



CHAPTER 9

Storing XML Data

158 Oracle Database 10g XML & SQL

In Oracle Database 10g, you have a number of choices for storing XML data. You can shred the XML documents and store the data in one or more relational tables, put them intact in CLOB XMLTypes, or register an XML schema and store them in an XML Schema-based XMLType with object-relational storage. If there is no requirement for updating the XML content, you can also store the XML documents externally by creating *External Tables*.

This chapter gives an overview of the XML storage options available in Oracle Database 10g and shows you various examples of how to use the technologies. You will also learn how to use the Oracle utilities including the SQL*Loader and XML SQL Utility (XSU) to load XML documents into either XMLType tables or relational tables in Oracle Database 10g. We start with the simplest storage format: the CLOB XMLTypes.

Storing XML Documents in CLOB XMLTypes

Using the CLOB XMLType, XML documents are stored as CLOBs with a set of XML interfaces provided by the XMLType. Though you can optionally carry out any XML processing during the data ingestion, such as validating the input XML against an XML schema or a DTD, the CLOB XMLType storage does not require any XML processing except well-formedness checking and entity resolution.

Updating and Querying CLOB XMLTypes

The CLOB XMLType storage best preserves the original format of XML documents and gives the maximum flexibility for XML schema evolution. However, storing XML documents in CLOB XMLTypes results in expensive processing overhead when querying the XML content, such as using the **XMLType.Extract()** or **XMLType.ExistsNode()** functions, because these operations require building an XML DOM tree in memory at run time and performing functional XPath evaluations. In addition, any update operation can be performed only at the document level. This means that you need to update the entire XML document for even a small change to one XML element. Therefore, normally you should avoid using XMLType functions to perform fine-grained XML updates or XPath-based queries on CLOB XMLTypes.

Instead, for XPath-based queries on CLOB XMLTypes, Oracle Text provides a full text search supporting a limited set of XPaths. This functionality allows you to perform XPath queries on CLOB XMLTypes utilizing the CONTEXT index created by Oracle Text, and it has proven very useful and scalable for enterprise applications, which we will discuss in Chapter 11.

Dealing with Character Encoding for CLOB XMLTypes

When storing XML documents in the Oracle database, you should know that a character set conversion is automatically performed during data insertions, which converts all the text data, including XML documents, to the database character set, except when stored as BLOB, NCHAR, or NCLOB data types.

Because of this implicit character set conversion, the actual XML data encoding and the encoding declaration in the `<?XML?>` prolog may not be the same. In the current Oracle Database 10g release,

XMLType APIs ignore the encoding declaration in the `<?XML?>` prolog and assume that XML data in CLOB XMLTypes is stored in the database character set. Therefore, when loading XML data from the client side, you need to make sure this conversion is properly performed.

To ensure proper conversion from the client character set to the database character set, you are required to set up the `NLS_LANG` environment variable to reflect the client character set encoding if the XML document is originally stored in a client character set that is different from the database character set. Otherwise, if the variable is set to be the same as the database character set, the original text will be stored as-is in the database without character validation and conversion.

In other words, if the `NLS_LANG` environment variable is not set or is set incorrectly and the XML document does not have the same encoding as the database, garbage data will be stored in the database.



NOTE

If the XML document contains characters that are invalid in the database character set, you will get an Invalid Character error during the data insertions to CLOB XMLTypes. The current solution for this is to use the NCLOB or BLOB for data storage in the database and build mid-tier XML applications or PL/SQL external procedures using the XDK APIs to process the XML data.

Because the character set conversion may result in conflict between the actual encoding and the encoding declaration in the `<?XML?>` prolog, when reading the XML data out of CLOB XMLTypes, you must do the reverse character set conversion or update the encoding declaration in the `<?XML?>` prolog to make them consistent. This is important because although an XML parser can use the first 4 bytes of the `<?XML?>` prolog to detect the encoding of XML documents, it can determine only whether the character encoding is an ASCII-based encoding or EBCDIC encoding. If it is an ASCII-based encoding, an XML parser can detect only whether it is UTF-8 or UTF-16. Otherwise, it depends on the encoding attributes in `<?XML?>`. Therefore, if you have XML documents not in UTF-8 or UTF-16 encoding, you *must* include a correct XML encoding declaration indicating which character encoding is in use, as follows:

```
<?xml version="1.0" encoding='Shift-JIS'?>
```

Storing XML Documents in XML Schema–based XMLTypes

To speed up the XPath queries and fine-grained updates on XMLTypes, you can create XML Schema–based XMLTypes. One way of doing this is to associate registered XML schemas with the XMLType columns or XMLType tables using `XMLSCHEMA`. You can also create XMLType tables by specifying the `DEFAULT TABLE` annotation in the registered XML schemas.

All these approaches create XML Schema–based XMLTypes, where sets of object-relational tables/objects are bound to the XML entities defined in the XML schema. The only difference between creating a default table during XML schema registration and using the `XMLSCHEMA` keyword is that the former approach allows XML documents conforming to the registered XML schema to be managed by Oracle XML DB repository. With the support of the XML DB repository,

160 Oracle Database 10g XML & SQL

you can not only retrieve or update XML in SQL, but also manage XML documents stored in the XML DB repository using protocol interfaces such as FTP and HTTP/WebDAV.

XML Schema Registration

XML schema registration defines the XML-to-SQL mapping and a hierarchical object-relational structure for storing XML documents in the Oracle database. We will explore this using the DEMO user and the WebDAV folder created in Chapter 8.

First, you need to copy the XML schema for customer records, **contact_simple.xsd**, into the **/public** WebDAV folder. The following is the content of this schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer" type="CustomerType"/>
  <xsd:complexType name="CustomerType">
    <xsd:sequence>
      <xsd:element name="NAME" type="xsd:string"/>
      <xsd:element name="EMAIL" type="xsd:string"/>
      <xsd:element name="ADDRESS" type="xsd:string"/>
      <xsd:element name="PHONE" type="phoneType"/>
      <xsd:element name="DESCRIPTION" type="contentType"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ContentType" mixed="true">
    <xsd:sequence>
      <xsd:any minOccurs="0" maxOccurs="unbounded" processContents="skip"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="phoneType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\(\d{3}\)\d{3}-\d{4}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

To register this XML schema to the XML DB, you can call the following PL/SQL procedure:

```
ALTER SESSION SET EVENTS='31098 trace name context forever';
BEGIN
  DBMS_XMLSCHEMA.registerURI (
    'http://localhost:8080/public/contact_simple.xsd',
    '/public/contact_simple.xsd',
    LOCAL=>TRUE, GENTYPES=>TRUE, GENBEAN=>FALSE, GENTABLES=>TRUE);
END;
```

NOTE

To use the **ALTER SESSION** command, you need to log in as **SYS** and grant the **ALTER SESSION** privilege to the **DEMO** user using "GRANT ALTER SESSION TO DEMO". Otherwise, you will get an **ORA-01031: Insufficient Privileges** error.

In the `DBMS_XMLSCHEMA.registerURI()` function, the first parameter is the schema URI, `http://localhost:8080/public/contact_simple.xsd`, which uniquely identifies the registered XML schema in the XML DB. The second parameter is an XML DB URI (XDBUri), `/public/contact_simple.xsd`, pointing to the `contact_simple.xsd` file in the `/public` folder of the XML DB repository. The following parameters control whether the XML schema is registered as a local (`LOCAL=>TRUE`) or global (`LOCAL=>FALSE`) schema, whether object types (`GENTYPES=>TRUE`) and default tables (`GENTABLES=>TRUE`) will be created. The `GENBEAN` parameter is optional and does not perform any function at this time. If the XML schema is registered as a global XML schema in the XML DB, it can be shared across different database users. Otherwise, XML schema sharing is not allowed.

You can set `GENTABLES=>FALSE` if you do not want Oracle XML DB to create default tables during the XML schema registration. In this case, you can create XMLType tables using the `XMLSCHEMA` keyword, as in:

```
CREATE TABLE customer_xmltype_tbl OF XMLTYPE
XMLSCHEMA "http://localhost:8080/public/contact_simple.xsd"
ELEMENT "Customer";
```

Additionally, you can use the following syntax to define XMLType columns using XML Schema-based storage:

```
CREATE TABLE customer_col_tbl(
  id NUMBER,
  record XMLType)
XMLTYPE COLUMN record STORE AS OBJECT RELATIONAL
XMLSCHEMA "http://localhost:8080/public/contact_simple.xsd"
ELEMENT "Customer";
```

Since the same storage techniques apply to both XMLType tables and XMLType columns, we will discuss only the details of how to use XML Schema-based XMLType tables in later sections.

During XML schema registration, you can use the following command to create a trace file in the `USER_DUMP_DIR` showing the DDLs used to create the object tables and datatypes:

```
ALTER SESSION SET EVENTS='31098' TRACE NAME CONTEXT FOREVER;
```

To locate the trace file, you need to check the current session ID by querying the `V$SESSION` and `V$PROCESS` views. Before selecting from the `V$SESSION` and `V$PROCESS` views in the `DEMO` user, you need to log in as `SYS` and grant to the `DEMO` user the `SELECT` privilege on `V_$SESSION` and `V_$PROCESS` views, as follows:

```
GRANT SELECT ON V_$SESSION TO DEMO;
GRANT SELECT ON V_$PROCESS TO DEMO;
```

NOTE

Since `V$SESSION` and `V$PROCESS` are just synonyms for the views, you cannot grant any privileges on them.

162 Oracle Database 10g XML & SQL

By issuing the following SQL command, you can find the ID of the session corresponding to the trace file:

```
SELECT a.spid
FROM V$PROCESS a, V$SESSION b
WHERE a.addr=b.paddr
AND b.audsid=userenv('sessionid');
```

This returns the following:

```
SPID
-----
2796
```

The trace file has a name structured as **orclX_ora_<Session_Id>.trc**, and you can get the **USER_DUMP_DIR** by issuing the following command in a SYS user account:

```
SQL> SHOW PARAMETERS user_dump_dest
NAME                TYPE                VALUE
-----
user_dump_dest      string              D:\ORACLE\ADMIN\ORCLX\UDUMP
```

Thus, the trace file is **orclX_ora_2796.trc**, in the **USER_DUMP_DIR** verified by running the following:

```
SQL> host ls d:\oracle\admin\orclx\udump\orclX_ora_2796.trc
orclX_ora_2796.trc
```

Because this file lists the set of DDLs used to create the object table and data types, it is a good reference when debugging XML schema registrations.

Now, let's examine the created storage structure in more detail by issuing the following command in SQL*Plus:

```
SQL> SELECT object_name, object_type
2 FROM USER_OBJECTS
3 WHERE object_name LIKE '%Customer%';
OBJECT_NAME                OBJECT_TYPE
-----
Customer260_TAB            TABLE
Customer260_TAB$xd         TRIGGER
CustomerType259_T          TYPE
```

The result shows that three objects were created during the XML schema registration. If you further examine the types and the table's definitions, you will see that the objects created are not limited to these. First, you can describe the **Customer260_TAB** table as follows:

```
SQL> DESC "Customer260_TAB";
```

This results in the following:

Chapter 9: Storing XML Data 163

Name	Null?	Type

TABLE of SYS.XMLTYPE(XMLSchema "http://localhost:8080/public/contact_simple.xsd" Element "Customer") STORAGE Object-relational TYPE "CustomerType259_T"		



NOTE

If an XML element uses mixed case or lowercase, the default table and object names by default will be case sensitive. Therefore, you need to use double quotes when referring these names, as in "Customer260_TAB".

The preceding description shows that:

- Customer260_TAB is an XMLType table.
- The XMLType objects in the table are associated with the registered XML schema, http://localhost:8080/public/contact_simple.xsd.
- The root element of the XML document is <Customer>.
- The object type used to store the XMLTypes is CustomerType259_T.

Looking at the description of CustomerType259_T, you can see that this type contains

```
SQL> DESC "CustomerType259_T"
"CustomerType259_T" is NOT FINAL
```

Name	Null?	Type

SYS_XDBPD\$		XDB.XDB\$RAW_LIST_T
NAME		VARCHAR2 (4000 CHAR)
EMAIL		VARCHAR2 (4000 CHAR)
ADDRESS		VARCHAR2 (4000 CHAR)
PHONE		VARCHAR2 (4000 CHAR)
DESCRIPTION		contentType257_T

All the XML elements in XMLTypes are mapped to the corresponding database data types. In this example, the NAME, EMAIL, ADDRESS, and PHONE elements as simple types in the XML schema are stored as VARCHAR2. Since there is no limit on the string length in the XML schema, Oracle XML DB sets 4000 characters as the default length for these columns. On the other hand, new object types are created for the complex types defined in the XML schema. In this example, contentType257_T is created to store the customer descriptions, which is further shown as follows:

```
SQL> DESC "contentType257_T";
"contentType257_T" is NOT FINAL
```

Name	Null?	Type

SYS_XDBPD\$		XDB.XDB\$RAW_LIST_T
SYS_XDBANY258\$		VARCHAR2 (4000 CHAR)

164 Oracle Database 10g XML & SQL

Note that Oracle XML DB defines the SYS_XDBANY258\$ column as a VARCHAR2 (4000) to store the <xsd:any/> element defined in the <DESCRIPTION> element. The SYS_XDBPD\$ column is a *position descriptor* column created by the XML DB to preserve the DOM fidelity of XML documents. Information, such as comments, processing instructions, namespace prefixes, and the order of sibling XML elements, is stored in this SYS_XDBPD\$ column. Therefore, this column is used to preserve the integrity of the original XML document for the DOM transversals.

To examine further details of the **Customer260_TAB** table, you can query the USER_TAB_COLS view:

```
SQL> SELECT column_name,data_type,
2      CASE WHEN hidden_column='YES' THEN 'hidden'
3            WHEN virtual_column='YES' THEN 'virtual'
4            ELSE null END as attr
5 FROM USER_TAB_COLS
6 WHERE table_name='Customer260_TAB'
7 ORDER by virtual_column desc, column_name;
```

COLUMN_NAME	DATA_TYPE	ATTR
SYS_NC_ROWINFO\$	XMLTYPE	virtual
XMLDATA	CustomerType259_T	hidden
ACLOID	RAW	hidden
OWNERID	RAW	hidden
SYS_NC00007\$	RAW	hidden
SYS_NC00014\$	RAW	hidden
SYS_NC_OID\$	RAW	hidden
SYS_NC00009\$	VARCHAR2	hidden
SYS_NC00010\$	VARCHAR2	hidden
SYS_NC00011\$	VARCHAR2	hidden
SYS_NC00012\$	VARCHAR2	hidden
SYS_NC00016\$	VARCHAR2	hidden
SYS_NC00008\$	XDB\$RAW_LIST_T	hidden
SYS_NC00015\$	XDB\$RAW_LIST_T	hidden
XMLEXTRA	XMLTYPEEXTRA	hidden
SYS_NC00004\$	XMLTYPEPEPI	hidden
SYS_NC00005\$	XMLTYPEPEPI	hidden
SYS_NC00013\$	contentType257_T	hidden

Note that the CASE expression selects a result from one or more alternatives. It uses an optional SELECTOR, to specify an expression whose value determines which alternative to return. A normal CASE expression has the following form:

```
CASE selector
  WHEN expression1 THEN result1
  WHEN expression2 THEN result2
  ...
  WHEN expressionN THEN resultN
  [ELSE resultN+1]
END;
```

From the query, you can see that the **Customer260_TAB** table contains one virtual column called **SYS_NC_ROWINFO\$** and several hidden columns, including **XMLDATA**, **ACLOID**, **OWNERID**, **XMLXTRA** and a set of **\$\$SYS_NC<number>\$** columns.

The virtual column, **SYS_NC_ROWINFO\$**, is an XMLType object that identifies the rows of the XMLType table. For example, in the triggers of XMLType tables, you can use **:new.SYS_NC_ROWINFO\$** to refer to the current row of data.

The **XMLDATA** column refers to the SQL objects used for storing the XMLTypes. It is useful when you want to query or create indexes on XMLTypes by directly working on the SQL objects. In the preceding example, **XMLDATA** is an alias for the **CustomerType259_T** object. Therefore, you can add a unique constraint on the **EMAIL** element by referring to it as **XMLDATA.EMAIL**, as follows:

```
ALTER TABLE Customer260_TAB ADD UNIQUE(XMLDATA.EMAIL);
```

The **XMLDATA.EMAIL** refers to the object storing the content of the **EMAIL** elements in the customer records. With the **UNIQUE** constraint added, if you try to insert the same customer record multiple times, you get the following error:

```
ORA-00001: unique constraint (DEMO.SYS_C003626) violated
```

Some of the hidden columns in **Customer260_TAB** are for Oracle XML DB repository use. For example, in Oracle XML DB repository, the *Access Control List (ACL)* defines the permissions for each resource. The **ACLOID** specifies the ACL permissions for the XMLType table and the **OWNERID** specifies the ID of the table owner. The other hidden columns are used to create the hierarchical relationships between XML elements.

Except for **XMLDATA** and **SYS_NC_ROWINFO\$**, you should never access or manipulate these XMLType table columns directly.

XML Schema Annotations

To control the mapping between XMLType storage and XML schemas, you need to use Oracle XML DB annotations. In Oracle Database 10g, these XML Schema annotations are a set of attributes added to an XML schema declaring the SQL object names, data types, and various storage options. All of these annotations are in the Oracle XML DB namespace, <http://xmlns.oracle.com/xdb>, normally using the **xdb** prefix. Basically, you can use these annotations to specify the following:

- **DefaultTable** The name and storage attributes of the default XMLType table storing the XML documents.
- **SQLNames** The SQL names for the XML elements defined in the XML schema.
- **SQLTypes** The names of the SQL data types used to store simple or complex data types defined in the XML schema. For an unbounded XML element mapping to a collection SQL type, **xdb:SQLCollType** is used to specify the type name.
- **MaintainDOM** The attribute that tells Oracle XML DB whether to preserve DOM fidelity of the element on output.
- **Storage Options** The XML DB annotations, such as **xdb:storeVarrayAsTable**, **xdb:mapUnboundedStringToLob**, **xdb:maintainOrder**, and **xdb:SQLInline**, specify the options for optimizing storage.

166 Oracle Database 10g XML & SQL

Let's examine the following annotated XML schema for the customer records, **customer_simple_ann.xsd**, explore some useful design techniques, and then register it to the XML DB.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xdb="http://xmlns.oracle.com/xdb" xdb:storeVarrayAsTable="true">
  <xsd:element name="Customer" type="CustomerType"
    xdb:defaultTable="CUSTOMER"/>
  <xsd:complexType name="CustomerType" xdb:maintainDOM="false">
    <xsd:sequence>
      <xsd:element name="NAME" type="xsd:string"
        xdb:SQLName="NAME" xdb:SQLType="VARCHAR2"/>
      <xsd:element name="EMAIL" type="xsd:string"
        xdb:SQLName="EMAIL" xdb:SQLType="VARCHAR2"/>
      <xsd:element name="ADDRESS" type="xsd:string" maxOccurs="unbounded"
        xdb:SQLName="ADDRESS" xdb:SQLCollType="ADDRESS_TYPE"
        xdb:SQLType="VARCHAR2" xdb:maintainOrder="false"/>
      <xsd:element name="PHONE" type="phoneType" xdb:SQLName="PHONE"/>
      <xsd:element name="DESCRIPTION" type="contentType"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="contentType" mixed="true"
    xdb:SQLType="CLOB" xdb:maintainDOM="true">
    <xsd:sequence>
      <xsd:any minOccurs="0" maxOccurs="unbounded" processContents="skip"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="phoneType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\\(\\d{3})\\d{3}-\\d{4}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Looking at the preceding example, the first thing to do when annotating the XML schema is to include the Oracle XML DB namespace declaration, **xmlns:xdb="http://xmlns.oracle.com/xdb"**, in the **<schema>** element. This namespace prefix is then used to qualify all the Oracle XML DB annotations.

Next, **xdb:storeVarrayAsTable="true"** is a global XML DB annotation, which tells the XML DB to store all the VARRAY elements in nested object tables. This annotation helps to speed up the queries on XML elements that are defined with **maxOccurs > 1**. For example, in **customer_simple_ann.xsd**, this annotation affects the storage of the **<ADDRESS>** elements.

In addition, you can specify an XML DB annotation **xdb:mapUnboundedStringToLob="true"** in the **<schema>** element to map unbounded strings to CLOB and unbounded binary data to BLOB with out-of-line table storage. By default, it is set to be **false** so that all unbounded strings defined in the XML schema map to VARCHAR2(4000) and unbounded binary data maps to RAW(2000) with inline table storage. Since inline table storage does not efficiently store large XML documents, you should set **xdb:mapUnboundedStringToLob="true"**.

For all the global complex and simple types, you can define the following XML DB annotations to specify the corresponding SQL names and data types:

- **xdb:SQLType** Specifies the SQL type mapped to the XML schema type definition. You can use this annotation to avoid having the XML DB generate names for the SQL data types.
- **xdb:maintainDOM** Specifies whether the complex type should maintain DOM fidelity. Normally, you should set this to **false**. Otherwise, the XML DB by default will add the SYS_XDBPD\$ attribute (position descriptor) to each created object type to preserve information such as comments, processing instructions, and the sibling element orders in XML, and thus increases the storage overhead. For example, to avoid maintaining the DOM fidelity in the customer records, **xdb:maintainDOM="false"** is set on **CustomerType**.

**NOTE**

xdb:SQLName is not allowed on the *complexType* or *simpleType* definitions. Otherwise, you will get the following error: ORA-30937:
No schema definition for 'SQLName' (namespace 'http://xmlns.oracle.com/xdb') in parent 'complexType'

For the root element of the XML document, you should specify the **xdb:defaultTable** attribute and optionally use **xdb:tableProps** to set the table attributes:

- **xdb:defaultTable** Specifies the name of the table into which XML instances of this schema should be stored. It establishes a link between the XML DB repository and this default table so that any insertion, update, or deletion of the XML documents conforming to this XML schema in the XML DB repository will have the corresponding changes in the default table, and vice versa. In the example, a **customer** table will be created as the default table.
- **xdb:tableProps** Specifies the default table properties in SQL syntax that is appended to the CREATE TABLE clause.

For all XML elements, you should specify the element names and the element type if the type they are based on is not among the global types defined in the XML schema that have already been annotated. The following lists the XML DB annotations for the XML elements:

- **xdb:SQLName** Specifies the name of the SQL object that maps to the XML element.
- **xdb:SQLType** Specifies the name of the SQL type corresponding to the XML element.
- **xdb:SQLInline** Specifies whether Oracle XML DB should generate a new object table and define XMLType REFs to store the XML elements. The default setting is **true**, which specifies not to define REFs. The **true** setting of this annotation affects all the top-level elements declared in the XML schema and the XML element with **maxOccurs > 1**. You need to set this to **false** for out-of-line storage. This will give better performance by avoiding table locks.
- **xdb:SQLCollType** Specifies the name of the SQL collection type corresponding to the XML element that has **maxOccurs > 1**. For example, in the <ADDRESS> element, the **xdb:SQLCollType="ADDRESS_TYPE"** is added. By default, the collection will use a VARRAY. Because **xdb:storeVarrayAsTable="true"** is set, the storage of the VARRAY

168 Oracle Database 10g XML & SQL

is a Ordered Collections in Table (OCTs) instead of LOBs (default). This is useful when you want to create constrains on the element.

Instead of covering all the possible annotations, we listed the most frequently used XML DB annotations. In summary, you should keep the following points in mind when annotating XML schemas.

First, you should specify the name of the default table using **xdb:defaultTable** and a SQL name for each XML element and data type in the XML schema using **xdb:SQLName**, **xdb:SQLCollType**, or **xdb:SQLType**. You should notice that, in the example:

- **xdb:SQLName** defines the SQL names for the XML elements
- **xdb:SQLCollType** defines the SQL names only for the XML elements with **maxOccurs>1**.
- **xdb:SQLType** defines the SQL names for all the complexTypes or the simpleTypes that do not use the default mapping provided by Oracle XML DB.

Specifying the SQL names using XML schema annotations is useful because the system-generated names are not easy to remember. You should also consider specifying all SQL names capitalized to eliminate case-sensitive names in the database, which require using double quotes when referring to the SQL objects. For example, without capitalization, you have to use "Customer260_TAB" versus CUSTOMER260_TAB to refer to the default table storing the customer records.



NOTE

For XML elements and types, if no **xdb:SQLName**, **xdb:SQLType**, or **xdb:SQLCollType** is specified, Oracle XML DB will use the name of the element or data type to create the SQL name. Because XML is case sensitive, the SQL name will be case sensitive, requiring you to use quotes around it for all references. These annotations are also useful if the XML element or type name is long, or has a name conflict in the XML schema.

Next, you should define the storage minimizing any extra data storage, such as avoid preserving DOM fidelity. It is also useful to store sub-trees or complex types as CLOBs by setting the **xdb:SQLTypes="CLOB"** when no XPath-based queries on the content are required. Oracle XML DB will not shred this XML data, thus saving time and resources.

Finally, when working with small but unbounded XML elements you should store the content as VARRAYs by setting the **xdb:storeVarrayAsTable="false"**. For large unbounded XML elements, you can instead use nested tables by specifying the **xdb:storeVarrayAsTables="true"** in the **<schema>** element or even use nested tables by setting **xdb:maintainOrder="false"** on the element for better performance.

XML Data Loading

After you have defined the XMLType storage, you can load data into XMLType tables by using SQL, the protocol APIs, or the SQL*Loader utility.

Using SQL Command

The simplest way to load XML data into XMLType tables is through the INSERT SQL command, such as in the following example:

```
INSERT INTO customer VALUES(XMLType('<Customer>
  <NAME>Steve Joes</NAME>
  <EMAIL>Steve.Joes@example.com</EMAIL>
  <ADDRESS>Someroad, Somecity, Redwood Shores, CA 94065, U.S.A</ADDRESS>
  <PHONE>6505723456</PHONE>
  <DESCRIPTION>Very Important US Customer</DESCRIPTION>
</Customer>').CreateSchemaBasedXML(
  'http://localhost:8080/public/contact_simple_ann.xsd'));
```

Using this approach, you can construct the XMLType instance from XML in VARCHAR2, CLOB, or BFILE and optionally use the XMLType.CreateSchemaBasedXML() function to refer to a registered XML schema.

Without the XMLType.CreateSchemaBasedXML() function, you can insert XML into XML schema-based XMLTypes by including an XML Schema reference in the root element of the XML document using the XML schema location attributes, including the **xsi:schemaLocation** or **xsi:noNamespaceSchemaLocation** attribute:

```
INSERT INTO customer
values(XMLType('<Customer xmlns:xsi="http://www.w3.org/2001/XMLSchema
  -instance" xsi:noNamespaceSchemaLocation="http://localhost:8080/public/
  contact_simple_ann.xsd">
  <NAME>Steve Joes</NAME>
  <EMAIL>Steve.Joes@example.com</EMAIL>
  <ADDRESS>Someroad, Somecity, Redwood Shores, CA 94065, U.S.A</ADDRESS>
  <PHONE>6505723456</PHONE>
  <DESCRIPTION>Very Important US Customer</DESCRIPTION>
</Customer>'));
```

The **xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"** attribute declares the namespace for the XML Schema instance. The **xsi:noNamespaceSchemaLocation="http://localhost:8080/public/contact_simple_ann.xsd"** attribute specifies the registered XML schema URL. In this example, since the XML document doesn't have a namespace, **xsi:noNamespaceSchemaLocation** is used. If the XML document contains a namespace, for example, the XML schema of the XML document defines a target namespace as **targetNamespace="http://www.example.com/customer"**, you need to use the **xsi:schemaLocation** attribute as follows:

```
xsi:schemaLocation= "http://www.example.com/customer http://localhost:8080/
public/contact_simple_ann.xsd"
```

The attribute contains the **targetNamespace**, **http://www.example.com/customer**, and the URL of the XML schema, **http://localhost:8080/public/contact_simple_ann.xsd**.

170 Oracle Database 10g XML & SQL

Using Oracle XML DB Repository Interfaces

The XML DB repository provides protocol interfaces, including FTP and WebDAV/HTTP interfaces, to insert XML and other types of documents. As discussed in Chapter 8, you can create a WebDAV folder and use it to copy or edit XML files in the XML DB repository as if it were another directory on your disk. When using the protocol interfaces, the XML document must have the XML schema location attributes to ensure that the data is inserted into the default tables created during the XML schema registration. The following example utilizes the FTP interface to insert a customer record to the default **customer** table after registering the **contact_simple_ann.xsd** to the XML DB:

```
D:\>ftp
ftp> open localhost 2100
Connected to [Machine_Name] 220 [Machine_Name].FTP Server (Oracle XML
DB/Oracle Database 10g Enterprise Edition Release X.X.X.X.X) ready.
User ([Machine_Name]:(none)): demo
331 pass required for DEMO
Password:
230 DEMO logged in
ftp> cd public
250 CWD Command successful
ftp> put customer1.xml
200 PORT Command successful
150 ASCII Data Connection
226 ASCII Transfer Complete
ftp: 444 bytes sent in 0.00Seconds 444000.00Kbytes/sec.
ftp> ls customer1.xml
200 PORT Command successful
150 ASCII Data Connection
customer1.xml
226 ASCII Transfer Complete
ftp: 15 bytes received in 0.00Seconds 15000.00Kbytes/sec.
ftp>bye
```

After the operations, the new customer record is inserted into both the XML DB repository in **/public** directory and the default **customer** table. In addition to the two records inserted using SQL, there are now three records in the **customer** table:

```
SQL> SELECT count(1) FROM customer;
COUNT(1)
-----
3
```

We will discuss the XML DB repository features in the “Oracle XML DB Repository” section. For now, you just need to know that no matter what directory in the XML DB repository is used to store the XML document, the new customer record will always be inserted into the default XMLType table as long as it refers to the corresponding URL of the registered XML schema.

Using SQL*Loader

SQL*Loader has been the predominant tool for loading data into the Oracle databases. In Oracle Database 10g, SQL*Loader supports loading XML data into XMLType columns or XMLType tables

independent of the underlying storage. In other words, you can use this same method to load XML data to CLOBs or object-relational XMLTypes. Additionally, SQL*Loader allows XML data to be loaded using both conventional and direct path methods. The conventional path is the default mode that uses SQL to load data into Oracle databases. The direct path mode bypasses SQL and streams the data directly into the Oracle database files.

To load XML data using SQL*Loader, you need a control file describing the input data and the target table or table columns. For example, to insert two customer records as in **customer3.xml** and **customer4.xml** into the **customer** table, you can create a control file as shown in the following:

```
LOAD DATA
INFILE *
INTO TABLE customer
APPEND
XMLType (XMLDATA) (
  lobfn FILLER CHAR TERMINATED BY ',',
  XMLDATA LOBFILE(lobfn) TERMINATED BY EOF
)
BEGINDATA
xml/customer3.xml,
xml/customer4.xml
```

The control file tells SQL*Loader to load data (LOAD DATA) by appending (APPEND) the new data contained within the control file (INFILE *) to the **customer** table (INTO TABLE **customer**). XMLType(XMLDATA) refers to new data as XMLType. Since this is an appending operation, it means that SQL*Loader will load the new data without overwriting the old customer records. If you use REPLACE instead, the old customer records will be deleted before new data is inserted.

The **lobfn** operator is a FILLER field. In SQL*Loader, FILLER fields are used to collect the data from the inputs. In other words, the FILLER fields are not mapped to any table columns; instead they are used to skip or select data from the input data. In this example, **lobfn** is used to get the names of the XML documents after BEGIN DATA and the names are delimited by comas (TERMINATED BY ','). The actual XML data in the files are delimited by the end-of-file (EOF).

After the control file is created, you can set the **\$ORACLE_HOME\bin** directory in your PATH environment and run the following command to invoke the SQL*Loader command-line utility **sqlldr**:

```
D:\>sqlldr userid=demo/demo control=customerLoad.ctl
SQL*Loader: Release X on Thu Jun 26 22:26:53 2003
(c) Copyright 2001 Oracle Corporation. All rights reserved.
Commit point reached - logical record count 2
```

The **userid** specifies the username and password for the database user who owns the **customer** table. The **control** option specifies the filename for the control file. The result shows that two logical records are recognized by SQL*Loader. The further logging information for **sqlldr** can be found in the **<control_file_name>.log** file. You can specify **direct=y** if you want to use the direct path mode to load the XML data. Compared to the conventional path mode, the direct path mode is faster because it bypasses the SQL layer and streams XML data into the Oracle database files without invoking any triggers or constraint checking.

172 Oracle Database 10g XML & SQL

XML Schema Validation

During XML loading or after content updates on the XML Schema-based XMLTypes, Oracle XML DB simply checks to see if the XML document is well-formed augmented with object checks instead of performing a full XML Schema validation. In other words, Oracle XML DB performs only limited checks to make sure the XML document conforms to the object-relational storage. For example, the XML DB will check whether the <PHONE> element exists before inserting the customer records. It will not stop the data insertion when the phone numbers violate the string pattern defined in the XML schema.

To void invalid data that could be inserted into XMLTypes, you need to explicitly call for XML Schema validation. The simplest way to do this is to set up a TRIGGER before the INSERT actions as follows:

```
CREATE OR REPLACE TRIGGER customer_insert
AFTER INSERT ON customer
FOR EACH ROW
DECLARE
doc XMLType;
BEGIN
doc := :new.SYS_NC_ROWINFO$.
XMLType.schemaValidate(doc);
END;
```

After the trigger is created, full validation is performed when you insert sample data into the **customer** table:

```
INSERT INTO customer VALUES(
XMLType('<CUSTOMER xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://localhost:8080/public/
contact_simple_ann.xsd">
<NAME>Steve Joes</NAME>
<EMAIL>Steve.Joes@example.com</EMAIL>
<ADDRESS>Someroad, Somecity, Redwood Shores, CA 94065, U.S.A</ADDRESS>
<PHONE>6505723456</PHONE>
<DESCRIPTION>Very Important US Customer</DESCRIPTION>
</CUSTOMER>'));
```

Thus this example returns the following errors:

```
INSERT INTO customer
*
ERROR at line 1:
ORA-31154: invalid XML document
ORA-19202: Error occurred in XML processing
LSX-00333: literal "6505723456" is not valid with respect to the pattern
ORA-06512: at "SYS.XMLTYPE", line 333
ORA-06512: at "DEMO.CUSTOMER_INSERT", line 5
ORA-04088: error during execution of trigger 'DEMO.CUSTOMER_INSERT'
```

As you see, the error message states that the phone number does not follow the string pattern defined in the XML schema. After you have updated the phone number, you can try again:

```
SQL> INSERT INTO customer VALUES (
XMLType('<Customer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="http://localhost:8080/public/
      contact_simple_ann.xsd">
<NAME>Steve Joes</NAME>
<EMAIL>Steve.Joes@example.com</EMAIL>
<ADDRESS>Someroad, Somecity, Redwood Shores, CA 94065, U.S.A
</ADDRESS>
<PHONE>(650)572-3456</PHONE>
<DESCRIPTION>Very Important US Customer</DESCRIPTION>
</CUSTOMER>'));
```

The new valid customer record is inserted. You can check the XML Schema validation status of an XMLType object using either the **XMLType.isSchemaValid()** function or the **XMLType.isSchemaValidated()** function:

```
SQL> SELECT x.isSchemaValid() FROM customer x;
X.ISSCHEMAVALID()
-----
                1
                0
                ...0
```

The preceding result shows that, so far, there is only one record in the table and it is valid against the XML schema. The records inserted previously do not have a valid status. This is because the **XMLType.schemaValidate()** function validates the XMLType object and updates the validation status of XMLType objects in the XML DB.

NOTE

Turning on full validation will have a significant negative effect on INSERT performance, thus should be used only if necessary. It is usually better to do validation checks at the time of document creation or in the middle tier.

Oracle XML DB Repository

Oracle XML DB Repository can function as a file system in the Oracle database. Any data in the Oracle XML DB Repository maps to a resource which has a pathname (or a URL) and is stored either in a BLOB or an XMLType object. The XML DB repository provides extensive management facilities for these content resources.

You have learned how to load XML through the protocol interfaces of the XML DB repository. In this section, we will discuss other topics, such as applying version control to documents and creating links, and managing resources. We will also discuss the major PL/SQL packages that support this functionality:

- **DBMS_XDB** provides functions for resource and session management of the XML DB repository. It also provides functionality to rebuild the hierarchical indexes.
- **DBMS_XDB_VERSION** provides functions for the version control of the resources.

174 Oracle Database 10g XML & SQL

Resource Management

In Oracle Database 10g, you can use the `DBMS_XDB` package to create and delete resources, folders, and links for the resources. You can also use this package to lock/unlock resources when reading or updating XML data:

```
DECLARE
  res BOOLEAN;
  xr REF XMLType;
  x XMLType;
BEGIN
  FOR po_rec IN (SELECT rownum id, ref(p) xref FROM customer p
                ORDER BY rowid)
  LOOP
    res:=DBMS_XDB.createResource('/public/customer'||po_rec.id||
    '.xml', po_rec.xref);
  END LOOP;
END;
```

In this example, all of the customer records are read out of the `customer` table, and XML resource documents are created in the `/public` directory of the XML DB repository using the `DBMS_XDB.createResource ()` function. You can additionally create a `/public/important_customer` folder in the XML DB repository as follows:

```
DECLARE
  retb BOOLEAN;
BEGIN
  retb := DBMS_XDB.createFolder('/public/important_customer');
  COMMIT;
END;
/
```

Then, you can create a resource such as `README.txt` to explain the content in this folder:

```
DECLARE
  res BOOLEAN;
BEGIN
  res :=
  DBMS_XDB.createResource('/public/important_customer/README.txt',
    'This folder lists all of the US customer who are important to
    our business');
  COMMIT;
END;
/
```

Since you already have a set of customers listed in the `/public` directory, you can create a set of links instead of creating a second copy of the data:

```
EXEC DBMS_XDB.link('/public/customer1.xml',  
                  '/public/important_customer/', 'SteveJones.xml');
```

If you want to delete a resource, you can use the **DBMS_XDB.DeleteResource()** function, as follows:

```
DBMS_XDB.DeleteResource('/public/important_customer/SteveJones.xml');  
DBMS_XDB.DeleteResource ('/public/customer1.xml');
```

You can delete a resource with resources linking to it. However, after the original resource is removed, all the linked resources are no longer references. Each of them instead will hold a copy of the data.

Version Control

The **DBMS_XDB_VERSION** and **DBMS_XDB** PL/SQL packages implement Oracle XML DB versioning functions, which provide a way to create and manage different versions of a *Version-Controlled Resource (VCR)* in Oracle XML DB.

When an XML DB resource is turned into a VCR, a flag is set to mark it as a VCR and the current resource becomes the initial version. This version is not physically stored in the database. In other words, there is no extra copy of this resource stored when it is versioned. Subsequent versions are stored in the same tables. Since the version resource is a system-generated resource, it does not have a pathname. But you can still access the resource via the functions provided in the **DBMS_XDB_VERSION** package.

When the resource is checked out, no other user can make updates to it. When the resource is updated the first time, a copy of the resource is created. You can make several changes to the resource without checking it back in. You will always get the latest copy of the resource, even if you are a different user. When a resource is checked back in, the original version that was checked out is placed into historical version storage.

The versioning properties for the VCR are maintained within the XML DB repository. In this release, versioning *only* works for non-schema-based resources. Thus XMLTypes based upon shredded XML documents or XMLType CLOBs that have schemas attached are not officially supported to use VCRs. However, we have found that as long as you do not create unique meta-data associated with a particular version, such as an index, VCRs will work.

NOTE

You cannot switch a VCR back to a non-VCR.

Oracle XML DB provides functions to keep track of all changes on Oracle XML DB VCRs. The following code demonstrates these functions:

```
DECLARE  
  resid DBMS_XDB_VERSION.RESID_TYPE;  
BEGIN  
  resid := DBMS_XDB_VERSION.MakeVersioned('/public/important_customer/  
    SteveJones.xml');  
END;  
/
```

176 Oracle Database 10g XML & SQL

You can get the resource ID of the VCR as follows:

```
SET AUTOPRINT ON
VAR OUT CLOB
DECLARE
  resid DBMS_XDB_VERSION.RESID_TYPE;
  res XMLType;
BEGIN
  resid := DBMS_XDB_VERSION.MakeVersioned('/public/important_customer/SteveJones.xml');
  -- Obtain the resource
  res := DBMS_XDB_VERSION.GetResourceByResId(resid);
  SELECT res.getClobVal() INTO :OUT FROM dual;
END;
```

To update a VCR, you need to first check out the resource, make file updates, and then check them back in to the XML DB repository, as follows:

```
DECLARE
  resid DBMS_XDB_VERSION.RESID_TYPE;
BEGIN
  DBMS_XDB_VERSION.CheckOut('/public/important_customer/SteveJones.xml');
  resid :=
    DBMS_XDB_VERSION.CheckIn('/public/important_customer/SteveJones.xml');
END;
```

Note that the resource is not updated until the new file is checked in. If you want to cancel the updates after the checkout, you can “uncheck out” the resource as follows:

```
DECLARE
  resid DBMS_XDB_VERSION.RESID_TYPE;
BEGIN
  resid :=
    DBMS_XDB_VERSION.UncheckOut('/public/important_customer/SteveJones.xml ');
END;
```

Storing XML Documents in Relational Tables

Relational tables are normally designed without considering XML storage. However, in many cases these tables can be used to store *shredded* XML documents and produce a useful XML representation by creating XMLType views using Oracle Database 10g XML features or generating XML using the Oracle XDK.

Storing XML data in relational tables is useful if your application needs to avoid the limitations of the XMLType storage, such as limited XML schema evolution and data replication. Relational storage is also widely used by applications that require fine-grained access to the data in XML documents while not needing to preserve the complete hierarchical structure of XML.

Oracle Database 10g provides extensive support for loading, exporting, and processing XML data into relational tables. To load XML data, you can use the *XML SQL Utility (XSU)*. XSU provides both Java and PL/SQL program interfaces and command-line utilities. Built on XSU, the *TransX*

Utility (XML Translation) further simplifies the character set conversion during data loading, and the *XSQL Servlet* provides the HTTP interfaces. If the functionality provided by these utilities is not sufficient for your application, you can always use their programmatic APIs in conjunction with other XDK libraries to build your own custom solution.

XML SQL Utility

XSU provides Java-based APIs, command-line utilities, and PL/SQL packages that support loading XML data into relational tables including tables with XMLType columns. We will discuss how you can use its functionality in the following sections.

Canonical Mapping

The first thing you need to understand before using XSU is the canonical mapping used by XSU to map XML to relational tables and render the results of SQL queries in XML. In this canonical mapping, the **<ROWSET>** element is the root element of the XML document, and its child **<ROW>** elements map to rows of data in tables. The names of the child elements for each **<ROW>** element map to the table column names or object names from which the results are returned. The **num** attributes of the **<ROW>** elements are numbers that provide the order information. The following is an XML Schema representation of this metadata structure:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="ROWSET">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ROW">
          <xs:complexType>
            <xs:sequence>
              <xs:any/>
            </xs:sequence>
            <xs:attribute name="num" type="xs:string" use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XSU provides ways to change the names of the **<ROWSET>** and **<ROW>** elements. For example, a CUSTOMER_TBL table is defined as follows:

```
CREATE TABLE CUSTOMER_TBL (
  NAME VARCHAR2(100),
  ADDRESS VARCHAR2(200),
  EMAIL VARCHAR2(200),
  PHONE VARCHAR2(50),
  DESCRIPTION VARCHAR2(4000));
```

178 Oracle Database 10g XML & SQL

The XML document mapping to this table with the canonical mapping is shown as follows:

```
<ROWSET>
  <ROW>
    <NAME>Steve Jones</NAME>
    <EMAIL>Steve.Jones@example.com</EMAIL>
    <ADDRESS>Someroad, Somecity, Redwood Shores, CA 94065, U.S.A</ADDRESS>
    <PHONE>6505723456</PHONE>
    <DESCRIPTION>Very Important US Customer</DESCRIPTION>
  </ROW>
</ROWSET>
```

NOTE

By default, all table, column, and object names are uppercase; therefore, if you want to have mixed-case XML documents successfully inserted you need to specify the **ignoreCase** options when using XSU.

In order to run the XSU command-line utility, you need to set the following Java packages in your Java CLASSPATH:

- **xmlparserv2.jar** Oracle XML Parser for Java
- **classes12.jar** Oracle JDBC Drivers
- **xsu12.jar** Oracle XML SQL Utility

NOTE

You may need to add the **orai18n.jar** package to your Java CLASSPATH when handling XML with different character sets. Or, you might get an `oracle.xml.sql.OracleXMLSQLException`:
`'java.sql.SQLException: Non supported character set...'`

XSU depends on the XML parser to build a DOM and depends on the JDBC drivers to connect to the Oracle database and retrieve the table metadata. After the Java CLASSPATH is properly set, you can run the XSU command-line utility with `java OracleXML`, which has two options, **getXML** for querying the database and **putXML** for inserting data into the database.

NOTE

Update and delete operations are not included in the XSU command-line utility but they are supported by XSU through the Java and PL/SQL APIs.

For example, to insert the XML data in **contact01.xml**, you can run the following command:

```
java OracleXML putXML -conn "jdbc:oracle:thin:@localhost:1521:orclX"
-user "demo/demo" -fileName "customer1_xsu.xml" "customr_tbl"
```

Chapter 9: Storing XML Data 179

The data is inserted into the CUSTOMER_TBL table in the **demo** schema. To query the content in the table and return the results in XML, you can run the following XSU command:

```
java OracleXML getXML -conn "jdbc:oracle:thin:@localhost:1521:orclX"
-user "demo/demo" "SELECT * FROM customer_tbl"
```

The following XML document is returned:

```
<?xml version = '1.0'?>
<ROWSET>
  <ROW num="1">
    <NAME>Steve Jones</NAME>
    <ADDRESS>Someroad, Somecity, Redwood Shores, CA 94065, U.S.A</ADDRESS>
    <EMAIL>Steve.Jones@example.com</EMAIL>
    <PHONE>6505723456</PHONE>
    <DESCRIPTION>Very Important US Customer</DESCRIPTION>
  </ROW>
</ROWSET>
```

At this point, the XML document's data has been successfully loaded into the database. However, input XML documents are not always in the canonical format. How can you deal with these XML documents? The usual approach is to use an XSLT stylesheet to transform the XML document into the canonical format. On the other hand, you can create database object views mapping to the incoming XML format.

Object Views

If an XML document is not in the canonical format, you can create object views or XMLType views, to allow XSU to map XML documents to database tables. In the following **contact.xml** XML document, the contact information is stored as follows:

```
<Contact_List>
  <Contact>
    <User_id>userid</User_id>
    <First_Name>Steve</First_Name>
    <Last_Name>Jones</Last_Name>
    <Business>
      <Email>Steve.Jones@oracle.com</Email>
      <Phone>(650)5769801</Phone>
      <Address>
        <Street1>4op11</Street1>
        <Street2>500 Oracle Parkway</Street2>
        <City>Redwood Shores</City>
        <State>CA</State>
        <Zipcode>94065</Zipcode>
        <Country>USA</Country>
      </Address>
    </Business>
  </Contact>
</Contact_List>
```

180 Oracle Database 10g XML & SQL

The database schema is defined as follows:

```
CREATE TYPE address_typ AS OBJECT(  
    street1 VARCHAR2(200),  
    street2 VARCHAR2(200),  
    city VARCHAR2(100),  
    state VARCHAR2(20),  
    zipcode VARCHAR2(20),  
    country VARCHAR2(20));  
/  
CREATE TABLE contact_tbl(  
    contactid VARCHAR2(15) PRIMARY KEY,  
    firstname VARCHAR2(100),  
    lastname VARCHAR2(200),  
    midname VARCHAR2(50),  
    business_phone VARCHAR2(20),  
    home_phone VARCHAR2(10),  
    cell_phone VARCHAR2(20),  
    business_addr address_typ,  
    business_email VARCHAR2(150));
```

Using the canonical mapping, the XML document cannot directly map to the table columns as the document contains multiple levels; thus, to insert this XML document, you need to create the following object view:

```
CREATE TYPE contactinfo_type AS OBJECT(  
    phone VARCHAR2(20),  
    email VARCHAR2(150),  
    address address_typ);  
/  
-- Create Object View  
CREATE VIEW contact_view AS  
    SELECT contactid AS user_id, firstname AS first_name, lastname AS  
        last_name, midname AS mid_name,  
        contactinfo_type(business_phone, business_email,  
            business_addr) AS business  
    FROM contact_tbl;
```

Then, you can run a similar command to load the XML file into CUSTOMER_VIEW:

```
java OracleXML putXML -conn "jdbc:oracle:thin:@localhost:1521:orclx"  
    -user "demo/demo" -fileName "contacts.xml" "contact_view"
```

In this example, the **contact_view** is used by the Oracle database to map the XML data into underlying tables. However, in many cases these types of views are not updateable, when they include multiple table joins or object type inheritance. You then must create an INSTEAD-OF TRIGGER on the views to handle data population for these tables or objects.

Dividing XML Documents into Fragments

When storing XML documents, you sometimes do not want to map every XML element to relational table columns. Instead, you might want to store some of the XML fragments in XML into CLOBs

Chapter 9: Storing XML Data 181

or XMLTypes. The following example illustrates an approach using XSLT to create such XML fragments and insert those XML fragments into one XMLType table column using XSU. In the example, the input XML document is shown as follows:

```
<Contact_List>
  <Contact>
    <User_id>jwang</User_id>
    <First_Name>Jinyu</First_Name>
    <Last_Name>Wang</Last_Name>
    <Title>Senior Product Manager</Title>
    <Description>Jinyu manages the <PRODUCT>Oracle XML Developer's
      Kit</PRODUCT> product.</Description>
  </Contact>
</Contact_List>
```

The **<Description>** element contains mixed content, which we do not want to map to multiple table columns. A **contact_tbl** table is defined as follows:

```
CREATE TABLE contact_tbl (
  contactid VARCHAR2(15) PRIMARY KEY,
  firstname VARCHAR2(100),
  lastname VARCHAR2(200),
  midname VARCHAR2(50),
  description CLOB);
```

To map the **<Description>** element to the **description** column using XSU, you need to apply the following **setCDATA.xml** XSL stylesheet:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output cdata-section-elements="CODE"/>
  <!-- Identity transformation -->
  <xsl:template match="*|@*|comment()|processing-instruction()"
    text(">
  <xsl:copy>
    <xsl:apply-templates select="*|@*|comment()|processing-
      instruction()|text()"/>
  </xsl:copy>
</xsl:template>
<xsl:template match="Description">
  <xsl:element name="Description">
    <xsl:copy-of select="@*|comment()|processing-instruction()"/>
    <xsl:text disable-output-escaping="yes">&lt;![CDATA[</xsl:text>
    <xsl:apply-templates select="*|./@*|./*/comment()|./*/
      processing-instruction()|text()"/>
    <xsl:text disable-output-escaping="yes">]]&gt;</xsl:text>
  </xsl:element>
</xsl:template>
</xsl:stylesheet>
```

182 Oracle Database 10g XML & SQL

This XSLT transformation will transform the input XML document and include all the child elements of **<Description>** into one CDATA section so that each CDATA section can be stored to the **description** column by XSU.

You can modify the XSL stylesheet for your application by specifying different **match** attributes for the following template:

```
<xsl:template match="Description">... </ xsl:template>
```

The XPath in the **match** attribute specifies the root element of the XML fragment to be stored.

NOTE

Because XSLT requires a DOM to be built in memory, you may need to split the large documents before the transformation.

TransX Utility

When populating the Oracle database with multilingual data or data translations, or when encoding, validation is needed for each XML file. The traditional way is to switch the NLS_LANG setting as you switch loading files with different encoding information. The NLS_LANG setting has to reflect the character set of the file loaded into the database. This approach is error-prone because the encoding information is maintained separately from the data itself. Setting the NLS_LANG environment variable is also tedious work.

With the TransX Utility provided in the XDK, the encoding information is kept along with the data in an XML document of a predefined format so that multilingual data can be transferred without having to switch NLS_LANG settings. TransX Utility maintains the correct character set throughout the process of translating the data and successfully loading it into the database. We will not discuss the details of how to use the TransX Utility. However, we include some samples with the chapter source code for you to try its functionality.

DBMS_XMLSTORE

DBMS_XMLSTORE is a supplied PL/SQL package that supports inserting XML data into database tables. This C-based implementation provides better performance and system manageability than the Java-based DBMS_XMLSave package. This package eliminates the overhead of starting up Oracle JVM as well as translating Java class names for each method call. In addition, DBMS_XMLSTORE is built based on SAX parsing instead of DOM parsing. Therefore, it scales well for large XML documents. You can see this in the following comparison using the SH sample schema:

```
SQL> SELECT count(1) FROM sales;
COUNT(1)
-----
1136945
SQL> CREATE TABLE test AS SELECT * FROM sales;
Table created.
SQL> CREATE TABLE result AS SELECT * FROM sales WHERE 0=1;
Table created.
SQL> SELECT count(1) FROM test;
```

Chapter 9: Storing XML Data 183

```

COUNT(1)
-----
      1136945
SQL> SELECT count(1) FROM result;
COUNT(1)
-----
          0
SQL> SET timing ON
SQL> DECLARE
  2   qryCtx DBMS_XMLQuery.ctxHandle;
  3   v_clob CLOB;
  4   savCtx DBMS_XMLSave.ctxType;
  5   v_rows NUMBER;
  6 BEGIN
  7   -- Query out the content
  8   qryCtx := DBMS_XMLQuery.newContext('SELECT * FROM test');
  9   v_clob := DBMS_XMLQuery.getXml(qryCtx);
 10   DBMS_OUTPUT.PUT_LINE('CLOB size = ' || DBMS_LOB.GETLENGTH(v_clob));
 11   -- Save the content
 12   savCtx := DBMS_XMLSave.newContext('RESULT');
 13   v_rows := DBMS_XMLSave.insertxml(savCtx,v_clob);
 14   DBMS_XMLSave.closeContext(savCtx);
 15   DBMS_OUTPUT.PUT_LINE(v_rows || ' rows inserted...');
 16 END;
 17 /
DECLARE
*
ERROR at line 1:
ORA-29532: Java call terminated by uncaught Java exception:
java.lang.OutOfMemoryError
ORA-06512: at "SYS.DBMS_XMLSAVE", line 114
ORA-06512: at line 13
Elapsed: 00:11:57.05

```

In the preceding example, the **sales** table in the SH sample schema described in Chapter 8 is used to generate a large XML document that, when parsed, is too large for the configured Oracle JVM memory. You can increase the JAVA_POOL_SIZE to give more memory for processing; however, this may not be sufficient, especially when this memory takes away from the database memory pool. In Oracle Database 10g, you can use DBMS_XMLSTORE to resolve this issue as follows:

```

DECLARE
  v_clob CLOB;
  savCtx DBMS_XMLSTORE.ctxType;
  v_rows NUMBER;
BEGIN
  -- Query out the content
  SELECT doc INTO v_clob FROM temp_clob;
  -- Save the content
  savCtx := DBMS_XMLSTORE.newContext('RESULT');
  -- Set the update columns to improve performance

```

184 Oracle Database 10g XML & SQL

```
DBMS_XMLSTORE.SetUpdateColumn (savCtx, 'PROD_ID');
DBMS_XMLSTORE.SetUpdateColumn (savCtx, 'CUST_ID');
DBMS_XMLSTORE.SetUpdateColumn (savCtx, 'TIME_ID');
DBMS_XMLSTORE.SetUpdateColumn (savCtx, 'CHANNEL_ID');
DBMS_XMLSTORE.SetUpdateColumn (savCtx, 'PROMO_ID');
DBMS_XMLSTORE.SetUpdateColumn (savCtx, 'QUANTITY_SOLD');
DBMS_XMLSTORE.SetUpdateColumn (savCtx, 'AMOUNT_SOLD');
-- Insert the document
v_rows := DBMS_XMLSTORE.insertxml(savCtx,v_clob);
DBMS_XMLSTORE.closeContext (savCtx);
DBMS_OUTPUT.PUT_LINE(v_rows || ' rows inserted...');
END;
```

It is recommended to use the DBMS_XMLSTORE **SetUpdateColumn()** function where applicable, as shown in the preceding example, because this allows the DBMS_XMLSTORE program to know the list of columns that need to be updated so as to use explicit SQL binding to the XML data. The previous example uses the following SQL statement when preparing the data insertion:

```
INSERT INTO sales(prod_id, cust_id, ..., amount_sold) values
(:1, :2, ..., :6);
```

This speeds up the data-insertion process by eliminating the overhead of parsing of the SQL statements in the database.

Using External Tables

Introduced in Oracle9i, Oracle's external table feature offers a solution to define a table in the database while leaving the data stored outside of the database. Prior to Oracle Database 10g, external tables can be used only as read-only tables. In other words, if you create an external table for XML files, these files can be queried and the table can be joined with other tables. However, no DML operations, such as INSERT, UPDATE, and DELETE, are allowed on the external tables.

NOTE

In Oracle Database 10g, by using the ORACLE_DATAPUMP driver instead of the default ORACLE_DRIVER, you can write to external tables.

In Oracle Database 10g, you can define VARCHAR2 and CLOB columns in external tables to store XML documents. The following example shows how you can create an external table with a CLOB column to store the XML documents. First, you need to create a DIRECTORY to read the data files:

```
CREATE DIRECTORY data_file_dir AS 'D:\xmlbook\Examples\Chapter9\src\xml';
GRANT READ, WRITE ON DIRECTORY data_file_dir TO demo;
```

Then, you can use this DIRECTORY to define an external table:

```
CREATE TABLE customer_xt (doc CLOB)
ORGANIZATION EXTERNAL
(
```

```

TYPE ORACLE_LOADER
DEFAULT DIRECTORY data_file_dir
ACCESS PARAMETERS
(
  FIELDS (lobfn CHAR TERMINATED BY ',')
  COLUMN TRANSFORMS (doc FROM lobfile (lobfn))
)
LOCATION ('xml.dat')
)
REJECT LIMIT UNLIMITED;

```

The **xml.dat** file follows:

```

customer1.xml
customer2.xml

```

If you describe the table, you can see the following definition:

```

SQL> DESC customer_xt;

```

Name	Null?	Type
DOC		CLOB

Then, you can query the XML document as follows:

```

SELECT XMLType(doc).extract('/Customer/EMAIL')
FROM customer_xt;

```

Though the query requires run-time XMLType creation and XPath evaluation, this approach is useful when applications just need a few queries on the XML data and don't want to upload the XML data into database. In Oracle Database 10g, you cannot create external tables that contain pre-defined XMLType column types.

Schema Evolution

XML schemas evolve when there are new requirements for the XML data. Your ability to reflect these changes in the database is highly dependent on the storage.

If you use relational tables, you can change the table structure and update the XML views to reflect the new mapping from XML to relational tables. If you use CLOB XMLTypes, your new XML data can be directly inserted because this storage allows you to store XML conforming to different XML schemas. However, for XML Schema-based XMLTypes, evolution of their XML schemas is an expensive process because it requires updating of the object-relational structure of the XMLTypes. In Oracle Database 10g, this type of evolution is limited to either performing export/import of the data or using the **CopyEvolution()** function in the DBMS_XMLSCHEMA package.

186 Oracle Database 10g XML & SQL

Best Practices

If you need to accept XML data and store it in the database, the first thing that you should consider is whether your application requires preserving the XML structure in the database. As we discussed in Chapter 8, you need to evaluate the pros and cons of the XML storage options and analyze how the storage affects the retrieving and updating of the XML data. Additionally, you sometimes need to choose a particular XML storage model in order to support receiving XML in the presence of evolving XML schemas.

After selecting the right XML storage model for your application, the following sections provide some guidelines of what you need to know when storing XML in Oracle Database 10g.

Handling Document Type Definitions

Although DTDs are not used to define the storage structure for XMLTypes, Oracle XML DB resolves all the DTD definitions and entities defined or referenced in the inserted XML document. This is performed during the inserting process of XMLType when all the incoming XML documents are parsed. In this process, all the entities, including external or internal entities defined in DTDs, are resolved. This means that all the entities are replaced with their actual values and hence the original entity references are lost.

If you would like to preserve these entity references, you have to store the XML in CLOBs, instead of CLOB XMLTypes. You then can create temporary XMLTypes from these CLOBs whenever you need to resolve all the entities and use the XML content.

Creating XML Schema–based XMLTypes

You can create XML Schema–based XMLTypes using the XMLType construction functions or the **XMLType.CreateXML()** function. However, when you are using these functions to create XML Schema–based XMLTypes, the XML documents have to contain the XML **SchemaLocation** attributes. Sometimes XML documents do not contain such attributes. How can you create an XML Schema–based XMLType without changing the original XML document?

As you have seen in Chapter 8, you can use the **XMLType.CreateSchemaBasedXML** function and specify the URL of the XML schema as follows:

```
INSERT INTO product(id, name, description)
VALUES('xdk', 'XML Developer's Kit',
XMLTYPE(' <DESCRIPTION><KEYWORD>xdk</KEYWORD> is a set of
standards-based utilities that helps to build
<KEYWORD>XML</KEYWORD> applications. It contains XDK Java
Components, XDK C Components and XDK C++ Components.
</DESCRIPTION>').CreateSchemaBasedXML('http://xmlns.oracle.com/
xml/content.xsd'));
```

The URL <http://xmlns.oracle.com/xml/content.xsd> is the registered URL of the XML schema, and it will be used to store the product DESCRIPTION.

Specifying Namespaces

If a stored XML document has namespaces, all of the XML queries on the document have to be namespace-qualified because the **<Namespace:Element>** is not the same as **<Element>** in XML. Both the **XMLType.existsNode()** and the **XMLType.extract()** functions allow the user to specify the namespace in the second parameter as follows:

```
MEMBER FUNCTION existsNode(xpath in varchar2, nsmap in varchar2)
    RETURN number deterministic
MEMBER FUNCTION extract(xpath IN varchar2, nsmap IN varchar2)
    RETURN XMLType deterministic
```

In this case, the XPath needs to use fully qualified XML names, which contain the element name and its namespace. For example, you can insert an XML document with two namespace declarations into XMLTypes as follows:

```
CREATE TABLE temp (doc XMLType);
DECLARE
    v_temp XMLType;
BEGIN
    v_temp:= XMLType.createXML('<foo xmlns="http://www.example.com"
        xmlns:xsd="http://www.w3c.org/2001/XMLSchema">
        <foo_type xsd:type="date">03-11-1998</foo_type>
    </foo>');
    INSERT INTO temp VALUES(v_temp);
END;
```

To query the document, you can define the namespace and its prefix in the second parameter of the **XMLType.extract()** function and qualify the XPath using the prefix, as shown in the following SQL query:

```
SELECT a.doc.extract('/a:foo/a:foo_type',
                    'xmlns:a="http://www.example.com"')
FROM temp a;
```

The result is

```
<foo_type xmlns="http://www.example.com" xmlns:xsd="http://www.w3c.org/2001/
XMLSchema" xsd:type="date">03-11-1998</foo_type>
```

NOTE

If you do not use the namespace-qualified name in the XPath after providing namespaces, you will get an ORA-31013: Invalid XPath expression error.

If you have multiple namespaces, you can list them in the second parameter of the **XMLType.existsNode()** and the **XMLType.extract()** function and separate them with white spaces, as shown in the following example:

```
SELECT a.doc.extract('/a:foo/a:lastupdate/@b:type',
                    'xmlns:a="http://www.example.com"
                    xmlns:b="http://www.w3c.org/2001/XMLSchema"') AS result
FROM temp a;
RESULT
-----
date
```

188 Oracle Database 10g XML & SQL

Summary

The chapter discusses various XML storage options and the associated data loading strategies in Oracle Database 10g. Table 9-1 shows the relationships between the XML storage and the functionality offered in the XML data-loading utilities. You can choose one of these utilities or use the SQL and PL/SQL interfaces to load XML documents into the Oracle database.

Utilities	Functionality	Relational Storage with XMLType Views	XMLType Tables	XMLType Columns
SQL*Loader	Command-line utility	Limited support	Yes	Yes
XML SQL Utility	Command-line utility and programmatic interfaces in Java and PL/SQL	Yes	Yes	Yes
TransX Utility	Command-line utility and programmatic interfaces in Java	Yes	Yes	Yes
XSQL Servlet	Command-line utility, program interfaces in Java and the HTTP interfaces provided in the built-in action handlers	Yes	Yes	Yes
HTTP/WebDAV	HTTP/WebDAV folders	No	Yes, but the table has to be the default table created during the XML schema registration.	No
FTP Interfaces	FTP interfaces	No	Yes, but the table has to be the default table created during the XML schema registration.	No

TABLE 9-1. XML Data-Storage and Data-Loading Utilities