19

XPath Rewrite for Object-Relational Storage

For `XMLType` data stored object-relationally, queries involving XPath expression arguments to various SQL functions can often be automatically rewritten to queries against the underlying SQL tables, which are highly optimized.

- Overview of XPath Rewrite for Object-Relational Storage
  Oracle XML DB can often optimize queries that use XPath expressions — for example, queries involving SQL functions such as `XMLQuery`, `XMLTable`, and `XMLExists`, which take XPath (XQuery) expressions as arguments. The XPath expression is, in effect, evaluated against the XML document without ever constructing the XML document in memory.
- Common XPath Expressions that Are Rewritten
  The most common XPath expressions that are rewritten during XPath rewrite are described.
- XPath Rewrite for Out-Of-Line Tables
  XPath expressions that involve elements stored out of line can be automatically rewritten. The rewritten query involves a join with the out-of-line table.
- Guidelines for Using Execution Plans to Analyze and Optimize XPath Queries
  Guidelines are presented for using execution plans to analyze query execution in order to (a) determine whether XPath rewrite occurs and (b) optimize query execution by using secondary indexes. These guidelines apply only to `XMLType` data that is stored object-relationally.

Related Topics

- Performance Tuning for XQuery
  A SQL query that involves XQuery expressions can often be automatically rewritten (optimized) in one or more ways. This optimization is referred to as **XML query rewrite** or optimization. When this happens, the XQuery expression is, in effect, evaluated directly against the XML document without constructing a DOM in memory.

Overview of XPath Rewrite for Object-Relational Storage

Oracle XML DB can often optimize queries that use XPath expressions — for example, queries involving SQL functions such as `XMLQuery`, `XMLTable`, and `XMLExists`, which take XPath (XQuery) expressions as arguments. The XPath expression is, in effect, evaluated against the XML document without ever constructing the XML document in memory.

This optimization is called **XPath rewrite**. It is a proper subset of XML query optimization, which also involves optimization of XQuery expressions, such as FLWOR expressions, that are not XPath expressions. XPath rewrite also enables indexes, if present on the column, to be used in query evaluation by the Optimizer.

The XPath expressions that can be rewritten by Oracle XML DB are a proper subset of those that are supported by Oracle XML DB. Whenever you can do so without losing functionality, use XPath expressions that can be rewritten.

XPath rewrite can occur in these contexts (or combinations thereof):

- When XMLType data is stored in an object-relational column or table or when an XMLType view is built on relational data.
- When you use an XMLIndex index.

The first case, rewriting queries that use object-relational XML data or XMLType views, is covered here. The XMLType views can be XML schema-based or not. Object-relational storage of XMLType data is always XML schema-based. Examples in this chapter are related to XML schema-based tables.

When XPath rewrite is possible for object-relational XML data, the database optimizer can derive an execution plan based on conventional relational algebra. This in turn means that Oracle XML DB can leverage all of the features of the database and ensure that SQL statements containing XQuery and XPath expressions are executed in a highly performant and efficient manner. There is little overhead with this rewriting, so Oracle XML DB executes XQuery-based and XPath-based queries at near-relational speed.

In certain cases, XPath rewrite is not possible. This typically occurs when there is no SQL equivalent of the XPath expression. In this situation, Oracle XML DB performs a functional evaluation of the XPath expressions, which is generally more costly, especially if the number of documents to be processed is large.

Example 19-1 illustrates XPath rewrite for a simple query that uses an XPath expression.

Example 19-1 XPath Rewrite

```
SELECT po.OBJECT_VALUE FROM purchaseorder po
  WHERE XMLCast(XMLQuery('$p/PurchaseOrder/Requestor'
                         PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
                AS VARCHAR2(128))
        = 'Sarah J. Bell';
```

The XMLCast(XMLQuery...)) expression here is rewritten to the underlying relational column that stores the requestor information for the purchase order. The query is rewritten to something like the following:[1]

```
SELECT OBJECT_VALUE FROM purchaseorder p
 WHERE CAST (p."XMLDATA"."REQUESTOR" AS VARCHAR2(128)) = 'Sarah J. Bell';
```

Related Topics

- Relational Views over XML Data
  Relational database views over XML data provide conventional, relational access to XML content.
- XMLType Views
  You can create XMLType views over relational and object-relational data.
- Indexing XMLType Data Stored Object-Relationally
  You can effectively index XMLType data that is stored object-relationally by creating B-tree indexes on the underlying database columns that correspond to XML nodes.
- XMLIndex

[1] This example uses sample database schema OE and its table purchaseorder. The XML schema for this table is annotated with attribute SQLName to specify SQL object attribute names such as REQUESTOR — see Example A-2. Without such annotations, this example would use p."XMLDATA".**"Requestor"**, not p."XMLDATA".".**REQUESTOR**".

- XML Schema Annotation Guidelines for Object-Relational Storage
  For `XMLType` data stored object-relationally, careful planning is called for, to optimize performance. Similar considerations are in order as for relational data: entity-relationship models, indexing, data types, table partitions, and so on. To enable XPath rewrite and achieve optimal performance, you implement many such design choices using XML schema annotations.

Common XPath Expressions that Are Rewritten

The most common XPath expressions that are rewritten during XPath rewrite are described.

Table 19-1 presents the descriptions

Table 19-1 Sample of XPath Expressions that Are Rewritten to Underlying SQL Constructs

XPath Expression for Translation	Description
Simple XPath expressions (expressions with `child` and `attribute` axes only): `/PurchaseOrder/@Reference` `/PurchaseOrder/Requestor`	Involves traversals over object type attributes only, where the attributes are simple scalar or object types themselves.
Collection traversal expressions: `/PurchaseOrder/LineItems/LineItem/Part/@Id`	Involves traversal of collection expressions. The only axes supported are child and attribute axes. Collection traversal is not supported if the SQL function is used during a `CREATE INDEX` operation.
Predicates: `[Requestor = "Sarah J. Bell"]`	Predicates in the XPath are rewritten into SQL predicates.
List index (positional predicate): `LineItem[1]`	Indexes are rewritten to access the nth item in a collection.
Wildcard traversals: `/PurchaseOrder/*/Part/@Id`	If the wildcard can be translated to one or more simple XPath expressions, then it is rewritten.
Descendant axis (XML schema-based data only), without recursion: `/PurchaseOrder//Part/@Id`	Similar to a wildcard expression. The `descendant` axis is rewritten if it can be mapped to one or more simple XPath expressions.
Descendant axis (XML schema-based data only), with *recursion*: `/PurchaseOrder//Part/@Id`	The `descendant` axis is rewritten if both of these conditions holds: • All simple XPath expressions to which this XPath expression expands map to the same out-of-line table. • Any simple XPath expression to which this XPath expression does not expand does not map to that out-of-line table.
XPath functions	Some XPath functions are rewritten. These functions include `not`, `floor`, `ceiling`, `substring`, and `string-length`.

See Also:

Performance Tuning for XQuery for information about rewrite of XQuery expressions

XPath Rewrite for Out-Of-Line Tables

XPath expressions that involve elements stored out of line can be automatically rewritten. The rewritten query involves a join with the out-of-line table.

Example 19-2 shows such a query. The XQuery expression is rewritten to a SQL `EXISTS` subquery that queries table `addr_tab`, joining it with table `emp_tab` using the object identifier column in `addr_tab`. The optimizer uses full table scans of tables `emp_tab` and `addr_tab`. If there are many entries in the `addr_tab`, then you can try to make this query more efficient by creating an index on the city, as shown in Example 19-3. An explain-plan fragment for the same query as in Example 19-2 shows that the city index is picked up.

Note:

When gathering statistics for the optimizer on an `XMLType` table that is stored object-relationally, Oracle recommends that you gather statistics on *all* of the tables defined by the XML schema, that is, all of the tables in `USER_XML_TABLES`. You can use procedure `DBMS_STATS.gather_schema_stats` to do this, or use `DBMS_STATS.gather_table_stats` on each such table. This informs the optimizer about all of the dependent tables that are used to store the `XMLType` data.

Example 19-2   XPath Rewrite for an Out-Of-Line Table

```
SELECT XMLCast(XMLQuery('declare namespace x = "http://www.oracle.com/emp.xsd"; (: :)
                         /x:Employee/Name' PASSING OBJECT_VALUE RETURNING CONTENT)
               AS VARCHAR2(20))
  FROM emp_tab
  WHERE XMLExists('declare namespace x = "http://www.oracle.com/emp.xsd"; (: :)
                   /x:Employee/Addr[City="San Francisco"]' PASSING OBJECT_VALUE);

XMLCAST(XMLQUERY(...
--------------------
Abe Bee
Eve Fong
George Hu
Iris Jones
Karl Luomo
Marina Namur
Omar Pinano
Quincy Roberts

8 rows selected.
```

Example 19-3 Using an Index with an Out-Of-Line Table

```
CREATE INDEX addr_city_idx
  ON addr_tab (extractValue(OBJECT_VALUE, '/Addr/City'));


|   2 |   TABLE ACCESS BY INDEX ROWID| ADDR_TAB      |     1 |  2012 |     1   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN          | ADDR_CITY_IDX |     1 |       |     1   (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL          | EMP_TAB       |    16 | 32464 |     2   (0)| 00:00:01 |
```

Guidelines for Using Execution Plans to Analyze and Optimize XPath Queries

Guidelines are presented for using execution plans to analyze query execution in order to (a) determine whether XPath rewrite occurs and (b) optimize query execution by using secondary indexes. These guidelines apply only to `XMLType` data that is stored object-relationally.

Use these guidelines together, taking all that apply into consideration.

XPath rewrite for object-relational storage means that a query that selects XML fragments defined by an XPath expression is rewritten to a SQL `SELECT` statement on the underlying object-relational tables and columns. These underlying tables can include out-of-line tables.

You can use PL/SQL procedure `DBMS_XMLSTORAGE_MANAGE.XPath2TabColMapping` to find the names of the underlying tables and columns that correspond to a given XPath expression.

- Guideline: Look for underlying tables versus XML functions in execution plans
  The execution plan of a query that is rewritten refers to the names of the object-relational tables and columns that underlie the queried `XMLType` data. These names can be meaningful to you if they are derived from XML element or attribute names or if XML Schema annotation `xdb:defaultTable` was used.
- Guideline: Name the object-relational tables, so you recognize them in execution plans
  When designing an XML schema, use annotation `xdb:defaultTable` to name the underlying tables that correspond to elements that you select in queries where performance is important. This lets you easily recognize them in an execution plan, indicating by their presence or absence whether the query has been rewritten.
- Guideline: Create an index on a column targeted by a predicate
  You can sometimes improve the performance of a query that is rewritten to include a SQL predicate, by creating an index that applies to the column targeted by the predicate.
- Guideline: Create indexes on ordered collection tables
  If a collection is stored as an ordered collection table (OCT) or as an `XMLType` instance, then you can directly access members of the collection. Each member becomes a table row, so you can access it directly with SQL. You can often improve performance by indexing such collection members.
- Guideline: Use XMLOptimizationCheck to determine why a query is not rewritten
  If a query has not been optimized, you can use system variable `XMLOptimizationCheck` to try to determine why.

Related Topics

- XPath Rewrite for Out-Of-Line Tables
  XPath expressions that involve elements stored out of line can be automatically rewritten. The rewritten query involves a join with the out-of-line table.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information about procedure `XPath2TabColMapping`

Guideline: Look for underlying tables versus XML functions in execution plans

The execution plan of a query that is rewritten refers to the names of the object-relational tables and columns that underlie the queried `XMLType` data. These names can be meaningful to you if they are derived from XML element or attribute names or if XML Schema annotation `xdb:defaultTable` was used.

Otherwise, these names are system-generated and have no obvious meaning. In particular, they do not reflect the corresponding XML element or attribute names.

Also, some system-generated columns are generally hidden. You do not see them if you use the SQL `describe` command. They nevertheless show up in execution plans.

The plan of a query that has not been rewritten shows only the base table names, and it typically refers to user-level XML functions, such as `XMLExists`. Look for this difference to determine whether a query has been optimized. The XML function name shown in an execution plan is actually the internal name (for example, `XMLEXISTS2`), which is sometimes slightly different from the user-level name.

Example 19-4 shows the kind of execution plan output that is generated when Oracle XML DB cannot perform XPath rewrite. The plan here is for a query that uses SQL/XML function `XMLExists`. The corresponding internal function `XMLExists2` appears in the plan output, indicating that the query is not rewritten.

In this situation, Oracle XML DB constructs a pre-filtered result set based on any other conditions specified in the query `WHERE` clause. It then filters the rows in this potential result set to determine which rows belong in the result set. The filtering is performed by *constructing a DOM on each document* and performing a **functional evaluation** using the methods defined by the DOM API to determine whether or not each document is a member of the result set.

Example 19-4 Execution Plan Generated When XPath Rewrite Does Not Occur

```
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(XMLEXISTS2('$p/PurchaseOrder[User="SBELL"]' PASSING BY VALUE
              SYS_MAKEXML('61687B202644E297E040578C8A175C1D',4215,"PO"."XMLEXTRA","PO"."X
              MLDATA") AS "p")=1)
```

Guideline: Name the object-relational tables, so you recognize them in execution plans

When designing an XML schema, use annotation `xdb:defaultTable` to name the underlying tables that correspond to elements that you select in queries where performance is important. This lets you easily recognize them in an execution plan, indicating by their presence or absence whether the query has been rewritten.

For collection tables, there is no corresponding XML schema annotation. To give user-friendly names to your collection tables you must first register the XML schema. Then you can use PL/SQL procedure `DBMS_XMLSTORAGE_MANAGE.renameCollectionTable` to rename the tables that were created during registration, which have system-generated names.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information about procedure `renameCollectionTable`

Guideline: Create an index on a column targeted by a predicate

You can sometimes improve the performance of a query that is rewritten to include a SQL predicate, by creating an index that applies to the column targeted by the predicate.

A query resulting from XPath rewrite sometimes includes a SQL predicate (`WHERE` clause). This can happen even if the original query does not use an XPath predicate, and it can happen even if the original query does not have a SQL `WHERE` clause.

When this happens, you can sometimes improve performance by creating an index on the column that is targeted by the SQL predicate, or by creating an index on a function application to that column.

Example 19-1 illustrates XPath rewrite for a query that includes a `WHERE` clause. Example 19-5 shows the predicate information from an execution plan for this query.

The predicate information indicates that the expression `XMLCast(XMLQuery...))` is rewritten to an application of SQL function `cast` to the underlying relational column that stores the requestor information for the purchase order, `SYS_NC0021$`. This column name is system-generated. The execution plan refers to this system-generated name, in spite of the fact that the governing XML schema uses annotation `SQLName` to name this column `REQUESTOR`.

Because these two names (user-defined and system-generated) refer to the same column, you can create a B-tree index on this column using either name. Alternatively, you can use the `extractValue` shortcut to create the index, by specifying an XPath expression that targets the purchase-order requestor data.

You can obtain the names of the underlying table and columns that correspond to a given XPath expression using procedure `DBMS_XMLSTORAGE_MANAGE.XPath2TabColMapping`. Example 19-6 illustrates this for the XPath expression `/PurchaseOrder/Requestor` used in the `WHERE` clause of Example 19-1.

If you provide an XPath expression that contains a wildcard or a descendent axis then multiple tables and columns might be selected. In that case procedure `XPath2TabColMapping` returns multiple `<Mapping>` elements, one for each table-column pair.

You can then use the table and column names retrieved this way in a `CREATE INDEX` statement to create an index that corresponds to the XPath expression. Example 19-7 shows three equivalent ways to create a B-tree index on the predicate-targeted column.

However, for this particular query it makes sense to create a function-based index, using a functional expression that matches the one in the rewritten query. Example 19-8 illustrates this.

Example 19-9 shows an execution plan that indicates that the index is picked up.

In the particular case of this query, the original functional expression applies `XMLCast` to `XMLQuery` to target a singleton element, `Requestor`. This is a special case, where you can as a shortcut use such a functional expression directly in the `CREATE INDEX` statement. That statement is rewritten to create an index on the underlying scalar data. Example 19-10, which targets an XPath expression, thus has the same effect as Example 19-8, which targets the corresponding object-relational column.

See Also:

- Indexing Non-Repeating Text Nodes or Attribute Values for information about using the shortcut of `XMLCast` applied to `XMLQuery` and the `extractValue` shortcut to index singleton data
- *Oracle Database PL/SQL Packages and Types Reference* for information about procedure `XPath2TabColMapping`

Example 19-5   Analyzing an Execution Plan to Determine a Column to Index

```
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(CAST("PURCHASEORDER"."SYS_NC00021$" AS VARCHAR2(128))='Sarah
              J. Bell' AND SYS_CHECKACL("ACLOID","OWNERID",xmltype('<privilege
              xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
              http://xmlns.oracle.com/xdb/acl.xsd DAV:http://xmlns.oracle.com/xdb/dav.xsd
              "><read-properties/><read-contents/></privilege>'))=1)
```

Example 19-6   Using DBMS_XMLSTORAGE_MANAGE.XPATH2TABCOLMAPPING

```
SELECT DBMS_XMLSTORAGE_MANAGE.XPath2TabColMapping(USER,
                                                  'PURCHASEORDER',
                                                  '',
                                                  '/PurchaseOrder/Requestor',
                                                  '')
  FROM  DUAL;

DBMS_XMLSTORAGE_MANAGE.XPath2TabColMapping(US
---------------------------------------------
<Result>
  <Mapping TableName="PURCHASEORDER" ColumnName="SYS_NC00021$"/>
</Result>
```

Example 19-7   Creating an Index on a Column Targeted by a Predicate

```
CREATE INDEX requestor_index ON purchaseorder ("SYS_NC00021$");

CREATE INDEX requestor_index ON purchaseorder ("XMLDATA"."REQUESTOR");

CREATE INDEX requestor_index ON purchaseorder
  (extractvalue(OBJECT_VALUE, '/PurchaseOrder/Requestor'));
```

Example 19-8   Creating a Function-Based Index for a Column Targeted by a Predicate

```
CREATE INDEX requestor_index ON purchaseorder
  (cast("XMLDATA"."REQUESTOR" AS VARCHAR2(128)));
```

Example 19-9   Execution Plan Showing that Index Is Picked Up

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                   | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                 |     1 |   524 |     2   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| PURCHASEORDER   |     1 |   524 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN          | REQUESTOR_INDEX |     1 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(SYS_CHECKACL("ACLOID","OWNERID",xmltype('<privilege
              xmlns="http://xmlns.oracle.com/xdb/acl.xsd"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xmlns.oracle.com/xdb/acl.xsd
                                  http://xmlns.oracle.com/xdb/acl.xsd
              DAV:http://xmlns.oracle.com/xdb/dav.xsd">
              <read-properties/><read-contents/></privilege>'))=1)
   2 - access(CAST("SYS_NC00021$" AS VARCHAR2(128))='Sarah J. Bell')
```

Example 19-10   Creating a Function-Based Index for a Column Targeted by a Predicate

```
CREATE INDEX requestor_index
  ON purchaseorder po
     (XMLCast(XMLQuery('$p/PurchaseOrder/Requestor' PASSING po.OBJECT_VALUE AS "p"
                                                    RETURNING CONTENT)
              AS VARCHAR2(128)));
```

Guideline: Create indexes on ordered collection tables

If a collection is stored as an ordered collection table (OCT) or as an XMLType instance, then you can directly access members of the collection. Each member becomes a table row, so you can access it directly with SQL. You can often improve performance by indexing such collection members.

You do this by creating a *composite* index on (a) the object attribute that corresponds to the collection XML element or its attribute and (b) pseudocolumn NESTED_TABLE_ID.

Example 19-11 shows the execution plan for a query to find the Reference elements in documents that contain an order for part number 717951002372 (Part element with an Id attribute of value 717951002372). The collection of LineItem elements is stored as rows in the ordered collection table lineitem_table.

> **Note:**
>
> Example 19-11 does not use the purchaseorder table from sample database schema OE. It uses a purchaseorder table that uses an ordered collection table (OCT) named lineitem_table for the collection element LineItem.

The execution plan shows a full scan of ordered collection table lineitem_table. This could be acceptable if there were only a few hundred documents in the purchaseorder table, but it would be unacceptable if there were thousands or millions of documents in the table.

To improve the performance of such a query, you can create an index that provides direct access to pseudocolumn `NESTED_TABLE_ID`, given the value of attribute `Id`. Unfortunately, Oracle XML DB does not allow indexes on collections to be created using XPath expressions directly. To create the index, you must understand the structure of the SQL object that is used to manage the `LineItem` elements. Given this information, you can create the required index using conventional object-relational SQL.

In this case, element `LineItem` is stored as an instance of object type `lineitem_t`. Element `Part` is stored as an instance of SQL data type `part_t`. XML attribute `Id` is mapped to object attribute `part_number`. Given this information, you can create a *composite index* on attribute `part_number` and pseudocolumn `NESTED_TABLE_ID`, as shown in Example 19-12. This index provides direct access to those purchase-order documents that have `LineItem` elements that reference the required part.

Example 19-11   Execution Plan for a Selection of Collection Elements

```
SELECT XMLCast(XMLQuery('$p/PurchaseOrder/Reference'
                        PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
               AS VARCHAR2(4000)) "Reference"
  FROM purchaseorder
  WHERE XMLExists('$p/PurchaseOrder/LineItems/LineItem/Part[@Id="717951002372"]'
                  PASSING OBJECT_VALUE AS "p");
```

```
----------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                     | Rows   | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                          |    21 |  2352 |    20  (10)| 00:00:01 |
|*  1 |  HASH JOIN RIGHT SEMI        |                          |    21 |  2352 |    20  (10)| 00:00:01 |
|   2 |   JOIN FILTER CREATE         | :BF0000                  |    22 |   880 |    14   (8)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | LINEITEM_TABLE           |    22 |   880 |    14   (8)| 00:00:01 |
|   4 |   JOIN FILTER USE            | :BF0000                  |   132 |  9504 |     5   (0)| 00:00:01 |
|*  5 |    TABLE ACCESS FULL         | PURCHASEORDER            |   132 |  9504 |     5   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("NESTED_TABLE_ID"="PURCHASEORDER"."SYS_NC0003400035$")
   3 - filter("SYS_NC00011$"='717951002372')
   5 - filter(SYS_OP_BLOOM_FILTER(:BF0000,"PURCHASEORDER","SYS_NC0003400035$"))
```

Example 19-12   Creating an Index for Direct Access to an Ordered Collection Table

```
CREATE INDEX lineitem_part_index ON lineitem_table l (l.part.part_number,
                                                      l.NESTED_TABLE_ID);
```

Guideline: Use XMLOptimizationCheck to determine why a query is not rewritten

If a query has not been optimized, you can use system variable `XMLOptimizationCheck` to try to determine why.

Related Topics

- Diagnosis of XQuery Optimization: XMLOptimizationCheck
  You can examine an execution plan for your SQL code to determine whether XQuery optimization occurs or the plan is instead suboptimal.