9

Relational Views over XML Data

Relational database views over XML data provide conventional, relational access to XML content.

- Introduction to Creating and Using Relational Views over XML Data
  You can use the XML-specific functions and methods provided by Oracle XML DB to create conventional database views that provide relational access to XML content. This lets programmers, tools, and applications that understand Oracle Database, but not necessarily XML, work with XML content stored in the database.
- Creating a Relational View over XML: One Row for Each XML Document
  To expose each document in an `XMLType` table as a row in a relational view, use `CREATE OR REPLACE VIEW AS SELECT`, selecting from a join of the `XMLType` table and a relational table that you create from the XML data using SQL/XML function `XMLTable`.
- Creating a Relational View over XML: Mapping XML Nodes to Columns
  To expose data from multiple levels of an `XMLType` table as individual rows in a relational view, apply SQL/XML function `XMLTable` to each level. Use this technique whenever there is a one-to-*many* (1:N) relationship between documents in the `XMLType` table and rows in the view.
- Indexing Binary XML Data Exposed Using a Relational View
  If the relational columns of the structured component of an `XMLIndex` index over binary XML data match the columns of a relational view over that data, then the view too is effectively indexed.
- Querying XML Content As Relational Data
  Examples here show relational queries of XML data. They illustrate some of the benefits provided by creating relational views over `XMLType` tables and columns.

Introduction to Creating and Using Relational Views over XML Data

You can use the XML-specific functions and methods provided by Oracle XML DB to create conventional database views that provide relational access to XML content. This lets programmers, tools, and applications that understand Oracle Database, but not necessarily XML, work with XML content stored in the database.

The relational views can use XQuery expressions and SQL/XML functions such as `XMLTable` to define a mapping between columns in the view and nodes in an XML document.

Related Topics

- XQuery and Oracle XML DB
  The XQuery language is one of the main ways that you interact with XML data in Oracle XML DB. Support for the language includes SQL*Plus command`XQUERY` and SQL/XML functions `XMLQuery`, `XMLTable`, `XMLExists`, and `XMLCast`.

- Indexes for XMLType Data
  You can create indexes on your XML data, to focus on particular parts of it that you query often and thus improve performance. There are various ways that you can index XMLType data, whether it is XML schema-based or non-schema-based, and regardless of the XMLType storage model you use.

Creating a Relational View over XML: One Row for Each XML Document

To expose each document in an XMLType table as a row in a relational view, use CREATE OR REPLACE VIEW AS SELECT, selecting from a join of the XMLType table and a relational table that you create from the XML data using SQL/XML function XMLTable.

You use standard SQL/XML function XMLTable to map nodes in the XML document to columns in the view. Use this technique whenever there is a one-to-*one* (1:1) relationship between documents in the XMLType table and the rows in the view.

Example 9-1 creates relational view purchaseorder_master_view, which has one row for each row in XMLType table po_binaryxml.

Example 9-1   Creating a Relational View of XML Content

```
CREATE TABLE po_binaryxml OF XMLType
  XMLTYPE STORE AS BINARY XML;

INSERT INTO po_binaryxml SELECT OBJECT_VALUE FROM OE.purchaseorder;

CREATE OR REPLACE VIEW purchaseorder_master_view AS
  SELECT po.*
    FROM po_binaryxml pur,
         XMLTable(
           '$p/PurchaseOrder' PASSING pur.OBJECT_VALUE as "p"
           COLUMNS
             reference       VARCHAR2(30)   PATH 'Reference',
             requestor       VARCHAR2(128)  PATH 'Requestor',
             userid          VARCHAR2(10)   PATH 'User',
             costcenter      VARCHAR2(4)    PATH 'CostCenter',
             ship_to_name    VARCHAR2(20)   PATH 'ShippingInstructions/name',
             ship_to_address VARCHAR2(256)  PATH 'ShippingInstructions/address',
             ship_to_phone   VARCHAR2(24)   PATH 'ShippingInstructions/telephone',
             instructions    VARCHAR2(2048) PATH 'SpecialInstructions') po;

View created.

DESCRIBE purchaseorder_master_view

Name                Null?    Type
--------------------------------------------
REFERENCE                    VARCHAR2(30)
REQUESTOR                    VARCHAR2(128)
USERID                       VARCHAR2(10)
COSTCENTER                   VARCHAR2(4)
SHIP_TO_NAME                 VARCHAR2(20)
SHIP_TO_ADDRESS              VARCHAR2(256)
```

```
SHIP_TO_PHONE             VARCHAR2(24)
INSTRUCTIONS              VARCHAR2(2048)
```

Creating a Relational View over XML: Mapping XML Nodes to Columns

To expose data from multiple levels of an XMLType table as individual rows in a relational view, apply SQL/XML function XMLTable to each level. Use this technique whenever there is a one-to-*many* (1:N) relationship between documents in the XMLType table and rows in the view.

That is, you use the same general approach as for breaking up a single level (see Creating a Relational View over XML: One Row for Each XML Document): Define the columns making up the view, and map the XML nodes to those columns. But in this case you apply XMLTable to each document level that is to be broken up and stored in relational columns.

For example, each PurchaseOrder element contains a LineItems element, which in turn contains one or more LineItem elements. Each LineItem element has child elements, such as Description, and an ItemNumber attribute. To make such lower-level data accessible as a relational value, use XMLTable to project both the PurchaseOrder element and the LineItem collection.

When element PurchaseOrder is broken up, its descendant LineItem element is mapped to a column of type XMLType, which contains an XML fragment. That column is then passed to a second call to XMLTable to be broken into its various parts as multiple columns of relational values.

Example 9-2 illustrates this. It uses XMLTable to effect a one-to-*many* (1:N) relationship between the documents in XMLType table po_binaryxml and the rows in relational view purchaseorder_detail_view. The view provides access to the individual members of a collection and exposes the collection members as a set of rows.

In Example 9-2, there is one row in view purchaseorder_detail_view for each LineItem element in the XML documents stored in XMLType table po_binaryxml.

The CREATE OR REPLACE VIEW statement of Example 9-2 defines the set of relational columns that make up the view. The SELECT statement passes table po_binaryxml as context to function XMLTable to create virtual table p, which has columns reference and lineitem. These columns contain the Reference and LineItem elements of the purchase-order documents, respectively.

Column lineitem contains a collection of LineItem elements as an XMLType instance — one row for each element. These rows are in turn passed to a second XMLTable expression to serve as its context. This second XMLTable expression creates a virtual table of line-item rows, with columns corresponding to various descendant nodes of element LineItem. Most of these descendants are attributes (ItemNumber, Part/@Id, and so on). One of the descendants is the child element Description.

Element Reference is projected in view purchaseorder_detail_view as column reference. It provides a foreign key that can be used to join rows in view purchaseorder_detail_view to corresponding rows in view purchaseorder_master_view. The correlated join in the CREATE OR REPLACE VIEW statement ensures that the one-to-many (1:N) relationship between element Reference and the associated LineItem elements is maintained whenever the view is accessed.

Example 9-2 Accessing Individual Members of a Collection Using a View

```
CREATE OR REPLACE VIEW purchaseorder_detail_view AS
  SELECT po.reference, li.*
    FROM po_binaryxml p,
         XMLTable('/PurchaseOrder' PASSING p.OBJECT_VALUE
                  COLUMNS
                    reference VARCHAR2(30) PATH 'Reference',
                    lineitem  XMLType      PATH 'LineItems/LineItem') po,
         XMLTable('/LineItem' PASSING po.lineitem
                  COLUMNS
                    itemno      NUMBER(38)    PATH '@ItemNumber',
                    description VARCHAR2(256) PATH 'Description',
                    partno      VARCHAR2(14)  PATH 'Part/@Id',
                    quantity    NUMBER(12, 2) PATH 'Part/@Quantity',
                    unitprice   NUMBER(8, 4)  PATH 'Part/@UnitPrice') li;

View created.

DESCRIBE purchaseorder_detail_view
Name           Null?    Type
----------------------------
REFERENCE               VARCHAR2(30)
ITEMNO                  NUMBER(38)
DESCRIPTION             VARCHAR2(256)
PARTNO                  VARCHAR2(14)
QUANTITY                NUMBER(12,2)
UNITPRICE               NUMBER(8,4)
```

Indexing Binary XML Data Exposed Using a Relational View

If the relational columns of the structured component of an `XMLIndex` index over binary XML data match the columns of a relational view over that data, then the view too is effectively indexed.

When the `XMLType` data that is exposed in a relational view is stored as binary XML, you can typically improve performance by creating an `XMLIndex` index that has a structured component that matches the view columns. Such an index projects parts of the XML data onto relational columns, just as the view does. When the columns of the index match the columns of the view, the view is itself indexed.

To simplify the creation of such an `XMLIndex` index, you can PL/SQL function `DBMS_XMLSTORAGE_MANAGE.getSIDXDefFromView` to provide exactly the `XMLTable` expression needed for creating the index. That is the sole purpose of this function: to return an `XMLTable` expression that you can use to create an `XMLIndex` index for a relational view. It takes the view as argument and returns a `CLOB` instance. Example 9-3 illustrates this.

Example 9-4 shows the `XMLTable` expression used in Example 9-3.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL function `DBMS_XMLSTORAGE_MANAGE.getSIDXDefFromView`

Example 9-3 XMLIndex Index that Matches Relational View Columns

```
CALL DBMS_XMLINDEX.registerParameter(
  'my_param',
  DBMS_XMLSTORAGE_MANAGE.getSIDXDefFromView('PURCHASEORDER_MASTER_VIEW'));[1]

CREATE INDEX my_idx on po_binaryxml (OBJECT_VALUE) INDEXTYPE IS XDB.XMLIndex
  PARAMETERS ('PARAM my_param');
```

Example 9-4 XMLTable Expression Returned by PL/SQL Function getSIDXDefFromView

```
SELECT DBMS_XMLSTORAGE_MANAGE.getSIDXDefFromView('PURCHASEORDER_MASTER_VIEW')
  FROM DUAL;

XMLTABLE po_binaryxml_XTAB_1 '/PurchaseOrder' PASSING OBJECT_VALUE
  COLUMNS
    reference       VARCHAR2   (30) PATH 'Reference',
    requestor       VARCHAR2  (128) PATH 'Requestor',
    userid          VARCHAR2   (10) PATH 'User',
    costcenter      VARCHAR2    (4) PATH 'CostCenter',
    ship_to_name    VARCHAR2   (20) PATH 'ShippingInstructions/name',
    ship_to_address VARCHAR2  (256) PATH 'ShippingInstructions/address',
    ship_to_phone   VARCHAR2   (24) PATH 'ShippingInstructions/telephone',
    instructions    VARCHAR2 (2048) PATH 'SpecialInstructions'
```

Related Topics

- Use of XMLIndex with a Structured Component
  An XMLIndex structured component indexes specific islands of structure in your XML data.

Querying XML Content As Relational Data

Examples here show relational queries of XML data. They illustrate some of the benefits provided by creating relational views over XMLType tables and columns.

Example 9-5 and Example 9-6 show how to query master and detail relational views of XML data. Example 9-5 queries the master view to select the rows where column userid starts with S.

Example 9-6 joins the master view and the detail view. It selects the purchaseorder_detail_view rows where the value of column itemno is 1 and the corresponding purchaseorder_master_view row contains a userid column with the value SBELL.

Example 9-7 shows how to use relational views over XML content to perform business-intelligence queries on XML documents. The example query selects PurchaseOrder documents that contain orders for titles identified by UPC codes 715515009058 and 715515009126.

The query in Example 9-7 determines the number of copies of each film title that are ordered in each PurchaseOrder document. For example, for part number 715515009126, there are four PurchaseOrder documents where one copy of the item is ordered and seven PurchaseOrder documents where three copies of the item are ordered.

[1] The view-name argument to getSIDXDefFromView must be uppercase, because that is how the name is recorded.

Example 9-5   Querying Master Relational View of XML Data

```
SELECT reference, costcenter, ship_to_name
  FROM purchaseorder_master_view
  WHERE userid LIKE 'S%';

REFERENCE                      COST SHIP_TO_NAME
------------------------------ ---- --------------
SBELL-20021009123336231PDT     S30  Sarah J. Bell
SBELL-20021009123336331PDT     S30  Sarah J. Bell
SKING-20021009123336321PDT     A10  Steven A. King
...
36 rows selected.
```

Example 9-6   Querying Master and Detail Relational Views of XML Data

```
SELECT d.reference, d.itemno, d.partno, d.description
  FROM purchaseorder_detail_view d, purchaseorder_master_view m
  WHERE m.reference = d.reference
    AND m.userid = 'SBELL'
    AND d.itemno = 1;

REFERENCE                      ITEMNO PARTNO         DESCRIPTION
--------------------------------------------------------------------------
SBELL-20021009123336231PDT          1 37429165829    Juliet of the Spirits
SBELL-20021009123336331PDT          1 715515009225   Salo
SBELL-20021009123337353PDT          1 37429141625    The Third Man
SBELL-20021009123338304PDT          1 715515009829   Nanook of the North
SBELL-20021009123338505PDT          1 37429122228    The 400 Blows
SBELL-20021009123335771PDT          1 37429139028    And the Ship Sails on
SBELL-20021009123335280PDT          1 715515011426   All That Heaven Allows
SBELL-2002100912333763PDT           1 715515010320   Life of Brian - Python
SBELL-2002100912333601PDT           1 715515009058   A Night to Remember
SBELL-20021009123336362PDT          1 715515012928   In the Mood for Love
SBELL-20021009123336532PDT          1 37429162422    Wild Strawberries
SBELL-20021009123338204PDT          1 37429168820    Red Beard
SBELL-20021009123337673PDT          1 37429156322    Cries and Whispers

13 rows selected.
```

Example 9-7   Business-Intelligence Query of XML Data Using a View

```
SELECT partno, count(*) "No of Orders", quantity "No of Copies"
  FROM purchaseorder_detail_view
  WHERE partno IN (715515009126, 715515009058)
  GROUP BY rollup(partno, quantity);

PARTNO         No of Orders No of Copies
-------------- ------------ ------------
715515009058              7            1
715515009058              9            2
715515009058              5            3
715515009058              2            4
715515009058             23
715515009126              4            1
```

```
715515009126               7             3
715515009126              11
                          34
9 rows selected.
```