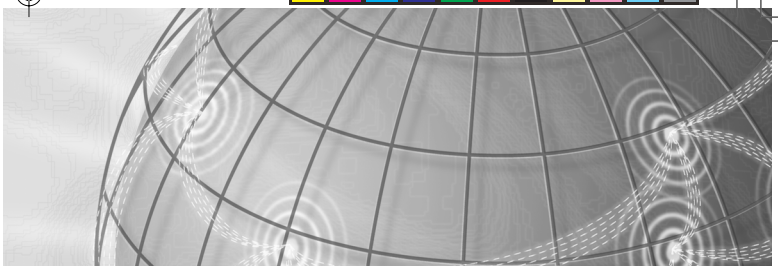

2

Retrieving XML Data Using Transact-SQL

In Chapter 1, I described the use of XML in business integration solutions and the relationship between relational data and XML documents. Now let's turn our attention to extracting data from Microsoft SQL Server in XML format.

Most database application developers are accustomed to retrieving large sets of data from a database server in a rowset format, such as a Microsoft ActiveX Data Objects (ADO) recordset. In a typical application, a SQL SELECT statement is used to select rows from one or more tables in the database and return those rows to the client for processing. SQL Server 2000 extends the SELECT statement to enable the retrieval of data as XML.

The ability to extract SQL Server data as XML is extremely useful in a number of scenarios. Most important, the data is retrieved in a neutral format, which is essential for the creation of business integration solutions in which business documents might need to be exchanged between different systems and different organizations. In this chapter, I'll show you how the developers of the Northwind Traders' e-commerce solution can extract order data to generate XML invoices that can be sent to customers over the Internet.

The SELECT...FOR XML Statement

To help you retrieve XML data from the database, SQL Server 2000 provides an extension to the Transact-SQL SELECT statement in the form of the FOR XML keywords. By appending FOR XML to a SELECT statement, you can indicate to the SQL Server query processor that you want the results to be returned as an XML stream instead of a rowset. In addition to including the FOR XML keywords, you must also specify a *mode* to indicate the format of the XML that should be

returned. This mode can be specified as RAW, AUTO, or EXPLICIT. Here's the basic syntax for the SELECT…FOR XML statement:

```
SELECT select_list
FROM table_source
WHERE search_condition
FOR XML AUTO | RAW | EXPLICIT [, XMLDATA] [, ELEMENTS] [, BINARY BASE64]
```

You use the XMLDATA option to return an XML-Data Reduced (XDR) schema defining the document being retrieved. You use the ELEMENTS option with AUTO mode to return columns as subelements rather than as the default attributes, and you use BINARY BASE64 to specify that binary data should be returned in BASE64 encoding. We'll look at each of these options at greater length later in this chapter.

Before we examine the SELECT…FOR XML statement in detail, you need to understand one important issue. The stream that's returned by a SELECT…FOR XML query isn't a complete XML document but an XML fragment containing an element for each row returned by the query. You must include code in the client application to add a root element to the stream to create a full, well-formed XML document. For example, the following XML fragment could be returned by a SELECT…FOR XML query:

```
<OrderItem OrderID="10248" ProductID="11" Quantity="12"/>
<OrderItem OrderID="10249" ProductID="42" Quantity="10"/>
```

Well-Formed XML

The rules for describing data using XML are fairly strict. Although the rules can cause headaches when you're trying to figure out what's wrong with the document you've created, they're necessary so that XML parsers can easily read and expose XML documents.

First, XML elements must be strictly nested: each opening tag must have a closing tag. Second, XML tags are case sensitive. When you're creating an element using an opening and a closing tag, the case used in the opening tag must match that of the closing tag. Third, all elements in the document must be contained within a single root element. You can have only one top-level element per document. Fourth, all subelements must be wholly contained within their parent element.

An XML document that obeys all these rules is described as being *well formed*.

This sample would be considered a valid XML document only if a root element was added to the fragment, as shown in the following example:

```
<Invoice>
    <OrderItem OrderID="10248" ProductID="11" Quantity="12"/>
    <OrderItem OrderID="10249" ProductID="42" Quantity="10"/>
</Invoice>
```

Using RAW Mode

RAW mode is probably the easiest of the FOR XML modes to understand. Queries executed using RAW mode simply return an XML element for each row in the resulting rowset. The element contains an attribute for each column retrieved. The elements returned are simply given the generic name *row*, while each attribute of a row element takes the name of the corresponding column.

To generate an invoice, for example, the developers of the Northwind Traders' e-commerce solution need to extract a list of items in a particular order as XML. The following FOR XML query could be used:

```
SELECT OrderID, ProductID, UnitPrice, Quantity
FROM [Order Details]
WHERE OrderID = 10248
FOR XML RAW
```

This query would produce the following XML fragment:

```
<row OrderID="10248" ProductID="11" UnitPrice="14" Quantity="12"/>
<row OrderID="10248" ProductID="42" UnitPrice="9.8" Quantity="10"/>
<row OrderID="10248" ProductID="72" UnitPrice="34.8" Quantity="5"/>
```

You can execute this query by running RAW.vbs in the Demos\Chapter2 folder on the companion CD.

Using Joins in RAW Mode Queries

Note that since each row in a RAW mode result set is represented by a single element, all elements in the fragment are empty—that is, they contain no values or subelements. All data is contained in attributes. As I mentioned, mapping columns in a table to attributes in an XML document is referred to as *attribute-centric mapping*. RAW mode queries always return attribute-centric XML, including queries containing a join. For example, to generate an invoice containing order data such as the order date as well as the list of items ordered, the query would need to retrieve data from both the Orders and Order Details tables, as shown in the following example:

```
SELECT Orders.OrderID, OrderDate, ProductID, UnitPrice, Quantity
FROM Orders JOIN [Order Details]
```

(continued)

```
ON Orders.OrderID = [Order Details].OrderID
WHERE Orders.OrderID = 10248
FOR XML RAW
```

This query returns the following XML fragment:

```
<row OrderID="10248" OrderDate="1996-07-04T00:00:00" ProductID="11"
    UnitPrice="14" Quantity="12"/>
<row OrderID="10248" OrderDate="1996-07-04T00:00:00" ProductID="42"
    UnitPrice="9.8" Quantity="10"/>
<row OrderID="10248" OrderDate="1996-07-04T00:00:00" ProductID="72"
    UnitPrice="34.8" Quantity="5"/>
```

Using Column Aliases to Specify Attribute Names

Column aliases can be used to change the names of the attributes returned or to provide a name for a calculated column. However, in a RAW mode query, there's no way to change the name of the elements; you must always use the generic *row*. The following example shows how to use an alias to specify the names of the attributes returned:

```
SELECT OrderID InvoiceNo,
       SUM(Quantity) TotalItems
FROM [Order Details]
WHERE OrderID = 10248
GROUP BY OrderID
FOR XML RAW
```

This query returns the following XML fragment:

```
<row InvoiceNo="10248" TotalItems="27"/>
```

You can execute this query by running RAWGroupBy.vbs in the Demos\Chapter2 folder on the companion CD.

Using AUTO Mode

AUTO mode gives you more control over the XML returned. By default, each row in the result set is represented as an XML element named after the table it was selected from. For example, data could be retrieved from the Orders table using an AUTO mode query, as shown in this example:

```
SELECT OrderID, CustomerID
FROM Orders
WHERE OrderID = 10248
FOR XML AUTO
```

This query returns the following XML fragment:

```
<Orders OrderID="10248" CustomerID="VINET"/>
```

In cases in which table names contain spaces, the resulting XML element names contain encoding characters. For example, we could retrieve our invoice data from the Order Details table using the following AUTO mode query:

```
SELECT OrderID, ProductID, UnitPrice, Quantity
FROM [Order Details]
WHERE OrderID = 10248
FOR XML AUTO
```

However, the resulting XML fragment would look like this:

```
<Order_x0020_Details OrderID="10248" ProductID="11" UnitPrice="14"
    Quantity="12"/>
<Order_x0020_Details OrderID="10248" ProductID="42" UnitPrice="9.8"
    Quantity="10"/>
<Order_x0020_Details OrderID="10248" ProductID="72" UnitPrice="34.8"
    Quantity="5"/>
```

You can execute this query by running AUTOSpaces.vbs in the Demos\Chapter2 folder on the companion CD.

Using Aliases in AUTO Mode Queries

To get around the problem of the resulting XML element names containing encoding characters, we can use aliases. As with RAW mode, column aliases can be used to rename attributes. In AUTO mode queries, however, you can also rename the elements using table aliases, as shown in the following example:

```
SELECT OrderID InvoiceNo,
       ProductID,
       UnitPrice Price,
       Quantity
FROM [Order Details] Item
WHERE OrderID = 10248
FOR XML AUTO
```

The XML fragment returned by this query follows. Note that the element name has been returned as *Item*, which is the alias used in the query.

```
<Item InvoiceNo="10248" ProductID="11" Price="14" Quantity="12"/>
<Item InvoiceNo="10248" ProductID="42" Price="9.8" Quantity="10"/>
<Item InvoiceNo="10248" ProductID="72" Price="34.8" Quantity="5"/>
```

You can execute this query by running AUTOAlias.vbs in the Demos\Chapter2 folder on the companion CD.

Joins in AUTO Mode

Queries with joins in AUTO mode behave differently from RAW mode queries containing joins. Each table in the join results in a nested XML element. For

example, a query to generate an invoice from the Orders and Order Details tables could be written as an AUTO mode query, as shown here:

```
SELECT Invoice.OrderID InvoiceNo,
       OrderDate,
       ProductID,
       UnitPrice Price,
       Quantity
FROM Orders Invoice JOIN [Order Details] Item
ON Invoice.OrderID = Item.OrderID
WHERE Invoice.OrderID = 10248
FOR XML AUTO
```

When executed in AUTO mode, the XML fragment returned is significantly different from the results of a JOIN query using RAW mode, as shown by this partial result set:

```
<Invoice InvoiceNo="10248" OrderDate="1996-07-04T00:00:00">
    <Item ProductID="11" Price="14" Quantity="12"/>
    <Item ProductID="42" Price="9.8" Quantity="10"/>
    <Item ProductID="72" Price="34.8" Quantity="5"/>
</Invoice>
```

You can execute this query by running AUTOJoin.vbs in the Demos\Chapter2 folder on the companion CD.

Using the ELEMENTS Option

Another difference between the RAW and AUTO modes is that the ELEMENTS option can be used in AUTO mode to produce element-centric XML results. When ELEMENTS is specified in an AUTO mode query, all columns are returned as subelements of the element representing the table they belong to. For example, here's how the query used to retrieve invoice data would look with the ELEMENTS option specified:

```
SELECT Invoice.OrderID InvoiceNo,
       OrderDate,
       ProductID,
       UnitPrice Price,
       Quantity
FROM Orders Invoice JOIN [Order Details] Item
ON Invoice.OrderID = Item.OrderID
WHERE Invoice.OrderID = 10248
FOR XML AUTO, ELEMENTS
```

The resulting XML fragment contains an *Invoice* element with a subelement for each column. The *Invoice* element contains an *Item* element, which also has a subelement for each column, as shown in this partial result set:

```
<Invoice>
    <InvoiceNo>10248</InvoiceNo>
    <OrderDate>1996-07-04T00:00:00</OrderDate>
    <Item>
        <ProductID>11</ProductID>
        <Price>14</Price>
        <Quantity>12</Quantity>
    </Item>
    <Item>
        <ProductID>42</ProductID>
        <Price>9.8</Price>
        <Quantity>10</Quantity>
    </Item>
    <Item>
        <ProductID>72</ProductID>
        <Price>34.8</Price>
        <Quantity>5</Quantity>
    </Item>
</Invoice>
```

You can execute this query by running AUTOJoinElements.vbs in the Demos\Chapter2 folder on the companion CD.

> **Note** The ELEMENTS option is an all-or-nothing choice; either all columns are returned as elements or all columns are returned as attributes. You can't use AUTO mode to retrieve XML containing a mixture of element-centric and attribute-centric mappings.

AUTO mode's greater control over the format of the XML returned means that it allows you to retrieve more flexible document structures than RAW mode does. However, GROUP BY queries and aggregate functions aren't supported in AUTO mode, so if you need aggregate data in an XML document, you might want to stick with RAW mode.

Using EXPLICIT Mode

EXPLICIT mode requires a more complex query syntax but gives you the greatest control over the resulting XML. EXPLICIT mode queries define XML fragments in terms of a universal table, which consists of a column for each piece of data you require and two additional columns that are used to define the metadata for the XML fragment. The Tag column uniquely identifies the XML tag that will be used to represent each row in the results, and the Parent column is used to control

the nesting of elements. Each row of data in the universal table represents an element in the resulting XML document.

Identifying the Required Universal Table

The easiest way to understand the EXPLICIT syntax is to begin with the XML document fragment that you want to produce and work backward to figure out the universal table needed to create that particular XML structure. Let's take a simple example to begin with—imagine that we need to produce a simple list of UK-based customers in the following XML format:

```
<Item InvoiceNo=OrderID>ProductID</Item>
<Item InvoiceNo=OrderID>ProductID</Item>
⋮
```

The task of figuring out the universal table required to produce this XML structure requires that you identify the columns needed to define the metadata and data in the document. To identify the metadata columns, you need to examine the hierarchy of elements in the document, noting the different tags in the document that map to tables in the database and the parent/child relationships between the elements. In this case, that's fairly simple. The required XML fragment contains only one tag that's mapped to a table: <Item>, so all Tag fields will have a value of 1. Elements at the top level of the fragment have no parent element, so the Parent of each element is NULL.

Having worked out that each row in the universal table will contain 1 in the Tag column and NULL in the Parent column, we must now turn our attention to the columns required for the data. In our document, we have two pieces of required data, both of which belong to the *Item* element. One is an attribute of the *Item* element, while the other is the actual value of the *Item* element.

Universal tables use the name of the data columns to dictate how the data will be defined in an XML document. Column names in a universal table consist of up to four parameters, as shown here:

```
ElementName!TagNumber!AttributeName!Directive
```

The *ElementName* and *TagNumber* parameters are required to specify the name and tag number of the element the data belongs to, so in our example the column names must all begin with *Item!1* to indicate that the data belongs to an element named *Item*, which is represented by tag number 1. Column names with no attribute name or directive result in element values, which is what we want for the ProductID column. So the column name for the ProductID column is simply Item!1.

Adding the *AttributeName* parameter creates an attribute in the specified element, which is what we want for the InvoiceNo column. To create a column with the required attribute, we need to name the column Item!1!InvoiceNo.

> **Note** The use of the *TagNumber* parameter together with the *ElementName* parameter might at first appear to be redundant because each column in a query against a single table must always use the same ElementName and Tag values. However, when we retrieve data from multiple tables to produce a nested XML hierarchy, the *ElementName* and *Tag* parameters are used to map the values in the columns into the appropriate element in the XML hierarchy.

So we now know that we're looking for the following universal table:

Tag	Parent	Item!1	Item!1!InvoiceNo
1	NULL	ProductID	OrderID
1	NULL	ProductID	OrderID
...	...	...	...

The Transact-SQL code required to produce this table from the data in the Customers table is shown here:

```
SELECT   1 AS Tag,
NULL AS Parent,
ProductID AS [Item!1],
OrderID AS [Item!1!InvoiceNo]
FROM [Order Details]
WHERE OrderID = 10248
```

Note that the Tag and Parent values are explicitly assigned in the SELECT statement. This ensures that every row in the rowset returned by this query will have a Tag column with a value of 1 and a Parent column with a value of NULL.

To generate the required XML document, simply add the FOR XML EXPLICIT clause to the query. This action produces the following results:

```
<Item InvoiceNo="10248">11</Item>
<Item InvoiceNo="10248">42</Item>
<Item InvoiceNo="10248">72</Item>
```

> **Note** In Transact-SQL, the AS keyword is optional when you're assigning an alias. The query could have been written as *SELECT 1 Tag* ..., and so on.

Directives in EXPLICIT Mode Queries

The fourth part of the column name in a universal table is used to provide further control over how the data is represented. The directives supported by FOR XML EXPLICIT queries are the following:

- ***element***: Used to indicate that the data in this column should be encoded and represented as a subelement in the resulting XML fragment.
- ***xml***: Used to indicate that the column should be represented as a subelement in the resulting XML fragment. No encoding of data takes place.
- ***hide***: Used to indicate that a particular column should be present in the universal table but not in the XML fragment returned.
- ***xmltext***: Used to retrieve XML data from an overflow column and append it to the current element. This directive is customarily used when an overflow column has been used to store XML strings that don't belong elsewhere in the table.
- ***cdata***: Used to represent data in this column as a CDATA section in the resulting XML fragment.
- ***ID, IDREF*, and *IDREFS***: Used together with the XMLDATA option to return an inline schema with attributes of type *ID, IDREF,* or *IDREFS*. These directives can be used to create relationships between elements across multiple documents.

Retrieving Subelements with the *element* and *xml* Directives

The most commonly used directive is *element*. It specifies that the data in the column should be rendered as a subelement, rather than as an attribute. To see how this directive is used, let's extend our required XML result to the following format:

```
<Item InvoiceNo=OrderID>
    ProductID
    <Price>UnitPrice</Price>
</Item>
<Item InvoiceNo=OrderID>
    ProductID
    <Price>UnitPrice</Price>
</Item>
⋮
```

We've added an extra piece of data to the XML document, and therefore to the universal table, that needs to be implemented as a subelement of the *Item* element. The universal table required for this structure follows:

Tag	Parent	Item!1	Item!1!InvoiceNo	Item!1!Price!element
1	NULL	ProductID	OrderID	UnitPrice
1	NULL	ProductID	OrderID	UnitPrice
...	...	...	...	...

The Transact-SQL statement to produce an XML fragment based on this universal table is shown in the following example:

```
SELECT   1 AS Tag,
NULL AS Parent,
ProductID AS [Item!1],
OrderID AS [Item!1!InvoiceNo],
UnitPrice AS [Item!1!Price!element]
FROM [Order Details]
WHERE OrderID = 10248
FOR XML EXPLICIT
```

This code produces the following XML fragment containing an *Item* element with an *InvoiceNo* attribute, the ID of the product as a value, and a *Price* subelement:

```
<Item InvoiceNo="10248">
    11
    <Price>14</Price>
</Item>
<Item InvoiceNo="10248">
    42
    <Price>9.8</Price>
</Item>
<Item InvoiceNo="10248">
    72
    <Price>34.8</Price>
</Item>
```

You can execute this query by running EXPLICIT.vbs in the Demos\Chapter2 folder on the companion CD.

The *element* directive encodes the data in the column. For example, if we suppose a column contained the data *>5*, the *element* directive would encode this as *>5*. The *xml* directive performs the same function as *element* but doesn't encode the data.

The *element* and *xml* directives make it possible to retrieve XML fragments that contain a mixture of attribute-centric and element-centric mappings with EXPLICIT mode queries. The other directives are useful in certain specific circumstances, and we'll examine these shortly. But first let's see how we can use EXPLICIT mode to retrieve data from multiple tables.

Using EXPLICIT Mode to Retrieve Related Data

So far we've used EXPLICIT mode queries to retrieve data from a single table. What if you need data from more than one table? For example, let's suppose you want to retrieve an XML fragment containing the name of the products ordered. To do this, we can use a query joining the Order Details and Products tables, as shown here:

```
SELECT   1 AS Tag,
NULL AS Parent,
ProductName AS [Item!1],
OrderID AS [Item!1!InvoiceNo],
OD.UnitPrice AS [Item!1!Price!element]
FROM [Order Details] OD JOIN Products P
ON OD.ProductID = P.ProductID
WHERE OrderID = 10248
FOR XML EXPLICIT
```

The XML result for this query is shown here:

```
<Item InvoiceNo="10248">
    Queso Cabrales
    <Price>14</Price>
</Item>
<Item InvoiceNo="10248">
    Singaporean Hokkien Fried Mee
    <Price>9.8</Price>
</Item>
<Item InvoiceNo="10248">
    Mozzarella di Giovanni
    <Price>34.8</Price>
</Item>
```

In the preceding example, a JOIN operator was used to replace a foreign key column with data from the related table. This is relatively simple and is no different from how you would perform the same task in an AUTO or RAW mode query. However, suppose we wanted to retrieve the order header data for each order detail so that the XML produced contains a nested heirarchy in which each order is represented by an element that contains child elements representing the order details. To retrieve parent/child data in an EXPLICIT mode query is trickier than it first appears. You might imagine that you can retrieve related data. We'll do this simply by adding another table to the query like this:

```
SELECT   1 AS Tag,
NULL AS Parent,
ProductName AS [Item!1],
O.OrderID AS [Item!1!InvoiceNo],
OrderDate AS [Item!1!Date],
```

```
OD.UnitPrice AS [Item!1!Price!element]
FROM Orders O
JOIN [Order Details] OD ON O.OrderID = OD.OrderID
JOIN Products P ON OD.ProductID = P.ProductID
WHERE O.OrderID= 10248
FOR XML EXPLICIT
```

The results are shown here:

```
<Item InvoiceNo="10248" Date="1996-07-04T00:00:00">
    Queso Cabrales
    <Price>14</Price>
</Item>
<Item InvoiceNo="10248" Date="1996-07-04T00:00:00">
    Singaporean Hokkien Fried Mee
    <Price>9.8</Price>
</Item>
<Item InvoiceNo="10248" Date="1996-07-04T00:00:00">
    Mozzarella di Giovanni
    <Price>34.8</Price>
</Item>
```

You can execute this query by running EXPLICITJoin.vbs in the Demos\Chapter2 folder on the companion CD.

As you can see, this strategy does return a list of order items for a particular order. However, it's not the most efficient XML representation of the data. The order header information (the *OrderID* and *OrderDate* values) is repeated for each item. Ideally, we want to group all the order heading data under a single *Invoice* element, with a subelement containing the data relating to each item. A better XML structure for the data might look something like this:

```
<Invoice InvoiceNo="10248" Date="1996-07-04T00:00:00">
    <Item Product="Queso Cabrales">
        <Price>14</Price>
    </Item>
    <Item Product="Singaporean Hokkien Fried Mee">
        <Price>9.8</Price>
    </Item>
    <Item Product="Mozzarella di Giovanni">
        <Price>34.8</Price>
    </Item>
</Invoice>
```

To retrieve the data in this structure, we need to identify the required universal table. The first step is to identify the metadata column values we need, and this is where we encounter a major difference from the queries we have executed up to now. There are *two* tags that map to tables in the required fragment: *<Invoice>,* which maps to the Products table and *<Item>*, which maps to

the Order Details table. The Tag and Parent values for these elements must be different; for example, we could assign the <*Invoice*> tag a value of 1 in the tag column and <*Item*> could be identified by the value 2. Notice also that the Parent values must be different as well. The *Invoice* element has no parent, and can therefore be assigned NULL in the Parent column, but the *Item* element is a child of the *Invoice* element, and so must have tag number 1 assigned in the Parent column.

Now, since the Tag and Parent values are explicitly assigned in the SELECT statement, we have a rather tricky problem: how do we assign two different sets of values in one query? The answer is we don't. The Transact-SQL UNION ALL keywords allow us to create multiple separate queries and collate the results. We can use the UNION ALL operator to build the universal table we need from two queries: one for the *Invoice* element and the other for the *Item* element.

> **Note** We use the UNION ALL operator rather than just UNION to eliminate any duplicate rows returned by any of the queries.

Here's the universal table we need to create. The first and fourth rows are returned by the *Invoice* element query. (Note that the Product and Price columns are NULL.) The other rows are returned by the *Item* element query.

Tag	Parent	Invoice!1!InvoiceNo	Invoice!1!Date	Item!2!Product	Item!2!Price!element
1	NULL	InvoiceNo	OrderDate	NULL	NULL
2	1	InvoiceNo	NULL	ProductName	UnitPrice
2	1	InvoiceNo	NULL	ProductName	UnitPrice
1	NULL	InvoiceNo	OrderDate	NULL	NULL
2	1	InvoiceNo	NULL	ProductName	UnitPrice
...	...	...	...	...	...

Let's first turn our attention to the *Invoice* element query. This query is fairly straightforward. The only difference from previous examples is that since the results are going to be combined with the *Item* element query using the UNION ALL operator, we need to specify the same columns in both queries. This means we need to specify a column for the Product and Price fields, even though the values aren't returned by this query. We get around this inconvenience by explicitly assigning a NULL in those columns.

```
SELECT   1 AS Tag,
         NULL AS Parent,
         OrderID AS [Invoice!1!InvoiceNo],
         OrderDate AS [Invoice!1!Date],
         NULL AS [Item!2!Product],
         NULL AS [Item!2!Price!element]
FROM Orders
WHERE OrderID = 10248
```

The *Item* element query is a little more complex. First, the Tag column needs to indicate that this data maps to tag number 2 in the hierarchy, and the Parent column needs to indicate that it's a child of tag number 1 (*Invoice*). Second, we need to return data from the Orders, Products, and Order Details tables so that we can match order details to their orders. This means that we have to use two joins. We must also use the same column layout as in the *Invoice* element query, so we include the *OrderID* column, which will be used to collate Orders with Order Details, and specify a NULL placeholder for the *OrderDate* columns.

```
SELECT  2,
        1,
        O.OrderID,
        NULL,
        P.ProductName,
        OD.UnitPrice
FROM Orders O JOIN [Order Details] OD
ON O.OrderID = OD.OrderID
JOIN Products P
ON OD.ProductID = P.ProductID
WHERE O.OrderID = 10248
```

The final task is to use the UNION ALL operator to combine the two queries and use an ORDER BY clause to ensure that the XML elements are collated properly, as shown here:

```
SELECT  1 AS Tag,
        NULL AS Parent,
        OrderID AS [Invoice!1!InvoiceNo],
        OrderDate AS [Invoice!1!Date],
        NULL AS [Item!2!Product],
        NULL AS [Item!2!Price!element]
FROM Orders
WHERE OrderID = 10248
UNION ALL
SELECT  2,
        1,
        O.OrderID,
        NULL,
```

(continued)

```
        P.ProductName,
        OD.UnitPrice
FROM Orders O JOIN [Order Details] OD
ON O.OrderID = OD.OrderID
JOIN Products P
ON OD.ProductID = P.ProductID
WHERE O.OrderID = 10248
ORDER BY [Invoice!1!InvoiceNo], [Item!2!Product]
FOR XML EXPLICIT
```

You can execute this query by running EXPLICITUnion.vbs in the Demos\Chapter2 folder on the companion CD.

You can use EXPLICIT mode to retrieve documents that contain data from multiple tables by simply using UNION ALL to add another query for each tag that maps to a table. Although the syntax seems complex at first, the key to building any EXPLICIT query is to start with the XML structure you want to retrieve and then count how many different tags there are that map to tables. Once you have done this, you can figure out the layout of the required universal table and work out the necessary Transact-SQL statement to generate the table.

Sorting Data with the *hide* Directive

You use the *hide* directive to retrieve columns you don't want to display in the resulting XML fragment. This might seem like a strange thing to want to do at first, but the practice is useful if you want to arrange the data in a specific order (using an ORDER BY clause) but don't need the sort column in the results. For ordinary queries, you don't need the *hide* directive to do this; any field can be used in an ORDER BY clause as long as it belongs to a table referenced in the FROM clause. However, when you're using the UNION ALL operator, all fields in the ORDER BY clause must appear in the SELECT list.

For example, the following query could be used to sort all invoices for a particular customer in order of date:

```
SELECT   1 AS Tag,
NULL AS Parent,
CustomerID AS [Invoice!1!Customer],
OrderID AS [Invoice!1!InvoiceNo],
OrderDate AS [Invoice!1!Date!hide],
NULL AS [Item!2!Product],
NULL AS [Item!2!Price!element]
FROM Orders
WHERE CustomerID = 'VINET'
UNION ALL
SELECT   2,
         1,
```

```
O.CustomerID,
O.OrderID,
O.OrderDate,
P.ProductName,
OD.UnitPrice
FROM Orders O JOIN [Order Details] OD
ON O.OrderID = OD.OrderID
JOIN Products P
ON OD.ProductID = P.ProductID
WHERE O.CustomerID = 'VINET'
ORDER BY [Invoice!1!InvoiceNo], [Item!2!Product], [Invoice!1!Date!hide]
FOR XML EXPLICIT
```

This code produces the following XML fragment in which the customer invoices are sorted by date but the date field isn't included in the results:

```
<Invoice Customer="VINET" InvoiceNo="10248">
    <Item Product="Mozzarella di Giovanni">
        <Price>34.8</Price>
    </Item>
    <Item Product="Queso Cabrales">
        <Price>14</Price>
    </Item>
    <Item Product="Singaporean Hokkien Fried Mee">
        <Price>9.8</Price>
    </Item>
</Invoice>
<Invoice Customer="VINET" InvoiceNo="10274">
    <Item Product="Flotemysost">
        <Price>17.2</Price>
    </Item>
    <Item Product="Mozzarella di Giovanni">
        <Price>27.8</Price>
    </Item>
</Invoice>
⋮
```

Using the *xmltext* Directive to Retrieve XML Values

One interesting problem facing developers of integration solutions is matching the data entities in one application with those in another. For example, let's suppose that when you're building the e-commerce solution for Northwind Traders, data from customers is received in XML documents. A customer might send the following customer details update document to the Northwind database:

```
<Customerdetails>
    <CustomerID>AROUT</CustomerID>
    <CompanyName>Around the Horn</CompanyName>
```

(continued)

```
    <ContactName>Thomas Hardy</ContactName>
    <ContactTitle>Sales Representative</ContactTitle>
    <Address>120 Hanover Sq.</Address>
    <City>London</City>
    <Region>Europe</Region>
    <PostalCode>WA1 1DP</PostalCode>
    <Country>UK</Country>
    <Phone>(171) 555-7788</Phone>
    <Fax>(171) 555-6750</Fax>
    <Web email="sales@aroundhorn.co.uk"
        site="www.aroundhorn.co.uk">
</Customerdetails>
```

The Customers table in the Northwind database has a matching column for each element in this document except for the *Web* element. Of course, customers could send much more data than is actually required by the Customers table. Rather than simply discard this data, you might create an overflow column in the Customers table and add the extra data as XML to this column. In the preceding example, we'd simply insert *<Web email="sales@aroundhorn.co.uk" site="www.aroundhorn.co.uk"/>* into the overflow column.

To retrieve XML data from the overflow column, we can use the *xmltext* directive. When you're using this directive, the position of the retrieved XML in the document depends on whether you specify an attribute name. If an attribute name is specified, the data is retrieved as a subelement with the specified name. If no attribute name is specified, the data is merged into the parent element. Let's see an example of each approach. First we'll specify an attribute name, as shown here:

```
SELECT  1 AS Tag,
        NULL AS Parent,
        companyname AS [customer!1!companyname],
        phone AS [customer!1!phone],
        overflow AS [customer!1!overflow!xmltext]
FROM Customers
WHERE CustomerID = 'AROUT'
FOR XML EXPLICIT
```

The results are shown here:

```
<customer companyname="Around the Horn" phone="(171) 555-7788">
    <overflow email="sales@aroundhorn.co.uk"
        site="www.aroundhorn.co.uk"/>
</customer>
```

The effect of not specifying an attribute name is shown here:

```
SELECT 1 AS Tag,
       NULL AS Parent,
       companyname AS [customer!1!companyname],
       phone AS [customer!1!phone],
       overflow AS [customer!1!!xmltext]
FROM Customers
WHERE CustomerID = 'AROUT'
FOR XML EXPLICIT
```

This code produces the following XML fragment:

```
<customer companyname="Around the Horn"
       phone="(171) 555-7788"
    email="sales@aroundhorn.co.uk"
    site="www.aroundhorn.co.uk"/>
```

This flexibility is a very powerful feature of the EXPLICIT statement. Entire XML documents can be stored in a single column and extracted using this statement.

Retrieving CDATA with the *cdata* Directive

XML documents often need to contain nonparsed character data. For example, you might want to include the text *Elements look like <this>* in an XML document, but if the text were parsed, the word *<this>* would be interpreted as an element. To avoid this problem, XML documents support the creation of CDATA sections. A CDATA section contains character data that isn't parsed by an XML parser.

To retrieve data from a table and place it in a CDATA section, you can use the *cdata* directive. The only rule you have to remember when using the *cdata* directive is that no attribute name can be specified.

In the following example, the telephone number of the Around the Horn customer is returned as CDATA:

```
SELECT 1 AS Tag,
       NULL AS Parent,
       companyname AS [customer!1!companyname],
       phone AS [customer!1!!cdata]
FROM customers
WHERE CustomerID = 'AROUT'
FOR XML EXPLICIT
```

The results are shown here:

```
<customer companyname="Around the Horn">
    <![CDATA[(171) 555-7788]]>
</customer>
```

Using the *ID*, *IDREF*, and *IDREFS* Directives and the XMLDATA Option

In Chapter 1, I described the use of the *ID, IDREF,* and *IDREFS* data types to represent relational data in an XML document. This can be a useful technique for exchanging complex data while minimizing the amount of data duplication in the document.

You can use the *ID, IDREF*, and *IDREFS* directives in EXPLICIT mode queries to specify relational fields in the resulting XML document. Of course, this approach is useful only if a schema is used to define the document and identify the fields employed to link one entity to another. The XMLDATA option provides a way to generate an inline schema for the XML document returned by a FOR XML query in RAW, AUTO, or EXPLICIT mode, and when used together with the *ID, IDREF,* or *IDREFS* directives in an EXPLICIT mode query, it can be used to identify relational fields in a document. For example, a list of all invoices for a particular customer might be required. Rather than duplicate product data for each invoice, you could include a separate list of products in the document and use an *ID/IDREF* relationship to link the products to the orders. You can see the query used to retrieve this data here:

```
SELECT 1 AS Tag,
       NULL AS Parent,
       ProductID AS [Product!1!ProductID!id],
       ProductName AS [Product!1!Name],
       NULL AS [Order!2!OrderID],
       NULL AS [Order!2!ProductNo!idref]
FROM Products
UNION ALL
SELECT 2,
       NULL,
       NULL,
       NULL,
       OrderID,
       ProductID
FROM [Order Details]
ORDER BY [Order!2!OrderID]
FOR XML EXPLICIT, XMLDATA
```

A partial result of this query appears here:

```
<Schema name="Schema1" xmlns="urn:schemas-microsoft-com:xml-data"
    xmlns:dt="urn:schemas-microsoft-com:datatypes">
    <ElementType name="Product" content="mixed" model="open">
        <AttributeType name="ProductID" dt:type="id"/>
        <AttributeType name="Name" dt:type="string"/>
```

```
        <attribute type="ProductID"/>
        <attribute type="Name"/>
    </ElementType>
    <ElementType name="Order" content="mixed" model="open">
        <AttributeType name="OrderID" dt:type="i4"/>
        <AttributeType name="ProductNo" dt:type="idref"/>
        <attribute type="OrderID"/>
        <attribute type="ProductNo"/>
    </ElementType>
</Schema>
<Product xmlns="x-schema:#Schema1" ProductID="1" Name="Chai"/>
<Product xmlns="x-schema:#Schema1" ProductID="2" Name="Chang"/>
<Product xmlns="x-schema:#Schema1" ProductID="3" Name="Aniseed Syrup"/>
⋮
<Order xmlns="x-schema:#Schema1" OrderID="10248" ProductNo="11"/>
<Order xmlns="x-schema:#Schema1" OrderID="10248" ProductNo="42"/>
<Order xmlns="x-schema:#Schema1" OrderID="10249" ProductNo="72"/>
```

The resulting XML fragment contains an inline schema, which defines the elements and attributes in the document. The fields specified as *ID* and *IDREF* in the EXPLICIT mode query are assigned to XML data types *ID* and *IDREF*. For the other fields, an appropriate data type has been selected based on the data returned by the query. The *ID* and *IDREF* fields create a relationship between the *ProductID* attribute in the *Product* element and the *ProductNo* attribute in the *Order* element.

Retrieving Binary Fields with the BINARY BASE64 Option

Binary data such as images can be retrieved in an XML document in BASE64-encoded format, which is useful if you need to send binary data to an application or trading partner. To retrieve binary BASE64 data, you must specify the BINARY BASE64 option in a FOR XML query, as shown here:

```
SELECT picture
FROM categories
WHERE categoryid = 1
FOR XML RAW, BINARY BASE64
```

This code returns an encoded image, as shown in the following partial XML fragment. (The binary data has been truncated.)

```
<row picture="FRwvAAIAAAANAA4AFAAhAP////9CaXRtYXAgSW1hZ2UAUGFpbnQu ... "/>
```

You can also retrieve a reference to binary data when using AUTO mode. This reference can be used to retrieve the data over HTTP through a SQL Server

virtual root. (I'll talk about HTTP access to SQL Server in Chapter 4.) To retrieve a reference to binary data, you must include a primary key field in the query, as you can see here:

```
SELECT categoryid, picture
FROM categories
WHERE categoryid = 1
FOR XML AUTO
```

The resulting XML fragment contains an XPath reference to the record containing the binary data, as shown here:

```
<categories categoryid="1"
    picture="dbobject/categories[@CategoryID='1']/@Picture"/>
```

Summary

FOR XML queries give you a flexible way to extract data from SQL Server as XML. This technology enables you to generate complex business documents for exchange between applications and trading partners. Often, RAW or AUTO mode will be adequate for your needs, but for more complex XML formats, EXPLICIT mode makes it possible to extract data to your exact specification.

Of course, writing queries to extract XML is only part of the picture. You need to build software that can connect to the database server and consume the XML data produced. In the next chapter, we'll examine how the Microsoft ActiveX Data Objects 2.6 library can be used to build XML-aware client applications.