SQL ServerCentral.com

Stairway

Stairway to XML

By Robert Sheldon

redgate

STAIRWAY TO XML

By ROBERT SHELDON

Published by SimpleTalk Publishing, 2017

First published on SQLServerCentral

ISBN: 978-1-910035-14-6

Cover Image: Andy Martin
Typeset: Gower Associates

Table of Contents

About the Author

After being dropped 35 feet from a helicopter and spending the next year recovering (http://preview.tinyurl.com/zexgcdu), Robert Sheldon left the Colorado Rockies and emergency rescue work to pursue safer and less painful interests—thus his entry into the world of technology. He is now a technical consultant and the author of numerous books, articles, and training material related to Microsoft Windows, various relational database management systems, and business intelligence design and implementation. He has also written news stories, feature articles, restaurant reviews, legal summaries, and the novels "Last Stand" and "Dancing the River Lightly." You can find more information at http://www.rhsheldon.com.

Introduction

Although XML is conceptually simple, its use as an equal partner datatype within a relational database, with full searching, validation and manipulation of data, is not intuitive. Now that the industry is more conscious of the use of semi-structured data and data defined by document markup, it is becoming more important than ever for Database Developers and DBAs to become conversant with the technology and appreciative of the cases where XML technologies enhance applications and their development. In this book, originally a series of articles on SQLServerCentral.com, Robert Sheldon flexes his talent to make the complicated seem simple.

Summary

Level 1 – Introduction to XML

An explanation of what XML is, and the components of an XML document, Elements and Attributes. The basics of tags, entity references, enclosed text, comments, and declarations.

Level 2 – The XML Data Type

SQL Server's XML Data Type, showing that it is as easy to configure a variable, column, or parameter with the **`XML`** data type as configuring one of these objects with any other data type.

Level 3 – Working with Typed XML

Enforce the validation of an **`XML`** data type, variable or column by associating it with an XML Schema Collection. SQL Server validates a typed XML value against the rules defined in the schema collection so that **`INSERT`** or **`UPDATE`** operations will succeed only if the value being inserted or updated is valid as per the rules defined in the Schema Collection.

Level 4 – Querying XML Data

Extract a subset of data from an XML instance by using the **`query()`** method, and you can use the **`value()`** method to retrieve individual element and attribute values from an XML instance.

Level 5 – The XML `exist()` and `nodes()` Methods

The XML **`exist()`** method is used, often in a **`WHERE`** clause, to check the existence of an element within an XML document or fragment. The **`nodes()`** method lets you shred an XML instance and return the information as relational data.

Level 6 – Inserting Data into an XML Instance

The modify method lets you manipulate XML data using XML DML. It can insert, alter or delete data. How to use the method to insert a node into an XML instance.

Level 7 – Updating Data in an XML Instance

Provide the necessary keywords and define the XQuery and value expressions in your XML DML expression in order to use the **`modify()`** method to update element and attribute values in either typed or untyped XML instances in an XML column.

Level 8 – Deleting Data from an XML Instance

In order to use the **`modify()`** method to delete data from typed and untyped XML instances, you must pass an XML DML expression as an argument to the method. That expression must include the delete keyword, along with an XQuery expression that defines the XML component to be deleted.

Level 9 – Creating XML-based Functions

How to use XML methods within user-defined functions to return XML fragments and values from your target XML instance.

Level 10 – Converting XML Data

How to convert string values to XML and how to convert XML to character types.

Level 1 – Introduction to XML

Support for the eXtensible Markup Language (XML) was first introduced in SQL Server with the release of SQL Server 2000. However, XML-related features were limited to data management capabilities that focused on mapping relational and XML data. For example, SQL Server 2000 added the **`FOR XML`** clause, which lets you return relational query results as XML.

However, it wasn't until the release of SQL Server 2005—when the **`XML`** data type was added—that support for XML got interesting. The **`XML`** data type lets you natively store XML documents in columns and variables configured with that type. The data type also supports a set of methods you can use to retrieve and modify specific components within the XML document.

To take full advantage of the XML-related features supported in SQL Server, you might find it useful to have a fundamental understanding of XML itself. To that end, this first Level of the book explains what XML is and describes the various components that make up an XML document.

An overview of XML

Similar to the HyperText Markup Language (HTML), XML is a markup language that uses tags to delineate and describe the nature of the data associated with those tags. What makes XML extensible is its self-describing nature; that is, you create tags that are specific to the data values contained in the XML document. In HTML, those tags are pre-defined. (XML's extensible nature will become clearer as we work through the XML components.)

Despite its extensibility, XML is still a standardized language that must conform to a specific set of formatting rules, as defined by the World Wide Web Consortium (W3C). Because of this standardization, the language has been widely adopted in order to transport and store data, unlike HTML, which is used to display data. XML makes it possible to easily share data among heterogeneous systems, regardless of hardware, operating system, or application type, and XML's universal adoption means that data can be processed with little human intervention. At the same time, you can control how the data is described, while also controlling how the data is ordered and displayed.

XML components

The primary components that make up an XML document—and the rules that govern the use of those components—are generally very straightforward, but you must adhere strictly to these rules in order for an XML document to be properly processed by the SQL Server XML parser.

There are primarily two types of information included in an XML document: the data to be stored and the tags that describe the data. A tag is made up of a set of angle brackets (**< >**) that enclose a descriptive word or compound word (no spaces) that describes the data associated with the tag. It's because of the self-describing nature of these tags that XML is often considered a *metalanguage*.

Each discrete piece of stored data is enclosed in an opening tag and a closing tag, as shown in Listing 1-1.

```
<Person>John Doe</Person>
```

Listing 1-1

In this case, the opening tag is **<Person>**, and the closing tag is **</Person>**. Notice that a forward slash precedes the tag description in the end tag. A forward slash must precede all end tags, but the language of the tag must be the same as the opening tag, which in the example above is **Person**. I could have chosen a name other than **Person**, including a name that has nothing to do with people, but a good practice is to always provide tag names that best describe the data enclosed in the opening and closing tags. In this case, the tags are describing the name of a person, John Doe, thus the tag name **<Person>**.

Together, the tags and enclosed data represent a single *element*. However, an element does not always have to contain data. An empty element can be rendered in one of two ways. The first is by specifying the opening and closing tags, but including no data, as in Listing 1-2.

```
<Person></Person>
```

Listing 1-2

Another way to represent an empty element is to use only one set of brackets, but still include the forward slash (Listing 1-3).

```
<Person />
```

Listing 1-3

Again, this method can be used only when an element contains no value. As you'll see later in the book, a schema might require an element for which there is no value. In that case, you can use the shortened format to represent the both tags of the element.

Whether or not an element contains a value, whenever both tags are used, the opening and closing tags must match exactly, down to the capitalization (except for the forward slash in the closing tag). For instance, the element in Listing 1-4 generates an error in the SQL Server XML parser because the case is different between the two tags.

```
<person>John Doe</Person>
```

Listing 1-4

The descriptive word in the opening tag is all lowercase; however, the descriptive word in the closing tag begins with a capital letter. The opening and closing tags must match to be considered proper, or well-formed, XML.

You can, however, embed elements within each other. In Listing 1-5, I embed two instances of the **<Person>** element within the **<People>** element.

```
<People>
<Person>John Doe</Person>
<Person>Jane Doe</Person>
</People>
```

Listing 1-5

Notice that each **<Person>** element is complete in itself. It includes the opening and closing tags and the data they enclose. Elements embedded in other elements are referred to as *child* elements or, in some cases, *subelements*. The outer element, in this case, **<People>**, is the *parent* element. The parent element at the highest Level of an XML document is considered the *root* element. All XML documents must have one, and only one, root element. So the **<People>** element in the example above is the parent element to the two **<Person>** elements, and it is the root element for the XML document.

SQL Server also permits you to store XML *fragments* in an **XML** column or variable.

A fragment is a chunk of XML code without a root element, such as the two elements shown in Listing 1-6.

```
<Person>John Doe</Person>
<Place>Seattle, WA</Place>
```

Listing 1-6

The elements must still be well-formed XML, that is, have matching tags that enclose the data, but they don't have to be an XML document. As you'll see later in the book, you can specify that only XML documents be permitted in an **XML** column or variable but, for now, just know that SQL Server distinguishes between XML documents and fragments and can store both.

When you embed elements within other elements, you must ensure that the child elements are complete before you end the parent element. For instance, in Listing 1-7, I end the **<People>** element before the **<Person>** element, which causes the SQL Server XML parser to generate an error.

```
<People><Person>John Doe</People></Person>
```

Listing 1-7

You must ensure that your child elements are complete no matter how many levels contain embedded elements. In the following example, the **<FirstName>** and **<LastName>** elements are embedded in each **<Person>** element, and the **<Person>** elements are embedded in the **<People>** element.

```
<People>
<Person>
<FirstName>John</FirstName>
<LastName>Doe</LastName>
</Person>
<Person>
<FirstName>Jane</FirstName>
<LastName>Doe</LastName>
</Person>
</People>
```

Listing 1-8

In this case, the **<Person>** elements act as both child and parent elements. Notice, however, that each embedded element, regardless of the level, falls completely within the opening and closing tags of the parent element. For example, the first instances of the **<FirstName>** and **<LastName>** elements fall completely within the first instance of the **<Person>** element, and the two instances of the **<Person>** elements fall completely within the **<People>** element, which is the document's root element.

Elements can also have attributes associated with them. An attribute is a property to which you can assign a value. The attribute is defined as part of the element's opening tag. In Listing 1-9, I've added the **id** attribute to each instance of the **<Person>** element.

```
<People>
<Person id="1234">
<FirstName>John</FirstName>
<LastName>Doe</LastName>
</Person>
<Person id="5678">
<FirstName>Jane</FirstName>
<LastName>Doe</LastName>
</Person>
</People>
```

Listing 1-9

As the example demonstrates, an attribute consists of the attribute name (in this case, **id**), followed by an equal sign and the attribute value, enclosed in double quotes. So the **id** attribute for the first instance of the **<Person>** element has a value of **1234**, and the **id** attribute for the second instance of the **<Person>** element has a value of **5678**.

Another component contained in many XML documents is the *declaration*, which at a minimum specifies the version of the XML standard that the document conforms to. To date, there are only two versions: 1.0 and 1.1. If using XML 1.0, the declaration is not necessary; however, XML 1.1 requires one. For that reason, you should be aware of how to include a declaration in your XML document.

If you include a declaration, you must place it at the beginning of the document, start the declaration with the **<?** opening tag, and end it with the **?>** closing tag. In addition, you must include the **xml** keyword (lowercase) and the **version** attribute (also lowercase). Another attribute commonly included, although optional, is **encoding**, which specifies the character encoding used for the XML document.

In Listing 1-10, I include a declaration that specifies version 1.0 and an encoding of UTF-8, which means the data is stored as a sequence of 8-bit Unicode characters.

```
<?xml version="1.0" encoding="UTF-8"?>
<People>
<Person id="1234">
<FirstName>John</FirstName>
<LastName>Doe</LastName>
</Person>
<Person id="5678">
<FirstName>Jane</FirstName>
<LastName>Doe</LastName>
</Person>
</People>
```

Listing 1-10

You can also add comments to your XML documents. To do so, simply precede the comment with the **<!--** tag and end it with the **-->** tag, as I've done in Listing 1-11.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A list of current clients -->
<People>
<Person id="1234">
<FirstName>John</FirstName>
<LastName>Doe</LastName>
</Person>
<Person id="5678">
<FirstName>Jane</FirstName>
<LastName>Doe</LastName>
</Person>
</People>
```

Listing 1-11

As you can see, I've added the comment, **`A list of current clients`**, which I've enclosed in the comment tags. The SQL Server XML parser will ignore anything within the tags, so you can use the commenting feature not only to provide information about the XML document and its data, but also to preserve parts of the XML code that you want to hang on to but you don't want to have processed as part of the document.

Another consideration when working with XML is that certain characters cannot be parsed when they appear in element values. For example, you cannot include an ampersand (&) in an element's value, as I've done in the **<FavoriteBook>** child element in Listing 1-12.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A list of current clients -->
<People>
<Person id="1234">
<FirstName>John</FirstName>
<LastName>Doe</LastName>
</Person>
<Person id="5678">
<FirstName>Jane</FirstName>
<LastName>Doe</LastName>
<FavoriteBook>Crime & Punishment</FavoriteBook>
</Person>
</People>
```

Listing 1-12

If I try to assign this XML document to an **XML** column or variable, the **<FavoriteBook>** child element will cause the parser to generate an error because the value **Crime & Punishment** includes the ampersand. You must replace this type of character with an *entity reference* that tells the parser to preserve the character as it is originally intended. An entity reference begins with an ampersand and ends with a semicolon and in between includes a multi-character code that represents the original value. For an ampersand, the entity reference should be **&**, which I use in Listing 1-13.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A list of current clients -->
<People>
<Person id="1234">
<FirstName>John</FirstName>
<LastName>Doe</LastName>
</Person>
<Person id="5678">
<FirstName>Jane</FirstName>
<LastName>Doe</LastName>
<FavoriteBook>Crime &amp; Punishment</FavoriteBook>
```

```
</Person>
</People>
```

Listing 1-13

Notice that I've replaced the ampersand with the **`&`** entity reference. Now the XML parser will handle the **`<FavoriteBook>`** element with no problem. But note that the ampersand is not the only character that will generate an error. The XML standard identifies five characters that should be replaced with entity references, as I've done in Listing 1-13.

- **Less than** (**<**): replace with **`&lt`**
- **Greater than** (**>**): replace with **`&gt`**
- **Ampersand** (**&**): replace with **`&amp`**
- **Apostrophe** (**'**): replace with **`&apos`**
- **Quotation mark** (**"**): replace with **`&quot`**

Another issue that the example raises is the fact that the child elements do not have to be the same from one parent instance to the next. As you can see, the first instance of the **`<Person>`** element contains only the **`<FirstName>`** and **`<LastName>`** child elements, but the second instance of the **`<Person>`** element contains the **`<FirstName>`** and **`<LastName>`** child elements, as well as the **`<FavoriteBook>`** element. As long as your child elements are well formed, you can include whatever elements necessary to delineate and define your data.

Conclusion

In this chapter, we've looked at the primary components that make up an XML document. Elements serve as the basic building blocks for all XML documents, with each element being delineated by an opening tag and a closing tag and the data value itself being enclosed between those two tags. Elements can be embedded within each other, but one element—the root—must act as the parent to all other elements in an XML document. An element can also include attributes, which are defined as part of an element's opening tag.

As handy as it might be to know how to put together an XML document, the purpose of this Level has not been to train you in how to create these types of documents, but rather to provide an introduction to XML so you can more effectively work with XML in SQL Server. In the next Level, we'll look at how the **`XML`** data type is implemented in SQL Server and how it can be assigned to columns and variables in order to store both XML documents and XML fragments.

Level 2 – The XML Data Type

At the heart of SQL Server's support for XML lies the **XML** data type, which lets you store XML data in database objects such as variables, columns, and parameters. When you configure one of these objects with the **XML** data type, you simply specify the type name as you would any other SQL Server type. What sets the **XML** data type apart from the other types are a number of features that affect the way you can store, query, modify, and index XML data—all concepts we'll be covering as we progress through this book.

The **XML** data type ensures that your XML data is *well formed,* that is, conforms to ISO standards. You can use the data type to store either XML documents or fragments. As you'll recall from Level 1, an XML document is one that has a single top-level, or root, element. Without a single root element, the XML is considered a fragment.

Before you define an object with the **XML** data type, you should be aware that it has several limitations, including the following:

- An instance of an **XML** column cannot exceed 2 GB.
- An **XML** column cannot be an index key.
- You cannot use an **XML** object in a **GROUP BY** clause.
- You cannot compare or sort data that uses the **XML** type.

Even if you can work within these limitations, there are times when using the **XML** data type isn't necessary. For example, if you're simply storing your XML documents in their entirety and don't plan to query or modify individual XML components, you should consider using large object storage instead, such as **VARCHAR(MAX)** . That way, you're not invoking the XML parser, which helps to minimize the overhead necessary to store your XML documents. However, if you want to take advantage of the capabilities specific to the **XML** data type, then by all means, use it when defining your variables, columns, and parameters.

Defining an XML variable

As mentioned above, to define a database object with the **XML** data type, you specify the type as you would any other SQL Server type. In the case of variables, simply provide the variable name, followed by the **XML** type, as shown in the Transact-SQL code in Listing 2-1.

```
DECLARE @ClientList XML
SET @ClientList =
'<?xml version="1.0" encoding="UTF-8"?>
<!-- A list of current clients -->
<People>
<Person id="1234">
<FirstName>John</FirstName>
<LastName>Doe</LastName>
</Person>
<Person id="5678">
<FirstName>Jane</FirstName>
<LastName>Doe</LastName>
</Person>
</People>'
SELECT @ClientList
GO
```

Listing 2-1

The example begins by using a **DECLARE** statement to define the **@ClientList** variable. When I declare the variable, I simply include the **XML** data type name after the variable name.

I then use the **SET** statement to set the value of the variable to equal a small but well-formed XML document. The first line of the document is the declaration, which specifies the XML version and encoding. The next line is a comment about the nature of the content in the XML document. The SQL Server XML parser essentially ignores any commented information. The rest of the document is the actual XML, which contains the root element **<People>** and two instances of the **<Person>** child element.

> ***Note***
>
> *The components that make up an XML document are described in Level 1 of the book. If you're not familiar with these XML components and have not yet reviewed the first Level, you might benefit from reading that one first, before continuing with this Level.*

After I set the variable's value, I use a **SELECT** statement to retrieve the value. As you would expect, the statement returns the XML document, as shown in the results in Listing 2-2.

```
<!-- A list of current clients -->
<People>
<Person id="1234">
<FirstName>John</FirstName>
<LastName>Doe</LastName>
</Person>
<Person id="5678">
<FirstName>Jane</FirstName>
<LastName>Doe</LastName>
</Person>
</People>
```

Listing 2-2

That's all there is to defining an **XML** variable. Now let's look at how to define an **XML** column.

Defining an XML column

There are no real magic tricks when it comes to creating a database object configured with the **XML** data type. If you can define an **XML** variable, you can just as easily define an **XML** column. In Listing 2-3, I create the **StoreClients** table, which stores an ID and list of client information for each company store:

```
USE AdventureWorks2008R2
GO
IF OBJECT_ID('dbo.StoreClients') IS NOT NULL
DROP TABLE dbo.StoreClients
GO
CREATE TABLE dbo.StoreClients
(
StoreID INT IDENTITY PRIMARY KEY,
ClientInfo XML NOT NULL
)
GO
```

Listing 2-3

The ID, of course, is stored in the **StoreID** column, which is configured with the **INT** data type. The client information is stored in an **XML** column, in this case, **ClientInfo**. Because I use an **XML** column, I can take advantage of the extensible nature of **XML** and store client-related data that does not conform easily to a relational model, while still associating that data with a specific store, as identified in the **StoreID** column.

When I define the **ClientInfo** column, I simply specify the **XML** data type as I would any other type. In fact, at its most basic level, the column is treated just like any other column. As a result, I can include an **XML** column in a table along with other column types. Each XML document or fragment stored in the **XML** column is treated as an individual value, just like the values in any other columns, thus preserving the table's atomic nature. That way, I can build a table with any assortment of columns yet include one or more **XML** columns.

After I create the **StoreClients** table, I can insert data into the table, including XML documents and fragments. In Listing 2-4, I declare an **XML** variable and use that variable in an **INSERT** statement that adds a row of data into the table.

```
DECLARE @ClientList XML
SET @ClientList =
'<?xml version="1.0" encoding="UTF-8"?>
<!-- A list of current clients -->
<People>
<Person id="1234">
<FirstName>John</FirstName>
<LastName>Doe</LastName>
</Person>
<Person id="5678">
<FirstName>Jane</FirstName>
<LastName>Doe</LastName>
</Person>
</People>'
INSERT INTO dbo.StoreClients (ClientInfo)
VALUES(@ClientList)
GO
```

Listing 2-4

In this case, the **INSERT** statement adds the XML data to the table's **ClientInfo** column. Although the **@ClientList** variable holds an entire XML document, the **INSERT** statement treats the document as a single value when adding that document to the **ClientInfo** column. (The value in the **StoreID** column is generated automatically because it's an **IDENTITY** column.)

If you were to query the table after running the **INSERT** statement, you would receive the **StoreID** value (a **1** if this was the first row inserted into the table) and the full XML document from the **ClientInfo** column.

As you can see, creating an **XML** column and inserting data into that column is a fairly straightforward process. There are, of course, ways to make this process far more complicated, which you'll learn how to do as we work through the book. Until then, know that, at its most basic, working with the **XML** data type is for the most a painless endeavor. So let's look at how to create an **XML** parameter.

Defining an XML parameter

When creating a stored procedure in SQL Server, you might want to pass data into the procedure when you call it or have the procedure return data after it executes. You can do this by including parameters in your procedure definition. Not surprisingly, you can configure those parameters with the **XML** data type, as you can variables and columns.

For example, in the **CREATE PROCEDURE** statement in Listing 2-5, I define the **@StoreClients** input parameter, which is configured with the **XML** data type.

```
USE AdventureWorks2008R2
GO
IF OBJECT_ID('dbo.AddClientInfo', 'P') IS NOT NULL
DROP PROCEDURE dbo.AddClientInfo
GO
CREATE PROCEDURE dbo.AddClientInfo
@StoreClients XML
AS
INSERT INTO dbo.StoreClients (ClientInfo)
VALUES(@StoreClients)
GO
```

Listing 2-5

As you can see, my parameter definition follows the **CREATE PROCEDURE** clause. I specify that the parameter be defined with the **XML** data type by including the data type name after the parameter name, just like I do for columns and variables. Notice that the **INSERT** statement contained within the procedure definition uses the parameter in its **VALUES** clause.

That means, when you run the **AddClientInfo** stored procedure, you must specify a value for the **@StoreClients** input parameter, and that value must be an **XML** document or fragment so it can be inserted into the **ClientInfo** column.

For instance, in Listing 2-6, I declare the **@ClientList** variable, assign an XML document to the variable, and then use the variable to provide a value to the **@StoreClients** input parameter when I call the **AddClientInfo** stored procedure.

```
DECLARE @ClientList XML
SET @ClientList =
'<?xml version="1.0" encoding="UTF-8"?>
<!-- A list of current clients -->
<People>
<Person id="1234">
<FirstName>John</FirstName>
<LastName>Doe</LastName>
</Person>
<Person id="5678">
<FirstName>Jane</FirstName>
<LastName>Doe</LastName>
</Person>
</People>'
EXEC dbo.AddClientInfo @ClientList
```

Listing 2-6

As in the previous examples, the **@ClientList** variable is defined with the **XML** data type and set to equal the **<People>** **XML** document. Because the **@StoreClients** parameter is also configured with the **XML** data type, I can pass in the **@ClientList** value into the parameter. As a result, when I run the stored procedure, the XML document is added to the **StoreClients** table. If I were to then query the table, I would find that a second row has been added, with a **StoreID** value of **2** and a **ClientInfo** value equal to the XML document I passed into the **@ClientList** variable.

Conclusion

As you've seen in this Level, configuring a variable, column, or parameter with the **`XML`** data type is as easy as configuring one of these objects with any other type. However, as you work through subsequent Levels in the book, you'll find that there are additional options available to you when defining an **`XML`** object. For instance, you can associate a schema with an **`XML`** object to ensure that your XML documents and fragments conform to a more-rigidly defined format. In fact, you'll learn how to do just that in the next Level. So stand by. More fun is coming your way.

Level 3 – Working with Typed XML

In Level 2, you were introduced to the **XML** data type and shown how to use it to create columns, variables, and parameters that can store XML data. As you saw, you simply specify the **XML** data type when you define the object, as you would any other type. However, the examples in that Level told only part of the story. SQL Server actually supports two kinds of **XML** objects: *typed* and *untyped*.

What distinguishes the two is whether the **XML** column, variable, or parameter is associated with a specific *schema collection,* a database entity (like a table or stored procedure) that specifies the structure and data types that an XML document must adhere to. If a database object is associated with a collection, it is considered typed, otherwise it is untyped. The **XML** columns, variables, and parameters defined in the Level 2 examples are all untyped **XML** objects because no schema collections are associated with them. In this level, you'll learn how to work with schema collections and how to create typed database objects.

The XML schema collection

An XML schema collection is made up of one or more XML Schema Definition (XSD) schemas that are used to validate XML data stored in a typed **XML** object. The XSD schemas contain the actual formatting information that defines the structure and data types an XML instance must use when saved to a typed **XML** object. As a result, before you can associate an XML schema collection with an **XML** object, the collection must exist as an entity within the database. In other words, you must specifically create the collection before you can reference it.

To create a schema collection, you use the **CREATE XML SCHEMA COLLECTION** statement to specify a collection name and define at least one XSD schema. In the T-SQL code shown in Listing 3-1, I create the **ClientDB** database and then add the **ClientInfoCollection** XML schema collection to the database. The collection includes a single XSD schema (enclosed in single quotes after the **AS** keyword).

```
USE master
GO

IF DB_ID('ClientDB') IS NOT NULL
DROP DATABASE ClientDB
GO

CREATE DATABASE ClientDB
GO

USE ClientDB
GO

CREATE XML SCHEMA COLLECTION ClientInfoCollection AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="urn:ClientInfoNamespace"
targetNamespace="urn:ClientInfoNamespace"
elementFormDefault="qualified">
  <xsd:element name="People">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Person" minOccurs="1"
maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="FirstName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="LastName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="FavoriteBook" type="xsd:string"
minOccurs="0" maxOccurs="5" />
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:integer"
use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 3-1: Creating an XML schema collection in the **`ClientDB`** database.

Before I try to explain what's going on here, let me qualify my remarks. The rules governing XSD schemas are quite complex. I can provide an overview of what I've done here, but a full explanation of the XSD syntax is beyond the scope of this Level—or even this book. However, you can find detailed information about the various schema elements at www.w3schools.com/xml/schema_intro.asp or, preferably, read Jacob Sebastian's book, "The Art of XSD."

In the meantime, let's look at the **`CREATE XML SCHEMA COLLECTION`** statement in more detail. As you can see in Listing 3-1, I provide a name for the collection (**`ClientInfoCollection`**) after the **`COLLECTION`** keyword.

I follow the name with the **`AS`** keyword and an XSD schema definition, enclosed in single quotes. The definition begins with the **`<xsd:schema>`** element, which provides the details necessary to establish this code as an XSD schema. In this case, I've included four attributes in the **`<xsd:schema>`** element, which are defined in Table 3-1.

Attribute	Description
`xmlns` (first instance)	Specifies the source namespace (http://www.w3.org/2001/XMLSchema) for the elements and data types used in the schema. The **`:xsd`** that follows the **`xmlns`** attribute name is an alias used throughout the schema to reference the source namespace.
`xmlns` (second instance)	Specifies the schema's default namespace (**`urn:ClientInfoNamespace`**). This is the namespace that XML instances must reference when stored in a typed XML object. The **`urn:`** that precedes the actual namespace name refers to *Uniform Resource Name* and is used to denote this as a location-independent resource identifier, as opposed to a Uniform Resource Locator (URL), such as the one specified in the first instance of the **`xmlns`** attribute.

targetNamespace	Specifies the namespace (**urn:ClientInfoNamespace**) associated with the elements defined in the schema. This namespace corresponds to the one specified in the second instance of the **xmlns** attribute.
elementFormDefault	Specifies that all elements in the XML instance associated with the schema must be qualified with the default namespace, either explicitly or implicitly.

Table 3-1: The attributes defined in the **<xsd:schema>** element.

After I specify the opening **<xsd:schema>** tag, I define the actual structure that will be applied to the XML instances stored in the typed **XML** objects. I begin by using an **<xsd:element>** element to define the XML document root, which is called **People**, as determined by the **name** attribute. Notice that both the **<xsd:schema>** element and the **<xsd:element>** element are preceded by the **xsd:** alias, which refers to the first namespace defined in **<xsd:schema>**. All element definitions in the schema must include the **xsd:** preface.

Much of the remaining XSD structure is devoted to delineating the child elements and attributes that make up the root element. In fact, you can find the root's closing tag, **</xsd:element>**, on the second-to-last line of the XSD definition. Everything between and including the opening and closing tags defines the structure that each XML instance in an **XML** object must conform to.

The next element in the XSD schema is **<xsd:complexType>**, which indicates that the current element can contain other elements or attributes. When you include the **<xsd:complexType>** element in an element that will include child elements, you must also include an indicator element that determines the order of the child elements and, in some cases, the number of those elements. In this situation, I'm using the **<xsd:sequence>** indicator element to specify that child elements must appear in a specific order. Although order isn't a factor in this particularly situation, the **<xsd:sequence>** indicator is the most appropriate option of the available indicators.

After the **<xsd:sequence>** element, I add a child element named **Person**, as indicated by the **name** attribute. Notice that the element also includes the **minOccurs** attribute, which indicates that the XML instance must include at least one **<Person>** element, and the **maxOccurs** attribute, which has a value of **unbounded**, indicating that there is no limit to the number of **<Person>** elements that the XML instance can contain.

Because the **<Person>** element can itself include child elements, it, too, is followed by the **<xsd:complexType>** element and the **<xsd:sequence>** indicator element. The first of the child elements is called **FirstName**. The type attribute, which indicates the data type of the element value, is set to **xsd:string**. The **minOccurs** attribute indicates that at least one instance of the element is required, and the **maxOccurs** attribute indicates that no more than one instance of the element is permitted.

The second of the child elements is named **LastName**, and it is configured like the **<FirstName>** element. Note, however, that because the **<xsd:sequence>** element is included, the XML instance must specify these elements in their defined order.

The last of the **<Person>** child elements is the one named **FavoriteBook**. Notice that this element requires no minimum instances but limits the number of maximum instances to five.

There's one more item in the **<Person>** element worth noting—an attribute named **id**. The attribute is defined with the **xsd:int** data type and is required with each **<Person>** element in the XML instance, as indicated by the **use** attribute, which is set to **required**.

That about covers this schema. As you can see, it's structured like an XML document, with **<xsd:schema>** as the root element. Again, I've provided only a high-Level overview of what makes up an XSD schema. And the example I've used is a very simply one. If you want to see more complex schemas, check out the **AdventureWorks** sample database. It includes several examples. Now let's look at what you can do with a schema.

Typed XML

Once you've added your schema collection to the database, you can create typed **XML** columns, variables, and parameters. (You can also breathe a sigh of relief, knowing that the hardest part is behind you.)

When defining a typed **XML** object, you specify the **XML** data type as you do for untyped objects, but you also add the name of the schema collection, enclosed in parentheses. For example, in Listing 3-2, I declare the **@xml** variable, assign the **XML** data type to the variable, and specify the **ClientInfoCollection** schema collection (created in Listing 3-1).

```
DECLARE @xml XML(ClientInfoCollection)
SET @xml =
  '<?xml version="1.0" encoding="UTF-8"?>
  <!-- A list of current clients -->
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
      <FavoriteBook>Crime &amp; Punishment</FavoriteBook>
    </Person>
  </People>'
SELECT @xml
```

Listing 3-2: Defining a typed **XML** variable based on the **ClientInfoCollection** schema collection.

As you can see, to create a typed **XML** object, you need only add the schema collection name. Where things can get tricky is when you insert data into that object. Once you've typed an **XML** object, your XML instance must conform to the format defined by the XSD schema included in the specified collection. In addition, you must ensure that your XML instance references the default namespace specified in the schema.

If you refer back to Listing 3-2, you'll see that I include the **xmlns** attribute in the **<People>** element, the XML document's root element. The **xmlns** attribute refers specifically to the default namespace defined in the XSD schema. Now any elements I include in my XML instance will apply to that namespace.

Because I've created a typed variable, I must also ensure that my XML document conforms to the format defined in the XSD schema. That means, for example, my **<People>** element must contain at least one instance of the **<Person>** child element, and each instance of the **<Person>** element must contain one instance of the **<FirstName>** element, one instance of the **<LastName>** element, and zero to five instances of the **<FavoriteBook>** element. In addition, each **<Person>** element must include an **id** attribute.

When I executed the statements in Listing 3-2, I was able to successfully assign my XML instance to the **@xml** variable, which means that the XML instance had been properly formatted, as evidenced by the value returned by the **SELECT** statement (shown in Listing 3-3).

```
<!-- A list of current clients -->
<People xmlns="urn:ClientInfoNamespace">
  <Person id="1234">
    <FirstName>John</FirstName>
    <LastName>Doe</LastName>
  </Person>
  <Person id="5678">
    <FirstName>Jane</FirstName>
    <LastName>Doe</LastName>
    <FavoriteBook>Crime &amp; Punishment</FavoriteBook>
  </Person>
</People>
```

Listing 3-3: The XML instance stored in the `@xml` variable.

As you would expect, the XML instance is returned in its entirety, including the **xmlns** attribute and its reference to the XSD default namespace.

A typed **XML** object ensures that your XML instance is correctly formatted. If it's not, SQL Server will return an error. For example, the code shown in Listing 3-4 declares a typed **XML** variable, but this time tries to assign an improperly formatted XML document to the variable.

```
DECLARE @xml XML(ClientInfoCollection)
SET @xml =
  '<?xml version="1.0" encoding="UTF-8"?>
  <!-- A list of current clients -->
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="1234">
      <FirstName>John</FirstName>
      <MiddleInit>T</MiddleInit>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
      <FavoriteBook>Crime &amp; Punishment</FavoriteBook>
    </Person>
  </People>'
```

Listing 3-4: Assigning an improperly formatted XML instance to a typed **XML** variable.

In this case, the first instance of the **<Person>** element includes the **<MiddleInit>** child element, which is not specified in the XSD schema. As a result, when I try to run the **SET** statement, SQL Server returns the error shown in Listing 3-5.

```
Msg 6965, Level 16, State 1, Line 2
XML Validation: Invalid content.
Expected element(s): '{urn:ClientInfoNamespace}LastName'.
Found:element '{urn:ClientInfoNamespace}MiddleInit' instead.
Location: /*:People[1]/*:Person[1]/*:MiddleInit[1].
```

Listing 3-5: Error returned as a result of an improper element.

As the error indicates, the XML parser expected the **<LastName>** element to follow the **<FirstName>** element, but instead found **<MiddleInit>**, which of course doesn't belong there.

But non-defined elements are not the only problem. Because the XSD schema defines the **<Person>** element as a complex type and qualifies the type with the **<xsd:sequence>** element, introducing the child elements in the wrong order also causes the parser to generate an error. For example, in Listing 3-6, I switch the **<FirstName>** and **<LastName>** elements in the first instance of the **<Person>** element.

```
DECLARE @xml XML(ClientInfoCollection)
SET @xml =
'<?xml version="1.0" encoding="UTF-8"?>
  <!-- A list of current clients -->
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="1234">
      <LastName>Doe</LastName>
      <FirstName>John</FirstName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
      <FavoriteBook>Crime &amp; Punishment</FavoriteBook>
    </Person>
  </People>'
SELECT @xml
```

Listing 3-6: Listing elements in the wrong order in an XML instance.

Again, the XML parser chokes when it doesn't get what it expects. When the order is changed, SQL Server returns the error shown in Listing 3-7. The XML parser expects the **<FirstName>** element but instead gets the **<LastName>** element, so an error is returned.

```
Msg 6965, Level 16, State 1, Line 2
XML Validation: Invalid content.
Expected element(s):'{urn:ClientInfoNamespace}FirstName'.
Found: element '{urn:ClientInfoNamespace}LastName' instead.
Location: /*:People[1]/*:Person[1]/*:LastName[1].
```

Listing 3-7: Error returned as a result of specifying elements in an incorrect order.

Although storing data as typed XML might seem like more trouble than it's worth, it does in fact help to optimize queries, data storage, and data modification. So if you have an XSD schema available, it's worth adding it to a collection and implementing typed **XML** objects.

XML documents and fragments

There's one other aspect of working with typed XML that's worth mentioning. SQL Server distinguishes between XML *documents* and XML *fragments*. A document is an XML instance that has one root element, as you saw in the examples above. An XML fragment does not have this restriction. The XML must still be well formed, but it can have multiple root elements.

By default, SQL Server lets you store both documents and fragments in an **XML** object. However, if the object is typed, you can specify that it store only documents.

The **XML** data type provides two options that let you define how to store fragments and documents. When you specify the name of the schema collection, you can also specify either the **CONTENT** or **DOCUMENT** option. **CONTENT** is the default, and therefore permits either fragments or documents. If you want, you can include **CONTENT** when defining your **XML** object. For example, in Listing 3-8, I added the **CONTENT** keyword when declaring my variable.

```
DECLARE @xml XML(CONTENT ClientInfoCollection)
SET @xml =
  '<?xml version="1.0" encoding="UTF-8"?>
  <!-- A list of current clients -->
  <People xmlns="urn:ClientInfoNamespace">
```

```
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
      <FavoriteBook>Crime &amp; Punishment</FavoriteBook>
    </Person>
  </People>'
SELECT @xml
```

Listing 3-8: Accepting XML documents or fragments in an XML object.

Notice that the addition of the **CONTENT** keyword is the only thing that changed. The XML instance will be saved to the variable as in previous examples, and the **SELECT** statement will return the full XML instance, as expected. In this case, that XML is considered a document because it has only one root element. However, I could just as easily have saved a well-formed fragment to the variable.

If I want to limit my XML instances to documents only, I can instead specify the **DOCUMENT** option, as shown in Listing 3-9.

```
DECLARE @xml XML(DOCUMENT ClientInfoCollection)
SET @xml =
  '<?xml version="1.0" encoding="UTF-8"?>
  <!-- A list of current clients -->
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
      <FavoriteBook>Crime &amp; Punishment</FavoriteBook>
    </Person>
  </People>'
SELECT @xml
```

Listing 3-9: Limiting an **XML** object to XML documents.

Again, making this change presents no problems because the XML instance is a well-formed document. But suppose I try to pass in an XML fragment when I specify the **DOCUMENT** option, as I do in Listing 3-10.

```
DECLARE @xml XML(DOCUMENT ClientInfoCollection)
SET @xml =
  '<?xml version="1.0" encoding="UTF-8"?>
  <!-- A list of current clients -->
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
      <FavoriteBook>Crime &amp; Punishment</FavoriteBook>
    </Person>
  </People>'
SELECT @xml
```

Listing 3-10: Passing an XML fragment into an **XML** object configured for documents.

Notice that my XML instance is now a fragment that includes two instances of the **<People>** element at the root. Not surprisingly, the XML parser doesn't like such behavior. When I run the **SET** statement, SQL Server returns the error shown in Listing 3-11.

```
Msg 6901, Level 16, State 1, Line 2
XML Validation: XML instance must be a document.
```

Listing 3-11: Error generated by SQL Server as a result of the XML fragment.

Of course, this is easily fixed by changing the **DOCUMENT** option to **CONTENT**, as I do in Listing 3-12. I still pass in an XML fragment, but now the XML parser happily accepts it.

```
DECLARE @xml XML(CONTENT ClientInfoCollection)
SET @xml =
  '<?xml version="1.0" encoding="UTF-8"?>
  <!-- A list of current clients -->
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
      <FavoriteBook>Crime &amp; Punishment</FavoriteBook>
    </Person>
  </People>'
SELECT @xml
```

Listing 3-12: Passing an XML fragment into an **XML** object configured for fragments or documents.

As expected, the **SELECT** statement returns the XML fragment in its entirety. I could just as easily have added more instances of the **<People>** element and the XML parser would have accepted it, as long as the fragment was still well formed and adhered to the formatting structure defined in the XSD schema.

Conclusion

Working with typed **XML** objects—and their associated schema collections—gets to be a bit more complicated than working with simple untyped **XML** objects. If you plan to save data as typed XML, you should have a basic understanding of how to implement and work with XSD schemas. However, creating the XML objects themselves is still a very basic process, whether those objects are typed or untyped. In the next Level, you'll learn how to incorporate **XML** objects in other database entities, such as views, functions, and computed columns. Until then, you might want to dig deeper into XSD schemas. There's a lot to learn there, and the better you understand them, the more effectively you'll be able to store typed XML.

Level 4 – Querying XML Data

In Levels 2 and 3, you learned how to use the **XML** data type when defining columns, variables, and parameters. You also learned the difference between typed and untyped **XML** objects and how you can associate typed objects with an XML schema collection. As you have seen, the **XML** data type makes it relatively easy to store XML documents and fragments.

However, in those Levels, we merely scratched the surface when it comes to understanding the full power of the **XML** data type. It turns out that the type supports five methods—**query()**, **value()**, **exist()**, **nodes()**, and **modify()**—that let you query and manipulate the elements and attributes within the instances stored in the **XML** objects. In this Level, we'll cover the two most common methods used to query XML data: **query()** and **value()**. In the Levels that follow, we'll cover the others.

Each **XML** method requires, at a minimum, one argument that is an XQuery expression. XQuery is a powerful scripting language used to access XML data. The language contains the functions, operators, variables, values, and other elements necessary to create complex expressions. SQL Server supports a subset of the XQuery language that you use to create the expressions you pass into the **XML** methods. With these expressions, you can identify very specifically the components in the XML instances you want to retrieve or modify.

> ***Note***
>
> *Because XQuery is such a complex language, we can touch upon only some of its components in this Level. For a more thorough understanding of XQuery and how it's implemented in SQL Server, see the MSDN XQuery language reference.*

Now let's get started with the **query()** and **value()** methods. In this Level, we look at a number of examples that use them to access XML data. The examples are based on a database and table I created on a local instance of SQL Server 2008 R2. The Transact-SQL in Listing 4-1 creates the test environment necessary to run these examples. The environment includes the **ClientDB** database, the **ClientInfoCollection** XML schema collection, and the **ClientInfo** table.

```
USE master;
GO

IF DB_ID('ClientDB') IS NOT NULL
```

```
DROP DATABASE ClientDB;
GO

CREATE DATABASE ClientDB;
GO

USE ClientDB;
GO

IF OBJECT_ID('ClientInfoCollection') IS NOT NULL
DROP XML SCHEMA COLLECTION ClientInfoCollection;
GO

CREATE XML SCHEMA COLLECTION ClientInfoCollection AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="urn:ClientInfoNamespace"
targetNamespace="urn:ClientInfoNamespace"
elementFormDefault="qualified">
  <xsd:element name="People">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Person" minOccurs="1"
maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="FirstName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="LastName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="FavoriteBook" type="xsd:string"
minOccurs="0" maxOccurs="5" />
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:integer"
use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>';
GO

IF OBJECT_ID('ClientInfo') IS NOT NULL
DROP TABLE ClientInfo;
```

```
GO

CREATE TABLE ClientInfo
(
  ClientID INT PRIMARY KEY IDENTITY,
  Info_untyped XML,
  Info_typed XML(ClientInfoCollection)
);

INSERT INTO ClientInfo (Info_untyped, Info_typed)
VALUES
(
  '<?xml version="1.0" encoding="UTF-8"?>
  <People>
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>',
  '<?xml version="1.0" encoding="UTF-8"?>
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>'
);
```

Listing 4-1: Setting up the test environment for the examples in this Level.

As Listing 4-1 shows, the **ClientInfo** table includes a typed **XML** column (**Info_typed**) and an untyped **XML** column (**Info_untyped**). The typed column is associated with the **ClientInfoCollection** XML schema collection, which includes only one XSD schema. The schema's namespace is **urn:ClientInfoNamespace**. For details about working with XML schema collections and typed database objects, refer back to Level 2.

The XML query() method

Perhaps the simplest of the **XML** methods to understand and use is the **query()** method. The method is most often used to return an instance of untyped XML that is a subset of the targeted XML data. To use the method, you specify the **XML** database object, the method itself, and the XQuery expression enclosed in parentheses and single quotes, as shown in the following syntax:

```
db_object.query('xquery_exp')
```

When calling the **query()** method, you replace the **db_object** placeholder with the name of the actual database object and replace the **xquery_exp** placeholder with the XQuery expression.

Let's look at an example that demonstrates how this works. In the **SELECT** statement shown in Listing 4-2, I use the **query()** method to retrieve data from the **Info_untyped** column. I first specify the column name, a period, and the method name. Then, enclosed in parentheses and single quotes, I add the XQuery expression **('/People')**.

```
SELECT Info_untyped.query('/People')
  AS People_untyped
FROM ClientInfo;
```

Listing 4-2: Using the **query()** method to return the **<People>** element.

In this case, the XQuery expression is as about simple as it can get. I am essentially telling the **query()** method to return the **<People>** element and all its contents (the child elements, attributes, and their values). To do so, as you can see, I needed only to specify the word **People**, preceded by a forward slash. Listing 4-3 shows the results returned by the **SELECT** statement.

```
<People>
    <Person id="1234">
        <FirstName>John</FirstName>
        <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
        <FirstName>Jane</FirstName>
        <LastName>Doe</LastName>
    </Person>
</People>
```

Listing 4-3: The results returned by the **/People** XQuery expression.

If I want to instead retrieve the **<People>** element from the typed column, I need to modify XQuery expression to include a reference to the namespace specified in the XSD schema defined in the XML schema collection. Listing 4-4 shows how I modified the statement to retrieve data from the typed **XML** column.

```
SELECT Info_typed.query(
  'declare namespace ns="urn:ClientInfoNamespace";
  /ns:People') AS People_typed
FROM ClientInfo;
```

Listing 4-4: Using the **query()** method on a typed **XML** column.

Notice that the query expression is broken into two sections, separated with a semicolon. The first section declares the namespace. It begins with the **declare namespace** keywords, followed by a namespace alias, in this case **ns**. The alias is followed with an equal sign and then the namespace itself (**urn:ClientInfoNamespace**), enclosed in double quotes.

The second section of the XQuery expression is similar to the expression used for an untyped column, except that the element name is first preceded by the namespace alias and a colon (**ns:**). For typed columns, your element names must be fully qualified, that is, they must reflect the schema they're associated with, and that is done by specifying the associated namespace. However, instead of typing the entire namespace for each element, you can instead use an alias, which is what I've done.

When you run the **SELECT** statement, it returns the results shown in Listing 4-5.

```
<People xmlns="urn:ClientInfoNamespace">
  <Person id="1234">
    <FirstName>John</FirstName>
    <LastName>Doe</LastName>
  </Person>
  <Person id="5678">
    <FirstName>Jane</FirstName>
    <LastName>Doe</LastName>
  </Person>
</People>
```

Listing 4-5: The results returned by an XQuery expression used for a typed column.

As you can see, the results returned from the typed column are nearly identical to those returned by the untyped column, except that now the results include a reference to the namespace.

You may have noticed that the last two examples return what is essentially the entire XML instance saved to the each column. Although the examples are useful for demonstrating how the **`query()`** method works, using the method to return the entire instance is not very useful because you can do that without using the method at all.

The key, of course, is to refine the XQuery expression to return more specific data.

Suppose, for example, instead of retrieving the entire XML document, we want to return each instance of the **`<Person>`** element, along with its child elements and attributes. For the untyped column, we would need to modify our XQuery expression by adding **`/Person`** to our path name so that our expression reads **`/People/Person`**, as shown in Listing 4-6.

```
SELECT
  Info_untyped.query(
    '/People/Person') AS People_untyped,
  Info_typed.query(
    'declare namespace ns="urn:ClientInfoNamespace";
    /ns:People/ns:Person') AS People_typed
FROM ClientInfo;
```

Listing 4-6: Retrieving the **`<Person>`** elements from the XML instance.

What I've done here is to further qualify the path name within the expression by adding the second element. Now only the **<Person>** elements are returned and not the **<People>** element. Listing 4-7 shows the results returned for the untyped column. Notice that only the two instances of the **<Person>** element have been returned.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
</Person>
<Person id="5678">
  <FirstName>Jane</FirstName>
  <LastName>Doe</LastName>
</Person>
```

Listing 4-7: The **<Person>** elements returned from the untyped column.

If you refer back to Listing 4-6, you'll see that for the typed column, the expression includes the **ns:** alias prefix before each element within the path name. Listing 4-8 shows the results returned for that column. This time, the namespace reference is included with each instance of the **<Person>** element.

```
<ns:Person xmlns:ns="urn:ClientInfoNamespace" id="1234">
  <ns:FirstName>John</ns:FirstName>
  <ns:LastName>Doe</ns:LastName>
</ns:Person>
<ns:Person xmlns:ns="urn:ClientInfoNamespace" id="5678">
  <ns:FirstName>Jane</ns:FirstName>
  <ns:LastName>Doe</ns:LastName>
</ns:Person>
```

Listing 4-8: The **<Person>** elements returned from the typed column.

Now suppose we want to return a specific instance of the **<Person>** element. One way we can do this is to further qualify the XQuery expression by adding a reference to the **id** attribute and a specific attribute value. In the example shown in Listing 4-9, I've added a reference to the **id** attribute for both the typed and untyped columns.

```
SELECT
  Info_untyped.query(
    '/People/Person[@id=1234]') AS People_untyped,
  Info_typed.query(
    'declare namespace ns="urn:ClientInfoNamespace";
    /ns:People/ns:Person[@id=5678]') AS People_typed
FROM ClientInfo;
```

Listing 4-9: Retrieving data for a specific **<Person>** element.

The first thing to notice is that I've enclosed the attribute reference in brackets. In addition, I preceded the attribute name with an at (**@**) sign and followed it with an equal and then provided the attribute value. For the untyped column, I used the value **1234**. As a result, the XML returned from that column includes only the **<Person>** element whose **id** value equals **1234**, as shown in Listing 4-10.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
</Person>
```

Listing 4-10: The **<Person>** element with an **id** value of **1234**.

For the typed column, I used the value **5678** in the attribute reference. Notice, however, that I do not need to include the namespace alias prefix along with the attribute name. Referencing the namespace in the element name is enough.

> ***Note***
>
> *When specifying a specific value in your XQuery expression, as I do for the* **id** *attribute value, string values should be enclosed in double quotes. However, the rules for numeric values are somewhat different. For untyped columns, you can also specify numeric values in double quotes or without the quotes, but when working with typed columns, you must conform to the schema, which in this case specifies the* **id** *attribute as an* **INT** *value. Consequently, you cannot enclose the value in quotes. If you do, you'll receive an error when running your statement.*

Not surprisingly, the XML returned from the typed column includes only the **<Person>** element whose **id** value equals **5678**, as shown in Listing 4-11.

```
<ns:Person xmlns:ns="urn:ClientInfoNamespace" id="5678">
  <ns:FirstName>Jane</ns:FirstName>
  <ns:LastName>Doe</ns:LastName>
</ns:Person>
```

Listing 4-11: The **<Person>** element with an **id** value of **5678**.

We can refine our XQuery expression even further by adding another child element to the path name, in this case the **<FirstName>** element. For the untyped column, we simply add **/FirstName** to the expression, and for the typed column, we add **/ns:FirstName**. Listing 4-12 shows what our **SELECT** statement now looks like.

```
SELECT
  Info_untyped.query(
    '/People/Person[@id=1234]/FirstName') AS People_untyped,
  Info_typed.query(
    'declare namespace ns="urn:ClientInfoNamespace";
    /ns:People/ns:Person[@id=5678]/ns:FirstName') AS People_typed
FROM ClientInfo;
```

Listing 4-12: Retrieving the **<FirstName>** element for a specific **<Person>** element.

The XML returned from the untyped column now includes only the **<FirstName>** child element of the **<Person>** element whose **id** value is **1234**, as shown in Listing 4-13.

```
<FirstName>John</FirstName>
```

Listing 4-13: The **<FirstName>** child element for the **<Person>** element with an **id** value of **1234**.

The XML returned from the typed column now includes only the **<FirstName>** child element of the **<Person>** element whose **id** value is **5678**, as shown in Listing 4-14. Notice that, even at this level, the namespace is included in the returned value.

```
<ns:FirstName xmlns:ns="urn:ClientInfoNamespace">Jane</
ns:FirstName>
```

Listing 4-14: The **<FirstName>** child element for the **<Person>** element with an **id** value of **5678**.

Another way you can reference a specific element within an XQuery expression is to specify the element's position number, relative to other instances of that element. For example, our source data includes two instances of the **<Person>** element. The first instance of that element is implicitly assigned the number 1 and the second instance the number 2. In the **SELECT** statement in Listing 4-15, I use the numbers **1** and **2** to reference those instances.

```
SELECT
  Info_untyped.query(
    '/People/Person[1]/FirstName') AS People_untyped,
  Info_typed.query(
    'declare namespace ns="urn:ClientInfoNamespace";
    /ns:People/ns:Person[2]/ns:FirstName') AS People_typed
FROM ClientInfo;
```

Listing 4-15: Using instance numbers to reference instances of the **<Person>** element.

Notice that instead of specifying an attribute reference, I specify the value **[1]** for the untyped column and the value **[2]** for the typed column. That means, in the case of the untyped column, the **[1]** indicates that the first instance of the **<Person>** element should be returned, or rather, the **<FirstName>** element of that instance, and for the typed column, the **[2]** indicates that the second instance of the **<Person>** element should be returned. Although I've used the numerical references, the **SELECT** statement in Listing 4-15 returns the same results as the statement in Listing 4-12.

The XML value() method

As handy as the **query()** method can be, there might be times that you want to retrieve a specific element or attribute value, rather than returning an XML element. That's where the **value()** method comes in. The method not only retrieves a specific value, but does so as a specified data type. For this reason, when you call the **value()** method, you must pass in two arguments—the XQuery expression and the Transact-SQL data type—as shown in the following syntax:

```
db_object.value('xquery_exp', 'sql_type')
```

Notice that you call the **value()** method in much the same way you call the **query()** method. The only difference is the second argument, in which you specify the data type. For example, the **SELECT** statement shown in Listing 4-16 retrieves the **<FirstName>** value from the XML and returns it with the **VARCHAR** data type.

```
SELECT
  Info_untyped.value(
    '(/People/Person[1]/FirstName)[1]',
    'varchar(20)') AS Name_untyped,
  Info_typed.value(
    'declare namespace ns="urn:ClientInfoNamespace";
    (/ns:People/ns:Person[2]/ns:FirstName)[1]',
    'varchar(20)') AS Name_typed
FROM ClientInfo;
```

Listing 4-16: Retrieving the **<FirstName>** values from the XML instances.

As the listing shows, you first specify the XQuery expression, followed by a comma, and then the data type. Like the XQuery expression, the data type must be enclosed in single quotes. That part should be fairly straightforward. What is not so straightforward is the XQuery expression itself. Although for both the typed and untyped columns the expressions are much the same as their counterparts in Listing 4-15, there is a significant difference. Each expression is enclosed in parentheses and followed by **[1]**. The parentheses ensure that the expression is treated as a single unit to which the **[1]** can be applied.

The **[1]** means that the first instance of the returned instances is the instance that the expression should use. For example, suppose your XQuery expression returns multiple **<Person>** elements. Surrounding the expression with parentheses and adding the **[1]** indicates that the first instance of **<Person>** should be used. Note, however, that even if your expression returns only one instance, you must still include the **[1]** because a singleton value is required by the **value()** method, and the **[1]** ensures that only one value can be returned.

In Listing 4-16, I specify **[1]** after the XQuery expression for both the untyped and typed columns. And because I've used the **value()** method in both cases, the **SELECT** statement returns only the first names of the two people listed in the XML documents, as shown below.

```
Name_untyped         Name_typed
-------------------- --------------------
John                 Jane
```

Listing 4-17: The **<FirstName>** values for the two **<Person>** elements.

In some cases, you can eliminate the internal numerical identifier after a specific element and use only the outer one to identify the XML element. If you do this, however, you must make sure your outer reference identifies the correct instance. For example, in Listing 4-18, I removed the numerical references associated with the **<Person>** element and then modified the expression for the typed column by changing the final **[1]** to **[2]**.

```
SELECT
  Info_untyped.value(
    '(/People/Person/FirstName)[1]',
    'varchar(20)') AS Name_untyped,
  Info_typed.value(
    'declare namespace ns="urn:ClientInfoNamespace";
    (/ns:People/ns:Person/ns:FirstName)[2]',
    'varchar(20)') AS Name_typed
FROM ClientInfo;
```

Listing 4-18: Retrieving the first and second instances of **<FirstName>**.

Because there are two instances of the **<FirstName>** element, you can use the final numeric qualifier to distinguish which instance you want to return. The statement returns the same results as the statement in Listing 4-16.

You can also use the same strategy to return an attribute value. For example, the **SELECT** statement in Listing 4-19 retrieves the values of each instance of the **id** attribute and assigns the **INT** data type to the returned values.

```
SELECT
  Info_untyped.value(
    '(/People/Person/@id)[1]',
    'int') AS Name_untyped,
  Info_typed.value(
    'declare namespace ns="urn:ClientInfoNamespace";
    (/ns:People/ns:Person/@id)[2]',
    'int') AS Name_typed
FROM ClientInfo;
```

Listing 4-19: Retrieving the two instances of the **id** attributes.

As you would expect, the **SELECT** statement returns only the attribute values, as shown in Listing 4-20.

```
Name_untyped         Name_typed
-------------------- --------------------
1234                 5678
```

Listing 4-20: The returned values for the two **id** attributes.

In addition to defining path names in your XQuery expressions, you can incorporate XQuery functions that let you further refine your query and manipulate data. For instance, XQuery supports the **count()** function, which provides a count of the number of instances returned by an expression. In Listing 4-21, I use the **count()** function to return the number of **<Person>** elements in the XML document in each **XML** column.

```
SELECT
  Info_untyped.value(
    'count(/People/Person)',
    'int') AS Number_untyped,
  Info_typed.value(
    'declare namespace ns="urn:ClientInfoNamespace";
    count(/ns:People/ns:Person)',
    'int') AS Number_typed
FROM ClientInfo;
```

Listing 4-21: Using the **count()** function to retrieve the number of **<Person>** elements.

For each XQuery expression, I specify the **count()** function, followed by the path name, which is enclosed in parenthesis. Because the **count()** function itself returns a singleton value, I do not have to tag the **[1]** onto the data path, even though I'm using the **value()** method. Listing 4-22 shows the results returned for each **XML** column. As you would expect, the value **2** is returned in both cases.

```
Number_untyped Number_typed
-------------- ------------
2              2
```

Listing 4-22: The number of **<Person>** elements in each **XML** column.

Another example of an XQuery function is **concat()**, which lets you concatenate two or more values from an XML document. To use the function, you specify each segment that you want to concatenate as an argument to the function, as demonstrated in Listing 4-23.

```
SELECT
  Info_untyped.value(
    'concat((/People/Person/FirstName)[2], " ",
      (/People/Person/LastName)[2])',
    'varchar(25)') AS FullName
FROM ClientInfo;
```

Listing 4-23: Using the **concat()** function to concatenate values.

In this case, I'm passing three arguments into the **concat()** function, which I enclose in parentheses and separate with commas. The first and third arguments are basic XQuery expressions that are themselves qualified with parentheses and a numerical tag to indicate which element instance to return, exactly the sort of expression you would expect to pass to the **value()** method. The second argument is merely a blank space, enclosed in double quotes. The space will be inserted between the first and last names. Listing 4-24 shows the results that the statement returns.

```
FullName
------------------------
Jane Doe
```

Listing 4-24: Returning the full name from the second instance of the **<Person>** element.

The first and last names have been concatenated into a single value. Both names come from the second instance of the **<Person>** element.

Conclusion

As you've seen in this Level, you can use the **query()** method to retrieve a subset of data from an XML instance, and you can use the **value()** method to retrieve individual element and attribute values from an XML instance. In Level 5, we'll cover the **exist()** and **nodes()** methods. Although these methods are also used to query XML data, the results they return are not simple XML instances or values.

In fact, the methods are often used in conjunction with the **`query()`** and **`value()`** methods because of the type of data they return.

In Level 6, we'll review the **`modify()`** method, the only **`XML`** method that lets you manipulate XML data. But keep in mind that, as stated earlier, XQuery expressions can get far more complicated than what I've demonstrated so far or will be demonstrating, so I recommend you review the XQuery Language Reference if you plan to write many XQuery expressions. Also note that we'll be using the same test environment in the next Level, so you might want to keep that around.

Level 5 – The XML exist() and nodes() Methods

In Level 4, I introduced you to the **query()** and **value()** methods, which are available to the **XML** data type and can be used to query data from an XML instance. As you'll recall, the **query()** method returns a subset of untyped XML from the target **XML** column (or other **XML** object), and the **value()** method returns a scalar value of a specified data type.

In this Level, I introduce you to two more **XML** methods: **exist()** and **nodes()**. Like the **query()** and **value()** methods, the **exist()** and **nodes()** methods let you query XML data by specifying an XQuery expression. However, the results returned by the methods are much different from **query()** and **value()**. The **exist()** method returns a **BIT** value, and the **nodes()** method returns a rowset view used to shred the XML instance. This will all become clearer as we work through the exercises.

> ***Note***
>
> *As mentioned in Level 4, XQuery is a complex language. We can touch upon only some of its elements in this Level. For a more thorough understanding of XQuery and how it's implemented in SQL Server, see the MSDN XQuery language reference.*

To demonstrate the **exist()** and **nodes()** methods, I used the same test environment I set up in Level 4. I created a database named **ClientDB**, an XML schema collection named **ClientInfoCollection**, and a table named **ClientInfo** (all created on a local instance of SQL Server 2008 R2), as shown in Listing 5-1.

```
USE master;
GO

IF DB_ID('ClientDB') IS NOT NULL
DROP DATABASE ClientDB;
GO

CREATE DATABASE ClientDB;
GO

USE ClientDB;
GO
```

```
IF OBJECT_ID('ClientInfoCollection') IS NOT NULL
DROP XML SCHEMA COLLECTION ClientInfoCollection;
GO

CREATE XML SCHEMA COLLECTION ClientInfoCollection AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="urn:ClientInfoNamespace"
targetNamespace="urn:ClientInfoNamespace"
elementFormDefault="qualified">
  <xsd:element name="People">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Person" minOccurs="1"
maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="FirstName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="LastName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="FavoriteBook" type="xsd:string"
minOccurs="0" maxOccurs="5" />
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:integer"
use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>';
GO

IF OBJECT_ID('ClientInfo') IS NOT NULL
DROP TABLE ClientInfo;
GO

CREATE TABLE ClientInfo
(
  ClientID INT PRIMARY KEY IDENTITY,
  Info_untyped XML,
  Info_typed XML(ClientInfoCollection)
);
```

```
INSERT INTO ClientInfo (Info_untyped, Info_typed)
VALUES
(
  '<?xml version="1.0" encoding="UTF-8"?>
  <People>
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>',
  '<?xml version="1.0" encoding="UTF-8"?>
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>'
);
```

Listing 5-1: Setting up the test environment for the examples in this Level.

Notice in Listing 5-1 that the **`ClientInfo`** table includes two **`XML`** columns, one untyped and one typed. In addition, the listing includes an **`INSERT`** statement that adds a row to the table. If you decide to try out the examples in this Level, you'll need to run this code before proceeding.

The XML exist() method

The **exist()** method tests for the existence of an element in the targeted XML instance. That element is specified by the XQuery expression passed into the method. As the following syntax shows, you enclose the expression in single quotes and parentheses:

```
db_object.exist('xquery_exp')
```

The *db_object* placeholder is the XML column, variable, or parameter that contains the targeted XML instance, and the *xquery_exp* placeholder is an expression made up of the XQuery elements supported by SQL Server.

So far, this is all pretty much like the **query()** and **value()** methods described in Level 4. However, the difference comes in the results being returned.

The **exist()** method does not retrieve an XML element or one of its values, but instead returns one of the following values, based on the existence of the element specified in the XQuery expression:

- A **BIT** value of **1** if the XQuery expression returns a nonempty result, that is, if the element exists.
- A **BIT** value of **0** if the XQuery expression returns an empty result, that is, if the element does not exist.
- A **NULL** value if the XML instance is null.

The best way to understand how the **exist()** method works is to see it in action. In Listing 5-2, I use the method to test for the existence of a **<FirstName>** element with a value equal to **Jane**. Notice that I've enclosed the element in brackets and used double quotes for the string value.

```
SELECT
  Info_untyped.exist(
    '/People/Person[FirstName="Jane"]')
FROM ClientInfo;
```

Listing 5-2: Verifying the existence of a specific element.

Because the specified element exists in the targeted XML instance, the **SELECT** statement returns the value **1**, which is the result we expected.

As you can see, it's a relatively straightforward process to use the **exist()** method against an untyped **XML** column. To use the method against a typed **XML** object, you must include the required namespace-related information, as shown in Listing 5-3.

```
SELECT
  Info_untyped.exist(
    '/People/Person[FirstName="Jane"]'),
  Info_typed.exist(
    'declare namespace ns="urn:ClientInfoNamespace";
    /ns:People/ns:Person[ns:FirstName="Jane"]')
FROM ClientInfo;
```

Listing 5-3: Verifying the existence of an element in a typed column.

As you would expect from reading Level 4, your **xquery_exp** argument is divided into two parts, separated by a semicolon, with the entire argument enclosed in single quotes. The first part is the namespace declaration, which specifies the namespace used by the targeted XML instance and defines an alias (**ns**) for that namespace. The second part of the argument is the path name itself, with **ns:** inserted before each element. Because the typed column, like the untyped column, contains the element specified by the XQuery expression, it, too, returns a value of **1**.

But suppose the element specified in the expression does not exist, as is the case in Listing 5-4. This time, for both the untyped and typed columns, the **exist()** method is looking for a **<FirstName>** element whose value is equal to **Ralph**.

```
SELECT
  Info_untyped.exist(
    '/People/Person[FirstName="Ralph"]'),
  Info_typed.exist(
    'declare namespace ns="urn:ClientInfoNamespace";
    /ns:People/ns:Person[ns:FirstName="Ralph"]')
FROM ClientInfo;
```

Listing 5-4: Testing for a nonexistent value in the typed and untyped columns.

Of course, such an element does not exist. As a result, the **exist()** method, and by extension the **SELECT** statement, return a value of **0** for both columns.

So far, the examples have merely provided a way to demonstrate how the **exist()** method works. In reality, you're much more likely to use the method to check the existence of an element before carrying out another operation.

For example, in Listing 5-5, I use the **exist()** method in a **SELECT** statement's **WHERE** clause to check for the existence of a **<Person>** element whose **id** attribute has a value equal to **5678**. If this element exists—that is, the value returned by the **exist()** method equals **1**—the condition specified in the **WHERE** clause evaluates to **TRUE** and the data queried in the **SELECT** list is returned.

```
SELECT
  ClientID,
  Info_untyped.value(
    'concat((/People/Person[@id=5678]/FirstName)[1], " ",
      (/People/Person[@id=5678]/LastName)[1])',
    'varchar(25)') AS FullName
FROM ClientInfo
WHERE
  Info_untyped.exist(
    '/People/Person[@id=5678]') = 1;
```

Listing 5-5: Using the **WHERE** clause to test for the existence of an attribute value.

The **SELECT** list itself uses the **concat()** XQuery function and **XML value()** method to concatenate the first and last names associated with the **<Person>** element whose **id** value equals **5678**.

Listing 5-6 shows the results returned by the **SELECT** statement. If the **WHERE** clause had evaluated to **FALSE**, the statement would have returned no rows.

```
ClientID    FullName
----------- ------------------------
1           Jane Doe
```

Listing 5-6: The results returned after an attribute's existence has been confirmed.

SQL Server also lets you pass variable values into your XQuery expression, which is handy if you want to reuse code. In Listing 5-7, for instance, I modified the preceding example so that the value **5678** could be passed in through the **@id** variable.

```
SELECT
  ClientID,
  Info_untyped.value(
    'concat((/People/Person[@id=5678]/FirstName)[1], " ",
      (/People/Person[@id=5678]/LastName)[1])',
    'varchar(25)') AS FullName
FROM ClientInfo
WHERE
  Info_untyped.exist(
    '/People/Person[@id=5678]') = 1;
```

Listing 5-7: Using a variable to pass a value into an XQuery expression.

Notice that, in order to call the variable value from within the XQuery expression, I specified **`sql:variable("@id")`**, rather than **`5678`**. Everything else about the **`SELECT`** statement is the same as the preceding example, and, as expected, the statement returns the same results.

The XML nodes() method

Of all the **`XML`** methods we've discussed so far, the **`nodes()`** method is perhaps the trickiest to understand. Unlike the previous methods, which return XML fragments or scalar values, the **`nodes()`** method returns a table (rowset view) with a single column, and each row of that table contains a logical copy of the targeted XML instance. The purpose of these results is to let you shred the targeted XML instance into relational data. (This will become clearer as we work through the examples.)

Because the **`nodes()`** method returns the data as a rowset view, you can use that method only where a table expression is expected in a Transact-SQL statement, such as in the **`FROM`** clause. In addition, you must assign table and column aliases to the method's results, as shown in the following syntax:

```
db_object.nodes('xquery_exp') AS table_alias(column_alias)
```

As with other **`XML`** methods, you must specify an **`XML`** object and an XQuery expression. But you then follow with the table alias and column alias, in parentheses. The aliases let you reference the rowset view from other parts of the **`SELECT`** statement.

The key to understanding how to use the **nodes()** method is in the concept of a *context node*. Every XML document has an implicit context node, which is at the top Level of the XML instance. You can think of the context node as a reference point within the XML instance. When you use the **nodes()** method, the context node is set to a specific element within each row of data returned by the method. That context node is identified by the XQuery expression you pass into the method. It's the context node that lets you shred the XML data in a meaningful way.

Let's look at an example to demonstrate how this works. But first, we need to add a row to our table. Listing 5-8 shows the **INSERT** statement I used to add the row, which adds data only to the untyped column (because that's all we need right now).

```
INSERT INTO ClientInfo (Info_untyped)
VALUES
(
  '<?xml version="1.0" encoding="UTF-8"?>
  <People>
    <Person id="4321">
      <FirstName>Jack</FirstName>
      <LastName>Smith</LastName>
    </Person>
    <Person id="8765">
      <FirstName>Jill</FirstName>
      <LastName>Smith</LastName>
    </Person>
  </People>'
);
```

Listing 5-8: Inserting an additional row into the **ClientInfo** table.

Now let's get down to the example. In Listing 5-9, I use the **nodes()** method in the **FROM** clause to return a rowset view of the targeted XML instances. When I call the method, I include an XQuery expression that sets the context node to **/People/Person**. I also provide the table and column aliases, **People** and **Person**, respectively, so I can reference the rowset view in the **SELECT** list. In addition, I use the **nodes()** method along with the **CROSS APPLY** operator in order to associate the **ClientInfo** table with the rowset view.

```
SELECT
  ClientID,
  Person.query('.') AS Person
FROM ClientInfo CROSS APPLY
  Info_untyped.nodes('/People/Person') AS People(Person);
```

Listing 5-9: Using the **nodes()** method to shred XML data.

Although the **nodes()** method returns a rowset view that you can reference in other parts of the statement, you can refer to that view only through an **XML** method, as I've done in the **SELECT** list. However, as you can see, my XQuery expression is merely a period, which is shorthand for referencing the context node. Because of this, the **SELECT** statement returns the results shown in Table 5-1. (I put the results in a table to make it easier to read the XML instances in the **Person** column.)

ClientID	Person
1	<Person id="1234"> <FirstName>John</FirstName> <LastName>Doe</LastName> </Person>
1	<Person id="5678"> <FirstName>Jane</FirstName> <LastName>Doe</LastName> </Person>
2	<Person id="4321"> <FirstName>Jack</FirstName> <LastName>Smith</LastName> </Person>
2	<Person id="8765"> <FirstName>Jill</FirstName> <LastName>Smith</LastName> </Person>

Table 5-1: The returned data relative to the context node.

Notice that the results include the **<Person>** element for each person in each row of the **ClientInfo** table. In other words, because each row in the source table contains two instances of the **<Person>** element, the rowset view includes two rows for each row in the table, one for each **<Person>** element. SQL Server then uses the context node to iterate through each XML instance in the rowset view and to return the appropriate instance of **<Person>**.

Keep in mind that it is the context node that provides the ability to return different results for each row in the rowset view, not the rowset view itself. As you'll recall, each row in the rowset view contains a full copy of the targeted XML instance. SQL Server iterates through each instance based on the element specified by the context node, similar to how a cursor identifies a current row. However, if you were to call the parent of the context node, your results would be much different. For example, in Listing 5-10, I use the double period for the **query()** method's XQuery expression, which is shorthand for the context node's parent.

```
SELECT
  ClientID,
  Person.query('..') AS Person
FROM ClientInfo CROSS APPLY
  Info_untyped.nodes('/People/Person') AS People(Person);
```

Listing 5-10: Using the context node's parent accessor to return data.

Because we're calling the context node's parent, every child element within that parent is also returned, as shown in Table 5-2. As a result, unique elements are no longer returned for each row in the target table. What this demonstrates, essentially, is that each row in the rowset view contains the entire XML instance.

ClientID	Person
1	<People> <Person id="1234"> <FirstName>John</FirstName> <LastName>Doe</LastName> </Person> <Person id="5678"> <FirstName>Jane</FirstName> <LastName>Doe</LastName> </Person> </People>
1	<People> <Person id="1234"> <FirstName>John</FirstName> <LastName>Doe</LastName> </Person> <Person id="5678"> <FirstName>Jane</FirstName> <LastName>Doe</LastName> </Person> </People>
2	<People> <Person id="4321"> <FirstName>Jack</FirstName> <LastName>Smith</LastName> </Person> <Person id="8765"> <FirstName>Jill</FirstName> <LastName>Smith</LastName> </Person> </People>

2	<People> <Person id="4321"> <FirstName>Jack</FirstName> <LastName>Smith</LastName> </Person> <Person id="8765"> <FirstName>Jill</FirstName> <LastName>Smith</LastName> </Person> </People>

Table 5-2: The returned data based on the parent accessor.

Chances are, if you're going to use the **nodes()** method to shred an XML instance, you'll want to do it in a more meaningful way than what I've done so far. In Listing 5-11, I use the **value()** method and **concat()** function to return the full name for each instance of the **<Person>** element.

```
SELECT
  ClientID,
  Person.value('concat(./FirstName[1], " ",
    ./LastName[1])', 'varchar(30)') AS FullName
FROM ClientInfo CROSS APPLY
  Info_untyped.nodes('/People/Person') AS People(Person);
```

Listing 5-11: Using the context node to return values from child elements.

Because I'm using the **value()** method, my results from the shredded XML are now returned as **VARCHAR** values, as shown in Listing 5-12.

```
ClientID  FullName
--------  ----------
1         John Doe
1         Jane Doe
2         Jack Smith
2         Jill Smith
```

Listing 5-12: Shredded XML returned by the **`SELECT`** statement.

As you can see, the results are now much more useful. The first and last names of each person listed in the two rows of the **`ClientInfo`** table are now returned as relational data. It can take some practice to get used to using the **`nodes()`** method, but when you do get it figured out, you'll find it a useful tool.

Conclusion

As this Level has demonstrated, you can use the **`exist()`** method to check the existence of an element within an XML document or fragment. Most often, you'll be using the method in the **`WHERE`** clause to verify an element's existence before proceeding with the rest of the statement. The **`nodes()`** method serves a different function. It lets you shred an XML instance and return the information as relational data. In the next level, I'll discuss the **`modify()`** method, which is the only **`XML`** method that lets you manipulate XML data. In the meantime, don't forget to review the XQuery Language Reference so you better understand how to write XQuery expressions.

Level 6 – Inserting Data into an XML Instance

In Levels 4 and 5, we looked at the methods you can use to retrieve element-specific data from an **XML** column, variable, or parameter. Those methods include **query()**, **value()**, **exist()**, and **nodes()**, which each provide a different means for accessing data in an XML instance. As a minimum, when you call one of those methods, you pass in an XQuery expression that defines what data to retrieve from that instance.

In this Level, we look at the **modify()** method, the only method available to the **XML** data type that lets you manipulate XML data. Unlike the other **XML** methods, the **modify()** method takes an XML Data Modification Language (XML DML) expression as an argument, rather than a regular XQuery expression. XML DML is an extension of the XQuery language that lets you insert, update, and delete XML data. In this Level, we'll be concerned specifically with how to use the method to insert data. In subsequent Levels, we'll review how to use the method to modify and delete data.

When you call the **modify()** method, you must pass in an XML DML expression. The expression is the method's only argument, as shown in the following syntax:

```
db_object.modify('xml_dml')
```

As you can see, you simply append the **XML** object name with a period and method name, followed by the XML DML expression enclosed in parentheses and single quotes. Not surprisingly, it's the expression itself where things get a bit more complicated. This Level focuses on how to create various XML DML expressions and provides a number of examples that demonstrate how to insert data into an XML instance.

Note

Many of the elements that make up an XML DML expression use basic XQuery syntax, which itself is a complex language. As with previous Levels, we can touch upon only some of the XQuery elements. For a more thorough understanding of XQuery and how it's implemented in SQL Server, see the MSDN XQuery language reference. For more details about XML DML, see the MSDN article "XML Data Modification Language (XML DML)."

Setting up your test environment

If you want to try out the exercises in this Level, you'll first need to run the Transact-SQL code shown in Listing 6-1. The code creates the **ClientDB** database, adds the **ClientInfoCollection** XML schema collection to the database, and then creates and populates the **ClientInfo** table.

```
USE master;
GO

IF DB_ID('ClientDB') IS NOT NULL
DROP DATABASE ClientDB;
GO

CREATE DATABASE ClientDB;
GO

USE ClientDB;
GO

IF OBJECT_ID('ClientInfoCollection') IS NOT NULL
DROP XML SCHEMA COLLECTION ClientInfoCollection;
GO

CREATE XML SCHEMA COLLECTION ClientInfoCollection AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="urn:ClientInfoNamespace"
targetNamespace="urn:ClientInfoNamespace"
elementFormDefault="qualified">
  <xsd:element name="People">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Person" minOccurs="1"
maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="FirstName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="LastName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="FavoriteBooks" minOccurs="0"
```

```
maxOccurs="1">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="Book" type="xsd:string"
minOccurs="0" maxOccurs="5" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:integer"
use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>';
GO

IF OBJECT_ID('ClientInfo') IS NOT NULL
DROP TABLE ClientInfo;
GO

CREATE TABLE ClientInfo
(
  ClientID INT PRIMARY KEY IDENTITY,
  Info_untyped XML,
  Info_typed XML(ClientInfoCollection)
);

INSERT INTO ClientInfo (Info_untyped, Info_typed)
VALUES
(
  '<?xml version="1.0" encoding="UTF-8"?>
  <People>
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>',
```

```
  '<?xml version="1.0" encoding="UTF-8"?>
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>'
);
```

Listing 6-1: Setting up the test environment for the examples in this Level.

Notice that the code includes an **INSERT** statement that adds a row of data to the **ClientInfo** table. That data includes two XML instances, one that targets the untyped column and one that targets the typed column.

If you tried out the examples in Levels 4 and 5, you'll find the code in Listing 6-1 to be slightly different from those Levels. The schema defined in this collection contains a few extra elements. Also note, you should run the examples in the order they're provided, because some of them build on previous ones.

Inserting data into an XML instance

To insert data into an XML instance, your XML DML expression must include the keywords and XQuery expressions necessary to indicate the type of data modification operation to perform as well as what and where to add the data, as shown in the following syntax:

```
db_object.modify (
'insert xquery_exp1
[as first | as last] into | after | before
xquery_exp2 ')
```

As you can see, when we break apart the XML DML expression, our method call becomes much more complex. Notice that the expression includes several individual elements:

- The **INSERT** keyword indicates that this is an insert operation.
- The **xquery_exp1** placeholder is an XQuery expression that defines one or more XML components to be inserted into the XML data.

- The following directional keywords identify where in the targeted node (as defined by **xquery_exp2**) to insert the data:
  - **[as first | as last] into**: The data is inserted as one or more child nodes to the targeted node. If child nodes already exist, you must also specify the **as first** or **as last** keywords. If you specify **as first**, the new data is added before the existing child nodes. If you specify **as last**, the new data is added after the existing child nodes.
  - **after**: Data is inserted as siblings to the targeted node, directly after that node.
  - **before**: Data is inserted as siblings to the targeted node, directly before that node.
- The **xquery_exp2** placeholder is an XQuery expression that defines the XML node that is the target of the data to be inserted.

Once you understand how the pieces fit together, the XML DML expression is fairly straightforward. And the best way to gain that understanding is to see these expressions in action. So let's get started.

Listing 6-2 shows an **UPDATE** statement that includes the **modify()** method, which I use to insert the **<FavoriteBooks>** element into the first instance of the **<Person>** element, as identified by the **id** attribute value **1234**.

```
UPDATE ClientInfo
SET Info_untyped.modify(
  'insert <FavoriteBooks />
   as last into
   (/People/Person[@id=1234])[1] ')
WHERE ClientID = 1;

SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 6-2: Adding an element to an XML fragment.

The first thing worth pointing out is that I'm using the **modify()** method as part of the **SET** clause of an **UPDATE** statement. When you use the **modify()** method, you must do so within a data modification structure such as a **SET** clause.

As for the XML DML expression itself, I start with the **INSERT** keyword, followed by the expression that identifies the data to be inserted, in this case, the **<FavoriteBooks>** element. Notice that I use the shorthand notation (**/>**) to specify the closing element, rather than specifying **<FavoriteBooks></FavoriteBooks>**. However, you can take either approach.

Next, I include the **as last into** keywords to specify that the new element should be added to the end of the child elements of the target node.

The final expression, **(/People/Person[@id=1234])[1]**, is the target node. In this case, that node is the first instance of the **<Person>** element. Notice that I add **[1]** to the end of the expression. The **modify()** method requires that the expression return a single target node. Adding a bracketed value in this way ensures that only one value is returned, in this case, the first one. Even if only one node would be returned (as is the case here), you must still specify the **[1]**.

That's all there is to my XML DML expression, except that I've also enclosed it in parentheses and single quotes. I then tagged a **SELECT** statement onto the example to verify the operation. Listing 6-3 shows the XML fragment returned by that statement. As you can see, the **<FavoriteBooks>** node has been added as a child node to the **<Person>** element, after the existing child elements.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks />
</Person>
```

Listing 6-3: XML fragment with new element.

You probably noticed that the example shown in Listing 6-2 modifies data in the untyped **XML** column (**Info_untyped**). However, I can achieve similar results in the typed columns. In Listing 6-4, I modify the XML DML expression to include the namespace reference. As you saw with the XQuery expressions used for the other **XML** methods, the XML DML expression is divided into two sections, separated by a semicolon. The first section is the namespace declaration.

```
UPDATE ClientInfo
SET Info_typed.modify(
  'declare namespace ns="urn:ClientInfoNamespace";
   insert <ns:FavoriteBooks />
   as last into
   (/ns:People/ns:Person[@id=1234])[1] ')
WHERE ClientID = 1;

SELECT Info_typed.query(
  'declare namespace ns="urn:ClientInfoNamespace";
  /ns:People/ns:Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 6-4: Adding an element to a typed XML column.

The second section of the XML DML expression is similar to the previous example, except that I precede each referenced node with the namespace alias and a colon (**ns:**). Everything else is the same.

Listing 6-5 shows the results returned by the **SELECT** statement. As you can see, the **<FavoriteBooks>** element has been added to the typed XML instance.

```
<ns:Person xmlns:ns="urn:ClientInfoNamespace" id="1234">
  <ns:FirstName>John</ns:FirstName>
  <ns:LastName>Doe</ns:LastName>
  <ns:FavoriteBooks />
</ns:Person>
```

Listing 6-5: XML fragment with new element.

Now let's look at another example. In Listing 6-6, I add the **<Book>** element as a child to the **<FavoriteBooks>** element. This time, however, I include an element value, **Slaughterhouse-Five**.

```
UPDATE ClientInfo
SET Info_untyped.modify(
  'insert <Book>Slaughterhouse-Five</Book>
   into
   (/People/Person[@id=1234]/FavoriteBooks)[1] ')
WHERE ClientID = 1;
```

```
SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 6-6: Adding a book to favorite books.

Notice that, after the first expression, I include the **into** keyword, but not the **as first** or **as last** keywords. Because the **<FavoriteBooks>** node currently contains no child elements, I do not need either of these options. Listing 6-7 shows the results returned by the **SELECT** statement in this example. As you can see, the **<Book>** node has been added as a child element to the **<FavoriteBooks>** node.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks>
    <Book>Slaughterhouse-Five</Book>
  </FavoriteBooks>
</Person>
```

Listing 6-7: XML fragment with a new book listed.

Again, we can do the same thing for the typed column. As Listing 6-8 shows, I need only add the necessary namespace declaration and references. That includes preceding each node with the namespace alias and colon (**ns:**), even if it is a closing node.

```
UPDATE ClientInfo
SET Info_typed.modify(
  'declare namespace ns="urn:ClientInfoNamespace";
   insert <ns:Book>Slaughterhouse-Five</ns:Book>
   into
   (/ns:People/ns:Person[@id=1234]/ns:FavoriteBooks)[1] ')
WHERE ClientID = 1;

SELECT Info_typed.query(
  'declare namespace ns="urn:ClientInfoNamespace";
  /ns:People/ns:Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 6-8: Adding a book to the XML in a typed column.

Not surprisingly, the **SELECT** statement returns an XML instance that includes the **<Book>** element, as shown in Listing 6-9.

```
<ns:Person xmlns:ns="urn:ClientInfoNamespace" id="1234">
  <ns:FirstName>John</ns:FirstName>
  <ns:LastName>Doe</ns:LastName>
  <ns:FavoriteBooks>
    <ns:Book>Slaughterhouse-Five</ns:Book>
  </ns:FavoriteBooks>
</ns:Person>
```

Listing 6-9: XML fragment with new book.

Now let's look at how to add an attribute to an existing element. To do so, you must specify the **attribute** keyword, attribute name, and attribute value after the **INSERT** keyword. For example, the **UPDATE** statement in Listing 6-10 creates an attribute named **rating** and sets its value to **5**.

```
UPDATE ClientInfo
SET Info_untyped.modify(
  'insert attribute rating {"5"}
   into
   (/People/Person[@id=1234]/FavoriteBooks/Book)[1] ')
WHERE ClientID = 1;

SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 6-10: Adding an attribute to an XML element.

Notice that I enclose the attribute value in curly brackets and double quotes and that I specify the **into** keyword without the **as first** or **as last** option. Because we're not concerned with child elements in this case, the optional keywords aren't necessary.

The final expression in the XML DML expression identifies the target node, which in this case is the **<Book>** element. This is the element that will receive the new attribute. Listing 6-11 shows the results returned by the **SELECT** statement. As you would expect, the **<Book>** element now includes the **rating** attribute and its associated value of **5**.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks>
    <Book rating="5">Slaughterhouse-Five</Book>
  </FavoriteBooks>
</Person>
```

Listing 6-11: XML element with the new attribute.

Now suppose we want to add another book to our list of books. One way we can do this is to use the **into** keyword along with one of the optional values to specify where to place the new element. However, another approach is to instead use the **after** keyword, which inserts the node as a sibling element after the specified node. That means your target node must be specific enough to identify where the new element should be inserted. For example, in Listing 6-12, I add a second **<Book>** element after the first one. To do so, my target expression specifically references that first **<Book>** node.

```
UPDATE ClientInfo
SET Info_untyped.modify(
  'insert <Book>Beloved</Book>
   after
   (/People/Person[@id=1234]/FavoriteBooks/Book)[1] ')
WHERE ClientID = 1;

SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 6-12: Inserting a second book in the XML fragment.

By taking this approach, I do not have to be concerned with the optional keywords **as first** or **as last**. The **after** keyword is enough. The key is to make sure my second expression properly targets the instance of **<Book>** that I want my new element to follow, which I do by using the **[1]** to indicate that the first instance should be used. As expected, the **SELECT** statement returns the results shown in Listing 6-13. Notice that the second **<Book>** element has been added in the expected location.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks>
    <Book rating="5">Slaughterhouse-Five</Book>
    <Book>Beloved</Book>
  </FavoriteBooks>
</Person>
```

Listing 6-13: XML fragment with the second book.

Up to this point, we've added only one node to our target element in each of the examples. However, you can specify multiple nodes in a single XML DML expression. For example, in the **UPDATE** statement shown in Listing 6-14, I insert two instances of the **<Book>** element into the target node.

```
UPDATE ClientInfo
SET Info_untyped.modify(
  'insert (
     <Book>Mrs Dalloway</Book>,
     <Book>One Hundred Years of Solitude</Book>)
   after
   (/People/Person[@id=1234]/FavoriteBooks/Book)[2] ')
WHERE ClientID = 1;

SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 6-14: Inserting two books at once into the XML element.

As you can see, I enclose the two **<Book>** elements in parentheses and separate them with a comma. The rest of the XML DML expression is just like the preceding example, except for the target element. In this case, I use **[2]** to specify that the new books should follow the second **<Book>** instance, rather than the first. Listing 6-15 shows the results returned by the **SELECT** statement. As expected, the **<FavoriteBooks>** element now contains four child elements.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks>
    <Book rating="5">Slaughterhouse-Five</Book>
    <Book>Beloved</Book>
    <Book>Mrs Dalloway</Book>
    <Book>One Hundred Years of Solitude</Book>
  </FavoriteBooks>
</Person>
```

Listing 6-15: XML element with the two additional books.

Now let's look at the example shown in Listing 6-16. This time, I insert a comment as a child to the **<FavoriteBooks>** element, but before all the **<Book>** elements. To do so, I specify the **INSERT** keyword followed by an XQuery expression, as I do in the other examples. However, the expression in this case is the comment, which is denoted by the opening comment tag (**<!--**) tag and the closing tag (**-->**).

```
UPDATE ClientInfo
SET Info_untyped.modify(
  'insert <!-- Books rated on scale 1-5 -->
   before
    (/People/Person[@id=1234]/FavoriteBooks/Book)[1] ')
WHERE ClientID = 1;

SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 6-16: Inserting a comment in an XML fragment.

After the first expression, I specify the **before** keyword, followed by the node that I want to precede with the comment, which in this case is the first **<Book>** element. Listing 6-17 shows the results now returned by the **SELECT** statement.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks>
    <!-- Books rated on scale 1-5 -->
    <Book rating="5">Slaughterhouse-Five</Book>
    <Book>Beloved</Book>
    <Book>Mrs Dalloway</Book>
    <Book>One Hundred Years of Solitude</Book>
  </FavoriteBooks>
</Person>
```

Listing 6-17: XML fragment with the new comment.

As the results indicate, the comment has been added as a child to the **<FavoriteBooks>** element, before all the **<Book>** elements.

Conclusion

In this Level, you learned about the many ways you can use the **modify()** method to insert a node into an XML instance. As you've seen, you can add a node as a child element of the targeted node or as a sibling element to that node. You can also specify where the new element should be located among the other elements. In addition, the **modify()** method lets you add attributes and comments to your XML instance, as well as adding new elements. In the next Level, you'll learn how to use the method to modify element and attribute values in your XML instance.

Level 7 – Updating Data in an XML Instance

Level 6 introduced you to the **modify()** method, which is available to the **XML** data type for manipulating data. The Level showed you how to use the method to insert data into an XML instance. As the examples demonstrated, the method provides several options that let you control how you add the data.

In this Level, you'll learn how to use the **modify()** method to update data in an XML instance. As is the case when inserting data, the method takes an XML Data Modification Language (XML DML) expression as an argument when updating the data. XML DML is an extension of the XQuery language that lets you insert, update, and delete data in an XML instance.

> ***Note***
>
> *As with previous Levels, we can touch upon only some of the XML DML and XQuery elements in this Level. For a more thorough understanding of XQuery and how it's implemented in SQL Server, see the MSDN XQuery language reference. For more details about XML DML, see the MSDN article "XML Data Modification Language (XML DML)."*

To try out the examples in this Level, you'll first need to run the Transact-SQL code shown in Listing 7-1. The code creates the **ClientDB** database, adds the **ClientInfo-Collection** XML schema collection to the database, and then creates and populates the **ClientInfo** table.

```
USE master;
GO

IF DB_ID('ClientDB') IS NOT NULL
DROP DATABASE ClientDB;
GO

CREATE DATABASE ClientDB;
GO

USE ClientDB;
```

```
GO

IF EXISTS(
  SELECT * FROM sys.xml_schema_collections
  WHERE name = 'ClientInfoCollection')
DROP XML SCHEMA COLLECTION ClientInfoCollection;
GO

CREATE XML SCHEMA COLLECTION ClientInfoCollection AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="urn:ClientInfoNamespace"
targetNamespace="urn:ClientInfoNamespace"
elementFormDefault="qualified">
  <xsd:element name="People">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Person" minOccurs="1"
maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="FirstName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="LastName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="FavoriteBooks" minOccurs="0"
maxOccurs="1">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="Book" minOccurs="0"
maxOccurs="5">
                      <xsd:complexType>
                        <xsd:simpleContent>
                          <xsd:extension base="xsd:string">
                            <xsd:attribute name="rating"
type="xsd:decimal" />
                            <xsd:attribute name="recommend"
type="xsd:string" />
                          </xsd:extension>
                        </xsd:simpleContent>
                      </xsd:complexType>
                    </xsd:element>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
```

```
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:integer"
use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>';
GO

IF OBJECT_ID('ClientInfo') IS NOT NULL
DROP TABLE ClientInfo;
GO

CREATE TABLE ClientInfo
(
  ClientID INT PRIMARY KEY IDENTITY,
  Info_untyped XML,
  Info_typed XML(ClientInfoCollection)
);

INSERT INTO ClientInfo (Info_untyped, Info_typed)
VALUES
(
  '<?xml version="1.0" encoding="UTF-8"?>
  <People>
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
      <FavoriteBooks>
        <!-- Books rated on scale 1-5 -->
        <Book rating="5">Slaughterhouse-Five</Book>
      </FavoriteBooks>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>',
  '<?xml version="1.0" encoding="UTF-8"?>
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
      <FavoriteBooks>
        <!-- Books rated on scale 1-5 -->
```

```
        <Book rating="5">Slaughterhouse-Five</Book>
      </FavoriteBooks>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>'
);
```

Listing 7-1: Setting up the test environment for the examples in this Level.

The code in Listing 7-1 includes an **INSERT** statement that adds a row of data to the **ClientInfo** table. The row contains two XML instances, one that targets the untyped column and one that targets the typed column. Otherwise, the elements, attributes, and values that make up each instance are the same.

I created the code in Listing 7-1 on a local instance of SQL Server 2012, and then created the following examples in the same environment. Once you've set up this environment on your system, you'll be ready to try out these examples.

Updating data in an XML instance

You can use the **modify()** method to update specific element and attribute values in an XML instance. When the method is used in this way, the XML DML expression must include the **replace value of** keywords and the **with** keyword, along with two expressions, as shown in the following syntax:

```
db_object.modify(
'replace value of xquery_exp
with value_exp')
```

The first expression, *xquery_exp*, is an XQuery expression that defines the target element or attribute whose value will be modified. The second expression is a literal value or expression that defines the new value to be inserted into the target element or attribute. Together the keywords and expressions must be enclosed in single quotes and parentheses.

Let's look at an example that demonstrates how the method works to modify data. In Listing 7-2, the **UPDATE** statement uses the method to change the name of the book listed in the **Info_untyped** column.

```
UPDATE ClientInfo
SET Info_untyped.modify(
  'replace value of
   (/People/Person[@id=1234]/FavoriteBooks/Book/text())[1]
   with "The Catcher in the Rye" ')
WHERE ClientID = 1;

SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 7-2: Updating an element value in an untyped XML instance.

The first thing to note is that the XML DML expression begins with the **replace value of** keywords, followed by an XQuery expression that specifies the first **<Book>** child element for the person with an **id** attribute value of **1234**. As you saw in other examples of the **modify()** method, the XQuery expression in this case must return a scalar value. For this example, **[1]** is used to indicate that the first instance of the **<Book>** element be returned. Even if there is only one instance of an element, as in this situation, the numerical qualifier must still be specified.

Notice also the **text()** function appended to the end of the path expression. The function returns only the element value, as opposed to the metadata that defines it. For an element in an untyped column, you must specify this function (or some comparable expression) so that your path specifically targets that value. If the function is not specified, SQL Server returns an error.

The next component of the XML DML expression is the **with** keyword, followed by the value expression, which in this case is the literal value **The Catcher in the Rye**. Notice that you must enclose literal values in double quotes. When you run the **UPDATE** statement, this value replaces the existing value (**Slaughterhouse-Five**). The **SELECT** statement appended onto Listing 7-2 confirms that this is the case, as shown in the results in Listing 7-3. As you can see, the **<Book>** element now includes the new title.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks>
    <!-- Books rated on scale 1-5 -->
    <Book rating="5">The Catcher in the Rye</Book>
  </FavoriteBooks>
</Person>
```

Listing 7-3: The updated element value in the untyped XML instance.

The process for updating an element value in a typed column is similar to that of an untyped column. As to be expected, you must specify the namespace information, as shown in Listing 7-4. As you've seen with other XQuery and XML DML expressions, the expression is divided into two parts, separated by a semicolon. The first part declares the namespace and assigns an alias to that namespace. The second part is similar to what you specify for an untyped column, except that you include the namespace alias in your element references and you do not use the **`text()`** function in your XQuery expression.

```
UPDATE ClientInfo
SET Info_typed.modify(
  'declare namespace ns="urn:ClientInfoNamespace";
   replace value of
   (/ns:People/ns:Person[@id=1234]/ns:FavoriteBooks/ns:Book)[1]
   with "The Catcher in the Rye" ')
WHERE ClientID = 1;

SELECT Info_typed.query(
  'declare namespace ns="urn:ClientInfoNamespace";
  /ns:People/ns:Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 7-4: Updating an element value in a typed XML instance.

SQL Server does not support the use of the **`text()`** function for typed columns. If you use it, SQL Server will return an error. This, of course, is opposite from what happens with untyped columns, so when you use the **`modify()`** method to update element values, you need to be aware of this difference.

Otherwise, there are no other surprises when working with typed columns. As long as you declare your namespace correctly and specify the alias reference (in this case, **ns:**), you should have no problem, and your **SELECT** statement should return results similar to those shown in Listing 7-5. As you can see, the book title has been updated to the new value.

```
<ns:Person xmlns:ns="urn:ClientInfoNamespace" id="1234">
  <ns:FirstName>John</ns:FirstName>
  <ns:LastName>Doe</ns:LastName>
  <ns:FavoriteBooks>
    <!-- Books rated on scale 1-5 -->
    <ns:Book rating="5">The Catcher in the Rye</ns:Book>
  </ns:FavoriteBooks>
</ns:Person>
```

Listing 7-5: The updated element value in the typed XML instance.

You can just as easily update an attribute value as you can an element value. In your XQuery expression, specify a path that targets the specific attribute. For example, the XQuery expression in Listing 7-6 targets the **<Book>** element's **rating** attribute. Notice that you simply append the name of attribute—along with the "at" (**@**) symbol—onto the path expression.

```
UPDATE ClientInfo
SET Info_untyped.modify(
   'replace value of
   (/People/Person[@id=1234]/FavoriteBooks/Book/@rating)[1]
   with "4.5" ')
WHERE ClientID = 1;

SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 7-6: Updating an attribute value in an untyped XML instance.

After you've identified the target attribute, you can then specify a value expression that provides a new value for that attribute. In this case, the new value is **4.5**, which is confirmed in the results returned by the **SELECT** statement (shown in Listing 7-7). As you can see, the new value has been assigned to the attribute.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks>
    <!-- Books rated on scale 1-5 -->
    <Book rating="4.5">The Catcher in the Rye</Book>
  </FavoriteBooks>
</Person>
```

Listing 7-7: The updated attribute value in the untyped XML instance.

As is to be expected, the process of updating an attribute column in a typed column is similar to an untyped column, except for having to provide the namespace information. Listing 7-8 shows the **`UPDATE`** statement needed to update the **`rating`** attribute in the **`Info_typed`** column. Notice that the XML DML expression includes the namespace declaration and uses the namespace alias in all the element references. Otherwise, the basic components are the same.

```
UPDATE ClientInfo
SET Info_typed.modify(
  'declare namespace ns="urn:ClientInfoNamespace";
   replace value of
   (/ns:People/ns:Person[@id=1234]/ns:FavoriteBooks/ns:Book/@
rating)[1]
   with 4.5 ')
WHERE ClientID = 1;

SELECT Info_typed.query(
  'declare namespace ns="urn:ClientInfoNamespace";
   /ns:People/ns:Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 7-8: Updating an attribute value in a typed XML instance.

Once again, if we run the **`SELECT`** statement appended to the listing, we'll find that the attribute value has been updated to **`4.5`**, as shown in Listing 7-9.

```
<ns:Person xmlns:ns="urn:ClientInfoNamespace" id="1234">
  <ns:FirstName>John</ns:FirstName>
  <ns:LastName>Doe</ns:LastName>
  <ns:FavoriteBooks>
    <!-- Books rated on scale 1-5 -->
    <ns:Book rating="4.5">The Catcher in the Rye</ns:Book>
  </ns:FavoriteBooks>
</ns:Person>
```

Listing 7-9: The updated attribute value in the typed XML instance.

In the examples we've looked at so far, our value expression has been a literal value enclosed in double quotes. However, that expression can be far more complex. In the example shown in Listing 7-10, the value expression in the second **`UPDATE`** statement is an **`if`**…**`then`**…**`else`** expression that sets the value of the **`recommend`** attribute based on the value of the **`rating`** attribute.

```
UPDATE ClientInfo
SET Info_untyped.modify(
  'insert attribute recommend {"true/false"}
   into
   (/People/Person[@id=1234]/FavoriteBooks/Book)[1] ')
WHERE ClientID = 1;

UPDATE ClientInfo
SET Info_untyped.modify(
  'replace value of
   (/People/Person[@id=1234]/FavoriteBooks/Book/@recommend)[1]
   with (
     if (/People/Person[@id=1234]/FavoriteBooks/Book[1]/@rating >
4)
     then "true"
     else "false") ')
WHERE ClientID = 1;

SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 7-10: Using conditional logic to update an attribute value in an untyped XML instance.

The first **UPDATE** statement adds the **recommend** attribute to the **<Book>** element and sets its initial value to **true/false**. The second **UPDATE** statement then modifies the attribute's value. The beginning of the XML DML expression in that statement is similar to what you've seen in previous examples. After the **replace value of** keywords, an XQuery expression identifies the target attribute, **recommend**. This expression is then followed by the **with** keyword. Everything after that keyword, enclosed in parentheses, is the value expression.

The value expression begins with the **if** clause, which specifies that the **rating** attribute must have a value greater than **4** in order for the clause's condition to evaluate to true. If the condition does evaluate to true, the value of the **recommend** attribute is set to **true**, as specified in the **then** clause. Otherwise, the **recommend** value is set to **false**, as specified in the **else** clause. In other words, the **rating** attribute must have a value greater than **4** in order for the **recommend** attribute is set to **true**, otherwise the attribute is set to **false**.

Because the **rating** attribute currently has a value of **4.5**, the **recommend** attribute will be set to **true** when you run the **UPDATE** statement. You can verify these changes by viewing the results of the **SELECT** statement, which are shown in Listing 7-11. As you can see, the **recommend** attribute has been added to the **<Book>** element and the attribute's value has been set to **true**.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks>
    <!-- Books rated on scale 1-5 -->
    <Book rating="4.5" recommend="true">The Catcher in the Rye</
Book>
  </FavoriteBooks>
</Person>
```

Listing 7-11: The updated attribute value in the untyped XML instance.

You can achieve the same results for the typed column by providing the expected namespace information, as shown in Listing 7-12. Notice that the namespace is declared and referenced throughout. That includes the **if** clause in the value expression of the second **UPDATE** statement.

```
UPDATE ClientInfo
SET Info_typed.modify(
  'declare namespace ns="urn:ClientInfoNamespace";
   insert attribute recommend {"true/false"}
   into
   (/ns:People/ns:Person[@id=1234]/ns:FavoriteBooks/ns:Book)[1] ')
WHERE ClientID = 1;

UPDATE ClientInfo
SET Info_typed.modify(
  'declare namespace ns="urn:ClientInfoNamespace";
   replace value of
   (/ns:People/ns:Person[@id=1234]/ns:FavoriteBooks/ns:Book/@
recommend)[1]
   with (
     if (/ns:People/ns:Person[@id=1234]/ns:FavoriteBooks/
ns:Book[1]/@rating > 4)
 then "true"
 else "false") ')
WHERE ClientID = 1;

SELECT Info_typed.query(
  'declare namespace ns="urn:ClientInfoNamespace";
   /ns:People/ns:Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 7-12: Using conditional logic to update an attribute value in a typed XML instance.

Once again, if you run the **SELECT** statement appended to the example, your results will reflect the new attribute and its updated value, as shown in Listing 7-13, overleaf.

```
<ns:Person xmlns:ns="urn:ClientInfoNamespace" id="1234">
  <ns:FirstName>John</ns:FirstName>
  <ns:LastName>Doe</ns:LastName>
  <ns:FavoriteBooks>
    <!-- Books rated on scale 1-5 -->
    <ns:Book rating="4.5" recommend="true">The Catcher in the Rye</
ns:Book>
  </ns:FavoriteBooks>
</ns:Person>
```

Listing 7-13: The updated attribute value in the typed XML instance.

As before, you can see that the **`recommend`** attribute has been added to the **`<Book>`** element and the attribute's value has been set to **`true`**.

Conclusion

Using the **`modify()`** method to update data in an XML column requires that you provide the necessary keywords and define the XQuery and value expressions in your XML DML expression. You can use this approach to update both element and attribute values in either typed or untyped XML instances. When updating data in a typed column, you must provide the necessary namespace information, just like you saw it done in previous Levels. In Level 8, you'll learn how to use the **`modify()`** method to delete elements and attributes from an XML instance.

Level 8 – Deleting Data from an XML Instance

In Levels 6 and 7, you learned how to use the **`modify()`** method (available to the **`XML`** data type) to insert and update data in an XML instance. As you saw, the method provides several options that let you control how you manipulate the data.

In this Level, you'll learn how to use the **`modify()`** method to delete data from an XML instance. As is the case when inserting or updating data, the method takes an XML Data Modification Language (XML DML) expression as an argument when deleting the data. XML DML is an extension of the XQuery language that lets you insert, update, and delete data in an XML instance.

> ***Note***
>
> *As with previous Levels, we can touch upon only some of the XML DML and XQuery elements in this Level. For a more thorough understanding of XQuery and how it's implemented in SQL Server, see the MSDN XQuery language reference. For more details about XML DML, see the MSDN article "XML Data Modification Language (XML DML)".*

This Level includes several examples that demonstrate how to delete data from both typed and untyped XML instances. If you want to try these examples, you'll first need to run the Transact-SQL code shown in Listing 8-1. The code creates the **`ClientDB`** database, adds the **`ClientInfoCollection`** XML schema collection to the database, and then creates and populates the **`ClientInfo`** table.

```
USE master;
GO

IF DB_ID('ClientDB') IS NOT NULL
DROP DATABASE ClientDB;
GO

CREATE DATABASE ClientDB;
GO

USE ClientDB;
```

```
GO

IF EXISTS(
  SELECT * FROM sys.xml_schema_collections
  WHERE name = 'ClientInfoCollection')
DROP XML SCHEMA COLLECTION ClientInfoCollection;
GO

CREATE XML SCHEMA COLLECTION ClientInfoCollection AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="urn:ClientInfoNamespace"
targetNamespace="urn:ClientInfoNamespace"
elementFormDefault="qualified">
  <xsd:element name="People">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Person" minOccurs="1"
maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="FirstName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="LastName" type="xsd:string"
minOccurs="1" maxOccurs="1" />
              <xsd:element name="FavoriteBooks" minOccurs="0"
maxOccurs="1">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="Book" minOccurs="0"
maxOccurs="5">
                      <xsd:complexType>
                        <xsd:simpleContent>
                          <xsd:extension base="xsd:string">
                            <xsd:attribute name="rating"
type="xsd:decimal" />
                          </xsd:extension>
                        </xsd:simpleContent>
                      </xsd:complexType>
                    </xsd:element>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:integer"
```

```
use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>';
GO

IF OBJECT_ID('ClientInfo') IS NOT NULL
DROP TABLE ClientInfo;
GO

CREATE TABLE ClientInfo
(
  ClientID INT PRIMARY KEY IDENTITY,
  Info_untyped XML,
  Info_typed XML(ClientInfoCollection)
);

INSERT INTO ClientInfo (Info_untyped, Info_typed)
VALUES
(
  '<?xml version="1.0" encoding="UTF-8"?>
  <People>
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
      <FavoriteBooks>
        <!-- Books rated on scale 1-5 -->
        <Book rating="5">Slaughterhouse-Five</Book>
      </FavoriteBooks>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>',
  '<?xml version="1.0" encoding="UTF-8"?>
  <People xmlns="urn:ClientInfoNamespace">
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
      <FavoriteBooks>
        <!-- Books rated on scale 1-5 -->
```

```
        <Book rating="5">Slaughterhouse-Five</Book>
      </FavoriteBooks>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>'
);
```

Listing 8-1: Setting up the test environment for the examples in this Level.

The code in Listing 8-1 is similar to what you saw in Level 7. In addition to creating the schema and table, it also includes an **INSERT** statement that adds a row of data to the **ClientInfo** table. The row contains two XML instances, one that targets the untyped column and one that targets the typed column. Otherwise, the elements, attributes, and values that make up each instance are the same.

I created the code in Listing 8-1 on a local instance of SQL Server 2012. I then created the examples in the following section in that same environment. Once you've set up this environment on your system, you'll be ready to try out these examples.

Deleting data from an XML instance

You can use the **modify()** method to delete specific components from an XML instance. Using the method to delete data is for the most part easier than using it to insert or update data. You simply specify the **delete** keyword, followed by an XQuery expression that identifies the XML component to be deleted. The following syntax shows how to use the **modify()** method to delete XML data:

```
db_object.modify('delete xquery_exp')
```

Notice that, as you saw when inserting and deleting data, the XML DML expression is enclosed in single quotes and parentheses. The XML DML expression itself is very straightforward.

To demonstrate how easy it is to delete data, let's start by removing an attribute from the untyped XML instance in our **ClientInfo** table. In Listing 8-2, the **UPDATE** statement uses the **modify()** method to delete data from the **Info_untyped** column.

```
UPDATE ClientInfo
SET Info_untyped.modify(
  'delete
   /People/Person[@id=1234]/FavoriteBooks/Book[1]/@rating ')
WHERE ClientID = 1;

SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 8-2: Deleting an attribute from an untyped XML instance.

We begin our XML DML expression by specifying the **delete** keyword. This is followed by an XQuery path expression that specifies the attribute to be deleted. Notice that the path specifies the first instance of the **<Book>** element and the **rating** attribute within that element. The attribute name, which is preceded with the at (**@**) symbol, follows the element name within the path.

Because the **rating** attribute within the **<Book>** element is being specified, the attribute will be removed from the XML instance when you run the **UPDATE** statement. Listing 8-3 shows the results returned by the **SELECT** statement tagged onto the example in Listing 8-2.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks>
    <!-- Books rated on scale 1-5 -->
    <Book>Slaughterhouse-Five</Book>
  </FavoriteBooks>
</Person>
```

Listing 8-3: The **<Book>** element without the **rating** attribute in the untyped XML.

As the listing shows, the **rating** attribute is no longer included in the **<Book>** element. You can see that deleting an attribute from an untyped XML instance is pretty painless. And it's almost just as easy to delete an attribute from a typed instance. The main difference, of course, is that you must specify the necessary namespace declaration and references. Listing 8-4 demonstrates how this is done.

```
UPDATE ClientInfo
SET Info_typed.modify(
  'declare namespace ns="urn:ClientInfoNamespace";
   delete
   /ns:People/ns:Person[@id=1234]/ns:FavoriteBooks/ns:Book[1]/@
rating ')
WHERE ClientID = 1;

SELECT Info_typed.query(
  'declare namespace ns="urn:ClientInfoNamespace";
   /ns:People/ns:Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 8-4: Deleting an attribute from a typed XML instance.

If you tried examples in previous Levels that access typed XML instances, there should be no surprises here. You divide your XML DML expression into two parts, separated by a semicolon. In the first part, you declare you namespace and assign an alias to that namespace. In this case, the alias is **ns**. You then use that alias, along with a colon, in the element references in the second part of your XML DML expression. However, as you can see, you don't have to include the namespace reference for your attribute. Listing 8-5 shows the results now returned by the **SELECT** statement.

```
<ns:Person xmlns:ns="urn:ClientInfoNamespace" id="1234">
  <ns:FirstName>John</ns:FirstName>
  <ns:LastName>Doe</ns:LastName>
  <ns:FavoriteBooks>
    <!-- Books rated on scale 1-5 -->
    <ns:Book>Slaughterhouse-Five</ns:Book>
  </ns:FavoriteBooks>
</ns:Person>
```

Listing 8-5: The **<Book>** element without the **rating** attribute in the typed XML.

Notice that the **rating** attribute has been removed from the **<Book>** element. Also notice that, whether working with typed or untyped columns, when you remove a component such as an attribute, you're also removing any data values associated with that component.

In addition to removing attributes, you can remove components such as comments from an XML instance. To do so, you tag the **comment()** function onto the XQuery expression that identifies the element containing the comment. For example, in Listing 8-6 I use the **comment()** function to remove the comment from the **<FavoriteBooks>** element.

```
UPDATE ClientInfo
SET Info_untyped.modify(
  'delete
   /People/Person[@id=1234]/FavoriteBooks/comment()[1] ')
WHERE ClientID = 1;

SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 8-6: Removing a comment from an untyped column.

Because an element can contain multiple comments, you should add to the end of your XQuery expression a numerical reference that points to the comment that should be deleted. In this case, I use **[1]** to designate that the first comment should be deleted. (There is only one comment in the XML instance.) The **SELECT** statement now returns the results shown in Listing 8-7.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks>
    <Book>Slaughterhouse-Five</Book>
  </FavoriteBooks>
</Person>
```

Listing 8-7: The **<FavoriteBooks>** element without the comment.

You can, of course, just as easily delete a comment from a typed XML instance, as long as you include the proper namespace declaration and references, as shown in Listing 8-8.

```
UPDATE ClientInfo
SET Info_typed.modify(
  'declare namespace ns="urn:ClientInfoNamespace";
   delete
```

```
   /ns:People/ns:Person[@id=1234]/ns:FavoriteBooks/comment()[1] ')
WHERE ClientID = 1;

SELECT Info_typed.query(
  'declare namespace ns="urn:ClientInfoNamespace";
   /ns:People/ns:Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 8-8: Removing a comment from a typed column.

Not surprisingly, the XML DML expression is divided into two parts. The first part is the declaration, and the second part contains the **`delete`** keyword and XQuery expression, with the proper namespace references included. Listing 8-9 shows the results the **`SELECT`** statement now returns.

```
<ns:Person xmlns:ns="urn:ClientInfoNamespace" id="1234">
  <ns:FirstName>John</ns:FirstName>
  <ns:LastName>Doe</ns:LastName>
  <ns:FavoriteBooks>
    <ns:Book>Slaughterhouse-Five</ns:Book>
  </ns:FavoriteBooks>
</ns:Person>
```

Listing 8-9: The **`<FavoriteBooks>`** element without the comment.

Now let's look at how to remove an element from an XML instance. To do so, your XQuery expression must identify the element that should be deleted, as shown in Listing 8-10.

```
UPDATE ClientInfo
SET Info_untyped.modify(
  'delete
   /People/Person[@id=1234]/FavoriteBooks/Book[1] ')
WHERE ClientID = 1;

SELECT Info_untyped.query(
  '/People/Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 8-10: Removing an element from an untyped column.

Notice that, as with the previous examples, the XML DML expression starts with the **delete** keyword, following by the XQuery expression. As that expression shows, we're removing the first instance of the **<Book>** child element within the **<FavoriteBooks>** element. Listing 8-11 shows the results returned by the **SELECT** statement.

```
<Person id="1234">
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <FavoriteBooks />
</Person>
```

Listing 8-11: The **<FavoriteBooks>** element without the **<Book>** child element.

As you would expect, the **<Book>** element has been removed, and the **<FavoriteBooks>** element no longer contains child elements. If you want to remove the same element from the typed column, you simply include the necessary namespace declaration and references, as shown in Listing 8-12.

```
UPDATE ClientInfo
SET Info_typed.modify(
  'declare namespace ns="urn:ClientInfoNamespace";
   delete
   /ns:People/ns:Person[@id=1234]/ns:FavoriteBooks/ns:Book[1] ')
WHERE ClientID = 1;

SELECT Info_typed.query(
  'declare namespace ns="urn:ClientInfoNamespace";
  /ns:People/ns:Person[@id=1234]')
FROM ClientInfo
WHERE ClientID = 1;
```

Listing 8-12: Removing an element from a typed column.

Again, be sure to include the namespace alias and colon when referencing the elements within your XQuery expression. The **SELECT** statement now returns the results shown in Listing 8-13.

```
<ns:Person xmlns:ns="urn:ClientInfoNamespace" id="1234">
  <ns:FirstName>John</ns:FirstName>
  <ns:LastName>Doe</ns:LastName>
  <ns:FavoriteBooks />
</ns:Person>
```

Listing 8-13: The **<FavoriteBooks>** element without the **<Book>** child element.

As you saw with the untyped column, the **<Book>** element has been removed and the **<FavoriteBooks>** element no longer contains any child elements.

Conclusion

This Level explained how to use the **modify()** method to delete data from typed and untyped XML instances. As the Level demonstrated, you must pass an XML DML expression as an argument to the method. That expression must include the **delete** keyword, along with an XQuery expression that defines the XML component to be deleted.

Up to this point, our discussions about XML have generally centered around the standard ways XML is implemented in SQL Server, primarily as columns or variables configured with the **XML** data type. However, XML can also play a role when working with such objects as views, functions, defaults, computed columns, and check constraints. As we progress through this book, you'll learn how to take what we've covered up till now and apply that information to other objects in a SQL Server database.

Level 9 – Creating XML-based Functions

In previous Levels, we looked at the methods available to the **XML** data type that let you view and modify specific components of an XML instance. We also reviewed a number of examples that demonstrated different ways you can use the methods. However, those examples were limited primarily to basic Transact-SQL queries. But you can also use the methods in such database objects as user-defined functions, stored procedures, and views. In this Level, we'll look at how to use **XML** methods within user-defined functions to return XML fragments and values from your target XML instance.

When incorporating **XML** methods into your functions, you create them in much the same way you would any function. If you're unfamiliar with how to create functions, refer to SQL Server Books Online for details about the different function types and how to define them. This Level is concerned primarily with the XML-related components.

The sections to follow include a number of examples that demonstrate how to use **XML** methods within your functions. If you plan to try out these examples, you should first run the code shown in Listing 9-1. It creates the **ClientDB** database and the **ClientInfo** table within that database. The code then inserts sample data into the table.

```
USE master;
GO

IF DB_ID('ClientDB') IS NOT NULL
DROP DATABASE ClientDB;
GO

CREATE DATABASE ClientDB;
GO

USE ClientDB;
GO

IF OBJECT_ID('ClientInfo') IS NOT NULL
DROP TABLE ClientInfo;
GO

CREATE TABLE ClientInfo
```

```
(
  ClientID INT PRIMARY KEY IDENTITY,
  Info XML
);

INSERT INTO ClientInfo (Info)
VALUES
(
'<People>
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="2468">
      <FirstName>John</FirstName>
      <LastName>Smith</LastName>
    </Person>
    <Person id="1357">
      <FirstName>Jane</FirstName>
      <LastName>Smith</LastName>
    </Person>
  </People>'
);
```

Listing 9-1: Setting up the initial test environment.

If you've worked through the previous Levels, you'll notice that the code shown in Listing 9-1 doesn't include a typed **`XML`** column in the **`ClientInfo`** table. For the purposes of demonstrating how to use the **`XML`** methods within user-defined functions, this Level focuses on how that is done, rather than distinguishing between typed and untyped **`XML`** objects. Just know that if you're working with typed XML, you should follow the same processes you saw in previous Levels to access the data. You must declare your namespace and provide the proper namespace references in your element paths when referencing an XML instance in an **`XML`** object. That said, let's get started on how to create functions that access XML data.

Creating XML-based functions

Before we get into using an **XML** method within a user-defined function, let's first look at a function that returns an XML instance. The code in Listing 9-2 creates the **udfClient** function, which retrieves the XML instance from the **Info** column of the **ClientInfo** table, based on the specified client ID. The function returns the instance as a single **XML** value.

```
IF OBJECT_ID('udfClient') IS NOT NULL
DROP FUNCTION udfClient;
GO

CREATE FUNCTION udfClient (@ClientID INT)
RETURNS XML
AS BEGIN
RETURN
(
  SELECT Info
  FROM ClientInfo
  WHERE ClientID = @ClientID
)
END;
GO
```

Listing 9-2: Creating a function that returns XML data.

The **udfClient** function is itself very straightforward. It includes a **SELECT** statement whose **WHERE** clause limits the results to the row associated through the **ClientID** value, as specified by the **@ClientID** parameter value passed into the function when calling it. Notice that the **RETURNS** clause specifies the **XML** data type. That means the **SELECT** statement must return a scalar value that conforms to that data type, which it does.

Once you've created the function, you can use a **SELECT** statement, such as the one shown in Listing 9-3, to test that the function returns the results you expect. Note that you must specify the schema name when calling a user-defined function, even if that function was created within the **dbo** schema. You must also specify a value to pass in as an argument that identifies the client ID (in this case, **1**).

```
SELECT dbo.udfClient(1);
```

Listing 9-3: Testing the **udfClient** user-defined function.

When you run the **SELECT** statement, it should return the entire XML instance that you inserted into the table when you first set up your test environment. You can refer back to Listing 9-1 to verify that the statement returns the correct data.

Because the function returns an **XML** value, you can use the **XML** methods when calling the function. For example, the **SELECT** statement in Listing 9-4 uses the **query()** method to return the **<FirstName>** element for the person whose **id** attribute value is **1234**.

```
SELECT dbo.udfClient(1).query(
  '/People/Person[@id=1234]/FirstName');
```

Listing 9-4: Using the **query()** method when calling your function.

Notice that when calling the **udfClient** function, you add a period after the function's closing parenthesis, followed by the method name. You then pass in the necessary XQuery expression as an argument to the method, just like you saw in previous Levels. Now the function returns only the value **<FirstName>John</FirstName>**.

If you want to return only an element's value, and not its tags, you can instead use the **value()** method. Just remember that, when calling this method, you must provide two arguments: the XQuery expression and the data type of the returned value. For example, the **SELECT** statement shown in Listing 9-5 retrieves the first name of the person with an **id** attribute value of **5678**, so the second argument to the method specifies the **varchar(20)** data type.

```
SELECT dbo.udfClient(1).value(
  '(/People/Person[@id=5678]/FirstName)[1]',
  'varchar(20)') AS FirstName;
```

Listing 9-5: Using the **value()** method when calling your function.

Now the statement returns the value **Jane**. As you can see, because the **udfClient** function returns an **XML** value, you can use the **XML** methods as you would when specifying them with columns or other **XML** objects.

Of course, creating a user-defined function that returns an entire XML instance will probably be useful only in rare circumstances because the complexity lies when you call the function, not when you define it. Chances are, if you plan to use a function to access XML data, you'll want to use one or more of the **XML** methods within the function so you can persist complex queries to the database.

Let's look at an example that demonstrates how this works. Listing 9-6 shows the code used to create the **udfFullName** function, which returns a list of full names for a specific client in the **ClientInfo** table.

```
IF OBJECT_ID('udfFullName') IS NOT NULL
DROP FUNCTION udfFullName;
GO

CREATE FUNCTION udfFullName (@ClientID INT)
RETURNS TABLE
AS
RETURN
(
  SELECT Person.value(
    'concat(./FirstName[1], " ", ./LastName[1])',
    'varchar(50)') AS FullName
  FROM ClientInfo CROSS APPLY
    Info.nodes('/People/Person') AS People(Person)
  WHERE ClientID = @ClientID
);
GO
```

Listing 9-6: Using the **value()** and **nodes()** methods within your function.

Notice that the **RETURNS** clause in the **CREATE FUNCTION** statement specifies that the results returned by the **SELECT** statement conform to the **TABLE** data type, which is a special type used to store a tabular result set. That means our **SELECT** statement can return any number of rows and columns, rather than only a scalar value.

The **SELECT** statement itself includes the **value()** and **nodes()** methods to retrieve a list of names. The **FROM** clause uses the **nodes()** method to parse the XML instance. The clause then uses the **CROSS APPLY** operator to join the results returned by the **nodes()** method to the **ClientInfo** table.

The **SELECT** clause contains the **value()** method, which retrieves the names from the joined results, and the method's XQuery expression uses the **concat()** function to concatenate the values from the **<FirstName>** and **<LastName>** elements.

Once you've defined your function, you can then use a **SELECT** statement similar to the one shown in Listing 9-7 to test that the function returns the expected results.

```
SELECT * FROM dbo.udfFullName(1);
```

Listing 9-7: Testing the **udfFullName** user-defined function.

Because the **udfFullName** function returns data as a table, you can use the function only where a table expression is accepted which, in this case, is the **FROM** clause. Listing 9-8 shows the results returned by the **SELECT** statement.

```
FullName
-----------
John Doe
Jane Doe
John Smith
Jane Smith
```

Listing 9-8: Results returned by the **udfFullName** function.

As you can see, a function makes it easy to persist complex code to the database so you can call it when you need it, which can be particularly handy when calling the function from within other database objects. So let's look at how to do just that.

Using XML-based functions in computed columns

A computed column is one in which an expression is used to generate the values for that particular column. The expression can reference other columns within the table, including **XML** columns.

However, it's rare that you'll want to create a computed column based on an entire XML instance, unless you simply plan to convert that column to another type. In most cases, you'll probably want to use only a value or two from within the XML. The problem is, SQL Server does not let you use **XML** methods in a computed column. The way around this is to create a function that uses the necessary methods, and then call the function from within your computed column expression.

For example, suppose we want to add a column to our **ClientInfo** table that calculates the number of **<Person>** elements within the XML instance associated with a particular row. We would start by first creating a function similar to the one shown in Listing9-9. The function uses the **value()** method and the XQuery **count()** function to return an **INT** value that shows the number of **<Person>** elements.

```
IF OBJECT_ID('udfPersonCount') IS NOT NULL
DROP FUNCTION udfPersonCount;
GO

CREATE FUNCTION udfPersonCount (@ClientID INT)
RETURNS INT
AS BEGIN
RETURN
(
  SELECT Info.value('count(/People/Person)', 'int')
  FROM ClientInfo
  WHERE ClientID = @ClientID
)
END;
GO
```

Listing 9-9: Creating a function that returns the number of people within an XML document.

As you can see, when you call the function, you pass in the client ID. The function then returns the element count for the row associated with that ID.

You can test that the function works by using a **SELECT** statement, as shown in Listing 9-10. In this case, the function should return a scalar value of **4**.

```
SELECT dbo.udfPersonCount(1) AS PersonCount;
```

Listing 9-10: Testing the **udfPersonCount** user-defined function.

Now let's update the **ClientInfo** table to include the computed column. Listing 9-11 shows an **ALTER TABLE** statement that adds the **PersonCount** column to the table. Notice that the **ADD** clause calls the **udfPersonCount** function and passes in the **ClientID** column as an argument.

```
ALTER TABLE ClientInfo
ADD PersonCount AS dbo.udfPersonCount(ClientID);
GO
```

Listing 9-11: Using the **udfPersonCount** function to create a calculated column.

To verify your computed column, you need only retrieve the **PersonCount** column from the **ClientInfo** table, as shown in Listing 9-12. In this case, I've included a **WHERE** clause that limits the results to the row with a client ID of **1**.

```
SELECT PersonCount FROM ClientInfo
WHERE ClientID = 1;
```

Listing 9-12: Verifying the data in the **PersonCount** calculated column.

As to be expected, the **SELECT** statement returns the value **4**. If the table had included additional rows, and you retrieved the **PersonCount** data from one of those rows, the value would be specific to the number of **<Person>** elements in that row. Now let's look at how to incorporate the function in a check constraint.

Using XML-based functions in check constraints

As with computed columns, you cannot use the **XML** methods within a check constraint expression, but you can include a function in the expression, and that function can include the methods.

For example, suppose you want to ensure that a row can be added to the **ClientInfo** table only if the **XML** document contains more than one instance of the **<People>** element. You can create a check constraint that uses the **udfPersonCount** function in the constraint's expression, as shown in Listing 9-13.

```
ALTER TABLE ClientInfo
WITH NOCHECK ADD CONSTRAINT ck_count
  CHECK (dbo.udfPersonCount(ClientID) > 1);
```

Listing 9-13: Using the **udfPersonCount** function in a check constraint.

As you can see, the **ALTER TABLE** statement adds the **ck_count** check constraint. The constraint's expression compares the output from the **udfPersonCount** function to the value **1**. For the expression to evaluate to true, the XML document must contain more than one instance of the **<People>** element.

Once we've added the check constraint to our table, we can test it by trying to add an XML document that contains only one **<Person>** element, as shown in Figure 9-14.

```
INSERT INTO ClientInfo (Info)
VALUES
(
 '<People>
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>'
);
```

Listing 9-14: Inserting a single **<Person>** instance into the **Info** column.

Not surprisingly, this statement returns an error message (shown in Listing 9-15) because the **INSERT** statement causes the check constraint expression to evaluate to false. So the row cannot be inserted into the table.

```
The INSERT statement conflicted with the CHECK constraint "ck_
count". The conflict occurred in database "ClientDB", table "dbo.
ClientInfo", column 'ClientID'.
The statement has been terminated.
```

Listing 9-15: Error message returned by SQL Server.

Now let's try to add an XML document with two **<People>** elements. Listing 9-16 shows the XML document with the additional element.

```
INSERT INTO ClientInfo (Info)
VALUES
(
 '<People>
    <Person id="1234">
      <FirstName>John</FirstName>
      <LastName>Doe</LastName>
    </Person>
    <Person id="5678">
      <FirstName>Jane</FirstName>
      <LastName>Doe</LastName>
    </Person>
  </People>'
);
-- 1 row inserted
```

Listing 9-16: Inserting two **<Person>** instances into the **Info** column.

This time you should receive a message saying that one row has been inserted into the table. You can confirm this by running the **SELECT** statement in Figure 9-17.

```
SELECT * FROM ClientInfo;
```

Listing 9-17: Verifying that a second row as been added to the **ClientInfo** table.

As to be expected, the **SELECT** statement returns two rows, the original row and the new row. The original row had four instances of the **<People>** element, and the new one has two instances. Therefore, the calculated column **PersonCount** should contain the values **4** and **2**, respectively.

Conclusion

Being able to use the XML methods within your functions can be a handy tool, regardless of how you plan to use those functions. Yet, as this Level has demonstrated, such functions are particularly useful when implementing calculated columns or check constraints because neither supports the direct use of the **XML** methods. However, other objects do permit their use. In a later level, we'll cover how to incorporate the methods into views and stored procedures.

Many of the principles we covered in this Level will apply to the next one, but that Level will help to round out our discussion on the **`XML`** methods, and will give you a more complete picture of the various ways you can access data from an XML instance. In the meantime, for additional information on the topics we discussed in this Level, be sure to refer to SQL Server Books Online.

Level 10 – Converting XML Data

Previous Levels of this book covered different ways to work with XML in SQL Server, with much of the focus on the **XML** methods used to access and update components within an XML instance. One of those methods, **value()**, lets you return a specific element or attribute value as a T-SQL data type such as **VARCHAR**. For the most part, this represented the only discussion we've had about converting XML data. However, there might be times when you want to convert an entire XML instance or fragment to a character data type or convert a string value to XML.

In this Level, we look at how to convert string values to XML and how to convert XML to character types. We'll be using variables to demonstrate how these conversions work, so there's no setup required to try out the examples, other than to have access to a SQL Server instance. I wrote the examples on a local instance of SQL Server 2012, but you are by no means limited to this environment. In addition, the methods shown here to convert data can easily be applied to **XML** columns. But note that you can convert XML data to and from character types only, such as **CHAR** or **VARCHAR**. You cannot, for example, convert XML directly to the **DATETIME** type.

Converting string values to XML data

When converting a string value to XML, you can do so implicitly or explicitly, whether you're using a literal value or accessing the value through an object configured with a character type. One of the most basic examples of an implicit conversion is to assign a literal string value to an **XML** object, as I do in Listing 10-1.

```
DECLARE @xmlPerson XML;
SET @xmlPerson = '<People><Person>John Doe</Person></People>';
SELECT @xmlPerson;
```

Listing 10-1: Implicitly converting a string value to XML.

First, I declare the **@xmlPerson** variable with the **XML** data type. I then assign the **<People>** element and its contents to the variable. The element in this case is simply a string value that I assign to the variable. SQL Server automatically converts the literal value to XML. When I then use a **SELECT** statement to retrieve the value from the variable, it's returned as an XML fragment, as shown in Listing 10-2.

```
<People><Person>John Doe</Person></People>
```

Listing 10-2: The XML fragment returned by the T-SQL query.

As you can see, an implicit conversion is fairly straightforward, and it's just as easy to convert the value in an **XML** object. For instance, in the example shown in Listing 10-3, I convert the value assigned to a variable configured with the **NVARCHAR** data type.

```
DECLARE @strPerson NVARCHAR(100);
DECLARE @xmlPerson XML;
SET @strPerson = '<People><Person>John Doe</Person></People>';
SET @xmlPerson = @strPerson;
SELECT @xmlPerson;
```

Listing 10-3: Implicitly converting a character type to XML.

First, I declare the **@strPerson** variable with the **NVARCHAR** type, and then I declare the **@xmlPerson** variable with the **XML** type. Next, I assign the **<People>** element and its contents (defined as a string literal) to the **@strPerson** variable. Then I simply assign the **@strPerson** value to **@xmlPerson**. Once again, SQL Server automatically converts the data from the **NVARCHAR** type to the **XML** type. The **SELECT** statement returns the same results as the **SELECT** statement in the previous example. (Refer back to Listing 10-2.)

We could just as easily have assigned a different character type to the **@strPerson** variable. For instance, the example shown in Listing 10-4 assigns the **VARCHAR(MAX)** data type to the variable.

```
DECLARE @strPerson VARCHAR(MAX);
DECLARE @xmlPerson XML;
SET @strPerson = '<People><Person>John Doe</Person></People>';
SET @xmlPerson = @strPerson;
SELECT @xmlPerson;
```

Listing 10-4: Implicitly converting a character type to XML.

Once again, SQL Server automatically converts the character value to **XML**, and the **SELECT** statement returns the same results as we saw in the previous examples.

If you plan to port your SQL scripts to another database system, you can't assume that the system will support implicit conversions in the same way as SQL Server. In such circumstances, you should use the **CAST** function to explicitly convert your string values to **XML**. The **CAST** function conforms to ANSI specifications and consequently is supported by most database systems.

In the example shown in Listing 10-5, I use the **CAST** function to convert the **@strPerson** value to **XML** before assigning the value to the **@xmlPerson** variable.

```
DECLARE @strPerson VARCHAR(MAX);
DECLARE @xmlPerson XML;
SET @strPerson = '<People><Person>John Doe</Person></People>';
SET @xmlPerson = CAST(@strPerson AS XML);
SELECT @xmlPerson;
```

Listing 10-5: Using the **CAST** function to explicitly convert a character type.

As you can see, the **CAST** function takes only two arguments, separated by the **AS** keyword. The first is the source value, in this case, the **@strPerson** variable, and the second argument is the target data type—**XML**. Once again, the **SELECT** statement returns the same XML element as in the previous examples.

We can easily achieve the same results using the **CONVERT** function, but we need to structure the arguments differently, as shown in Listing 10-6.

```
DECLARE @strPerson VARCHAR(MAX);
DECLARE @xmlPerson XML;
SET @strPerson = '<People><Person>John Doe</Person></People>';
SET @xmlPerson = CONVERT(XML, @strPerson);
SELECT @xmlPerson;
```

Listing 10-6: Using the **CONVERT** function to explicitly convert a character type.

In this case, we first specify the target data type (**XML**) and then the source value (**@strPerson**), separated by a comma. However, the **CONVERT** function does not port to other systems; it is specific to T-SQL in SQL Server. The only reason you would use the **CONVERT** function is to take advantage of additional options available to the function not available to **CAST**.

Let's look at a couple of examples to better understand how this works. In Listing 10-7, I start by declaring the two variables and assigning a string value to **@strPerson**, as I did in the previous examples. But notice that this time I've add whitespace and line breaks to the string value.

```
DECLARE @strPerson VARCHAR(MAX);
DECLARE @xmlPerson XML;
SET @strPerson = '
<People>
  <Person>John Doe</Person>
</People>';
SET @xmlPerson = CONVERT(XML, @strPerson);
SELECT @xmlPerson;
```

Listing 10-7: Trying to preserve whitespace and line breaks when converting string data to XML.

The whitespace and line breaks have no impact on the XML itself. In fact, when SQL Server converts the string to the **XML** type, it removes the whitespace and line breaks. Consequently, the **SELECT** statement returns the same results as the previous examples, as shown in Listing 10-8.

```
<People><Person>John Doe</Person></People>
```

Listing 10-8: The XML fragment returned without the whitespace and line breaks.

If we want to preserve the whitespace and line breaks, we need to add a third argument to the **CONVERT** function. The SQL Server documentation refers to this as the *style* argument, which is an integer that specifies how to translate the value returned by the expression in the second argument. The styles available are specific to the data type specified in the first argument. For the **XML** type, we have only a few options available. Two of those are **0** and **1**. The **0** option, which is the default, ignores whitespace and line breaks. The **1** option preserves them. Listing 10-9 shows the **CONVERT** function when we include **1** as the third argument.

```
DECLARE @strPerson VARCHAR(MAX);
DECLARE @xmlPerson XML;
SET @strPerson = '
<People>
  <Person>John Doe</Person>
</People>';
SET @xmlPerson = CONVERT(XML, @strPerson, 1);
SELECT @xmlPerson;
```

Listing 10-9: Preserving whitespace and line breaks when converting string data to XML.

As you can see, I've simply added a comma and the **1** argument to the **CONVERT** function. Everything else in the example is the same as the preceding one. However, the **SELECT** statement now returns the XML with the whitespace and line breaks preserved, as shown in Listing 10-10.

```
<People>
  <Person>John Doe</Person>
</People>
```

Listing 10-10: The XML fragment returned with the whitespace and line breaks.

Preserving the whitespace is particularly handy when your XML contains more elements and is subsequently more difficult to read. For instance, Listing 10-11 includes an additional **<Person>** element in the string value.

```
DECLARE @strPerson VARCHAR(MAX);
DECLARE @xmlPerson XML;
SET @strPerson = '
<People>
  <Person>John Doe</Person>
  <Person>Jane Doe</Person>
</People>';
SET @xmlPerson = CONVERT(XML, @strPerson, 1);
SELECT @xmlPerson;
```

Listing 10-11: Preserving whitespace and line breaks when converting string data to XML.

Once again, I've used the **CONVERT** function with the third argument set to **1**. As Listing 10-12 shows, the results have preserved the additional whitespace and line break.

```
<People>
  <Person>John Doe</Person>
  <Person>Jane Doe</Person>
</People>
```

Listing 10-12: The XML fragment returned with an additional element.

Keep in mind, however, as handy as the **CONVERT** function is, in terms of letting you specify how data is converted, the fact that the function cannot be ported to other systems is an important one. If the possibility exists that you will one day need to run your T-SQL scripts against a system other than SQL Server, then you should use the **CAST** function, and avoid both implicit conversions and the **CONVERT** function.

Converting XML values to string data

At times, you might find it handy to convert XML data to string data. For example, you might decide you don't need to use the **XML** data type to store your data, and want to switch over to one of the character data types. However, SQL Server does not support implicit conversions from the **XML** type to a character type. To convert your data in this direction, you must use the **CAST** or **CONVERT** function.

If you do try to implicitly convert XML data, you will receive an error. For instance, in Listing 10-13, I define the same two variables you saw in earlier examples. Only, this time I assign the string value (the **<People>** element) to the **@xmlPerson** variable and then assign that variable to the **@strPerson** variable.

```
DECLARE @xmlPerson XML;
DECLARE @strPerson VARCHAR(MAX);
SET @xmlPerson = '<People><Person>Jane Doe</Person></People>';
SET @strPerson = @xmlPerson;
SELECT @strPerson;
```

Listing 10-13: Trying to implicitly convert XML data to a character type.

When I try to run these statements, SQL Server returns the error shown in Listing 10-14. Notice that the error is at Line 4, which is where I try to implicitly convert the **XML** value to a **VARCHAR(MAX)** value.

```
Msg 257, Level 16, State 3, Line 4
Implicit conversion from data type xml to varchar(max) is not
allowed. Use the CONVERT function to run this query.
```

Listing 10-14: The error message returned when trying to implicitly convert the XML fragment.

This, of course, is an easy fix. Simply use the **CAST** function to implicitly convert the data, as shown in Listing 10-15.

```
DECLARE @xmlPerson XML;
DECLARE @strPerson VARCHAR(MAX);
SET @xmlPerson = '<People><Person>Jane Doe</Person></People>';
SET @strPerson = CAST(@xmlPerson AS VARCHAR(MAX));
SELECT @strPerson;
```

Listing 10-15: Using the **CAST** function to explicitly convert XML data.

Notice that I specify the **CAST** function, with the **@xmlPerson** variable as the first argument and the **VARCHAR(MAX)** data type as the second argument. As expected, the conversion now works without a hitch, and the **SELECT** statement returns the expected results, as shown in Listing 10-16.

```
<People><Person>Jane Doe</Person></People>
```

Listing 10-16: The XML fragment returned by the query.

I can also use the **CONVERT** function to achieve the same results. Listing 10-17 uses the function with the same two arguments used in the previous example for the **CAST** function.

```
DECLARE @xmlPerson XML;
DECLARE @strPerson VARCHAR(MAX);
SET @xmlPerson = '<People><Person>Jane Doe</Person></People>';
SET @strPerson = CONVERT(VARCHAR(MAX), @xmlPerson);
SELECT @strPerson;
```

Listing 10-17: Using the **CONVERT** function to explicitly convert XML data.

As we saw when converting string data to XML data, there might be times when we want to preserve the whitespace and line breaks. The obvious solution is to simply add the third argument to the **CONVERT** function. So let's look at what happens when we do. In Listing 10-18, my string value now includes whitespace and line breaks, and my **CONVERT** function includes **1** as the third argument.

```
DECLARE @xmlPerson XML;
DECLARE @strPerson VARCHAR(MAX);
SET @xmlPerson = '
<People>
  <Person>Jane Doe</Person>
</People>';
SET @strPerson = CONVERT(VARCHAR(MAX), @xmlPerson, 1);
SELECT @strPerson;
```

Listing 10-18: Trying to preserve whitespace and line breaks when converting XML data.

Unfortunately, this solution will not preserve the whitespace or line breaks, and our **SELECT** statement again returns the string as a single line, as shown in Listing 10-19.

```
<People><Person>Jane Doe</Person></People>
```

Listing 10-19: The XML fragment returned without the whitespace and line breaks.

There's a reason for this. Earlier in this Level, in Listing 10-1, you saw how SQL Server implicitly converts a string literal to XML when you assign the value to an **XML** object. When converting the data, SQL Server removes the whitespace and line breaks. As a result, you must explicitly convert the data when first assigning it to your object, as shown in Listing 10-20.

```
DECLARE @xmlPerson XML;
DECLARE @strPerson VARCHAR(MAX);
SET @xmlPerson = CONVERT(XML, '
<People>
  <Person>Jane Doe</Person>
</People>', 1);
SET @strPerson = CONVERT(VARCHAR(MAX), @xmlPerson, 1);
SELECT @strPerson;
```

Listing 10-20: Preserving whitespace and line breaks when preserving XML data.

As you can see, I've used the **CONVERT** function when assigning the data to the **@xmlPerson** variable and then again when assigning that variable value to the **@strPerson** variable. As Listing 10-21 shows, the **SELECT** statement now returns the expected result.

```
<People>
  <Person>Jane Doe</Person>
</People>
```

Listing 10-21: The XML fragment returned with the whitespace and line breaks.

Of course, it makes little sense to explicitly convert a string value to XML and then explicitly convert it back to its original value. But this example helps to demonstrate what happens when converting XML data, so if you get results you don't expect, you might have some understanding of what's going on. Also keep in mind that, when viewing an XML document, the application you use might automatically display the XML in a readable format, even though the XML itself doesn't contain any extra whitespace or line breaks. Yet if you were to view the same XML document as a text file, you might see only one line of text.

Conclusion

As this Level has demonstrated, converting XML data to a string value is a relatively easy process when using the **CAST** or **CONVERT** function. And converting a string value to XML is just as easy, if not easier. You can use either one of the two functions, or you can let SQL Server implicitly convert the value. Even if you were to use an **XML** method to retrieve only a fragment from an XML document, you can still convert the output. For example, you might use the **query()** method to return an XML element and then convert that element to a string. The key is in understanding how XML data is converted. Once you have that understanding, you'll be better able to work with the XML documents in your database.