

## Chapter 7

# INTEGRATING XQUERY AND RELATIONAL DATABASE SYSTEMS

Michael Rys

**X**ML allows the combination of highly structured, semi-structured, and **markup data** in one common representation, and thus provides a more flexible data representation model than the existing **relational data model**. On the other hand, relational database systems have a proven track record of providing efficient and cost-effective management of **structured data**. Since most relational database systems have begun to add XML support and features to their relational database engines and are deeply entrenched in the data management arena, one need not be a prophet to predict that relational database systems will most likely capture a large market for both virtual and native XML data storage and processing. Querying XML data will be an integral part of the complete XML support, and XQuery is very likely to become an important component of XML-enabled relational database systems.

This chapter outlines how to integrate XQuery with a relational database system. It shows how to query XML in the context of and in conjunction with relational data and how to provide specific query-hosting environments in the context of relational systems that the XQuery specification assumes but does not describe. In particular, it covers how to apply XQuery over XML views on relational data (using an XML view mechanism such as annotated schemata) and how XQuery can be used on a relational **XML datatype**.

## Getting Started

XML and the vocabularies defined with XML have established themselves as the prevalent and most promising lingua franca of business-to-business, business-to-consumer, or generally, any-to-any data interchange and integration. One of the major reasons for the success of XML in this space is that XML is a simple, **Unicode**-based, platform-independent syntax for which simple and efficient parsers are widely available. Another important factor in favor of XML is its ability not only to represent structured data, but also to provide a uniform syntax for structured data, semi-structured data, and markup data. *Structured data* is data that easily fits into a predefined, homogeneous type structure such as relations or nested objects. *Semi-structured data* is data that may change its structure from instance to instance. For example, it may have some components that are one-off annotations that appear only on single instances or it may have components that are of heterogeneous type structure (e.g., one time the address data is a string, another it is a complex object). Finally, *markup data* is data that is mainly free-flowing text with interspersed markup to call out document structure and highlight important aspects of the text. Listing 7.1 gives a short example of each type of data.

### Listing 7.1 Examples of Structured, Semi-Structured and Markup Data

```
structured data:
<Customer>
  <ID>C1</ID>
  <FirstName>Janine</FirstName>
  <LastName>Smith</LastName>
  <Address>
    <Street>1 Broadway Way</Street>
    <City>Seattle</City>
    <Zip>WA 98000</Zip>
  </Address>
  <Order>
    <ID>01</ID>
    <OrderDate>2003-01-21</OrderDate>
    <Amount>7</Amount>
    <ProductID>P1</ProductID>
  </Order>
  <Order>
    <ID>02</ID>
    <OrderDate>2003-06-24</OrderDate>
    <Amount>3</Amount>
    <ProductID>P3</ProductID>
  </Order>
</Customer>
```

**Listing 7.1** Examples of Structured, Semi-Structured and Markup Data (*continued*)

```

semistructured data:
<PatientRecord pid="P1">
  <FirstName>Janine</FirstName>
  <LastName>Smith</LastName>
  <Address>
    <Street>1 Broadway Way</Street>
    <City>Seattle</City>
    <Zip>WA 98000</Zip>
  </Address>
  <Visit date="2002-01-05">
    Janine came in with a <symptom>rash</symptom>. We identified a
    <diagnosis>antibiotics allergy</diagnosis> and <remedy>changed her
    cold prescription</remedy>.
  </Visit>
</PatientRecord>
<PatientRecord>
  <pid>P2</pid>
  <Name>Nils Soerensen</Name>
  <Address>
    23 NE 40th Street, New York
  </Address>
</PatientRecord>

Marked-up data:
  <Visit date="2002-01-05">Janine came in with a
  <symptom>rash</symptom>. We identified a <diagnosis>antibiotics
  allergy</diagnosis> and <remedy>changed her cold
  prescription</remedy>.</Visit>

```

Much of the data interchanged today originates from relational databases and is finally stored again in relational databases. Since relational systems today are predominantly used to manage structured data (an educated guess would be 80 percent or more), most of the XML generated and being consumed at this time in the context of data interchange also fits the relational model of structured data. Therefore most relational database systems have first focused on providing XML capabilities that fit this most common usage scenario and have just recently begun to provide XML capabilities to deal with data that doesn't fit into the structured mold.

By adding the more general capabilities to deal with any form of XML data, relational database systems will be able to extend their efficient and effective processing of structured data to new application areas such as XML-based document management, management of XML messages such as **SOAP** messages, and the processing of audit logs in XML format. By handling XML natively instead of requiring shredding into relational

structures, relational database systems will be able to provide seamless support for these applications.

Relational database systems not only store XML data, but also provide query and update support on the relational and XML data they store. While SQL has proven itself as the query language for relational data, it is not suitable in its current form as the direct query language over XML data. Since a query language should have as little impedance mismatch as possible to the data model it queries, the W3C, in cooperation with its member companies, decided to design a new XML query language named XQuery. To provide the necessary support for native XML data storage and processing, XQuery must become an integral part of XML-enabled relational database systems. In order to store and query XML data natively in a relational database, the XML data must be exposed in the relational system as a manageable entity. Relational systems store and expose XML in one of the following forms:

- *Virtual XML view*: a virtual document—also called a virtual XML view—over predominantly relational data
- *LOB (large object) storage*: an LOB-based XML document, either as a character LOB, called a CLOB, a binary LOB, called a **BLOB**, or a native XML datatype (XML)
- *Relational table mappings*: an XML document decomposed into relational data structures, possibly with subtrees mapped to LOBs

Chapter 6, “Mapping between XML and Relational Data,” provides the in-depth investigation of the multitude of options for storing and querying virtual XML views over relational and XML documents that have been decomposed into relational data. Each of these options operates as a conceptual mid-tier or client-side component that uses the existing relational interfaces. They all expose an XML view against one or multiple relational databases. XML queries posed against the exposed view are translated into relational queries that are shipped over to the database server. After they have been executed, the result is returned either as XML or relational data for potential further post-processing.

Unlike Chapter 6, this chapter concentrates on the storage and processing of XML data that integrates with the relational system’s capabilities, such as the relational query processor and execution environment. We

begin with an overview of the XML datatype—the primary mechanism provided by relational systems for native XML data storage—before explaining how XQuery access is provided on the XML data and how XQuery is embedded in the relational query system. Then we outline how the concept of tables as collections of tuples can be generalized and applied to collections of XML documents and how XQuery can be integrated with that approach.

The section on the XML datatype discusses the appearance of the type to the user (its logical model), the physical implementation options, and the relationship to the XML Schema metadata. The section on integrating XQuery and SQL describes how to use XQuery to query an XML datatype instance, how to combine XQuery with SQL, how to correlate relational and XML data. We conclude with an overview on how to physically map XQuery in the context of a relational query engine.

As this is written, many of the features described below are available in one form or another from current or upcoming versions of the major relational database systems such as (in alphabetical order) IBM DB2, Microsoft SQL Server, and Oracle. Since the feature sets of individual database systems are changing from release to release, we do not usually attribute the features described here to any particular database system. Instead, we refer to the SQL/XML part of the **SQL-2003** standard [SQL2003] where appropriate and use a pseudo-notation for anything that is not (yet) described in SQL-2003. For example, we use an abstract functional notation of the form `isvalid(XML, SchemaComponents, 'lax'|'strict')→boolean` for functions on the XML datatype. Different implementations may provide different syntax, ranging from method calls on the XML type instance to SQL keywords.

## Relational Storage of XML: The XML Type

---

XQuery processing on XML stored in the database as an LOB depends heavily on the actual physical storage mechanism chosen to store the XML. All LOB-based storage mechanisms have in common that they basically provide a built-in XML datatype. In the following, we use the relational type named `XML` to designate a new SQL built-in datatype that provides the logical abstraction for XML data stored in a relational database.

Listing 7.2 gives an example of a relational table where one column is of type XML. While the example uses an XML document representing structured data that could also be mapped into relational tables, the discussion of the XML datatype below applies equally to any form of XML data (semi-structured and markup data). In what follows, we look first at the logical models for such an XML datatype, discuss the different physical representations, and look into some other important aspects of XML datatypes, such as the relationships of **encodings** and database collations and their association with XML schemata.

**Listing 7.2** Example of Relational Table with XML Datatype

```
CREATE TABLE Order(id int, orderdate date, PODetail XML)
```

Example data (using a string representation of the XML datatype instances):

id	orderdate	PODetail
4023	2001-12-01	<pre>&lt;purchaseOrder   xmlns="http://po.example.com"&gt;   &lt;originator billId="0013579"&gt;     &lt;contactName&gt;       Fred Allen     &lt;/contactName&gt;     &lt;contactAddress&gt;       &lt;street&gt;123 2nd Ave. NW&lt;/street&gt;       &lt;city&gt;Anytown&lt;/city&gt;       &lt;state&gt;OH&lt;/state&gt;       &lt;zip-four&gt;99999-1234&lt;/zip-four&gt;     &lt;/contactAddress&gt;     &lt;phone&gt;(330) 555-1212&lt;/phone&gt;     &lt;originatorReferenceNumber&gt;       AS 1132     &lt;/originatorReferenceNumber&gt;   &lt;/originator&gt;   &lt;order&gt;     &lt;item code="34xdl 1278 12ct"       quantity="1" /&gt;     &lt;item code="57xdl 7789"       quantity="1"       colorcode="012" /&gt;   &lt;/order&gt;   &lt;shipAddress sameAsContact="true" /&gt;   &lt;shipCode carrier="02" /&gt; &lt;/purchaseOrder&gt;</pre>
5327	2002-04-23	<pre>&lt;purchaseOrder   xmlns="http://po.example.com"&gt;   &lt;originator     ...</pre>

## Logical Models for the XML Datatype

The SQL-2003 standard defines an `XML` datatype as part of its XML-related extensions that are commonly referred to as SQL/XML [SQLXML]. The standard does not prescribe the exact storage format as long as the type satisfies the `XML` datatype requirements. One requirement is that the type must be able to represent not only an XML 1.0 document but any element content, including multiple top-level element nodes and text nodes. Listing 7.3 gives an example of an `XML` datatype instance that has multiple top-level element nodes. The current logical model of the `XML` datatype is based on an extended XML Infoset in which the document information item has been extended to allow top-level text nodes and more than one top-level element node.

**Listing 7.3** Example of XML Datatype Instance with Multiple Top-Level Elements

```
<Order>
  <ID>01</ID>
  <OrderDate>2003-01-21</OrderDate>
  <Amount>7</Amount>
  <ProductID>P1</ProductID>
</Order>
<Order>
  <ID>02</ID>
  <OrderDate>2003-06-24</OrderDate>
  <Amount>3</Amount>
  <ProductID>P3</ProductID>
</Order>
```

This Infoset-based model can easily be mapped to an untyped instance of the XQuery data model by applying the Infoset mapping outlined by [XQ-DM]. The addition of type information is explained below. This model is somewhat more limited than the XQuery data model in the sense that certain components of the XQuery data model, such as top-level attribute nodes, multiple document nodes, top-level typed values and top-level heterogeneous sequences containing such items, cannot be represented. However, these items can still be used inside XQuery expressions. Also, typed values returned by XQuery expressions that cannot be represented as `XML` datatype instances but must be exposed in the relational model have to be mapped into the relational type system.

A future version of the `XML` datatype's logical model may be extended to an XQuery data model. However, in what follows, we work with the

Infoset-based model extended with the ability to deal with typed data. As a built-in “scalar” type in the SQL type system, such a type can be cast, for example, from and to the SQL character types using the SQL `cast` expression. Casting from string to XML will parse and casting from XML to string will serialize the XML data.

## Physical Models for the XML Datatype

There exist a wide variety of physical representations for XML datatypes (regardless of the exact logical model). In addition, the storage format has some relation to the questions of whether the XQuery processor should integrate with the existing relational query processor or be a separate component that only executes XQuery expressions, and how the storage format affects the update performance. We will examine several possible storage mechanisms and identify the major issues related to the query-processing approaches and updates. Query-processing aspects are discussed in more detail in a later section. All the storage mechanisms assume that the data actually has been verified to be a well-formed instance of the XML datatype.

Users may have different expectations about **XML storage fidelity**. Generally, we can distinguish among *string-level fidelity*, in which the XML datatype instance preserves the original XML document code-point for code-point; *Infoset-level fidelity*, in which the XML datatype instance preserves the XML document structure on the level of the XML Infoset model; and the *relational fidelity level*, which only preserves information important from a relational point of view, such as the property-value association, and disregards XML-specific properties, such as document order. Obviously the XML datatype’s logical model requires at least Infoset-level fidelity.

### **Character LOB**

With CLOB, the XML is stored in a character representation. The representation may preserve the original XML data exactly (string-level fidelity), or the data may have been transformed by changing the

encoding or by **canonicalizing** the content, while still providing Infoset-level fidelity.

This storage format does not lend itself well to efficient, server-side XQuery processing, since it either requires extensive **indexing** mechanisms or requires every query to parse the data before executing the query in a separate XQuery component. The indexing mechanisms end up replicating one of the other storage mechanisms in order to take advantage of the existing relational query processing. In addition, basic character LOBs are hard to update on a node level, since most LOB update mechanisms are based on positional ranges and not logical subtrees.

### ***Binary LOB***

The binary format may represent preprocessed XML in the form of a W3C **Document Object Model** (DOM) tree, which has been shown empirically to increase the storage size by a factor of two to six over the textual representation due to the storage overhead of the DOM nodes. Some other preprocessed binary format is more likely to be used that may provide more efficient index processing, compression, or other benefits to the query processing. While a binary format could provide string-level fidelity, it usually provides only Infoset-level fidelity.

A binary LOB provides an additional level of abstraction over CLOBs at the cost of not being able to read the XML without an access method that reproduces the original textual representation. As with CLOB, XQuery processing often requires additional indexing to take advantage of the relational query processor, or the use of a separate XQuery component such as an embedded XQuery engine written in Java or C#. Propagating node-level updates are often costly since the general BLOB support in relational systems is not designed for logical updates.

### ***User-Defined Type***

All the major relational database systems provide (or are going to provide) programmatic extensions inside the database server that allow users to add their own code and datatypes. Of course the database system can

also use this mechanism to provide additional, more complex, built-in datatypes such as XML. Such user-defined types may again represent the XML data in many ways, ranging from a string representation over a DOM tree representation to an Infoset object representation. The user-defined type may require the whole instance to be loaded into memory before it can be operated upon, or it may provide some integration into the relational buffers to provide partial loading of the data.

Two major approaches are currently advocated for supporting user-defined types. The deep integration approach provides tight integration into the actual query processor at the cost of complex programming and registration requirements, since, for every user-defined type, code specific to the query processor must be provided through predefined APIs (this approach is often called *data blade*, *extender*, or *cartridge*). The other approach uses a virtual machine such as the **Java Virtual Machine** or the **Common Language Runtime** to add user-defined types (the VM approach). This approach trades off better programmability against tighter integration.

The integration approach can be used to implement most of the other physical designs outlined in this chapter and integrates tightly into the components of the relational database engine's architecture, such as memory management and query processing, at the cost of flexibility and programmability. The VM approach is an alternative that gives great flexibility in processing XML data at the cost of minimal or no integration with the general storage engine and query processing. In this case, the XQuery engine is usually a component separate from the relational engine that can operate on XML datatype instances that are loaded fully into the VM's memory and use a memory-based representation of the data model. Some examples of the latter are query components written in Java or C# that run inside the virtual machines. Eventually, these two approaches may merge and combine the VM's comparable ease of programming and the efficiency of the integration approach.

Updates are provided as another method on the user-defined type. Depending on the internal structure, they may or may not be very efficient.

### ***Relational Table Mappings***

If XQuery processing is being integrated with the relational query processor, the actual XML data needs to be mapped to relational data and indices. As with mid-tier mapping strategies, a variety of approaches exist. Some of them hide the generated tables from the relational user and thus serve as purely physical models, while others expose the tables in the relational context.

By mapping the XML into relational structures, we can define additional relational indices on the tables, integrate the XQuery processing with relational query processing, and perform update processing on the node level. Every relational table mapping has to provide Infoset-level fidelity in order to be usable as physical storage of the XML datatype. Note that several XML-to-relational mappings provided outside of the relational database engine often only provide relational fidelity.

#### **Node Tables**

The most general approach generates the so-called **node table** (sometimes also called **edge table**) that represents every node in the XML instance as a row in the relational table. It is the most general, since it can represent any XML document structure without loss of information.

Unlike the node table that is normally used in mid-tier mappings (see Chapter 6), such node tables may be combined with additional mechanisms such as clustered keys based on a hierarchical Dewey-decimal numbering system (see Figure 7.1 for an example), tokenization of types and names, and additional path and value indices. The Dewey numbered keys, for example, provide document order and physical nearness of parent nodes and their children; combined with the cluster on the keys, this can provide very efficient tree traversals and subtree retrievals. These mechanisms and the ability to map XQuery directly into physical operations instead of SQL queries overcome many of the disadvantages of the mid-tier node-table approach.

Updates generally encounter no problems, although the keys based on the Dewey-decimal system must be managed carefully to keep them

balanced. Thus, some variants on the Dewey numbering provide improved key management in the context of updates. Such node-tables are normally not directly exposed to the relational user but hidden behind the abstraction of the XML datatype.

### Table Shredding

If additional structural information is available in the form of a schema, we can use the schema information along with potential annotations to provide *mappings into related tables*, which is colloquially called *shredding* (see Figure 7.2). Chapter 6 provides a more detailed overview of such shredding-based approaches. Such tables can be exposed to relational users for relational processing or remain hidden behind the XML datatype abstraction. However, this approach still requires additional hidden information that encodes the original order information and therefore does not work well with data formats that are not easily mapped into the relational or extended-relational (e.g., nested) tables of the database system.

XML (untyped)

```
<doc>
  <Customer cid="ALFKI">
    <Order oid="01"/>
  </Customer>
  <Customer cid="BONOD"/>
</doc>
```

XML node table

id	parentid	nodetype	localname	ns-uri	datatype	value
1	0	E	doc	NULL	NULL	NULL
1.1	1	E	Customer	NULL	NULL	NULL
1.1.1	1.1	A	cid	NULL	$\frac{\text{xd}}{\text{xd}}:\text{untypedAtomic}$	ALFKI
1.1.2	1.1	E	Order	NULL	NULL	NULL
1.1.2.1	1.1.2	A	oid	NULL	$\frac{\text{xd}}{\text{xd}}:\text{untypedAtomic}$	01
1.2	1	E	Customer	NULL	NULL	NULL
1.2.1	1.2	A	cid	NULL	$\frac{\text{xd}}{\text{xd}}:\text{untypedAtomic}$	BONOD

**Figure 7.1** An XML Datatype Node-Table Format with Dewey-Decimal-Numbered Keys

XML with a schema that describes Customer and Order

```
<Customer cid="ALFKI" customername="Alfred's Futterkiste">
  <Order oid="01" orderdate="2002-10-10"/>
</Customer>
<Customer cid="BONOD" customername="Bon Odessa"/>
```

Relational tables

Customer		Order	
cid	customername	oid	orderdate
ALFKI	Alfred's Futterkiste	01	2002-10-10
BONOD	Bon Odessa		

**Figure 7.2** An XML Datatype as Virtual Schema-Driven View

### Combinations

Of course these different physical models can be combined. For example, a BLOB approach may provide explicit, XML schema-driven indices that map structured XML into relational tables and provide an index based on the node-table approach for unstructured XML.

## Encodings and Collations

XML documents normally carry their encodings with them in the XML declaration (or are defaulted to UTF-8 or UTF-16). The XML parser uses that encoding information to map the encoded character stream into the Unicode code points of the character information items of the XML Infoset.

In relational systems, information is available about the character set and collation used by the database system for storing, indexing, and comparing character data. Collation information describes the equivalence of different code points, characters, and glyphs for different languages and uses. Because it indicates what language ordering and other comparison options, such as case-sensitivity, should be applied when comparing character data, it is important for both the query semantics and for indexing.

In relational database systems, collations are attached to data (e.g., to table columns), whereas the XML model does not provide a way to do this. Instead, XQuery attaches collations to individual string operations. For example, it is possible to sort the same data in two different ways, with two different collations, once for case-sensitive order, and once for case-insensitive order. The impedance mismatch between these two approaches leads to some potential confusion. Relational database systems can make different collation information available for storage and indexing, whereas XQuery assumes a default collation and provides overriding capability only in the operational context.

When storing XML, the `XML` datatype as described in SQL-2003 [SQL2003] translates the given XML encoding into Unicode following the Infoset model. The SQL-2003 standard is currently only concerned with storing and retrieving full instances of XML and does not provide for collation information on the XML datatype. However implementations can associate the relational default collation with the XML data and make it the default collation when executing XQuery expressions on it.

For schema-directed mappings, a relational system may also use schema annotations in the mapping to identify character sets and collations to be used when mapping string data to relational columns. It usually also provides a way to directly associate a collation to the XML datatype that is to be used for indexing and querying as in the following example:

```
CREATE TABLE Order (  
    id int,  
    orderdate date,  
    PODetail XML COLLATION EN-US-CASE-SENSITIVE  
)
```

The collation information provided when storing the XML can then be picked up as the default collation used by the XQuery expressions (see below for more details). This enables the queries to use the collation-specific indices.

## Typing an XML Datatype

One of the important features of XQuery and its data model is the ability to use XML schema information to operate on typed data. Thus, we must

be able not only to validate or physically map a relational XML datatype, but to type its value according to an XML schema.

Several schema languages could be used to perform the typing, as long as they map into the type system expected by XQuery; here we use the W3C XML Schema language as our XML schema language in order to illustrate the necessary capabilities. This also appears to be the primarily supported XML schema format in all the major XML-enabled relational database systems. In order to provide type information from an XML schema, the type-relevant information, such as the attribute, element, and type declarations contained in the schema, must be managed in the metadata component of the database.

W3C XML Schema information is organized and scoped according to target namespace URIs and schema location hints. On the other hand, the naming and scoping of relational database systems components are based on a variant of dot-separated SQL identifiers that identify the catalog, schema, and schema object. For example, the table `Customer` in the catalog `MyCat` and the schema `MySchema` have the three-part name `MyCat.MySchema.Customer`. Since identifiers fit into the relational naming system but target namespace URIs do not, a relational system usually provides a mechanism to map the target namespaces to SQL identifiers.

For example, a schema could be registered using the data-definition statement given in the following example, which registers the content of the schema under the target namespace `http://www.example.com/po-schema` and any of its imported schemata and gives it the SQL identifier `POSchema`.

```
CREATE XML SCHEMA POSchema NAMESPACE
  N'<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace = "http://www.example.com/po-schema">
    ...
  </schema>'
```

This identifier is then used in the SQL context to refer to the registered XML schemata, whereas the target namespace URIs continue to be used in the XML contexts such as XML Schema and XQuery. The schema management component must also provide for some schema evolution capability, which is outside of the scope of this discussion.

Once the relational system provides for an XML schema repository, these schemata can be used for validating individual XML datatype instances, constraining the value of XML-typed columns, and providing type information to XQuery. XML Schema validation offers two functions:

- *Basic validation*: verifies conformance to the constraints of the document class described in the schema
- *typing*: identifies the datatypes of the document's information items during the generation of the *Post-Schema-Validation Information Set* (PSVI)

It also offers three validation modes:

- *Skip validation*: performs no validation (and thus does not need to be provided as an explicit option)
- *Lax validation*: validates a subtree only if it finds an applicable schema component
- *Strict validation*: requires that all data conform to the schema

Basic validation provides functionality that is often needed in the context of relational check constraints during table definitions and in relational query predicates. Relational systems therefore usually provide a validation function such as `isvalid(XML, SchemaComponents, 'lax'|'strict')→boolean` that takes an XML instance, schema component identifiers such as the schema's SQL identifier, and the validation mode, and returns a `true` or `false` result depending on whether the validation succeeded.

Users may want to either validate against the collection of schemata described by the schema's SQL identifier, or restrict it to a single namespace contained in the schema collection or even a single element. Thus the schema-component argument may provide a combination of the SQL identifier with optional indication of a namespace and an element name.

While an instance of type XML can be typed via an XML Schema using a dynamic validate operator, it seems more appropriate to make an XML Schema-constrained XML datatype a distinct SQL type. Using the validate operation would require a dynamic association and would not provide enough static information for any form of static processing, while the second approach provides a static association that can be used to perform static processing.

The static association allows type information to be used for static constraints of the content of XML columns and variables, to provide additional static hints for storage, and to be used by the static phase of query processing, including XQuery's static typing. We are going to indicate a typed XML type in the following form (as always, the actual syntax may look different): XML TYPED AS SchemaComponents where SchemaComponents denotes a schema collection identifier and, optionally, a namespace or top-level element contained in the schema collection. Listing 7.4 gives an example of a relational table that contains two XML-typed columns. The PurchaseOrders column is statically constrained by an XML Schema, and the OrderForm column is dynamically validated by schema information given in another column using a table-level check constraint.

**Listing 7.4** Example of XML-Typed Columns

```
CREATE TABLE Customers (  
    CustomerID int PRIMARY KEY,  
    CustomerName nvarchar(100),  
    PurchaseOrders XML TYPED AS POSchema,  
    OrderForm XML,  
    OrderFormType nvarchar(1000)  
    CHECK isvalid(OrderForm, OrderFormType, 'strict')  
)
```

The main difference between the handling of the two columns is that the statically typed column can take advantage of the XML Schema information when mapping to the physical level. It can be used to map the relationally structured parts of the XML document to relational tables, fold simple-typed values into the element nodes instead of keeping them as separate text nodes, or provide additional information to design path or value indices. The dynamically constrained column, on the other hand, cannot provide such static information, and physical designs normally do not take such dynamic information into account. In addition, static and dynamic constraints affect the type information accessible to queries over the XML datatypes (see below).

In order to type and validate an “untyped” XML datatype instance or retype/revalidate an already typed instance, we can use the SQL casting functionality, as shown in the following example, to cast from the unconstrained to the statically constrained XML type:

```
SELECT CAST(OrderForm AS  
           XML TYPED AS POSchema)  
FROM Customers
```

## Other Aspects of the XML Datatype

There are many other aspects of the `XML` datatype that are not directly related to its queryability and thus outside of the scope of this chapter. Among them are SQL functions defined on the `XML` datatype such as parsing, serializing or concatenation, the issues of how the database APIs provide access to `XML` datatype instances and columns, and how to cast to and from the `XML` datatype. For example, should casting a string to `XML` be equivalent to parsing the string, and casting from `XML` to string be equivalent to serializing it?

## Integrating XQuery and SQL: Querying XML Datatypes

Once the `XML` data is safely stored in a relational table, the easiest way to get at the data is to use SQL to query the table and retrieve the `XML`-typed column, as in the following example:

```
SELECT CustomerName, OrderForm
FROM Customers
WHERE CustomerID = 1
```

However, SQL per se cannot query information inside an `XML` datatype instance. That is where XQuery comes in. XQuery allows us to query and transform the `XML` data. What we need to understand now is how to integrate the two, so that we can invoke XQuery functionality from SQL, and also provide information from the relational environment to the XQuery context.

This section introduces three ways to query `XML` datatype instances in the context of the SQL query language. It explains how the XQuery expressions access both static and dynamic information provided by the SQL context, including data stored in other SQL columns.

To understand how we combine relational queries and XQuery, we must first have a conceptual understanding of how relational queries operate. First, all tables in the `FROM` clause are combined into a larger relational

table using either a Cartesian product (if they are combined using a comma) or the given join operation (using the SQL-92 join syntax [SQL92]). For example, the query shown below produces a table consisting of the Cartesian product of Table A and the left outer join between B and C:

```
SELECT A.*, function(B.b) as b
FROM A, B LEFT OUTER JOIN C ON B.id = C.b
WHERE C.id = 1 AND A.id = B.a
```

Second, the predicate expression given in the `WHERE` clause is applied to each tuple in the table. Finally, the `SELECT` clause indicates the columns of the tuples satisfying the predicate filter that will be returned in the final resulting table (potentially after applying a final transformation).

The relational processing model is very close to the conceptual processing model of XQuery. However, there are some fundamental differences such as the typing rules based on a different type system and the handling of order. Unless an explicit order is given in the SQL statement, the SQL query processor is free to neglect the order of the tuples, which enables the query-execution engine to apply algorithms that may reorder the tuples in order to achieve a more efficient execution.

SQL is also a strongly and statically typed language on the relational type system, which means that the inferred relational type of each expression must be known at query compilation (or static analysis) time.

## XQuery Functionality in SQL

The SQL query-processing model together with static typing indicate that the following query functionality should be provided in order to support XQuery in the context of SQL:

### *Transforming XML Datatype Instances*

We must be able to transform an XML datatype instance to another XML datatype. This functionality is usually used in the SQL `SELECT` clause; it makes use of the XQuery construction functionality.

### *Extracting a Value into the SQL Value Space*

We must have a way to extract information from an XML instance that fits into the SQL type system:

- Values that fit into the scalar SQL types, such as character types and numeric types (e.g., integer, decimal, float, etc.)
- XML nodes and subtrees that will be represented with the XML datatype

### *Testing the XML Structure for Existence*

We need a way to test for the presence or absence of nodes and values based on a query expression. While this functionality could be provided using the XQuery function `empty()` and the value-extraction function, having such a function available at the SQL level seems more user-friendly. The following functions provide the functionality specified in the above sections:

- `query(XML, XQueryString) → XML`

This function applies the XQuery expression instance and returns an XML type instance. While the input instance can be statically constrained, the result is always untyped (an additional parameter could provide the expected schema type). Since the result type has to be a valid instance of the XML datatype, XQuery expressions that return attribute nodes at the top level must raise an error, while queries that return values must convert them into top-level text nodes. This function provides the mechanism to return subtrees and to transform the XML instance into a different shape (see Listing 7.5 for an example).

#### **Listing 7.5** Transforming the XML Column

```
SELECT CustomerName, query(PurchaseOrders,
    'declare namespace po = "http://www.example.com/po-schema"
    for $p in /po:purchase-order
    where $p/@orderdate < xs:date("2002-10-31")
    return
        <purchaseorder date="{ $p/@orderdate }">{
            $p/*
        }</purchaseorder>')
FROM Customers
WHERE CustomerID = 1
```

■ `value(XML, XQueryString, SQLType) → SQLType`

This function applies the XQuery expression to the XML type instance and returns a relational value typed according to the SQL type indicated with the third argument `SQLType`. Since the value's SQL type must be statically known to the SQL compiler, the type must be provided as either an SQL type name or a constant string.

The XML data model value calculated by the XQuery expression is converted to the SQL value by converting the lexical representation of the XML value to the required SQL type. If the XQuery expression results in a node, the `data()` function is explicitly applied. If the XQuery result is an empty sequence, the result is the relational `NULL` value. If the result is a sequence of values and the SQL type is a single-column table type, the result is a table.

This function provides the mechanism to return values that can then be used in SQL expressions or select clauses. The following example shows how two XQuery expressions are used to extract a calculated integer value and extract the value of an attribute as an SQL date value, which are then used in an SQL query.

```
SELECT CustomerName, value(PurchaseOrders,
    'declare namespace po = "http://www.example.com/po-schema"
    count(/po:purchase-order)', integer) as ordercount
FROM Customers
WHERE value(PurchaseOrders,
    'declare namespace po = "http://www.example.com/po-schema"
    /po:purchase-order[1]/@orderdate', date) < date(2002-10-31)
```

■ `exists(XML, XQueryString) → boolean`

This function applies the XQuery expression to the XML type instance and returns `true` if the result of the expression is not the empty sequence and `false` otherwise, as shown in the following example:

```
SELECT CustomerName,
    CAST(OrderForm AS
        XML TYPED AS 'http://www.example.com/orderform-schema1')
    as OrderForm1
FROM Customers
WHERE exists(OrderForm,
    'declare namespace o = "http://www.example.com/orderform-schema1"
    /o:order')
```

### ***Setting the Dynamic Context Item in the XQuery Functions***

Since the XML datatype always provides the extended document information item as the XQuery's context item, there is no need to use the `doc()` or `collection()` functions to refer to data. On the contrary, besides being problematic in the context of the SQL processing model, these two functions would need a URI-based naming scheme for documents and collections that again would need to be integrated with the relational naming framework. Thus, all XPath expressions in these embedded XQuery expressions can start directly with the starting slash.

### ***Compiling the XQuery Expressions***

The XQuery expressions could be given dynamically or as constant strings. If the XQuery expression is given as a constant string, then relational systems can compile the XQuery at the same time as the SQL statement.

### **Augmenting the XQuery Static Context**

XQuery has a static context that provides several predefined bindings for such things as mapping namespace prefixes to namespace URIs, built-in functions and variables (such as the XQuery functions), built-in schema types (such as the XML Schema and XQuery built-in types), and the default collation. When integrating XQuery into the relational framework of an XML datatype and SQL, the relational system can add some additional static bindings to simplify the query writer's task.

First, the system can predefine some namespace bindings for SQL-related functions and types. For the purpose of this chapter, we assume that the following prefixes are added to the predefined static namespace context:

- The prefix `sql` is bound to the namespace URI representing the relational database system's built-in functions.
- The prefix `sqltypes` is bound to the namespace URI representing the mapping of the relational types into the XQuery's type model.

In addition, the query writer can provide predefined namespace bindings using the SQL/XML extension to the SQL `WITH` clause as described in the SQL-2003 standard.

Next, the XQuery's default collation is implicitly set to the relational collation in effect for the XML datatype (either from the database system's default or the collation associated with the XML datatype). The functions belonging to the namespace referenced with the prefix `sql` are added to the static function context, as are constructor functions for the SQL built-in types. The SQL built-in types are added to the static type context, as are the types of the schema components in case of a statically constrained XML datatype.

Note that for the schema types that are used to constrain the XML datatype, the namespace prefix also needs to be provided. If only one schema is being associated with an XML datatype instance, then the system could associate a predefined prefix such as `data` or the default namespace with the schema's target namespace URI. If multiple schemata can be used to constrain an XML datatype, the query writers will still need to provide the explicit namespace declarations in the XQuery prolog or use the SQL/XML extension to the SQL `WITH` clause.

Having the XQuery static context extended implicitly in the way described above removes a large amount of syntactic repetition from the embedded XQuery prologs.

## Providing Access to SQL Data inside XQuery

The functions above are capable of extracting XML values into the relational processing context. Sometimes, however, an XQuery expression needs to access data from the relational realm. If the relational database system provides variables, then the XQuery system probably wants to provide access to the variables. In any case, we also need access to the columns of the `SELECT` statement tuple set. This section presents the mechanisms to access relational data in the context of XQuery and uses the same mechanisms to provide access to other XML datatype instances to allow joining two XML datatype instances.

Access to variables can be provided in XQuery in two ways: either by mapping the relational variables directly into the static variable context (and mapping the relational types into their equivalent XML Schema/XQuery types) or by using a functional approach. In the first case, the variables should belong to the `sql` namespace. However, since SQL names allow more of the Unicode code points as name characters than XML names, such characters need to be encoded. For example, an XML name cannot contain a space (code point 0x20) but an SQL name can. The SQL/XML part of the SQL-2003 standard provides such a mapping by encoding invalid XML name characters into `_xHHHH_`, where `HHHH` denotes the Unicode code point of the invalid XML character (e.g., `_x0020_` for a space). Since a user would have to remember to encode such names, a more convenient alternative would be to provide a built-in pseudo-function `sql:variable()` that takes the name of the SQL variable as a string constant argument and returns the variable's value with its type mapped to the XML Schema/XQuery type. This approach requires more typing by the user, but it avoids the name-encoding issues.

Access to a column of an SQL statement during the query execution is very useful for integrating relational values into an XQuery. The access can be provided by a built-in pseudo-function called `sql:column()` that takes an SQL column identifier as a constant string argument and refers to the corresponding cell of the tuple that the SQL statement's iterator is operating on. Listing 7.6 gives an example of a simple XQuery expression and how the column-reference function accesses the relational values. Both `sql:column()` and `sql:variable()` can refer to an instance of the XML datatype. This allows joins across different XML datatype instances in any of the query functions above. The two functions operate equivalent to the built-in XQuery function `fn:doc()` in that they return the document node of the other XML datatype instance.

**Listing 7.6** Example of `sql:column()` Usage

Table A:		Table B:	
id:int	v:int	id:int	v:int
1	42	1	1
2	44	2	3
3	46	4	7

**Listing 7.6** Example of `sql:column()` Usage (*continued*)

```

Query:
SELECT A.id, query(CAST('' AS XML),
  '<A><v>{sql:column("A.v")}</v></A>,
  if (not(empty(sql:column("B.v")))
  then
    <B><v>{sql:column("B.v")}</v></B>
  else ())'
  as values
FROM A LEFT OUTER JOIN B ON A.id=B.id

```

```

Result:
id:int      values:XML

```

1	<A><v>42</v></A><B><v>1</v></B>
2	<A><v>44</v></A><B><v>3</v></B>
3	<A><v>46</v></A>

**Mapping SQL Types to XML Schema Types**

In order to be able to import the relational values into the XQuery context, the SQL types must be mapped to XML Schema types that describe the SQL types for the XQuery type system. Since XML Schema does not allow the addition of new primitive simple types as direct derivations from `xs:anySimpleType`, these types must be added as derivations from the built-in XML Schema simple types. The SQL/XML part of the ISO SQL-2003 standard [SQL2003] provides such a mapping that maps to the most appropriate derivation of the XML Schema simple types by using the type facets to model the range and conditions of the SQL types. Additionally, it allows relational systems to provide additional information about the original SQL type using the XML Schema annotation mechanism. An example of such a mapping follows.

```

Relational Type: INTEGER
                  (implementation range is [-2147483648,2147483647])
maps to:
<xs:simpletype name="INTEGER">
  <xs:restriction base="xsd:int"/>
</xs:simpletype>

```

However, given that many relational systems provide additional built-in SQL types, the user must expect relational systems to provide mappings that describe the implementation types; thus, the user should probably

provide the types in implementation-specific namespaces. Also, the current ISO SQL/XML mapping has a severe shortcoming in that it maps the character string types into individual XML Schema types, one for each length that is directly derived from `xs:string`. This not only means that there are way too many individual string types, but also that they are not type-compatible with each other since they each belong to a separate subtype tree of `xs:string`. Thus, we should expect built-in functions that return mapped SQL string types to return the main relational character types as simple derivations, such as `sqltypes:nvarchar` being derived from `xs:string`, disregarding the length component.

Once the SQL types are mapped into XML Schema simple types, mapping values in general becomes simple. The SQL/XML part of the ISO SQL-2003 standard provides detailed rules for the value mapping. However, bear in mind that there are still certain SQL character type values that are allowed in SQL but are considered invalid XML characters. For example, most of the low-range ASCII control characters such as ETX (code point 0x3) or BELL (code point 0x7) are not allowed in XML documents. Thus the integration must take them into account and provide correct error handling.

## Adding XQuery Function Libraries

XQuery provides a way of importing externally defined XQuery function libraries. The import mechanism follows a model based on namespace URIs and location hints similar to the XML Schema import model. Given this analogy, a relational system can therefore use a similar mechanism to provide storage of XQuery function libraries by extending the metadata to allow them to be stored according to their namespace URI or an SQL identifier. Such libraries would then be loaded and stored in a precompiled format and imported into the XQuery static context when referred to by the user. Listing 7.7 gives an example in which the function library is defined using a target namespace and then used inside a query.

### Listing 7.7 Example of an XQuery Function Library

```
CREATE XML FUNCTION NAMESPACE
  N'module "http://www.example.com/myfns"
  declare namespace myf = "http://www.example.com/myfns"
```

**Listing 7.7** Example of an XQuery Function Library (*continued*)

```

define function myf:in-King-County-WA
  ($zip as xs:integer)
  as xs:boolean
  {$zip < 98100 and $zip>=98000 }

define function myf:King-County-WA-salestax($x as xs:decimal)
  as xs:decimal
  {$x * 0.088}

SELECT CustomerName, query(PurchaseOrder,
  'import module namespace myf="http://www.example.com/myfns"
  declare namespace po = "http://www.example.com/po-schema"

  for $p in /po:purchase-order, $d in $p/po:details
  let $net as xs:decimal := $d/@qty * $d/@price
  where myf:in-King-County-WA($p/po:shipTo/po:zip)
  return
    <po-detail id="{ $p/@id }"
      total="{ $net +
        myf:King-County-WA-salestax($net) }"/>'
  as po-detail-price
FROM Customers

```

## A Note on the XQuery Data Modification Language

XQuery has not yet defined an update language, but it will undoubtedly do so. When that occurs, we will require a way to integrate that update capability with the SQL data modification language. Simply populating a column cell with an XML datatype instance can be performed using the relational insertion (see Listing 7.8, which assumes an implicit cast from string to XML that parses the data). However, updating XML also requires the ability to make finer-grained, node-level changes to an XML column. We assume that this is done via a relational update to the XML datatype instance combined with a new **mutator**: `modify(XML, XQueryDMLString)`, where `XQueryDMLString` is expressed in the as-yet-undefined update language for XQuery. In the following examples, we use an update language extension to XQuery. Other update languages, such as so-called updategrams [RYS2001], will probably be provided as well in similar fashion.

**Listing 7.8** Inserting XML Data

```

INSERT INTO Customers VALUES (42, N'Jane Doe',
    N'<po:purchase-order
      xmlns:po="http://www.example.com/po-schema" id="2000">
      <po:shipTo>
        <po:address>42 Main Street</po:address>
        <po:city>Redmond</po:city>
        <po:state>WA</po:state>
        <po:zip>98052</po:zip>
      </po:shipTo>
      <po:details qty="1" price="23.44"/>
    </po:purchase-order>', NULL, NULL)

```

Since an XML datatype is a relational column type, even a node-level change will have to lock the whole XML datatype either directly or indirectly via the row locks at the isolation level of the relational system's context. Making the concurrency control aware of the tree structure of the XML datatype instances will be certainly an important research area to improve the performance. The following is an example of how `po:detail` elements may be deleted from the `PurchaseOrders` XML instances. The concurrency control will not only lock the parent of the deleted subtrees but the whole XML instance of all the rows that satisfy the update statements where clause.

```

UPDATE Customers
SET modify(PurchaseOrders,
    'declare namespace po = "http://www.example.com/po-schema"
    delete /po:purchase-order/po:details'
WHERE exists(PurchaseOrders,
    'declare namespace po = "http://www.example.com/po-schema"
    /po:purchase-order[po:details and po:shipTo/po:zip=98052]')

```

If the XML datatype is constrained by an XML Schema, then updates may fail if they would result in an instance that violates the XML Schema constraints.

## Physical Mappings of XQuery

The previous sections have presented the functional description of how to integrate XQuery and XML into the SQL processing model of relational systems. From a performance point of view, another interesting aspect is how such combined queries are being mapped into a physical

operator tree. Since this mapping depends on the physical mapping of the XML datatype and also depends heavily on the vendor-specific architectures, this section provides a more general, higher-level discussion of the different mapping strategies of XQuery into physical execution plans.

In the following, we assume that the XQuery expressions are provided as constants; thus, compilation can occur at the same time as the compilation of the SQL expressions. The handling of parameterized queries is left as an exercise for the reader. The general processing model of an SQL statement is as follows (individual steps may vary in actual implementations; e.g., a logical operator tree may contain more or less physical information):

■ Static compilation phase:

1. Parse the statement into an abstract syntax tree (AST).
2. Normalize the AST (example: perform the implicit conversions of single-cell rowsets into scalar values) and check it for type-correctness.
3. Convert the AST into a logical operator tree representing the relational algebra of the implementation engine. A logical operator tree may indicate a join operation, but does not indicate, for example, which physical join implementation is used.
4. Store the operator tree for later dynamic processing after potentially applying some additional static rewrite rules.

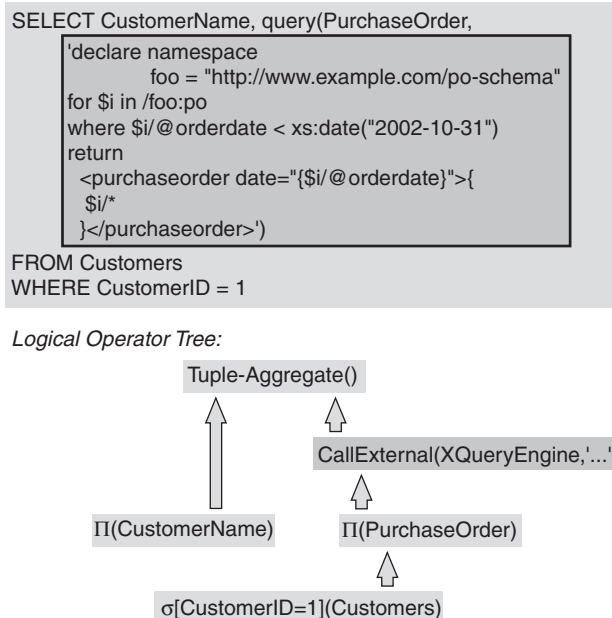
■ Dynamic execution phase:

1. Compile the logical operator tree into a physical operator tree using dynamic statistics such as costing information. This transformation may reorder operations and will choose among the different access methods (e.g., scan vs. index) and join methods.
2. Execute the physical operator tree.

There are two major approaches to compiling and executing XQuery expressions in the context of SQL statements: a *decoupled* and an *integrated* approach. The *decoupled approach* works with a standalone XQuery engine that is added to the relational system. It is usually employed when the XML datatype is being modeled as a user-defined datatype that appears as a binary object to the relational processor. In this case, the

query function is processed as is any other external function in the SQL compilation: The XQuery expression is passed to the XQuery processor outside of the general SQL processing model, the query is processed, and the result is returned to the SQL environment. If XQuery has been extended with access to the relational processing environment (for example, the `sql:column()` function mentioned above), then the external XQuery engine must be able to call back into the relational processing environment to access the data. Figure 7.3 shows the decoupled approach in form of a block diagram. It uses the Greek symbols  $\sigma$  and  $\Pi$  for selection and projection respectively.

The *integrated approach* maps the XQuery expressions into logical operator trees that are integrated with the logical operator tree of the SQL statement. This approach requires that the physical design of the XML datatype use one of the relational mappings described above. Note that such a relational physical design can occur as the primary storage or as part of an indexing scheme on an LOB-based storage model or even a just-in-time shredding of such an LOB when processing XQueries. The XQuery



**Figure 7.3** Decoupled XQuery Processing Approach

logical operator tree must consist of operations that the relational query processor understands such as projections ( $\Pi$ ), selections ( $\sigma$ ), and aggregations. Figure 7.4 provides the block diagram of the integrated approach.

Note that the integrated approach is not as simple as it sounds. First, since XQuery has a more complex, nested and sequence-based data model, the relational algebra must be extended with operations that capture the necessary semantics, such as nest/unnest and document-ordering. Furthermore, the XML Schema built-in scalar datatypes and their operations do not normally map one-to-one to the built-in relational scalar datatypes and their operations. For example, relational string comparisons may disregard trailing spaces during string comparisons, whereas XQuery's string compare does not, or the relational expression service that provides the operation semantics may deal only with true scalar types, while the XML Schema types may be not truly scalar, as is true of the XML date and time datatypes that provide a (value, timezone) pair. Thus relational systems must extend their expression services to deal with the new datatypes and different operational semantics.

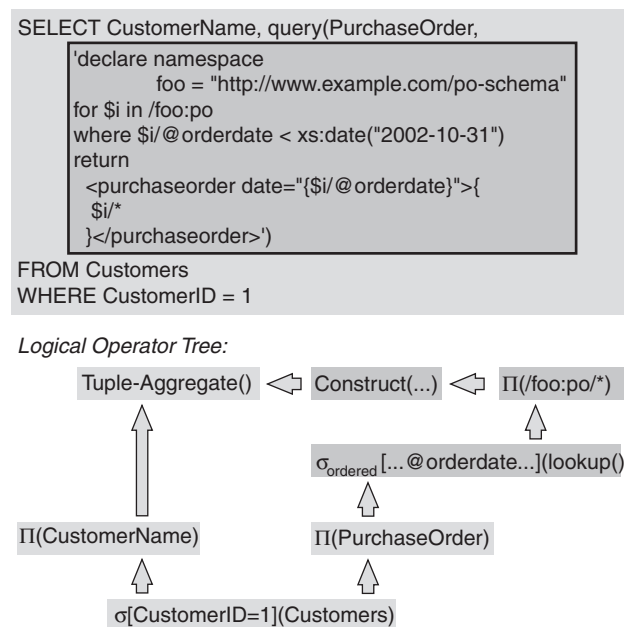


Figure 7.4 Integrated XQuery Processing Approach

Finally, integrating XQuery processing into the SQL processing model also means that the cost-based optimizer will have to choose execution plans that preserve the XQuery semantics and understand potentially additional cost factors. One important aspect that affects the execution plan is the preservation of document and input order. An optimizer must either choose only order-preserving `join` and `set` algorithms or add specific `reorder` operators to allow the use of more efficient algorithms that are not order-aware.

Another area that makes the XQuery evaluation more complex than the evaluation of SQL statements is the execution order. The presence of an index often prompts the optimizer to choose a so-called bottom-up evaluation strategy, where first indices are used to filter the processed tuples before any of the other operations are evaluated. Since the naïve execution strategy of XQuery is described top-down, an optimizer may produce dynamic errors by reordering the evaluation order that the top-down evaluation strategy would have avoided. A simple example of this situation appears in the following XQuery expression:

```
for $i in //A
where $i/@a castable as xs:integer
return
  for $j in $i/B
  where xs:integer($i/@a) > 10
  return
    $j
```

A naïve, top-down evaluation of the nested `for` expression would only execute the inner `for` expression if the value of `$i/@a` is really castable to `xs:integer`. However, an optimizer may choose to rewrite the above query to the following equivalent:

```
for $i in //A, $j in $i/B
where $i/@a castable as xs:integer and xs:integer($i/@a) > 10
return
  $j
```

and then choose to execute the comparison before the check for castability. If then there are `a` attributes that are not castable to `xs:integer`, the cast in the comparison will fail with a runtime error that in the naïve evaluation would have been avoided.

Note that this type of behavior can already occur in processing normal SQL statements without XQuery—although less frequently. The XQuery language specification also recognizes the benefits of processing optimizations and thus indeed allows these rewrites. However, an XQuery system that integrates with such an optimizer should provide an optimization hint that can force the result of the naive top-down evaluation strategy to provide the query writer with the option to force a correct, error-free evaluation.

### **Issues of Combining SQL, XML Datatype, and XQuery**

Combining XQuery and SQL for querying XML datatype instances is indeed a powerful combination of the two languages. However, some major issues arise with this approach:

- XQuery users interested in querying across multiple documents need to understand and use SQL to iterate over the collection of documents and to use SQL joins to be able to join among two different documents.
- SQL users interested in querying into XML documents need to learn the new XQuery language.

Since the languages are closely related, somebody familiar with one language should be able to learn the other easily. However, the need to know two languages is still an additional level of complexity that future relational systems will need to address. Today's relational database systems address the second problem by shredding the XML data into relational tables that can then be queried relationally, thus replacing the need for a second query language with a simpler mapping approach. However, this mapping approach often only provides relational fidelity and thus does not address order preservation and markup scenarios. Thus either the SQL model itself must be extended to deal with the nonrelational aspects of order and heterogeneity (this still means that a query writer would have to learn and understand these extensions), or the SQL model will eventually be subsumed under the XML and XQuery model (which would mean that a large number of people would have to learn this new

model). Relational database systems will probably provide a combination of the two approaches, thus providing both an SQL- and an XQuery-based approach, each addressing a different skill set.

## Top-Level XQuery

Top-level XQuery provides a way to query collections of XML documents and forests, and XML views of relational data without the need to use SQL. It becomes the top-level language that is used as the primary query language at the same level as SQL is being used for relational data. It basically subsumes SQL under the XML and XQuery model and enables XML-savvy users to operate on both relational and XML data without having to use SQL. Unlike queries on an XML datatype, XQuery expressions can combine different XML documents and XML views of relational data. Another advantage of having top-level XQuery expressions is the ability to provide finer-grained concurrency control and locking (such as node-level tree locking).

Many relational database systems and tools providers offer this approach as a mid-tier extension, providing views over the relational data and an XQuery data integration engine (see Chapter 6). However, if a relational database system already provides an XML datatype, then it seems straightforward to extend the programming model to provide top-level XQuery support over a collection of XML instances and XML views of relational data. In the following, we use the keyword `XQUERY` to indicate a top-level XQuery and enclose the XQuery expression in curly braces. Such an expression returns either a single instance of the relational XML datatype or (depending on the relational database system's programming model) a single-column/single-row table or a singleton collection of XML.

## XML Document (Fragment) Collections

There are two ways to provide (potentially typed) collections of XML data. One approach simply uses an XQuery function in the top-level XQuery expression to refer to a table column that is typed as XML. The XQuery `collection()` function could be used to refer to such a column, if the relational system provided a mapping of the relational names into the URI space.

Alternatively, the database could provide a function `sql:collection($c as xs:string) as document*` that refers to the column and returns a sequence of the extended document root nodes. An example based on the table specified in Listing 7.4 is given below. If the column contains typed XML, then the static type information can be used to type the top-level XQuery.

```
XQUERY {
  declare namespace po = "http://www.example.com/po-schema"
  sql:collection('Customers.PurchaseOrders')/po:po/details
}
```

Another approach is to extend the notion of a table to allow the creation of a table of `type`, where the type would not be a rowtype but the XML datatype. The example in Listing 7.9 first defines a collection of purchase-order instances statically constrained by the purchase-order schema; then it provides two alternate ways to populate it (one using an SQL `INSERT` statement and one using a pseudo-XQuery-like syntax), and finally it queries the collection.

**Listing 7.9** Example of an XML Collection Using `CREATE TABLE OF type` and a Top-Level XQuery

```
CREATE TABLE PurchaseOrders OF TYPE XML TYPED AS POSchema

INSERT INTO PurchaseOrders SELECT PurchaseOrders FROM Customers

XQUERY-MODIFY {
  insert
    <po:purchase-order
      xmlns:po="http://www.example.com/po-schema" id="2001">
    <po:shipTo>
      <po:address>42 2nd Avenue</po:address>
      <po:city>Bellevue</po:city>
      <po:state>WA</po:state>
      <po:zip>98006</po:zip>
    </po:shipTo>
    <po:details qty="4" price="3.44"/>
  </po:purchase-order>
  as last into
    sql:collection('PurchaseOrders')
}
```

```
XQUERY {
  declare namespace po = "http://www.example.com/po-schema"
  sql:collection('PurchaseOrders')/po:purchase-order/po:details
}
```

## XML Views over Relational Data

Besides providing access to collections of XML instances, top-level XQuery expressions should also provide access to XML views over relational data. As described in Chapter 6, there are different view mechanisms.

Since the SQL/XML part of the ISO SQL-2003 standard provides two general forms of default mappings of relational tables to XML [SQLXML], relational systems will most likely provide at least one of them if they provide XQuery access to relational data. They may also provide support for other mappings, such as an annotated schema-driven view mechanism.

The SQL/XML relational-to-XML mapping provides two main mapping approaches: The table-to-doc approach maps every table to an XML document where the root element is named according to the table name. Every row is mapped to an element named `row` underneath that root element and contains the column data of the row as subelements, where the names of the subelements are based on the column names. The table-to-forest approach maps a table into an XML element forest where every row is mapped to an element whose name is based on the table name. As in the other mapping, the column data is mapped to subelements. Both approaches have additional mapping options, such as an option to map columns containing the relational `NULL` values to either absent elements or to elements marked with `xsi:nil="true"`. The table-to-doc approach is well suited for XML-based relational-to-relational data transport, whereas the table-to-forest approach is better suited for querying the relational data with XQuery, since it avoids the additional step expressions introduced by the artificial `row` elements.

Listing 7.10 gives an example of an XML view of the relational table in Listing 7.2 that uses the table-to-forest approach. Whitespace has been added to help in formatting.

**Listing 7.10** Example of an XML View of a Relational Table

```
<ORDER>
  <ID>4023</ID>
  <ORDERDATE>2001-12-01</ORDERDATE>
  <PODETAIL>
    <purchaseOrder xmlns=?http://po.example.com?>
      <originator billId=?0013579?>
```

**Listing 7.10** Example of an XML View of a Relational Table (*continued*)

```

    <contactName>Fred Allen</contactName>
    <contactAddress>
      <street>123 2nd Ave. NW</street>
      <city>Anytown</city>
      <state>OH</state>
      <zip-four>99999-1234</zip-four>
    </contactAddress>
    <phone>(330) 555-1212</phone>
    <originatorReferenceNumber>AS 1132</originatorReferenceNumber>
  </originator>
  <order>
    <item code=?34xdl 1278 12ct? quantity=?1?/>
    <item code=?57xdl 7789? quantity=?1? colorcode=?012?/>
  </order>
  <shipAddress sameAsContact=?true?/>
  <shipCode carrier=?02?/>
</purchaseOrder>
</PODETAIL>
</ORDER>
<ORDER>
  <ID>5327</ID>
  <ORDERDATE>2002-04-23</ORDERDATE>
  <PODETAIL>
    <purchaseOrder xmlns=?http://po.example.com?>
      <originator ...
    </purchaseOrder>
  </PODETAIL>
</ORDER>
...

```

The top-level XQuery environment can be given access to these views in various ways. Generally this is done either through the existing `doc()` and `collection()` functions, where the location and mapping information is encoded in the URI, or possibly through additional built-in functions.

Any of the views can be represented as a single document by making the view's top-level element(s) become the children of a document node. The result will not necessarily correspond to a well-formed XML document, but is allowed by the XQuery data model. In this case, the system must either specify a URI encoding of the view parameters for use with the XQuery `doc()` function or define a new, built-in XQuery function. For example, top-level XQuery could provide the following built-in XQuery function:

```

sql:table($table as xs:string
          [, $table_map_option as xs:string
          [, $null_map_option as xs:string]]) as element*

```

This function returns the view of the given table `$table` with the optional parameters indicating the preferred table- and null-mapping options. For instance, `sql:table('Order', 'table-to-forest', 'xsinil')` would return the view of Listing 7.10.

Alternatively, the system could also provide access to the views as a collection of individual document nodes, one for each top-level element. In that case, the system would have to provide a function `sql:collection()` or define a URI mapping for use with XQuery's built-in function `fn:collection()`. In any case, the view will probably not guarantee a deterministic order of the top-level elements, because relational tuples have no implicit order. If a relational system provides other relational-to-XML mapping techniques, such as Microsoft SQL Server's annotated schema views [RYS2001], additional functions can be provided to expose them in top-level XQuery expressions.

Turning to implementation, we note that the actual physical mapping of the XQuery expressions against the views—regardless of how they are accomplished—can be done directly against the relational engine's logical algebra operations. This means that an implementation can add new logical operations as needed to express specific semantics not available otherwise. This ability allows an easier mapping than the mid-tier approach, which must either map the query into an SQL statement or ship more data than needed to the mid-tier and post-process the data to get the correct execution semantics.

Any of these view-generation techniques present a major problem for data modification statements. Since they assume a node-based data model, where every node has a unique identification, mapping them into SQL-based update statements on relational tables that may not have such a unique identification requires a careful design of the update propagation through the view.

## Conclusion and Issues

This chapter has given an overview of how to integrate relational database systems with XQuery. It introduced the XML datatype and presented both the mixed approach of using XQuery in conjunction with

SQL to query XML datatype instances and the top-level XQuery approach of querying collections of XML instances and XML views of relational data. It also provided an insight into the impact of the actual physical data model of the XML datatype on the processing and mapping of XQuery in the context of relational systems.

Many relational database systems at the time of this writing have just started to add native XML support to their support for bidirectional mapping between relational and XML data, and so far no system ships XQuery as described in this chapter. Before long, however, I believe all major relational database systems will provide functionality of the form described (most vendors have already announced support for a built-in XML datatype).

Many open issues need further investigation and are currently active research topics. Among them are the following:

- Which of the physical mappings of the XML data are the best approaches to support XML inside a relational context? Depending on the data, there may be more than one. In that case the question becomes how to determine the best one and provide support for all the approaches.
- How should the query-processing model be extended? In particular:
  - What additions are needed to the logical algebra?
  - What additions can be made to the optimizer to improve efficiency?
  - What indices help in improving efficiency?
  - How is distributed query processing best supported?
- Another area of concern is the area of concurrency control. How can the concurrency control of relational data be extended to deal with tree-structured data and provide fine-grained control? Especially in the context of combined relational and XML queries on relational tables containing XML, the current row-level locking granularity is inadequate if access similar to **online transaction processing (OLTP)** is going to be part of the query mix. Thus, one question is how to improve the parallelism in such queries and how to integrate fine-grained concurrency control of XML data (such as tree-based concurrency control) with the relational concurrency control.

