Mit der Version 8i wurden die ersten Strukturen und Werkzeuge für den Umgang mit XML-Daten in der Datenbank eingeführt. Seitdem wuchs die Funktionalität zur Verwaltung von XML stetig. So verfügt die Anwendungsentwicklung heute mit XML-DB über eine Reihe mächtiger Werkzeuge, um XML in der Datenbank zu erzeugen, zu manipulieren und auszuwerten. Etwas weniger stringent ist die Unterstützung, wenn es darum geht, relational vorliegende Daten in valide und wohlgeformte XML-Strukturen zu überführen und diese als Datei auszugeben.

XML aus relationalen Datenbank-Strukturen generieren

Rolf Wesp, Trivadis AG

Im Rahmen eines Kundenprojekts war eine CSV-Massendaten-Schnittstelle durch eine XML-Lösung zu ersetzen, um den geänderten Anforderungen des Datenabnehmers weiter gerecht werden zu können. Die Aufgabe lässt sich in zwei Teile zerlegen, wenn man einmal von der theoretisch gegebenen, aber wenig professionellen Möglichkeit absieht, auf die relationalen Daten mit klassischen PL/SQL-Mitteln zuzugreifen, die XML-Formatierung zu bewerkstelligen, die so erzeugten Sätze in eine Datendatei zu schreiben und selbst dafür zu sorgen, dass diese valide und wohlgeformt ist:

1. Erzeugung eines validen und wohlgeformten XML in der Datenbank, das den Vorgaben des XSD entspricht
2. Ausgabe dieser Struktur auf dem Dateisystem

Auf der Suche nach Funktionen der Datenbank, die diese Aufgabe unterstützen, wird man in verschiedenen Bereichen fündig. Da gibt es zunächst einmal die SQL-Syntax-Erweiterungen der SQL/XML-Funktionen, die vom relational geprägten Entwickler relativ wenig Umdenken erfordern. Sodann lassen sich die objektrelationalen Funktionen der Datenbank nutzen, um zum gewünschten Ergebnis zu kommen. Zudem ist mit XMLTYPE-Views und „SYS_XMLGEN“ sowie dem etwas älteren Ansatz „XSU“ die Reihe noch nicht abgeschlossen. Der Artikel stellt die beiden erstgenannten Methoden im Überblick dar und bewertet sie mit Blick auf die Anforderungen im Projekt.

SQL/XML-Funktionen

SQL/XML-Funktionen stellen eine Erweiterung des SQL-Sprachumfangs dar und werden in das normale SQL eingebettet. Die Ergebnismenge ist vom Datentyp „XMLTYPE“ und stellt bereits wohlgeformtes XML dar. Die wichtigsten Funktionen aus dieser Erweiterung sind:

- XMLELEMENT erzeugt ein benanntes Element, zum Beispiel <Artikel>1000</Artikel >:

```
select xmlelement („Artikel“,
artnr)
from artikelstamm
where …
```

- XMLATTRIBUTES ist ein optionales Argument für XMLELEMENT und erzeugt Attribute für das Element, zum Beispiel <Artikel ArtNr=”1000” Bezeichnung=” Radiergummi “></Artikel>:

```
select  xmlelement („Artikel“
            ,xm-
lattributes (artnr
as „ArtNr“

,get_text(text_id) as  „Be-
zeichnung“))
from artikelstamm
where …
```

- XMLFOREST ist ebenfalls ein optionales Argument für XMLELEMENT und erzeugt eine Liste von benannten (Sub-)Elementen zum Element, zum Beispiel <Artikel><ArtNr> 1000</ArtNr><Bezeichnung> Radiergummi </Bezeichnung></Artikel>:

```
select  xmlelement („Artikel“
          ,xmlforest (artnr
as „ArtNr“

,get_text(text_id) as  „Be-
zeichnung“))
from artikelstamm
where …
```

- XMLAGG aggregiert Elemente als Liste unter einem Element und erzeugt so eine Element-Hierarchie, zum Beispiel <ArtListe><Artikel>1000</Artikel><Artikel>1001</Artikel><Artikel>… </ArtListe>:

```
select  xmlelement(„ArtListe“

,xmlagg(xmlelement(„Artikel“,
artnr)))
from artikelstamm
where …
```

Mit diesen vier Funktionen kommt man schon recht weit. Die Benennung der XML-Elemente unter Berücksichtigung der Groß-/Kleinschreibung gemäß XSD ist gewährleistet und die dort vorgegebene Struktur kann ebenfalls generiert werden, auch wenn das im Einzelfall keine triviale Aufgabe darstellt. Die Problematik dieses Ansatzes liegt eher im Umfang und in der Komplexität der SQL-Statements, wenn die zugrunde liegende relationale Struktur viele Tabellen umfasst, die in einer hierarchischen Beziehung zueinander stehen und

aus denen eine umfangreiche XML-Struktur mit zahlreichen Elementen und tiefer Verschachtelung zu erzeugen ist. Das gesamte XSD ist in einem einzigen SQL-Statement abzubilden. Darunter leidet die Lesbarkeit dieses Statements erheblich schnell, und die Wartung wird anspruchsvoll und entsprechend aufwändig.

Durch die Modularisierung einzelner Passagen lässt sich der schlechten Wartbarkeit zum Teil entgegenwirken. Schließlich befindet man sich in einer relationalen Datenbank und kann beispielsweise die Erzeugung einer Liste von Elementen in eine parametrierte Funktion auslagern, die XMLTYPE zurückgibt, und sie in bekannter Weise aufrufen. Damit ist außerdem die Wiederverwendbarkeit gegeben.

Objektrelationale Funktionen

Objekttypen sind benutzerdefinierte Datentypen. Sie bestehen aus beliebig vielen Attributen, die ihrerseits je einen Datentyp haben, der nativ oder ein anderer benutzerdefinierter Typ ist. Ein solcher Typ kann auch eine Liste eines Objekttyps sein (nested table). Damit lassen sich Objekte anlegen, deren Struktur den Vorgaben des XSD entspricht. Die erste Aufgabe besteht also darin, die erforderlichen Typen zu definieren. Dazu bieten sich zwei Vorgehensweisen an:

1. *Manuell*
   Hier ist zu beachten, dass bei der Benennung der Attribute Groß-/Kleinschreibung berücksichtigt wird, da deren Namen als XML-Elementnamen verwendet werden. Die Attributbezeichnung muss also dem Element aus dem XSD entsprechen. Wird dem Attributnamen ein „@“ vorangestellt, bewirkt dies, dass das Attribut als Attribut des XML-Elements verstanden wird, anderenfalls wird es zu einem Sub-Element des XML-Elements:

```
create or replace type
artikel_t
as object
(“ArtNr“ number(10),  “Be-
zeichnung“ varchar2(100));
```

2. *Generieren*
   Dazu registriert man das XSD in XML-DB und lässt XML-DB alle Typen erzeugen, welche die im XSD vorgefundene Struktur abbildet:

```
dbms_xmlschema.RegisterSchema
(schemaurl    => …
,schemadoc => …
,gentypes      => TRUE
,…);
```

   XML-DB erzeugt und benennt die Typen dabei nach definierten internen Regeln, die über sogenannte „Annotations“ im XSD beeinflusst werden können, sodass Art und Namen der Typen tatsächlich den Erfordernissen entsprechen.

Existieren die Objekttypen in der Datenbank, werden ihre Konstruktoren in einem (objekt-) relationalen SELECT-Statement verwendet, um eine Ergebnismenge zu erzeugen, deren Datentyp gleich dem Objekttyp ist. Zur Bildung von Listen werden die Funktionen COLLECT oder MULTISET eingesetzt:

```
select  artikel_t (artnr, get_
text(text_id) )
from artikelstamm
where …
```

Auch ein solches Statement ist lesbar und wartungsarm – wie oben angesprochen abhängig von der Komplexität und der Schachtelungstiefe. Ein Teil der Komplexität ist hier jedoch in den Objekttypen versteckt, sodass das SELECT-Statement in der Regel besser zu lesen ist als ein entsprechendes SQL/XML-Statement. Selbstverständlich bietet sich auch in diesem Ansatz die Möglichkeit der Modularisierung mit allen genannten Vorteilen. Hier geben die Funktionen nicht XMLTYPE, sondern den erwarteten (benutzerdefinierten) Typ zurück.

Persistenz oder nicht?

Gibt es einen Bedarf, die objektrelationalen Daten persistent zu speichern, etwa um sie auszuwerten oder manipulieren zu können? Diese Frage führt zur nächsten zu treffenden Entscheidung: Wenn es nur darum geht, die Daten unverändert im XML-Format in eine Datei auszugeben, reicht es völlig aus, dies über eine Object View zu realisieren. Eine Object View bezieht sich auf einen Objekttyp und stellt die über das oben formulierte objektrelationale SQL-Statement selektierten, relationalen Daten als Instanzen des auf dem Objekttypen basierenden Objekts dar:

```
create or replace view arti-
kel_ov
of artikel_t
with object_identifier (“@
ArtNr“) as
<object-relational select>;
```

Wenn die Ergebnismenge persistent gespeichert werden soll, gibt es zwei Möglichkeiten:

1. *Column Object in einer relationalen Tabelle*
   Die Objekt-Instanzen werden in einer Spalte einer relationalen Tabelle abgelegt, deren Datentyp ein Objekttyp ist. Die Tabelle kann weitere Spalten mit nativen Datentypen enthalten.
2. *Row Object in einer Object Table*
   Hier werden die Objekt-Instanzen in einer Objekt-Tabelle gespeichert. Eine Object-Table zeichnet sich unter anderem dadurch aus, dass sie genau eine Spalte umfasst, deren Datentyp ein Objekttyp ist.

Die Unterschiede zwischen Column und Row Object sind hier nicht von Belang. Beiden gemeinsam ist, dass sie von einem Objekttyp sind und deshalb mit einem INSERT-Statement gefüllt werden, dessen Ergebnismenge diesen Objekttyp liefert. Es ist genau dasselbe Statement, welches auch in der Object View verwendet wurde. Handelt es sich um ein Column Object, muss das INSERT-Statement außerdem noch Werte für die anderen Spalten in der Tabelle liefern.

Ausgabe als Datei

Mit der Aufbereitung der Daten als XML nach einer der beschriebenen Methoden wäre der erste Teil der Aufgabe erledigt. Auch der zweite Teil, die Ausgabe des XML in eine Datei,

ist nicht ganz so einfach zu lösen, wie man vermutet: Versucht man etwa, die Ergebnismenge eines SQL/XML-Statements in eine Datei zu spoolen, wird man feststellen, dass dies – selbst mit entsprechend gesetzten SET-Parametern in SQL*Plus wie LONG, WRAP, PAGESIZE etc. – nicht formatgerecht gelingt. Man erhält kein pretty print XML, denn die typisch hierarchische Anordnung der Elemente ist nicht gegeben. Abhilfe schafft, die Ergebnismenge, die vom Typ „XMLTYPE“ ist, in eine XMLTYPE-Spalte einer Hilfstabelle einzufügen. Diese lässt sich in einen BLOB konvertieren. Der Inhalt des BLOB wird anschließend mit DBMS_LOB.SUBSTR() ausgelesen und mit UTL_FILE.PUT_RAW() in die Datei geschrieben.

Auch bei den objektrelationalen Ansätzen erhält man durch Spooling nicht direkt das gewünschte Ergebnis. Soll Spooling verwendet werden, führt ein Weg dazu über das Auslesen der Object View beziehungsweise des Column oder Row Objects mit DBMS_XMLGEN.GETXML(). Diese Operation erzeugt ein CLOB-XML, das, in eine CLOB-Spalte einer Hilfstabelle eingefügt, anschließend mit Spool ausgegeben werden kann.

Eine weitere Möglichkeit im objektrelationalen Ansatz besteht darin, die Objekt-Instanzen mit DBMS_XMLGEN.GETXMLTYPE() als XML zu lesen und das Ergebnis so zu behandeln, wie es für die SQL/XML-Statements beschrieben wurde, also INSERT INTO XMLTYPE-Spalte einer Hilfstabelle, DBMS_LOB.SUBSTR(), UTL_FILE.PUT_RAW().

DBMS_XMLGEN.GETXMLTYPE beziehungsweise DBMS_XMLGEN.GETXML wird eine Query übergeben, die sich auf die Object View beziehungsweise das Column oder Row Object bezieht. Bei größeren Datenmengen verlangt die Anwendung von DBMS_XMLGEN nach reichlich RAM, da das gesamte XML-Dokument gemäß dem DOM-Ansatz (Document Object Model) zunächst im Hauptspeicher aufgebaut wird, bevor es weiterverarbeitet werden kann.

Deutlich weniger anspruchsvoll in dieser Hinsicht und schneller ist der SAX-Ansatz (Simple API for XML). Oracle stellt dafür zwei Java-Klassen (OracleXMLQuery und XMLSAXSerializer) zur Verfügung. In eine Java-Routine eingebunden, erzeugen sie auf der Basis einer als Parameter übergebenen Query die XML-Datei, ohne den DOM-Tree aufzubauen.

Im Projekt hat sich der Autor für den SAX-Ansatz in Verbindung mit persistenter Speicherung als Column Object entschieden. Die Speicherung bietet eine elegante Möglichkeit, Objekt-Instanzen miteinander zu vergleichen, und dies weitaus einfacher und performanter, als das in der relationalen Struktur möglich ist. So können Abweichungen zwischen zwei Datenlieferungen erkannt und behandelt werden.

Abbildung 1 zeigt alle bisher angesprochenen Komponenten und veranschaulicht ihr Zusammenwirken. Im objektrelationalen Ansatz bieten sich alternative Pfade an. Diese sind optisch durch die blauen Komponenten gekennzeichnet.

Fazit

Object Views und Column oder Row Objects setzen voraus, dass objektrelationale Strukturen aufgebaut werden. Das erfordert zunächst einmal ein Umdenken für den „relationalen“ Entwickler, bildet er die Entitäten doch nicht mehr ausschließlich in flachen Tabellen ab, sondern in geschalteten Objekten. Und es erhöht natürlich den initialen Entwicklungsaufwand. Dafür wird man mit leichterer Wartbarkeit belohnt und behält auch bei umfangreichen Strukturen den Überblick.

Die Entscheidung zwischen Object Views (nicht persistent) und Column oder Row Objects (persistent) wird wesentlich davon geprägt sein, ob die Objekt-Instanzen manipuliert, ausgewertet oder miteinander verglichen werden sollen. Besteht dieser Bedarf, beispielsweise um letzte Anpassungen der Struktur an das XSD vorzunehmen, kommt man an persistenter Speicherung nicht vorbei. Dass es auf dem Weg zur Ablage des XML im Dateisystem ganz ohne „Umweg“ über eine Zwischenspeicherung in einer CLOB- oder XMLTYPE-Spalte nicht geht, ist angesichts der einfachen und sicheren Realisierbarkeit mittels der verfügbaren Packages bzw. der Java-Klassen leicht zu verschmerzen.

Rolf Wesp
Trivadis AG
rolf.wesp@trivadis.com

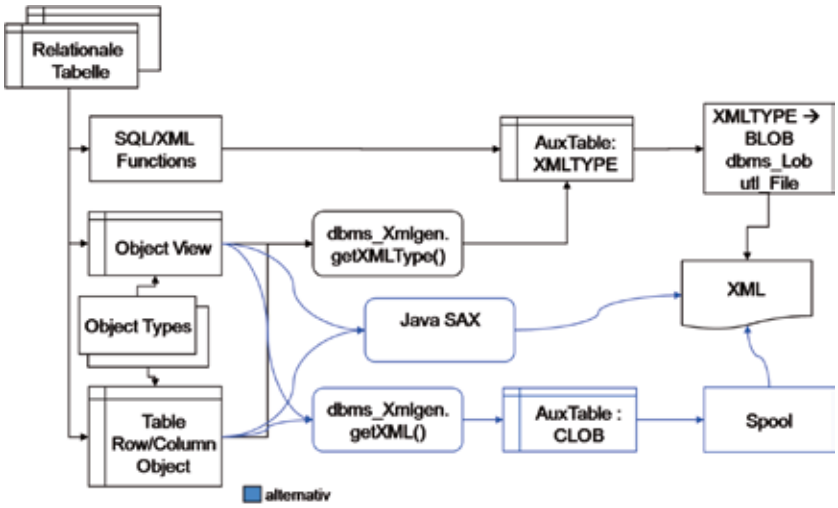

Abbildung 1: Die Komponenten im Überblick