XQuery Language Reference

SQL Server 2012 Books Online

Reference

Microsoft

XQuery Language Reference

SQL Server 2012 Books Online

Summary: XQuery is a language that can query structured or semi-structured XML data. With the xml data type support provided in the Database Engine, documents can be stored in a database and then queried by using XQuery. XQuery is based on the existing XPath query language, with support added for better iteration, better sorting results, and the ability to construct the necessary XML.

Category: Reference
Applies to: SQL Server 2012
Source: SQL Server Books Online (link to source content)
E-book publication date: June 2012

Contents

XQuery Language Reference (SQL Server)

Transact-SQL supports a subset of the XQuery language that is used for querying the **xml** data type. This XQuery implementation is aligned with the July 2004 Working Draft of XQuery. The language is under development by the World Wide Web Consortium (W3C), with the participation of all major database vendors and also Microsoft. Because the W3C specifications may undergo future revisions before becoming a W3C recommendation, this implementation may be different from the final recommendation. This topic outlines the semantics and syntax of the subset of XQuery that is supported in SQL Server.

For more information, see the W3C XQuery 1.0 Language Specification.

XQuery is a language that can query structured or semi-structured XML data. With the **xml** data type support provided in the Database Engine, documents can be stored in a database and then queried by using XQuery.

XQuery is based on the existing XPath query language, with support added for better iteration, better sorting results, and the ability to construct the necessary XML. XQuery operates on the XQuery Data Model. This is an abstraction of XML documents, and the XQuery results that can be typed or untyped. The type information is based on the types provided by the W3C XML Schema language. If no typing information is available, XQuery handles the data as untyped. This is similar to how XPath version 1.0 handles XML.

To query an XML instance stored in a variable or column of **xml** type, you use the xml Data Type Methods. For example, you can declare a variable of **xml** type and query it by using the **query()** method of the **xml** data type.

```
DECLARE @x xml
SET @x = '<ROOT><a>111</a></ROOT>'
SELECT @x.query('/ROOT/a')
```

In the following example, the query is specified against the Instructions column of **xml** type in ProductModel table in the AdventureWorks database.

```
SELECT Instructions.query('declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
    /AWMI:root/AWMI:Location[@LocationID=10]
') as Result
FROM  Production.ProductModel
WHERE ProductModelID=7
```

The XQuery includes the namespace declaration, `declare namespace AWMI=...`, and the query expression, `/AWMI:root/AWMI:Location[@LocationID=10]`.

Note that the XQuery is specified against the Instructions column of **xml** type. The query() method of the xml data type is used to specify the XQuery.

The following table lists the related topics that can help in understanding the implementation of XQuery in the Database Engine.

Topic	Description
XML Data (SQL Server)	Explains the support for the **xml** data type in the Database Engine and the methods you can use against this data type. The **xml** data type forms the input XQuery data model on which the XQuery expressions are executed.
XML Schema Collections (SQL Server)	Describes how the XML instances stored in a database can be typed. This means you can associate an XML schema collection with the **xml** type column. All the instances stored in the column are validated and typed against the schema in the collection and provide the type information for XQuery.

Note

The organization of this section is based on the World Wide Web Consortium (W3C) XQuery working draft specification. Some of the diagrams provided in this section are taken from that specification. This section compares the Microsoft XQuery implementation to the W3C specification, describes how Microsoft XQuery is different from the W3C and indicates what W3C features are not supported. The W3C specification is available at http://www.w3.org/TR/2004/WD-xquery-20040723.

In This Section

Topic	Description
XQuery Basics	Provides a basic overview of XQuery concepts, and also the expression evaluation (static and dynamic context), atomization, effective Boolean value, XQuery type system, sequence type matching, and error handling.

XQuery Expressions	Describes XQuery primary expressions, path expressions, sequence expressions, arithmetic comparison and logical expressions, XQuery construction, FLWOR expression, conditional and quantified expressions, and various expressions on sequence types.
Modules and Prologs (XQuery)	Describes XQuery prolog.
XQuery Functions against the xml Data Type	Describes a list of the XQuery functions that are supported.
XQuery Operators Against the xml Data Type	Describes XQuery operators that are supported.
Additional Sample XQueries Against the xml Data Type	Provides additional XQuery samples.

See Also

XML Schema Collections (SQL Server)

XML Data (SQL Server)

Examples of Bulk Import and Export of XML Documents (SQL Server)

XQuery Basics

This section describes the fundamentals of XQuery.

In this Section

Sequence and QNames (XQuery)

Describes sequence and also QNames and predefined namespaces.

Expression Context and Query Evaluation (XQuery)

Describes the two contexts in which XQuery is evaluated. These two contexts are static and dynamic.

Atomization (XQuery)

Describes atomization, which is a process of extracting the typed value of an item.

Effective Boolean Value (XQuery)

Describes the effective Boolean value. This value can be computed for expressions that return a single Boolean value, a node sequence, or an empty sequence.

Type System (XQuery)

Describes the XQuery type system with various predefined types. XQuery is a strongly typed language for schema types and a weakly typed language for untyped data.

Error Handling (XQuery)

Describes the handling of static, dynamic, and type errors in XQuery.

Comments in XQuery

Describes how to add comments in XQuery by using the "(:" and ":)" delimiters

XQuery and static typing

Describes XQuery in SQL Server as a statically typed language

See Also

XQuery Against the XML Data Type

Sequence and QNames

This topic describes the following fundamental concepts of XQuery:

- Sequence
- QNames and predefined namespaces

Sequence

In XQuery, the result of an expression is a sequence that is made up of a list of XML nodes and instances of XSD atomic types. An individual entry in a sequence is referred to as an item. An item in a sequence can be either of the following:

- A node such as an element, attribute, text, processing instruction, comment, or document
- An atomic value such as an instance of an XSD simple type

For example, the following query constructs a sequence of two element-node items:

```
SELECT Instructions.query('
     <step1> Step 1 description goes here</step1>,
     <step2> Step 2 description goes here </step2>
') AS Result
FROM Production.ProductModel
```

```
WHERE ProductModelID=7;
```

This is the result:

```
<step1> Step 1 description goes here </step1>
<step2> Step 2 description goes here </step2>
```

In the previous query, the comma (,) at the end of the `<step1>` construction is the sequence constructor and is required. The white spaces in the results are added for illustration only and are included in all the example results in this documentation.

Following is additional information that you should know about sequences:

- If a query results in a sequence that contains another sequence, the contained sequence is flattened into the container sequence. For example, the sequence ((1,2, (3,4,5)),6) is flattened in the data model as (1, 2, 3, 4, 5, 6).

  ```
  DECLARE @x xml;
  SET @x = '';
  SELECT @x.query('(1,2, (3,4,5)),6');
  ```

- An empty sequence is a sequence that does not contain any item. It is represented as "()".
- A sequence with only one item can be treated as an atomic value, and vice versa. That is, (1) = 1.

In this implementation, the sequence must be homogeneous. That is, either you have a sequence of atomic values or a sequence of nodes. For example, the following are valid sequences:

```
DECLARE @x xml;
SET @x = '';
-- Expression returns a sequence of 1 text node (singleton).
SELECT @x.query('1');
-- Expression returns a sequence of 2 text nodes
SELECT @x.query('"abc", "xyz"');
-- Expression returns a sequence of one atomic value. data() returns
-- typed value of the node.
SELECT @x.query('data(1)');
-- Expression returns a sequence of one element node.
-- In the expression XML construction is used to construct an element.
SELECT @x.query('<x> {1+2} </x>');
```

The following query returns an error, because heterogeneous sequences are not supported.

```
SELECT @x.query('<x>11</x>, 22');
```

QName

Every identifier in an XQuery is a QName. A QName is made up of a namespace prefix and a local name. In this implementation, the variable names in XQuery are QNames and they cannot have prefixes.

Consider the following example in which a query is specified against an untyped **xml** variable:

```
DECLARE @x xml;
SET @x = '<Root><a>111</a></Root>';
SELECT @x.query('/Root/a');
```

In the expression (`/Root/a`), `Root` and `a` are QNames.

In the following example, a query is specified against a typed **xml** column. The query iterates over all <step> elements at the first workcenter location.

```
SELECT Instructions.query('
   declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
for $Step in /AWMI:root/AWMI:Location[1]/AWMI:step
      return
           string($Step)
') AS Result
FROM Production.ProductModel
WHERE ProductModelID=7;
```

In the query expression, note the following:

- `AWMI root`, `AWMI:Location`, `AWMI:step`, and `$Step` are all QNames. `AWMI` is a prefix, and `root`, `Location`, and `Step` are all local names.
- The `$step` variable is a QName and does not have a prefix.

The following namespaces are predefined for use with XQuery support in SQL Server.

Prefix	URI
xs	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance
xdt	http://www.w3.org/2004/07/xpath-datatypes
fn	http://www.w3.org/2004/07/xpath-functions

Prefix	URI
(no prefix)	urn:schemas-microsoft-com:xml-sql
sqltypes	http://schemas.microsoft.com/sqlserver/2004/sqltypes
xml	http://www.w3.org/XML/1998/namespace
(no prefix)	http://schemas.microsoft.com/sqlserver/2004/SOAP

Every database you create has the **sys** XML schema collection. It reserves these schemas so they can be accessed from any user-created XML schema collection.

Note
This implementation does not support the `local` prefix as described in the XQuery specification in http://www.w3.org/2004/07/xquery-local-functions.

See Also

XQuery Basics

Expression Context and Query Evaluation

The context of an expression is the information that is used to analyze and evaluate it. Following are the two phases in which XQuery is evaluated:

- **Static context** – This is the query compilation phase. Based on the information available, errors are sometimes raised during this static analysis of the query.
- **Dynamic context** – This is the query execution phase. Even if a query has no static errors, such as errors during query compilation, the query may return errors during its execution.

Static Context

Static context initialization refers to the process of putting together all the information for static analysis of the expression. As part of static context initialization, the following is completed:

- The **boundary white space** policy is set to strip. Therefore, the boundary white space is not preserved by the **any element** and **attribute** constructors in the query. For example:

```
declare @x xml
set @x=''
select @x.query('<a>  {"Hello"}  </a>,

    <b> {"Hello2"}  </b>')
```

This query returns the following result, because the boundary space is stripped away during parsing of the XQuery expression:

```
<a>Hello</a><b>Hello2</b>
```

- The prefix and the namespace binding are initialized for the following:
  - A set of predefined namespaces.
  - Any namespaces defined using WITH XMLNAMESPACES. For more information, see Managing XML Schema Collections on the Server).
  - Any namespaces defined in the query prolog. Note that the namespace declarations in the prolog may override the namespace declaration in the WITH XMLNAMESPACES. For example, in the following query, WITH XMLNAMESPACES declares a prefix (pd) that binds it to namespace (http://someURI). However, in the WHERE clause, the query prolog overrides the binding.

```
WITH XMLNAMESPACES ('http://someURI' AS pd)
SELECT ProductModelID, CatalogDescription.query('
    <Product
        ProductModelID= "{ sql:column("ProductModelID") }"
        />
') AS Result
FROM Production.ProductModel
WHERE CatalogDescription.exist('
    declare namespace
pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     /pd:ProductDescription[(pd:Specifications)]'
    ) = 1
```

  All these namespace bindings are resolved during static context initialization.
- If querying a typed **xml** column or variable, the components of the XML schema collection associated with the column or variable are imported into the static context. For more information, see Typed vs. Untyped XML.
- For every atomic type in the imported schemas, a casting function is also made available in the static context. This is illustrated in the following example. In this example, a query is specified against a typed **xml** variable. The XML schema collection associated with this variable defines an atomic type, myType. Corresponding to this type, a casting function, **myType()**, is available during the static analysis. The query expression (`ns:myType(0)`) returns a value of myType.

```
-- DROP XML SCHEMA COLLECTION SC
-- go
```

```
CREATE XML SCHEMA COLLECTION SC AS '<schema
xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="myNS" xmlns:ns="myNS"
xmlns:s="http://schemas.microsoft.com/sqlserver/2004/sqltypes">
      <import
namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
/>
      <simpleType name="myType">
            <restriction base="int">
             <enumeration value="0" />
              <enumeration value="1"/>
            </restriction>
      </simpleType>
      <element name="root" type="ns:myType"/>
</schema>'
go

DECLARE @var XML(SC)
SET @var = '<root xmlns="myNS">0</root>'
-- specify myType() casting function in the query
SELECT @var.query('declare namespace ns="myNS"; ns:myType(0)')
```

In the following example, the casting function for the **int** built-in XML type is specified in the expression.

```
declare @x xml
set @x = ''
select @x.query('xs:int(5)')
go
```

After the static context is initialized, the query expression is analyzed (compiled). The static analysis involves the following:

1. Query parsing.
2. Resolving the function and type names specified in the expression.
3. Static typing of the query. This makes sure that the query is type safe. For example, the following query returns a static error, because the **+** operator requires numeric primitive type arguments:

```
declare @x xml
```

```
set @x=''
SELECT @x.query('"x" + 4')
```

In the following example, the **value()** operator requires a singleton. As specified in the XML schema, there can be multiple <Elem> elements. Static analysis of the expression determines that it is not type safe and a static error is returned. To resolve the error, the expression must be rewritten to explicitly specify a singleton (`data(/x:Elem)[1]`).

```
DROP XML SCHEMA COLLECTION SC
go
CREATE XML SCHEMA COLLECTION SC AS '<schema
xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="myNS" xmlns:ns="myNS"
xmlns:s="http://schemas.microsoft.com/sqlserver/2004/sqltypes">
      <import
namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
/>
      <element name="Elem" type="string"/>
</schema>'
go

declare @x xml (SC)
set @x='<Elem xmlns="myNS">test</Elem><Elem
xmlns="myNS">test2</Elem>'
SELECT @x.value('declare namespace x="myNS";
data(/x:Elem)[1]','varchar(20)')
```

For more information, see XQuery and Static Typing.

Implementation Restrictions

Following are the limitations related to static context:

- XPath compatibility mode is not supported.
- For XML construction, only the strip construction mode is supported. This is the default setting. Therefore, the type of the constructed element node is of **xdt:untyped** type and the attributes are of **xdt:untypedAtomic** type.
- Only ordered ordering mode is supported.
- Only strip XML space policy is supported.
- Base URI functionality is not supported.
- **fn:doc()** is not supported.

- **fn:collection()** is not supported.
- XQuery static flagger is not provided.
- The collation associated with the **xml** data type is used. This collation is always set to the Unicode codepoint collation.

Dynamic Context

Dynamic context refers to the information that must be available at the time the expression is executed. In addition to the static context, the following information is initialized as part of dynamic context:

- The expression focus, such as the context item, context position, and context size, is initialized as shown in the following. Note that all these values can be overridden by the nodes() method.
  - The **xml** data type sets the context item, the node being processed, to the document node.
  - The context position, the position of the context item relative to the nodes being processed, is first set to 1.
  - The context size, the number of items in the sequence being processed, is first set to 1, because there is always one document node.

Implementation Restrictions

Following are the limitations related to dynamic context:

- The **Current date and time** context functions, **fn:current-date**, **fn:current-time**, and **fn:current-dateTime**, are not supported.
- The **implicit timezone** is fixed to UTC+0 and cannot be changed.
- The **fn:doc()** function is not supported. All queries are executed against **xml** type columns or variables.
- The **fn:collection()** function is not supported.

See Also

XQuery Basics

Typed vs. Untyped XML

Managing XML Schema Collections on the Server

Atomization

Atomization is the process of extracting the typed value of an item. This process is implied under certain circumstances. Some of the XQuery operators, such as arithmetic and comparison operators, depend on this process. For example, when you apply arithmetic operators directly to nodes, the typed value of a node is first retrieved by implicitly invoking the data function. This passes the atomic value as an operand to the arithmetic operator.

For example, the following query returns the total of the LaborHours attributes. In this case, **data()** is implicitly applied to the attribute nodes.

```
declare @x xml
set @x='<ROOT><Location LID="1" SetupTime="1.1" LaborHours="3.3" />
<Location LID="2" SetupTime="1.0" LaborHours="5" />
<Location LID="3" SetupTime="2.1" LaborHours="4" />
</ROOT>'
-- data() implicitly applied to the attribute node sequence.
SELECT @x.query('sum(/ROOT/Location/@LaborHours)')
```

Although not required, you can also explicitly specify the **data()** function:

```
SELECT @x.query('sum(data(ROOT/Location/@LaborHours))')
```

Another example of implicit atomization is when you use arithmetic operators. The **+** operator requires atomic values, and **data()** is implicitly applied to retrieve the atomic value of the LaborHours attribute. The query is specified against the Instructions column of the **xml** type in the ProductModel table. The following query returns the LaborHours attribute three times. In the query, note the following:

- In constructing the OrignialLaborHours attribute, atomization is implicitly applied to the singleton sequence returned by (`$WC/@LaborHours`). The typed value of the LaborHours attribute is assigned to OrignialLaborHours.
- In constructing the UpdatedLaborHoursV1 attribute, the arithmetic operator requires atomic values. Therefore, **data()** is implicitly applied to the LaborHours attribute that is returned by (`$WC/@LaborHours`). The atomic value 1 is then added to it. The construction of attribute UpdatedLaborHoursV2 shows the explicit application of **data()**, but is not required.

```
SELECT Instructions.query('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
for $WC in /AWMI:root/AWMI:Location[1]
        return
            <WC OriginalLaborHours = "{ $WC/@LaborHours }"
                UpdatedLaborHoursV1 = "{ $WC/@LaborHours + 1 }"
                UpdatedLaborHoursV2 = "{ data($WC/@LaborHours) + 1 }" >
            </WC>') as Result
FROM Production.ProductModel
where ProductModelID=7
```

This is the result:

```
<WC OriginalLaborHours="2.5"
    UpdatedLaborHoursV1="3.5"
    UpdatedLaborHoursV2="3.5" />
```

The atomization results in an instance of a simple type, an empty set, or a static type error.

Atomization also occurs in comparison expression parameters passed to functions, values returned by functions, **cast()** expressions, and ordering expressions passed in the order by clause.

See Also

XQuery Functions Against the xml Data Type

Comparison Expressions (XQuery)

XQuery Functions Against the XML Data Type

Effective Boolean Value

These are the effective Boolean values:

- False if the operand is an empty sequence or a Boolean false.
- Otherwise, the value is true.

The effective Boolean value can be computed for expressions that return a single Boolean value, a node sequence, or an empty sequence. Note that the Boolean value is computed implicitly when the following types of expressions are processed:

- Logical expressions
- The not function
- The WHERE clause of a FLWOR expression
- Conditional expressions
- QuantifiedeExpressions

Following is an example of an effective Boolean value. When the **if** expression is processed, the effective Boolean value of the condition is determined. Because `/a[1]` returns an empty sequence, the effective Boolean value is false. The result is returned as XML with one text node (false).

```
value is false

DECLARE @x XML

SET @x = '<b/>'

SELECT @x.query('if (/a[1]) then "true" else "false"')

go
```

In the following example, the effective Boolean value is true, because the expression returns a nonempty sequence.

```
DECLARE @x XML
SET @x = '<a/>'
SELECT @x.query('if (/a[1]) then "true" else "false"')
go
```

When querying typed **xml** columns or variables, you can have nodes of Boolean type. The **data()** in this case returns a Boolean value. If the query expression returns a Boolean true value, the effective Boolean value is true, as shown in the next example. The following is also illustrated in the example:

- An XML schema collection is created. The element <b> in the collection is of Boolean type.
- A typed **xml** variable is created and queried.
- The expression `data(/b[1])` returns a Boolean true value. Therefore, the effective Boolean value in this case is true.
- The expression `data(/b[2])` returns a Boolean false value. Therefore, the effective Boolean value in this case is false.

```
CREATE XML SCHEMA COLLECTION SC AS '
<schema xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="s" type="string"/>
      <element name="b" type="boolean"/>
</schema>'
go
DECLARE @x XML(SC)
SET @x = '<b>true</b><b>false</b>'
SELECT @x.query('if (data(/b[1])) then "true" else "false"')
SELECT @x.query('if (data(/b[2])) then "true" else "false"')
go
```

See Also

FLWOR Statement and Iteration (XQuery)

FLWOR Statement and Iteration (XQuery)

Type System

XQuery is a strongly-typed language for schema types and a weakly-typed language for untyped data. The predefined types of XQuery include the following:

- Built-in types of XML schema in the **http://www.w3.org/2001/XMLSchema** namespace.
- Types defined in the **http://www.w3.org/2004/07/xpath-datatypes** namespace.

This topic also describes the following:

- The typed value versus the string value of a node.
- The data function (XQuery) and the string function (XQuery).
- Matching the sequence type returned by an expression.

Built-in Types of XML Schema

The built-in types of XML schema have a predefined namespace prefix of xs. Some of these types include **xs:integer** and **xs:string**. All these built-in types are supported. You can use these types when you create an XML schema collection.

When querying typed XML, the static and dynamic type of the nodes is determined by the XML schema collection associated with the column or variable that is being queried. For more information about static and dynamic types, see Expression Context and Query Evaluation (XQuery). For example, the following query is specified against a typed **xml** column (`Instructions`). The expression uses `instance of` to verify that the typed value of the `LotSize` attribute returned is of `xs:decimal` type.

```
SELECT Instructions.query('
   DECLARE namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
   data(/AWMI:root[1]/AWMI:Location[@LocationID=10][1]/@LotSize)[1]
instance of xs:decimal
') AS Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

This typing information is provided by the XML schema collection associated with the column.

Types Defined in XPath Data Types Namespace

The types defined in the **http://www.w3.org/2004/07/xpath-datatypes** namespace have a predefined prefix of **xdt**. The following applies to these types:

- You cannot use these types when you are creating an XML schema collection. These types are used in the XQuery type system and are used for static typing. You can cast to the atomic types, for example, **xdt:untypedAtomic**, in the **xdt** namespace.
- When querying untyped XML, the static and dynamic type of element nodes is **xdt:untyped**, and the type of attribute values is **xdt:untypedAtomic**. The result of a

query() method generates untyped XML. This means that the XML nodes are returned as **xdt:untyped** and **xdt:untypedAtomic**, respectively.

- The **xdt:dayTimeDuration** and **xdt:yearMonthDuration** types are not supported.

In the following example, the query is specified against an untyped XML variable. The expression, `data(/a[1])`, returns a sequence of one atomic value. The `data()` function returns the typed value of the element `<a>`. Because the XML being queried is untyped, the type of the value returned is `xdt:untypedAtomic`. Therefore, `instance of` returns true.

```
DECLARE @x xml
SET @x='<a>20</a>'
SELECT @x.query( 'data(/a[1]) instance of xdt:untypedAtomic' )
```

Instead of retrieving the typed value, the expression (`/a[1]`) in the following example returns a sequence of one element, element `<a>`. The `instance of` expression uses the element test to verify that the value returned by the expression is an element node of `xdt:untyped type`.

```
DECLARE @x xml
SET @x='<a>20</a>'
-- Is this an element node whose name is "a" and type is xdt:untyped.
SELECT @x.query( '/a[1] instance of element(a, xdt:untyped?)')
-- Is this an element node of type xdt:untyped.
SELECT @x.query( '/a[1] instance of element(*, xdt:untyped?)')
-- Is this an element node?
SELECT @x.query( '/a[1] instance of element()')
```

Note

When you are querying a typed XML instance and the query expression includes the parent axis, the static type information of the resulting nodes is no longer available. However, the dynamic type is still associated with the nodes.

Typed Value vs. String Value

Every node has a typed value and a string value. For typed XML data, the type of the typed value is provided by the XML schema collection associated with the column or variable that is being queried. For untyped XML data, the type of the typed value is **xdt:untypedAtomic**.

You can use the **data()** or **string()** function to retrieve the value of a node:

- The data function returns the typed value of a node.
- The string function returns the string value of the node.

In the following XML schema collection, the `<root>` element of the integer type is defined:

```
CREATE XML SCHEMA COLLECTION SC AS N'
<schema xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="root" type="integer"/>
</schema>'
GO
```

In the following example, the expression first retrieves the typed value of `/root[1]` and then adds `3` to it.

```
DECLARE @x xml(SC)
SET @x='<root>5</root>'
SELECT @x.query('data(/root[1]) + 3')
```

In the next example, the expression fails, because the `string(/root[1])` in the expression returns a string type value. This value is then passed to an arithmetic operator that takes only numeric type values as its operands.

```
-- Fails because the argument is string type (must be numeric primitive
type).
DECLARE @x xml(SC)
SET @x='<root>5</root>'
SELECT @x.query('string(/root[1]) + 3')
```

The following example computes the total of the `LaborHours` attributes. The `data()` function retrieves the typed values of `LaborHours` attributes from all the `<Location>` elements for a product model. According to the XML schema associated with the `Instruction` column, `LaborHours` is of **xs:decimal** type.

```
SELECT Instructions.query('
DECLARE namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
             sum(data(//AWMI:Location/@LaborHours))
') AS Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

This query returns 12.75 as the result.

Note

The explicit use of the **data()** function in this example is for illustration only. If it is not specified, **sum()** implicitly applies the **data()** function to extract the typed values of the nodes.

See Also

Using SQL Server Profiler
XQuery Basics

Sequence Type Matching

An XQuery expression value is always a sequence of zero or more items. An item can be either an atomic value or a node. The sequence type refers to the ability to match the sequence type returned by a query expression with a specific type. For example:

- If the expression value is atomic, you may want to know if it is an integer, decimal, or string type.
- If the expression value is an XML node, you may want to know if it is a comment node, a processing instruction node, or a text node.
- You may want to know if the expression returns an XML element or an attribute node of a specific name and type.

You can use the `instance of` Boolean operator in sequence type matching. For more information about the `instance of` expression, see SequenceType Expressions (XQuery).

Comparing the Atomic Value Type Returned by an Expression

If an expression returns a sequence of atomic values, you may have to find the type of the value in the sequence. The following examples illustrate how sequence type syntax can be used to evaluate the atomic value type returned by an expression.

Example: Determining whether a sequence is empty

The **empty()** sequence type can be used in a sequence type expression to determine whether the sequence returned by the specified expression is an empty sequence.

In the following example, the XML schema allows the `<root>` element to be nilled:

```
CREATE XML SCHEMA COLLECTION SC AS N'
<schema xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="root" nillable="true" type="byte"/>
</schema>'
GO
```

Now, if a typed XML instance specifies a value for the `<root>` element, `instance of empty()` returns False.

```
DECLARE @var XML(SC1)
SET @var = '<root>1</root>'
-- The following returns False
SELECT @var.query('data(/root[1]) instance of  empty() ')
GO
```

If the `<root>` element is nilled in the instance, its value is an empty sequence and the `instance of empty()` returns True.

```
DECLARE @var XML(SC)
SET @var = '<root xsi:nil="true"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" />'
SELECT @var.query('data(/root[1]) instance of  empty() ')
GO
```

Example: Determining the type of an attribute value

Sometimes, you may want to evaluate the sequence type returned by an expression before processing. For example, you may have an XML schema in which a node is defined as a union type. In the following example, the XML schema in the collection defines the attribute `a` as a union type whose value can be of decimal or string type.

```
-- Drop schema collection if it exists.
-- DROP XML SCHEMA COLLECTION SC.
-- GO
CREATE XML SCHEMA COLLECTION SC AS N'
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="root">
    <complexType>
       <sequence/>
         <attribute name="a">
            <simpleType>
               <union memberTypes="decimal string"/>
            </simpleType>
         </attribute>
     </complexType>
  </element>
</schema>'
GO
```

Before processing a typed XML instance, you may want to know the type of the attribute `a` value. In the following example, the attribute `a` value is a decimal type. Therefore, `instance of xs:decimal` returns True.

```
DECLARE @var XML(SC)
SET @var = '<root a="2.5"/>'
SELECT @var.query('data((/root/@a)[1]) instance of xs:decimal')
```

```
GO
```

Now, change the attribute `a` value to a string type. The `instance of xs:string` will return True.

```
DECLARE @var XML(SC)
SET @var = '<root a="Hello"/>'
SELECT @var.query('data((/root/@a)[1]) instance of xs:string')
GO
```

Example: Cardinality in sequence expressions

This example illustrates the effect of cardinality in a sequence expression. The following XML schema defines a `<root>` element that is of byte type and is nillable.

```
CREATE XML SCHEMA COLLECTION SC AS N'
<schema xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="root" nillable="true" type="byte"/>
</schema>'
GO
```

In the following query, because the expression returns a singleton of byte type, `instance of` returns True.

```
DECLARE @var XML(SC)
SET @var = '<root>111</root>'
SELECT @var.query('data(/root[1]) instance of  xs:byte ')
GO
```

If you make the `<root>` element nil, its value is an empty sequence. That is, the expression, `/root[1]`, returns an empty sequence. Therefore, `instance of xs:byte` returns False. Note that the default cardinality in this case is 1.

```
DECLARE @var XML(SC)
SET @var = '<root xsi:nil="true"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"></root>'
SELECT @var.query('data(/root[1]) instance of  xs:byte ')
GO
-- result = false
```

If you specify cardinality by adding the occurrence indicator (`?`), the sequence expression returns True.

```
DECLARE @var XML(SC)
SET @var = '<root xsi:nil="true"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"></root>'
```

```
SELECT @var.query('data(/root[1]) instance of  xs:byte? ')
GO
-- result = true
```

Note that the testing in a sequence type expression is completed in two stages:

1. First, the testing determines whether the expression type matches the specified type.
2. If it does, the testing then determines whether the number of items returned by the expression matches the occurrence indicator specified.

If both are true, the `instance of` expression returns True.

Example: Querying against an xml type column

In the following example, a query is specified against an Instructions column of **xml** type in the database. It is a typed XML column because it has a schema associated with it. The XML schema defines the `LocationID` attribute of the integer type. Therefore, in the sequence expression, the `instance of xs:integer?` returns True.

```
SELECT Instructions.query('
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
data(/AWMI:root[1]/AWMI:Location[1]/@LocationID) instance of
xs:integer?') as Result
FROM Production.ProductModel
WHERE ProductModelID = 7
```

Comparing the Node Type Returned by an Expression

If an expression returns a sequence of nodes, you may have to find the type of the node in the sequence. The following examples illustrate how sequence type syntax can be used to evaluate the node type returned by an expression. You can use the following sequence types:

- **item()** – Matches any item in the sequence.
- **node()** – Determines whether the sequence is a node.
- **processing-instruction()** – Determines whether the expression returns a processing instruction.
- **comment()** – Determines whether the expression returns a comment.
- **document-node()** – Determines whether the expression returns a document node.

The following example illustrates these sequence types.

Example: Using sequence types

In this example, several queries are executed against an untyped XML variable. These queries illustrate the use of sequence types.

```
DECLARE @var XML
```

```
SET @var = '<?xml-stylesheet href="someValue" type="text/xsl" ?>
<root>text node
  <!-- comment 1 -->
  <a>Data a</a>
  <!-- comment  2 -->
</root>'
```

In the first query, the expression returns the typed value of element `<a>`. In the second query, the expression returns element `<a>`. Both are items. Therefore, both queries return True.

```
SELECT @var.query('data(/root[1]/a[1]) instance of item()')
SELECT @var.query('/root[1]/a[1] instance of item()')
```

All the XQuery expressions in the following three queries return the element node child of the `<root>` element. Therefore, the sequence type expression, `instance of node()`, returns True, and the other two expressions, `instance of text()` and `instance of document-node()`, return False.

```
SELECT @var.query('(/root/*)[1] instance of node()')
SELECT @var.query('(/root/*)[1] instance of text()')
SELECT @var.query('(/root/*)[1] instance of document-node()')
```

In the following query, the `instance of document-node()` expression returns True, because the parent of the `<root>` element is a document node.

```
SELECT @var.query('(/root/..)[1] instance of document-node()') -- true
```

In the following query, the expression retrieves the first node from the XML instance. Because it is a processing instruction node, the `instance of processing-instruction()` expression returns True.

```
SELECT @var.query('(/node())[1] instance of processing-instruction()')
```

Implementation Limitations

These are the specific limitations:

- **document-node()** with content type syntax is not supported.
- **processing-instruction(name)** syntax is not supported.

Element Tests

An element test is used to match the element node returned by an expression to an element node with a specific name and type. You can use these element tests:

```
element ()
element(ElementName)
element(ElementName, ElementType?)
element(*, ElementType?)
```

Attribute Tests

The attribute test determines whether the attribute returned by an expression is an attribute node. You can use these attribute tests.

```
attribute()
attribute(AttributeName)
attribute(AttributeName, AttributeType)
```

Test Examples

The following examples illustrate scenarios in which element and attribute tests are useful.

Example A

The following XML schema defines the `CustomerType` complex type where `<firstName>` and `<lastName>` elements are optional. For a specified XML instance, you may have to determine whether the first name exists for a particular customer.

```
CREATE XML SCHEMA COLLECTION SC AS N'
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="myNS" xmlns:ns="myNS">
  <complexType name="CustomerType">
     <sequence>
        <element name="firstName" type="string" minOccurs="0"
                  nillable="true" />
        <element name="lastName" type="string" minOccurs="0"/>
     </sequence>
  </complexType>
  <element name="customer" type="ns:CustomerType"/>
</schema>
'
GO
DECLARE @var XML(SC)
SET @var = '<x:customer xmlns:x="myNS">
<firstName>SomeFirstName</firstName>
<lastName>SomeLastName</lastName>
</x:customer>'
```

The following query uses an `instance of element (firstName)` expression to determine whether the first child element of `<customer>` is an element whose name is `<firstName>`. In this case, it returns True.

```
SELECT @var.query('declare namespace x="myNS";
     (/x:customer/*)[1] instance of element (firstName)')
GO
```

If you remove the <firstName> element from the instance, the query will return False.

You can also use the following:

- The element(ElementName, ElementType?) sequence type syntax, as shown in the following query. It matches a nilled or non-nilled element node whose name is firstName and whose type is xs:string.

```
SELECT @var.query('declare namespace x="myNS";
(/x:customer/*)[1] instance of element (firstName, xs:string?)')
```

- The element(*, type?) sequence type syntax, as shown in the following query. It matches the element node if its type is xs:string, regardless of its name.

```
SELECT @var.query('declare namespace x="myNS";
(/x:customer/*)[1] instance of element (*, xs:string?)')
GO
```

Example B

The following example illustrates how to determine whether the node returned by an expression is an element node with a specific name. It uses the **element()** test.

In the following example, the two <Customer> elements in the XML instance that are being queried are of two different types, CustomerType and SpecialCustomerType. Assume that you want to know the type of the <Customer> element returned by the expression. The following XML schema collection defines the CustomerType and SpecialCustomerType types.

```
CREATE XML SCHEMA COLLECTION SC AS N'
<schema xmlns="http://www.w3.org/2001/XMLSchema"
          targetNamespace="myNS"  xmlns:ns="myNS">
  <complexType name="CustomerType">
    <sequence>
      <element name="firstName" type="string"/>
      <element name="lastName" type="string"/>
    </sequence>
  </complexType>
  <complexType name="SpecialCustomerType">
     <complexContent>
       <extension base="ns:CustomerType">
        <sequence>
```

```
            <element name="Age" type="int"/>
         </sequence>
        </extension>
      </complexContent>
     </complexType>
    <element name="customer" type="ns:CustomerType"/>
</schema>
'
GO
```

This XML schema collection is used to create a typed **xml** variable. The XML instance assigned to this variable has two `<customer>` elements of two different types. The first element is of `CustomerType` and the second element is of `SpecialCustomerType` type.

```
DECLARE @var XML(SC)
SET @var = '
<x:customer xmlns:x="myNS">
   <firstName>FirstName1</firstName>
   <lastName>LastName1</lastName>
</x:customer>
<x:customer xsi:type="x:SpecialCustomerType" xmlns:x="myNS"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <firstName> FirstName2</firstName>
   <lastName> LastName2</lastName>
   <Age>21</Age>
</x:customer>'
```

In the following query, the `instance of element (*, x:SpecialCustomerType ?)` expression returns False, because the expression returns the first customer element that is not of `SpecialCustomerType` type.

```
SELECT @var.query('declare namespace x="myNS";
    (/x:customer)[1] instance of element (*, x:SpecialCustomerType ?)')
```

If you change the expression of the previous query and retrieve the second `<customer>` element (`/x:customer)[2]`), the `instance of` will return True.

Example C

This example uses the attribute test. The following XML schema defines the CustomerType complex type with CustomerID and Age attributes. The Age attribute is optional. For a specific XML instance, you may want to determine whether the Age attribute is present in the `<customer>` element.

```
CREATE XML SCHEMA COLLECTION SC AS N'
<schema xmlns="http://www.w3.org/2001/XMLSchema"
       targetNamespace="myNS" xmlns:ns="myNS">
<complexType name="CustomerType">
  <sequence>
     <element name="firstName" type="string" minOccurs="0"
               nillable="true" />
     <element name="lastName" type="string" minOccurs="0"/>
  </sequence>
  <attribute name="CustomerID" type="integer" use="required" />
  <attribute name="Age" type="integer" use="optional" />
 </complexType>
 <element name="customer" type="ns:CustomerType"/>
</schema>
'
GO
```

The following query returns True, because there is an attribute node whose name is `Age` in the XML instance that is being queried. The `attribute(Age)` attribute test is used in this expression. Because attributes have no order, the query uses the FLWOR expression to retrieve all the attributes and then test each attribute by using the `instance of` expression. The example first creates an XML schema collection to create a typed **xml** variable.

```
DECLARE @var XML(SC)
SET @var = '<x:customer xmlns:x="myNS" CustomerID="1" Age="22" >
<firstName>SomeFName</firstName>
<lastName>SomeLName</lastName>
</x:customer>'
SELECT @var.query('declare namespace x="myNS";
FOR $i in /x:customer/@*
RETURN
    IF ($i instance of attribute (Age)) THEN
        "true"
        ELSE
        ()')
GO
```

If you remove the optional `Age` attribute from the instance, the previous query will return False.

You can specify attribute name and type (`attribute(name,type)`) in the attribute test.

```
SELECT @var.query('declare namespace x="myNS";

FOR $i in /x:customer/@*

RETURN

    IF ($i instance of attribute (Age, xs:integer)) THEN

        "true"

        ELSE

        ()')
```

Alternatively, you can specify the `attribute(*, type)` sequence type syntax. This matches the attribute node if the attribute type matches the specified type, regardless of the name.

Implementation Limitations

These are the specific limitations:

- In the element test, the type name must be followed by the occurrence indicator (**?**).
- **element(ElementName, TypeName)** is not supported.
- **element(*, TypeName)** is not supported.
- **schema-element()** is not supported.
- **schema-attribute(AttributeName)** is not supported.
- Explicitly querying for **xsi:type** or **xsi:nil** is not supported.

See Also

Type System (XQuery)

Error Handling

The W3C specification allows type errors to be raised statically or dynamically, and defines static, dynamic, and type errors.

Compilation and Error Handling

Compilation errors are returned from syntactically incorrect Xquery expressions and XML DML statements. The compilation phase checks static type correctness of XQuery expressions and DML statements, and uses XML schemas for type inferences for typed XML. It raises static type errors if an expression could fail at run time because of a type safety violation. Examples of static error are the addition of a string to an integer and querying for a nonexistent node for typed data.

As a deviation from the W3C standard, XQuery run-time errors are converted into empty sequences. These sequences may propagate as empty XML or NULL to the query result, depending upon the invocation context.

Explicit casting to the correct type allows users to work around static errors, although run-time cast errors will be transformed to empty sequences.

Static Errors

Static errors are returned by using the Transact-SQL error mechanism. In SQL Server, XQuery type errors are returned statically. For more information, see XQuery and Static Typing.

Dynamic Errors

In XQuery, most dynamic errors are mapped to an empty sequence ("()"). However, these are the two exceptions: Overflow conditions in XQuery aggregator functions and XML-DML validation errors. Note that most dynamic errors are mapped to an empty sequence. Otherwise, query execution that takes advantages of the XML indexes may raise unexpected errors. Therefore, to provide an efficient execution without generating unexpected errors, SQL Server Database Engine maps dynamic errors to ().

Frequently, in the situation where the dynamic error would occur inside a predicate, not raising the error is not changing the semantics, because () is mapped to False. However, in some cases, returning () instead of a dynamic error may cause unexpected results. The following are examples that illustrate this.

Example: Using the avg() Function with a String

In the following example, the avg function is called to compute the average of the three values. One of these values is a string. Because the XML instance in this case is untyped, all the data in it is of untyped atomic type. The **avg()** function first casts these values to **xs:double** before computing the average. However, the value, `"Hello"`, cannot be cast to **xs:double** and creates a dynamic error. In this case, instead of returning a dynamic error, the casting of `"Hello"` to **xs:double** causes an empty sequence. The **avg()** function ignores this value, computes the average of the other two values, and returns 150.

```
DECLARE @x xml
SET @x=N'<root xmlns:myNS="test">
 <a>100</a>
 <b>200</b>
 <c>Hello</c>
</root>'
SELECT @x.query('avg(//*)')
```

Example: Using the not Function

When you use the not function in a predicate, for example, `/SomeNode[not(Expression)]`, and the expression causes a dynamic error, an empty sequence will be returned instead of an error. Applying **not()** to the empty sequence returns True, instead of an error.

Example: Casting a String

In the following example, the literal string "NaN" is cast to xs:string, then to xs:double. The result is an empty rowset. Although the string "NaN" cannot successfully be cast to xs:double, this cannot be determined until runtime because the string is first cast to xs:string.

```
DECLARE @x XML
SET @x = ''
SELECT @x.query(' xs:double(xs:string("NaN")) ')
GO
```

In this example, however, a static type error occurs.

```
DECLARE @x XML
SET @x = ''
SELECT @x.query(' xs:double("NaN") ')
GO
```

Implementation Limitations

The **fn:error()** function is not supported.

See Also

XQuery Against the XML Data Type

XQuery Basics

Comments in XQuery

You can add comments to XQuery. The comment strings are added by using the "`(:`" and "`:)`" delimiters. For example:

```
declare @x xml
set @x=''
SELECT @x.query('
(: simple query to construct an element :)
<ProductModel ProductModelID="10" />
')
```

Following is another example in which a query is specified against an Instruction column of the **xml** type:

```
SELECT Instructions.query('
(: declare prefix and namespace binding in the prolog. :)
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
  (: Following expression retrieves the <Location> element children of
the <root> element. :)
  /AWMI:root/AWMI:Location
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

XQuery and Static Typing

XQuery in SQL Server is a statically typed language. That is, it raises type errors during query compilation when an expression returns a value that has a type or cardinality that is not accepted by a particular function or operator. Additionally, static type checking can also detect if a path expression on a typed XML document has been mistyped. The XQuery compiler first applies the normalization phase that adds the implicit operations, such as atomization, and then performs static type inference and static type checking.

Static Type Inference

Static type inference determines the return type of an expression. It determines this by taking the static types of the input parameters and the static semantics of the operation and inferring the static type of the result. For example, the static type of the expression 1 + 2.3 is determined in the following way:

- The static type of 1 is **xs:integer** and the static type of 2.3 is **xs:decimal**. Based on the dynamic semantics, the static semantics of the **+** operation converts the integer to a decimal and then returns a decimal. The inferred static type would then be **xs:decimal**.

For untyped XML instances, there are special types to indicate that the data was not typed. This information is used during static type checking and to perform certain implicit casts).

For typed data, the input type is inferred from the XML schema collection that constrains the XML data type instance. For example, if the schema allows only elements of type **xs:integer**, the results of a path expression using that element will be zero or more elements of type **xs:integer**. This is currently expressed by using an expression such as `element(age,xs:integer)*` where the asterisk (*) indicates the cardinality of the resulting type. In this example, the expression may result in zero or more elements of name "age" and type **xs:integer**. Other cardinalities are exactly one and are expressed by

using the type name alone, zero or one and expressed by using a question mark (**?**), and 1 or more and expressed by using a plus sign (**+**).

Sometimes, the static type inference can infer that an expression will always return the empty sequence. For example, if a path expression on a typed XML data type looks for a <name> element inside a <customer> element (/customer/name), but the schema does not allow a <name> inside a <customer>, the static type inference will infer that the result will be empty. This will be used to detect incorrect queries and will be reported as a static error, unless the expression was () or **data(())**.

The detailed inference rules are provided in the formal semantics of the XQuery specification. Microsoft has modified these only slightly to work with typed XML data type instances. The most important change from the standard is that the implicit document node knows about the type of the XML data type instance. As a result, a path expression of the form /age will be precisely typed based on that information.

By using SQL Server Profiler, you can see the static types returned as part of query compilations. To see these, your trace must include the XQuery Static Type event in the TSQL event category.

Static Type Checking

Static type checking ensures that the run-time execution will only receive values that are the appropriate type for the operation. Because the types do not have to be checked at run time, potential errors can be detected early in the compilation. This helps improve performance. However, static typing requires that the query writer be more careful in formulating a query.

Following are the appropriate types that can be used:

- Types explicitly allowed by a function or operation.
- A subtype of an explicitly allowed type.

Subtypes are defined, based on the subtyping rules for using derivation by restriction or extension of the XML schema. For example, a type S is a subtype of type T, if all the values that have the type S are also instances of the type T.

Additionally, all integer values are also decimal values, based on the XML schema type hierarchy. However, not all decimal values are integers. Therefore, an integer is a subtype of decimal, but not vice versa. For example, the **+** operation only allows values of certain types, such as the numeric types **xs:integer**, **xs:decimal**, **xs:float**, and **xs:double**. If values of other types, such as **xs:string**, are passed, the operation raises a type error. This is referred to as strong typing. Values of other types, such as the atomic type used to indicate untyped XML, can be implicitly converted to a value of a type that the operation accepts. This is referred to as weak typing.

If it is required after an implicit conversion, static type checking guarantees that only values of the allowed types with the correct cardinality are passed to an operation. For "string" + 1, it recognizes that the static type of "string" is **xs:string**. Because this is not an allowed type for the **+** operation, a type error is raised.

In the case of adding the result of an arbitrary expression E1 to an arbitrary expression E2 (E1 + E2), static type inference first determines the static types of E1 and E2 and then checks their static types with the allowed types for the operation. For example, if the static type of E1 can be either an **xs:string** or an **xs:integer**, the static type check raises a type error, even though some values at run time might be integers. The same would be the case if the static type of E1 were **xs:integer***. Because the **+** operation only accepts exactly one integer value and E1 could return zero or more than 1, the static type check raises an error.

As mentioned earlier, type inference frequently infers a type that is broader than what the user knows about the type of the data that is being passed. In these cases, the user has to rewrite the query. Some typical cases include the following:

- The type infers a more general type such as a supertype or a union of types. If the type is an atomic type, you should use the cast expression or constructor function to indicate the actual static type. For example, if the inferred type of the expression E1 is a choice between **xs:string** or **xs:integer** and the addition requires **xs:integer**, you should write `xs:integer(E1) + E2` instead of `E1+E2`. This expression may fail at run time if a string value is encountered that cannot be cast to **xs:integer**. However, the expression will now pass the static type check. Beginning with SQL Server 2005, this expression is mapped to the empty sequence.
- The type infers a higher cardinality than what the data actually contains. This occurs frequently, because the **xml** data type can contain more than one top-level element, and an XML schema collection cannot constrain this. In order to reduce the static type and guarantee that there is indeed at most one value being passed, you should use the positional predicate `[1]`. For example, to add 1 to the value of the attribute `c` of the element `b` under the top-level a element, you must `write (/a/b/@c)[1]+1`. Additionally, the DOCUMENT keyword can be used together with an XML schema collection.
- Some operations lose type information during inference. For example, if the type of a node cannot be determined, it becomes **anyType**. This is not implicitly cast to any other type. These conversions occur most notably during navigation by using the parent axis. You should avoid using such operations and rewrite the query, if the expression will create a static type error.

Type Checking of Union Types

Union types require careful handling because of type checking. Two of the problems are illustrated in the following examples.

Example: Function over Union Type

Consider an element definition for `<r>` of a union type:

```
<xs:element name="r">
<xs:simpleType>
   <xs:union memberTypes="xs:int xs:float xs:double"/>
```

```
</xs:simpleType>
</xs:element>
```

Within XQuery context, the "average" function `fn:avg (//r)` returns a static error, because the XQuery compiler cannot add values of different types (**xs:int**, **xs:float** or **xs:double**) for the `<r>` elements in the argument of **fn:avg()**. To solve this, rewrite the function invocation as `fn:avg(for $r in //r return $r cast as xs:double ?)`.

Example: Operator over Union Type

The addition operation ('+') requires precise types of the operands. As a result, the expression `(//r)[1] + 1` returns a static error that has the previously described type definition for element `<r>`. One solution is to rewrite it as `(//r)[1] cast as xs:int? +1`, where the "?" indicates 0 or 1 occurrences. Beginning with SQL Server 2005, SQL Server requires "cast as" with "?", because any cast can cause the empty sequence as a result of run-time errors.

See Also

XQuery Against the XML Data Type

XQuery Expressions

This topic describes XQuery expressions.

In This Section

XQuery Language Reference (Database Engine)

Describes XQuery primary expressions. These include literals, variable references, context item expressions, constructors, and function calls.

Path Expressions (XQuery)

Describes XQuery path expressions. These locate nodes, such as element, attribute, and text, in a document.

Sequence Expressions (XQuery)

Describes XQuery operators to work with sequences of numbers.

Arithmetic Expressions (XQuery)

Describes using arithmetic expressions in XQuery.

Comparison Expressions (XQuery)

Describes the comparison expressions supported by XQuery. That is, the general, value, node comparison, and node type comparison expressions.

Logical Expressions (XQuery)

Describes XQuery support for the logical **and** and **or** operators.

XML Construction (XQuery)

Describes XQuery constructors that allow you to construct XML within a query.

FLWOR Statement and Iteration (XQuery)

Describes the FLOWR iteration syntax. This stands for FOR, LET, WHERE, ORDER BY, and RETURN. LET is not supported.

Ordered and Unordered Expressions (XQuery)

Describes the ordering mode for XQuery operations. By default, the ordering mode is set to **ordered**.

Conditional Expressions (XQuery)

Describes XQuery support for the conditional **if-then-else** statement.

Quantified Expressions (XQuery)

Describes the existential and universal quantifiers in XQuery.

SequenceType Expressions (XQuery)

Describes the SequenceType syntax in XQuery.

Validate Expressions (XQuery)

The **validate** expression is not supported.

See Also

XQuery Against the XML Data Type

Primary Expressions

The XQuery primary expressions include literals, variable references, context item expressions, constructors, and function calls.

Literals

XQuery literals can be numeric or string literals. A string literal can include predefined entity references, and an entity reference is a sequence of characters. The sequence starts with an ampersand that represents a single character that otherwise might have syntactic significance. Following are the predefined entity references for XQuery.

Entity reference	Represents
&lt;	<
&gt;	>

Entity reference	Represents
&amp;	&
&quot;	"
&apos;	'

A string literal can also contain a character reference, an XML-style reference to a Unicode character, that is identified by its decimal or hexadecimal code point. For example, the Euro symbol can be represented by the character reference, "€".

Note
SQL Server uses XML version 1.0 as the basis for parsing.

Examples

The following examples illustrate the use of literals and also entity and character references.

This code returns an error, because the <' and '> characters have special meaning.

```
DECLARE @var XML
SET @var = ''
SELECT @var.query(' <SalaryRange>Salary > 50000 and <
100000</SalaryRange>')
GO
```

If you use an entity reference instead, the query works.

```
DECLARE @var XML
SET @var = ''
SELECT @var.query(' <SalaryRange>Salary &gt; 50000 and &lt;
100000</SalaryRange>')
GO
```

The following example illustrates the use of a character reference to represent the Euro symbol.

```
DECLARE @var XML
SET @var = ''
SELECT @var.query(' <a>&#8364;12.50</a>')
```

This is the result.

```
<a>€12.50</a>
```

In the following example, the query is delimited by apostrophes. Therefore, the apostrophe in the string value is represented by two adjacent apostrophes.

```
DECLARE @var XML
SET @var = ''
SELECT @var.query('<a>I don''t know</a>')
Go
```

This is the result.

```
<a>I don't know</a>
```

The built-in Boolean functions, **true()** and **false()**, can be used to represent Boolean values, as shown in the following example.

```
DECLARE @var XML
SET @var = ''
SELECT @var.query('<a>{true()}</a>')
GO
```

The direct element constructor specifies an expression in curly braces. This is replaced by its value in the resulting XML.

This is the result.

```
<a>true</a>
```

Variable References

A variable reference in XQuery is a QName preceded by a $ sign. This implementation supports only unprefixed variable references. For example, the following query defines the variable `$i` in the FLWOR expression.

```
DECLARE @var XML
SET @var = '<root>1</root>'
SELECT @var.query('
 for $i in /root return data($i)')
GO
```

The following query will not work because a namespace prefix is added to the variable name.

```
DECLARE @var XML
SET @var = '<root>1</root>'
SELECT @var.query('
DECLARE namespace x="http://X";
for $x:i in /root return data($x:i)')
GO
```

You can use the **sql:variable()** extension function to refer to SQL variables, as shown in the following query.

```
DECLARE @price money
SET @price=2500
DECLARE @x xml
SET @x = ''
SELECT @x.query('<value>{sql:variable("@price") }</value>')
```

This is the result.

```
<value>2500</value>
```

Implementation Limitations

These are the implementation limitations:

- Variables with namespace prefixes are not supported.
- Module import is not supported.
- External variable declarations are not supported. A solution to this is to use the sql:variable() function.

Context Item Expressions

The context item is the item currently being processed in the context of a path expression. It is initialized in a not-NULL XML data type instance with the document node. It can also be changed by the **nodes()** method, in the context of XPath expressions or the [] predicates.

The context item is returned by an expression that contains a dot (.). For example, the following query evaluates each element `<a>` for the presence of attribute `attr`. If the attribute is present, the element is returned. Note that the condition in the predicate specifies that the context node is specified by single period.

```
DECLARE @var XML
SET @var = '<ROOT>
<a>1</a>
<a attr="1">2</a>
</ROOT>'
SELECT @var.query('/ROOT[1]/a[./@attr]')
```

This is the result.

```
<a attr="1">2</a>
```

Function Calls

You can call built-in XQuery functions and the SQL Server **sql:variable()** and **sql:column()** functions. For a list of implemented functions, see XML Construction (XQuery).

Implementation Limitations

These are the implementation limitations:

- Function declaration in the XQuery prolog is not supported.
- Function import is not supported.

See Also

XML Construction (XQuery)

Path Expressions

XQuery path expressions locate nodes, such as element, attribute, and text nodes, in a document. The result of a path expression always occurs in document order without duplicate nodes in the result sequence. In specifying a path, you can use either unabbreviated or abbreviated syntax. The following information focuses on the unabbreviated syntax. Abbreviated syntax is described later in this topic.

Note

Because the sample queries in this topic are specified against the **xml** type columns, **CatalogDescription** and **Instructions**, in the **ProductModel** table, you should familiarize yourself with the contents and structure of the XML documents stored in these columns.

A path expression can be relative or absolute. Following is a description of both of these:

- A relative path expression is made up of one or more steps separated by one or two slash marks (/ or //). For example, `child::Features` is a relative path expression, where `Child` refers only to child nodes of the context node. This is the node that is currently being processed. The expression retrieves the <Features> element node children of the context node.
- An absolute path expression starts with one or two slash marks (/ or //), followed by an optional, relative path. For example, the initial slash mark in the expression, `/child::ProductDescription`, indicates that it is an absolute path expression. Because a slash mark at the start of an expression returns the document root node of the context node, the expression returns all the <ProductDescription> element node children of the document root.

  If an absolute path starts with a single slash mark, it may or may not be followed by a relative path. If you specify only a single slash mark, the expression returns the root node of the context node. For an XML data type, this is its document node.

A typical path expression is made up of steps. For example, the absolute path expression, `/child::ProductDescription/child::Summary`, contains two steps separated by a slash mark.

- The first step retrieves the <ProductDescription> element node children of the document root.

- The second step retrieves the <Summary> element node children for each retrieved <ProductDescription> element node, which in turn becomes the context node.

A step in a path expression can be an axis step or a general step.

Axis Step

An axis step in a path expression has the following parts.

axis

Defines the direction of movement. An axis step in a path expression that starts at the context node and navigates to those nodes that are reachable in the direction specified by the axis.

node test

Specifies the node type or node names to be selected.

Zero or more optional predicates

Filters the nodes by selecting some and discarding others.

The following examples use an **axis step** in the path expressions:

- The absolute path expression, `/child::ProductDescription`, contains only one step. It specifies an axis (`child`) and a node test (`ProductDescription`).
- The relative path expression, `child::ProductDescription/child::Features`, contains two steps separated by a slash mark. Both steps specify a child axis. ProductDescription and Features are node tests.
- The relative path expression, `child::root/child::Location[attribute::LocationID=10]`, contains two steps separated by a slash mark. The first step specifies an axis (`child`) and a node test (`root`). The second step specifies all three components of an axis step: an axis (child), a node test (`Location`), and a predicate (`[attribute::LocationID=10]`).

For more information about the components of an axis step, see Specifying Axis in a Path Expression Step, Specifying Node Test in a Path Expression Step, and Specifying a Predicate in a Path Expression Step.

General Step

A general step is just an expression that must evaluate to a sequence of nodes.

The XQuery implementation in SQL Server supports a general step as the first step in a path expression. Following are examples of path expressions that use general steps.

```
(/a, /b)/c
```

```
id(/a/b)
```

For more information about the id function see, id Function (XQuery).

In This Section

Specifying Axis in a Path Expression Step

Describes working with the axis step in a path expression.

Specifying Node Test in a Path Expression Step

Describes working with node tests in a path expression.

Specifying Predicates in a Path Expression Step

Describes working with predicates in a path expression.

Using Abbreviated Syntax in a Path Expression

Describes working with abbreviated syntax in a path expression.

Specifying Axis in a Path Expression Step

An axis step in a path expression includes the following components:

- An axis
- A node test
- Zero or more step qualifiers (optional)

For more information, see FLWOR Statement and Iteration (XQuery).

The XQuery implementation in SQL Server supports the following axis steps,

Axis	Description
child	Returns children of the context node.
descendant	Returns all descendants of the context node.
parent	Returns the parent of the context node.
attribute	Returns attributes of the context node.
self	Returns the context node itself.
descendant-or-self	Returns the context node and all descendants of the context node.

All these axes, except the **parent** axis, are forward axes. The **parent** axis is a reverse axis, because it searches backward in the document hierarchy. For example, the relative path expression `child::ProductDescription/child::Summary` has two steps, and each step specifies a `child` axis. The first step retrieves the <ProductDescription> element children of the context node. For each <ProductDescription> element node, the second step retrieves the <Summary> element node children.

The relative path expression, `child::root/child::Location/attribute::LocationID`, has three steps. The first two steps each specify a `child` axis, and the third step specifies the `attribute` axis. When executed against the manufacturing instructions XML documents in the **Production.ProductModel** table, the expression returns the `LocationID` attribute of the <Location> element node child of the <root> element.

Examples

The query examples in this topic are specified against **xml** type columns in the **AdventureWorks** database.

A. Specifying a child axis

For a specific product model, the following query retrieves the <Features> element node children of the <ProductDescription> element node from the product catalog description stored in the `Production.ProductModel` table.

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
  /child::PD:ProductDescription/child::PD:Features')
FROM Production.ProductModel
WHERE ProductModelID=19
```

Note the following from the previous query:

- The `query()` method of the **xml** data type specifies the path expression.
- Both steps in the path expression specify a `child` axis and the node names, `ProductDescription` and `Features`, as node tests. For information about node tests, see Specifying Node Test in a Path Expression Step.

B. Specifying descendant and descendant-or-self axes

The following example uses descendant and descendant-or-self axes. The query in this example is specified against an **xml** type variable. The XML instance is simplified in order to easily illustrate the difference in the generated results.

```
declare @x xml
set @x='
<a>
 <b>text1
   <c>text2
     <d>text3</d>
   </c>
 </b>
```

```
</a>'
declare @y xml
set @y = @x.query('
  /child::a/child::b
')
select @y
```

In the following result, the expression returns the <b> element node child of the <a> element node:

```
<b>text1
   <c>text2
     <d>text3</d>
   </c>
</b>
```

In this expression, if you specify a descendant axis for the path expression, /child::a/child::b/descendant::*, you are asking for all descendants of the <b> element node.

The asterisk (*) in the node test represents the node name as a node test. Therefore, the primary node type of the descendant axis, the element node, determines the types of nodes returned. That is, the expression returns all the element nodes.. Text nodes are not returned. For more information about the primary node type and its relationship with the node test, see Specifying Node Test in a Path Expression Step topic.

The element nodes <c> and <d> are returned, as shown in the following result:

```
<c>text2
     <d>text3</d>
</c>
<d>text3</d>
```

If you specify a descendant-or-self axis instead of the descendant axis, /child::a/child::b/descendant-or-self::* returns the context node, element <b>, and its descendant.

This is the result:

```
<b>text1
   <c>text2
     <d>text3</d>
   </c>
</b>
```

```
<c>text2
     <d>text3</d>
</c>


<d>text3</d>
```

The following sample query against the **AdventureWorks** database retrieves all the descendant element nodes of the <Features> element child of the <ProductDescription> element:

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
  /child::PD:ProductDescription/child::PD:Features/descendant::*
')
FROM  Production.ProductModel
WHERE ProductModelID=19
```

C. Specifying a parent axis

The following query returns the <Summary> element child of the <ProductDescription> element in the product catalog XML document stored in the Production.ProductModel table.

This example uses the parent axis to return to the parent of the <Feature> element and retrieve the <Summary> element child of the <ProductDescription> element.

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";


/child::PD:ProductDescription/child::PD:Features/parent::PD:ProductDesc
ription/child::PD:Summary
')
FROM   Production.ProductModel
WHERE  ProductModelID=19
```

In this query example, the path expression uses the parent axis. You can rewrite the expression without the parent axis, as shown in the following:

```
  /child::PD:ProductDescription[child::PD:Features]/child::PD:Summary
```

A more useful example of the parent axis is provided in the following example.

Each product model catalog description stored in the **CatalogDescription** column of the **ProductModel** table has a `<ProductDescription>` element that has the `ProductModelID` attribute and `<Features>` child element, as shown in the following fragment:

```
<ProductDescription ProductModelID="..." >
  ...
  <Features>
    <Feature1>...</Feature1>
    <Feature2>...</Feature2>
   ...
</ProductDescription>
```

The query sets an iterator variable, `$f`, in the FLWOR statement to return the element children of the `<Features>` element. For more information, see FLWOR Statement and Iteration in XQuery. For each feature, the `return` clause constructs an XML in the following form:

```
<Feature ProductModelID="...">...</Feature>
<Feature ProductModelID="...">...</Feature>
```

To add the `ProductModelID` for each `<Feature>` element, the `parent` axis is specified:

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
declare namespace
wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain";
  for $f in /child::PD:ProductDescription/child::PD:Features/child::*
  return
   <Feature
     ProductModelID="{
($f/parent::PD:Features/parent::PD:ProductDescription/attribute::Produc
tModelID)[1]}" >
          { $f }
   </Feature>
')
FROM  Production.ProductModel
```

```
WHERE ProductModelID=19
```

This is the partial result:

```
<Feature ProductModelID="19">
  <wm:Warranty
   xmlns:wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelWarrAndMain">
    <wm:WarrantyPeriod>3 years</wm:WarrantyPeriod>
    <wm:Description>parts and labor</wm:Description>
  </wm:Warranty>
</Feature>
<Feature ProductModelID="19">
  <wm:Maintenance
   xmlns:wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelWarrAndMain">
    <wm:NoOfYears>10 years</wm:NoOfYears>
    <wm:Description>maintenance contract available through your dealer
                or any AdventureWorks retail store.</wm:Description>
  </wm:Maintenance>
</Feature>
<Feature ProductModelID="19">
  <p1:wheel
   xmlns:p1="http://www.adventure-works.com/schemas/OtherFeatures">
      High performance wheels.
  </p1:wheel>
</Feature>
```

Note that the predicate `[1]` in the path expression is added to ensure that a singleton value is returned.

Specifying Node Test in a Path Expression Step

An axis step in a path expression includes the following components:

- An axis
- A node test
- Zero or more step qualifiers (optional)

For more information, see XQuery Prolog.

A node test is a condition and is the second component of the axis step in a path expression. All the nodes selected by a step must satisfy this condition. For the path expression, `/child::ProductDescription`, the node test is `ProductDescription`. This step retrieves only those element node children whose name is ProductDescription.

A node test condition can include the following:

- A node name. Only nodes of the principal node kind with the specified name are returned.
- A node type. Only nodes of the specified type are returned.

Note
Node names that are specified in XQuery path expressions are not subject to the same collation-sensitive rules as Transact-SQL queries are and are always case-sensitive.

Node Name as Node Test

When specifying a node name as a node test in a path expression step, you must understand the concept of principal node kind. Every axis, child, parent, or attribute, has a principal node kind. For example:

- An attribute axis can contain only attributes. Therefore, the attribute node is the principal node kind of the attribute axis.
- For other axes, if the nodes selected by the axis can contain element nodes, element is the principal node kind for that axis.

When you specify a node name as a node test, the step returns the following types of nodes:

- Nodes that areof the principal node kind of the axis.
- Nodes that have the same name as specified in the node test.

For example, consider the following path expression:

```
child::ProductDescription
```

This one-step expression specifies a `child` axis and the node name `ProductDescription` as the node test. The expression returns only those nodes that are of the principal node kind of the child axis, element nodes, and which have ProductDescription as their name.

The path expression, `/child::PD:ProductDescription/child::PD:Features/descendant::*`, has three steps. These steps specify child and descendant axes. In each step, the node name is specified as the node test. The wildcard character (`*`) in the third step indicates all nodes of the principle node kind for the descendant axis. The principal node kind of the axis determines the type of nodes selected, and the node name filters that the nodes selected.

As a result, when this expression is executed against product catalog XML documents in the **ProductModel** table, it retrieves all the element node children of the <Features> element node child of the <ProductDescription> element.

The path expression, `/child::PD:ProductDescription/attribute::ProductModelID`, is made up of two steps. Both of these steps specify a node name as the node test. Also, the second step uses the attribute axis. Therefore, each step selects nodes of the principal node kind of its axis that has the name specified as the node test. Thus, the expression returns **ProductModelID** attribute node of the <ProductDescription> element node.

When specifying the names of nodes for node tests, you can also use the wildcard character (*) to specify the local name of a node or for its namespace prefix, as shown in the following example:

```
declare @x xml
set @x = '
<greeting xmlns="ns1">
   <salutation>hello</salutation>
</greeting>
<greeting xmlns="ns2">
   <salutation>welcome</salutation>
</greeting>
<farewell xmlns="ns1" />'
select @x.query('//*:greeting')
select @x.query('declare namespace ns="ns1"; /ns:*')
```

Node Type as Node Test

To query for node types other than element nodes, use a node type test. As shown in the following table, there are four node type tests available.

Node type	Returns	Example
`comment()`	True for a comment node.	`following::comment()` selects all the comment nodes that appear after the context node.
`node()`	True for a node of any kind.	`preceding::node()` selects all the nodes that appear before the context node.
`processing-instruction()`	True for a processing instruction node.	`self::processing instruction()` selects all the processing instruction nodes

Node type	Returns	Example
		within the context node.
`text()`	True for a text node.	`child::text()` selects the text nodes that are children of the context node.

If node type, such as text() or comment() ..., is specified as the node test, the step just returns nodes of the specified kind, regardless of the principal node kind of the axis. For example, the following path expression returns only the comment node children of the context node:

```
child::comment()
```

Similarly, `/child::ProductDescription/child::Features/child::comment()` retrieves comment node children of the <Features> element node child of the <ProductDescription> element node.

Examples

The following examples compare node name and node kind.

A. Results of specifying the node name and the node type as node tests in a path expression

In the following example, a simple XML document is assigned to an **xml** type variable. The document is queried by using different path expressions. The results are then compared.

```
declare @x xml
set @x='
<a>
 <b>text1
   <c>text2
     <d>text3</d>
   </c>
 </b>
</a>'
select @x.query('
/child::a/child::b/descendant::*
')
```

This expression asks for the descendant element nodes of the <b> element node.

The asterisk (*) in the node test indicates a wildcard character for node name. The descendant axis has the element node as its primary node kind. Therefore, the

expression returns all the descendant element nodes of element node <b>. That is, element nodes <c> and <d> are returned, as shown in the following result:

```
<c>text2
     <d>text3</d>
</c>
<d>text3</d>
```

If you specify a descendent-or-self axis instead of specifying a descendant axis, , the context node is returned and also its descendants:

```
/child::a/child::b/descendant-or-self::*
```

This expression returns the element node <b> and its descendant element nodes. In returning the descendant nodes, the primary node kind of the descendant-or-self axis, element node type, determines what kind of nodes are returned.

This is the result:

```
<b>text1
   <c>text2
     <d>text3</d>
   </c>
</b>

<c>text2
     <d>text3</d>
</c>

<d>text3</d>
```

The previous expression used a wildcard character as a node name. Instead, you can use the `node()` function, as show in this expression:

```
/child::a/child::b/descendant::node()
```

Because `node()` is a node type, you will receive all the nodes of the descendant axis. This is the result:

```
text1
<c>text2
     <d>text3</d>
</c>
text2
<d>text3</d>
```

```
text3
```

Again, if you specify descendant-or-self axis and `node()` as the node test, you will receive all the descendants, elements, and text nodes, and also the context node, the `<b>` element.

```
<b>text1
   <c>text2
     <d>text3</d>
   </c>
</b>
text1
<c>text2
     <d>text3</d>
</c>
text2
<d>text3</d>
text3
```

B. Specifying a node name in the node test

The following example specifies a node name as the node test in all the path expressions. As a result, all the expressions return nodes of the principal node kind of the axis that have the node name specified in the node test.

The following query expression returns the `<Warranty>` element from the product catalog XML document stored in the `Production.ProductModel` table:

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
declare namespace
wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain";
 /child::PD:ProductDescription/child::PD:Features/child::wm:Warranty
')
FROM Production.ProductModel
WHERE ProductModelID=19
```

Note the following from the previous query:

- The `namespace` keyword in the XQuery prolog defines a prefix that is used in the query body. For more information, about the XQuery prolog, see XQuery Prolog .

- All three steps in the path expression specify the child axis and a node name as the node test.
- The optional step-qualifier part of the axis step is not specified in any of the steps in the expression.

The query returns the <Warranty> element children of the <Features> element child of the <ProductDescription> element.

This is the result:

```
<wm:Warranty
xmlns:wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
  <wm:WarrantyPeriod>3 years</wm:WarrantyPeriod>
  <wm:Description>parts and labor</wm:Description>
</wm:Warranty>
```

In the following query, the path expression specifies a wildcard character (*) in a node test.

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
declare namespace
wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain";
 /child::PD:ProductDescription/child::PD:Features/child::*
')
FROM Production.ProductModel
WHERE ProductModelID=19
```

The wildcard character is specified for the node name. Thus, the query returns all the element node children of the <Features> element node child of the <ProductDescription> element node.

The following query is similar to the previous query except that together with the wildcard character, a namespace is specified. As a result, all the element node children in that namespace are returned. Note that the <Features> element can contain elements from different namespaces.

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
```

```
declare namespace
wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain";
 /child::PD:ProductDescription/child::PD:Features/child::wm:*
')
FROM Production.ProductModel
WHERE ProductModelID=19
```

You can use the wildcard character as a namespace prefix, as shown in this query:

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
declare namespace
wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain";
 /child::PD:ProductDescription/child::PD:Features/child::*:Maintenance
')
FROM Production.ProductModel
WHERE ProductModelID=19
```

This query returns the <Maintenance> element node children in all the namespaces from the product catalog XML document.

C. Specifying node kind in the node test

The following example specifies the node kind as the node test in all the path expressions. As a result, all the expressions return nodes of the kind specified in the node test.

In the following query, the path expression specifies a node kind in its third step:

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
declare namespace
wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain";
 /child::PD:ProductDescription/child::PD:Features/child::text()
')
FROM Production.ProductModel
```

```
WHERE ProductModelID=19
```

In the next query, the following is specified:

- The path expression has three steps separated by a slash mark (/).
- Each of these steps specifies a child axis.
- The first two steps specify a node name as the node test, and the third step specifies a node kind as the node test.
- The expression returns text node children of the <Features> element child of the <ProductDescription> element node.

Only one text node is returned. This is the result:

```
These are the product highlights.
```

The following query returns the comment node children of the <ProductDescription> element:

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
declare namespace
wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain";
 /child::PD:ProductDescription/child::comment()
')
FROM Production.ProductModel
WHERE ProductModelID=19
```

Note the following from the previous query:

- The second step specifies a node kind as the node test.
- As a result, the expression returns the comment node children of the <ProductDescription> element nodes.

This is the result:

```
<!-- add one or more of these elements... one for each specific product
in this product model -->
<!-- add any tags in <specifications> -->
```

The following query retrieves the top-level, processing-instruction nodes:

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
```

```
declare namespace
wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain";
 /child::processing-instruction()
')
FROM Production.ProductModel
WHERE ProductModelID=19
```

This is the result:

```
<?xml-stylesheet href="ProductDescription.xsl" type="text/xsl"?>
```

You can pass a string literal parameter to the `processing-instruction()` node test. In this case, the query returns the processing instructions whose name attribute value is the string literal specified in the argument.

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
declare namespace
wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain";
 /child::processing-instruction("xml-stylesheet")
')
FROM Production.ProductModel
WHERE ProductModelID=19
```

Implementation Limitations

Following are the specific limitations

- The extended SequenceType node tests are not supported.
- processing-instruction(name) is not supported. Instead, put the name in quotation marks.

Specifying Predicates in a Path Expression Step

As described in the topic, Path Expressions in XQuery, an axis step in a path expression includes the following components:

- An axis.
- A node test. For more information, see exist() Method (xml Data Type).
- Zero or more predicates. This is optional.

The optional predicate is the third part of the axis step in a path expression.

Predicates

A predicate is used to filter a node sequence by applying a specified test. The predicate expression is enclosed in a square bracket and is bound to the last node in a path expression.

For example, assume that an SQL parameter value (x) of the **xml** data type is declared, as shown in the following:

```
declare @x xml
set @x = '
<People>
  <Person>
    <Name>John</Name>
    <Age>24</Age>
  </Person>
  <Person>
    <Name>Goofy</Name>
    <Age>54</Age>
  </Person>
  <Person>
    <Name>Daffy</Name>
    <Age>30</Age>
  </Person>
</People>
'
```

In this case, the following are valid expressions that use a predicate value of [1] at each of three different node levels:

```
select @x.query('/People/Person/Name[1]')
select @x.query('/People/Person[1]/Name')
select @x.query('/People[1]/Person/Name')
```

Note that in each case, the predicate binds to the node in the path expression where it is applied. For example, the first path expression selects the first `<Name>` element within each /People/Person node and, with the provided XML instance, returns the following:

```
<Name>John</Name><Name>Goofy</Name><Name>Daffy</Name>
```

However, the second path expression selects all `<Name>` elements that are under the first /People/Person node. Therefore, it returns the following:

```
<Name>John</Name>
```

Parentheses can also be used to change the order of evaluation of the predicate. For example, in the following expression, a set of parentheses is used to separate the path of (/People/Person/Name) from the predicate [1]:

```
select @x.query('(/People/Person/Name)[1]')
```

In this example, the order in which the predicate is applied changes. This is because the enclosed path is first evaluated (/People/Person/Name) and then the predicate [1] operator is applied to the set that contains all nodes that matched the enclosed path. Without the parentheses, the order of operation would be different in that the [1] is applied as a `child::Name` node test, similar to the first path expression example.

Quantifiers and Predicates

Quantifiers can be used and added more than one time within the braces of the predicate itself. For example, using the previous example, the following is a valid use of more than one quantifier within a complex predicate subexpression.

```
select @x.query('/People/Person[contains(Name[1], "J") and
xs:integer(Age[1]) < 40]/Name/text()')
```

The result of a predicate expression is converted to a Boolean value and is referred to as the predicate truth value. Only the nodes in the sequence for which the predicate truth value is True are returned in the result. All other nodes are discarded.

For example, the following path expression includes a predicate in its second step:

```
/child::root/child::Location[attribute::LocationID=10]
```

The condition specified by this predicate is applied to all the `<Location>` element node children. The result is that only those work center locations whose LocationID attribute value is 10 are returned.

The previous path expression is executed in the following SELECT statement:

```
SELECT Instructions.query('
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
 /child::AWMI:root/child::AWMI:Location[attribute::LocationID=10]
')
FROM Production.ProductModel
WHERE ProductModelID=7
```

Computing Predicate Truth Values

The following rules are applied in order to determine the predicate truth value, according to the XQuery specifications:

1. If the value of the predicate expression is an empty sequence, the predicate truth value is False.

   For example:

```
SELECT Instructions.query('
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
 /child::AWMI:root/child::AWMI:Location[attribute::LotSize]
')
FROM Production.ProductModel
WHERE ProductModelID=7
```

The path expression in this query returns only those <Location> element nodes that have a LotSize attribute specified. If the predicate returns an empty sequence for a specific <Location>, that work center location is not returned in the result.

2. Predicate values can only be xs:integer, xs:Boolean, or node*. For node*, the predicate evaluates to True if there are any nodes, and False for an empty sequence. Any other numeric type, such as double and float type, generates a static typing error. The predicate truth value of an expression is True if and only if the resulting integer is equal to the value of the context position. Also, only integer literal values and the **last()** function reduce the cardinality of the filtered step expression to 1.

   For example, the following query retrieves the third child element node of the <Features> element.

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
declare namespace
wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain";
 /child::PD:ProductDescription/child::PD:Features/child::*[3]
')
FROM Production.ProductModel
WHERE ProductModelID=19
```

   Note the following from the previous query:

   - The third step in the expression specifies a predicate expression whose value is 3. Therefore, the predicate truth value of this expression is True only for the nodes whose context position is 3.
   - The third step also specifies a wildcard character (*) that indicates all the nodes in the node test. However, the predicate filters the nodes and returns only the node in the third position.

- The query returns the third child element node of the `<Features>` element children of the `<ProductDescription>` element children of the document root.

3. If the value of the predicate expression is one simple type value of type Boolean, the predicate truth value is equal to the value of the predicate expression.

   For example, the following query is specified against an **xml** type variable that holds an XML instance, the customer survey XML instance. The query retrieves those customers who have children. In this query, that would be <HasChildren>1</HasChildren>.

```
declare @x xml
set @x='
<Survey>
  <Customer CustomerID="1" >
  <Age>27</Age>
  <Income>20000</Income>
  <HasChildren>1</HasChildren>
  </Customer>
  <Customer CustomerID="2" >
  <Age>27</Age>
  <Income>20000</Income>
  <HasChildren>0</HasChildren>
  </Customer>
</Survey>
'
declare @y xml
set @y = @x.query('
  for $c in /child::Survey/child::Customer[(
child::HasChildren[1] cast as xs:boolean ? )]
  return
      <CustomerWithChildren>
          { $c/attribute::CustomerID }
      </CustomerWithChildren>
')
select @y
```

   Note the following from the previous query:

- The expression in the **for** loop has two steps, and the second step specifies a predicate. The value of this predicate is a Boolean type value. If this value is True, the truth value of the predicate is also True.
- The query returns the <Customer> element children, whose predicate value is True, of the <Survey> element children of the document root. This is the result:

  ```
  <CustomerWithChildren CustomerID="1"/>
  ```

4. If the value of the predicate expression is a sequence that contains at least one node, the predicate truth value is True.

For example, the following query retrieves ProductModelID for product models whose XML catalog description includes at least one feature, a child element of the <Features> element, from the namespace associated with the **wm** prefix.

```
SELECT ProductModelID
FROM   Production.ProductModel
WHERE CatalogDescription.exist('
             declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
             declare namespace
wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain";
             /child::PD:ProductDescription/child::PD:Features[wm:*]
             ') = 1
```

Note the following from the previous query:

- The WHERE clause specifies the exist() method (XML data type).
- The path expression inside the **exist()** method specifies a predicate in the second step. If the predicate expression returns a sequence of at least one feature, the truth value of this predicate expression is True. In this case, because the **exist()** method returns a True, the ProductModelID is returned.

Static Typing and Predicate Filters

The predicates may also affect the statically inferred type of an expression. Integer literal values and the **last()** function reduce the cardinality of the filtered step expression to at most one.

Using Abbreviated Syntax in a Path Expression

All the examples in Understanding the Path Expressions in XQuery use unabbreviated syntax for path expressions. The unabbreviated syntax for an axis step in a path expression includes the axis name and node test, separated by double colon, and followed by zero or more step qualifiers.

For example:

```
child::ProductDescription[attribute::ProductModelID=19]
```

XQuery supports the following abbreviations for use in path expressions:

- The **child** axis is the default axis. Therefore, the **child::** axis can be omitted from a step in an expression. For example, `/child::ProductDescription/child::Summary` can be written as `/ProductDescription/Summary`.
- An **attribute** axis can be abbreviated as @. For example, `/child::ProductDescription[attribute::ProductModelID=10]` can be written as `/ProudctDescription[@ProductModelID=10]`.
- A **/descendant-or-self::node()/** can be abbreviated as //. For example, `/descendant-or-self::node()/child::act:telephoneNumber` can be written as `//act:telephoneNumber`.

  The previous query retrieves all telephone numbers stored in the AdditionalContactInfo column in the Contact table. The schema for AdditionalContactInfo is defined in a way that a <telephoneNumber> element can appear anywhere in the document. Therefore, to retrieve all the telephone numbers, you must search every node in the document. The search starts at the root of the document and continues through all the descendant nodes.

  The following query retrieves all the telephone numbers for a specific customer contact:

```
SELECT AdditionalContactInfo.query('
            declare namespace
act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes";
            declare namespace crm="http://schemas.adventure-
works.com/Contact/Record";
            declare namespace
ci="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactInfo";
            /descendant-or-
self::node()/child::act:telephoneNumber
            ') as result
 FROM Person.Contact
 WHERE ContactID=1
```

  If you replace the path expression with the abbreviated syntax, `//act:telephoneNumber`, you receive the same results.

- The **self::node()** in a step can be abbreviated to a single dot (.). However, the dot is not equivalent or interchangeable with the **self::node()**.

  For example, in the following query, the use of a dot represents a value and not a node:

  ```
  ("abc", "cde")[. > "b"]
  ```
- The **parent::node()** in a step can be abbreviated to a double dot (..).

Sequence Expressions

SQL Server supports the XQuery operators that are used to construct, filter, and combine a sequence of items. An item can be an atomic value or a node.

Constructing Sequences

You can use the comma operator to construct a sequence that concatenates items into a single sequence.

A sequence can contain duplicate values. Nested sequences, a sequence within a sequence, are collapsed. For example, the sequence (1, 2, (3, 4, (5))) becomes (1, 2, 3, 4, 5). These are examples of constructing sequences.

Example A

The following query returns a sequence of five atomic values:

```
declare @x xml
set @x=''
select @x.query('(1,2,3,4,5)')
go
-- result 1 2 3 4 5
```

The following query returns a sequence of two nodes:

```
-- sequence of 2 nodes
declare @x xml
set @x=''
select @x.query('(<a/>, <b/>)')
go
-- result
<a />
<b />
```

The following query returns an error, because you are constructing a sequence of atomic values and nodes. This is a heterogeneous sequence and is not supported.

```
declare @x xml
```

```
set @x=''
select @x.query('(1, 2, <a/>, <b/>)')
go
```

Example B

The following query constructs a sequence of atomic values by combining four sequences of different length into a single sequence.

```
declare @x xml
set @x=''
select @x.query('(1,2),10,(),(4, 5, 6)')
go
-- result = 1 2 10 4 5 6
```

You can sort the sequence by using FLOWR and ORDER BY:

```
declare @x xml
set @x=''
select @x.query('for $i in ((1,2),10,(),(4, 5, 6))
                 order by $i
                 return $i')
go
```

You can count the items in the sequence by using the **fn:count()** function.

```
declare @x xml
set @x=''
select @x.query('count( (1,2,3,(),4) )')
go
-- result = 4
```

Example C

The following query is specified against the AdditionalContactInfo column of the **xml** type in the Contact table. This column stores additional contact information, such as one or more additional telephone numbers, pager numbers, and addresses. The <telephoneNumber>, <pager>, and other nodes can appear anywhere in the document. The query constructs a sequence that contains all the <telephoneNumber> children of the context node, followed by the <pager> children. Note the use of the comma sequence operator in the return expression, `($a//act:telephoneNumber, $a//act:pager)`.

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes' AS act,
 'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactInfo' AS aci)

SELECT AdditionalContactInfo.query('
   for $a in /aci:AdditionalContactInfo
   return ($a//act:telephoneNumber, $a//act:pager)
') As Result
FROM Person.Contact
WHERE ContactID=3
```

This is the result:

```
<act:telephoneNumber
xmlns:act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes">
  <act:number>333-333-3333</act:number>
</act:telephoneNumber>
<act:telephoneNumber
xmlns:act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes">
  <act:number>333-333-3334</act:number>
</act:telephoneNumber>
<act:pager
xmlns:act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes">
  <act:number>999-555-1244</act:number>
  <act:SpecialInstructions>
Page only in case of emergencies.
</act:SpecialInstructions>
</act:pager>
```

Filtering Sequences

You can filter the sequence returned by an expression by adding a predicate to the expression. For more information, see XQuery Expressions. For example, the following query returns a sequence of three `<a>` element nodes:

```
declare @x xml
set @x = '<root>
<a attrA="1">111</a>
<a></a>
<a></a>
</root>'
SELECT @x.query('/root/a')
```

This is the result:

```
<a attrA="1">111</a>
<a />
<a />
```

To retrieve only <a> elements that have the attribute attrA, you can specify a filter in the predicate. The resulting sequence will have only one <a> element.

```
declare @x xml
set @x = '<root>
<a attrA="1">111</a>
<a></a>
<a></a>
</root>'
SELECT @x.query('/root/a[@attrA]')
```

This is the result:

```
<a attrA="1">111</a>
```

For more information about how to specify predicates in a path expression, see Specifying Predicates in a Path Expression Step.

The following example builds a sequence expression of subtrees and then applies a filter to the sequence.

```
declare @x xml
set @x = '
<a>
  <c>C under a</c>
</a>
<b>
   <c>C under b</c>
</b>
```

```
<c>top level c</c>
<d></d>
'
```

The expression in `(/a, /b)` constructs a sequence with subtrees `/a` and `/b` and from the resulting sequence the expression filters element `<c>`.

```
SELECT @x.query('
  (/a, /b)/c
')
```

This is the result:

```
<c>C under a</c>
<c>C under b</c>
```

The following example applies a predicate filter. The expression finds elements `<a>` and `<b>` that contain element `<c>`.

```
declare @x xml
set @x = '
<a>
  <c>C under a</c>
</a>
<b>
   <c>C under b</c>
</b>

<c>top level c</c>
<d></d>
'
SELECT @x.query('
  (/a, /b)[c]
')
```

This is the result:

```
<a>
  <c>C under a</c>
</a>
<b>
  <c>C under b</c>
```

```
</b>
```

Implementation Limitations

These are the limitations:

- XQuery range expression is not supported.
- Sequences must be homogeneous. Specifically, all items in a sequence must be either nodes or atomic values. This is statically checked.
- Combining node sequences by using the union, intersect, or except operator is not supported.

See Also

XQuery Expressions

Arithmetic Expressions

All arithmetic operators are supported, except for **idiv**. The following examples illustrate the basic use of arithmetic operators:

```
DECLARE @x xml
SET @x=''
SELECT @x.query('2 div 2')
SELECT @x.query('2 * 2')
```

Because **idiv** is not supported, a solution is to use the **xs:integer()** constructor:

```
DECLARE @x xml
SET @x=''
-- Following will not work
-- SELECT @x.query('2 idiv 2')
-- Workaround
SELECT @x.query('xs:integer(2 div 3)')
```

The resulting type from an arithmetic operator is based on the types of the input values. If the operands are different types, either one or both when required will be cast to a common primitive base type according to the type hierarchy. For information about type hierarchy, see Type Casting Rules in XQuery.

Numeric type promotion occurs if the two operations are different numeric base types. For example, adding an **xs:decimal** to an **xs:double** would first change the decimal value to a double . Next, addition would be performed that would result in a double value.

Untyped atomic values are cast to the other operand's numeric base type, or to **xs:double** if the other operand is also untyped.

Implementation Limitations

These are the limitations:

- Arguments for the arithmetic operators must be of numeric type or **untypedAtomic**.
- Operations on **xs:integer** values result in a value of type **xs:decimal** instead of **xs:integer**.

Comparison Expressions

XQuery provides the following types of comparison operators:

- General comparison operators
- Value comparison operators
- Node comparison operators
- Node order comparison operators

General Comparison Operators

General comparison operators can be used to compare atomic values, sequences, or any combination of the two.

The general operators are defined in the following table.

Operator	Description
=	Equal
!=	Not equal
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

When you are comparing two sequences by using general comparison operators and a value exists in the second sequence that compares True to a value in the first sequence, the overall result is True. Otherwise, it is False. For example, (1, 2, 3) = (3, 4) is True, because the value 3 appears in both sequences.

```
declare @x xml
set @x=''
select @x.query('(1,2,3) = (3,4)')
```

The comparison expects that the values are of comparable types. Specifically, they are statically checked. For numeric comparisons, numeric type promotion can occur. For

example, if a decimal value of 10 is compared to a double value 1e1, the decimal value is changed to double. Note that this can create inexact results, because double comparisons cannot be exact.

If one of the values is untyped, it is cast to the other value's type. In the following example, value 7 is treated as an integer. Before being compared, the untyped value of /a[1] is converted to an integer. The integer comparison returns True.

```
declare @x xml
set @x='<a>6</a>'
select @x.query('/a[1] < 7')
```

Conversely, if the untyped value is compared to a string or another untyped value, it will be cast to xs:string. In the following query, string 6 is compared to string "17". The following query returns False, because of the string comparison.

```
declare @x xml
set @x='<a>6</a>'
select @x.query('/a[1] < "17"')
```

The following query returns small-size pictures of a product model from the product catalog provided in the AdventureWorks sample database. The query compares a sequence of atomic values returned by `PD:ProductDescription/PD:Picture/PD:Size` with a singleton sequence, "small". If the comparison is True, it returns the <Picture> element.

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS PD)
SELECT CatalogDescription.query('
    for $P in /PD:ProductDescription/PD:Picture[PD:Size = "small"]
    return $P') as Result
FROM   Production.ProductModel
WHERE  ProductModelID=19
```

The following query compares a sequence of telephone numbers in <number> elements to the string literal "112-111-1111". The query compares the sequence of telephone number elements in the AdditionalContactInfo column to determine if a specific telephone number for a specific customer exists in the document.

```
WITH XMLNAMESPACES (
  'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes' AS act,
  'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactInfo' AS aci)
```

```
SELECT AdditionalContactInfo.value('
   /aci:AdditionalContactInfo//act:telephoneNumber/act:number = "112-
111-1111"', 'nvarchar(10)') as Result
FROM Person.Contact
WHERE ContactID=1
```

The query returns True. This indicates that the number exists in the document. The following query is a slightly modified version of the previous query. In this query, the telephone number values retrieved from the document are compared to a sequence of two telephone number values. If the comparison is True, the <number> element is returned.

```
WITH XMLNAMESPACES (
  'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes' AS act,
  'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactInfo' AS aci)


SELECT AdditionalContactInfo.query('
  if (/aci:AdditionalContactInfo//act:telephoneNumber/act:number =
("222-222-2222","112-111-1111"))
  then
     /aci:AdditionalContactInfo//act:telephoneNumber/act:number
  else
    ()') as Result
FROM Person.Contact
WHERE ContactID=1
```

This is the result:

```
<act:number
  xmlns:act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes">
    111-111-1111
</act:number>
<act:number
```

```
  xmlns:act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes">
    112-111-1111
</act:number>
```

Value Comparison Operators

Value comparison operators are used to compare atomic values. Note that you can use general comparison operators instead of value comparison operators in your queries.

The value comparison operators are defined in the following table.

Operator	Description
eq	Equal
ne	Not equal
lt	Less than
gt	Greater than
le	Less than or equal to
ge	Greater than or equal to

If the two values compare the same according to the chosen operator, the expression will return True. Otherwise, it will return False. If either value is an empty sequence, the result of the expression is False.

These operators work on singleton atomic values only. That is, you cannot specify a sequence as one of the operands.

For example, the following query retrieves <Picture> elements for a product model where the picture size is "small:

```
SELECT CatalogDescription.query('
              declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
              for $P in /PD:ProductDescription/PD:Picture[PD:Size eq
"small"]
              return
                    $P
             ') as Result
FROM Production.ProductModel
```

```
WHERE ProductModelID=19
```

Note the following from the previous query:

- `declare namespace` defines the namespace prefix that is subsequently used in the query.
- The <Size> element value is compared with the specified atomic value, "small".
- Note that because the value operators work only on atomic values, the **data()** function is implicitly used to retrieve the node value. That is, `data($P/PD:Size) eq "small"` produces the same result.

This is the result:

```
<PD:Picture
  xmlns:PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription">
  <PD:Angle>front</PD:Angle>
  <PD:Size>small</PD:Size>
  <PD:ProductPhotoID>31</PD:ProductPhotoID>
</PD:Picture>
```

Note that the type promotion rules for value comparisons are the same as for general comparisons. Also, SQL Server uses the same casting rules for untyped values during value comparisons as it uses during general comparisons. In contrast, the rules in the XQuery specification always cast the untyped value to xs:string during value comparisons.

Node Comparison Operator

The node comparison operator, **is**, applies only to node types. The result it returns indicates whether two nodes passed in as operands represent the same node in the source document. This operator returns True if the two operands are the same node. Otherwise, it returns False.

The following query checks whether the work center location 10 is the first in the manufacturing process of a specific product model.

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions' AS AWMI)

SELECT ProductModelID, Instructions.query('
    if (  (//AWMI:root/AWMI:Location[@LocationID=10])[1]
          is
          (//AWMI:root/AWMI:Location[1])[1] )
    then
```

```
          <Result>equal</Result>
    else
          <Result>Not-equal</Result>
         ') as Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

This is the result:

```
ProductModelID       Result
-------------- --------------------------
7              <Result>equal</Result>
```

Node Order Comparison Operators

Node order comparison operators compare pairs of nodes, based on their positions in a document.

These are the comparisons that are made, based on document order:

- << : Does **operand 1** precede **operand 2** in the document order.
- >> : Does **operand 1** follow **operand 2** in the document order.

The following query returns True if the product catalog description has the <Warranty> element appearing before the <Maintenance> element in the document order for a particular product.

```
WITH XMLNAMESPACES (
  'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS PD,
  'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain' AS WM)

SELECT CatalogDescription.value('
     (/PD:ProductDescription/PD:Features/WM:Warranty)[1] <<
          (/PD:ProductDescription/PD:Features/WM:Maintenance)[1]',
'nvarchar(10)') as Result
FROM  Production.ProductModel
where ProductModelID=19
```

Note the following from the previous query:

- The **value()** method of the **xml** data type is used in the query.
- The Boolean result of the query is converted to **nvarchar(10)** and returned.
- The query returns True.

See Also

XQuery Expressions

XQuery Expressions

Logical Expressions

XQuery supports the logical **and** and **or** operators.

```
expression1 and expression2
```

```
expression1 or expression2
```

The test expressions, `expression1`, `expression2`, in SQL Server can result in an empty sequence, a sequence of one or more nodes, or a single Boolean value. Based on the result, their effective Boolean value is determined in the following manner:

- If the test expression results in an empty sequence, the result of the expression is False.
- If the test expression results in a single Boolean value, this value is the result of the expression.
- If the test expression results in a sequence of one or more nodes, the result of the expression is True.
- Otherwise, a static error is raised.

The logical **and** and **or** operator is then applied to the resulting Boolean values of the expressions with the standard logical semantics.

The following query retrieves from the product catalog the front-angle small pictures, the `<Picture>` element, for a specific product model. Note that for each product description document, the catalog can store one or more product pictures with different attributes, such as size and angle.

```
SELECT CatalogDescription.query('
     declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     for $F in /PD:ProductDescription/PD:Picture[PD:Size="small"
                                                 and PD:Angle="front"]
     return
         $F
    ') as Result
FROM  Production.ProductModel
where ProductModelID=19
```

This is the result:

```
<PD:Picture
  xmlns:PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelDescription">
  <PD:Angle>front</PD:Angle>
  <PD:Size>small</PD:Size>
  <PD:ProductPhotoID>31</PD:ProductPhotoID>
</PD:Picture>
```

See Also

XQuery Expressions

XML Construction

In XQuery, you can use the **direct** and **computed** constructors to construct XML structures within a query.

Note
The ability to compute node names when you use computed constructors is not supported before SQL Server 2005. Therefore, there is no difference between the **direct** and **computed** constructors.

InSQL Server 2000, only the construction mode, strip, and the XMLSpace policy, or boundary white space policy, strip are supported.

Using Direct Constructors

When you use direct constructors, you specify XML-like syntax when you construct the XML. The following examples illustrate XML construction by the direct constructors.

Constructing Elements

In using XML notations, you can construct elements. The following example uses the direct element constructor expression and creates a <ProductModel> element. The constructed element has three child elements

- A text node.
- Two element nodes, <Summary> and <Features>.
  - The <Summary> element has one text node child whose value is "Some description".
  - The <Features> element has three element node children, <Color>, <Weight>, and <Warranty>. Each of these nodes has one text node child and have the values Red, 25, 2 years parts and labor, respectively.

```
declare @x xml
```

```
set @x=''
select @x.query('<ProductModel ProductModelID="111">
This is product model catalog description.
<Summary>Some description</Summary>
<Features>
  <Color>Red</Color>
  <Weight>25</Weight>
  <Warranty>2 years parts and labor</Warranty>
</Features></ProductModel>')
```

This is the resulting XML:

```
<ProductModel ProductModelID="111">
  This is product model catalog description.
  <Summary>Some description</Summary>
  <Features>
    <Color>Red</Color>
    <Weight>25</Weight>
    <Warranty>2 years parts and labor</Warranty>
  </Features>
</ProductModel>
```

Although constructing elements from constant expressions, as shown in this example, is useful, the true power of this XQuery language feature is the ability to construct XML that dynamically extracts data from a database. You can use curly braces to specify query expressions. In the resulting XML, the expression is replaced by its value. For example, the following query constructs a `<NewRoot>` element with one child element (`<e>`). The value of element `<e>` is computed by specifying a path expression inside curly braces ("{ … }").

```
DECLARE @x xml
SET @x='<root>5</root>'
SELECT @x.query('<NewRoot><e> { /root } </e></NewRoot>')
```

The braces act as context-switching tokens and switch the query from XML construction to query evaluation. In this case, the XQuery path expression inside the braces, `/root`, is evaluated and the results are substituted for it.

This is the result:

```
<NewRoot>
```

```
  <e>
    <root>5</root>
  </e>
</NewRoot>
```

The following query is similar to the previous one. However, the expression in the curly braces specifies the **data()** function to retrieve the atomic value of the `<root>` element and assigns it to the constructed element, `<e>`.

```
DECLARE @x xml
SET @x='<root>5</root>'
DECLARE @y xml
SET @y = (SELECT @x.query('
                            <NewRoot>
                              <e> { data(/root) } </e>
                            </NewRoot>' ))
SELECT @y
```

This is the result:

```
<NewRoot>
  <e>5</e>
</NewRoot>
```

If you want to use the curly braces as part of your text instead of context-switching tokens, you can escape them as "}}" or "{{", as shown in this example:

```
DECLARE @x xml
SET @x='<root>5</root>'
DECLARE @y xml
SET @y = (SELECT @x.query('
<NewRoot> Hello, I can use {{ and  }} as part of my text</NewRoot>'))
SELECT @y
```

This is the result:

```
<NewRoot> Hello, I can use { and  } as part of my text</NewRoot>
```

The following query is another example of constructing elements by using the direct element constructor. Also, the value of the `<FirstLocation>` element is obtained by executing the expression in the curly braces. The query expression returns the manufacturing steps at the first work center location from the Instructions column of the Production.ProductModel table.

```
SELECT Instructions.query('
```

```
    declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
        <FirstLocation>
           { /AWMI:root/AWMI:Location[1]/AWMI:step }
        </FirstLocation>
') as Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

This is the result:

```
<FirstLocation>
  <AWMI:step
xmlns:AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions">
      Insert <AWMI:material>aluminum sheet MS-2341</AWMI:material> into
the <AWMI:tool>T-85A framing tool</AWMI:tool>.
  </AWMI:step>
  <AWMI:step
xmlns:AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions">
      Attach <AWMI:tool>Trim Jig TJ-26</AWMI:tool> to the upper and
lower right corners of the aluminum sheet.
  </AWMI:step>
   ...
</FirstLocation>
```

Element Content in XML Construction

The following example illustrates the behavior of the expressions in constructing element content by using the direct element constructor. In the following example, the direct element constructor specifies one expression. For this expression, one text node is created in the resulting XML.

```
declare @x xml
set @x='
<root>
  <step>This is step 1</step>
  <step>This is step 2</step>
```

```
  <step>This is step 3</step>
</root>'
select @x.query('
<result>
 { for $i in /root[1]/step
    return string($i)
 }
</result>')
```

The atomic value sequence resulting from the expression evaluation is added to the text node with a space added between the adjacent atomic values, as shown in the result. The constructed element has one child. This is a text node that contains the value shown in the result.

```
<result>This is step 1 This is step 2 This is step 3</result>
```

Instead of one expression, if you specify three separate expressions generating three text nodes, the adjacent text nodes are merged into a single text node, by concatenation, in the resulting XML.

```
declare @x xml
set @x='
<root>
  <step>This is step 1</step>
  <step>This is step 2</step>
  <step>This is step 3</step>
</root>'
select @x.query('
<result>
 { string(/root[1]/step[1]) }
 { string(/root[1]/step[2]) }
 { string(/root[1]/step[3]) }
</result>')
```

The constructed element node has one child. This is a text node that contains the value shown in the result.

```
<result>This is step 1This is step 2This is step 3</result>
```

Constructing Attributes

When you are constructing elements by using the direct element constructor, you can also specify attributes of the element by using XML-like syntax, as shown in this example:

```
declare @x xml
set @x=''
select @x.query('<ProductModel ProductModelID="111">
This is product model catalog description.
<Summary>Some description</Summary>
</ProductModel>')
```

This is the resulting XML:

```
<ProductModel ProductModelID="111">
  This is product model catalog description.
  <Summary>Some description</Summary>
</ProductModel>
```

The constructed element `<ProductModel>` has a ProductModelID attribute and these child nodes:

- A text node, `This is product model catalog description.`
- An element node, `<Summary>`. This node has one text node child, `Some description`.

When you are constructing an attribute, you can specify its value with an expression in curly braces. In this case, the result of the expression is returned as the attribute value.

In the following example, the **data()** function is not strictly required. Because you are assigning the expression value to an attribute, **data()** is implicitly applied to retrieve the typed value of the specified expression.

```
DECLARE @x xml
SET @x='<root>5</root>'
DECLARE @y xml
SET @y = (SELECT @x.query('<NewRoot attr="{ data(/root) }"
></NewRoot>'))
SELECT @y
```

This is the result:

```
<NewRoot attr="5" />
```

Following is another example in which expressions are specified for LocationID and SetupHrs attribute construction. These expressions are evaluated against the XML in the Instruction column. The typed valued of the expression is assigned to the attributes.

```
SELECT Instructions.query('
```

```
    declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
        <FirstLocation
         LocationID="{ (/AWMI:root/AWMI:Location[1]/@LocationID)[1] }"
         SetupHrs = "{ (/AWMI:root/AWMI:Location[1]/@SetupHours)[1] }"
>
           { /AWMI:root/AWMI:Location[1]/AWMI:step }
        </FirstLocation>
') as Result
FROM  Production.ProductModel
where ProductModelID=7
```

This is the partial result:

```
<FirstLocation LocationID="10" SetupHours="0.5" >
  <AWMI:step …
  </AWMI:step>
  ...
</FirstLocation>
```

Implementation Limitations

These are the limitations:

- Multiple or mixed (string and XQuery expression) attribute expressions are not supported. For example, as shown in the following query, you construct XML where `Item` is a constant and the value `5` is obtained by evaluating a query expression:

  ```
  <a attr="Item 5" />
  ```

  The following query returns an error, because you are mixing constant string with an expression ({/x}) and this is not supported:

  ```
  DECLARE @x xml
  SET @x ='<x>5</x>'
  SELECT @x.query( '<a attr="Item {/x}"/>' )
  ```

  In this case, you have the following options:

  - Form the attribute value by the concatenation of two atomic values. These atomic values are serialized into the attribute value with a space between the atomic values:

    ```
    SELECT @x.query( '<a attr="{''Item'', data(/x)}"/>' )
    ```

    This is the result:

```
<a attr="Item 5" />
```

- Use the concat function to concatenate the two string arguments into the resulting attribute value:

  ```
  SELECT @x.query( '<a attr="{concat(''Item'', /x[1])}"/>' )
  ```

  In this case, there is no space added between the two string values. If you want a space between the two values, you must explicitly provide it.

  This is the result:

  ```
  <a attr="Item5" />
  ```

- Multiple expressions as an attribute value are not supported. For example, the following query returns an error:

  ```
  DECLARE @x xml
  SET @x ='<x>5</x>'
  SELECT @x.query( '<a attr="{/x}{/x}"/>' )
  ```

- Heterogeneous sequences are not supported. Any attempt to assign a heterogeneous sequence as an attribute value will return an error, as shown in the following example. In this example, a heterogeneous sequence, a string "Item" and an element <x>, is specified as the attribute value:

  ```
  DECLARE @x xml
  SET @x ='<x>5</x>'
  select @x.query( '<a attr="{''Item'', /x }" />')
  ```

  If you apply the **data()** function, the query works because it retrieves the atomic value of the expression, /x, which is concatenated with the string. Following is a sequence of atomic values:

  ```
  SELECT @x.query( '<a attr="{''Item'', data(/x)}"/>' )
  ```

  This is the result:

  ```
  <a attr="Item 5" />
  ```

- Attribute node order is enforced during serialization rather than during static type checking. For example, the following query fails because it attempts to add an attribute after a non-attribute node.

  ```
  select convert(xml, '').query('
  element x { attribute att { "pass" }, element y { "Element text"
  }, attribute att2 { "fail" } }
  ')
  go
  ```

  The above query returns the following error:

```
XML well-formedness check: Attribute cannot appear outside of
element declaration. Rewrite your XQuery so it returns well-
formed XML.
```

Adding Namespaces

When constructing XML by using the direct constructors, the constructed element and attribute names can be qualified by using a namespace prefix. You can bind the prefix to the namespace in the following ways:

- By using a namespace declaration attribute.
- By using the WITH XMLNAMESPACES clause.
- In the XQuery prolog.

Using a Namespace Declaration Attribute to Add Namespaces

The following example uses a namespace declaration attribute in the construction of element <a> to declare a default namespace. The construction of the child element <b> undoes the declaration of the default namespace declared in the parent element.

```
declare @x xml
set @x ='<x>5</x>'
select @x.query( '
  <a xmlns="a">
    <b xmlns=""/>
  </a>' )
```

This is the result:

```
<a xmlns="a">
  <b xmlns="" />
</a>
```

You can assign a prefix to the namespace. The prefix is specified in the construction of element <a>.

```
declare @x xml
set @x ='<x>5</x>'
select @x.query( '
  <x:a xmlns:x="a">
    <b/>
  </x:a>' )
```

This is the result:

```
<x:a xmlns:x="a">
  <b />
```

```
</x:a>
```

Beginning with SQL Server 2005, you can un-declare a default namespace in the XML construction, but you cannot un-declare a namespace prefix. The following query returns an error, because you cannot un-declare a prefix as specified in the construction of element <b>.

```
declare @x xml
set @x ='<x>5</x>'
select @x.query( '
  <x:a xmlns:x="a">
    <b xmlns:x=""/>
  </x:a>' )
```

The newly constructed namespace is available to use inside the query. For example, the following query declares a namespace in constructing the element, <FirstLocation>, and specifies the prefix in the expressions for the LocationID and SetupHrs attribute values.

```
SELECT Instructions.query('
        <FirstLocation
xmlns:AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions"
         LocationID="{ (/AWMI:root/AWMI:Location[1]/@LocationID)[1] }"
         SetupHrs = "{ (/AWMI:root/AWMI:Location[1]/@SetupHours)[1] }"
>
           { /AWMI:root/AWMI:Location[1]/AWMI:step }
        </FirstLocation>
') as Result
FROM  Production.ProductModel
where ProductModelID=7
```

Note that creating a new namespace prefix in this way will override any pre-existing namespace declaration for this prefix. For example, the namespace declaration, AWMI="http://someURI", in the query prolog is overridden by the namespace declaration in the <FirstLocation> element.

```
SELECT Instructions.query('
declare namespace AWMI="http://someURI";
        <FirstLocation
xmlns:AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions"
```

```
          LocationID="{ (/AWMI:root/AWMI:Location[1]/@LocationID)[1] }"
          SetupHrs = "{ (/AWMI:root/AWMI:Location[1]/@SetupHours)[1] }"
>
            { /AWMI:root/AWMI:Location[1]/AWMI:step }
         </FirstLocation>
') as Result
FROM  Production.ProductModel
where ProductModelID=7
```

Using a Prolog to Add Namespaces

This example illustrates how namespaces can be added to the constructed XML. A default namespace is declared in the query prolog.

```
declare @x xml
set @x ='<x>5</x>'
select @x.query( '
            declare default element namespace "a";
             <a><b xmlns=""/></a>' )
```

Note that in the construction of element <b>, the namespace declaration attribute is specified with an empty string as its value. This un-declares the default namespace that is declared in the parent.

```
This is the result:
<a xmlns="a">
  <b xmlns="" />
</a>
```

XML Construction and White Space Handling

The element content in XML construction can include white-space characters. These characters are handled in the following ways:

- The white-space characters in namespace URIs are treated as the XSD type **anyURI**. Specifically, this is how they are handled:
  - Any white-space characters at the start and end are trimmed.
  - Internal white-space character values are collapsed into a single space
- The linefeed characters inside the attribute content are replaced by spaces. All other white-space characters remain as they are.
- The white space inside elements is preserved.

The following example illustrates white-space handling in XML construction:

```
-- line feed is repaced by space.
```

```
declare @x xml
set @x=''
select @x.query('

declare namespace myNS="   http://
 abc/
xyz

";
<test attr="    my
test   attr
value    " >

<a>

This     is  a

test

</a>
</test>
') as XML_Result
```

This is the result:

```
-- result
<test attr="<test attr="    my test   attr  value    "><a>

This     is  a

test

</a></test>
```

```
"><a>

This     is  a

test

</a></test>
```

Other Direct XML Constructors

The constructors for processing instructions and XML comments use the same syntax as that of the corresponding XML construct. Computed constructors for text nodes are also supported, but are primarily used in XML DML to construct text nodes.

Note For an example of using an explicit text node constructor, see the specific example in insert (XML DML).

In the following query, the constructed XML includes an element, two attributes, a comment, and a processing instruction. Note that a comma is used before the <FirstLocation>, because a sequence is being constructed.

```
SELECT Instructions.query('
  declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
   <?myProcessingInstr abc="value" ?>,
   <FirstLocation
        WorkCtrID = "{ (/AWMI:root/AWMI:Location[1]/@LocationID)[1] }"
        SetupHrs = "{ (/AWMI:root/AWMI:Location[1]/@SetupHours)[1] }" >
       <!-- some comment -->
       <?myPI some processing instructions ?>
       { (/AWMI:root/AWMI:Location[1]/AWMI:step) }
    </FirstLocation>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

This is the partial result:

```
<?myProcessingInstr abc="value" ?>
<FirstLocation WorkCtrID="10" SetupHrs="0.5">
```

```
  <!-- some comment -->
  <?myPI some processing instructions ?>
  <AWMI:step
xmlns:AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions">I
  nsert <AWMI:material>aluminum sheet MS-2341</AWMI:material> into the
<AWMI:tool>T-85A framing tool</AWMI:tool>.
  </AWMI:step>
    ...
/FirstLocation>
```

Using Computed Constructors

. In this case, you specify the keywords that identify the type of node you want to construct. Only the following keywords are supported:

- element
- attribute
- text

For element and attribute nodes, these keywords are followed by node name and also by the expression, enclosed in braces, that generates the content for that node. In the following example, you are constructing this XML:

```
<root>
  <ProductModel PID="5">Some text <summary>Some
Summary</summary></ProductModel>
</root>
```

This is the query that uses computed constructors do generate the XML:

```
declare @x xml
set @x=''
select @x.query('element root
               {
                  element ProductModel
     {
attribute PID { 5 },
text{"Some text "},
    element summary { "Some Summary" }
 }
```

```
               } ')
```

The expression that generates the node content can specify a query expression.

```
declare @x xml
set @x='<a attr="5"><b>some summary</b></a>'
select @x.query('element root
               {
                  element ProductModel
     {
attribute PID { /a/@attr },
text{"Some text "},
    element summary { /a/b }
 }
               } ')
```

Note that the computed element and attribute constructors, as defined in the XQuery specification, allow you to compute the node names. When you are using direct constructors in SQL Server, the node names, such as element and attribute, must be specified as constant literals. Therefore, there is no difference in the direct constructors and computed constructors for elements and attributes.

In the following example, the content for the constructed nodes is obtained from the XML manufacturing instructions stored in the Instructions column of the **xml** data type in the ProductModel table.

```
SELECT Instructions.query('
  declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
   element FirstLocation
     {
        attribute LocationID {
(/AWMI:root/AWMI:Location[1]/@LocationID)[1] },
        element   AllTheSteps { /AWMI:root/AWMI:Location[1]/AWMI:step }
     }
') as Result
FROM  Production.ProductModel
where ProductModelID=7
```

This is the partial result:

```
<FirstLocation LocationID="10">
  <AllTheSteps>
    <AWMI:step> ... </AWMI:step>
    <AWMI:step> ... </AWMI:step>
    ...
  </AllTheSteps>
</FirstLocation>
```

Additional Implementation Limitations

Computed attribute constructors cannot be used to declare a new namespace. Also, the following computed constructors are not supported in SQL Server:

- Computed document node constructors
- Computed processing instruction constructors
- Computed comment constructors

See Also

XQuery Expressions

FLWOR Statement and Iteration

XQuery defines the FLWOR iteration syntax. FLWOR is the acronym for `for`, `let`, `where`, `order by`, and `return`.

A FLWOR statement is made up of the following parts:

- One or more FOR clauses that bind one or more iterator variables to input sequences.

  Input sequences can be other XQuery expressions such as XPath expressions. They are either sequences of nodes or sequences of atomic values. Atomic value sequences can be constructed using literals or constructor functions. Constructed XML nodes are not allowed as input sequences in SQL Server.
- An optional `let` clause. This clause assigns a value to the given variable for a specific iteration. The assigned expression can be an XQuery expression such as an XPath expression, and can return either a sequence of nodes or a sequence of atomic values. Atomic value sequences can be constructed by using literals or constructor functions. Constructed XML nodes are not allowed as input sequences in SQL Server.
- An iterator variable. This variable can have an optional type assertion by using the `as` keyword.
- An optional `where` clause. This clause applies a filter predicate on the iteration.
- An optional `order by` clause.

- A `return` expression. The expression in the `return` clause constructs the result of the FLWOR statement.

For example, the following query iterates over the <Step> elements at the first manufacturing location and returns the string value of the <Step> nodes:

```
declare @x xml
set @x='<ManuInstructions ProductModelID="1"
ProductModelName="SomeBike" >
<Location LocationID="L1" >
  <Step>Manu step 1 at Loc 1</Step>
  <Step>Manu step 2 at Loc 1</Step>
  <Step>Manu step 3 at Loc 1</Step>
</Location>
<Location LocationID="L2" >
  <Step>Manu step 1 at Loc 2</Step>
  <Step>Manu step 2 at Loc 2</Step>
  <Step>Manu step 3 at Loc 2</Step>
</Location>
</ManuInstructions>'
SELECT @x.query('
   for $step in /ManuInstructions/Location[1]/Step
   return string($step)
')
```

This is the result:

```
Manu step 1 at Loc 1 Manu step 2 at Loc 1 Manu step 3 at Loc 1
```

The following query is similar to the previous one, except that it is specified against the Instructions column, a typed xml column, of the ProductModel table. The query iterates over all the manufacturing steps, <step> elements, at the first work center location for a specific product.

```
SELECT Instructions.query('
   declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
for $Step in //AWMI:root/AWMI:Location[1]/AWMI:step
      return
           string($Step)
```

```
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

Note the following from the previous query:

- The `$Step` is the iterator variable.
- The path expression, `//AWMI:root/AWMI:Location[1]/AWMI:step`, generates the input sequence. This sequence is the sequence of the `<step>` element node children of the first `<Location>` element node.
- The optional predicate clause, `where`, is not used.
- The `return` expression returns a string value from the `<step>` element.

The string function (XQuery) is used to retrieve the string value of the `<step>` node.

This is the partial result:

```
Insert aluminum sheet MS-2341 into the T-85A framing tool.
Attach Trim Jig TJ-26 to the upper and lower right corners of
the aluminum sheet. ....
```

These are examples of additional input sequences that are allowed:

```
declare @x xml
set @x=''
SELECT @x.query('
for $a in (1, 2, 3)
  return $a')
-- result = 1 2 3

declare @x xml
set @x=''
SELECT @x.query('
for $a in
   for $b in (1, 2, 3)
      return $b
return $a')
-- result = 1 2 3

declare @x xml
set @x='<ROOT><a>111</a></ROOT>'
```

```
SELECT @x.query('
  for $a in (xs:string( "test"), xs:double( "12" ), data(/ROOT/a ))
  return $a')
-- result test 12 111
```

In SQL Server, heterogeneous sequences are not allowed. Specifically, sequences that contain a mixture of atomic values and nodes are not allowed.

Iteration is frequently used together with the XML Construction syntax in transforming XML formats, as shown in the next query.

In the AdventureWorks sample database, the manufacturing instructions stored in the **Instructions** column of the **Production.ProductModel** table have the following form:

```
<Location LocationID="10" LaborHours="1.2"
            SetupHours=".2" MachineHours=".1">
  <step>describes 1st manu step</step>
   <step>describes 2nd manu step</step>
   ...
</Location>
...
```

The following query constructs new XML that has the `<Location>` elements with the work center location attributes returned as child elements:

```
<Location>
   <LocationID>10</LocationID>
   <LaborHours>1.2</LaborHours>
   <SetupHours>.2</SteupHours>
   <MachineHours>.1</MachineHours>
</Location>
...
```

This is the query:

```
SELECT Instructions.query('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
        for $WC in /AWMI:root/AWMI:Location
        return
          <Location>
            <LocationID> { data($WC/@LocationID) } </LocationID>
```

```
            <LaborHours>   { data($WC/@LaborHours) }   </LaborHours>
            <SetupHours>   { data($WC/@SetupHours) }   </SetupHours>
            <MachineHours> { data($WC/@MachineHours) } </MachineHours>
          </Location>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

Note the following from the previous query:

- The FLWOR statement retrieves a sequence of `<Location>` elements for a specific product.
- The data function (XQuery) is used to extract the value of each attribute so they will be added to the resulting XML as text nodes instead of as attributes.
- The expression in the RETURN clause constructs the XML that you want.

This is a partial result:

```
<Location>
  <LocationID>10</LocationID>
  <LaborHours>2.5</LaborHours>
  <SetupHours>0.5</SetupHours>
  <MachineHours>3</MachineHours>
</Location>
<Location>
   ...
<Location>
...
```

Using the let Clause

You can use the `let` clause to name repeating expressions that you can refer to by referring to the variable. The expression assigned to a `let` variable is inserted into the query every time the variable is referenced in the query. This means that the statement is executed as many times as the expression gets referenced.

In the database, the manufacturing instructions contain information about the tools required and the location where the tools are used. The following query uses the `let` clause to list the tools required to build a production model, as well as the locations where each tool is needed.

```
SELECT Instructions.query('
```

```
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
        for $T in //AWMI:tool
            let $L := //AWMI:Location[.//AWMI:tool[.=data($T)]]
        return
          <tool desc="{data($T)}" Locations="{data($L/@LocationID)}"/>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

Using the where Clause

You can use the `where` clause to filter results of an interation. This is illustrated in this next example.

In the manufacturing of a bicycle, the manufacturing process goes through a series of work center locations. Each work center location defines a sequence of manufacturing steps. The following query retrieves only those work center locations that manufacture a bicycle model and have less than three manufacturing steps. That is, they have less than three `<step>` elements.

```
SELECT Instructions.query('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
for $WC in /AWMI:root/AWMI:Location
      where count($WC/AWMI:step) < 3
      return
          <Location >
           { $WC/@LocationID }
          </Location>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

Note the following in the previous query:

- The `where` keyword uses the **count()** function to count the number of `<step>` child elements in each work center location.
- The `return` expression constructs the XML that you want from the results of the iteration.

This is the result:

```
<Location LocationID="30"/>
```

The result of the expression in the `where` clause is converted to a Boolean value by using the following rules, in the order specified. These are the same as the rules for predicates in path expressions, except that integers are not allowed:

1. If the `where` expression returns an empty sequence, its effective Boolean value is False.
2. If the `where` expression returns one simple Boolean type value, that value is the effective Boolean value.
3. If the `where` expression returns a sequence that contains at least one node, the effective Boolean value is True.
4. Otherwise, it raises a static error.

Multiple Variable Binding in FLWOR

You can have a single FLWOR expression that binds multiple variables to input sequences. In the following example, the query is specified against an untyped xml variable. The FLOWR expression returns the first `<Step>` element child in each `<Location>` element.

```
declare @x xml
set @x='<ManuInstructions ProductModelID="1"
ProductModelName="SomeBike" >
<Location LocationID="L1" >
  <Step>Manu step 1 at Loc 1</Step>
  <Step>Manu step 2 at Loc 1</Step>
  <Step>Manu step 3 at Loc 1</Step>
</Location>
<Location LocationID="L2" >
  <Step>Manu step 1 at Loc 2</Step>
  <Step>Manu step 2 at Loc 2</Step>
  <Step>Manu step 3 at Loc 2</Step>
</Location>
</ManuInstructions>'
SELECT @x.query('
   for $Loc in /ManuInstructions/Location,
       $FirstStep in $Loc/Step[1]
   return
```

```
       string($FirstStep)
')
```

Note the following from the previous query:

- The `for` expression defines `$Loc` and `$FirstStep` variables.
- The `two` expressions, `/ManuInstructions/Location` and `$FirstStep in $Loc/Step[1]`, are correlated in that the values of `$FirstStep` depend on the values of `$Loc`.
- The expression associated with `$Loc` generates a sequence of `<Location>` elements. For each `<Location>` element, `$FirstStep` generates a sequence of one `<Step>` element, a singleton.
- `$Loc` is specified in the expression associated with the `$FirstStep` variable.

This is the result:

```
Manu step 1 at Loc 1
Manu step 1 at Loc 2
```

The following query is similar, except that it is specified against the Instructions column, typed **xml** column, of the **ProductModel** table. XML Construction (XQuery) is used to generate the XML that you want.

```
SELECT Instructions.query('
     declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
for $WC in /root/Location,
            $S  in $WC/step
      return
          <Step LocationID= "{$WC/@LocationID }" >
            { $S/node() }
          </Step>
') as Result
FROM  Production.ProductModel
WHERE ProductModelID=7
```

Note the following in the previous query:

- The `for` clause defines two variables, `$WC` and `$S`. The expression associated with `$WC` generates a sequence of work center locations in the manufacturing of a bicycle product model. The path expression assigned to the `$S` variable generates a sequence of steps for each work center location sequence in the `$WC`.

- The return statement constructs XML that has a `<Step>` element that contains the manufacturing step and the **LocationID** as its attribute.
- The **declare default element namespace** is used in the XQuery prolog so that all the namespace declarations in the resulting XML appear at the top-level element. This makes the result more readable. For more information about default namespaces, see Atomization (XQuery).

This is the partial result:

```
<Step xmlns=
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions"
  LocationID="10">
     Insert <material>aluminum sheet MS-2341</material> into the
<tool>T-
     85A framing tool</tool>.
</Step>
...
<Step xmlns=
      "http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions"
    LocationID="20">
        Assemble all frame components following blueprint
        <blueprint>1299</blueprint>.
</Step>
...
```

Using the order by Clause

Sorting in XQuery is performed by using the `order by` clause in the FLWOR expression. The sorting expressions passed to the `order by` clause must return values whose types are valid for the **gt** operator. Each sorting expression must result in a singleton a sequence with one item. By default, sorting is performed in ascending order. You can optionally specify ascending or descending order for each sorting expression.

Note

Sorting comparisons on string values performed by the XQuery implementation in SQL Server are always performed by using the binary Unicode codepoint collation.

The following query retrieves all the telephone numbers for a specific customer from the AdditionalContactInfo column. The results are sorted by telephone number.

```
USE AdventureWorks2012;
GO
SELECT AdditionalContactInfo.query('
   declare namespace
act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes";
   declare namespace
aci="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactInfo";
   for $a in /aci:AdditionalContactInfo//act:telephoneNumber
   order by $a/act:number[1] descending
   return $a
') As Result
FROM Person.Person
WHERE BusinessEntityID=291;
```

Note that the Atomization (XQuery) process retrieves the atomic value of the `<number>` elements before passing it to `order by`. You can write the expression by using the **data()** function, but that is not required.

```
order by data($a/act:number[1]) descending
```

This is the result:

```
<act:telephoneNumber
xmlns:act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes">
  <act:number>333-333-3334</act:number>
</act:telephoneNumber>
<act:telephoneNumber
xmlns:act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes">
  <act:number>333-333-3333</act:number>
</act:telephoneNumber>
```

Instead of declaring the namespaces in the query prolog, you can declare them by using WITH XMLNAMESPACES.

```
WITH XMLNAMESPACES (
   'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes' AS act,
```

```
   'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactInfo'  AS aci)

SELECT AdditionalContactInfo.query('
   for $a in /aci:AdditionalContactInfo//act:telephoneNumber
   order by $a/act:number[1] descending
   return $a
') As Result
FROM Person.Person
WHERE BusinessEntityID=291;
```

You can also sort by attribute value. For example, the following query retrieves the newly created <Location> elements that have the LocationID and LaborHours attributes sorted by the LaborHours attribute in descending order. As a result, the work center locations that have the maximum labor hours are returned first.

```
SELECT Instructions.query('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
for $WC in /AWMI:root/AWMI:Location
order by $WC/@LaborHours descending
        return
          <Location>
             { $WC/@LocationID }
             { $WC/@LaborHours }
          </Location>
') as Result
FROM Production.ProductModel
WHERE ProductModelID=7;
```

This is the result:

```
<Location LocationID="60" LaborHours="4"/>
<Location LocationID="50" LaborHours="3"/>
<Location LocationID="10" LaborHours="2.5"/>
<Location LocationID="20" LaborHours="1.75"/>
<Location LocationID="30" LaborHours="1"/>
<Location LocationID="45" LaborHours=".5"/>
```

In the following query, the results are sorted by element name. The query retrieves the specifications of a specific product from the product catalog. The specifications are the children of the <Specifications> element.

```
SELECT CatalogDescription.query('
     declare namespace
 pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
      for $a in /pd:ProductDescription/pd:Specifications/*
     order by local-name($a)
      return $a
    ') as Result
FROM Production.ProductModel
where ProductModelID=19;
```

Note the following from the previous query:

- The `/p1:ProductDescription/p1:Specifications/*` expression returns element children of <Specifications>.
- The `order by (local-name($a))` expression sorts the sequence by the local part of the element name.

This is the result:

```
<Color>Available in most colors</Color>
<Material>Almuminum Alloy</Material>
<ProductLine>Mountain bike</ProductLine>
<RiderExperience>Advanced to Professional riders</RiderExperience>
<Style>Unisex</Style>
```

Nodes in which the ordering expression returns empty are sorted to the start of the sequence, as shown in the following example:

```
declare @x xml
set @x='<root>
  <Person Name="A" />
  <Person />
  <Person Name="B" />
</root>
'
select @x.query('
  for $person in //Person
```

```
  order by $person/@Name
  return   $person
')
```

This is the result:

```
<Person />
<Person Name="A" />
<Person Name="B" />
```

You can specify multiple sorting criteria, as shown in the following example. The query in this example sorts `<Employee>` elements first by Title and then by Administrator attribute values.

```
declare @x xml
set @x='<root>
  <Employee ID="10" Title="Teacher"        Gender="M" />
  <Employee ID="15" Title="Teacher"  Gender="F" />
  <Employee ID="5" Title="Teacher"         Gender="M" />
  <Employee ID="11" Title="Teacher"        Gender="F" />
  <Employee ID="8" Title="Administrator"   Gender="M" />
  <Employee ID="4" Title="Administrator"   Gender="F" />
  <Employee ID="3" Title="Teacher"         Gender="F" />
  <Employee ID="125" Title="Administrator" Gender="F" /></root>'
SELECT @x.query('for $e in /root/Employee
order by $e/@Title ascending, $e/@Gender descending

  return
     $e
')
```

This is the result:

```
<Employee ID="8" Title="Administrator" Gender="M" />
<Employee ID="4" Title="Administrator" Gender="F" />
<Employee ID="125" Title="Administrator" Gender="F" />
<Employee ID="10" Title="Teacher" Gender="M" />
<Employee ID="5" Title="Teacher" Gender="M" />
<Employee ID="11" Title="Teacher" Gender="F" />
<Employee ID="15" Title="Teacher" Gender="F" />
```

```
<Employee ID="3" Title="Teacher" Gender="F" />
```

Implementation Limitations

These are the limitations:

- The sorting expressions must be homogeneously typed. This is statically checked.
- Sorting empty sequences cannot be controlled.
- The empty least, empty greatest, and collation keywords on `order by` are not supported

See Also

XQuery Expressions

Ordered and Unordered Expressions

By default, the ordering mode for all operations in SQL Server is **ordered**. Therefore, the node sequences returned by the path expressions and the FLWOR expressions, without the **order by** clause, are in document order.

The additional **ordered** and **unordered** syntax described in the XQuery specification is not supported.

See Also

Path Expressions (XQuery)

FLWOR Statement and Iteration in XQuery

Path Expressions in XQuery

Conditional Expressions

XQuery supports the following conditional **if-then-else** statement:

```
if (<expression1>)
then
  <expression2>
else
  <expression3>
```

Depending on the effective Boolean value of `expression1`, either `expression2` or `expression3` is evaluated. For example:

- If the test expression, `expression1`, results in an empty sequence, the result is False.
- If the test expression, `expression1`, results in a simple Boolean value, this value is the result of the expression.

- If the test expression, `expression1`, results in a sequence of one or more nodes, the result of the expression is True.
- Otherwise, a static error is raised.

Also note the following:

- The test expression must be enclosed between parentheses.
- The **else** expression is required. If you do not need it, you can return " ( ) ", as illustrated in the examples in this topic.

For example, the following query is specified against the **xml** type variable. The **if** condition tests the value of the SQL variable (@v) inside the XQuery expression by using the sql:variable() function extension function. If the variable value is "FirstName", it returns the `<FirstName>` element. Otherwise, it returns the `<LastName>` element.

```
declare @x xml
declare @v varchar(20)
set @v='FirstName'
set @x='
<ROOT rootID="2">
  <FirstName>fname</FirstName>
  <LastName>lname</LastName>
</ROOT>'
SELECT @x.query('
if ( sql:variable("@v")="FirstName" ) then
  /ROOT/FirstName
 else
   /ROOT/LastName
')
```

This is the result:

```
<FirstName>fname</FirstName>
```

The following query retrieves the first two feature descriptions from the product catalog description of a specific product model. If there are more features in the document, it adds a `<there-is-more>` element with empty content.

```
SELECT CatalogDescription.query('
     declare namespace
p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     <Product>
          { /p1:ProductDescription/@ProductModelID }
```

```
          { /p1:ProductDescription/@ProductModelName }
          {
            for $f in
/p1:ProductDescription/p1:Features/*[position()<=2]
            return
            $f
          }
          {
            if (count(/p1:ProductDescription/p1:Features/*) > 2)
            then <there-is-more/>
            else ()
          }
     </Product>
') as x
FROM Production.ProductModel
WHERE ProductModelID = 19
```

In the previous query, the condition in the **if** expression checks whether there are more than two child elements in `<Features>`. If yes, it returns the `<there-is-more/>` element in the result.

This is the result:

```
<Product ProductModelID="19" ProductModelName="Mountain 100">
  <p1:Warranty
xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
    <p1:WarrantyPeriod>3 years</p1:WarrantyPeriod>
    <p1:Description>parts and labor</p1:Description>
  </p1:Warranty>
  <p2:Maintenance
xmlns:p2="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
    <p2:NoOfYears>10 years</p2:NoOfYears>
    <p2:Description>maintenance contract available through your dealer
or any AdventureWorks retail store.</p2:Description>
  </p2:Maintenance>
  <there-is-more />
```

```
</Product>
```

In the following query, a <Location> element with a LocationID attribute is returned if the work center location does not specify the setup hours.

```
SELECT Instructions.query('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
        for $WC in //AWMI:root/AWMI:Location
        return
        if ( $WC[not(@SetupHours)] )
        then
          <WorkCenterLocation>
             { $WC/@LocationID }
          </WorkCenterLocation>
         else
           ()
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

This is the result:

```
<WorkCenterLocation LocationID="30" />
<WorkCenterLocation LocationID="45" />
<WorkCenterLocation LocationID="60" />
```

This query can be written without the **if** clause, as shown in the following example:

```
SELECT Instructions.query('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
for $WC in //AWMI:root/AWMI:Location[not(@SetupHours)]
        return
          <Location>
             { $WC/@LocationID }
          </Location>
') as Result
```

```
FROM Production.ProductModel
where ProductModelID=7
```

See Also

XQuery Expressions

Quantified Expressions

Existential and universal quantifiers specify different semantics for Boolean operators that are applied to two sequences. This is shown in the following table.

Existential quantifier

Given two sequences, if any item in the first sequence has a match in the second sequence, based on the comparison operator that is used, the returned value is True.

Universal quantifier

Given two sequences, if every item in the first sequence has a match in the second sequence, the returned value is True.

XQuery supports quantified expressions in the following form:

```
( some | every ) <variable> in <Expression> (,…) satisfies <Expression>
```

You can use these expressions in a query to explicitly apply either existential or universal quantification to an expression over one or several sequences. In SQL Server, the expression in the `satisfies` clause has to result in one of the following: a node sequence, an empty sequence, or a Boolean value. The effective Boolean value of the result of that expression will be used in the quantification. The existential quantification that uses **some** will return True if at least one of the values bound by the quantifier has a True result in the satisfy expression. The universal quantification that uses **every** must have True for all values bound by the quantifier.

For example, the following query checks every <Location> element to see whether it has a LocationID attribute.

```
SELECT Instructions.query('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
        if (every $WC in //AWMI:root/AWMI:Location
            satisfies $WC/@LocationID)
        then
             <Result>All work centers have workcenterLocation
ID</Result>
         else
```

```
            <Result>Not all work centers have workcenterLocation
ID</Result>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

Because LocationID is a required attribute of the <Location> element, you receive the expected result:

```
<Result>All work centers have Location ID</Result>
```

Instead of using the query() method, you can use the value() method to return the result to the relational world, as shown in the following query. The query returns True if all work center locations have LocationID attributes. Otherwise, the query returns False.

```
SELECT Instructions.value('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
        every $WC in  //AWMI:root/AWMI:Location
            satisfies $WC/@LocationID',
  'nvarchar(10)') as Result
FROM Production.ProductModel
where ProductModelID=7
```

The following query checks to see if one of the product pictures is small. In the product catalog XML, various angles are stored for each product picture of a different size. You might want to ensure that each product catalog XML includes at least one small-sized picture. The following query accomplishes this:

```
SELECT ProductModelID, CatalogDescription.value('
     declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     some $F in /PD:ProductDescription/PD:Picture
        satisfies $F/PD:Size="small"', 'nvarchar(20)') as
SmallPicturesStored
FROM Production.ProductModel
WHERE ProductModelID = 19
```

This is a partial result:

```
ProductModelID SmallPicturesStored
-------------- --------------------
```

```
19              true
```

Implementation Limitations

These are the limitations:

- Type assertion is not supported as part of binding the variable in the quantified expressions.

See Also

XQuery Expressions

SequenceType Expressions

In XQuery, a value is always a sequence. The type of the value is referred to as a sequence type. The sequence type can be used in an **instance of** XQuery expression. The SequenceType syntax described in the XQuery specification is used when you need to refer to a type in an XQuery expression.

The atomic type name can also be used in the **cast as** XQuery expression. In SQL Server, the **instance of** and **cast as** XQuery expressions on SequenceTypes are partially supported.

instance of Operator

The **instance of** operator can be used to determine the dynamic, or run-time, type of the value of the specified expression. For example:

```
Expression instance of SequenceType[Occurrence indicator]
```

Note that the `instance of` operator, the `Occurrence indicator`, specifies the cardinality, number of items in the resulting sequence. If this is not specified, the cardinality is assumed to be 1. In SQL Server, only the question mark (**?**) occurrence indicator is supported. The **?** occurrence indicator indicates that `Expression` can return zero or one item. If the **?** occurrence indicator is specified, `instance of` returns True when the `Expression` type matches the specified `SequenceType`, regardless of whether `Expression` returns a singleton or an empty sequence.

If the **?** occurrence indicator is not specified, `sequence of` returns True only when the `Expression` type matches the `Type` specified and `Expression` returns a singleton.

Note The plus symbol (**+**) and the asterisk (*****) occurrence indicators are not supported in SQL Server.

The following examples illustrate the use of the **instance of** XQuery operator.

Example A

The following example creates an **xml** type variable and specifies a query against it. The query expression specifies an `instance of` operator to determine whether the dynamic type of the value returned by the first operand matches the type specified in the second operand.

The following query returns True, because the 125 value is an instance of the specified type, **xs:integer**:

```
declare @x xml
set @x=''
select @x.query('125 instance of xs:integer')
go
```

The following query returns True, because the value returned by the expression, /a[1], in the first operand is an element:

```
declare @x xml
set @x='<a>1</a>'
select @x.query('/a[1] instance of element()')
go
```

Similarly, `instance of` returns True in the following query, because the value type of the expression in the first expression is an attribute:

```
declare @x xml
set @x='<a attr1="x">1</a>'
select @x.query('/a[1]/@attr1 instance of attribute()')
go
```

In the following example, the expression, `data(/a[1]`, returns an atomic value that is typed as xdt:untypedAtomic. Therefore, the `instance of` returns True.

```
declare @x xml
set @x='<a>1</a>'
select @x.query('data(/a[1]) instance of xdt:untypedAtomic')
go
```

In the following query, the expression, `data(/a[1]/@attrA`, returns an untyped atomic value. Therefore, the `instance of` returns True.

```
declare @x xml
set @x='<a attrA="X">1</a>'
select @x.query('data(/a[1]/@attrA) instance of xdt:untypedAtomic')
go
```

Example B

In this example, you are querying a typed XML column in the AdventureWorks sample database. The XML schema collection associated with the column that is being queried provides the typing information.

In the expression, **data()** returns the typed value of the ProductModelID attribute whose type is xs:string according to the schema associated with the column. Therefore, the `instance of` returns True.

```
SELECT CatalogDescription.query('
   declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
   data(/PD:ProductDescription[1]/@ProductModelID) instance of
xs:string
') as Result
FROM Production.ProductModel
WHERE ProductModelID = 19
```

For more information, see Type System (XQuery).

The following queries use the Boolean `instance of` expression to determine whether the LocationID attribute is of xs:integer type:

```
SELECT Instructions.query('
   declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
   /AWMI:root[1]/AWMI:Location[1]/@LocationID instance of
attribute(LocationID,xs:integer)
') as Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

The following query is specified against the CatalogDescription typed XML column. The XML schema collection associated with this column provides typing information.

The query uses the `element(ElementName, ElementType?)` test in the `instance of` expression to verify that the `/PD:ProductDescription[1]` returns an element node of a specific name and type.

```
SELECT CatalogDescription.query('
     declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     /PD:ProductDescription[1] instance of
element(PD:ProductDescription, PD:ProductDescription?)
    ') as Result
```

```
FROM  Production.ProductModel
where ProductModelID=19
```

The query returns True.

Example C

When using union types, the `instance of` expression in SQL Server has a limitation: Specifically, when the type of an element or attribute is a union type, `instance of` might not determine the exact type. Consequently, a query will return False, unless the atomic types used in the SequenceType is the highest parent of the actual type of the expression in the simpleType hierarchy. That is, the atomic types specified in the SequenceType must be a direct child of anySimpleType. For information about the type hierarchy, see Type Casting Rules in XQuery.

The next query example performs the following:

- Create an XML schema collection with a union type, such as an integer or string type, defined in it.
- Declare a typed **xml** variable by using the XML schema collection.
- Assign a sample XML instance to the variable.
- Query the variable to illustrate the `instance of` behavior when dealing with a union type.

This is the query:

```
CREATE XML SCHEMA COLLECTION MyTestSchema AS '
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://ns" xmlns:ns="http://ns">
<simpleType name="MyUnionType">
<union memberTypes="integer string"/>
</simpleType>
<element name="TestElement" type="ns:MyUnionType"/>
</schema>'
Go
```

The following query returns False, because the SequenceType specified in the `instance of` expression is not the highest parent of the actual type of the specified expression. That is, the value of the `<TestElement>` is an integer type. The highest parent is xs:decimal. However, it is not specified as the second operand to the `instance of` operator.

```
SET QUOTED_IDENTIFIER ON
DECLARE @var XML(MyTestSchema)
```

```
SET @var = '<TestElement xmlns="http://ns">123</TestElement>'

SELECT @var.query('declare namespace ns="http://ns"
   data(/ns:TestElement[1]) instance of xs:integer')
go
```

Because the highest parent of xs:integer is xs:decimal, the query will return True if you modify the query and specify xs:decimal as the SequenceType in the query.

```
SET QUOTED_IDENTIFIER ON
DECLARE @var XML(MyTestSchema)
SET @var = '<TestElement xmlns="http://ns">123</TestElement>'
SELECT @var.query('declare namespace ns="http://ns"
   data(/ns:TestElement[1]) instance of xs:decimal')
go
```

Example D

In this example, you first create an XML schema collection and use it to type an **xml** variable. The typed **xml** variable is then queried to illustrate the `instance of` functionality.

The following XML schema collection defines a simple type, myType, and an element, `<root>`, of type myType:

```
drop xml schema collection SC
go
CREATE XML SCHEMA COLLECTION SC AS '
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="myNS"
xmlns:ns="myNS"
xmlns:s="http://schemas.microsoft.com/sqlserver/2004/sqltypes">
      <import
namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes"/>
      <simpleType name="myType">
           <restriction base="s:varchar">
                 <maxLength value="20"/>
            </restriction>
      </simpleType>
      <element name="root" type="ns:myType"/>
</schema>'
Go
```

Now create a typed **xml** variable and query it:

```
DECLARE @var XML(SC)
SET @var = '<root xmlns="myNS">My data</root>'
SELECT @var.query('declare namespace sqltypes =
"http://schemas.microsoft.com/sqlserver/2004/sqltypes";
declare namespace ns="myNS";
   data(/ns:root[1]) instance of ns:myType')
go
```

Because myType type derives by restriction from a varchar type that is defined in the sqltypes schema, `instance of` will also return True.

```
DECLARE @var XML(SC)
SET @var = '<root xmlns="myNS">My data</root>'
SELECT @var.query('declare namespace sqltypes =
"http://schemas.microsoft.com/sqlserver/2004/sqltypes";
declare namespace ns="myNS";
data(/ns:root[1]) instance of sqltypes:varchar?')
go
```

Example E

In the following example, the expression retrieves one of the values of the IDREFS attribute and uses `instance of` to determine whether the value is of IDREF type. The example performs the following:

- Creates an XML schema collection in which the `<Customer>` element has an **OrderList** IDREFS type attribute, and the `<Order>` element has an **OrderID** ID type attribute.
- Creates a typed **xml** variable and assigns a sample XML instance to it.
- Specifies a query against the variable. The query expression retrieves the first order ID value from the OrderList IDRERS type attribute of the first `<Customer>`. The value retrieved is IDREF type. Therefore, `instance of` returns True.

```
create xml schema collection SC as
'<schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:Customers="Customers" targetNamespace="Customers">
            <element name="Customers" type="Customers:CustomersType"/>
            <complexType name="CustomersType">
                        <sequence>
                            <element name="Customer"
type="Customers:CustomerType" minOccurs="0" maxOccurs="unbounded" />
```

```
                        </sequence>
            </complexType>
             <complexType name="OrderType">
                <sequence minOccurs="0" maxOccurs="unbounded">
                            <choice>
                                <element name="OrderValue"
type="integer" minOccurs="0" maxOccurs="unbounded"/>
                            </choice>
                </sequence>
                <attribute name="OrderID" type="ID" />
            </complexType>

            <complexType name="CustomerType">
                <sequence minOccurs="0" maxOccurs="unbounded">
                            <choice>
                                <element name="spouse" type="string"
minOccurs="0" maxOccurs="unbounded"/>
                                <element name="Order"
type="Customers:OrderType" minOccurs="0" maxOccurs="unbounded"/>
                            </choice>
                </sequence>
                <attribute name="CustomerID" type="string" />
                <attribute name="OrderList" type="IDREFS" />
            </complexType>
 </schema>'
go
declare @x xml(SC)
set @x='<CustOrders:Customers xmlns:CustOrders="Customers">
                <Customer CustomerID="C1" OrderList="OrderA OrderB"  >
                              <spouse>Jenny</spouse>
                                <Order
OrderID="OrderA"><OrderValue>11</OrderValue></Order>
                                <Order
OrderID="OrderB"><OrderValue>22</OrderValue></Order>
```

```
                </Customer>
                <Customer CustomerID="C2" OrderList="OrderC OrderD" >
                                <spouse>John</spouse>
                                <Order
OrderID="OrderC"><OrderValue>33</OrderValue></Order>
                                <Order
OrderID="OrderD"><OrderValue>44</OrderValue></Order>

                        </Customer>
                <Customer CustomerID="C3"  OrderList="OrderE OrderF" >
                                <spouse>Jane</spouse>
                                <Order
OrderID="OrderE"><OrderValue>55</OrderValue></Order>
                                <Order
OrderID="OrderF"><OrderValue>55</OrderValue></Order>
                </Customer>
                <Customer CustomerID="C4"  OrderList="OrderG"  >
                                <spouse>Tim</spouse>
                                <Order
OrderID="OrderG"><OrderValue>66</OrderValue></Order>
                        </Customer>
                <Customer CustomerID="C5"  >
                </Customer>
                <Customer CustomerID="C6" >
                </Customer>
                <Customer CustomerID="C7"  >
                </Customer>
</CustOrders:Customers>'

select @x.query(' declare namespace CustOrders="Customers";
 data(CustOrders:Customers/Customer[1]/@OrderList)[1] instance of
xs:IDREF ? ') as XML_result
```

Implementation Limitations

These are the limitations:

- The **schema-element()** and **schema-attribute()** sequence types are not supported for comparison to the `instance of` operator.
- Full sequences, for example, `(1,2) instance of xs:integer*`, are not supported.
- When you are using a form of the **element()** sequence type that specifies a type name, such as `element(ElementName, TypeName)`, the type must be qualified with a question mark (?). For example, `element(Title, xs:string?)` indicates that the element might be null. SQL Server does not support run-time detection of the **xsi:nil** property by using `instance of`.
- If the value in `Expression` comes from an element or attribute typed as a union, SQL Server can only identify the primitive, not derived, type from which the value's type was derived. For example, if `<e1>` is defined to have a static type of (xs:integer | xs:string), the following will return False.

  ```
  data(<e1>123</e1>) instance of xs:integer
  ```

  However, `data(<e1>123</e1>) instance of xs:decimal` will return True.
- For the **processing-instruction()** and **document-node()** sequence types, only forms without arguments are allowed. For example, `processing-instruction()` is allowed, but `processing-instruction('abc')` is not allowed.

cast as Operator

The **cast as** expression can be used to convert a value to a specific data type. For example:

```
Expression cast as  AtomicType?
```

In SQL Server, the question mark (?) is required after the `AtomicType`. For example, as shown in the following query, `"2" cast as xs:integer?` converts the string value to an integer:

```
declare @x xml
set @x=''
select @x.query('"2" cast as xs:integer?')
```

In the following query, **data()** returns the typed value of the ProductModelID attribute, a string type. The `cast as` operator converts the value to xs:integer.

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS PD)
SELECT CatalogDescription.query('
   data(/PD:ProductDescription[1]/@ProductModelID) cast as xs:integer?
') as Result
FROM Production.ProductModel
```

```
WHERE ProductModelID = 19
```

The explicit use of **data()** is not required in this query. The `cast as` expression performs implicit atomization on the input expression.

Constructor Functions

You can use the atomic type constructor functions. For example, instead of using the `cast as` operator, `"2" cast as xs:integer?`, you can use the **xs:integer()** constructor function, as in the following example:

```
declare @x xml

set @x=''

select @x.query('xs:integer("2")')
```

The following example returns an xs:date value equal to 2000-01-01Z.

```
declare @x xml

set @x=''

select @x.query('xs:date("2000-01-01Z")')
```

You can also use constructors for the user-defined atomic types. For example, if the XML schema collection associated with the XML data type defines a simple type, a **myType()** constructor can be used to return a value of that type.

Implementation Limitations

- The XQuery expressions **typeswitch**, **castable**, and **treat** are not supported.
- **cast as** requires a question mark (?)after the atomic type.
- **xs:QName** is not supported as a type for casting. Use **expanded-QName** instead.
- **xs:date**, **xs:time**, and **xs:datetime** require a time zone, which is indicated by a Z.

  The following query fails, because time zone is not specified.

  ```
  DECLARE @var XML

  SET @var = ''

  SELECT @var.query(' <a>{xs:date("2002-05-25")}</a>')

  go
  ```

  By adding the Z time zone indicator to the value, the query works.

  ```
  DECLARE @var XML

  SET @var = ''

  SELECT @var.query(' <a>{xs:date("2002-05-25Z")}</a>')

  go
  ```

  This is the result:

  ```
  <a>2002-05-25Z</a>
  ```

See Also

XQuery Expressions

Type System (XQuery)

Validate Expressions

In this implementation, the **validate** expression is not supported. The results of XQuery construction expressions are always untyped. If the result of an XQuery expression should be typed, use the SQL CAST expression to cast the result to an **xml** data type with the preferred schema collection.

See Also

XQuery Expressions

XQuery Expressions

Modules and Prologs

XQuery Prolog is a series of namespace declarations. In using the declare namespace in prolog, you can specify prefix to namespace binding and use the prefix in the query body.

Implementation Limitations

The following XQuery specifications are not supported in this implementation:

- Module declaration (`version`)
- Module declaration (`module namespace`)
- Xmpspace declaration (`xmlspace`)
- Default collation declaration (`declare default collation`)
- Base URI declaration (`declare base-uri`)
- Construction declaration (`declare construction`)
- Default ordering declaration (`declare ordering`)
- Schema import (`import schema namespace`)
- Module import (`import module`)
- Variable declaration in the prolog (`declare variable`)
- Function declaration (`declare function`)

In This Section

XQuery Language Reference (Database Engine)

Describes the XQuery prolog.

See Also

XQuery Against the XML Data Type

XQuery Prolog

An XQuery query is made up of a prolog and a body. The XQuery prolog is a series of declarations and definitions that together create the required environment for query processing. In SQL Server, the XQuery prolog can include namespace declarations. The XQuery body is made up of a sequence of expressions that specify the intended query result.

For example, the following XQuery is specified against the Instructions column of **xml** type that stores manufacturing instructions as XML. The query retrieves the manufacturing instructions for the work center location `10`. The `query()` method of the **xml** data type is used to specify the XQuery.

```
SELECT Instructions.query('declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
    /AWMI:root/AWMI:Location[@LocationID=10]
') AS Result
FROM  Production.ProductModel
WHERE ProductModelID=7
```

Note the following from the previous query:

- The XQuery prolog includes a namespace prefix (AWMI) declaration, `(namespace AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManuInstructions";.`
- The `declare namespace` keyword defines a namespace prefix that is used later in the query body.
- `/AWMI:root/AWMI:Location[@LocationID="10"]` is the query body.

Namespace Declarations

A namespace declaration defines a prefix and associates it with a namespace URI, as shown in the following query. In the query, `CatalogDescription` is an **xml** type column.

In specifying XQuery against this column, the query prolog specifies the `declare namespace` declaration to associate the prefix `PD`, product description, with the namespace URI. This prefix is then used in the query body instead of the namespace URI.

The nodes in the resulting XML are in the namespace associated with the namespace URI.

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
         /PD:ProductDescription/PD:Summary
    ') as Result
FROM Production.ProductModel
where ProductModelID=19
```

To improve query readability, you can declare namespaces by using WITH XMLNAMESPACES instead of declaring prefix and namespace binding in the query prolog by using `declare namespace`.

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS PD)

SELECT CatalogDescription.query('
         /PD:ProductDescription/PD:Summary
    ') as Result
FROM Production.ProductModel
where ProductModelID=19
```

For more information, see, Adding Namespaces Using WITH XMLNAMESPACES.

Default Namespace Declaration

Instead of declaring a namespace prefix by using the `declare namespace` declaration, you can use the `declare default element namespace` declaration to bind a default namespace for element names. In this case, you do not specify any prefix.

In the following example, the path expression in the query body does not specify a namespace prefix. By default, all element names belong to the default namespace specified in the prolog.

```
SELECT CatalogDescription.query('
     declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
        /ProductDescription/Summary
    ') as Result
```

```
FROM  Production.ProductModel
WHERE ProductModelID=19
```

You can declare a default namespace by using WITH XMLNAMESPACES:

```
WITH XMLNAMESPACES (DEFAULT
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription')
SELECT CatalogDescription.query('
        /ProductDescription/Summary
    ') as Result
FROM  Production.ProductModel
WHERE ProductModelID=19
```

See Also

Adding Namespaces Using WITH XMLNAMESPACES

Type Casting Rules in XQuery

The following W3C XQuery 1.0 and XPath 2.0 Functions and Operators specifications diagram shows the built-in data types. This includes the built-in primitive and built-in derived types.

This topic describes the type casting rules that are applied when casting from one type to another by using one of the following methods:

- Explicit casting that you do by using **cast as** or the type constructor functions (for example, `xs:integer("5")`).
- Implicit casting that occurs during type promotion

Explicit Casting

The following table outlines the allowed type casting between the built-in primitive types.

- A built-in primitive type can cast to another built-in primitive type, based on the rules in the table.
- A primitive type can be cast to any type derived from that primitive type. For example, you can cast from **xs:decimal** to **xs:integer**, or from **xs:decimal** to **xs:long**.

- A derived type can be cast to any type that is its ancestor in the type hierarchy, all the way up to its built-in primitive base type. For example, you can cast from **xs:token** to **xs:normalizedString** or to **xs:string**.
- A derived type can be cast to a primitive type if its primitive ancestor can be cast to the target type. For example, you can cast **xs:integer**, a derived type, to an **xs:string**, primitive type, because **xs:decimal**, **xs:integer**'s primitive ancestor, can be cast to **xs:string**.
- A derived type can be cast to another derived type if the source type's primitive ancestor can be cast to the target type's primitive ancestor. For example, you can cast from **xs:integer** to **xs:token**, because you can cast from **xs:decimal** to **xs:string**.
- The rules for casting user-defined types to built-in types are the same as for the built-in types. For example, you can define a **myInteger** type derived from **xs:integer** type. Then, **myInteger** can be cast to **xs:token**, because **xs:decimal** can be cast to **xs:string**.

The following kinds of casting are not supported:

- Casting to or from list types is not allowed. This includes both user-defined list types and built-in list types such as **xs:IDREFS**, **xs:ENTITIES**, and **xs:NMTOKENS**.
- Casting to or from **xs:QName** is not supported.
- **xs:NOTATION** and the fully ordered subtypes of duration, **xdt:yearMonthDuration** and **xdt:dayTimeDuration**, are not supported. As a result, casting to or from these types is not supported.

The following examples illustrate explicit type casting.

Example A

The following example queries an xml type variable. The query returns a sequence of a simple type value typed as xs:string.

```
declare @x xml
set @x = '<e>1</e><e>2</e>'
select @x.query('/e[1] cast as xs:string?')
go
```

Example B

The following example queries a typed xml variable. The example first creates an XML schema collection. It then uses the XML schema collection to create a typed xml variable. The schema provides the typing information for the XML instance assigned to the variable. Queries are then specified against the variable.

```
create xml schema collection myCollection as N'
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="root">
```

```
            <xs:complexType>
                  <xs:sequence>
                        <xs:element name="A" type="xs:string"/>
                        <xs:element name="B" type="xs:string"/>
                        <xs:element name="C" type="xs:string"/>
                  </xs:sequence>
            </xs:complexType>
      </xs:element>
</xs:schema>'
go
```

The following query returns a static error, because you do not know how many top level <root> elements are in the document instance.

```
declare @x xml(myCollection)
set @x = '<root><A>1</A><B>2</B><C>3</C></root>
          <root><A>4</A><B>5</B><C>6</baz></C>'
select @x.query('/root/A cast as xs:string?')
go
```

By specifying a singleton <root> element in the expression, the query succeeds. The query returns a sequence of a simple type value typed as xs:string.

```
declare @x xml(myCollection)
set @x = '<root><A>1</A><B>2</B><C>3</C></root>
              <root><A>4</A><B>5</B><C>6</C></root>'
select @x.query('/root[1]/A cast as xs:string?')
go
```

In the following example, the xml type variable includes a document keyword that specifies the XML schema collection. This indicates that the XML instance must be a document that has a single top-level element. If you create two <root> elements in the XML instance, it will return an error.

```
declare @x xml(document myCollection)
set @x = '<root><A>1</A><B>2</B><C>3</C></root>
              <root><A>4</A><B>5</B><C>6</C></root>'
go
```

You can change the instance to include only one top level element and the query works. Again, the query returns a sequence of a simple type value typed as xs:string.

```
declare @x xml(document myCollection)
```

```
set @x = '<root><A>1</A><B>2</B><C>3</C></root>'
select @x.query('/root/A cast as xs:string?')
go
```

Implicit Casting

Implicit casting is allowed only for numeric types and untyped atomic types. For example, the following **min()** function returns the minimum of the two values:

```
min(xs:integer("1"), xs:double("1.1"))
```

In this example, the two values passed in to the XQuery **min()** function are of different types. Therefore, implicit conversion is performed where **integer** type is promoted to **double** and the two **double** values are compared.

The type promotion as described in this example follows these rules:

- A built-in derived numeric type may be promoted to its base type. For example, **integer** may be promoted to **decimal**.
- A **decimal** may be promoted to **float,** and a **float** may be promoted to **double**.

Because implicit casting is allowed only for numeric types, the following is not allowed:

- Implicit casting for string types is not allowed. For example, if two **string** types are expected and you pass in a **string** and a **token**, no implicit casting occurs and an error is returned.
- Implicit casting from numeric types to string types is not allowed. For example, if you pass an integer type value to a function that is expecting a string type parameter, no implicit casting occurs and an error is returned.

Casting values

When casting from one type to another, the actual values are transformed from the source type's value space to the target type's value space. For example, casting from an xs:decimal to an xs:double will transform the decimal value into a double value.

Following are some of the transformation rules.

Casting a value from a string or untypedAtomic type

The value that is being cast to a string or untypedAtomic type is transformed in the same manner as validating the value based on the target type's rules. This includes eventual pattern and white-space processing rules. For example, the following will succeed and generate a double value, 1.1e0:

```
xs:double("1.1")
```

When casting to binary types such as xs:base64Binary or xs:hexBinary from a string or untypedAtomic type, the input values have to be base64 or hex encoded, respectively.

Casting a value to a string or untypedAtomic type

Casting to a string or untypedAtomic type transforms the value to its XQuery canonical lexical representation. Specifically, this can mean that a value that may have obeyed a specific pattern or other constraint during input will not be represented according to that constraint. To inform users about this, SQL Server flags types where the type constraint can be a problem by providing a warning when those types are loaded into the schema collection.

When casting a value of type xs:float or xs:double, or any one of their subtypes, to a string or untypedAtomic type, the value is represented in scientific notation. This is done only when the value's absolute value is less than 1.0E-6, or greater than or equal to 1.0E6. This means that 0 is serialized in scientific notation to 0.0E0.

For example, `xs:string(1.11e1)` will return the string value `"11.1"`, while `xs:string(-0.00000000002e0)` will return the string value, `"-2.0E-11"`.

When casting binary types, such as xs:base64Binary or xs:hexBinary, to a string or untypedAtomic type, the binary values will be represented in their base64 or hex encoded form, respectively.

Casting a value to a numeric type

When casting a value of one numeric type to a value of another numeric type, the value is mapped from one value space to the other without going through string serialization. If the value does not satisfy the constraint of a target type, the following rules apply:

- If the source value is already numeric and the target type is either xs:float or a subtype thereof that allows -INF or INF values, and casting of the source numeric value would result in an overflow, the value is mapped to INF if the value is positive or -INF if the value is negative. If the target type does not allow INF or -INF, and an overflow would occur, the cast fails and the result in this release of SQL Server is the empty sequence.
- If the source value is already numeric and the target type is a numeric type that includes 0, -0e0, or 0e0 in its accepted value range, and casting of the source numeric value would result in an underflow, the value is mapped in the following ways:
  - The value is mapped to 0 for a decimal target type.
  - The value is mapped to -0e0 when the value is a negative underflow.
  - The value is mapped to 0e0 when the value is a positive underflow for a float or double target type.

  If the target type does not include zero in its value space, the cast fails and the result is the empty sequence.

Note that casting a value to a binary floating point type, such as xs:float, xs:double, or any one of their subtypes, may lose precision.

Implementation Limitations

These are the limitations:

- The floating point value NaN is not supported.
- Castable values are restricted by the target types implementation restrictions. For example, you cannot cast a date string with a negative year to **xs:date**. Such casts will result in the empty sequence if the value is provided at run time (instead of raising a run-time error).

See Also

Serialization of XML Data

XQuery Functions against the xml Data Type

This topic and its subtopics describe the functions you can use when specifying XQuery against the **xml** data type. For the W3C specifications, see http://www.w3.org/TR/2004/WD-xpath-functions-20040723.

The XQuery functions belong to the http://www.w3.org/2004/07/xpath-functions namespace. The W3C specifications use the "fn:" namespace prefix to describe these functions. You do not have to specify the "fn:" namespace prefix explicitly when you are using the functions. Because of this and to improve readability, the namespace prefixes are generally not used in this documentation.

The following table lists the XQuery functions that are supported against the **xml** data type.

Category	Function Name
Implementing XML in SQL Server	ceiling
	floor
	round
Functions on String Values	concat
	contains
	substring
	lower-case Function (XQuery)

Category	Function Name
	string-length
	upper-case Function (XQuery)
Functions on Boolean Values	not
Functions on Nodes	number
	local-name Function (XQuery)
	namespace-uri Function (XQuery)
Context Functions	last
	position
Functions on Sequences	empty
	distinct-values
	id Function (XQuery)
Aggregate Functions (XQuery)	count
	avg
	min
	max
	sum
Constructor Functions (XQuery)	Constructor Functions
Data Accessor Functions	string
	data
Boolean Constructor Functions (XQuery)	true Function (XQuery)
	false Function (XQuery)
Functions Related to QNames (XQuery)	expanded-QName (XQuery)
	local-name-from-QName (XQuery)
	namespace-uri-from-QName (XQuery)
SQL Server XQuery Extension Functions	sql:column() function (XQuery)
	sql:variable() function (XQuery)

See Also

xml Data Type Methods

XQuery Against the XML Data Type

XML Data (SQL Server)

Functions on Numeric Values

The following topics discuss and provide sample code for the functions on numeric values.

In This Section

Category	Function Name
Functions on numeric values	ceiling
	floor
	round

ceiling Function

Returns the smallest number without a fractional part and that is not less than the value of its argument. If the argument is an empty sequence, it returns the empty sequence.

Syntax

```
fn:ceiling ( $arg as numeric?) as numeric?
```

Arguments

$arg

Number to which the function is applied.

Remarks

If the type of $arg is one of the three numeric base types, **xs:float**, **xs:double**, or **xs:decimal**, the return type is the same as the $arg type.

If the type of $arg is a type that is derived from one of the numeric types, the return type is the base numeric type.

If the input to the fn:floor, fn:ceiling, or fn:round functions is **xdt:untypedAtomic**, it is implicitly cast to **xs:double**.

Any other type generates a static error.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the ceiling() XQuery function

For product model 7, this query returns a list of the work center locations in the manufacturing process of the product model. For each work center location, the query returns the location ID, labor hours, and lot size, if documented. The query uses the **ceiling** function to return the labor hours as values of type **decimal**.

```
SELECT ProductModelID, Instructions.query('
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
     for $i in /AWMI:root/AWMI:Location
     return
       <Location LocationID="{ $i/@LocationID }"
                  LaborHrs="{ ceiling($i/@LaborHours) }" >
                   {
                     $i/@LotSize
                   }
       </Location>
') AS Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

Note the following from the previous query:

- The AWMI namespace prefix stands for Adventure Works Manufacturing Instructions. This prefix refers to the same namespace used in the document being queried.
- **Instructions** is an **xml** type column. Therefore, the query() method (XML data type) is used to specify XQuery. The XQuery statement is specified as the argument to the query method.
- **for ... return** is a loop construct. In the query, the **for** loop identifies a list of <Location> elements. For each work center location, the **return** statement in the **for** loop describes the XML to be generated:
  - A <Location> element that has LocationID and LaborHrs attributes. The corresponding expression inside the braces ({ }) retrieves the required values from the document.
  - The { $i/@LotSize } expression retrieves the LotSize attribute from the document, if present.
  - This is the result:

```
ProductModelID Result
-------------- -----------------------------------------------------
7      <Location LocationID="10" LaborHrs="3" LotSize="100"/>
       <Location LocationID="20" LaborHrs="2" LotSize="1"/>
       <Location LocationID="30" LaborHrs="1" LotSize="1"/>
       <Location LocationID="45" LaborHrs="1" LotSize="20"/>
       <Location LocationID="60" LaborHrs="3" LotSize="1"/>
       <Location LocationID="60" LaborHrs="4" LotSize="1"/>
```

Implementation Limitations

These are the limitations:

- The **ceiling()** function maps all integer values to xs:decimal.

See Also

floor function (XQuery)

round function (XQuery)

floor Function

Returns the largest number with no fraction part that is not greater than the value of its argument. If the argument is an empty sequence, it returns the empty sequence.

Syntax

```
fn:floor ($arg as numeric?) as numeric?
```

Arguments

$arg

Number to which the function is applied.

Remarks

If the type of $arg is one of the three numeric base types, **xs:float**, **xs:double**, or **xs:decimal**, the return type is same as the $arg type. If the type of $arg is a type that is derived from one of the numeric types, the return type is the base numeric type.

If input to the fn:floor, fn:ceiling, or fn:round functions is **xdt:untypedAtomic**, untyped data, it is implicitly cast to **xs:double**. Any other type generates a static error.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks sample database.

You can use the working sample in the ceiling function (XQuery) for the **floor()** XQuery function. All you have to do is replace the **ceiling()** function in the query with the **floor()** function.

Implementation Limitations

These are the limitations:

- The **floor()** function maps all integer values to xs:decimal.

See Also

ceiling function (XQuery)

round function (XQuery)

XQuery Functions Against the XML Data Type

round Function

Returns the number not having a fractional part that is closest to the argument. If there is more than one number like that, the one that is closest to positive infinity is returned. For example:

If the argument is 2.5, **round()** returns 3.

If the argument is 2.4999, **round()** returns 2.

If the argument is -2.5, **round()** returns -2.

If the argument is an empty sequence, **round()** returns the empty sequence.

Syntax

```
fn:round ( $arg as numeric?) as numeric?
```

Arguments

$arg

Number to which the function is applied.

Remarks

If the type of $arg is one of the three numeric base types, **xs:float**, **xs:double**, or **xs:decimal**, the return type is same as the $arg type. If the type of $arg is a type that is derived from one of the numeric types, the return type is the base numeric type.

If input to the **fn:floor**, **fn:ceiling**, or **fn:round** functions is **xdt:untypedAtomic**, untyped data, it is implicitly cast to **xs:double**.

Any other type generates a static error.

Examples

This topic provides XQuery examples against XML instances stored in various **xml** type columns in the AdventureWorks database.

You can use the working sample in the ceiling function (XQuery) for the **round()** XQuery function. All you have to do is replace the **ceiling()** function in the query with the **round()** function.

Implementation Limitations

These are the limitations:

- The **round()** function maps integer values to xs:decimal.
- The **round()** function of xs:double and xs:float values between -0.5e0 and -0e0 are mapped to 0e0 instead of -0e0.

See Also

floor function (XQuery)

ceiling function (XQuery)

XQuery Functions on String Values

The following topics discuss and provide sample code for the functions on string values.

In This Section

Category	Function Name
Functions on string values	concat
	contains
	substring
	string-length
	upper-case Function (XQuery)
	lower-case Function (XQuery)

concat Function

Accepts zero or more strings as arguments and returns a string created by concatenating the values of each of these arguments.

Syntax

```
fn:concat ($string as xs:string?
        ,$string as xs:string?
        [, ...]) as xs:string
```

Arguments

$string

Optional string to concatenate.

Remarks

The function requires at least two arguments. If an argument is an empty sequence, it is treated as the zero-length string.

Supplementary Characters (Surrogate Pairs)

The behavior of surrogate pairs in XQuery functions depends on the database compatibility level and, in some cases, on the default namespace URI for functions. For more information, see the section "XQuery Functions Are Surrogate-Aware" in the topic Breaking Changes to Database Engine Features in SQL Server 2012. Also see ALTER DATABASE Compatibility Level (Transact-SQL) and Collation and Unicode Support.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks sample database.

A. Using the concat() XQuery function to concatenate strings

For a specific product model, this query returns a string created by concatenating the warranty period and warranty description. In the catalog description document, the `<Warranty>` element is made up of `<WarrantyPeriod>` and `<Description>` child elements.

```
WITH XMLNAMESPACES (
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS pd,
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain' AS wm)
SELECT CatalogDescription.query('
    <Product
        ProductModelID= "{ (/pd:ProductDescription/@ProductModelID)[1]
}"
        ProductModelName = "{ sql:column("PD.Name") }" >
        {
          concat(
string((/pd:ProductDescription/pd:Features/wm:Warranty/wm:WarrantyPerio
d)[1]), "-",

string((/pd:ProductDescription/pd:Features/wm:Warranty/wm:Description)[
1]))
         }
     </Product>
 ') as Result
FROM Production.ProductModel PD
```

```
WHERE  PD.ProductModelID=28
```

Note the following from the previous query:

- In the SELECT clause, CatalogDescription is an **xml** type column. Therefore, the query() method (XML data type), Instructions.query(), is used. The XQuery statement is specified as the argument to the query method.
- The document against which the query is executed uses namespaces. Therefore, the **namespace** keyword is used to define the prefix for the namespace. For more information, see XQuery Prolog.

This is the result:

```
<Product ProductModelID="28" ProductModelName="Road-450">1 year-parts
and labor</Product>
```

The previous query retrieves information for a specific product. The following query retrieves the same information for all the products for which XML catalog descriptions are stored. The **exist()** method of the **xml** data type in the WHERE clause returns True if the XML document in the rows has a `<ProductDescription>` element.

```
WITH XMLNAMESPACES (
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS pd,
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain' AS wm)

SELECT CatalogDescription.query('
    <Product
        ProductModelID= "{ (/pd:ProductDescription/@ProductModelID)[1]
}"
        ProductName = "{ sql:column("PD.Name") }" >
        {
          concat(
string((/pd:ProductDescription/pd:Features/wm:Warranty/wm:WarrantyPerio
d)[1]), "-",

string((/pd:ProductDescription/pd:Features/wm:Warranty/wm:Description)[
1]))
         }
     </Product>
```

```
 ') as Result
FROM Production.ProductModel PD
WHERE CatalogDescription.exist('//pd:ProductDescription ') = 1
```

Note that the Boolean value returned by the **exist()** method of the **xml** type is compared with 1.

Implementation Limitations

These are the limitations:

- The **concat()** function in SQL Server only accepts values of type xs:string. Other values have to be explicitly cast to xs:string or xdt:untypedAtomic.

See Also

XQuery Functions Against the XML Data Type

contains Function

Returns a value of type xs:boolean indicating whether the value of $arg1 contains a string value specified by $arg2.

Syntax

```
fn:contains ($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean?
```

Arguments

$arg1

String value to test.

$arg2

Substring to look for.

Remarks

If the value of $arg2 is a zero-length string, the function returns **True**. If the value of $arg1 is a zero-length string and the value of $arg2 is not a zero-length string, the function returns **False**.

If the value of $arg1 or $arg2 is the empty sequence, the argument is treated as the zero-length string.

The contains() function uses XQuery's default Unicode code point collation for the string comparison.

The substring value specified for $arg2 has to be less than or equal to 4000 characters. If the value specified is greater than 4000 characters, a dynamic error condition occurs and the contains() function returns an empty sequence instead of a Boolean value of **True** or **False**. SQL Server does not raise dynamic errors on XQuery expressions.

In order to get case-insensitive comparisons, the upper-case or lower-case functions can be used.

Supplementary Characters (Surrogate Pairs)

The behavior of surrogate pairs in XQuery functions depends on the database compatibility level and, in some cases, on the default namespace URI for functions. For more information, see the section "XQuery Functions Are Surrogate-Aware" in the topic Breaking Changes to Database Engine Features in SQL Server 2012. Also see ALTER DATABASE Compatibility Level (Transact-SQL) and Collation and Unicode Support.

Examples

This topic provides XQuery examples against XML instances stored in various xml- type columns in the AdventureWorks database.

A. Using the contains() XQuery function to search for a specific character string

The following query finds products that contain the word Aerodynamic in the summary descriptions. The query returns the ProductID and the `<Summary>` element for such products.

```
--The product model description document uses
--namespaces. The WHERE clause uses the exit()
--method of the xml data type. Inside the exit method,
--the XQuery contains()function is used to
--determine whether the <Summary> text contains the word
--Aerodynamic.

USE AdventureWorks
GO
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS pd)
SELECT ProductModelID, CatalogDescription.query('
      <Prod>
         { /pd:ProductDescription/@ProductModelID }
         { /pd:ProductDescription/pd:Summary }
      </Prod>
 ') as Result
FROM Production.ProductModel
where CatalogDescription.exist('
```

```
    /pd:ProductDescription/pd:Summary//text()
     [contains(., "Aerodynamic")]') = 1
```

Results

```
ProductModelID Result
-------------- ---------
28     <Prod ProductModelID="28">
       <pd:Summary xmlns:pd=
        "http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription">
        <p1:p xmlns:p1="http://www.w3.org/1999/xhtml">
         A TRUE multi-sport bike that offers streamlined riding and
         a revolutionary design. Aerodynamic design lets you ride with
        the pros, and the gearing will conquer hilly roads.</p1:p>
       </pd:Summary>
      </Prod>
```

See Also

XQuery Functions Against the XML Data Type

substring Function

Returns part of the value of $sourceString, starting at the position indicated by the value of $startingLoc, and continues for the number of characters indicated by the value of $length.

Syntax

```
fn:substring($sourceString as xs:string?,
             $startingLoc  as as xs:decimal?) as xs:string?

fn:substring($sourceString as xs:string?,
             $startingLoc as xs:decimal?,
             $length as xs:decimal?) as xs:string?
```

Arguments

$sourceString

Source string.

$startingLoc

Starting point in the source string from which the substring starts. If this value is negative or 0, only those characters in positions greater than zero are returned. If it is

greater than the length of the $sourceString, the zero-length string is returned.

$length

[optional] Number of characters to retrieve. If not specified, it returns all the characters from the location specified in $startingLoc up to the end of string.

Remarks

The three-argument version of the function returns the characters in `$sourceString` whose position `$p` obeys:

```
fn:round($startingLoc) <= $p < fn:round($startingLoc) +
fn:round($length)
```

The value of $length can be greater than the number of characters in the value of $sourceString following the start position. In this case, the substring returns the characters up to the end of $sourceString.

The first character of a string is located at position 1.

If the value of $sourceString is the empty sequence, it is handled as the zero-length string. Otherwise, if either $startingLoc or $length is the empty sequence, the empty sequence is returned.

Supplementary Characters (Surrogate Pairs)

The behavior of surrogate pairs in XQuery functions depends on the database compatibility level and, in some cases, on the default namespace URI for functions. For more information, see the section "XQuery Functions Are Surrogate-Aware" in the topic Breaking Changes to Database Engine Features in SQL Server 2012. Also see ALTER DATABASE Compatibility Level (Transact-SQL) and Collation and Unicode Support.

Implementation Limitations

SQL Server requires the $startingLoc and $length parameters to be of type xs:decimal instead of xs:double.

SQL Server allows $startingLoc and $length to be the empty sequence, because the empty sequence is a possible value as a result of dynamic errors being mapped to ().

Examples

This topic provides XQuery examples against XML instances stored in various **xml** type columns in the database.

A. Using the substring() XQuery function to retrieve partial summary product-model descriptions

The query retrieves the first 50 characters of the text that describes the product model, the `<Summary>` element in the document.

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS pd)
SELECT ProductModelID, CatalogDescription.query('
```

```
    <Prod>{ substring(string((/pd:ProductDescription/pd:Summary)[1]),
1, 50) }</Prod>
 ') as Result
FROM Production.ProductModel
where CatalogDescription.exist('/pd:ProductDescription')  = 1;
```

Note the following from the previous query:

- The **string()** function returns the string value of the `<Summary>` element. This function is used, because the `<Summary>` element contains both the text and subelements (html formatting elements), and because you will skip these elements and retrieve all the text.
- The **substring()** function retrieves the first 50 characters from the string value retrieved by the **string()**.

This is a partial result:

```
ProductModelID Result
-------------- ----------------------------------------------------
19      <Prod>Our top-of-the-line competition mountain bike.</Prod>
23      <Prod>Suitable for any type of riding, on or off-roa</Prod>
...
```

See Also

XQuery Functions Against the XML Data Type

string-length Function

Returns the length of the string in characters.

Syntax

```
fn:string-length() as xs:integer
fn:string-length($arg as xs:string?) as xs:integer
```

Arguments

$arg

Source string whose length is to be computed.

Remarks

If the value of $arg is an empty sequence, an **xs:integer** value of 0 is returned.

The behavior of surrogate pairs in XQuery functions depends on the database compatibility level. If the compatibility level is 110 or later, each surrogate pair is counted as a single character. For earlier compatibility levels, they are counted as two characters.

For more information, see ALTER DATABASE Compatibility Level (Transact-SQL) and Collation and Unicode Support.

If the value contains a 4-byte Unicode character that is represented by two surrogate characters, SQL Server will count the surrogate characters individually.

The **string-length()** without a parameter can only be used inside a predicate. For example, the following query returns the <ROOT> element:

```
DECLARE @x xml;

SET @x='<ROOT>Hello</ROOT>';

SELECT @x.query('/ROOT[string-length()=5]');
```

Supplementary Characters (Surrogate Pairs)

The behavior of surrogate pairs in XQuery functions depends on the database compatibility level and, in some cases, on the default namespace URI for functions. For more information, see the section "XQuery Functions Are Surrogate-Aware" in the topic Breaking Changes to Database Engine Features in SQL Server 2012. Also see ALTER DATABASE Compatibility Level (Transact-SQL) and Collation and Unicode Support.

Examples

This topic provides XQuery examples against XML instances stored in various **xml** type columns in the AdventureWorks database.

A. Using the string-length() XQuery function to retrieve products with long summary descriptions

For products whose summary description is greater than 50 characters, the following query retrieves the product ID, the length of the summary description, and the summary itself, the <Summary> element.

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' as pd)

SELECT CatalogDescription.query('

      <Prod ProductID= "{ /pd:ProductDescription[1]/@ProductModelID }"
>

       <LongSummary SummaryLength =

           "{string-length(string(
(/pd:ProductDescription/pd:Summary)[1] )) }" >

           { string( (/pd:ProductDescription/pd:Summary)[1] ) }

       </LongSummary>

      </Prod>

 ') as Result

FROM Production.ProductModel
```

```
WHERE CatalogDescription.value('string-length( string(
(/pd:ProductDescription/pd:Summary)[1]))', 'decimal') > 200;
```

Note the following from the previous query:

- The condition in the WHERE clause retrieves only the rows where the summary description stored in the XML document is longer than 200 characters. It uses the value() method (XML data type).
- The SELECT clause just constructs the XML that you want. It uses the query() method (XML data type) to construct the XML and specify the necessary XQuery expression to retrieve data from the XML document.

This is a partial result:

```
Result
------------------
<Prod ProductID="19">
      <LongSummary SummaryLength="214">Our top-of-the-line competition
            mountain bike. Performance-enhancing options include the
            innovative HL Frame, super-smooth front suspension, and
            traction for all terrain.
      </LongSummary>
</Prod>
...
```

B. Using the string-length() XQuery function to retrieve products whose warranty descriptions are short

For products whose warranty descriptions are less than 20 characters long, the following query retrieves XML that includes the product ID, length, warranty description, and the <Warranty> element itself.

Warranty is one of the product features. An optional <Warranty> child element follows after the <Features> element.

```
WITH XMLNAMESPACES (
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS pd,
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain' AS wm)

SELECT CatalogDescription.query('
      for   $ProdDesc in /pd:ProductDescription,
            $pf in $ProdDesc/pd:Features/wm:Warranty
```

```
      where string-length( string(($pf/wm:Description)[1]) ) < 20
      return
          <Prod >
             { $ProdDesc/@ProductModelID }
             <ShortFeature FeatureDescLength =
                           "{string-length(
string(($pf/wm:Description)[1]) ) }" >
                 { $pf }
             </ShortFeature>
          </Prod>
     ') as Result
FROM Production.ProductModel
WHERE CatalogDescription.exist('/pd:ProductDescription')=1;
```

Note the following from the previous query:

- **pd** and **wm** are the namespace prefixes used in this query. They identify the same namespaces used in the document that is being queried.
- The XQuery specifies a nested FOR loop. The outer FOR loop is required, because you want to retrieve the **ProductModelID** attributes of the `<ProductDescription>` element. The inner FOR loop is required, because you want only those products that have warranty feature descriptions that are less than 20 characters long.

This is the partial result:

```
Result
-------------------------
<Prod ProductModelID="19">
  <ShortFeature FeatureDescLength="15">
    <wm:Warranty

xmlns:wm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
      <wm:WarrantyPeriod>3 years</wm:WarrantyPeriod>
      <wm:Description>parts and labor</wm:Description>
    </wm:Warranty>
   </ShortFeature>
</Prod>
...
```

See Also

XQuery Functions Against the XML Data Type

lower-case Function

The lower-case function converts each character in $arg to its lower case equivalent. The Microsoft Windows binary case conversion for Unicode code points specifies how characters are converted to lower case. This standard is not identical to the mapping for Unicode code point standard.

Syntax

```
fn:lower-case($arg as xs:string?) as xs:string
```

Arguments

Term	Definition
$arg	The string value to be converted to lower case.

Remarks

If the value of $arg is empty, a zero length string is returned.

Examples

A. Changing a string to upper case

The following example changes the input string 'abcDEF!@4' to lower case.

```
DECLARE @x xml = N'abcDEF!@4';
SELECT @x.value('fn:lower-case(/text()[1])', 'nvarchar(10)');
```

Here is the result set.

abcdef!@4

B. Search for a specific character string

This example shows you how to use the lower-case function to perform a case insensitive search.

```
USE AdventureWorks
GO
--WITH XMLNAMESPACES clause specifies the namespace prefix
--to use.
```

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS pd)
--The XQuery contains() function is used to determine whether
--any of the text nodes below the <Summary> element contain
--the word 'frame'. The lower-case() function makes the
--case insensitive.

SELECT ProductModelID, CatalogDescription.query('
      <Prod>
         { /pd:ProductDescription/@ProductModelID }
         { /pd:ProductDescription/pd:Summary }
      </Prod>
 ') as Result
FROM Production.ProductModel
where CatalogDescription.exist('
/pd:ProductDescription/pd:Summary//text()[
          contains(lower-case(.), "FRAME")]')  = 1
```

Here is the result set.

ProductModelID Result

---------- ---------

19 <Prod ProductModelID="19">
<pd:Summary xmlns:pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelDescription">
<p1:p xmlns:p1="http://www.w3.org/1999/xhtml">Our top-of-the-line competition mountain bike.
Performance-enhancing options include the innovative HL Frame,
super-smooth front suspension, and traction for all terrain.
</p1:p>
</pd:Summary>
</Prod>

25 <Prod ProductModelID="25">
<pd:Summary xmlns:pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelDescription">

<p1:p xmlns:p1="http://www.w3.org/1999/xhtml">This bike is ridden by race winners. Developed with the
Adventure Works Cycles professional race team, it has a extremely light
heat-treated aluminum frame, and steering that allows precision control.
</p1:p>
</pd:Summary>
</Prod>

See Also

XQuery Functions Against the XML Data Type

upper-case Function

This function converts each character in $arg to its upper case equivalent. The Microsoft Windows binary case conversion for Unicode code points specifies how characters are converted to upper case. This standard is different than the mapping for Unicode standard code point standard.

Syntax

```
fn:upper-case($arg as xs:string?) as xs:string
```

Arguments

Term	Definition
$arg	The string value to be converted to upper case.

Remarks

If the value of $arg is empty, a zero length string is returned.

Examples

A. Changing a string to upper case

The following example changes the input string 'abcDEF!@4' to upper case.

```
DECLARE @x xml = N'abcDEF!@4';
SELECT @x.value('fn:upper-case(/text()[1])', 'nvarchar(10)');
```

B. Search for a Specific Character String

This example shows how to use the upper-case function to perform a case-insensitive search.

```
USE AdventureWorks
```

```
GO
--WITH XMLNAMESPACES clause specifies the namespace prefix
--to use.
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS pd)
--The XQuery contains() function is used to determine whether
--any of the text nodes below the <Summary> element contain
--the word 'frame'. The upper-case() function is used to make
--the search case-insensitive.

SELECT ProductModelID, CatalogDescription.query('
      <Prod>
         { /pd:ProductDescription/@ProductModelID }
         { /pd:ProductDescription/pd:Summary }
      </Prod>
 ') as Result
FROM Production.ProductModel
where CatalogDescription.exist('
/pd:ProductDescription/pd:Summary//text()[
          contains(upper-case(.), "FRAME")]')  = 1
```

Here is the result set.

ProductModelID Result

---------- ---------

19 <Prod ProductModelID="19">
<pd:Summary
xmlns:pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelDescription">
<p1:p xmlns:p1="http://www.w3.org/1999/xhtml">Our top-of-the-line competition mountain bike.
Performance-enhancing options include the innovative HL Frame,
super-smooth front suspension, and traction for all terrain.
</p1:p>
</pd:Summary>
</Prod>

```
25    <Prod ProductModelID="25">
        <pd:Summary
xmlns:pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription">
          <p1:p xmlns:p1="http://www.w3.org/1999/xhtml">This bike is ridden by race
winners. Developed with the
          Adventure Works Cycles professional race team, it has a extremely light
          heat-treated aluminum frame, and steering that allows precision control.
          </p1:p>
        </pd:Summary>
      </Prod>
```

See Also

XQuery Functions Against the XML Data Type

Functions on Boolean Values

The following topic discusses and provides sample code for the function on Boolean values.

In This Section

Category	Function name
Functions on Boolean values	not

See Also

Boolean Constructor Functions (XQuery)

not Function

Returns TRUE if the effective Boolean value of $arg is false, and returns FALSE if the effective Boolean value of $arg is true.

Syntax

```
fn:not($arg as item()*) as xs:boolean
```

Arguments

$arg

A sequence of items for which there is an effective Boolean value.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the not() XQuery function to find product models whose catalog descriptions do not include the <Specifications> element.

The following query constructs XML that contains product model IDs for product models whose catalog descriptions do not include the `<Specifications>` element.

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS pd)
SELECT ProductModelID, CatalogDescription.query('
       <Product
           ProductModelID="{ sql:column("ProductModelID") }"
        />
') as Result
FROM Production.ProductModel
WHERE CatalogDescription.exist('
     /pd:ProductDescription[not(pd:Specifications/*)]  '
     ) = 0
```

Note the following from the previous query:

- Because the document uses namespaces, the sample uses the WITH NAMESPACES statement. Another option is to use the **declare namespace** keyword in the XQuery Prolog to define the prefix.
- The query then constructs the XML that includes the `<Product>` element and its **ProductModelID** attribute.
- The WHERE clause uses the exist() method (XML data type) to filter the rows. The **exist()** method returns True if there are <ProductDescription> elements that do not have <Specification> child elements. Note the use of the **not()** function.

This result set is empty, because each product model catalog description includes the <Specifications> element.

B. Using the not() XQuery function to retrieve work center locations that do not have a MachineHours attribute

The following query is specified against the Instructions column. This column stores manufacturing instructions for the product models.

For a particular product model, the query retrieves work center locations that do not specify MachineHours. That is, the attribute **MachineHours** is not specified for the <Location> element.

```
SELECT ProductModelID, Instructions.query('
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions" ;
     for $i in /AWMI:root/AWMI:Location[not(@MachineHours)]
     return
       <Location LocationID="{ $i/@LocationID }"
                   LaborHrs="{ $i/@LaborHours }" >
        </Location>
') as Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

In the preceding query, note the following:

- The **declare namespace** in XQuery Prolog defines the Adventure Works manufacturing instructions namespace prefix. It represents the same namespace used in the manufacturing instructions document.
- In the query, the **not(@MachineHours)** predicate returns True if there is no **MachineHours** attribute.

This is the result:

```
ProductModelID Result
-------------- -----------------------------------------
7              <Location LocationID="30" LaborHrs="1"/>
               <Location LocationID="50" LaborHrs="3"/>
               <Location LocationID="60" LaborHrs="4"/>
```

Implementation Limitations

These are the limitations:

- The **not()** function only supports arguments of type xs:boolean, or node()*, or the empty sequence.

See Also

XQuery Functions Against the XML Data Type

Functions on Nodes

The following topics discuss and provide sample code for the functions on nodes.

In This Section

Category	Function Name
Functions on nodes	number
	local-name
	namespace-uri

number Function

Returns the numeric value of the node that is indicated by $arg.

Syntax

```
fn:number() as xs:double?
fn:number($arg as node()?) as xs:double?
```

Arguments

$arg

Node whose value will be returned as a number.

Remarks

If $arg is not specified, the numeric value of the context node, converted to a double, is returned. In SQL Server, **fn:number()** without an argument can only be used in the context of a context-dependent predicate. Specifically, it can only be used inside brackets ([ ]). For example, the following expression returns the <ROOT> element.

```
declare @x xml
set @x='<ROOT>111</ROOT>'
select @x.query('/ROOT[number()=111]')
```

If the value of the node is not a valid lexical representation of a numeric simple type, as defined in **XML Schema Part 2:Datatypes, W3C Recommendation**, the function returns an empty sequence. NaN is not supported.

Examples

This topic provides XQuery examples against XML instances stored in various **xml** type columns in the AdventureWorks database.

A. Using the number() XQuery function to retrieve the numeric value of an attribute

The following query retrieves the numeric value of the lot size attribute from the first work center location in the manufacturing process of Product Model 7.

```
SELECT ProductModelID, Instructions.query('
```

```
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions" ;
     for $i in (//AWMI:root//AWMI:Location)[1]
     return
       <Location LocationID="{ ($i/@LocationID) }"
                 LotSizeA="{  $i/@LotSize }"
                 LotSizeB="{  number($i/@LotSize) }"
                 LotSizeC="{ number($i/@LotSize) + 1 }" >

       </Location>
') as Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

Note the following from the previous query:

- The **number()** function is not required, as shown by the query for the **LotSizeA** attribute. This is an XPath 1.0 function and is included primarily for backward compatibility reasons.
- The XQuery for **LotSizeB** specifies the number function and is redundant.
- The query for **LotSizeD** illustrates the use of a number value in an arithmetic operation.

This is the result:

```
ProductModelID   Result
-----------------------------------------------
7              <Location LocationID="10"
                         LotSizeA="100"
                         LotSizeB="100"
                         LotSizeC="101" />
```

Implementation Limitations

These are the limitations:

- The **number()** function only accepts nodes. It does not accept atomic values.
- When values cannot be returned as a number, the **number()** function returns the empty sequence instead of NaN.

See Also

XQuery Functions Against the XML Data Type

local-name Function

Returns the local part of the name of $arg as an xs:string that will either be the zero-length string or will have the lexical form of an xs:NCName. If the argument is not provided, the default is the context node.

Syntax

```
fn:local-name() as xs:string

fn:local-name($arg as node()?) as xs:string
```

Arguments

$arg

Node name whose local-name part will be retrieved.

Remarks

- In SQL Server, **fn:local-name()** without an argument can only be used in the context of a context-dependent predicate. Specifically, it can only be used inside brackets ([ ]).
- If the argument is supplied and is the empty sequence, the function returns the zero-length string.
- If the target node has no name, because it is a document node, a comment, or a text node, the function returns the zero-length string.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Retrieve local name of a specific node

The following query is specified against an untyped XML instance. The query expression, `local-name(/ROOT[1])`, retrieves the local name part of the specified node.

```
declare @x xml

set @x='<ROOT><a>111</a></ROOT>'

SELECT @x.query('local-name(/ROOT[1])')

-- result = ROOT
```

The following query is specified against the Instructions column, a typed xml column, of the ProductModel table. The expression, `local-name(/AWMI:root[1]/AWMI:Location[1])`, returns the local name, `Location`, of the specified node.

```
SELECT Instructions.query('

declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions" ;

     local-name(/AWMI:root[1]/AWMI:Location[1])') as Result
```

```
FROM Production.ProductModel
WHERE ProductModelID=7
-- result = Location
```

B. Using local-name without argument in a predicate

The following query is specified against the Instructions column, typed **xml** column, of the ProductModel table. The expression returns all the element children of the <root> element whose local name part of the QName is "Location". The **local-name()** function is specifed in the predicate and it has no arguments The context node is used by the function.

```
SELECT Instructions.query('
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions" ;
  /AWMI:root//*[local-name() = "Location"]') as Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

The query returns all the <Location> element children of the <root> element.

See Also

Functions on Nodes

namespace-uri Function (XQuery)

namespace-uri Function

Returns the namespace URI of the QName specified in $arg as a xs:string.

Syntax

```
fn:namespace-uri() as xs:string
fn:namespace-uri($arg as node()?) as xs:string
```

Arguments

$arg

Node name whose namespace URI part will be retrieved.

Remarks

- If the argument is omitted, the default is the context node.
- In SQL Server, **fn:namespace-uri()** without an argument can only be used in the context of a context-dependent predicate. Specifically, it can only be used inside brackets ([ ]).
- If $arg is the empty sequence, the zero-length string is returned.

- If $arg is an element or attribute node whose expanded-QName is not in a namespace, the function returns the zero-length string

Examples

This topic provides XQuery examples against XML instances stored in various **xml** type columns in the AdventureWorks database.

A. Retrieve namespace URI of a specific node

The following query is specified against an untyped XML instance. The query expression, `namespace-uri(/ROOT[1])`, retrieves the namespace URI part of the specified node.

```
set @x='<ROOT><a>111</a></ROOT>'

SELECT @x.query('namespace-uri(/ROOT[1])')
```

Because the specified QName does not have the namespace URI part but only the local name part, the result is a zero-length string.

The following query is specified against the Instructions typed **xml** column. The expression, `namespace-uri(/AWMI:root[1]/AWMI:Location[1])`, returns the namespace URI of the first `<Location>` element child of the `<root>` element.

```
SELECT Instructions.query('

declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions" ;

     namespace-uri(/AWMI:root[1]/AWMI:Location[1])') as Result

FROM Production.ProductModel

WHERE ProductModelID=7
```

This is the result:

```
http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions
```

B. Using namespace-uri() without argument in a predicate

The following query is specified against the CatalogDescription typed xml column. The expression returns all the element nodes whose namespace URI is `http://www.adventure-works.com/schemas/OtherFeatures`. The namespace-**uri()** function is specified without an argument and uses the context node.

```
SELECT CatalogDescription.query('

declare namespace
p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";

   /p1:ProductDescription//*[namespace-uri() = "http://www.adventure-
works.com/schemas/OtherFeatures"]

') as Result
```

```
FROM Production.ProductModel
WHERE ProductModelID=19
```

This is partial result:

```
<p1:wheel xmlns:p1="http://www.adventure-
works.com/schemas/OtherFeatures">High performance wheels.</p1:wheel>
<p2:saddle xmlns:p2="http://www.adventure-
works.com/schemas/OtherFeatures">
  <p3:i xmlns:p3="http://www.w3.org/1999/xhtml">Anatomic design</p3:i>
and made from durable leather for a full-day of riding in
comfort.</p2:saddle>
...
```

You can change the namespace URI in the previous query to `http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelWarrAndMain`. You will then receive all the element node children of the `<ProductDescription>` element whose namespace URI part of the expanded QName is `http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelWarrAndMain`.

Implementation Limitations

These are the limitations:

- The **namespace-uri()** function returns instances of type xs:string instead of xs:anyURI.

See Also

Functions on Nodes

local-name Function (XQuery)

Context Functions

The following topics discuss and provide sample code for the context functions.

In This Section

Category	Function Name
Context functions	last
	position

last Function

Returns the number of items in the sequence that is currently being processed. Specifically, it returns the integer index of the last item in the sequence. The first item in the sequence has an index value of 1.

Syntax

```
fn:last() as xs:integer
```

Remarks

In SQL Server, **fn:last()** can only be used in the context of a context-dependent predicate. Specifically, it can only be used inside brackets (`[ ]`).

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the last() XQuery function to retrieve the last two manufacturing steps

The following query retrieves the last two manufacturing steps for a specific product model. The value, the number of manufacturing steps, returned by the **last()** function is used in this query to retrieve the last two manufacturing steps.

```
SELECT ProductModelID, Instructions.query('
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
  <LastTwoManuSteps>
   <Last-1Step>
     { (/AWMI:root/AWMI:Location)[1]/AWMI:step[(last()-1)]/text() }
   </Last-1Step>
   <LastStep>
     { (/AWMI:root/AWMI:Location)[1]/AWMI:step[last()]/text() }
   </LastStep>
  </LastTwoManuSteps>
') as Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

In the preceding query, the **last()** function in `/AWMI:root//AWMI:Location)[1]/AWMI:step[last()]` returns the number of

manufacturing steps. This value is used to retrieve the last manufacturing step at the work center location.

This is the result:

```
ProductModelID Result
-------------- -----------------------------------
7      <LastTwoManuSteps>
         <Last-1Step>
            When finished, inspect the forms for defects per
            Inspection Specification .
         </Last-1Step>
         <LastStep>Remove the frames from the tool and place them
            in the Completed or Rejected bin as appropriate.
         </LastStep>
       </LastTwoManuSteps>
```

See Also

XQuery Functions Against the XML Data Type

position Function

Returns an integer value that indicates the position of the context item within the sequence of items currently being processed.

Syntax

```
fn:position() as xs:integer
```

Remarks

In SQL Server, **fn:position()** can only be used in the context of a context-dependent predicate. Specifically, it can only be used inside brackets ([ ]).Comparing against this function does not reduce the cardinality during static type inference.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the database.

A. Using the position() XQuery function to retrieve the first two product features

The following query retrieves the first two features, the first two child elements of the `<Features>` element, from the product model catalog description. If there are more features, it adds a `<there-is-more/>` element to the result.

```
SELECT CatalogDescription.query('
```

```
     declare namespace
pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     <Product>
          { /pd:ProductDescription/@ProductModelID }
          { /pd:ProductDescription/@ProductModelName }
          {
            for $f in
/pd:ProductDescription/pd:Features/*[position()<=2]
            return
            $f
          }
          {
            if (count(/pd:ProductDescription/pd:Features/*) > 2)
            then <there-is-more/>
            else ()
          }
     </Product>
') as x
FROM Production.ProductModel
WHERE CatalogDescription is not null
```

Note the following from the previous query:

- The **namespace** keyword in the XQuery Prolog defines a namespace prefix that is used in the query body.
- The query body constructs XML that has a <Product> element with **ProductModelID** and **ProductModelName** attributes, and has product features returned as child elements.
- The **position()** function is used in the predicate to determine the position of the <Features> child element in context. If it is the first or second feature, it is returned.
- The IF statement adds a <there-is-more/> element to the result if there are more than two features in the product catalog.
- Because not all product models have their catalog descriptions stored in the table, the WHERE clause is used to discard rows where CatalogDescriptions is NULL.

This is a partial result:

```
<Product ProductModelID="19" ProductModelName="Mountain 100">
```

```
  <p1:Warranty
xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
    <p1:WarrantyPeriod>3 year</p1:WarrantyPeriod>
    <p1:Description>parts and labor</p1:Description>
  </p1:Warranty>
  <p2:Maintenance
xmlns:p2="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
    <p2:NoOfYears>10</p2:NoOfYears>
    <p2:Description>maintenance contact available through your dealer
or
                    any AdventureWorks retail store.</p2:Description>
  </p2:Maintenance>
  <there-is-more/>
</Product>
...
```

See Also

XQuery Functions Against the XML Data Type

Functions on Sequences

The following topics discuss and provide sample code for the functions on sequences:

In This Section

Category	Function Name
Functions on sequences	empty
	distinct-values
	id

empty Function

Returns True if the value of $arg is an empty sequence. Otherwise, the function returns False.

Syntax

```
fn:empty($arg as item()*) as xs:boolean
```

Arguments

$arg

A sequence of items. If the sequence is empty, the function returns True. Otherwise, the function returns False.

Remarks

The **fn:exists()** function is not supported. As an alternative, the **not()** function can be used.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the empty() XQuery function to determine if an attribute is present

In the manufacturing process for Product Model 7, this query returns all the work center locations that do not have a **MachineHours** attribute.

```
SELECT ProductModelID, Instructions.query('
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
     for $i in /AWMI:root/AWMI:Location[empty(@MachineHours)]
     return
       <Location
            LocationID="{ ($i/@LocationID) }"
            LaborHrs="{ ($i/@LaborHours) }" >
            {
              $i/@MachineHours
            }
       </Location>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

This is the result:

```
ProductModelID      Result
--------------- ----------------------------------------
```

```
7              <Location LocationID="30" LaborHrs="1"/>
               <Location LocationID="50" LaborHrs="3"/>
               <Location LocationID="60" LaborHrs="4"/>
```

The following, slightly modified, query returns "NotFound" if the **MachineHour** attribute is not present:

```
SELECT ProductModelID, Instructions.query('
declare namespace
p14="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
     for $i in /p14:root/p14:Location
     return
       <Location
            LocationID="{ ($i/@LocationID) }"
            LaborHrs="{ ($i/@LaborHours) }" >
            {
                 if (empty($i/@MachineHours)) then
                    attribute MachineHours { "NotFound" }
                 else
                    attribute MachineHours { data($i/@MachineHours) }
            }
       </Location>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

This is the result:

```
ProductModelID Result
-------------- ----------------------------------
7
  <Location LocationID="10" LaborHrs="2.5" MachineHours="3"/>
  <Location LocationID="20" LaborHrs="1.75" MachineHours="2"/>
  <Location LocationID="30" LaborHrs="1" MachineHours="NotFound"/>
  <Location LocationID="45" LaborHrs="0.5" MachineHours="0.65"/>
  <Location LocationID="50" LaborHrs="3" MachineHours="NotFound"/>
  <Location LocationID="60" LaborHrs="4" MachineHours="NotFound"/>
```

See Also

XQuery Functions Against the XML Data Type

exist() method (XML data type)

distinct-values Function

Removes duplicate values from the sequence specified by $arg. If $arg is an empty sequence, the function returns the empty sequence.

Syntax

```
fn:distinct-values($arg as xdt:anyAtomicType*) as xdt:anyAtomicType*
```

Arguments

$arg

Sequence of atomic values.

Remarks

All types of the atomized values that are passed to **distinct-values()** have to be subtypes of the same base type. Base types that are accepted are the types that support the **eq** operation. These types include the three built-in numeric base types, the date/time base types, xs:string, xs:boolean, and xdt:untypedAtomic. Values of type xdt:untypedAtomic are cast to xs:string. If there is a mixture of these types, or if other values of other types are passed, a static error is raised.

The result of **distinct-values()** receives the base type of the passed in types, such as xs:string in the case of xdt:untypedAtomic, with the original cardinality. If the input is statically empty, empty is implied and a static error is raised.

Values of type xs:string are compared to the XQuery default Unicode Codepoint Collation.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the distinct-values() function to remove duplicate values from the sequence

In this example, an XML instance that contains telephone numbers is assigned to an **xml** type variable. The XQuery specified against this variable uses the **distinct-values()** function to compile a list of telephone numbers that do not contain duplicates.

```
declare @x xml
set @x = '<PhoneNumbers>
 <Number>111-111-1111</Number>
 <Number>111-111-1111</Number>
```

```
 <Number>222-222-2222</Number>
</PhoneNumbers>'
-- 1st select
select @x.query('
  distinct-values( data(/PhoneNumbers/Number) )
') as result
```

This is the result:

```
 111-111-1111 222-222-2222
```

In the following query, a sequence of numbers (1, 1, 2) is passed in to the **distinct-values()** function. The function then removes the duplicate in the sequence and returns the other two.

```
declare @x xml
set @x = ''
select @x.query('
  distinct-values((1, 1, 2))
') as result
```

The query returns 1 2.

Implementation Limitations

These are the limitations:

- The **distinct-values()** function maps integer values to xs:decimal.
- The **distinct-values()** function only supports the previously mentioned types and does not support the mixture of base types.
- The **distinct-values()** function on xs:duration values is not supported.
- The syntactic option that provides a collation is not supported.

See Also

XQuery Functions Against the XML Data Type

id Function

Returns the sequence of element nodes with xs:ID values that match the values of one or more of the xs:IDREF values supplied in $arg.

Syntax

```
fn:id($arg as xs:IDREF*) as element()*
```

Arguments

$arg

One or more xs:IDREF values.

Remarks

The result of the function is a sequence of elements in the XML instance, in document order, that has an xs:ID value equal to one or more of the xs:IDREFs in the list of candidate xs:IDREFs.

If the xs:IDREF value does not match any element, the function returns the empty sequence.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the database.

A. Retrieving elements based on the IDREF attribute value

The following example uses fn:id to retrieve the <employee> elements, based on the IDREF manager attribute. In this example, the manager attribute is an IDREF type attribute and the eid attribute is an ID type attribute.

For a specific manager attribute value, the **id()** function finds the <employee> element whose ID type attribute value matches the input IDREF value. In other words, for a specific employee, the **id()** function returns employee manager.

This is what happens in the example:

- An XML schema collection is created.
- A typed **xml** variable is created by using the XML schema collection.
- The query retrieves the element that has an ID attribute value referenced by the **manager** IDREF attribute of the <employee> element.

```
-- If exists, drop the XML schema collection (SC).
-- drop xml schema collection SC
-- go

create xml schema collection SC as
'<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:e="emp"
targetNamespace="emp">
            <element name="employees" type="e:EmployeesType"/>
            <complexType name="EmployeesType">
                 <sequence>
                      <element name="employee" type="e:EmployeeType"
minOccurs="0" maxOccurs="unbounded" />
                 </sequence>
```

```
            </complexType>

            <complexType name="EmployeeType">
                        <attribute name="eid" type="ID" />
                        <attribute name="name" type="string" />
                        <attribute name="manager" type="IDREF" />
            </complexType>
</schema>'
go
declare @x xml(SC)
set @x='<e:employees xmlns:e="emp">
<employee eid="e1" name="Joe" manager="e10" />
<employee eid="e2" name="Bob" manager="e10" />
<employee eid="e10" name="Dave" manager="e10" />
</e:employees>'

select @x.value(' declare namespace e="emp";
 (fn:id(e:employees/employee[@name="Joe"]/@manager)/@name)[1]',
'varchar(50)')
Go
```

The query returns "Dave" as the value. This indicates that Dave is Joe's manager.

B. Retrieving elements based on the OrderList IDREFS attribute value

In the following example, the OrderList attribute of the `<Customer>` element is an IDREFS type attribute. It lists the order ids for that specific customer. For each order id, there is an `<Order>` element child under the `<Customer>` providing the order value.

The query expression, `data(CustOrders:Customers/Customer[1]/@OrderList)[1]`, retrieves the first value from the IDRES list for the first customer. This value is then passed to the **id()** function. The function then finds the `<Order>` element whose OrderID attribute value matches the input to the **id()** function.

```
drop xml schema collection SC
go
create xml schema collection SC as
'<schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:Customers="Customers" targetNamespace="Customers">
            <element name="Customers" type="Customers:CustomersType"/>
```

```
            <complexType name="CustomersType">
                        <sequence>
                            <element name="Customer"
type="Customers:CustomerType" minOccurs="0" maxOccurs="unbounded" />
                        </sequence>
            </complexType>
             <complexType name="OrderType">
                <sequence minOccurs="0" maxOccurs="unbounded">
                            <choice>
                                <element name="OrderValue"
type="integer" minOccurs="0" maxOccurs="unbounded"/>
                            </choice>
                </sequence>
                <attribute name="OrderID" type="ID" />
            </complexType>

            <complexType name="CustomerType">
                <sequence minOccurs="0" maxOccurs="unbounded">
                            <choice>
                                <element name="spouse" type="string"
minOccurs="0" maxOccurs="unbounded"/>
                                <element name="Order"
type="Customers:OrderType" minOccurs="0" maxOccurs="unbounded"/>
                            </choice>
                </sequence>
                <attribute name="CustomerID" type="string" />
                <attribute name="OrderList" type="IDREFS" />
            </complexType>
 </schema>'
go
declare @x xml(SC)
set @x='<CustOrders:Customers xmlns:CustOrders="Customers">
                <Customer CustomerID="C1" OrderList="OrderA OrderB"  >
                            <spouse>Jenny</spouse>
```

```
                                <Order
OrderID="OrderA"><OrderValue>11</OrderValue></Order>
                                <Order
OrderID="OrderB"><OrderValue>22</OrderValue></Order>

                </Customer>
                <Customer CustomerID="C2" OrderList="OrderC OrderD" >
                                <spouse>John</spouse>
                                <Order
OrderID="OrderC"><OrderValue>33</OrderValue></Order>
                                <Order
OrderID="OrderD"><OrderValue>44</OrderValue></Order>

                        </Customer>
                <Customer CustomerID="C3"  OrderList="OrderE OrderF" >
                                <spouse>Jane</spouse>
                                <Order
OrderID="OrderE"><OrderValue>55</OrderValue></Order>
                                <Order
OrderID="OrderF"><OrderValue>55</OrderValue></Order>
                </Customer>
                <Customer CustomerID="C4"  OrderList="OrderG"  >
                                <spouse>Tim</spouse>
                                <Order
OrderID="OrderG"><OrderValue>66</OrderValue></Order>
                        </Customer>
                <Customer CustomerID="C5"  >
                </Customer>
                <Customer CustomerID="C6" >
                </Customer>
                <Customer CustomerID="C7"  >
                </Customer>
</CustOrders:Customers>'
select @x.query('declare namespace CustOrders="Customers";
  id(data(CustOrders:Customers/Customer[1]/@OrderList)[1])')
```

```
-- result
<Order OrderID="OrderA">
  <OrderValue>11</OrderValue>
</Order>
```

Implementation Limitations

These are the limitations:

- SQL Server does not support the two-argument version of **id()**.
- SQL Server requires the argument type of **id()** to be a subtype of xs:IDREF*.

See Also

Functions on Sequences

Aggregate Functions

The following topics discuss and provide sample code for the aggregate functions.

In This Section

Category	Function Name
Aggregate functions	count
	avg
	min
	max
	sum

count Function

Returns the number of items that are contained in the sequence specified by $arg.

Syntax

```
fn:count($arg as item()*) as xs:integer
```

Arguments

$arg

Items to count.

Remarks

Returns 0 if $arg is an empty sequence.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the count() XQuery function to count the number of work center locations in the manufacturing of a product model

The following query counts the number of work center locations in the manufacturing process of a product model (ProductModelID=7).

```
SELECT Production.ProductModel.ProductModelID,
       Production.ProductModel.Name,
       Instructions.query('
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
       <NoOfWorkStations>
          { count(/AWMI:root/AWMI:Location) }
       </NoOfWorkStations>
') as WorkCtrCount
FROM Production.ProductModel
WHERE Production.ProductModel.ProductModelID=7
```

Note the following from the previous query:

- The **namespace** keyword in XQuery Prolog defines a namespace prefix. The prefix is then used in the XQuery body.
- The query constructs XML that includes the <NoOfWorkStations> element.
- The **count()** function in the XQuery body counts the number of <Location> elements.

This is the result:

```
ProductModelID   Name                 WorkCtrCount
-------------- ---------------------------------------------------
7              HL Touring Frame  <NoOfWorkStations>6</NoOfWorkStations>
```

You can also construct the XML to include the product model ID and name, as shown in the following query:

```
SELECT Instructions.query('
```

```
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
       <NoOfWorkStations
             ProductModelID= "{
sql:column("Production.ProductModel.ProductModelID") }"
             ProductModelName = "{
sql:column("Production.ProductModel.Name") }" >
          { count(/AWMI:root/AWMI:Location) }
       </NoOfWorkStations>
') as WorkCtrCount
FROM Production.ProductModel
WHERE Production.ProductModel.ProductModelID= 7
```

This is the result:

```
<NoOfWorkStations ProductModelID="7"
                  ProductModelName="HL Touring
Frame">6</NoOfWorkStations>
```

Instead of XML, you may return these values as non-xml type, as shown in the following query. The query uses the value() method (xml data type) to retrieve the work center location count.

```
SELECT  ProductModelID,
        Name,
        Instructions.value('declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
           count(/AWMI:root/AWMI:Location)', 'int' ) as WorkCtrCount
FROM Production.ProductModel
WHERE ProductModelID=7
```

This is the result:

```
ProductModelID    Name            WorkCtrCount
------------- ---------------------------------
7             HL Touring Frame        6
```

See Also

XQuery Functions Against the XML Data Type

min Function

Returns from a sequence of atomic values, $arg, the one item whose value is less than that of all the others.

Syntax

```
fn:min($arg as xdt:anyAtomicType*) as xdt:anyAtomicType?
```

Arguments

$arg

Sequence of items from which to return the minimum value.

Remarks

All types of the atomized values that are passed to **min()** have to be subtypes of the same base type. Base types that are accepted are the types that support the **gt** operation. These types include the three built-in numeric base types, the date/time base types, xs:string, xs:boolean, and xdt:untypedAtomic. Values of type xdt:untypedAtomic are cast to xs:double. If there is a mixture of these types, or if other values of other types are passed, a static error is raised.

The result of **min()** receives the base type of the passed in types, such as xs:double in the case of xdt:untypedAtomic. If the input is statically empty, empty is implied and a static error is returned.

The **min()** function returns the one value in the sequence that is smaller than any other in the input sequence. For xs:string values, the default Unicode Codepoint Collation is being used. If an xdt:untypedAtomic value cannot be cast to xs:double, the value is ignored in the input sequence, $arg. If the input is a dynamically calculated empty sequence, the empty sequence is returned.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the min() XQuery function to find the work center location that has the fewest labor hours

The following query retrieves all work center locations in the manufacturing process of the product model (ProductModelID=7) that have the fewest labor hours. Generally, as shown in the following, a single location is returned. If multiple locations had an equal number of minimum labor hours, they would all be returned.

```
select ProductModelID, Name, Instructions.query('
  declare namespace AWMI=
    "http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
  for   $Location in /AWMI:root/AWMI:Location
```

```
  where $Location/@LaborHours =
          min( /AWMI:root/AWMI:Location/@LaborHours )
return
  <Location WCID=     "{ $Location/@LocationID }"
              LaborHrs= "{ $Location/@LaborHours }" />
  ') as Result
FROM  Production.ProductModel
WHERE ProductModelID=7
```

Note the following from the previous query:

- The **namespace** keyword in the XQuery prolog defines a namespace prefix. This prefix is then used in the XQuery body.

The XQuery body constructs the XML that has a <Location> element with WCID and **LaborHrs** attributes.

- The query also retrieves the ProductModelID and name values.

This is the result:

```
ProductModelID   Name              Result
--------------   ---------------   --------------------------------
7                HL Touring Frame  <Location WCID="45" LaborHrs="0.5"/>
```

Implementation Limitations

These are the limitations:

- The **min()** function maps all integers to xs:decimal.
- The **min()** function on values of type xs:duration is not supported.
- Sequences that mix types across base type boundaries are not supported.
- The syntactic option that provides a collation is not supported.

See Also

XQuery Functions Against the xml Data Type

max Function

Returns from a sequence of atomic values, $arg, the one item whose value is greater than that of all the others.

Syntax

```
fn:max($arg as xdt:anyAtomicType*) as xdt:anyAtomicType?
```

Arguments

$arg

Sequence of atomic values from which to return the maximum value.

Remarks

All types of the atomized values that are passed to **max()** have to be subtypes of the same base type. Base types that are accepted are the types that support the **gt** operation. These types include the three built-in numeric base types, the date/time base types, xs:string, xs:boolean, and xdt:untypedAtomic. Values of type xdt:untypedAtomic are cast to xs:double. If there is a mixture of these types, or if other values of other types are passed, a static error is raised.

The result of **max()** receives the base type of the passed in types, such as xs:double in the case of xdt:untypedAtomic. If the input is statically empty, empty is implied and a static error is raised.

The **max()** function returns the one value in the sequence that is greater than any other in the input sequence. For xs:string values, the default Unicode Codepoint Collation is being used. If an xdt:untypedAtomic value cannot be cast to xs:double, the value is ignored in the input sequence, $arg. If the input is a dynamically calculated empty sequence, the empty sequence is returned.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the database.

A. Using the max() XQuery function to find work center locations in the manufacturing process that have the most labor hours

The query provided in min function (XQuery) can be rewritten to use the **max()** function.

Implementation Limitations

These are the limitations:

- The **max(**) function maps all integers to xs:decimal.
- The **max()** function on values of type xs:duration is not supported.
- Sequences that mix types across base type boundaries are not supported.
- The syntactic option that provides a collation is not supported.

See Also

XQuery Functions Against the XML Data Type

avg Function

Returns the average of a sequence of numbers.

Syntax

```
fn:avg($arg as xdt:anyAtomicType*) as xdt:anyAtomicType?
```

Arguments

$arg

The sequence of atomic values whose average is computed.

Remarks

All the types of the atomized values that are passed to **avg()** have to be a subtype of exactly one of the three built-in numeric base types or xdt:untypedAtomic. They cannot be a mixture. Values of type xdt:untypedAtomic are treated as xs:double. The result of **avg()** receives the base type of the passed in types, such as xs:double in the case of xdt:untypedAtomic.

If the input is statically empty empty is implied and a static error is raised.

The **avg()** function returns the average of the numbers computed. For example:

```
sum($arg) div count($arg)
```

If $arg is an empty sequence, the empty sequence is returned.

If an xdt:untypedAtomic value cannot be cast to xs:double, the value is disregarded in the input sequence, $arg.

In all other cases, the function returns a static error.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the avg() XQuery function to find work center locations in the manufacturing process in which labor hours are greater than the average for all work center locations.

You can rewrite the query provided in min function (XQuery) to use the **avg()** function.

Implementation Limitations

These are the limitations:

- The **avg()** function maps all integers to xs:decimal.
- The **avg()** function on values of type xs:duration is not supported.
- Sequences that mix types across base type boundaries are not supported.

See Also

XQuery Functions Against the XML Data Type

sum Function

Returns the sum of a sequence of numbers.

Syntax

```
fn:sum($arg as xdt:anyAtomicType*) as xdt:anyAtomicType
```

Arguments

$arg

Sequence of atomic values whose sum is to be computed.

Remarks

All types of the atomized values that are passed to **sum()** have to be subtypes of the same base type. Base types that are accepted are the three built-in numeric base types or xdt:untypedAtomic. Values of type xdt:untypedAtomic are cast to xs:double. If there is a mixture of these types, or if other values of other types are passed, a static error is raised.

The result of **sum()** receives the base type of the passed in types such as xs:double in the case of xdt:untypedAtomic, even if the input is optionally the empty sequence. If the input is statically empty, the result is 0 with the static and dynamic type of xs:integer.

The **sum()** function returns the sum of the numeric values. If an xdt:untypedAtomic value cannot be cast to xs:double, the value is ignored in the input sequence, $arg. If the input is a dynamically calculated empty sequence, the value 0 of the used base type is returned.

The function returns a runtime error when an overflow or out of range exception occurs.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the database.

A. Using the sum() XQuery function to find the total combined number of labor hours for all work center locations in the manufacturing process

The following query finds the total labor hours for all work center locations in the manufacturing process of all product models for which manufacturing instructions are stored.

```
SELECT Instructions.query('
   declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
  <ProductModel PMID= "{
sql:column("Production.ProductModel.ProductModelID") }"
  ProductModelName = "{ sql:column("Production.ProductModel.Name") }" >
   <TotalLaborHrs>
     { sum(//AWMI:Location/@LaborHours) }
   </TotalLaborHrs>
 </ProductModel>
    ') as Result
FROM Production.ProductModel
```

```
WHERE Instructions is not NULL
```

This is the partial result.

```
<ProductModel PMID="7" ProductModelName="HL Touring Frame">
   <TotalLaborHrs>12.75</TotalLaborHrs>
</ProductModel>
<ProductModel PMID="10" ProductModelName="LL Touring Frame">
  <TotalLaborHrs>13</TotalLaborHrs>
</ProductModel>
...
```

Instead of returning the result as XML, you can write the query to generate relational results, as shown in the following query:

```
SELECT ProductModelID,
        Name,
        Instructions.value('declare namespace
      AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
    sum(//AWMI:Location/@LaborHours)', 'float') as TotalLaborHours
FROM Production.ProductModel
WHERE Instructions is not NULL
```

This is a partial result:

```
ProductModelID Name                 TotalLaborHours
-------------- -------------------------------------------------
7              HL Touring Frame           12.75
10             LL Touring Frame           13
43             Touring Rear Wheel         3
...
```

Implementation Limitations

These are the limitations:

- Only the single argument version of **sum()** is supported.
- If the input is a dynamically calculated empty sequence, the value 0 of the used base type is returned instead of type xs:integer.
- The **sum()** function maps all integers to xs:decimal.
- The **sum()** function on values of type xs:duration is not supported.
- Sequences that mix types across base type boundaries are not supported.
- The sum((xs:double("INF"), xs:double("-INF"))) raises a domain error.

See Also

XQuery Functions Against the XML Data Type

Data Accessor Functions

The topics in this section discuss and provide sample code for the data-accessor functions.

Understanding fn:data(), fn:string(), and text()

XQuery has a function **fn:data()** to extract scalar, typed values from nodes, a node test **text()** to return text nodes, and the function **fn:string()** that returns the string value of a node. Their use can be confusing. The following are guidelines for using them correctly in SQL Server. The XML instance <age>12</age> is used for the purpose of illustration.

- Untyped XML: The path expression /age/text() returns the text node "12". The function fn:data(/age) returns the string value "12" and so does fn:string(/age).
- Typed XML: The expression /age/text() returns a static error for any simple typed <age> element. On the other hand, fn:data(/age) returns integer 12. The fn:string(/age) yields the string "12".

In This Section

- string Function
- data Function

See Also

Path Expressions (XQuery)

string Function

Returns the value of $arg represented as a string.

Syntax

```
fn:string() as xs:string
fn:string($arg as item()?) as xs:string
```

Arguments

$arg

Is a node or an atomic value.

Remarks

- If $arg is the empty sequence, the zero-length string is returned.

- If $arg is a node, the function returns the string value of the node that is obtained by using the string-value accessor. This is defined in the W3C XQuery 1.0 and XPath 2.0 Data Model specification.
- If $arg is an atomic value, the function returns the same string that is returned by the expression cast as **xs:string**, $arg, except when noted otherwise.
- If the type of $arg is **xs:anyURI**, the URI is converted to a string without escaping special characters.
- Inthis implementation, **fn:string()** without an argument can only be used in the context of a context-dependent predicate. Specifically, it can only be used inside brackets ([ ]).

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the string function

The following query retrieves the <Features> child element node of the <ProductDescription> element.

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
 /PD:ProductDescription/PD:Features
')
FROM Production.ProductModel
WHERE ProductModelID=19
```

This is the partial result:

```
<PD:Features
xmlns:PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription">
   These are the product highlights.
   <p1:Warranty
xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
    <p1:WarrantyPeriod>3 years</p1:WarrantyPeriod>
    <p1:Description>parts and labor</p1:Description>
   </p1:Warranty>
       ...
</PD:Features>
```

If you specify the **string()** function, you receive the string value of the specified node.

```
SELECT CatalogDescription.query('
declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
 string(/PD:ProductDescription[1]/PD:Features[1])
')
FROM Production.ProductModel
WHERE ProductModelID=19
```

This is the partial result.

```
These are the product highlights.
3 yearsparts and labor...
```

B. Using the string function on various nodes

In the following example, an XML instance is assigned to an xml type variable. Queries are specified to illustrate the result of applying **string()** to various nodes.

```
declare @x xml
set @x = '<?xml version="1.0" encoding="UTF-8" ?>
<!--  This is a comment -->
<root>
  <a>10</a>
just text
  <b attr="x">20</b>
</root>
'
```

The following query retrieves the string value of the document node. This value is formed by concatenating the string value of all its descendent text nodes.

```
select @x.query('string(/)')
```

This is the result:

```
This is a comment 10
just text
 20
```

The following query tries to retrieve the string value of a processing instruction node. The result is an empty sequence, because it does not contain a text node.

```
select @x.query('string(/processing-instruction()[1])')
```

The following query retrieves the string value of the comment node and returns the text node.

```
select @x.query('string(/comment()[1])')
```

This is the result:

```
This is a comment
```

See Also

XQuery Functions Against the XML Data Type

data Function

Returns the typed value for each item specified by $arg.

Syntax

```
fn:data ($arg as item()*) as xdt:untypedAtomic*
```

Arguments

$arg

Sequence of items whose typed values will be returned.

Remarks

The following applies to typed values:

- The typed value of an atomic value is the atomic value.
- The typed value of a text node is the string value of the text node.
- The typed value of a comment is the string value of the comment.
- The typed value of a processing instruction is the content of the processing-instruction, without the processing instruction target name.
- The typed value of a document node is its string value.

The following applies to attribute and element nodes:

- If an attribute node is typed with an XML schema type, its typed value is the typed value, accordingly.
- If the attribute node is untyped, its typed value is equal to its string value that is returned as an instance of **xdt:untypedAtomic**.
- If the element node has not been typed, its typed value is equal to its string value that is returned as an instance of **xdt:untypedAtomic**.

The following applies to typed element nodes:

- If the element has a simple content type, **data()** returns the typed value of the element.
- If the node is of complex type, including xs:anyType, **data()** returns a static error.

Although using the **data()** function is frequently optional, as shown in the following examples, specifying the **data()** function explicitly increases query readability. For more information, see XQuery Functions Against the xml Data Type.

You cannot specify **data()** on constructed XML, as shown in the following:

```
declare @x xml
set @x = ''
select @x.query('data(<SomeNode>value</SomeNode>)')
```

Examples

This topic provides XQuery examples against XML instances stored in various **xml** type columns in the AdventureWorks database.

A. Using the data() XQuery function to extract typed value of a node

The following query illustrates how the **data()** function is used to retrieve values of an attribute, an element, and a text node:

```
WITH XMLNAMESPACES (
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS p1,
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain' AS wm)

SELECT CatalogDescription.query(N'
 for $pd in //p1:ProductDescription
 return
    <Root
      ProductID = "{ data( ($pd//@ProductModelID)[1] ) }"
      Feature =   "{ data(
($pd/p1:Features/wm:Warranty/wm:Description)[1] ) }" >
    </Root>
 ') as Result
FROM Production.ProductModel
WHERE ProductModelID = 19
```

This is the result:

```
<Root ProductID="19" Feature="parts and labor"/>
```

As mentioned, the **data()** function is optional when you are constructing attributes. If you do not specify the **data()** function, it is implicitly assumed. The following query produces the same results as the previous query:

```
WITH XMLNAMESPACES (
```

```
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS p1,
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain' AS wm)

SELECT CatalogDescription.query('
      for $pd in //p1:ProductDescription
         return
          <Root
                ProductID = "{ ($pd/@ProductModelID)[1] }"
                Feature =   "{
($pd/p1:Features/wm:Warranty/wm:Description)[1] }" >
           </Root>
 ') as Result
FROM Production.ProductModel
WHERE ProductModelID = 19
```

The following examples illustrate instances in which the **data()** function is required.

In the following query, **$pd/p1:Specifications/Material** returns the `<Material>` element. Also, **data($pd/p1:Specifications/ Material)** returns character data typed as xdt:untypedAtomic, because `<Material>` is untyped. When the input is untyped, the result of **data()** is typed as **xdt:untypedAtomic**.

```
SELECT CatalogDescription.query('
declare namespace
p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
      for $pd in //p1:ProductDescription
         return
          <Root>
             { $pd/p1:Specifications/Material }
             { data($pd/p1:Specifications/Material) }
           </Root>
 ') as Result
FROM Production.ProductModel
WHERE ProductModelID = 19
```

This is the result:

```
<Root>
  <Material>Almuminum Alloy</Material>Almuminum Alloy
</Root>
```

In the following query, **data($pd/p1:Features/wm:Warranty)** returns a static error, because `<Warranty>` is a complex type element.

```
WITH XMLNAMESPACES (
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelDescription' AS p1,
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelWarrAndMain' AS wm)

SELECT CatalogDescription.query('
 <Root>
   {     /p1:ProductDescription/p1:Features/wm:Warranty }
   { data(/p1:ProductDescription/p1:Features/wm:Warranty) }
 </Root>
 ') as Result
FROM  Production.ProductModel
WHERE ProductModelID = 23
```

See Also

XQuery Functions Against the XML Data Type

Constructor Functions

From a specified input, the constructor functions create instances of any of the XSD built-in or user-defined atomic types.

Syntax

```
TYP($atomicvalue as xdt:anyAtomicType?

) as TYP?
```

Arguments

$strval

String that will be converted.

TYP

Any built-in XSD type.

Remarks

Constructors are supported for base and derived atomic XSD types. However, the subtypes of **xs:duration**, which includes **xdt:yearMonthDuration and xdt:dayTimeDuration**, and **xs:QName**, **xs:NMTOKEN**, and **xs:NOTATION** are not supported. User-defined atomic types that are available in the associated schema collections are also available, provided they are directly or indirectly derived from the following types.

Supported Base Types

These are the supported base types:

- xs:string
- xs:boolean
- xs:decimal
- xs:float
- xs:double
- xs:duration
- xs:dateTime
- xs:time
- xs:date
- xs:gYearMonth
- xs:gYear
- xs:gMonthDay
- xs:gDay
- xs:gMonth
- xs:hexBinary
- xs:base64Binary
- xs:anyURI

Supported Derived Types

These are the supported derived types:

- xs:normalizedString
- xs:token
- xs:language

- xs:Name
- xs:NCName
- xs:ID
- xs:IDREF
- xs:ENTITY
- xs:integer
- xs:nonPositiveInteger
- xs:negativeInteger
- xs:long
- xs:int
- xs:short
- xs:byte
- xs:nonNegativeInteger
- xs:unsignedLong
- xs:unsignedInt
- xs:unsignedShort
- xs:unsignedByte
- xs:positiveInteger

SQL Server also supports constant folding for construction function invocations in the following ways:

- If the argument is a string literal, the expression will be evaluated during compilation. When the value does not satisfy the type constraints, a static error is raised.
- If the argument is a literal of another type, the expression will be evaluated during compilation. When the value does not satisfy the type constraints, the empty sequence is returned.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the dateTime() XQuery function to retrieve older product descriptions

In this example, a sample XML document is first assigned to an **xml** type variable. This document contains three sample `<ProductDescription>` elements, with each one that contain a `<DateCreated>` child element.

The variable is then queried to retrieve only those product descriptions that were created before a specific date. For purposes of comparison, the query uses the **xs:dateTime()** constructor function to type the dates.

```
declare @x xml
set @x = '<root>
<ProductDescription ProductID="1" >
  <DateCreated DateValue="2000-01-01T00:00:00Z" />
  <Summary>Some Summary description</Summary>
</ProductDescription>
<ProductDescription  ProductID="2" >
  <DateCreated DateValue="2001-01-01T00:00:00Z" />
  <Summary>Some Summary description</Summary>
</ProductDescription>
<ProductDescription ProductID="3" >
  <DateCreated DateValue="2002-01-01T00:00:00Z" />
  <Summary>Some Summary description</Summary>
</ProductDescription>
</root>'

select @x.query('
     for $PD in  /root/ProductDescription
     where xs:dateTime(data( ($PD/DateCreated/@DateValue)[1] )) <
xs:dateTime("2001-01-01T00:00:00Z")
     return
        element Product
       {
        ( attribute ProductID { data($PD/@ProductID ) },
        attribute DateCreated { data( ($PD/DateCreated/@DateValue)[1] )
} )
        }
 ')
```

Note the following from the previous query:

- The FOR ... WHERE loop structure is used to retrieve the <ProductDescription> element satisfying the condition specified in the WHERE clause.
- The **dateTime()** constructor function is used to construct **dateTime** type values so they can be compared appropriately.
- The query then constructs the resulting XML. Because you are constructing a sequence of attributes, commas and parentheses are used in the XML construction.

This is the result:

```
<Product
   ProductID="1"
   DateCreated="2000-01-01T00:00:00Z"/>
```

See Also

XML Construction (XQuery)

XQuery Functions Against the XML Data Type

Boolean Constructor Functions

The following topics discuss and provide sample code for the Boolean constructor functions.

In This Section

Category	Function Name
Boolean constructor functions	true
	false

See Also

XQuery Functions Against the xml Data Type

true Function

Returns the xs:boolean value True. This is equivalent to `xs:boolean("1")`.

Syntax

```
fn:true() as xs:boolean
```

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the true() XQuery Boolean function

The following example queries an untyped **xml** variable. The expression in the **value()** method returns Boolean **true()** if "aaa" is the attribute value. The **value()** method of the **xml** data type converts the Boolean value into a bit and returns it.

```
DECLARE @x XML
SET @x= '<ROOT><elem attr="aaa">bbb</elem></ROOT>'
```

```
select @x.value(' if ( (/ROOT/elem/@attr)[1] eq "aaa" ) then fn:true()
else fn:false() ', 'bit')
go
-- result = 1
```

In the following example, the query is specified against a typed **xml** column. The `if` expression checks the typed Boolean value of the `<ROOT>` element and returns the constructed XML, accordingly. The example performs the following:

- Creates an XML schema collection that defines the `<ROOT>` element of the xs:boolean type.
- Creates a table with a typed **xml** column by using the XML schema collection.
- Saves an XML instance in the column and queries it.

```
-- Drop table if exist
--DROP TABLE T
--go
DROP XML SCHEMA COLLECTION SC
go
CREATE XML SCHEMA COLLECTION SC AS '
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="QNameXSD" >
      <element name="ROOT" type="boolean" nillable="true"/>
</schema>'
go
CREATE TABLE T (xmlCol XML(SC))
go
-- following OK
insert into T values ('<ROOT xmlns="QNameXSD">true</ROOT>')
 go
-- Retrieve the local name.
SELECT xmlCol.query('declare namespace a="QNameXSD";
   if (/a:ROOT[1] eq true()) then
       <result>Found boolean true</result>
   else
       <result>Found boolean false</result>')
```

```
FROM T
-- result = <result>Found boolean true</result>
-- Clean up
DROP TABLE T
go
DROP XML SCHEMA COLLECTION SC
go
```

See Also

Boolean Constructor Functions (XQuery)

false Function

Returns the xs:boolean value False. This is equivalent to `xs:boolean("0")`.

Syntax

```
fn:false() as xs:boolean
```

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Using the false() XQuery Boolean function

For a working sample, see true Function (XQuery).

See Also

Boolean Constructor Functions (XQuery)

Functions Related to QNames

The following topics discuss and provide sample code for the data-accessor functions.

In This Section

Category	Function Name
Functions related to QNames	expanded-QName
	local-name-from-QName
	namespace-uri-from-QName

See Also

XQuery Functions Against the xml Data Type

expanded-QName

Returns a value of the xs:QName type with the namespace URI specified in the *$paramURI* and the local name specified in the *$paramLocal*. If *$paramURI* is the empty string or the empty sequence, it represents no namespace.

Syntax

```
fn:expanded-QName($paramURI as xs:string?, $paramLocal as xs:string?)
as xs:QName?
```

Arguments

$paramURI

Is the namespace URI for the QName.

$paramLocal

Is the local name part of the QName.

Remarks

The following applies to the **expanded-QName()** function:

- If the *$paramLocal* value specified is not in the correct lexical form for xs:NCName type, the empty sequence is returned and represents a dynamic error.
- Conversion from xs:QName type to any other type is not supported in SQL Server. Because of this, the **expanded-QName()** function cannot be used in XML construction. For example, when you are constructing a node, such as `<e> expanded-QName(...) </e>`, the value has to be untyped. This would require that you convert the xs:QName type value returned by `expanded-QName()` to xdt:untypedAtomic. However, this is not supported. A solution is provided in an example later in this topic.
- You can modify or compare the existing QName type values. For example, `/root[1]/e[1] eq expanded-QName("http://nsURI" "myNS")` compares the value of the element, `<e>`, with the QName returned by the **expanded-QName()** function.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the database.

A. Replacing a QName type node value

This example illustrates how you can modify the value of an element node of QName type. The example performs the following:

- Creates an XML schema collection that defines an element of QName type.
- Creates a table with an **xml** type column by using the XML schema collection.
- Saves an XML instance in the table.

- Uses the **modify()** method of the xml data type to modify the value of the QName type element in the instance. The **expanded-QName()** function is used to generate the new QName type value.

```
-- If XML schema collection (if exists)
-- drop xml schema collection SC
-- go
-- Create XML schema collection
CREATE XML SCHEMA COLLECTION SC AS N'
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="QNameXSD"
      xmlns:xqo="QNameXSD" elementFormDefault="qualified">
      <element name="Root" type="xqo:rootType" />
      <complexType name="rootType">
            <sequence minOccurs="1" maxOccurs="1">
                        <element name="ElemQN" type="xs:QName" />
            </sequence>
      </complexType>
</schema>'
go
-- Create table.
CREATE TABLE T( XmlCol xml(SC) )
-- Insert sample XML instnace
INSERT INTO T VALUES ('
<Root xmlns="QNameXSD" xmlns:ns="http://myURI">
      <ElemQN>ns:someName</ElemQN>
</Root>')
go
-- Verify the insertion
SELECT * from T
go
-- Result
<Root xmlns="QNameXSD" xmlns:ns="http://myURI">
  <ElemQN>ns:someName</ElemQN>
```

```
</Root>
```

In the following query, the <ElemQN> element value is replaced by using the **modify()** method of the xml data type and the replace value of XML DML, as shown.

```
-- the value.
UPDATE T
SET XmlCol.modify('
  declare default element namespace "QNameXSD";
  replace value of /Root[1]/ElemQN
  with expanded-QName("http://myURI", "myLocalName") ')
go
-- Verify the result
SELECT * from T
go
```

This is the result. Note that the element <ElemQN> of QName type now has a new value:

```
<Root xmlns="QNameXSD" xmlns:ns="urn">
  <ElemQN xmlns:p1="http://myURI">p1:myLocalName</ElemQN>
</Root>
```

The following statements remove the objects used in the example.

```
-- Cleanup
DROP TABLE T
go
drop xml schema collection SC
go
```

B. Dealing with the limitations when using the expanded-QName() function

The **expanded-QName** function cannot be used in XML construction. The following example illustrates this. To work around this limitation, the example first inserts a node and then modifies the node.

```
-- if exists drop the table T
--drop table T
-- go
-- Create XML schema collection
-- DROP XML SCHEMA COLLECTION SC
-- go
CREATE XML SCHEMA COLLECTION SC AS '
```

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="root" type="QName" nillable="true"/>
</schema>'
go
 -- Create table T with a typed xml column (using the XML schema
collection)
CREATE TABLE T (xmlCol XML(SC))
go
-- Insert an XML instance.
insert into T values ('<root xmlns:a="http://someURI">a:b</root>')
 go
-- Verify
SELECT *
FROM T
```

The following attempt adds another <root> element but fails, because the expanded-QName() function is not supported in XML construction.

```
update T SET xmlCol.modify('
insert <root>{expanded-QName("http://ns","someLocalName")}</root> as
last into / ')
go
```

A solution to this is to first insert an instance with a value for the <root> element and then modify it. In this example, a nil initial value is used when the <root> element is inserted. The XML schema collection in this example allows a nil value for the <root> element.

```
update T SET xmlCol.modify('
insert <root xsi:nil="true"/> as last into / ')
go
-- now replace the nil value with another QName.
update T SET xmlCol.modify('
replace value of /root[last()] with expanded-
QName("http://ns","someLocalName") ')
go
 -- verify
SELECT * FROM T
go
```

```
-- result
<root>b</root>
<root xmlns:a="http://someURI">a:b</root>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p1="http://ns">p1:someLocalName</root>
```

You can compare the QName value, as shown in the following query. The query returns only the <root> elements whose values match the QName type value returned by the **expanded-QName()** function.

```
SELECT xmlCol.query('
    for $i in /root
    return
       if ($i eq expanded-QName("http://ns","someLocalName") ) then
          $i
       else
          ()')
FROM T
```

Implementation Limitations

There is one limitation: The **expanded-QName()** function accepts the empty sequence as the second argument and will return empty instead of raising a run-time error when the second argument is incorrect.

See Also

Functions Related to QNames (XQuery)

local-name-from-QName

Returns an xs:NCNAME that represents the local part of QName specified by $arg. The result is an empty sequence if $arg is the empty sequence.

Syntax

```
fn:local-name-from-QName($arg as xs:QName?) as xs:NCName?
```

Arguments

$arg

Is the QName that the local name should be extracted from.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the database.

The following example uses the **local-name-from-QName()** function to retrieve the local name and namespace URI parts from a QName type value. The example performs the following:

- Creates an XML schema collection.
- Creates a table with an xml type column. The xml type is typed using the XML schema collection.
- Stores a sample XML instance in the table. Using the **query()** method of the xml data type, the query expression is executed to retrieve the local name part of the QName type value from the instance.

```
DROP TABLE T
go
DROP XML SCHEMA COLLECTION SC
go
CREATE XML SCHEMA COLLECTION SC AS '
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="QNameXSD" >
      <element name="root" type="QName" nillable="true"/>
</schema>'
go

CREATE TABLE T (xmlCol XML(SC))
go
-- following OK
insert into T values ('<root xmlns="QNameXSD"
xmlns:a="http://someURI">a:someLocalName</root>')
 go
-- Retrieve the local name.
SELECT xmlCol.query('declare default element namespace "QNameXSD";
local-name-from-QName(/root[1])')
FROM T
-- Result = someLocalName
-- You can retrive namespace URI part from the QName using the
namespace-uri-from-QName() function
SELECT xmlCol.query('declare default element namespace "QNameXSD";
namespace-uri-from-QName(/root[1])')
```

```
FROM T
-- Result = http://someURI
```

See Also

Functions Related to QNames (XQuery)

namespace-uri-from-QName

Returns a string representing the namespace uri of the QName specified by $arg. The result is the empty sequence if $arg is the empty sequence.

Syntax

```
namespace-uri-from-QName($arg as xs:QName?) as xs:string?
```

Arguments

$arg

Is the QName whose namespace URI is returned.

Examples

This topic provides XQuery examples against XML instances that are stored in various **xml** type columns in the AdventureWorks database.

A. Retrieve the namespace URI from a QName

For a working sample, see local-name-from-QName (XQuery).

Implementation Limitations

These are the limitations:

- The **namespace-uri-from-QName()** function returns instances of xs:string instead of xs:anyURI.

See Also

Functions Related to QNames (XQuery)

SQL Server XQuery Extension Functions

As described in the topic, Binding Relational Data Inside XML, you use the **sql:column()** and **sql:variable()** XQuery extension functions to bind relational data inside XML. The following topics discuss these functions.

In This Section

Category	Function Name
XQuery extension functions	sql:column()
	sql:variable()

See Also

XQuery Functions Against the xml Data Type

sql:column() Function

As described in the topic, Binding Relational Data Inside XML, you can use the **sql:column(()** function when you use XML Data Type Methods to expose a relational value inside XQuery.

For example, the query() method (XML data type) is used to specify a query against an XML instance that is stored in a variable or column of **xml** type. Sometimes, you may also want your query to use values from another, non-XML column, to bring relational and XML data together. To do this, you use the **sql:column()** function.

The SQL value will be mapped to a corresponding XQuery value and its type will be an XQuery base type that is equivalent to the corresponding SQL type.

Syntax

```
sql:column("columnName")
```

Remarks

Note that reference to a column specified in the **sql:column()** function inside an XQuery refers to a column in the row that is being processed.

In SQL Server, you can only refer to an **xml** instance in the context of the source expression of an XML-DML insert statement; otherwise, you cannot refer to columns that are of type **xml** or a CLR user-defined type.

The **sql:column()** function is not supported in JOIN operations. The APPLY operation can be used instead.

Examples

A. Using sql:column() to retrieve the relational value inside XML

In constructing XML, the following example illustrates how you can retrieve values from a non-XML relational column to bind XML and relational data.

The query constructs XML that has the following form:

```
<Product ProductID="771" ProductName="Mountain-100 Silver, 38"
ProductPrice="3399.99" ProductModelID="19"
  ProductModelName="Mountain 100" />
```

Note the following in the constructed XML:

- The **ProductID**, **ProductName**, and **ProductPrice** attribute values are obtained from the **Product** table.
- The **ProductModelID** attribute value is retrieved from the **ProductModel** table.

- To make the query more interesting, the **ProductModelName** attribute value is obtained from the **CatalogDescription** column of **xml type**. Because the XML product model catalog information is not stored for all the product models, the `if` statement is used to retrieve the value only if it exists.

```
SELECT P.ProductID, CatalogDescription.query('
declare namespace
pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
       <Product
           ProductID=       "{ sql:column("P.ProductID") }"
           ProductName=     "{ sql:column("P.Name") }"
           ProductPrice=    "{ sql:column("P.ListPrice") }"
           ProductModelID= "{ sql:column("PM.ProductModelID") }"
>
           { if (not(empty(/pd:ProductDescription))) then
             attribute ProductModelName {
/pd:ProductDescription[1]/@ProductModelName }
            else
               ()
}
        </Product>
') as Result
FROM Production.ProductModel PM, Production.Product P
WHERE PM.ProductModelID = P.ProductModelID
AND   CatalogDescription is not NULL
ORDER By PM.ProductModelID
```

Note the following from the previous query:

- Because the values are retrieved from two different tables, the FROM clause specifies two tables. The condition in the WHERE clause filters the result and retrieves only products whose product models have catalog descriptions.
- The **namespace** keyword in the XQuery Prolog defines the XML namespace prefix, "pd", that is used in the query body. Note that the table aliases, "P" and "PM", are defined in the FROM clause of the query itself.
- The **sql:column()** function is used to bring non-XML values inside XML.

This is the partial result:

```
ProductID              Result
```

```
-----------------------------------------------------------------
771         <Product ProductID="771"
ProductName="Mountain-100 Silver, 38"
                  ProductPrice="3399.99" ProductModelID="19"
                  ProductModelName="Mountain 100" />
...
```

The following query constructs XML that contains product-specific information. This information includes the ProductID, ProductName, ProductPrice, and, if available, the ProductModelName for all products that belong to a specific product model, ProductModelID=19. The XML is then assigned to the @x variable of **xml** type.

```
declare @x xml
SELECT @x = CatalogDescription.query('
declare namespace
pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
       <Product
           ProductID=       "{ sql:column("P.ProductID") }"
           ProductName=     "{ sql:column("P.Name") }"
           ProductPrice=    "{ sql:column("P.ListPrice") }"
           ProductModelID= "{ sql:column("PM.ProductModelID") }" >
           { if (not(empty(/pd:ProductDescription))) then
             attribute ProductModelName {
/pd:ProductDescription[1]/@ProductModelName }
            else
               ()
}
        </Product>
')
FROM Production.ProductModel PM, Production.Product P
WHERE PM.ProductModelID = P.ProductModelID
And P.ProductModelID = 19
select @x
```

See Also

XML Data Modification Language (XML DML)

Typed vs. Untyped XML

XML Data (SQL Server)
Generating XML Instances
XML Data Type Methods
XML Data Modification Language (XML DML)

sql:variable() Function

Exposes a variable that contains a SQL relational value inside an XQuery expression.

Syntax

```
sql:variable("variableName") as xdt:anyAtomicType?
```

Remarks

As described in the topic Binding Relational Data Inside XML, you can use this function when you use XML data type methods to expose a relational value inside XQuery.

For example, the query() method is used to specify a query against an XML instance that is stored in an **xml** data type variable or column. Sometimes, you might also want your query to use values from a Transact-SQL variable, or parameter, to bring relational and XML data together. To do this, you use the **sql:variable** function.

The SQL value will be mapped to a corresponding XQuery value and its type will be an XQuery base type that is equivalent to the corresponding SQL type.

You can only refer to an **xml** instance in the context of the source expression of an XML-DML insert statement; otherwise you cannot refer to values that are of type **xml** or a common language runtime (CLR) user-defined type.

Examples

A. Using the sql:variable() function to bring a Transact-SQL variable value into XML

The following example constructs an XML instance that made up of the following:

- A value (`ProductID`) from a non-XML column. The sql:column() function is used to bind this value in the XML.
- A value (`ListPrice`) from a non-XML column from another table. Again, `sql:column()` is used to bind this value in the XML.
- A value (`DiscountPrice`) from a Transact-SQL variable. The `sql:variable()` method is used to bind this value into the XML.
- A value (`ProductModelName`) from an **xml** type column, to make the query more interesting.

This is the query:

```
DECLARE @price money
```

```
SET @price=2500.00
SELECT ProductID,
Production.ProductModel.ProductModelID,CatalogDescription.query('
declare namespace
pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";

       <Product
           ProductID="{ sql:column("Production.Product.ProductID") }"
           ProductModelID= "{
sql:column("Production.Product.ProductModelID") }"

ProductModelName="{/pd:ProductDescription[1]/@ProductModelName }"
           ListPrice="{ sql:column("Production.Product.ListPrice") }"
           DiscountPrice="{ sql:variable("@price") }"
        />')
FROM Production.Product
JOIN Production.ProductModel
ON Production.Product.ProductModelID =
Production.ProductModel.ProductModelID
WHERE ProductID=771
```

Note the following from the previous query:

- The XQuery inside the `query()` method constructs the XML.
- The `namespace` keyword is used to define a namespace prefix in the XQuery Prolog. This is done because the `ProductModelName` attribute value is retrieved from the `CatalogDescription xml` type column, which has a schema associated with it.

This is the result:

```
<Product ProductID="771" ProductModelID="19"
         ProductModelName="Mountain 100"
         ListPrice="3399.99" DiscountPrice="2500" />
```

See Also

XML Data Modification Language (XML DML)

Typed vs. Untyped XML

XML Data (SQL Server)

Generating XML Instances

XQuery Operators Against the xml Data Type

XQuery supports the following operators:

- Numeric operators (+, -, *, div, mod)
- Operators for value comparison (eq, ne, lt, gt, le, ge)
- Operators for general comparison (=, !=, <, >, <=, >=)

For more information about these operators, see XQuery Language Reference (Database Engine)

Examples

A. Using general operators

The query illustrates the use of general operators that apply to sequences, and also to compare sequences. The query retrieves a sequence of telephone numbers for each customer from the **AdditionalContactInfo** column of the **Contact** table. This sequence is then compared with the sequence of two telephone numbers ("111-111-1111", "222-2222").

The query uses the = comparison operator. Each node in the sequence on the right side of the = operator is compared with each node in the sequence on the left side. If the nodes match, the node comparison is **TRUE**. It is then converted to an int and compared with 1, and the query returns the customer ID.

```
WITH XMLNAMESPACES (
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactInfo' AS ACI,
'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes' AS ACT)
SELECT ContactID
FROM   Person.Contact
WHERE  AdditionalContactInfo.value('
      //ACI:AdditionalContactInfo//ACT:telephoneNumber/ACT:number =
          ("111-111-1111", "222-2222")',
      'bit')= cast(1 as bit)
```

There is another way to observe how the previous query works: Each phone telephone number value retrieved from the **AdditionalContactInfo** column is compared with the

set of two telephone numbers. If the value is in the set, that customer is returned in the result.

B. Using a numeric operator

The + operator in this query is a value operator, because it applies to a single item. For example, value 1 is added to a lot size that is returned by the query:

```
SELECT ProductModelID, Instructions.query('
     declare namespace
 AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
     for $i in (/AWMI:root/AWMI:Location)[1]
     return
       <Location LocationID="{ ($i/@LocationID) }"
                   LotSize  = "{  number($i/@LotSize) }"
                   LotSize2 = "{ number($i/@LotSize) + 1 }"
                   LotSize3 = "{ number($i/@LotSize) + 2 }" >


       </Location>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

C. Using a value operator

The following query retrieves the <Picture> elements for a product model where the picture size is "small":

```
SELECT CatalogDescription.query('
     declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     for $P in /PD:ProductDescription/PD:Picture[PD:Size eq "small"]
     return
           $P
    ') as Result
FROM Production.ProductModel
where ProductModelID=19
```

Because both the operands to the **eq** operator are atomic values, the value operator is used in the query. You can write the same query by using the general comparison operator (**=**).

See Also

XQuery Functions Against the XML Data Type

XML Data (SQL Server)

XQuery Against the XML Data Type

Additional Sample XQueries Against the xml Data Type

The topics in this section provide additional samples that show how to use XQuery queries against the **xml** data type:

- General XQuery Use Cases
- XQueries Involving Hierarchy
- XQueries Involving Order
- XQueries Handling Relational Data
- String Search in XQuery
- Handling Namespaces in XQuery

See Also

XQuery Functions Against the XML Data Type

XQuery Operators Against the XML Data Type

General XQuery Use Cases

This topic provides general examples of XQuery use.

Examples

A. Query catalog descriptions to find products and weights

The following query returns the product model IDs and weights, if they exist, from the product catalog description. The query constructs XML that has the following form:

```
<Product ProductModelID="…">
  <Weight>…</Weight>
</Product>
```

This is the query:

```
SELECT CatalogDescription.query('
declare namespace
p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
  <Product  ProductModelID="{
(/p1:ProductDescription/@ProductModelID)[1] }">
     {
       /p1:ProductDescription/p1:Specifications/Weight
     }
  </Product>
') as Result
FROM Production.ProductModel
WHERE CatalogDescription is not null
```

Note the following from the previous query:

- The **namespace** keyword in the XQuery prolog defines a namespace prefix that is used in the query body.
- The query body constructs the required XML.
- In the WHERE clause, the **exist()** method is used to find only rows that contain product catalog descriptions. That is, the XML that contains the `<ProductDescription>` element.

This is the result:

```
<Product ProductModelID="19"/>
<Product ProductModelID="23"/>
<Product ProductModelID="25"/>
<Product ProductModelID="28"><Weight>Varies with
size.</Weight></Product>
<Product ProductModelID="34"/>
<Product ProductModelID="35"/>
```

The following query retrieves the same information, but only for those product models whose catalog description includes the weight, the `<Weight>` element, in the specifications, the `<Specifications>` element. This example uses WITH XMLNAMESPACES to declare the pd prefix and its namespace binding. In this way, the binding is not described in both the **query()** method and in the **exist()** method.

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS pd)
SELECT CatalogDescription.query('
          <Product  ProductModelID="{
(/pd:ProductDescription/@ProductModelID)[1] }">
                {
                     /pd:ProductDescription/pd:Specifications/Weight
                }
          </Product>
') as x
FROM Production.ProductModel
WHERE
CatalogDescription.exist('/pd:ProductDescription/pd:Specifications//Wei
ght ') = 1
```

In the previous query, the **exist()** method of the **xml** data type in the WHERE clause checks to see if there is a `<Weight>` element in the `<Specifications>` element.

B. Find product model IDs for product models whose catalog descriptions include front-angle and small size pictures

The XML product catalog description includes the product pictures, the `<Picture>` element. Each picture has several properties. These include the picture angle, the `<Angle>` element, and the size, the `<Size>` element.

For product models whose catalog descriptions include front-angle and small-size pictures, the query constructs XML that has the following form:

```
< Product ProductModelID="…">
  <Picture>
    <Angle>front</Angle>
    <Size>small</Size>
  </Picture>
</Product>
```

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS pd)
SELECT CatalogDescription.query('
   <pd:Product  ProductModelID="{
(/pd:ProductDescription/@ProductModelID)[1] }">
```

```
      <Picture>
         {  /pd:ProductDescription/pd:Picture/pd:Angle }
         {  /pd:ProductDescription/pd:Picture/pd:Size }
      </Picture>
   </pd:Product>
') as Result
FROM  Production.ProductModel
WHERE CatalogDescription.exist('/pd:ProductDescription/pd:Picture') = 1
AND
CatalogDescription.value('(/pd:ProductDescription/pd:Picture/pd:Angle)[
1]', 'varchar(20)')  = 'front'
AND
CatalogDescription.value('(/pd:ProductDescription/pd:Picture/pd:Size)[1
]', 'varchar(20)')  = 'small'
```

Note the following from the previous query:

- In the WHERE clause, the **exist()** method is used to retrieve only rows that have product catalog descriptions with the `<Picture>` element.
- The WHERE clause uses the **value()** method two times to compare the values of the `<Size>` and `<Angle>` elements.

This is a partial result:

```
<p1:Product
  xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription"
  ProductModelID="19">
  <Picture>
    <p1:Angle>front</p1:Angle>
    <p1:Size>small</p1:Size>
  </Picture>
</p1:Product>
...
```

C. Create a flat list of the product model name and feature pairs, with each pair enclosed in the <Features> element

In the product model catalog description, the XML includes several product features. All these features are included in the `<Features>` element. The query uses XML

Construction (XQuery) to construct the required XML. The expression in the curly braces is replaced by the result.

```
SELECT CatalogDescription.query('
declare namespace
p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
  for $pd in /p1:ProductDescription,
   $f in $pd/p1:Features/*
  return
   <Feature>
     <ProductModelName> { data($pd/@ProductModelName) }
</ProductModelName>
     { $f }
  </Feature>
') as x
FROM Production.ProductModel
WHERE ProductModelID=19
```

Note the following from the previous query:

- $pd/p1:Features/* returns only the element node children of `<Features>`, but $pd/p1:Features/node() returns all the nodes. This includes the element nodes, text nodes, processing instructions, and comments.
- The two FOR loops generate a Cartesian product from which the product name and the individual feature are returned.
- The **ProductName** is an attribute. The XML construction in this query returns it as an element.

This is a partial result:

```
<Feature>
 <ProductModelName>Mountain 100</ProductModelName>
 <ProductModelID>19</ProductModelID>
 <p1:Warranty
   xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
    <p1:WarrantyPeriod>3 year</p1:WarrantyPeriod>
    <p1:Description>parts and labor</p1:Description>
 </p1:Warranty>
```

```
</Feature>
<Feature>
 <ProductModelName>Mountain 100</ProductModelName>
 <ProductModelID>19</ProductModelID>
 <p2:Maintenance
xmlns:p2="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
    <p2:NoOfYears>10</p2:NoOfYears>
    <p2:Description>maintenance contact available through your dealer
           or any AdventureWorks retail store.</p2:Description>
    </p2:Maintenance>
</Feature>
...
...
```

D. From the catalog description of a product model, list the product model name, model ID, and features grouped inside a <Product> element

Using the information stored in the catalog description of the product model, the following query lists the product model name, model ID, and features grouped inside a <Product> element.

```
SELECT ProductModelID, CatalogDescription.query('
     declare namespace
pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     <Product>
         <ProductModelName>
           { data(/pd:ProductDescription/@ProductModelName) }
         </ProductModelName>
         <ProductModelID>
           { data(/pd:ProductDescription/@ProductModelID) }
         </ProductModelID>
         { /pd:ProductDescription/pd:Features/* }
     </Product>
') as x
FROM Production.ProductModel
```

```
WHERE ProductModelID=19
```

This is a partial result:

```
<Product>
  <ProductModelName>Mountain 100</ProductModelName>
  <ProductModelID>19</ProductModelID>
  <p1:Warranty>... </p1:Warranty>
  <p2:Maintenance>...  </p2:Maintenance>
  <p3:wheel xmlns:p3="http://www.adventure-
works.com/schemas/OtherFeatures">High performance wheels.</p3:wheel>
  <p4:saddle xmlns:p4="http://www.adventure-
works.com/schemas/OtherFeatures">
    <p5:i xmlns:p5="http://www.w3.org/1999/xhtml">Anatomic
design</p5:i> and made from durable leather for a full-day of riding in
comfort.</p4:saddle>
  <p6:pedal xmlns:p6="http://www.adventure-
works.com/schemas/OtherFeatures">
    <p7:b xmlns:p7="http://www.w3.org/1999/xhtml">Top-of-the-
line</p7:b> clipless pedals with adjustable tension.</p6:pedal>
   ...
```

E. Retrieve product model feature descriptions

The following query constructs XML that includes a `<Product>` element that has **ProducModelID**, **ProductModelName** attributes, and the first two product features. Specifically, the first two product features are the first two child elements of the `<Features>` element. If there are more features, it returns an empty `<There-is-more/>` element.

```
SELECT CatalogDescription.query('
declare namespace
pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     <Product>
          { /pd:ProductDescription/@ProductModelID }
          { /pd:ProductDescription/@ProductModelName }
          {
            for $f in
/pd:ProductDescription/pd:Features/*[position()<=2]
            return
```

```
            $f
          }
          {
            if (count(/pd:ProductDescription/pd:Features/*) > 2)
            then <there-is-more/>
            else ()
          }
     </Product>
') as Result
FROM Production.ProductModel
WHERE CatalogDescription is not NULL
```

Note the following from the previous query:

- The FOR ... RETURN loop structure retrieves the first two product features. The **position()** function is used to find the position of the elements in the sequence.

F. Find element names from the product catalog description that end with "ons"

The following query searches the catalog descriptions and returns all the elements in the <ProductDescription> element whose name ends with "ons".

```
SELECT ProductModelID, CatalogDescription.query('
     declare namespace
p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
      for $pd in /p1:ProductDescription/*[substring(local-
name(.),string-length(local-name(.))-2,3)="ons"]
      return
          <Root>
             { $pd }
          </Root>
') as Result
FROM Production.ProductModel
WHERE CatalogDescription is not NULL
```

This is a partial result:

```
ProductModelID   Result
----------------------------------------
```

```
         19        <Root>
                     <p1:Specifications
xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription">
                           ...
                     </p1:Specifications>
                   </Root>
```

G. Find summary descriptions that contain the word "Aerodynamic"

The following query retrieves product models whose catalog descriptions contain the word "Aerodynamic" in the summary description:

```
WITH XMLNAMESPACES
('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription' AS pd)
SELECT ProductModelID, CatalogDescription.query('
          <Prod >
             { /pd:ProductDescription/@ProductModelID }
             { /pd:ProductDescription/pd:Summary }
          </Prod>
 ') as Result
FROM Production.ProductModel
WHERE CatalogDescription.value('
     contains( string( (/pd:ProductDescription/pd:Summary)[1]
),"Aerodynamic")','bit') = 1
```

Note that the SELECT query specifies **query()** and **value()** methods of the **xml** data type. Therefore, instead of repeating the namespaces declaration two times in two difference query prologs, the prefix pd is used in the query and is defined only once by using WITH XMLNAMESPACES.

Note the following from the previous query:

- The WHERE clause is used to retrieve only the rows where the catalog description contains the word "Aerodynamic" in the `<Summary>` element.
- The **contains()** function is used to see if the word is included in the text.
- The **value()** method of the **xml** data type compares the value returned by **contains()** to 1.

This is the result:

```
ProductModelID Result
-------------- ----------------------------------------
```

```
28     <Prod ProductModelID="28">
        <pd:Summary
xmlns:pd="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription">
       <p1:p xmlns:p1="http://www.w3.org/1999/xhtml">
         A TRUE multi-sport bike that offers streamlined riding and a
         revolutionary design. Aerodynamic design lets you ride with
the
         pros, and the gearing will conquer hilly roads.</p1:p>
       </pd:Summary>
      </Prod>
```

H. Find product models whose catalog descriptions do not include product model pictures

The following query retrieves ProductModelIDs for product models whose catalog descriptions do no include a <Picture> element.

```
SELECT  ProductModelID
FROM    Production.ProductModel
WHERE   CatalogDescription is not NULL
AND     CatalogDescription.exist('declare namespace
p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     /p1:ProductDescription/p1:Picture
') = 0
```

Note the following from the previous query:

- If the **exist()** method in the WHERE clause returns False (0), the product model ID is returned. Otherwise, it is not returned.
- Because all the product descriptions include a <Picture> element, the result set is empty in this case.

See Also

XQueries Involving Hierarchy
XQueries Involving Order
XQueries Handling Relational Data
String Search in XQuery
Handling Namespaces in XQuery
Adding Namespaces Using WITH XMLNAMESPACES

XML Data (SQL Server)
XQuery Language Reference (SQL Server)

XQueries Involving Hierarchy

Most **xml** type columns in the **AdventureWorks** database are semi-structured documents. Therefore, documents stored in each row may look different. The query samples in this topic illustrate how to extract information from these various documents.

Examples

A. From the manufacturing instructions documents, retrieve work center locations together with the first manufacturing step at those locations

For product model 7, the query constructs XML that includes the `<ManuInstr>` element, with **ProductModelID** and **ProductModelName** attributes, and one or more `<Location>` child elements.

Each `<Location>` element has its own set of attributes and one `<step>` child element. This `<step>` child element is the first manufacturing step at the work center location.

```
SELECT Instructions.query('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
   <ManuInstr  ProdModelID =
"{sql:column("Production.ProductModel.ProductModelID") }"
                ProductModelName = "{
sql:column("Production.ProductModel.Name") }" >
            {
              for $wc in //AWMI:root/AWMI:Location
              return
                <Location>
                 {$wc/@* }
                 <step1> { string( ($wc//AWMI:step)[1] ) } </step1>
                </Location>
            }
          </ManuInstr>
') as Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

Note the following from the previous query:

- The **namespace** keyword in the XQuery Prolog defines a namespace prefix. This prefix is used later in the query body.
- The context switching tokens, {) and (}, are used to switch the query from XML construction to query evaluation.
- The **sql:column()** is used to include a relational value in the XML that is being constructed.
- In constructing the <Location> element, $wc/@* retrieves all the work center location attributes.
- The **string()** function returns the string value from the <step> element.

This is a partial result:

```
<ManuInstr ProdModelID="7" ProductModelName="HL Touring Frame">
   <Location LocationID="10" SetupHours="0.5"
            MachineHours="3" LaborHours="2.5" LotSize="100">
     <step1>Insert aluminum sheet MS-2341 into the T-85A
             framing tool.</step1>
   </Location>
   <Location LocationID="20" SetupHours="0.15"
            MachineHours="2" LaborHours="1.75" LotSize="1">
      <step1>Assemble all frame components following
             blueprint 1299.</step1>
   </Location>
...
</ManuInstr>
```

B. Find all telephone numbers in the AdditionalContactInfo column

The following query retrieves additional telephone numbers for a specific customer contact by searching the whole hierarchy for the <telephoneNumber> element. Because the <telephoneNumber> element can appear anywhere in the hierarchy, the query uses the descendant and self operator (//) in the search.

```
SELECT AdditionalContactInfo.query('
 declare namespace
ci="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactInfo";
 declare namespace
act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ContactTypes";
```

```
for $ph in /ci:AdditionalContactInfo//act:telephoneNumber
   return
      $ph/act:number
') as x
FROM  Person.Contact
WHERE ContactID = 1
```

This is the result:

```
<act:number
  xmlns:act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ContactTypes">
  111-111-1111
</act:number>
<act:number
  xmlns:act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ContactTypes">
  112-111-1111
</act:number>
```

To retrieve only the top-level telephone numbers, specifically the `<telephoneNumber>` child elements of `<AdditionalContactInfo>`, the FOR expression in the query changes to

```
for $ph in /ci:AdditionalContactInfo/act:telephoneNumber.
```

See Also

Implementing XML in SQL Server

XML Construction (XQuery)

XML Data (SQL Server)

XQueries Involving Order

Relational databases do not have a concept of sequence. For example, you cannot make a request such as "Get the first customer from the database." However, you can query an XML document and retrieve the first <Customer> element. Then, you will always retrieve the same customer.

This topic illustrates queries based on the sequence in which nodes appear in the document.

Examples

A. Retrieve manufacturing steps at the second work center location for a product

For a specific product model, the following query retrieves manufacturing steps at the second work center location in a sequence of work center locations in the manufacturing process.

```
SELECT Instructions.query('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
    <ManuStep ProdModelID =
"{sql:column("Production.ProductModel.ProductModelID")}"
                ProductModelName = "{
sql:column("Production.ProductModel.Name") }" >
     <Location>
       { (//AWMI:root/AWMI:Location)[2]/@* }
       <Steps>
         { for $s in (//AWMI:root/AWMI:Location)[2]//AWMI:step
           return
              <Step>
               { string($s) }
              </Step>
         }
        </Steps>
      </Location>
     </ManuStep>
') as Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

Note the following from the previous query:

- The expressions in the curly braces are replaced by the result of its evaluation. For more information, see XML Construction (XQuery).
- **@*** retrieves all the attributes of the second work center location.
- The FLWOR iteration (FOR ... RETURN) retrieves all the `<step>` child elements of the second work center location.
- The sql:column() function (XQuery) includes the relational value in the XML that is being constructed.

This is the result:

```
<ManuStep ProdModelID="7" ProductModelName="HL Touring Frame">
  <Location LocationID="20" SetupHours="0.15"
              MachineHours="2"  LaborHours="1.75" LotSize="1">
  <Steps>
   <Step>Assemble all frame components following blueprint 1299.</Step>
     …
  </Steps>
 </Location>
</ManuStep>
```

The previous query retrieves just the text nodes. If you want the whole `<step>` element returned instead, remove the **string()** function from the query:

B. Find all the material and tools used at the second work center location in the manufacturing of a product

For a specific product model, the following query retrieves the tools and materials used at the second work center location in the sequence of work center locations in the manufacturing process.

```
SELECT Instructions.query('
    declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
   <Location>
      { (//AWMI:root/AWMI:Location)[1]/@* }
       <Tools>
         { for $s in
(//AWMI:root/AWMI:Location)[1]//AWMI:step//AWMI:tool
           return
              <Tool>
                { string($s) }
              </Tool>
          }
        </Tools>
        <Materials>
            { for $s in
(//AWMI:root/AWMI:Location)[1]//AWMI:step//AWMI:material
```

```
              return
                 <Material>
                    { string($s) }
                 </Material>
              }
          </Materials>
  </Location>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

Note the following from the previous query:

- The query constructs the <Location> element and retrieves its attribute values from the database.
- It uses two FLWOR (for...return) iterations: one to retrieve tools and one to retrieve the material used.

This is the result:

```
<Location LocationID="10" SetupHours=".5"
          MachineHours="3" LaborHours="2.5" LotSize="100">
  <Tools>
    <Tool>T-85A framing tool</Tool>
    <Tool>Trim Jig TJ-26</Tool>
    <Tool>router with a carbide tip 15</Tool>
    <Tool>Forming Tool FT-15</Tool>
  </Tools>
  <Materials>
    <Material>aluminum sheet MS-2341</Material>
  </Materials>
</Location>
```

C. Retrieve the first two product feature descriptions from the product catalog

For a specific product model, the query retrieves the first two feature descriptions from the <Features> element in the product model catalog.

```
SELECT CatalogDescription.query('
```

```
     declare namespace
p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     <ProductModel ProductModelID= "{ data(
(/p1:ProductDescription/@ProductModelID)[1] ) }"
                    ProductModelName = "{ data(
(/p1:ProductDescription/@ProductModelName)[1] ) }" >
       {
         for $F in /p1:ProductDescription/p1:Features
         return
           $F/*[position() <= 2]
       }
     </ProductModel>
      ') as x
FROM Production.ProductModel
where ProductModelID=19
```

Note the following from the previous query:

The query body constructs XML that includes the `<ProductModel>` element that has the ProductModelID and ProductModelName attributes.

- The query uses a FOR ... RETURN loop to retrieve the product model feature descriptions. The **position()** function is used to retrieve the first two features.

This is the result:

```
<ProductModel ProductModelID="19" ProductModelName="Mountain 100">
 <p1:Warranty
  xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
  <p1:WarrantyPeriod>3 year</p1:WarrantyPeriod>
  <p1:Description>parts and labor</p1:Description>
 </p1:Warranty>
 <p2:Maintenance
  xmlns:p2="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
  <p2:NoOfYears>10</p2:NoOfYears>
  <p2:Description>maintenance contact available through your dealer
            or any AdventureWorks retail store.
```

```
  </p2:Description>
 </p2:Maintenance>
</ProductModel>
```

D. Find the first two tools used at the first work center location in the manufacturing process of the product

For a product model, this query returns the first two tools used at the first work center location in the sequence of work center locations in the manufacturing process. The query is specified against the manufacturing instructions stored in the **Instructions** column of the **Production.ProductModel** table.

```
SELECT Instructions.query('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
   for $Inst in (//AWMI:root/AWMI:Location)[1]
   return
     <Location>
       { $Inst/@* }
       <Tools>
         { for $s in ($Inst//AWMI:step//AWMI:tool)[position() <= 2]
           return
             <Tool>
               { string($s) }
             </Tool>
         }
       </Tools>
     </Location>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

This is the result:

```
<Location LocationID="10" SetupHours=".5"
            MachineHours="3" LaborHours="2.5" LotSize="100">
  <Tools>
    <Tool>T-85A framing tool</Tool>
```

```
    <Tool>Trim Jig TJ-26</Tool>
  </Tools>
</Location>
```

E. Find the last two manufacturing steps at the first work center location in the manufacturing of a specific product

The query uses the **last()** function to retrieve the last two manufacturing steps.

```
SELECT Instructions.query('
declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
  <LastTwoManuSteps>
   <Last-1Step>
     { (/AWMI:root/AWMI:Location)[1]/AWMI:step[(last()-1)]/text() }
   </Last-1Step>
   <LastStep>
     { (/AWMI:root/AWMI:Location)[1]/AWMI:step[last()]/text() }
   </LastStep>
  </LastTwoManuSteps>') as Result
FROM Production.ProductModel
where ProductModelID=7
```

This is the result:

```
<LastTwoManuSteps>
   <Last-1Step>When finished, inspect the forms for defects per
               Inspection Specification .</Last-1Step>
   <LastStep>Remove the frames from the tool and place them in the
             Completed or Rejected bin as appropriate.</LastStep>
</LastTwoManuSteps>
```

See Also

XML Data (SQL Server)
XQuery Against the XML Data Type
XML Construction (XQuery)

XQueries Handling Relational Data

You specify XQuery against an **xml** type column or variable by using one of the XML Data Type Methods. These include **query()**, **value()**, **exist()**, or **modify()**. The XQuery is executed against the XML instance identified in the query generating the XML.

The XML generated by the execution of an XQuery can include values retrieved from other Transact-SQL variable or rowset columns. To bind non-XML relational data to the resulting XML, SQL Server provides the following pseudo functions as XQuery extensions:

- **sql:column()** function
- **sql:variable()** function

You can use these XQuery extensions when specifying an XQuery in the **query()** method of the **xml** data type. As a result, the **query()** method can produce XML that combines data from XML and non-**xml** data types.

You can also use these functions when you use the **xml** data type methods **modify()**, **value()**, **query()**, and **exist()**to expose a relational value inside XML.

For more information, see sql:column() function (XQuery) and sql:variable() function (XQuery).

See Also

XML Data (SQL Server)

XQuery Against the XML Data Type

XML Construction (XQuery)

String Search in XQuery

This topic provides sample queries that show how to search text in XML documents.

Examples

A. Find feature descriptions that contain the word "maintenance" in the product catalog

```
SELECT CatalogDescription.query('
     declare namespace
p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
    for $f in /p1:ProductDescription/p1:Features/*
     where contains(string($f), "maintenance")
     return
           $f ') as Result
FROM Production.ProductModel
```

```
WHERE ProductModelID=19
```

In the previous query, the `where` in the FLOWR expression filters the result of the `for` expression and returns only elements that satisfy the **contains()** condition.

This is the result:

```
<p1:Maintenance

xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelWarrAndMain">
 <p1:NoOfYears>10</p1:NoOfYears>
 <p1:Description>maintenance contact available through your
               dealer or any AdventureWorks retail
store.</p1:Description>
</p1:Maintenance>
```

See Also

XML Data (SQL Server)

XQuery Against the XML Data Type

Handling Namespaces in XQuery

This topic provides samples for handling namespaces in queries.

Examples

A. Declaring a namespace

The following query retrieves the manufacturing steps of a specific product model.

```
SELECT Instructions.query('
     declare namespace
AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
        /AWMI:root/AWMI:Location[1]/AWMI:step
    ') as x
FROM Production.ProductModel
WHERE ProductModelID=7
```

This is the partial result:

```
<AWMI:step
xmlns:AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions">Insert <AWMI:material>aluminum
```

```
sheet MS-2341</AWMI:material> into the <AWMI:tool>T-85A framing
tool</AWMI:tool>. </AWMI:step>
```

…

Note that the **namespace** keyword is used to define a new namespace prefix, "AWMI:". This prefix then must be used in the query for all elements that fall within the scope of that namespace.

B. Declaring a default namespace

In the previous query, a new namespace prefix was defined. That prefix then had to be used in the query to select the intended XML structures. Alternatively, you can declare a namespace as the default namespace, as shown in the following modified query:

```
SELECT Instructions.query('
     declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions";
        /root/Location[1]/step
    ') as x
FROM Production.ProductModel
where ProductModelID=7
```

This is the result

```
<step xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelManuInstructions">Insert <material>aluminum sheet MS-
2341</material> into the <tool>T-85A framing tool</tool>. </step>
```

…

Note in this example that the namespace defined, `"http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManuInstructions"`, is made to override the default, or empty, namespace. Because of this, you no longer have a namespace prefix in the path expression that is used to query. You also no longer have a namespace prefix in the element names that appear in the results. Additionally, the default namespace is applied to all elements, but not to their attributes.

C. Using namespaces in XML construction

When you define new namespaces, they are brought into scope not only for the query, but for the construction. For example, in constructing XML, you can define a new namespace by using the "`declare namespace ...`" declaration and then use that namespace with any elements and attributes that you construct to appear within the query results.

```
SELECT CatalogDescription.query('
```

```
     declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
     declare namespace myNS="uri:SomeNamespace";
       <myNS:Result>
          { /ProductDescription/Summary }
       </myNS:Result>

    ') as Result
FROM Production.ProductModel
where ProductModelID=19
```

This is the result:

```
<myNS:Result xmlns:myNS="uri:SomeNamespace">
  <Summary
xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription">
   <p1:p xmlns:p1="http://www.w3.org/1999/xhtml">
     Our top-of-the-line competition mountain bike. Performance-
enhancing
     options include the innovative HL Frame, super-smooth front
     suspension, and traction for all terrain.</p1:p>
  </Summary>
</myNS:Result>
```

Alternatively, you can also define the namespace explicitly at each point where it is used as part of the XML construction, as shown in the following query:

```
SELECT CatalogDescription.query('
     declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
       <myNS:Result xmlns:myNS="uri:SomeNamespace">
          { /ProductDescription/Summary }
       </myNS:Result>
    ') as Result
FROM Production.ProductModel
where ProductModelID=19
```

D. Construction using default namespaces

You can also define a default namespace for use in constructed XML. For example, the following query shows how you can specify a default namespace, "uri:SomeNamespace"\, to use as the default for the locally named elements that are constructed, such as the <Result> element.

```
SELECT CatalogDescription.query('
      declare namespace
PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription";
      declare default element namespace "uri:SomeNamespace";
       <Result>
          { /PD:ProductDescription/PD:Summary }
       </Result>

    ') as Result
FROM Production.ProductModel
where ProductModelID=19
```

This is the result:

```
<Result xmlns="uri:SomeNamespace">
  <PD:Summary
xmlns:PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/ProductModelDescription">
   <p1:p xmlns:p1="http://www.w3.org/1999/xhtml">
         Our top-of-the-line competition mountain bike. Performance-
         enhancing options include the innovative HL Frame, super-
smooth
         front suspension, and traction for all terrain.</p1:p>
  </PD:Summary>
</Result>
```

Note that by overriding the default element namespace or empty namespace, all the locally named elements in the constructed XML are subsequently bound to the overriding default namespace. Therefore, if you require flexibility in constructing XML to take advantage of the empty namespace, do not override the default element namespace.

See Also

XQuery Language Reference (Database Engine)

XML Data (SQL Server)
XQuery Against the XML Data Type