CHAPTER 4

Querying and Shredding XML

To allow developers to query and navigate XML documents from within SQL Server, the XQuery language can be combined with T-SQL queries. In this chapter, I will discuss how to use XQuery to filter, extract, and modify XML. I will also discuss shredding XML, which is the process of converting XML data into relational results sets. Finally, an overview of how to bind an XSD schema to a column of data type XML is provided.

Querying XML

XQuery is a language for querying XML, in the same way that SQL is a language for querying relational data. The language is built on XPath but has been enhanced for better iteration and sorting. It also allows for the construction of XML. The XQuery standard is developed by the W3C (World Wide Web Consortium), in conjunction with Microsoft and other major relational database management system (RDBMS) vendors.

XQuery in SQL Server supports five methods against the XML data type. An overview of these methods can be found in Table 4-1, and each of the methods will be demonstrated throughout this chapter. XQuery also supports the use of XSD schemas, to help interact with complex documents and `FLWOR` (pronounced flower) statements. `FLWOR` is an acronym of `for`, `let`, `where`, `order by`, and `return`.

P. A. Carter, *SQL Server Advanced Data Types*,
https://doi.org/10.1007/978-1-4842-3901-8_4

Table 4-1. *XQuery Methods*

Method	Description
`exist()`	Checks for the existence of a node with a specified value within an XML document. It returns 1 if the node exists with the specified value and 0 if it does not.
`modify()`	Performs data modification statements against an XML document. Acceptable DML actions are insert, delete, and replace value of.
`nodes()`	Returns a row set that contains copies of original XML instances. It is used for shredding XML into a relational result set.
`query()`	This returns a subset of an XML document, in XML format.
`value()`	This returns a single scalar value from an XML document, mapped to an SQL Server data type.

Tip The structure and benefits of XSD schemas are explained in Chapter 2.

`for` statements allow you to iterate through a sequence of nodes. `let` statements bind a sequence to a variable. `where` statements filter nodes based on a Boolean expression. `order by` statements order nodes before they are returned. `return` statements specify what should be returned.

To demonstrate the use of the XQuery methods in this chapter, we will create a table in the WideWorldImporters database, called Sales.CustomerOrderSummary. This table can be created using the script in Listing 4-1.

Listing 4-1. Creating the Sales.CustomerOrderSummary Table

```
USE WideWorldImporters
GO

CREATE TABLE Sales.CustomerOrderSummary
(
        ID INT NOT NULL IDENTITY,
        CustomerID INT NOT NULL,
        OrderSummary XML
) ;

INSERT INTO Sales.CustomerOrderSummary (CustomerID,
OrderSummary)
SELECT
        CustomerID,
        (
                SELECT
                         CustomerName 'OrderHeader/CustomerName'
                        , OrderDate 'OrderHeader/OrderDate'
                        , OrderID 'OrderHeader/OrderID'
                        , (
                                SELECT
                                          LineItems2.StockItemID
                                          '@ProductID'
                                        , StockItems.StockItem
                                        Name '@ProductName'
                                        , LineItems2.UnitPrice
                                        '@Price'
                                        , Quantity '@Qty'
                                FROM Sales.OrderLines LineItems2
                                INNER JOIN Warehouse.StockItems
                                StockItems
```

```
                                        ON LineItems2.StockItemID
                                        = StockItems.StockItemID
                                WHERE LineItems2.OrderID =
                                Base.OrderID
                                FOR XML PATH('Product'), TYPE
                        ) 'OrderDetails'
                FROM
                (
                        SELECT DISTINCT
                                  Customers.CustomerName
                                , SalesOrder.OrderDate
                                , SalesOrder.OrderID
                        FROM Sales.Orders SalesOrder
                        INNER JOIN Sales.OrderLines LineItem
                                ON SalesOrder.OrderID =
                                LineItem.OrderID
                        INNER JOIN Sales.Customers Customers
                                ON Customers.CustomerID =
                                SalesOrder.CustomerID
                        WHERE customers.CustomerID = OuterCust.
                        CustomerID
                ) Base
                FOR XML PATH('Order'), ROOT ('SalesOrders'), TYPE
        ) AS OrderSummary
FROM Sales.Customers OuterCust ;
```

Using exist()

The `exist()` method is used to check for the existence of a node with a specified value. For example, please consider the script in Listing 4-2. The query will programmatically check to see if the XML document containing sales order details contains the Chocolate sharks 250g product. The first

portion of the XQuery defines the path to the node to be evaluated, in this case, the `StockItemName` attribute. We have defined that the node is an attribute, by prefixing the node name with the `@` symbol. We then specify `eq` as the comparison operator, before supplying the value that we want to validate the node against. If the criteria are met, `exist()` will return `1`. If the criteria are not met, it will return `0`.

Listing 4-2. Checking for the Existence of a Value

```
DECLARE @SalesOrders XML

SET @SalesOrders =
'<SalesOrder OrderDate="2016-05-27" OrderID="73356">
          <Customers CustomerName="Agrita Abele">
                <Product StockItemName="Chocolate sharks 250g">
                  <LineItem Quantity="192" UnitPrice="8.55" />
                </Product>
          </Customers>
        </SalesOrder>' ;

SELECT @SalesOrders.exist('SalesOrder/Customers/Product[
(@StockItemName) eq "Chocolate sharks 250g"]') ;
```

The results of this script are displayed in Figure 4-1.

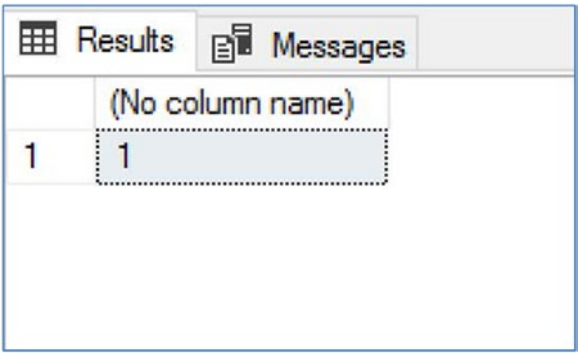


Figure 4-1. *Results of checking for the existence of a value*

You can see how the `exist()` method could easily be used in a `WHERE` clause, against a column of data type `XML`. For example, consider the script in Listing 4-3, which uses the `exist()` method in a `WHERE` clause against the Sales.CustomerOrderSummary table in the WideWorldImporters database, to return just the order summary for Tailspin Toys (Absecon, NJ).

You will notice that because we are evaluating the value of an element, as opposed to an attribute, we have used the `text()` method to extract the value. The `[1]` denotes that a singleton value will be returned.

Listing 4-3. Filtering a Table Using the `exist()` Method

```
SELECT
        CustomerID
      , OrderSummary
FROM WideWorldImporters.Sales.CustomerOrderSummary
WHERE OrderSummary.exist('SalesOrders/Order/OrderHeader/
CustomerName[(text()[1]) eq "Tailspin Toys (Absecon, NJ)"]') = 1 ;
```

Using value( )

The `value()` method is used to extract a single, scalar value from an XML document and map it to an SQL Server data type. For example, consider the script in Listing 4-4. The script uses the same XML document as Listing 4-1 but this time extracts the customer name from the document. In the same way as when we used the `exist()` method, we will have to use the `@` symbol to prefix an attribute. We also must denote that a singleton value will be returned.

Listing 4-4. Using the value() Method

```
DECLARE @SalesOrders XML ;

SET @SalesOrders =
'<SalesOrder OrderDate="2016-05-27" OrderID="73356">
          <Customers CustomerName="Agrita Abele">
                <Product StockItemName="Chocolate sharks 250g">
                  <LineItem Quantity="192" UnitPrice="8.55" />
                </Product>
          </Customers>
        </SalesOrder>' ;

SELECT @SalesOrders.value('(/SalesOrder/Customers/
@CustomerName)[1]', 'nvarchar(100)') AS CustomerName ;
```

The results of this script are illustrated in Figure 4-2.

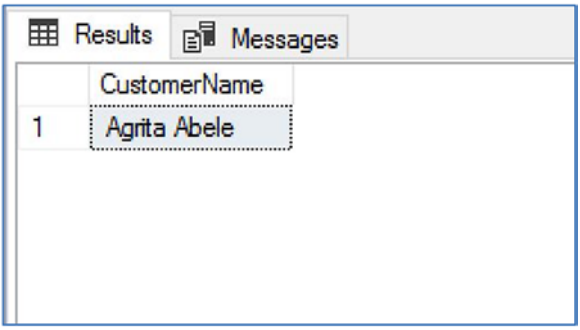


Figure 4-2. *Results of using the value() method*

The value() method can also be used against a table, to extract values from each row. For example, the script in Listing 4-5 can be used to extract the customer name from each row in the Sales.CustomerOrderSummary table.

Listing 4-5. Using the `value()` Method Against a Table

```
USE WideWorldImporters
GO

SELECT
      CustomerID
    , OrderSummary
    , OrderSummary.value('(/SalesOrders/Order/OrderHeader/
    CustomerName)[1]', 'nvarchar(100)') AS CustomerName
FROM Sales.CustomerOrderSummary ;
```

Partial results of this query can be found in Figure 4-3.

Results Messages

	CustomerID	OrderSummary	CustomerName
1	841	<SlesOrders><Order><OrderHeader><CustomerName>Ca...	Camille Authier
2	83	<SlesOrders><Order><OrderHeader><CustomerName>Tail...	Tailspin Toys (Absecon, NJ)
3	595	<SlesOrders><Order><OrderHeader><CustomerName>Wi...	Wingtip Toys (Accomac, VA)
4	84	<SlesOrders><Order><OrderHeader><CustomerName>Tail...	Tailspin Toys (Aceitunas, PR)
5	38	<SlesOrders><Order><OrderHeader><CustomerName>Tail...	Tailspin Toys (Airport Drive, MO)
6	537	<SlesOrders><Order><OrderHeader><CustomerName>Wi...	Wingtip Toys (Akhiok, AK)
7	853	<SlesOrders><Order><OrderHeader><CustomerName>Cat...	Caterina Pinto
8	589	<SlesOrders><Order><OrderHeader><CustomerName>Wi...	Wingtip Toys (Alcester, SD)
9	892	<SlesOrders><Order><OrderHeader><CustomerName>Ba...	Bahaar Asef zade
10	91	<SlesOrders><Order><OrderHeader><CustomerName>Tail...	Tailspin Toys (Alstead, NH)
11	566	<SlesOrders><Order><OrderHeader><CustomerName>Wi...	Wingtip Toys (Amado, AZ)
12	44	<SlesOrders><Order><OrderHeader><CustomerName>Tail...	Tailspin Toys (Amanda Park, WA)
13	85	<SlesOrders><Order><OrderHeader><CustomerName>Tail...	Tailspin Toys (Andrix, CO)
14	125	<SlesOrders><Order><OrderHeader><CustomerName>Tail...	Tailspin Toys (Annamoriah, WV)
15	151	<SlesOrders><Order><OrderHeader><CustomerName>Tail...	Tailspin Toys (Antares, AZ)

Query executed successfully. DATATYPES (14.0 RTM

Figure 4-3. *Results of using the `value()` method against a table*

The `value()` method can also be used in the `WHERE` clause of a query. For example, the query in Listing 4-6 is functionally equivalent to the query in Listing 4-5. It has simply been rewritten using the `value()` method instead of the `exist()` method.

Listing 4-6. Filtering a Table Using the value() Method

```
USE WideWorldImporters
GO

SELECT
        CustomerID
      , OrderSummary
FROM WideWorldImporters.Sales.CustomerOrderSummary
WHERE OrderSummary.value('(/SalesOrders/Order/OrderHeader/
CustomerName)[1]', 'nvarchar(100)') = 'Tailspin Toys (Absecon, NJ)' ;
```

Using query( )

The query() method is used to return an untyped XML document from an XML document. For example, consider the script in Listing 4-7. The query will extract the product details from the XML document in an XML format.

Listing 4-7. Extracting Product Details

```
DECLARE @SalesOrders XML ;

SET @SalesOrders =
'<SalesOrder OrderDate="2016-05-27" OrderID="73356">
          <Customers CustomerName="Agrita Abele">
                <Product StockItemName="Chocolate sharks 250g">
                  <LineItem Quantity="192" UnitPrice="8.55" />
                </Product>
          </Customers>
        </SalesOrder>' ;

SELECT @SalesOrders.query('/SalesOrder/Customers/Product') AS
ProductDetails ;
```

The results of this query can be seen in Figure 4-4.

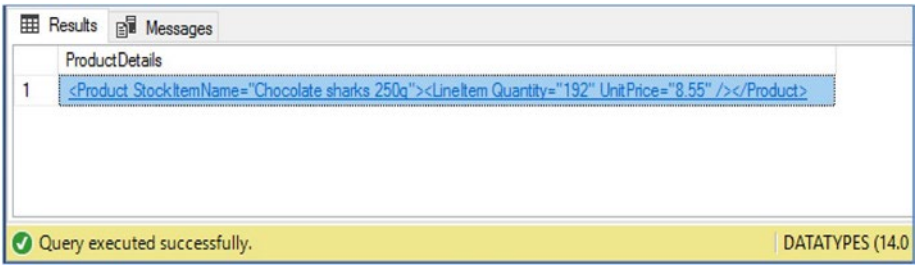


Figure 4-4. *Results of extracting product details*

To see how the `query()` method could be used when querying a table, consider the script in Listing 4-8. This query uses all of the XQuery methods that have been discussed so far. The `value()` method is used to extract the customer's name and order ID, while the `query()` method is used to extract details of the products that were ordered. The table is filtered using the `exist()` method.

Listing 4-8. Using `query()`, `value()`, and `exist()` Against a Table

```
USE WideWorldImporters
GO

SELECT
          OrderSummary
        , OrderSummary.value('(/SalesOrders/Order/OrderHeader/
        CustomerName)[1]', 'nvarchar(100)') AS CustomerName
         , OrderSummary.value('(/SalesOrders/Order/OrderHeader/
         OrderID)[1]', 'int') AS OrderID
         , OrderSummary.query('/SalesOrders/Order/OrderDetails/
         Product') AS ProductsOrdered
FROM Sales.CustomerOrderSummary
WHERE OrderSummary.exist('SalesOrders/Order/OrderHeader/
CustomerName[(text()[1]) eq "Tailspin Toys (Absecon, NJ)"]') = 1 ;
```

This query returns the results shown in Figure 4-5.

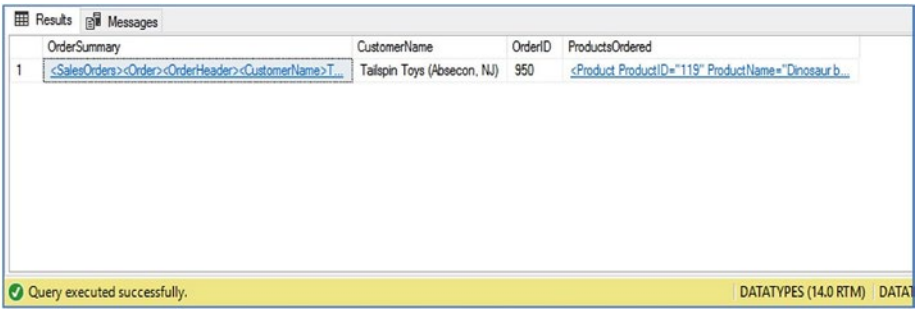


Figure 4-5. *Results of using* `query()`, `value()`, *and* `exist()` *against a table*

Using Relational Values in XQuery

Relational values from T-SQL variables and columns can also be passed into XQuery expressions. They can be used for filtering or even constructing data, but they are read-only. Therefore, the XQuery expression cannot be used to modify the relational variable or column value.

To see this functionality in action, consider the query in Listing 4-9. The query is functionally equivalent to the query in Listing 4-7. This time, however, instead of using a hard-coded filter for the customer name, we pass this in from a T-SQL variable. Queries that use T-SQL variables or columns from tables are known as cross-domain queries.

Listing 4-9. Parameterizing an `exist()` Method

```
USE WideWorldImporters
GO

DECLARE @CustomerName NVARCHAR(100) ;

SET @CustomerName = 'Tailspin Toys (Absecon, NJ)' ;

SELECT
```

```
        OrderSummary
      , OrderSummary.value('(/SalesOrders/Order/OrderHeader/
      CustomerName)[1]', 'nvarchar(100)') AS CustomerName
       , OrderSummary.value('(/SalesOrders/Order/OrderHeader/
       OrderID)[1]', 'int') AS OrderID
       , OrderSummary.query('/SalesOrders/Order/OrderDetails/
       Product') AS ProductsOrdered
FROM Sales.CustomerOrderSummary
WHERE OrderSummary.exist('SalesOrders/Order/OrderHeader/
CustomerName[(text()[1]) eq sql:variable("@CustomerName") ]') = 1 ;
```

T-SQL variables can also be used in the construction of XML. For example, the script in Listing 4-10 generates a new column in the result set, which contains an XML fragment. This fragment has an element called `CustomerDetails`. Within this element, you will notice an attribute called `GoldCustomer`. This is a flag that is configured from a T-SQL variable. Note that when used inside the `query()` method, we have enclosed the `sql:variable` statement inside double quotes and curly brackets.

Listing 4-10. Constructing an XML Fragment and Passing a Value from a T-SQL Variable

```
USE WideWorldImporters
GO

DECLARE @Gold NVARCHAR(3)

DECLARE @CustomerName NVARCHAR(100)

SET @Gold = 'Yes'

SET @CustomerName = 'Tailspin Toys (Absecon, NJ)'
```

```
SELECT
      OrderSummary
        , OrderSummary.value('(/SalesOrders/Order/OrderHeader/
        CustomerName)[1]', 'nvarchar(100)') AS CustomerName
        , OrderSummary.value('(/SalesOrders/Order/OrderHeader/
        OrderID)[1]', 'int') AS OrderID
        , OrderSummary.query('/SalesOrders/Order/OrderDetails/
        Product') AS ProductsOrdered
        , OrderSummary.query('<CustomerDetails>GoldCustomer =
        "{ sql:variable("@Gold") }"  </CustomerDetails>') AS
        CustomerDetails
FROM Sales.CustomerOrderSummary
WHERE OrderSummary.exist('SalesOrders/Order/OrderHeader/
CustomerName[(text()[1]) eq sql:variable("@CustomerName") ]') = 1 ;
```

The results of this query can be found in Figure 4-6. The resultant XML fragment can be found in the CustomerDetails column.

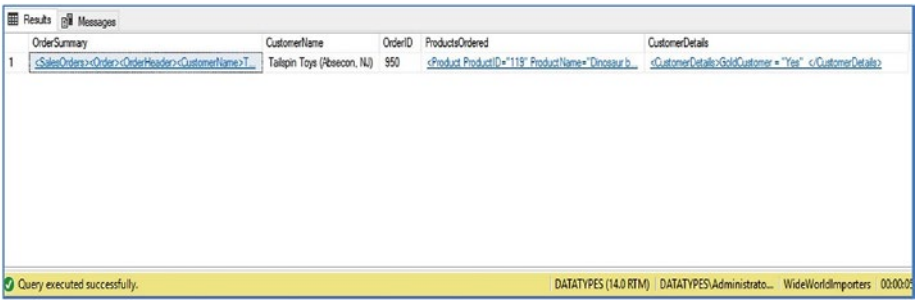


Figure 4-6. *Results of constructing an XML fragment and passing a value from a T-SQL variable*

When the XQuery expression is run against an XML column within a table, relational data, stored in other columns, can also be passed to the XQuery expression. For example, the script in Listing 4-11 expands the generated CustomerDetails XML fragment to include the Customer ID from the CustomerID column of the Sales.CustomerOrderSummary table.

Listing 4-11. Constructing an XML Fragment Using a Relational Column

```
USE WideWorldImporters
GO

DECLARE @Gold NVARCHAR(3)

DECLARE @CustomerName NVARCHAR(100)

SET @Gold = 'Yes'

SET @CustomerName = 'Tailspin Toys (Absecon, NJ)'

SELECT
      OrderSummary
        , OrderSummary.value('(/SalesOrders/Order/OrderHeader/
        CustomerName)[1]', 'nvarchar(100)') AS CustomerName
        , OrderSummary.value('(/SalesOrders/Order/OrderHeader/
        OrderID)[1]', 'int') AS OrderID
        , OrderSummary.query('/SalesOrders/Order/OrderDetails/
        Product') AS ProductsOrdered
        , OrderSummary.query('<CustomerDetails> CustomerID =
        "{ sql:column("CustomerID") }" GoldCustomer =
        "{ sql:variable("@Gold") }"  </CustomerDetails>')
        As CustomerDetails
FROM Sales.CustomerOrderSummary
WHERE OrderSummary.exist('SalesOrders/Order/OrderHeader/
CustomerName[(text()[1]) eq sql:variable("@CustomerName") ]') = 1 ;
```

The results of this query can be seen in Figure 4-7. Once again, pay attention to the CustomerDetails column, to see the resultant XML fragment.

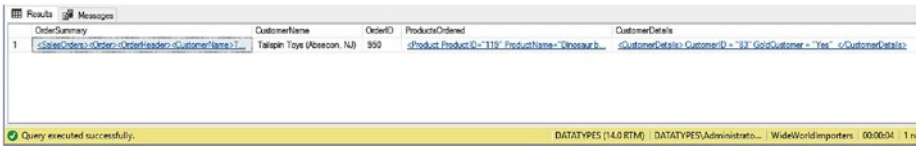


Figure 4-7. *Results of constructing an XML fragment using a relational column*

FLWOR

As previously mentioned, `FLWOR` stands for `for`, `let`, `where`, `order by`, and `return`. These statements provide granular control, allowing a developer to navigate to, iterate over, filter, and present XML nodes exactly as required.

The `for` statement binds a variable to an input sequence. The `let` statement is used to assign an XQuery expression to a variable, for use within an iteration of `FOR`. The expression can return either atomic values or a sequence of nodes. The `let` statement is optional. The `where` statement is also optional but can be used to filter the results that are returned. The `order by` statement can be used optionally to order the results of the `FLWOR` statement. The mandatory `return` statement specifies what data will be returned.

For example, consider the XML document in Listing 4-12. This XML document contains the first two orders from the XML document returned by the OrderSummary column in Listing 4-11.

Listing 4-12. XML Document for FLWOR Examples

```
DECLARE @XML XML = N'<SalesOrders>
  <Order>
    <OrderHeader>
      <CustomerName>Tailspin Toys (Absecon, NJ)</CustomerName>
      <OrderDate>2013-01-17</OrderDate>
      <OrderID>950</OrderID>
    </OrderHeader>
    <OrderDetails>
      <Product ProductID="119" ProductName="Dinosaur battery-
      powered slippers (Green) M" Price="32.00" Qty="2" />
      <Product ProductID="61" ProductName="RC toy sedan car with
      remote control (Green) 1/50 scale" Price="25.00" Qty="2" />
      <Product ProductID="194" ProductName="Black and orange glass
      with care despatch tape  48mmx100m" Price="4.10" Qty="216" />
      <Product ProductID="104" ProductName="Alien officer
      hoodie (Black) 3XL" Price="35.00" Qty="2" />
    </OrderDetails>
  </Order>
  <Order>
    <OrderHeader>
      <CustomerName>Tailspin Toys (Absecon, NJ)</CustomerName>
      <OrderDate>2013-01-29</OrderDate>
      <OrderID>1452</OrderID>
    </OrderHeader>
    <OrderDetails>
      <Product ProductID="33" ProductName="Developer joke mug -
      that's a hardware problem (Black)" Price="13.00" Qty="9" />
```

```
      <Product ProductID="121" ProductName="Dinosaur battery-
      powered slippers (Green) XL" Price="32.00" Qty="1" />
    </OrderDetails>
  </Order>
<SalesOrders>'
```

If we wanted to iterate over each `Product` element within the second order, we could use the `FOR` statement as follows in Listing 4-13.

Note that we use an internal variable, which we name `$product`, and the `in` keyword to designate the path to the `ProductName` attribute. This approach and syntax will be familiar to those of you who have worked with `foreach` loops in languages such as PowerShell.

The `return` statement is then used to append the `ProductName` attribute's value to the end of the `$product` string. Note that we use `[2]` in square brackets, to denote that we are interested in the second order, which will be a singleton element (albeit a complex element containing multiple other elements).

Caution The following code examples in this section expect a variable called `XML` to be declared that contains the XML document in Listing 4-12. For space reasons, this variable is not explicitly declared or set within each code example.

Listing 4-13. Using `for` to Iterate over Product Elements

```
SELECT @XML.query('
        for $product in /SalesOrders/Order[2]/OrderDetails/
        Product/@ProductName
        return string($product)
') ;
```

The results of this query are illustrated in Figure 4-8.

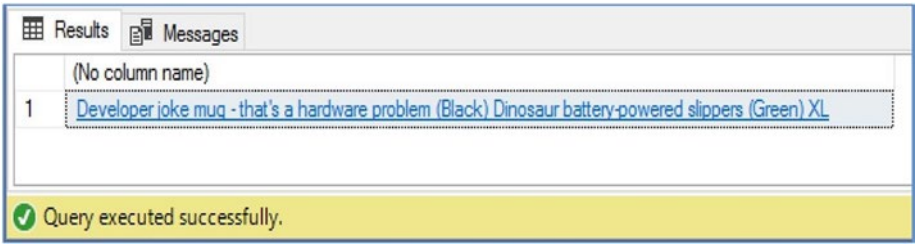


Figure 4-8. *Results of using* `for` *to iterate over product elements*

The query in Listing 4-14 uses a `let` statement, instead of a `for` iteration, to retrieve the name of the first product, within the first order, within the XML document. The `return` statement requires a singleton value, and as `let` simply assigns a value, as opposed to iterating over multiple values, we cannot return multiple products.

Listing 4-14. Using `let` to Find a Product Name

```
SELECT @XML.query('
         let $product := /SalesOrders/Order[1]/OrderDetails/
         Product/@ProductName
        return string($product[1])
') ;
```

The results of this query can be found in Figure 4-9.

Figure 4-9. *Results of using* `let` *to find a product name*

The query in Listing 4-15 combines both a for statement and a let statement to construct a new XML document containing the customer name and product name for each product sold.

Listing 4-15. Combining for and let

```
SELECT @XML.query('
    for $product in /SalesOrders/Order/OrderDetails/Product/
    @ProductName
         let $customer := /SalesOrders/Order/OrderHeader/
         CustomerName
        return
        <Customer>
          {$customer[1]}
        <OrderDetails>
          {$product}
        </OrderDetails>
        </Customer>
') ;
```

The resultant XML document can be found in Listing 4-16.

Listing 4-16. Results of Combining for and let

```
<Customer>
  <CustomerName>Tailspin Toys (Absecon, NJ)</CustomerName>
  <OrderDetails ProductName="Dinosaur battery-powered slippers
  (Green) M" />
</Customer>
<Customer>
  <CustomerName>Tailspin Toys (Absecon, NJ)</CustomerName>
  <OrderDetails ProductName="RC toy sedan car with remote
  control (Green) 1/50 scale" />
</Customer>
```

```
<Customer>
  <CustomerName>Tailspin Toys (Absecon, NJ)</CustomerName>
  <OrderDetails ProductName="Black and orange glass with care
  despatch tape  48mmx100m" />
</Customer>
<Customer>
  <CustomerName>Tailspin Toys (Absecon, NJ)</CustomerName>
  <OrderDetails ProductName="Alien officer hoodie (Black) 3XL" />
</Customer>
<Customer>
  <CustomerName>Tailspin Toys (Absecon, NJ)</CustomerName>
  <OrderDetails ProductName="Developer joke mug - that's a
  hardware problem (Black)" />
</Customer>
<Customer>
  <CustomerName>Tailspin Toys (Absecon, NJ)</CustomerName>
  <OrderDetails ProductName="Dinosaur battery-powered slippers
  (Green) XL" />
</Customer>
```

The query in Listing 4-17 enhances the query in Listing 4-15, to add a `where` statement, which will filter the products, so that only the dinosaur slippers are returned. The query also uses an `order by` statement, to guarantee that the order of the results is by product name.

Listing 4-17. Using `where` and `order by`

```
SELECT @XML.query('
    for $product in /SalesOrders/Order/OrderDetails/Product/
    @ProductName
         let $customer := /SalesOrders/Order/OrderHeader/
         CustomerName
```

```
         where $product = "Dinosaur battery-powered slippers
         (Green) M"
           or $product = "Dinosaur battery-powered slippers
           (Green) XL"
         order by $product
        return
        <Customer>
          {$customer[1]}
        <OrderDetails>
          {$product}
        </OrderDetails>
        </Customer>
') ;
```

Modifying XML Data

XML data can be modified using XQuery's `modify` method. When using `modify`, a developer has three options. You can use the `insert` option, the `delete` option, or the `replace value of` option. The `replace value of` option replaces an existing value within an XML document.

To understand how the `insert` option works, consider the script in Listing 4-18. The script populates a variable with an empty order. The `modify` method is then used to add an order line to the sales order.

Listing 4-18. Using the `modify` Insert

```
DECLARE @SalesOrder xml;
SET @SalesOrder = '
<Order>
  <OrderHeader>
    <CustomerName>Camille Authier</CustomerName>
    <OrderDate>2013-01-02</OrderDate>
```

```
    <OrderID>121</OrderID>
  </OrderHeader>
  <OrderDetails>
  </OrderDetails>
</Order>'  ;

SET @SalesOrder.modify('
insert <Product ProductID="22" ProductName="DBA joke mug - it
depends (White)" Price="13" Qty="6" />
into (/Order/OrderDetails)[1]') ;

SELECT @SalesOrder ;
```

The results of running this query can be seen in Listing 4-19.

Listing 4-19. Results of Using the `modify` Insert

```
<Order>
  <OrderHeader>
    <CustomerName>Camille Authier</CustomerName>
    <OrderDate>2013-01-02</OrderDate>
    <OrderID>121</OrderID>
  </OrderHeader>
  <OrderDetails>
    <Product ProductID="22" ProductName="DBA joke mug - it
    depends (White)" Price="13" Qty="6" />
  </OrderDetails>
</Order>
```

If `Product` elements already existed within the `OrderDetails` element, however, you could gain granular control over where you would like the element to be inserted by using the `as first`, `as last`, `before`, or `after` options. For example, the query in Listing 4-20 will insert the new `Product` element as the first element in the `OrderDetails` element.

Listing 4-20. Inserting an Element As First

```
DECLARE @SalesOrder xml;
SET @SalesOrder = '
<Order>
  <OrderHeader>
    <CustomerName>Camille Authier</CustomerName>
    <OrderDate>2013-01-02</OrderDate>
    <OrderID>121</OrderID>
  </OrderHeader>
  <OrderDetails>
    <Product ProductID="22" ProductName="DBA joke mug - it
    depends (White)" Price="13" Qty="6" />
  </OrderDetails>
</Order>'  ;

SET @SalesOrder.modify('
insert <Product ProductID="2" ProductName="USB rocket launcher
(Gray)" Price="25" Qty="9" /> as first
into (/Order/OrderDetails)[1]') ;

SELECT @SalesOrder ;
```

The results of this query can be seen in Listing 4-21.

Listing 4-21. Results of Inserting an Element As First

```
<Order>
  <OrderHeader>
    <CustomerName>Camille Authier</CustomerName>
    <OrderDate>2013-01-02</OrderDate>
    <OrderID>121</OrderID>
  </OrderHeader>
```

```
  <OrderDetails>
    <Product ProductID="2" ProductName="USB rocket launcher
    (Gray)" Price="25" Qty="9" />
    <Product ProductID="22" ProductName="DBA joke mug - it
    depends (White)" Price="13" Qty="6" />
  </OrderDetails>
</Order>
```

Alternatively, the query in Listing 4-22 demonstrates how you can insert an element after another, specific element. In this case, we will insert the Superhero Action Jacket after the USB Rocket Launcher.

Listing 4-22. Inserting an Element After Another Element

```
DECLARE @SalesOrder xml;
SET @SalesOrder = '
<Order>
  <OrderHeader>
    <CustomerName>Camille Authier</CustomerName>
    <OrderDate>2013-01-02</OrderDate>
    <OrderID>121</OrderID>
  </OrderHeader>
  <OrderDetails>
    <Product ProductID="2" ProductName="USB rocket launcher
    (Gray)" Price="25" Qty="9" />
    <Product ProductID="22" ProductName="DBA joke mug - it
    depends (White)" Price="13" Qty="6" />
  </OrderDetails>
</Order>'  ;

DECLARE @ProductName NVARCHAR(200) ;

SET @ProductName = 'USB rocket launcher (Gray)' ;
```

```
SET @SalesOrder.modify('
insert <Product ProductID="111" ProductName="Superhero action
jacket (Blue) M" Price="30" Qty="10" />
after (/Order/OrderDetails/Product[@ProductName =
sql:variable("@ProductName")])[1]') ;

SELECT @SalesOrder ;
```

This query uses `sql:variable` to pass the product's name from an T-SQL variable, using the techniques you learned in the "Using Relational Values in XQuery" section of this chapter. The results of the query can be seen in Listing 4-23.

Listing 4-23. Results of Inserting an Element After Another Element

```
<Order>
  <OrderHeader>
    <CustomerName>Camille Authier</CustomerName>
    <OrderDate>2013-01-02</OrderDate>
    <OrderID>121</OrderID>
  </OrderHeader>
  <OrderDetails>
    <Product ProductID="2" ProductName="USB rocket launcher
    (Gray)" Price="25" Qty="9" />
    <Product ProductID="111" ProductName="Superhero action
    jacket (Blue) M" Price="30" Qty="10" />
    <Product ProductID="22" ProductName="DBA joke mug - it
    depends (White)" Price="13" Qty="6" />
  </OrderDetails>
</Order>
```

The delete option can be used to remove nodes from an XML document. For example, consider the script in Listing 4-24. Here, we take the XML document produced by the query in Listing 4-22 and remove the last order line that we added to the document.

Listing 4-24. Using the delete Option

```
DECLARE @SalesOrder xml;
SET @SalesOrder = '
<Order>
  <OrderHeader>
    <CustomerName>Camille Authier</CustomerName>
    <OrderDate>2013-01-02</OrderDate>
    <OrderID>121</OrderID>
  </OrderHeader>
  <OrderDetails>
    <Product ProductID="2" ProductName="USB rocket launcher
    (Gray)" Price="25" Qty="9" />
    <Product ProductID="111" ProductName="Superhero action
    jacket (Blue) M" Price="30" Qty="10" />
    <Product ProductID="22" ProductName="DBA joke mug - it
    depends (White)" Price="13" Qty="6" />
  </OrderDetails>
</Order>'  ;

DECLARE @ProductName NVARCHAR(200) ;

SET @ProductName = 'Superhero action jacket (Blue) M' ;

SET @SalesOrder.modify('
delete (/Order/OrderDetails/Product[@ProductName =
sql:variable("@ProductName")])[1]') ;

SELECT @SalesOrder ;
```

You can see the resultant XML document in Listing 4-25.

Listing 4-25. Results of Using the `delete` Option

```
<Order>
  <OrderHeader>
    <CustomerName>Camille Authier</CustomerName>
    <OrderDate>2013-01-02</OrderDate>
    <OrderID>121</OrderID>
  </OrderHeader>
  <OrderDetails>
    <Product ProductID="2" ProductName="USB rocket launcher
    (Gray)" Price="25" Qty="9" />
    <Product ProductID="22" ProductName="DBA joke mug - it
    depends (White)" Price="13" Qty="6" />
  </OrderDetails>
</Order>
```

The `replace` value of `option` performs an update on an existing node. For example, the script in Listing 4-26 updates the quantity of DBA Joke Mugs ordered from six to ten. Once again, we bind SQL variables to allow us to pass in the `ProductName` that we wish to update and the new quantity. Note that we use `replace value of` to define the node that should be updated and `with` to define the new value.

Listing 4-26. Using `replace value of`

```
DECLARE @SalesOrder xml;
SET @SalesOrder = '
<Order>
  <OrderHeader>
    <CustomerName>Camille Authier</CustomerName>
```

```
    <OrderDate>2013-01-02</OrderDate>
    <OrderID>121</OrderID>
  </OrderHeader>
  <OrderDetails>
    <Product ProductID="2" ProductName="USB rocket launcher
    (Gray)" Price="25" Qty="9" />
    <Product ProductID="22" ProductName="DBA joke mug - it
    depends (White)" Price="13" Qty="6" />
  </OrderDetails>
</Order>'  ;

DECLARE @ProductName NVARCHAR(200) ;

SET @ProductName = 'DBA joke mug - it depends (White)' ;

DECLARE @Quantity INT ;

SET @Quantity = 10

SET @SalesOrder.modify('
replace value of (/Order/OrderDetails/Product[@Product
Name = sql:variable("@ProductName")]/@Qty)[1]
with "10"
') ;

SELECT @SalesOrder ;
```

The results of this query can be seen in Listing 4-27.

Listing 4-27. Results of Using `replace value of`

```
<Order>
  <OrderHeader>
    <CustomerName>Camille Authier</CustomerName>
    <OrderDate>2013-01-02</OrderDate>
    <OrderID>121</OrderID>
  </OrderHeader>
  <OrderDetails>
    <Product ProductID="2" ProductName="USB rocket launcher
    (Gray)" Price="25" Qty="9" />
    <Product ProductID="22" ProductName="DBA joke mug - it
    depends (White)" Price="13" Qty="10" />
  </OrderDetails>
</Order>
```

Shredding XML

Shredding XML is the process of taking an XML result set and converting it to a relational result set. In SQL Server, there are two ways to achieve this: the `OPENXML()` function and the `nodes()` XQuery method.

Shredding XML with OPENXML( )

`OPENXML()` is a function that takes an XML document and converts it to a relational result set. It accepts the parameters detailed in Table 4-2.

Table 4-2. *OPENXML() Parameters*

Parameter	Description
idoc	A pointer to an internal representation of the XML document
rowpattern	The path to the lowest level of the XML document that should be converted to rows
flags	An optional parameter that specifies the mapping between the XML data and relational result set. The possible values are detailed in Table 4-3.

Table 4-3 details the valid flags values that can be passed to OPENXML().

Table 4-3. *Flags Values*

Value	Description
0	Defaults to attribute-centric mapping
1	Use attribute-centric mapping, unless XML_ELEMENTS is specified. If XML_ELEMENTS is specified, attributes will be mapped first, followed by elements.
2	Element-centric mapping, unless XML_ATTRIBUTES is specified. If XML_ATTRIBUTES is specified, elements will be mapped first, followed by attributes.
8	XML_ATTRIBUTES and XML_ELEMENTS, combined with a logical OR

In addition to the function's parameters, a WITH clause can also be specified, which defines the schema of the result set to be returned. While the WITH clause is optional, not specifying the schema will cause an edge table to be returned. The schema is used to specify a relation column name, a relational data type, and the path to the relevant node.

Note The flags will be overridden by specific mapping in the WITH clause.

A limitation of the OPENXML() function is that it does not parse an XML document itself. Instead, you must parse the document before calling the function. This can be achieved using the sp_xml:preparedocument system stored procedure. This procedure parses an XML document, using the MSXML parser, and creates a handle to the parsed version of the document that is ready for consumption.

After the OPENXML() function has completed, you should use the sp_xml:removedocument system stored procedure to remove the parsed document from memory. It is important to remember to tear down the parsed document, or you could start running into memory issues, if the sp_xml:preparedocument procedure is called frequently.

To see the OPENXML() function in action, see the script in Listing 4-28. This script passes a sales orders document into a variable, before using OPENXML() to shred the data.

Tip The structure of the XML document has been changed from previous examples in this chapter, to allow a broader demonstration of the navigation of an XML document with OPENXML().

Listing 4-28. Using OPENXML()

```
DECLARE @SalesOrder XML ;
DECLARE @ParseDoc INT ;

SET @SalesOrder = '
<SalesOrders>
  <Order>
```

```
    <OrderDate>2013-01-02</OrderDate>
    <OrderHeader>
      <CustomerName>Camille Authier</CustomerName>
    </OrderHeader>
    <OrderDetails>
      <Product ProductID="45" ProductName="Developer joke
      mug - there are 10 types of people in the world (Black)"
      Price="13" Qty="7" />
      <Product ProductID="58" ProductName="RC toy sedan car with
      remote control (Black) 1/50 scale" Price="25" Qty="4" />
    </OrderDetails>
  </Order>
  <Order>
    <OrderDate>2013-01-02</OrderDate>
    <OrderHeader>
      <CustomerName>Camille Authier</CustomerName>
        </OrderHeader>
    <OrderDetails OrderID = "122">
      <Product ProductID="22" ProductName="DBA joke mug - it
      depends (White)" Price="13" Qty="6" />
      <Product ProductID="2" ProductName="USB rocket launcher
      (Gray)" Price="25" Qty="9" />
      <Product ProductID="111" ProductName="Superhero action
      jacket (Blue) M" Price="30" Qty="10" />
      <Product ProductID="116" ProductName="Superhero action
      jacket (Blue) 4XL" Price="34" Qty="4" />
    </OrderDetails>
  </Order>
</SalesOrders>' ;

EXEC sp_xml:preparedocument @ParseDoc OUTPUT, @SalesOrder ;
SELECT *
```

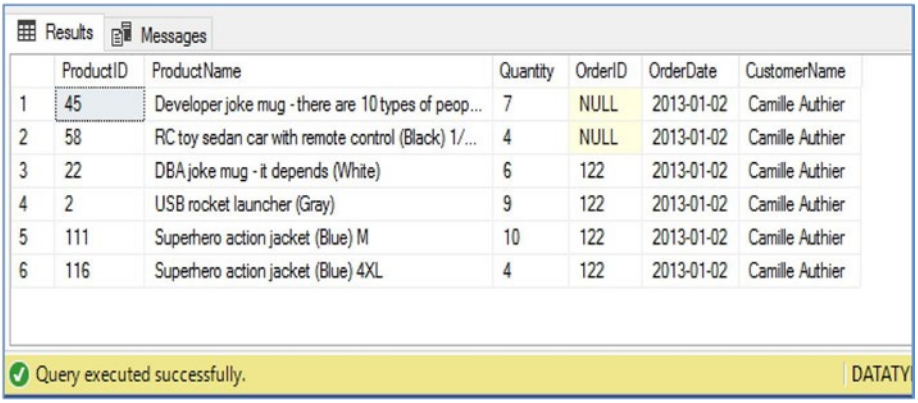
```
FROM OPENXML(@ParseDoc, '/SalesOrders/Order/OrderDetails/Product')
WITH (
                    ProductID          INT          '@ProductID',
    ProductName     NVARCHAR(200)    '@ProductName',
    Quantity           INT                          '@Qty',
    OrderID            INT                        '../@OrderID',
    OrderDate           DATE                '../../OrderDate',
    CustomerName  NVARCHAR(200)  '../../OrderHeader/CustomerName'
) ;

EXEC sp_xml:removedocument @ParseDoc ;
```

When you examine the `WITH` clause of the `OPENXML()` statement of this script, you will see that we first specify the relational column name. By looking at the `Quantity` mapping, you will see that the relational names do not have to match the names of the XML nodes. We then specify a relational data type for the resultant value, before specifying a path to the node that we wish to retrieve. This is the most interesting aspect of the query. You will notice that we have used the `rowpattern` parameter of the `OPENXML()` function to map down to the `Product` element, as the lowest level. This means that the `Product` element is the starting point for our paths. The `ProductID`, `ProductName`, and `Quantity` attributes are all attributes of the `Product` element. Therefore, we prefix these with the `@` symbol, to designate that they are attributes, but no other path mapping is required. The `OrderID` attribute is an attribute of the `OrderDetails` element. Because the `OrderDetails` element is one level above the `Product` element, we use the `../` syntax, to specify that we must navigate up one level. The `OrderDate` element is two levels above the `Product` element; therefore, we use `../` twice, to indicate we should move two levels up the hierarchy. Also, note that because `OrderDate` is an element, not an attribute, we have not prefixed the node with an `@` symbol. Finally,

to map to the `CustomerName` element, we first must navigate two levels up the hierarchy, using the `../` syntax. We then must drop down into a sibling node (`OrderHeader`), to retrieve the `CustomerName` element.

The results of the query can be found in Figure 4-10. Note that `NULL` values have been returned for `OrderID` against two of the products. This is because no `OrderID` element was specified against the second sales order.


Results Messages

	ProductID	ProductName	Quantity	OrderID	OrderDate	CustomerName
1	45	Developer joke mug - there are 10 types of peop...	7	NULL	2013-01-02	Camille Authier
2	58	RC toy sedan car with remote control (Black) 1/...	4	NULL	2013-01-02	Camille Authier
3	22	DBA joke mug - it depends (White)	6	122	2013-01-02	Camille Authier
4	2	USB rocket launcher (Gray)	9	122	2013-01-02	Camille Authier
5	111	Superhero action jacket (Blue) M	10	122	2013-01-02	Camille Authier
6	116	Superhero action jacket (Blue) 4XL	4	122	2013-01-02	Camille Authier

Query executed successfully. DATATY

Figure 4-10. *Results of using `OPENXML()`*

Shredding XML with Nodes

If you want to shred the data from a column instead of using a variable, you can avoid the need for iterative logic by shredding the data with XQuery. Specifically, the `nodes()` method can be used to identify the nodes that should be mapped to relational columns. This can be combined with the `value()` method, to extract the data from the nodes. For example, consider the query in Listing 4-29. This query is functionally equivalent to the query in Listing 4-28 but uses the `nodes()` method instead of `OPENXML()`.

Listing 4-29. Using `nodes()` to Shred XML

```
DECLARE @SalesOrder XML ;

SET @SalesOrder = '
<SalesOrders>
  <Order>
    <OrderDate>2013-01-02</OrderDate>
    <OrderHeader>
      <CustomerName>Camille Authier</CustomerName>
        </OrderHeader>
    <OrderDetails>
      <Product ProductID="45" ProductName="Developer joke
      mug - there are 10 types of people in the world (Black)"
      Price="13" Qty="7" />
      <Product ProductID="58" ProductName="RC toy sedan car with
      remote control (Black) 1/50 scale" Price="25" Qty="4" />
    </OrderDetails>
  </Order>
  <Order>
    <OrderDate>2013-01-02</OrderDate>
    <OrderHeader>
      <CustomerName>Camille Authier</CustomerName>
        </OrderHeader>
    <OrderDetails OrderID = "122">
      <Product ProductID="22" ProductName="DBA joke mug - it
      depends (White)" Price="13" Qty="6" />
      <Product ProductID="2" ProductName="USB rocket launcher
      (Gray)" Price="25" Qty="9" />
      <Product ProductID="111" ProductName="Superhero action
      jacket (Blue) M" Price="30" Qty="10" />
```

```
      <Product ProductID="116" ProductName="Superhero action
jacket (Blue) 4XL" Price="34" Qty="4" />
    </OrderDetails>
  </Order>
</SalesOrders>' ;

SELECT
          TempCol.value('@ProductID', 'INT') AS ProductID
        , TempCol.value('@ProductName', 'NVARCHAR(70)') AS
        ProductName
        , TempCol.value('@Qty', 'INT') AS Quantity
        , TempCol.value('../@OrderID', 'INT') AS OrderID
        , TempCol.value('../../OrderDate[1]', 'NVARCHAR(10)')
        AS OorderDate
        , TempCol.value('../../OrderHeader[1]/CustomerName[1]',
        'NVARCHAR(15)') AS CustomerName
FROM @SalesOrder.nodes('SalesOrders/Order/OrderDetails/
Product') TempTable(TempCol) ;
```

When reviewing this script, you should pay particular attention to the `FROM` clause. Here, we pass an XQuery expression into the `nodes()` method, to define the grain. We then define arbitrary names for a table and column, which will contain the intermediate results set.

The `nodes()` method can also be used in conjunction with the `query()` method, to shred an XML document into smaller XML documents. For example, consider the query in Listing 4-30. This query extracts the `Product` elements from the XML document, in XML format.

Listing 4-30. Using the nodes() Method with the query() Method

```
DECLARE @SalesOrder XML ;

SET @SalesOrder = '
<SalesOrders>
  <Order>
    <OrderDate>2013-01-02</OrderDate>
    <OrderHeader>
      <CustomerName>Camille Authier</CustomerName>
        </OrderHeader>
    <OrderDetails>
      <Product ProductID="45" ProductName="Developer joke
      mug - there are 10 types of people in the world (Black)"
      Price="13" Qty="7" />
      <Product ProductID="58" ProductName="RC toy sedan car
      with remote control (Black) 1/50 scale" Price="25"
      Qty="4" />
    </OrderDetails>
  </Order>
  <Order>
    <OrderDate>2013-01-02</OrderDate>
    <OrderHeader>
      <CustomerName>Camille Authier</CustomerName>
        </OrderHeader>
    <OrderDetails OrderID = "122">
      <Product ProductID="22" ProductName="DBA joke mug - it
      depends (White)" Price="13" Qty="6" />
      <Product ProductID="2" ProductName="USB rocket launcher
      (Gray)" Price="25" Qty="9" />
      <Product ProductID="111" ProductName="Superhero action
      jacket (Blue) M" Price="30" Qty="10" />
```

```
      <Product ProductID="116" ProductName="Superhero action
      jacket (Blue) 4XL" Price="34" Qty="4" />
    </OrderDetails>
  </Order>
</SalesOrders>' ;

SELECT
          TempCol.query('.') AS Product
FROM @SalesOrder.nodes('SalesOrders/Order/OrderDetails/
Product') TempTable(TempCol) ;
```

The results of this query can be seen in Figure 4-11.

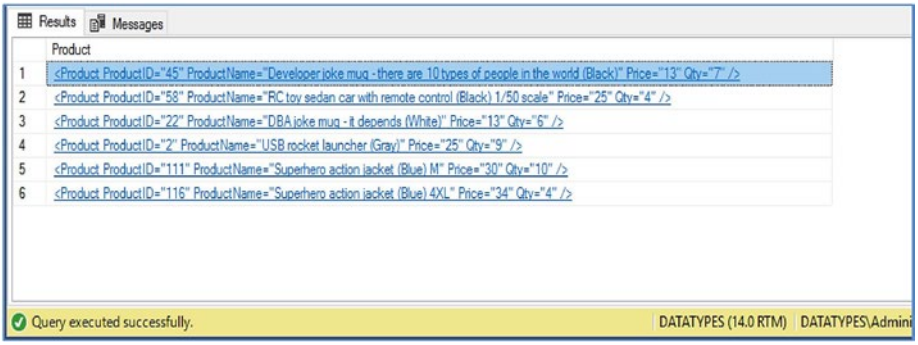


Figure 4-11. *Results of using the* `nodes()` *method with the* `query()` *method*

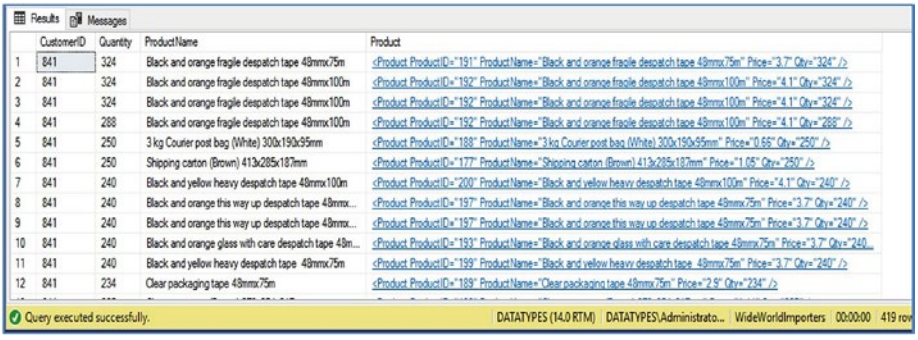
The biggest benefit of the `nodes()` method over `OPENXML()` is its ease of use against a table. The `CROSS APPLY` operator can be used to apply the `nodes()` method to multiple rows within a table. The query in Listing 4-31 demonstrates this, by calling the `nodes()` method against the OrderSummary column in the Sales.CustomerOrderSummary table. The query will return one row for every product on every sales order placed by a customer with the CustomerID of 814. The results are then ordered by the quantity of each product.

Listing 4-31. Using the nodes() Method Against a Table

```
USE WideWorldImporters
GO

SELECT
      CustomerID
        , TempCol.value('@Qty', 'INT') AS Quantity
        , TempCol.value('@ProductName', 'NVARCHAR(70)') AS
        ProductName
        , TempCol.query('.') AS Product
FROM Sales.CustomerOrderSummary
CROSS APPLY OrderSummary.nodes('SalesOrders/Order/OrderDetails/
Product') TempTable(TempCol)
WHERE CustomerID = 841
ORDER BY TempCol.value('@Qty', 'INT') DESC ;
```

Partial results of this query can be seen in Figure 4-12.



	CustomerID	Quantity	ProductName	Product
1	841	324	Black and orange fragile despatch tape 48mmx75m	<Product ProductID="191" ProductName="Black and orange fragile despatch tape 48mmx75m" Price="3.7" Qty="324" />
2	841	324	Black and orange fragile despatch tape 48mmx100m	<Product ProductID="192" ProductName="Black and orange fragile despatch tape 48mmx100m" Price="4.1" Qty="324" />
3	841	324	Black and orange fragile despatch tape 48mmx100m	<Product ProductID="192" ProductName="Black and orange fragile despatch tape 48mmx100m" Price="4.1" Qty="324" />
4	841	288	Black and orange fragile despatch tape 48mmx100m	<Product ProductID="192" ProductName="Black and orange fragile despatch tape 48mmx100m" Price="4.1" Qty="288" />
5	841	250	3 kg Courier post bag (White) 300x190x95mm	<Product ProductID="188" ProductName="3 kg Courier post bag (White) 300x190x95mm" Price="0.66" Qty="250" />
6	841	250	Shipping carton (Brown) 413x285x187mm	<Product ProductID="177" ProductName="Shipping carton (Brown) 413x285x187mm" Price="1.05" Qty="250" />
7	841	240	Black and yellow heavy despatch tape 48mmx100m	<Product ProductID="200" ProductName="Black and yellow heavy despatch tape 48mmx100m" Price="4.1" Qty="240" />
8	841	240	Black and orange this way up despatch tape 48mmx...	<Product ProductID="197" ProductName="Black and orange this way up despatch tape 48mmx75m" Price="3.7" Qty="240" />
9	841	240	Black and orange this way up despatch tape 48mmx...	<Product ProductID="197" ProductName="Black and orange this way up despatch tape 48mmx75m" Price="3.7" Qty="240" />
10	841	240	Black and orange glass with care despatch tape 48m...	<Product ProductID="193" ProductName="Black and orange glass with care despatch tape 48mmx75m" Price="3.7" Qty="240...
11	841	240	Black and yellow heavy despatch tape 48mmx75m	<Product ProductID="199" ProductName="Black and yellow heavy despatch tape 48mmx75m" Price="3.7" Qty="240" />
12	841	234	Clear packaging tape 48mmx75m	<Product ProductID="189" ProductName="Clear packaging tape 48mmx75m" Price="2.9" Qty="234" />

Figure 4-12. *Results of using the nodes() method against a table*

Using Schemas

As discussed in Chapter 3, an XML document can be bound to a schema, to ensure that its structure meets the client contract. In SQL Server, we can define an XSD schema by creating an `XML SCHEMA COLLECTION`. To demonstrate this, imagine that we wanted to bind our OrderSummary column, in the Sales.CustomerOrderSummary table, to a schema. The first step would be to create the `XSD SCHEMA COLLECTION`. We could achieve this using the code in Listing 4-32.

Listing 4-32. Creating an `XML SCHEMA COLLECTION`

```
USE WideWorldImporters
GO

CREATE XML SCHEMA COLLECTION OrderSummary AS
N'<?xml version="1.0" encoding="utf-16"?>
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified" xmlns:xs="http://www.
w3.org/2001/XMLSchema">
  <xs:element name="SalesOrders">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="Order">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="OrderHeader">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="CustomerName"
                    type="xs:string" />
                   <xs:element name="OrderDate" type="xs:date" />
```

```
                    <xs:element name="OrderID"
                    type="xs:unsignedInt" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="OrderDetails">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element maxOccurs="unbounded"
                    name="Product">
                      <xs:complexType>
                        <xs:attribute name="ProductID"
                        type="xs:unsignedByte" use="required" />
                        <xs:attribute name="ProductName"
                        type="xs:string" use="required" />
                        <xs:attribute name="Price"
                        type="xs:decimal" use="required" />
                        <xs:attribute name="Qty"
                        type="xs:unsignedShort" use="required" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>' ;
```

Tip Because the schema does not specify a namespace, it will be associated with the default empty string namespace. A namespace can be added to the schema, using the `<xs:schema>` attribute `xmlns:ns=http://your-namespace`.

We could bind our schema to the OrderSummary column by using the query in Listing 4-33.

Listing 4-33. Binding a Schema to a Column

```
ALTER TABLE Sales.CustomerOrderSummary
        ALTER COLUMN OrderSummary XML(OrderSummary) ;
```

We can also reference a schema when constructing or querying XML data. The `FOR XML` clause includes a `WITH XMLNAMESPACES` option that can be used to specify the target namespace of the resultant XML document, and XQuery methods, such as `query`, can begin with a declare namespace statement.

Tip A full discussion of the use of namespaces is beyond the scope of this book. However, examples of using `WITH XMLNAMESPACES` can be found at https://docs.microsoft.com/en-us/sql/t-sql/xml/with-xmlnamespaces?view=sql-server-2017, and examples of using a namespace with the `query` method can be found at https://docs.microsoft.com/en-us/sql/t-sql/xml/value-method-xml-data-type?view=sql-server-2017.

Summary

XQuery is a language that can be used to query XML data when it is stored in SQL Server columns and variables. The `exist()` method can be used to check for the existence of a node or a node containing a specific value. The `value()` method can be used to extract a scalar value from an XML document, and the `query()` method can be used to return a subset of an XML document still in XML format.

`FLWOR` statements can be used to help navigate and iterate an XML document. The `for` statement binds a variable to an input sequence. The `let` statement is used to assign an XQuery expression to a variable. The `where` statement can be used to filter the results that are returned. The `order by` statement can optionally be used to order the results of the `FLWOR` statement. The `return` statement specifies what data will be returned.

T-SQL variables and columns can be passed into XQuery statements. When this technique is adopted, it is known as a cross-domain query. It allows for values to easily be bound to XQuery statements, helping to simplify logic and reduce duplicate code.

XML data can be modified by using the `modify()` method. This method allows developers to use one of three options: `insert`, `replace value of`, or `delete`. When inserting data, there are further options that developers can take advantage of, to give them granular control over where the insert occurs. For example, you can choose to insert first, last, after a node, or before a node.

XML data can be converted into relational data (that is, shredded) by using either the `OPENXML()` function or the `nodes()` XQuery method. When the `OPENXML()` function is used, the XML document to be shredded must first be parsed. This can be achieved with the `sys.sp_xml:preparedocument` system-stored procedure. Once the document has been shredded, the parse tree should be removed from memory, by using the `sys.sp_xml:removedocument` system-stored procedure.

When the `nodes()` XQuery method is used to shred data, it can be used in conjunction with either the `value()` method, which will shred the data into relational results, or the `query()` method, which will shred the data into smaller XML documents. You can also use `nodes()` with a combination of both `value()` and `query()`. The biggest advantage of the `nodes()` method is the ease with which it can be applied to a column, using the `CROSS APPLY` operator.