target table
        </Row>
      </xsl:for-each>
    </TABLE NAME >
  </Table>
</OPERATION>
</Transaction>
</xsl:template>
</xsl:stylesheet>
```

Transaction Element

The `<Transaction>` element groups together operations that are executed as a single transaction during XSLT based shredding. Each `<Transaction>` element contains one or more operation elements, each of which describes an Insert, Update, Delete, or Upsert operation against a single table.

Operation Elements

The operation elements describe the type of the operation performed on the target tables during XSLT based shredding. This element can have the values `<Insert>`, `<Update>`, `<Delete>`, or `<Upsert>`. This element is case sensitive.

The operation element contains the `<Table>` element that identifies the target tables that is affected by shredding. The child elements contain a table name. If the table name is unqualified it refers to a table in the default database.

The table name element contains a *key* attribute that identifies the primary key of the table. The *key* attribute can contain a comma separated list of column names.

The `xsl:for-each` instruction is used to generate the rows to be applied to the target table. "For each" node identified by the "select" XPath expression, a "Row" is produced and applied to the target table.

The element `<Row>` contains the list of columns and their mappings to be processed against the sub-trees.

There are two ways to map the column value:

- `<xsl:copy-of select="">` is used when the column name and element name are the same.
- `<xsl:value-of select="">` is used when the column name and element name are different.

XSLT_SHRED

Use the XSLT_SHRED stored procedure for XML shredding (extracting values from a single XML document to be used to populate the database), and is based on an XSLT stylesheet.

XSLT_SHRED returns a result code.

Required Privileges

The XSLT_SHRED stored procedure is created under the user TD_SYSEXML.

The user invoking this stored procedure must have the following privileges:

- EXECUTE PROCEDURE on TD_SYSEXML
- SELECT privilege on the staging table
- GRANT ALL (insert/update/delete/upsert) on the target tables

XSLT_SHRED Shredding Mapping Definition

XSLT-based shredding is based on a mapping defined by XSLT stylesheet documents. The stylesheet defines how different parts of the XML document map to columns in target tables in a relational database. Shredding procedures interpret the stylesheet mappings as instructions on what data items to extract from XML documents and how to use those data items in updating the target tables.

The XSLT_SHRED mapping definition is similar to the mapping definition for XSLT_SHRED_BATCH, except for the following differences.

- *genexp* is a mandatory attribute for all the elements under <Row>.
- Only the <xsl:value-of> element can be used; <xsl:copy-of> is not allowed.
- CLOB and BLOB are not supported in XSLT_SHRED.

XSLT_SHRED Syntax

```
TD_SYSEXML.XSLT_SHRED (
  xmlDoc,
  xsltMapping,
  externalContext,
  resultCode
)
```

Syntax Elements

xmlDoc

The XML document that will be shredded to the target table.

IN parameter of data type XML.

An error is reported if this parameter is NULL.

xsltMapping

An XSLT stylesheet provides the mapping from the XML document elements to be shredded to the target table columns.

IN parameter of data type XML.

externalContext

A semicolon separated list of *name=value* pairs that parameterize the shredding operation. These values override configuration parameters, such as defaultDatabase, that are specified in the annotated stylesheet mapping.

IN parameter of data type VARCHAR(512), CHARACTER SET UNICODE.

resultCode

A return code where:

- 0 = success
- 1 = failure

OUT parameter of data type INTEGER.

Example: XSLT_SHRED Using genexp

This example shows how to use the `genexp` declaration, which is required for `XSLT_SHRED`, unlike `XSLT_SHRED_BATCH`.

Note:

The Primary Index in the following example is not used as a key for the update.

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED_BATCH and XSLT_SHRED Examples](#) for details.

Populate the staging table used to store the XML source document.

```
INSERT INTO DefaultValue3(C_id, C_Byteintc) values(1,1);
```

Call the `XSLT_SHRED` stored procedure to shred the XML document. The following arguments are used for the call.

- The *xmlDoc* is created automatically in this example with the `NEW XML` statement.
- The *xsltMapping* argument is supplied by invoking the `CREATEXML` function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* is `NULL`.
- The *resultCode* is returned in `:res`. A successful operation returns 0.

```
CALL TD_SYSEXML.XSLT_SHRED(NEW XML ('<Root>
<AllTypes>
  <Numeric>
    <Byteintc>1</Byteintc>
    <Smallintc>200</Smallintc>
    <Intc>200</Intc>
    <Floatc>200</Floatc>
    <Decimalc>200</Decimalc>
    <Numberc>1200</Numberc>
  </Numeric>
</AllTypes>
```

```

<AllTypes>
  <Numeric>
    <Byteintc>2</Byteintc>
    <Smallintc>400</Smallintc>
    <Intc>400</Intc>
    <Floatc>400</Floatc>
    <Decimalc>400</Decimalc>
    <Numberc>1400</Numberc>
  </Numeric>
</AllTypes></Root>'), CREATEXML('<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/Root">
  <Transaction>
    <Update>
      <Table>
        <xsltuser.DefaultValue3 pkey="C_Byteintc">
          <xsl:for-each select="AllTypes">
            <Row>
              <C_Id genexp="cast(? as int)" default="10">
                <xsl:value-of select="Numeric/Bytec"/>
              </C_Id>
              <C_Byteintc genexp="cast(? as byteint)">
                <xsl:value-of select="Numeric/Byteintc"/>
              </C_Byteintc>
              <C_Smallintc genexp="cast(? as smallint)" >
                <xsl:value-of select="Numeric/Smallintc"/>
              </C_Smallintc>
              <C_Intc genexp="cast(? as integer)">
                <xsl:value-of select="Numeric/Intc"/>
              </C_Intc>
            </Row>
          </xsl:for-each>
        </xsltuser.DefaultValue3>
      </Table>
    </Update>
  </Transaction>
</xsl:template>
</xsl:stylesheet>'), ', :res);

```

Result: To view the updated data in the target table, run:

```
SELECT * FROM DefaultValue3;
```

C_Id	C_Byteintc	C_Smallintc	C_Intc
-----	-----	-----	-----
10	1	200	200

Example: XSLT_SHRED Usage Example

This example shows how to use XSLT_SHRED.

Before running the example, ensure the test user is created with the required permissions and the required tables are created. See [Setting Up the XSLT_SHRED BATCH and XSLT_SHRED Examples](#) for details.

Call the XSLT_SHRED stored procedure to shred the XML document. The following arguments are used for the call.

- The *xmlDoc* is created automatically in this example with the NEW XML statement.
- The *xsltMapping* argument is supplied by invoking the CREATEXML function with a stylesheet as input. This stylesheet will be applied to the XML document.
- The *externalContext* is NULL.
- The *resultCode* is returned in :res. A successful operation returns 0.

```
CALL TD_SYSXML.XSLT_SHRED(NEW XML(
'<Root>
<AllTypes>
  <DateTime>
    <Datec>2000-09-09</Datec>
    <Timec>12:12:12</Timec>
    <Timewzc>12:12:12+00:00</Timewzc>
    <TimeStampc>2000-09-09T12:12:12</TimeStampc>
    <TimeStampwzc>2000-09-09T12:12:12+00:00</TimeStampwzc>
  </DateTime>
</AllTypes>
<AllTypes>
  <DateTime>
    <Datec>2000-09-10</Datec>
    <Timec>12:12:12</Timec>
    <Timewzc>12:12:12+00:00</Timewzc>
    <TimeStampc>2000-09-09T12:12:12</TimeStampc>
  </DateTime>
</AllTypes>
</Root>'),
  CREATEXML('<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:date="http://exslt.org/dates-and-times"
```

```

extension-element-prefixes="date" version="1.0" >
<xsl:param name="XML_TimeStamp" select="date:date-time()"/>
<xsl:template match="/Root">
  <Transaction>
    <Insert>
      <Table>
        <xsltuser.DefaultValue4>
          <xsl:for-each select="AllTypes">
            <Row>
              <C_Datec genexp="( ? as Date)"><xsl:value-of select="DateTime/Datec"/></C_Datec>
              <C_Timeec genexp="( ? as Time(6))">
                <xsl:value-of select="DateTime/Timeec"/></C_Timeec>
              <C_Timewzc genexp="( ? as Time(6) WITH TIME ZONE)">
                <xsl:value-of select="DateTime/Timewzc"/></C_Timewzc>
              <C_TimeStampc genexp="( ? as TIMESTAMP(6))">
                <xsl:value-of select="DateTime/TimeStampc"/></C_TimeStampc>
              <C_TimeStampwzc genexp="( ? as TIMESTAMP(6) WITH TIME ZONE)"
                default="{ $XML_TimeStamp }">
                <xsl:value-of select="DateTime/TimeStampwzc"/></C_TimeStampwzc>
            </Row>
          </xsl:for-each>
        </xsltuser.DefaultValue4>
      </Table>
    </Insert>
  </Transaction>
</xsl:template>
</xsl:stylesheet>'), ', :res);

```

Result: To view the updated data in the target table, run: `SELECT * FROM DefaultValue4;`

C_Datec	C_Timeec	C_Timewzc	C_TimeStampc	C_TimeStampwzc
00/09/10	12:12:12. 000000	12:12:12. 000000+00:00	2000-09-09 12:12: 12.000000	2014-05-07 05:08:17. 000000-04:00
00/09/09	12:12:12. 000000	12:12:12. 000000+00:00	2000-09-09 12:12: 12.000000	2000-09-09 12:12:12. 000000+00:00

XML Publishing

XML publishing is the process of formatting the results of an SQL query as XML. SQL query results are mapped to an XML format according to a mapping specification provided by the user.

To publish XML from Vantage, perform the following tasks:

1. Define the SQL query whose output you want to publish as XML.
2. Define a mapping from the SQL query results to the desired XML structure.
3. Store the mapping so that it can be easily referenced, for example, in a stylesheet repository table.
4. Call one of the XML publishing stored procedures (XMLPUBLISH or XMLPUBLISH_STREAM) with the following arguments:
 - The SQL query you defined in step 1
 - The mapping you saved in step 3

You define the XML publishing mapping as an XSLT stylesheet that defines a transformation from the canonical representation of the query result to the desired XML output structure.

If you pass NULL as the Xslt (mapping definition) argument to XMLPUBLISH or XMLPUBLISH_STREAM, the stored procedures will return the canonical XML representation as output.

Canonical XML Publishing

The canonical XML representation of SQL query results takes the form of a document element named QuerySchema which has 0 or more child elements named Row. Each Row element corresponds to a row in the SQL query result set. The Row elements have children elements corresponding to the projection list of the SQL query. For example, consider an SQL query such as the following:

```
SELECT CUSTDTL.ID AS CustomerID, CUSTDTL.NAME, CUSTDTL.ADDRESS, CUSTDTL.PHONE1,
      CUSTDTL.PHONE2, CUSTDTL.FAX, CUSTDTL.EMAIL, ORDERDTL.ORDER_ID AS OrderNumber,
      ORDERDTL.ORDER_DATE AS OrderDate, ORDERDTL.CONTACT AS OrderContact,
      ORDERDTL.CONTACT_PHONE AS OrderPhone, ORDERDTL.SHIP_TO_ADDRESS AS OrderShipTo,
      ORDERDTL.SUB_TOTAL AS OrderSubTotal, ORDERDTL.TAX AS OrderTax, ORDERDTL.TOTAL AS
      OrderTotal, LINEITEMDTL.ITEM_ID AS ItemID, LINEITEMDTL.QUANTITY AS ItemQuantity,
      LINEITEMDTL.PART_NUMBER AS ItemPartNumber, LINEITEMDTL.DESCRPTION
      AS ItemDescription,
      LINEITEMDTL.UNIT_PRICE AS ItemUnitPrice, LINEITEMDTL.PRICE AS ItemPrice
FROM CUSTDTL, ORDERDTL, LINEITEMDTL
WHERE CUSTDTL.ID=ORDERDTL.CUSTOMER_ID
AND ORDERDTL.ORDER_ID=LINEITEMDTL.ORDER_ID;
```

The CUSTDTL, ORDERDTL, and LINEITEMDTL tables reflect the purchase order history data.

Mapping SQL Query Results to an XML Format

You can create an XSLT stylesheet that provides a mapping definition describing how an SQL query result is mapped to an XML tree structure. XSLT (eXtensible Stylesheet Language Transformation) is the language used for describing transformations from one XML structure to another or to other text formats. For XML publishing, use the mapping stylesheet to define a transformation from the canonical XML representation of a query result to the desired XML structure.

The following is an example of an XSLT mapping for publishing the results of the query shown in the previous section. The example defines a transformation from the canonical XML representation. For example, the `xsl:for-each` instruction constructs a customer element for each Row element in the canonical XML representation.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <customers>
      <xsl:for-each select="QuerySchema/ROW">
        <customer teradata_group="CUSTID">
          <CUSTID>
            <xsl:value-of select="CUSTID"/>
          </CUSTID>
          <CNAME>
            <xsl:value-of select="CNAME"/>
          </CNAME>
          <ACCTID>
            <xsl:value-of select="ACCTID"/>
          </ACCTID>
          <ANAME>
            <xsl:value-of select="ANAME"/>
          </ANAME>
          <ORDERID>
            <xsl:value-of select="ORDERID"/>
          </ORDERID>
          <ODATE>
            <xsl:value-of select="ODATE"/>
          </ODATE>
          <REPID>
            <xsl:value-of select="REPID"/>
          </REPID>
          <REPNAME>
            <xsl:value-of select="REPNAME"/>
          </REPNAME>
          <ITEMID>
            <xsl:value-of select="ITEMID"/>
          </ITEMID>
          <QTY>
            <xsl:value-of select="QTY"/>
          </QTY>
        </customer>
      </xsl:for-each>
    </customers>
  </template>
</xsl:stylesheet>
```

```
</xsl:template>
</xsl:stylesheet>
```

Adding Annotations to the Mapping Stylesheet

The stylesheet provides structure and tagging information. Unfortunately, XSLT 1.0, the version most commonly supported by design tools and processors does not support operations such as grouping. Because grouping is a key aspect of converting a tabular structure to a hierarchy, Teradata allows you to annotate the stylesheet with special elements that provide hints as to how grouping should be done. These elements do not show up in the result document. The following sections describe these special elements.

Note:

The term annotation used here does not mean using comments. The process of constructing the stylesheet can be done within the graphical design environment provided by Stylus Studio and Tiger XSLT Mapper. The capability these tools have in common is to construct elements on the target schema; therefore, these annotations are in the form of XML Element nodes and not XML Comment nodes.

teradata_group Attribute

Use the `teradata_group` attribute in the stylesheet to annotate the result structure with information regarding which fields comprise repeating groups, specifying the key fields on which grouping will be done. To specify the grouping behavior, place a `teradata_group` attribute under which the grouping will happen and assign it a value on which the grouping will be done at that level.

In the previous Purchase Order example, the fields that comprise repeating groups are represented by the Customer, Order, and Item elements.

teradata_optional Attribute teradata_optional_attributes Attribute

You can specify the following in the stylesheet:

- The elements that are optional and will not be serialized if the corresponding value is a NULL
- The elements that are required. An empty element, `<el/>`, or attribute with a null string value, `at=""`, is created if the corresponding value is NULL.

Use the `teradata_optional` attribute to specify whether an element is generated as an empty element with an empty string value, or whether the element is skipped if the corresponding value in the SQL result set is NULL.

Use the `teradata_optional_attributes` attribute to specify which attribute(s) of the element is skipped if the corresponding value(s) in the SQL result set is NULL.

teradata_sort Attribute

If you want to specify the order of the XML elements in the hierarchy of the XML document, use the `teradata_sort` attribute to specify the value of the child element by which the elements will be ordered.

Related Information

- See [XMLPUBLISH](#).
- See [XMLPUBLISH_STREAM](#).

XMLPUBLISH

Publishes SQL query results in an XML format specified by the user.

The stored procedure takes the following as input arguments:

- An SQL query string
- A mapping stylesheet that will transform the results of the SQL query to the desired XML structure.

If you do not specify a mapping (you pass NULL as the *Xslt* argument), the procedure returns the canonical XML representation as output.

Recommendation: Use this stored procedure in the following cases:

- The size of the documents being published is small
- The use of Teradata specific instructions, like `teradata_group`, is not required

Note:

Teradata specific instructions are ignored by XMLPUBLISH.

- You need to publish multiple documents

For other cases, use the XMLPUBLISH_STREAM stored procedure instead.

Required Privileges

The XMLPUBLISH stored procedure is created under the user TD_SYSEXML.

The user invoking this stored procedure must have the following privileges:

- EXECUTE PROCEDURE on TD_SYSEXML
- SELECT privilege on the staging table
- GRANT ALL (insert/update/delete/upsert) on the target tables

Result Type

The result *XmlString* is a string representation of the XML to be published.

XMLPUBLISH returns a *resultCode* value:

- 0 indicates that the procedure executed successfully.
- -1 indicates that the procedure resulted in an error.

XMLPUBLISH Syntax

```
TD_SYSXML.XMLPUBLISH (
  queryString,
  Xslt,
  XmlString,
  resultCode
)
```

Syntax Elements

queryString

The SQL query string whose results are published in XML format.

IN parameter of data type VARCHAR(20000), CHARACTER SET UNICODE.

If *queryString* is NULL, the *XmlString* return value will be NULL.

Xslt

The XSLT stylesheet representing the mapping from the tabular model of the SQL query result set to the tree model of the XML output.

IN parameter of data type XML.

The *Xslt* argument is an XML type instance representing a consolidated stylesheet or simply a stylesheet if it does not use stylesheet includes. This stylesheet should map the user query results to the desired XML output.

If *Xslt* is NULL, the *XmlString* return value is the XML resulting from canonical publishing.

XmlString

The string representation of XML output.

IN parameter of data type CLOB, CHARACTER SET UNICODE.

resultCode

XMLPUBLISH returns a result code as follows:

- 0 indicates success
- -1 indicates failure

OUT parameter of data type INTEGER.

Related Information

- For information on defining an XSLT stylesheet that represents a mapping for XML publishing, see [Mapping SQL Query Results to an XML Format](#).
- See [Canonical XML Publishing](#).
- See [XMLPUBLISH_STREAM](#).

XMLPUBLISH_STREAM

Publishes one or more XML documents as a stream.

The XMLPUBLISH_STREAM stored procedure publishes one or more XML documents as a stream. The entire document to be published is not materialized all at once. Instead, XMLPUBLISH_STREAM returns XML in chunks that will be concatenated on the client side to create well-formed XML document(s). This allows construction of large documents within memory constraints.

While processing a document, XMLPUBLISH_STREAM converts certain special characters to character entity references; for example, < is converted to < and & is converted to &. This is done to escape special characters in the data to ensure that well-formed XML is produced. An empty element is returned for an empty string and the element will be missing for a null string.

The stored procedure takes the following as input arguments:

- An SQL query string
- A mapping stylesheet that will transform the results of the SQL query to the desired XML structure.

If you do not specify a mapping (you pass NULL as the *Xslt* argument), the procedure returns the canonical XML representation as output.

The procedure returns a result set composed of one column: *XmlString*. The client application concatenates the values in the *XmlString* column to create the XML document to be published.

If you specify *documentGroupingSpec*, XMLPUBLISH_STREAM allows the construction of multiple XML documents by grouping the query results based on *documentGroupingSpec*. The result set is interspersed with NULLs which mark the boundaries of the documents.

Recommendation: Use the XMLPUBLISH stored procedure instead of XMLPUBLISH_STREAM for these cases:

- The size of the documents being published is small
- The use of Teradata specific instructions, like *teradata_group*, is not required
- You need to publish multiple documents

Required Privileges

The XMLPUBLISH_STREAM stored procedure is created under the user TD_SYSEXML.

The user invoking this stored procedure must have the following privileges:

- EXECUTE PROCEDURE on TD_SYSEXML

- SELECT privilege on the staging table
- GRANT ALL (insert/update/delete/upsert) on the target tables

Result Type

The result set is composed of a single column, *XmlString*, whose values need to be concatenated to create the XML document to be published.

XMLPUBLISH_STREAM Syntax

```
TD_SYSXML.XMLPUBLISH_STREAM (
    queryString,
    Xslt,
    documentGroupingSpec
)
```

Syntax Elements

queryString

The SQL query string whose results are published in XML format.

IN parameter of data type VARCHAR(20000), CHARACTER SET UNICODE.

If *queryString* is NULL, the *XmlString* return value will be NULL.

Xslt

The XSLT stylesheet representing the mapping from the tabular model of the SQL query result set to the tree model of the XML output.

IN parameter of data type XML.

The *Xslt* argument is an XML type instance representing a consolidated stylesheet or simply a stylesheet if it does not use stylesheet includes. This stylesheet should map the user query results to the desired XML output.

If *Xslt* is NULL, the *XmlString* return value is the XML resulting from canonical publishing.

documentGroupingSpec

A string on the basis of which the query results are organized into XML documents. The string format is a comma-separated list of column names.

IN parameter of data type VARCHAR(512), CHARACTER SET UNICODE.

If *documentGroupingSpec* is NULL, the *XmlString* column in the result set represents a single document representing the query results in XML format. If *documentGroupingSpec*

is not NULL, the *XmlString* column represents multiple documents, with the documents separated by a NULL in the *XmlString* column.

Related Information

- For information on defining an XSLT stylesheet that represents a mapping for XML publishing, see [Mapping SQL Query Results to an XML Format](#).
- [Canonical XML Publishing](#).
- [XMLPUBLISH](#).

Notation Conventions

How to Read Syntax

This document uses the following syntax conventions.

Syntax Convention	Meaning
KEYWORD	Keyword. Spell exactly as shown. Many environments are case-insensitive. Syntax shows keywords in uppercase unless operating system restrictions require them to be lowercase or mixed-case.
<i>variable</i>	Variable. Replace with actual value.
<i>number</i>	String of one or more digits. Do not use commas in numbers with more than three digits. Example: 10045
[x]	x is optional.
[x y]	You can specify x, y, or nothing.
{ x y }	You must specify either x or y.
x [...]	You can repeat x, separating occurrences with spaces. Example: x x x See note after table.
x [, ...]	You can repeat x, separating occurrences with commas. Example: x, x, x See note after table.
x [<i>delimiter</i> ...]	You can repeat x, separating occurrences with specified delimiter. Examples: <ul style="list-style-type: none"> If <i>delimiter</i> is semicolon: x; x; x If <i>delimiter</i> is { , OR }, you can do either of the following: <ul style="list-style-type: none"> x, x, x x OR x OR x See note after table.

Note:

You can repeat only the immediately preceding item. For example, if the syntax is:

```
KEYWORD x [...]
```

You can repeat x. Do not repeat KEYWORD.

If there is no white space between x and the delimiter, the repeatable item is x and the delimiter. For example, if the syntax is:

```
[ x, [...] ] y
```

- You can omit x: y
- You can specify x once: x, y
- You can repeat x and the delimiter: x, x, x, y

Character Shorthand Notation Used in This Document

This document uses the Unicode naming convention for characters. For example, the lowercase character 'a' is more formally specified as either LATIN CAPITAL LETTER A or U+0041. The U+xxxx notation refers to a particular code point in the Unicode standard, where xxxx stands for the hexadecimal representation of the 16-bit value defined in the standard.

In parts of the document, it is convenient to use a symbol to represent a special character, or a particular class of characters. This is particularly true in discussion of the following Japanese character encodings:

- KanjiEBCDIC
- KanjiEUC
- KanjiShift-JIS

These encodings are further defined in *Teradata Vantage™ - Advanced SQL Engine International Character Set Support*, B035-1125.

Character Symbols

The symbols, along with character sets with which they are used, are defined in the following table.

Symbol	Encoding	Meaning
a-z A-Z 0-9	Any	Any single byte Latin letter or digit.
<u>a-z</u> <u>A-Z</u> <u>0-9</u>	Any	Any fullwidth Latin letter or digit.

Symbol	Encoding	Meaning
<	KanjiEBCDIC	Shift Out [SO] (0x0E). Indicates transition from single to multibyte character in KanjiEBCDIC.
>	KanjiEBCDIC	Shift In [SI] (0x0F). Indicates transition from multibyte to single byte KanjiEBCDIC.
T	Any	Any multibyte character. The encoding depends on the current character set. For KanjiEUC, code set 3 characters are always preceded by <code>ss3</code> .
!	Any	Any single byte Hankaku Katakana character. In KanjiEUC, it must be preceded by <code>ss2</code> , forming an individual multibyte character.
<u>Δ</u>	Any	Represents the graphic pad character.
Δ	Any	Represents a single or multibyte pad character, depending on context.
ss 2	KanjiEUC	Represents the EUC code set 2 introducer (0x8E).
ss 3	KanjiEUC	Represents the EUC code set 3 introducer (0x8F).

For example, string “TEST”, where each letter is intended to be a fullwidth character, is written as **TEST**. Occasionally, when encoding is important, hexadecimal representation is used.

For example, the following mixed single byte/multibyte character data in KanjiEBCDIC character set:

LMN<TEST>QRS

is represented as:

D3 D4 D5 0E 42E3 42C5 42E2 42E3 0F D8 D9 E2

Pad Characters

The following table lists the pad characters for the various character data types.

Server Character Set	Pad Character Name	Pad Character Value
LATIN	SPACE	0x20
UNICODE	SPACE	U+0020
GRAPHIC	IDEOGRAPHIC SPACE	U+3000
KANJISJIS	ASCII SPACE	0x20
KANJI1	ASCII SPACE	0x20

External Representations for the XML Type

This section describes the XML data type from the perspective of client applications and drivers.

Text Format for XML Values

XML is a markup language; therefore, data or text is marked up with tags that represent metadata that makes the XML self-describing. The universally recognized XML markup scheme is text-based.

Teradata drivers support a text-based format for XML values. A client application or driver that uses the text format might require access to an XML parser to be able to parse the XML received on the wire.

The text format returns the XML serialization of the XML value. XML values retrieved in the text format are encoded in UTF-8. The following shows an example of document retrieval.

```
<?xml version="1.0" encoding="UTF-8"?>
<customer>
  <Name>John Hancock</Name>
  <Address>100 1st Street One City, CA 12345</Address>
  <Phone1>(999)9999-999</Phone1>
  <Phone2>(999)9999-998</Phone2>
  <Fax>(999)9999-997</Fax>
  <Email>John@somecompany.com</Email>
  <order Number="NW-01-16366" Date="Feb/28/2001">
    <Contact>Mary Shannon</Contact>
    <Phone>(987)6543-210</Phone>
    <ShipTo>Widgets Inc., 123 Regency Parkway, Portland, OR 43211</ShipTo>
    <SubTotal>2355.00</SubTotal>
    <Tax>141.50</Tax>
    <Total>2496.50</Total>
    <item ID="001">
      <Quantity>100</Quantity>
      <PartNumber>F54709</PartNumber>
      <Description/>
      <UnitPrice>23.55</UnitPrice>
      <Price>2355.00</Price>
    </item>
  </order>
</customer>
```

For the text format, the server will not try to pretty-print the XML value when serializing it for transfer to the client. If the client application needs to display the XML, it is up to the application to decide whether to do pretty-printing, provide syntax-highlighting, or other display options.

Document Retrieval

The following sections show how users can store XML documents into XML type columns and retrieve them via SQL queries that either retrieve the entire document or evaluate an XPath/XQuery expression that returns a single document node.

Consider the following queries:

```
SELECT id, xmldoc FROM xmltab;
SELECT id, xmldoc.xmlextract('.', NULL) FROM xmltab;
```

The result of each query represents a single tree rooted in a document node. In this case, the server returns the serialized representation of the document. The serialization of the document is not guaranteed to be lexically equivalent to the document that was inserted, but it will be semantically equivalent (the canonical representation of the original document that was inserted and the document retrieved will be lexically equivalent). Drivers can either parse the document returned or pass on the stream to applications.

XPath/XQuery Result Retrieval

The following examples return results of an XQuery/XPath query evaluation. If the result is a single document or element node, it can be returned directly. If the result is a sequence, it can be cast to a string data type (VARCHAR, CLOB), or you can use the XMLSERIALIZE function to return a string representation of the value. Alternately, you can use the XMLTABLE function to split up items of a sequence. If the items are atomic values, you can cast them to the appropriate SQL type.

Examples

Example: XQuery/XPath Query Returns the First Customer Child

The following query returns the first customer child of the customers document element:

```
SELECT xmldoc.xmlextract('/customers/customer[1]', NULL)
AS firstcustomer
FROM xmltab;
```

The following shows the wire-format of the XML value that is the result of this query. The single element node when serialized is a well-formed XML document.

```
<customer>
  <Name>John Hancock</Name>
  <Address>100 1st Street One City, CA 12345</Address>
  <Phone1>(999)9999-999</Phone1>
  <Phone2>(999)9999-998</Phone2>
  <Fax>(999)9999-997</Fax>
```

```

<Email>John@somecompany.com</Email>
<order Number="NW-01-16366" Date="Feb/28/2001">
  <Contact>Mary Shannon</Contact>
  <Phone>(987)6543-210</Phone>
  <ShipTo>Widgets Inc., 123 Regency Parkway, Portland, OR 43211</ShipTo>
  <SubTotal>2355.00</SubTotal>
  <Tax>141.50</Tax>
  <Total>2496.50</Total>
  <item ID="001">
    <Quantity>100</Quantity>
    <PartNumber>F54709</PartNumber>
    <Description/>
    <UnitPrice>23.55</UnitPrice>
    <Price>2355.00</Price>
  </item>
</order>
</customer>

```

Example: XQuery/XPath Query Returns a Sequence of Customer Elements

The following query returns a sequence of customer elements:

```

SELECT CAST(xml doc.xml extract('/customers/customer', NULL) AS VARCHAR(1024))
AS customerlist
FROM xmltab;

```

The return value for this query is of VARCHAR type containing the following string (serialization of a sequence is the serialization of the items of the sequence separated by a single white space character):

```

<customer>
  <Name>John Hancock</Name>
  <Address>100 1st Street One City, CA 12345</Address>
  <Phone1>(999)9999-999</Phone1>
  <Phone2>(999)9999-998</Phone2>
  <Fax>(999)9999-997</Fax>
  <Email>John@somecompany.com</Email>
  <order Number="NW-01-16366" Date="Feb/28/2001">
    <Contact>Mary Shannon</Contact>
    <Phone>(987)6543-210</Phone>
    <ShipTo>Widgets Inc., 123 Regency Parkway, Portland, OR 43211</ShipTo>
    <SubTotal>2355.00</SubTotal>
    <Tax>141.50</Tax>
    <Total>2496.50</Total>
  </order>
</customer>

```

```

    <item ID="001">
      <Quantity>100</Quantity>
      <PartNumber>F54709</PartNumber>
      <Description/>
      <UnitPrice>23.55</UnitPrice>
      <Price>2355.00</Price>
    </item>
  </order>
</customer>
<customer>
  <Name>Jim Smith</Name>
  <Address>200 2nd Street, San Diego, CA 12345</Address>
  <Phone1>(858)555-1234</Phone1>
  <Phone2>(858)555-1234</Phone2>
  <Fax>(858)555-9876</Fax>
  <Email>Jim@somecompany.com</Email>
  <order Number="JS-01-16366" Date="Feb/29/2001">
    <Contact>Jim Smith</Contact>
    <Phone>(858)555-1234</Phone>
    <ShipTo>Acme co., 2467 Pioneer Road, San Diego, CA 12345</ShipTo>
    <SubTotal>1242.00</SubTotal>
    <Tax>141.50</Tax>
    <Total>1383.50</Total>
    <item ID="001">
      <Quantity>10</Quantity>
      <PartNumber>F54709</PartNumber>
      <Description/>
      <UnitPrice>124.20</UnitPrice>
      <Price>1242.00</Price>
    </item>
  </order>
</customer>

```

Example: XQuery/XPath Query Returns a Sequence of xs:integer or xs:double Values

This query returns a sequence of xs:integer or xs:double values depending on the value of the total element in the XML document:

```

SELECT CAST(XMLQUERY('for $cust in /customers/customer return sum(order/
total)' passing
xmldoc) AS VARCHAR(512))

```

```
AS ordertotals
FROM xmltab;
```

The result of this query is the following string value of type VARCHAR (XQuery cast of each item to xs:string, separated by a single white space character):

```
2496.50 1383.50
```

Example: XQuery/XPath Query Uses XMLTABLE to Retrieve Items From a Sequence

The following query uses XMLTABLE to retrieve items from a sequence:

```
SELECT x.item
FROM xmltab,
XMLTABLE('$custXML/customers/customer'
         passing xmltab.xmldoc as "custXML"
         COLUMNS "item" XML PATH ".")
AS x;
```

The result of this query is multiple rows, each containing a single customer element in the item column.

Row #1:

```
<customer>
  <Name>John Hancock</Name>
  <Address>100 1st Street One City, CA 12345</Address>
  <Phone1>(999)9999-999</Phone1>
  <Phone2>(999)9999-998</Phone2>
  <Fax>(999)9999-997</Fax>
  <Email>John@somecompany.com</Email>
  <order Number="NW-01-16366" Date="Feb/28/2001">
    <Contact>Mary Shannon</Contact>
    <Phone>(987)6543-210</Phone>
    <ShipTo>Widgets Inc., 123 Regency Parkway, Portland, OR 43211</ShipTo>
    <SubTotal>2355.00</SubTotal>
    <Tax>141.50</Tax>
    <Total>2496.50</Total>
    <item ID="001">
      <Quantity>100</Quantity>
      <PartNumber>F54709</PartNumber>
      <Description/>
      <UnitPrice>23.55</UnitPrice>
      <Price>2355.00</Price>
    </item>
```

```

    </order>
  </customer>

```

Row #2:

```

<customer>
  <Name>Jim Smith</Name>
  <Address>200 2nd Street, San Diego, CA 12345</Address>
  <Phone1>(858)555-1234</Phone1>
  <Phone2>(858)555-1234</Phone2>
  <Fax>(858)555-9876</Fax>
  <Email>Jim@somecompany.com</Email>
  <order Number="JS-01-16366" Date="Feb/29/2001">
    <Contact>Jim Smith</Contact>
    <Phone>(858)555-1234</Phone>
    <ShipTo>Acme co., 2467 Pioneer Road, San Diego, CA 12345</ShipTo>
    <SubTotal>1242.00</SubTotal>
    <Tax>141.50</Tax>
    <Total>1383.50</Total>
    <item ID="001">
      <Quantity>10</Quantity>
      <PartNumber>F54709</PartNumber>
      <Description/>
      <UnitPrice>124.20</UnitPrice>
      <Price>1242.00</Price>
    </item>
  </order>
</customer>

```

Example: XQuery/XPath Query Returns Multiple Atomic Values Cast to the DECIMAL Data Type

The following query returns multiple atomic values cast to the DECIMAL data type:

```

SELECT CAST(x.item AS DECIMAL(16,0))
FROM xmltab,
XMLTABLE(
  'for $cust in /customers/customer return sum(order/total)'
  passing xmltab.xml doc as "cust"
  COLUMNS "item" XML PATH ".") x;

```

An alternative way to write this query is as follows:

```

SELECT x.item
FROM xmltab,
XMLTABLE(
  'for $cust in /customers/customer return sum(order/total)'
  passing xmltab.xmldoc as "cust"
  COLUMNS "item" DECIMAL(16,0) PATH ".") x;

```

Both queries return the same results: each row returned will contain a single DECIMAL value.

Row #1:

2496.50

Row #2:

1383.50

Character Sets

XML documents declare their encoding in the XML declaration. The XML type implementation parses and stores XML in the database with the character data encoded in UTF-8 encoding. XML documents transferred from the client to the server using the text format are expected to be encoded in UTF-8. The encoding specified in the XML declaration is ignored by the XML type implementation on the server side in this case. Similarly, XML type values transferred from server to client in the text format are encoded in UTF-8.

Xerces supports the following encodings out of the box.

- | | |
|--|--|
| <ul style="list-style-type: none"> • ASCII • UTF-8 • UTF-16 (big/little endian) • UCS4 (big/little endian) | <ul style="list-style-type: none"> • EBCDIC code pages IBM037, IBM1047 and IBM1140 • ISO-8859-1 (Latin1) • Windows-1252 |
|--|--|

In the base case, documents can always be loaded and returned as the following:

- XML type because UTF-8 is supported out of the box by Xerces
- VARCHAR/CLOB because the transcoding to UTF-8/16 is handled by the DBS

Documents loaded as BLOBs can only be in the supported encodings.

XML Data Loading

You can load well-formed XML documents; however, loading sequences, atomic values, and similar values are not supported. These can be query results, but not values that are loaded into the database. When loading character string formats, XML documents that are less than 64 KB in size can be loaded the same way as VARCHAR type values. Larger values can be inserted by load utilities that support LOB types.

Encoding of Documents For Loading

An XML parser expects that the document encoding information is provided in one of the following ways:

- The encoding is declared in the XML declaration
- External information is provided to the parser. For example, the encoding set is provided through the parser API.

Otherwise, the encoding should be UTF-8 or UTF-16.

When transferring XML values from client to server, the following encoding rules apply.

IF the document is loaded as...	THEN the document encoding should be...
text format XML type value	UTF-8.
VARCHAR or CLOB	the same as the session character set.
BLOB	the same as indicated by the XML declaration.

XML Data Loading as XML Data Type

When an XML document is loaded as XML type, the driver encodes the XML document content in UTF-8. The document when received by the DBS might have an encoding that does not match the XML declaration, such as when the declaration states that the encoding is something other than UTF-8. In this case, the parser is told to ignore the XML declaration and assume a UTF-8 encoding.

XML Data Loading as VARCHAR or CLOB Data Type

The content is expected to be a well-formed XML document (a document node with a single child element) in its character string representation. Attempts to load a document that is not well-formed results in an error.

When loading the document as a VARCHAR or CLOB, the driver encodes the document in the session character set. The constructors of XML type values from VARCHAR or CLOB transcodes this to Unicode (UTF-8) and ignores the encoding specified in the XML declaration.

XML Data Loading as BLOB Data Type

Values transferred as BLOBs retain their original encoding which should match the encoding specified in the XML declaration. In case of a mismatch between the actual encoding and the encoding declared in the XML declaration, an error is raised by the parser.

XML Data Unloading

XML values retrieved from the server can either be returned as XML type values, or as VARCHAR, CLOB, or BLOB types using the XMLSERIALIZE function.

XML values can also be returned as CLOB, BLOB, VARCHAR, or VARBYTE based on the transform group that was used.

Encoding of Documents For Retrieval

When transferring XML values from server to client, the following encoding rules apply.

IF the value is returned as...	THEN the value is encoded as...
text format XML type value	UTF-8.
VARCHAR or CLOB	the same as the session character set.
BLOB	the encoding requested in the call to the XMLSERIALIZE function.

XML Data Retrieval as XML Data Type

XML data type values that are documents or elements can be retrieved in text format. Sequences must be transformed to elements either in XQuery by using element constructors or in SQL by using the XMLTABLE function.

In text format, the value is the serialization of the document or element node encoded in UTF-8.

XML Data Retrieval as VARCHAR or CLOB Data Type

XML values can be serialized to a character string and returned as values of VARCHAR or CLOB data types. This serialization can be used with any XML value such as documents, sequences, or atomic values.

The serialization of a document is its well-formed string representation. For a sequence, this is the serialization of its items with intervening white space. For atomic values, it is the canonical lexical representations of those values as described by the XML schema specification.

Values serialized in this fashion are returned to the client encoded in the session character set. If any of the characters cannot be represented in the session character set, an error is raised.

XML Data Retrieval as BLOB Data Type

If the XML data is retrieved serialized as BLOB and the value is a document node, it is returned in its original encoding. If an application wants to insert the document and get it back in the same encoding (round-tripping), it can insert and retrieve documents as BLOBs. In this case, there is no guarantee that the documents will be lexically identical; however, they will be equal in the sense that their canonical representations will be identical.

If the value is a query result, it is returned encoded in UTF-8.

Data Type Encoding Numbers for the XML Type

Some of the client and server interfacing parcels such as DataInfoX (parcel flavor number 146), PrepInfoX (flavor 125), and StatementInfo parcel (flavor 169) return the data type of the field. All parcels that contain the data type information use the following encoding for the XML data type. The encoding numbers defined follow the pattern for existing data types. For example, Nullable number is non-Nullable value + 1 and stored procedure IN parameter type number is 500 + non-nullable number.

	NULL Property		Stored Procedure Parameter Type		
	Non-nullable	Nullable	IN	INOUT	OUT
XML Text Inline - for small XML values	852	853	1352	1353	1354
XML Text Deferred	856	857	1356	1357	1358
XML Text Locator	860	861	1360	1361	1362

Configuration Response Parcel

The Configuration response parcel indicates support for XML data type and XQuery capabilities. The 17th byte in the SQL-Capabilities extension (item code 10) indicates support for XML and XQuery as follows:

- Binary zero indicates that the XML/XQuery feature is not supported.
- Binary one indicates that the XML/XQuery feature is supported.

StatementInfo Parcel

The StatementInfo parcel returns the parameter and/or query result metadata from the server to the client. The metadata items impacted by the XML data type are listed below along with the values they carry. The metadata items follow the rules as defined for the specific item.

Metadata Item Field	Item Field Description
Data type	2 byte integer
UDT indicator	2 byte integer or 3 bytes for internal UDT
Maximum data length in bytes	2 GB
Fully qualified type name length	0
Fully qualified type name	omitted
isSortable	Always "N" (XML is not comparable)

Data Parcels

The XML type is a LOB-type internal UDT and the data is transferred in a manner similar to other LOB types. The XML values are transferred between the server and client in text format. Large values are split across multiple parcels. Existing rules for CLOB data type regarding splitting a single character across multiple parcels apply to the text format. When receiving data, the client will concatenate the data and use the appropriate parser to get the XML value. When sending data, the client should have the capability to split the XML document serializations so that the server, after concatenating the data across the parcels, will have a well-formed document.

Related Information

- For information about the XMLSERIALIZE function, see [XMLSERIALIZE](#).
- For information about the XMLTABLE function, see [XMLTABLE](#).
- For a list of encoding names supported by the Teradata XML implementation, see [Encoding Names Supported by Teradata XML](#).

Encoding Names Supported by Teradata XML

Encoding Names Supported by Teradata XML

The following is a list of encoding names supported by the Teradata XML implementation. Some of these names represent multiple ways of referencing the same encoding. For example, UTF-8 and UTF8 are equivalent, and either name is acceptable.

- | | |
|--|--|
| <ul style="list-style-type: none"> • UTF-8 • UTF8 • UTF-16 • UTF16 • UTF-16 (BE) • UTF-16BE • UTF-16 (LE) • UTF-16LE • UCS2 • UCS-2 • UCS4 • UCS-4 • UCS_4 • ISO-10646-UCS-4 • UCS-4 (BE) • UCS-4BE • UCS-4 (LE) • UCS-4LE • IBM-1200 • IBM1200 • ASCII • US-ASCII • US_ASCII | <ul style="list-style-type: none"> • USASCII • ISO8859-1 • ISO-8859-1 • ISO_8859-1 • LATIN1 • LATIN-1 • LATIN_1 • IBM-819 • IBM819 • CP819 • CSISOLATIN1 • ISO-IR-100 • L1 • EBCDIC-CP-US • IBM037 • IBM1047 • IBM-1047 • IBM1140 • IBM01140 • CCSID01140 • CP01140 • WINDOWS-1252 |
|--|--|

Additional Information

Teradata Links

Link	Description
https://docs.teradata.com/	Search Teradata Documentation, customize content to your needs, and download PDFs. Customers: Log in to access Orange Books.
https://support.teradata.com	One-stop source for Teradata community support, software downloads, and product information. Log in for customer access to: <ul style="list-style-type: none">• Community support• Software updates• Knowledge articles
https://www.teradata.com/University/Overview	Teradata education network
https://support.teradata.com/community	Link to Teradata community