Oracle XML DB: Best Practices to Get Optimal Performance out of XML Queries

Releases 19c and 21c, Cloud and On-Premise

February 2022

Table of contents

Introduction 4

SQL/XML & XQuery 5

Migrating from Oracle Proprietary (XPath 1.0 based) syntax to Standard SQL/XML XQuery based syntax 8

Getting the best performance out of XQuery 11

Storage independent Best Practices 12

XQuery Guideline 1: Use XMLExists() and XMLQuery() to search and transform XML stored in XML DB 12

XQuery Guideline 2: Use XMLExists() to search the XML document to modify via XML DML operators 12

XQuery Guideline 3: Use XMLTable construct to query XML with relational access 13

XQuery Guideline 4: Use XMLCast() and XMLTable() constructs for GROUP BY and ORDER BY 14

XQuery Guideline 5: Use XQuery extension expression to indicate functional evaluation of XQuery 15

XQuery Guideline 6: Use XQuery in PL/SQL to manipulate PL/SQL XMLType Variable 16

XQuery Guideline 7: Use proper XQuery and SQL Typing 17

XQuery Guideline 8: XQuery expressions that are not optimizable with XML index 19

XQuery Guideline 9: Use the right XQuery expression to access data within Top XQuery 20

XQuery Guideline 10: Gather statistics 22

XQuery Guideline 11: Use SET XMLOPT[IMIZATIONCHECK] or events to determine why a query/DML is not rewritten 23

XQuery Guideline 12: Properly release resources for xmltype in client program 24

XQuery Guideline 13: Avoid calling getObject mutilple times for xmltype in client program 24

Storage dependent performance tuning 25

Binary XML 26

Binary XML Streaming Evaluation 26

Indexing Binary XML 29

XMLIndex Structured Component 32

Structured Index Guideline 1: Use Structured Index instead of multiple functional indexes and/or virtual columns 33

Structured Index Guideline 2: Make Index and Query datatypes correspond 33

Structured Index Guideline 3: Use XMLTable views with corresponding index, e.g BI style queries 34

Structured Index Guideline 4: Create Secondary Indexes, especially for predicates 35

Structured Index Guideline 5: Check the execution plan to see if structured index is used 36

Structured Index Guideline 6: Indexing Master-Detail relationships 37

Structured Index Guideline 7: Split fragement extraction and value search between SELECT and WHERE clause 38

Structured Index Guideline 8: For ordering query results, use SQL ORDER BY along with XMLTable 40

Text Index 41

Searching XML data using contains() 42

Searching XML data using ora:contains() 43

Conclusion 44

Appendix A: Semantic differences between the deprecated mainly XPath 1.0 based functions and standard SQL/XML XQuery based functions 45

Introduction

Oracle XML DB support for the XQuery language is provided through native implementation of SQL/XML functions XMLQuery, XMLTable, XMLExists, and XMLCast. A SQL statement that includes XMLQuery, XMLTable, XMLExists, or XMLCast is compiled and optimized as a whole, leveraging both relational database and XQuery-specific optimization technologies.

The XQuery optimizations can be divided into 2 broad areas:

- **Logical optimizations** are transformation of the XQuery into equivalent SQL query blocks extended with XML operators modeling XQuery semantics. These optimizations are generic XQuery optimizations that are independent of the XML storage or indexing model .
- **Physical optimizations** are transformation of the XML operators, in particular, XPath operators, into equivalent operations directly on the underlying internal storage and index tables that are specific to the XML storage and indexing model. The result of XQuery optimization can be exmained via explain plan of the SQL/XML query statement that invokes XQuery.

This paper talks about the XQuery Best Practices to get the best performance. It talks about both logical and physical query optimizations. It delves deep into various XML storage and indexing options, and talks about how to choose the right indexes for your query, and how to get the best performance out of your XQuery.

SQL/XML & XQuery

Oracle XML DB supports the latest version of the XQuery language specification, i.e., the W3C XQuery 1.0 Recommendation. XQuery 1.0 is the W3C language designed for querying XML data. It is similar to SQL in many ways, but just as SQL is designed for querying structured, relational data, XQuery is designed especially for querying semi-structured, XML data from a variety of data sources. You can use XQuery to query XML data wherever it is found, whether it is stored in database tables, available through Web Services, or otherwise created on the fly. For more information on XQuery 1.0, please see http://www.w3.org/TR/xquery/

In addition to XQuery language from W3C, SQL standard has defined standard SQL/XML functions XMLQuery(), XMLExists(), XMLCast() and table construct XMLTable() as a general interface between the SQL and XQuery languages. As is the case for the other SQL/XML functions, such as XMLElement(), XMLAgg(), XMLForest(), XMLConcat(), that are used to generate XML from relational data, XMLQuery(), XMLExists(), XMLCast() functions and XMLTable() table construct let you take advantage of the power and flexibility of both SQL and XML. Using these functions, you can query and manipulate XML, construct XML data using relational data, query relational data as if it were XML, and construct relational data from XML data.

Although SQL/XML functions XMLQuery(), XMLExists(), XMLCast() and XMLTable() construct all evaluate an XQuery expression over XMLType input, the way the result of XQuery is consumed varies among them. Therefore, they should be used in the different clauses of SQL to achieve the best performance. In the XQuery language, an expression always returns a sequence of items. The way the sequence of items is consumed in different SQL contexts is classified as below, with the proper usage of these SQL/XML functions and XMLTable table construct.

- To consume all the items in the result sequence as a single XML document or fragment, **XMLQuery()** is used as a functional expression, typically in the select list of SELECT clause of SQL, to aggregate the result sequence as one XMLType value representing an XML document or fragment. For example, the query below passes an XMLType column, oe.warehouse_spec, as context item to XQuery, using function XMLQuery with the PASSING clause. It constructs a Details element for each of the warehouses whose area is greater than 80,000: /Warehouse/ Area > 80000.

  Example 1: Using XMLQuery with PASSING Clause

```
SELECT warehouse_name,
       XMLQuery(
         'for $i in /Warehouse
          where  $i/Area > 80000
          return <Details>
                   <Docks num="{$i/Docks}"/>
                   <Rail>{if ($i/RailAccess = "Y")
                          then "true" else "false"}
                   </Rail>
                 </Details>'
         PASSING warehouse_spec RETURNING CONTENT) big_warehouses
  FROM warehouses;
```

- **XMLTable()** construct is used in the FROM clause of SQL to return evaluation result of XQuery as a table of rows, each of the XQuery item in the result sequence as an XMLType value. Users can generate a relational view over XML data using XMLTable. This is illustrated below:

  Example 2: Using XMLTable to generate a relational view over XML data.

```
SELECT lines.lineitem, lines.description, lines.partid,
       lines.unitprice, lines.quantity
```

```
FROM purchaseorder,
      XMLTable('for $i in /PurchaseOrder/LineItems/LineItem
                where $i/@ItemNumber >= 8
                 and $i/Part/@UnitPrice > 50
                 and $i/Part/@Quantity > 2
                return $i'
               PASSING OBJECT_VALUE
            COLUMNS lineitem    NUMBER       PATH '@ItemNumber',
                    description VARCHAR2(30) PATH 'Description',
                    partid      NUMBER       PATH 'Part/@Id',
                    unitprice   NUMBER       PATH 'Part/@UnitPrice',
                     quantity    NUMBER       PATH 'Part/@Quantity') lines;
```

- To determine if XQuery results in empty sequence or not, **XMLExists()**, which has a Boolean result, is typically used in the WHERE or HAVING clause of SQL or conditional expression of SQL CASE expression. The example below shows how XMLExists() can be used in the select list.

Example 3: Using XMLExists() with CASE Expression in select list

```
SELECT
      CASE WHEN XMLEXISTS('$po/PurchaseOrder/LineItems/Part'
      PASSING OBJECT_VALUE AS "po") THEN 1 ELSE 0 END
FROM purchaseorder,
```

- To cast sequence result, typically the leaf value of an XML node, as a SQL scalar type, such as NUMBER, VARCHAR, DATE, TIMESTAMP etc, **XMLCast()** is used as a functional expression resulting in a SQL scalar value item that is used in select list of SELECT clause, group by list of GROUP BY clause, or order by list of ORDER BY clause.

When XQuery is used in SQL/XML functions and XMLTable construct to query XMLType value from tables or views, Oracle XML DB compiles the XQuery expressions into a set of SQL query blocks and operators, and optimizes them by leveraging the underlying XML storage and indexes. This native XQuery/SQL/XML optimization model is achieved conceptually by using a 2-step process: logical optimization and physical optimization.

1. **Logical optimizations** are independent of the XML storage or indexing over the underlying XMLType value. The XQuery expressions that are passed as arguments to SQL/XML functions XMLQuery, XMLExists, XMLCast and XMLTable construct are compiled into internal SQL query blocks and operator trees that model the semantics of XQuery. One common internal operator is the XPath operator that navigates the input XMLType value. A SQL statement that includes XMLQuery, XMLTable, XMLExists, or XMLCast is compiled and optimized as a whole, leveraging both relational database and XQuery-specific optimization technologies.

2. **Physical optimizations** are specific to the underlying storage and indexing model. Depending on the XML storage and indexing methods used, the XPath internal operators can be further optimized into SQL query blocks operating on the underlying physical relational storage tables that are used for the underlying XML index or storage. The relational optimizer optimizes the resulting SQL query blocks and operator trees, in order to achieve the best execution plan.

The resulting query plan is then executed using the SQL row source iterator model. This native XQuery/SQL/XML optimization model achieves the performance goal of primarily using XQuery as a query language to search XML documents stored in the database with the proper XML storage and indexing model, or to present XML as relational results using XMLTable construct. Just as tuning a SQL query using 'explain plan' is important, understanding and tuning SQL/XML query using 'explain plan' is equally important. This is detailed in the subsequent sections of this document with different XML storage and index options.

Furthermore, XQuery can also be primarily used as a language to manipulate and transform XML documents. The input XMLType value is typically a single XML document or fragment retrieved from persistent XML or transient XMLType value. In such case, XQuery can be functionally evaluated in XML DB. Understanding and classifying XQuery usage in XML DB is critical to get the optimal performance. This is detailed later in this document in the section 'Getting the best performance out of XQuery.'

Migrating from Oracle Proprietary (XPath 1.0 based) syntax to Standard SQL/XML XQuery based syntax

Starting 11gR2, Oracle has deprecated many older proprietary mainly XPath 1.0 based operators in favor of standards based XQuery syntax, as listed in Table 1 below. If you don't have any code with the functions or operators being deprecated, you may jump to the next section.

TABLE 1. MIGRATING FROM OLD TO XQUERY SYNTAX

OLD ORACLE PROPRIETARY SYNTAX	NEW XQUERY SQL/XML BASED SYNTAX
extract()	XMLQuery()
extractValue	XMLCast(XMLQuery())
existsNode()	XMLExists()
Table (XMLSequence)	XMLTable
ora:instanceof	instanceof
ora:instanceof-only	@xsi:type
getNamespace	fn:namespace-uri
getRootElement	fn:local-name
getStringVal, getBlobVal, getClobVal	XMLSerialize
Xmltype()	XMLParse() for varchar, clob, blob input
DBMS_XMLQUERY	XMLQuery()
DBMS_XMLGEN	SQL/XML Operators
Oracle XML DML Operators	XQUery Update Facility

There are some important semantic differences between the deprecated mainly XPath 1.0 based sytnax and the XQuery based syntax. These are listed in Appendix A to make the migration easier for the users. Please also check "XQuery Guideline 6" in this document to see how to apply XQuery to PL/SQL XMLType variable instead of calling extract() and existsNode() methods of xmltype.

The table below shows examples of Oracle Proprietary XML DML operators and their equivalent Xquery Update syntax:

Note: Oracle Proprietary XMLDML does not have "rename" and "insert as first into" operations.

`Update warehouses set warehouse_spec =` **`appendChildXML`**`(warehouse_spec, 'Warehouse/Building', XMLType('<Owner>Grandco</Owner>'));`	`Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify (for $i in $tmp/Warehouse/Building return insert node <Owner>Grandco</Owner>  as last into $i) return $tmp' passing warehouse_spec returning content);`
`Update warehouses set warehouse_spec =` **`deleteXML`**`(value(po), '/Warehouse/Building');`	`Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify delete node $tmp/Warehouse/Building return $tmp' passing warehouse_spec returning content);`
`[Single Node Case]` `Update warehouses set warehouse_spec =` **`insertXML`**`(warehouse_spec, '/Warehouse/Building/Owner[2]', XMLType('<Owner>ThirdOwner</Owner>');`	`Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify insert node <Owner>ThirdOwner</Owner> into $tmp/Warehouse/Building/Owner[2] return $tmp' passing warehouse_spec returning content);`
`[Single Node Case]` `Update warehouses set warehouse_spec =` **`insertXMLBefore`**`(warehouse_spec, '/Warehouse/Building/Owner[2]', XMLType('<Owner>FirstOwner</Owner>');`	`Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify insert node <Owner>FirstOwner</Owner> before $tmp/Warehouse/Building/Owner[2] return $tmp' passing warehouse_spec returning content);`
`[Single Node Case]` `Update warehouses set warehouse_spec =` **`insertXMLAfter`**`(warehouse_spec, '/Warehouse/Building/Owner[2]', XMLType('<Owner>ThirdOwner</Owner>');`	`Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify insert node <Owner>ThirdOwner</Owner> after $tmp/Warehouse/Building/Owner[2] return $tmp' passing warehouse_spec returning content);`
`Update warehouses set warehouse_spec =` **`updateXML`**`(warehouse_spec, '/Warehouse/Docks/text()', 4);`	`Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify (for $i in $tmp/Warehouse/Docks/text()  return replace value of node $i with 4) return $tmp' passing warehouse_spec returning content);`
`Update warehouses set warehouse_spec =` **`insertChildXML`**`(warehouse_spec, '/Warehouse/Building', 'Owner', XMLType('<Owner>LesserCo</Owner>'));`	`Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify (for $i in $tmp/Warehouse/Building return insert node <Owner>LesserCo</Owner> into $i) return $tmp' passing warehouse_spec returning content);`
`Update warehouses set warehouse_spec =` **`insertChildXMLBefore`**`(warehouse_spec, '/Warehouse/Building', 'Owner', XMLType('<Owner>LesserCo</Owner>'));`	`Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify (for $i in $tmp/Warehouse/Building return insert node <Owner>LesserCo</Owner> before $i) return $tmp' passing warehouse_spec returning content);`
`Update warehouses set warehouse_spec =` **`insertChildXMLAfter`**`(warehouse_spec, '/Warehouse/Building', 'Owner', XMLType('<Owner>LesserCo</Owner>'));`	`Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify (for $i in $tmp/Warehouse/Building  return insert node <Owner>LesserCo</Owner> after $i) return $tmp' passing warehouse_spec returning content);`

[Collection Case] Update warehouses set warehouse_spec = **insertXML**(warehouse_spec, '/Warehouse/Building/Owner', XMLType('<Owner>AnotherOwner</Owner>');	Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify (for $i in $tmp/Warehouse/Building/Owner return insert node <Owner>AnotherOwner</Owner> into $i) return $tmp' passing warehouse_spec returning content);
[NULL Case] Update warehouses set warehouse_spec = **updateXML**(warehouse_spec, '/Warehouse/Docks', null);	Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify delete node $tmp/Warehouse/Docks return $tmp' passing warehouse_spec returning content);
[Empty Node Case] Update warehouses set warehouse_spec = **updateXML**(warehouse_spec, '/Warehouse/Docks', '');	Update warehouses set warehouse_spec = XMLQuery('copy $tmp := $p1 modify (for $j in $tmp/Warehouse/Docks return replace node $j with $p2) return $i' passing passing warehouse_spec "p1", '' as "p2" returning content) ;
[Multiple Path Case] Update warehouses set warehouse_spec = **updateXML**(warehouse_spec, '/Warehouse/Docks/text()', extractValue(warehouse_spec, '/Warehouse/Docks/text()')+4, '/Warehouse/Docks/text()', extractValue(warehouse_spec, '/Warehouse/Docks/text()')+4);	Update warehouses set warehouse_spec = XMLQuery('copy $tmp := . modify ((for $i in $tmp/Warehouse/Docks/text() return replace value of node $i with $i+4), (for $i in $tmp/Warehouse/Docks/text() return replace value of node $i with $i+4)) return $tmp' passing warehouse_spec returning content);

Getting the best performance out of XQuery

XQuery Best Practices and Performance Tuning can be divided into 2 parts:

- Best practices independent of the XMLType storage options. These are listed in the "Storage independent best practices" section.
- Best practices specific to the XMLType storage selected by the user. These include various indexes the user can create to speed up their XQueries. These are listed in the "Storage dependent performance tuning" section.

Storage independent Best Practices

In Oracle XML DB, XML documents are stored in either XMLType tables or XMLType columns of relational tables. XML DB is designed to store large number of XML documents, and to search using XQuery among these XML documents, in order to find qualified XML documents or document fragments for manipulation and transformation using XQuery, or to project relational views over XML using XMLTable construct so that they can be queried relationally and be integrated with mature relational applications.

XQuery Guideline 1: Use XMLExists() and XMLQuery() to search and transform XML stored in XML DB

The typical way of writing a SQL statement that searches XML documents stored in XMLType column and manipulates the searched result is stated below:

Example 4: Search and transform

```
SELECT XMLQUERY('…' PASSING T.X RETURNING CONTENT)
FROM purchaseorder T
WHERE XMLEXISTS('$p/PurchaseOrder/LineItems/LineItem/Part[@Id="702372"]'
          PASSING T.X AS "p");
```

In this SQL statement, XMLExists() is used in WHERE clause of the statement to accomplish the typical database task of "finding needle in a haystack." Since there can be billions of XML document stored in table purchaseorder, using proper index, instead of a table scan with functional evaluation of XQuery used in XMLExists() for each XML document, is critical to achieve query performance. To achieve the best performance, the XQuery used in XMLExists() should be index friendly.

If XQuery used in XMLExists() is not index friendly as a whole, then try to break the XQuery into index-friendly expressions and index-unfriendly expressions and use them in two different XMLExists() functions connected by the SQL AND construct. In this way, at least the index-friendly XMLExists() can be evaluated using index and the index-friendly XMLExists() can be evaluated as a post-index filter.

XQuery Guideline 2: Use XMLExists() to search the XML document to modify via XML DML operators

The typical way of writing a SQL statement that searches for and modifies XML documents stored in XMLType column is shown below.

Example 5: Updating XML document after searching using XMLExists()

```
UPDATE purchaseorder T SET T.X = DELETEXML(T.X,
                    '/purchaseOrder/LineItems/LineItem[itemName ="TV"]' )
WHERE XMLEXISTS('$p/PurchaseOrder/LineItems/LineItem/Part[@Id="717951002372"]'
                  PASSING T.X AS "p");
```

As in XQuery Guideline 1, XMLExists() is used here to identify which XML documents is to be modified, i.e., "finding needle in a haystack." The function used in the RHS of the UPDATE assignment can be any expression that returns XMLType. For example, it can be a PL/SQL function call that returns XMLType. Semantically, the RHS expression of the SQL UPDATE statement returns an XMLType instance document that is assigned to XMLType column on the LHS to do document replacement of the whole XMLType column value.

However, Oracle XML DB does XML DML operator rewrite optimization whenever possible, so as to partially update the underlying XML storage structures instead of replacing the whole document. For binary XML storage, there is XML DML operator rewrite for all XML DML operators when the XPath can be evaluated using streaming evaluation.

XML DML operator rewrite can be explicitly disabled by using the /*+NO_XML_DML_REWRITE */ SQL hint. This is true regardless of XML storage model.

XQuery Guideline 3: Use XMLTable construct to query XML with relational access

XML document is hierarchical in nature and has typical master-detail relationships. Therefore, it is common to project out master-detail constructs within XML document as a set of relational tables using XMLTable construct and project out leaf values of each construct as columns of XMLTable for search, as shown in the example below:

Example 6: Using XMLTable

```
SELECT li.description, li.lineitemFROM purchaseorder T,

         XMLTable('$p/PurchaseOrder/LineItems/LineItem'
      PASSING T.X AS "p"
      COLUMNS lineitem    NUMBER        PATH '@ItemNumber',
             description  VARCHAR2(30)  PATH 'Description',
             partid       NUMBER        PATH 'Part/@Id',
             unitprice    NUMBER        PATH 'Part/@UnitPrice',
             quantity     NUMBER        PATH 'Part/@Quantity') li
    WHERE li.unitprice > 30 and li.quantity < 20);
```

To process the XMLTable() construct efficiently, XQuery usage in XMLTable clause should be storage or index friendly so that native XQuery/SQL/XML optimization can find the best query plan leveraging the underlying XML storage and index models. If purchaseorder column is stored using binary XML, the underlying relational tables belonging to the XMLIndex are directly accessed in the resulting query plan.

To traverse multi-level hierarchy, XMLTable can be used in a chaining fashion.

XQuery Guideline 4: Use XMLCast() and XMLTable() constructs for GROUP BY and ORDER BY

There are GROUP BY and ORDER BY clauses that operate on SQL scalar types. One typical way of casting XQuery result into SQL scalar types for GROUP BY and ORDER BY purposes is shown in the example below.

Example 7: Using XMLCast() in GROUP BY / ORDER BY

```
SELECT XMLCAST(XMLQUERY('$p/PurchaseOrder/@poDate' PASSING T.X
                         RETURNING CONTENT) AS DATE), COUNT(*)
FROM purchaseorder T
WHERE …
GROUP BY XMLCAST(XMLQUERY('$p/PurchaseOrder/@poDate' PASSING T.X
             RETURNING CONTENT) AS DATE)
ORDER BY XMLCAST(XMLQUERY('$p/PurchaseOrder/@poDate' PASSING T.X
             RETURNING CONTENT) AS DATE);
```

When there are multiple scalar values that need to be grouped or ordered, it is better to write it with XMLTable construct that projects out all columns to be ordered or grouped as shown below.

Example 8: Using XMLTable() construct for GROUP BY / ORDER BY

```
SELECT po.DATE, po.poZip, count(*)
FROM purchaseorder T,
      XMLTable('$p/PurchaseOrder'
             PASSING T.X AS "p"
             COLUMNS
                   poDate    DATE        PATH '@poDate',
                   poZip     VARCHAR2(8)  PATH 'shipAddress/zipCode',
             ) po
WHERE ….
GROUP BY po.poDate, po.poZip
ORDER BY po.poDate, po.poZip
```

In this case, if purchaseOrder.X column uses binary XML storage with structured xmlindex, the query plan will directly use group by and order by of the columns from the underlying relational storage tables of the XML storage or xmlindex.

Note the XMLTable usage pattern in SQL/XML is very commonly adopted by users to create relational views over XML, so that XML query can be integrated with existing relational applications (such as BI applications) smoothly.

XQuery Guideline 5: Use XQuery extension expression to indicate functional evaluation of XQuery

XQuery is a language that blends both search and transformation of XML. While XQuery used for search in the WHERE clause is more amenable for XQuery rewrite optimization leveraging the underlying XML storage and indexing models, XQuery used for transformation in the SELECT clause might be more procedure-centric and hence suited for functional evaluation. You can use XQuery extension expression (#ora:xq_proc #) to indicate that the XQuery should be functionally evaluated, as shown in the example below.

Example 9: XQuery extension expression for functional evaluation

```
SELECT XMLQUERY('(#ora:xq_proc #){…}' PASSING T.X RETURNING CONTENT)
FROM purchaseorder T
WHERE  XMLEXISTS('$p/PurchaseOrder/LineItems/LineItem/Part[@Id="717951"]'
                 PASSING T.X AS "p");
```

The (#ora:xq_proc#){…} is an XQuery extension expression serving as a "pragma" to indicate the xquery expression enclosed in the curly braces needs to be evaluated functionally. It is available since Oracle 11gR2, release 11.2.0.2.

This mechanism is more fine-grained and hence more flexible than using /*+ NO_XML_QUERY_REWRITE */ SQL hint , which requests all XQuery used in a SQL statement to use functional evaluation. This may not be desirable for XQuery used in XMLExists() of the SQL statement.

XQuery Guideline 6: Use XQuery in PL/SQL to manipulate PL/SQL XMLType Variable

PL/SQL XMLType methods do not support XQuery invocation directly. However, one can invoke SQL/XML functions with XQuery to query on XMLType PL/SQL variables as shown in the following example. Since a PL/SQL XMLType variable value is not indexed, /*+ NO_XML_QUERY_REWRITE*/ SQL hint is used to evaluate XQuery functionally.

Example 10: Querying PL/SQL XMLType variable using XMLQuery() and XMLCast()

```
DECLARE
      v_x  XMLType;
      NumAcc NUMBER;
BEGIN
       v_x := XMLType(xmlfile(…)); /* initialize xmltype variable */
      SELECT /*+ NO_XML_QUERY_REWRITE */
          XMLCAST(XMLQUERY('declare default element namespace
                    "http://custacc";for $cust in $cadoc/Customer return
                    fn:count($cust/Addresses/Address)'
           PASSING v_x AS "cadoc" RETURNING CONTENT) AS NUMBER)
      INTO NumAcc
      FROM DUAL;
END;
```

Example 11: Querying PL/SQL XMLType variable using XMLExists()

```
DECLARE
      v_x  XMLType;
      ex NUMBER;
BEGIN
       v_x := XMLType(xmlfile(…)); /* initialize xmltype variable */
      SELECT /*+ NO_XML_QUERY_REWRITE */
          CASE WHEN XMLEXISTS('declare default element namespace
                    "http://custacc"; $cadoc/Customer/Addresses/Address)'
           PASSING v_x AS "cadoc")
           THEN 1 ELSE 0 END
      INTO ex
      FROM DUAL;
END;
```

XQuery Guideline 7: Use proper XQuery and SQL Typing

XQuery type system is based on XML Schema type system. Although XQuery type system and SQL type system are not exactly aligned, there are equivalent mappings between the types in each system, as shown in the table below. Note that xs:date, xs:time, xs:dateTime have optional timezone component, therefore, they are mapped to 'TIMESTAMP WITH TIMEZONE' SQL type. When the timezone component is not used, then you may map to DATE or TIMESTAMP SQL types.

TABLE 2. XML AND SQL DATA TYPE CORRESPONDENCE FOR XMLINDEX

XML DATA TYPE	SQL DATA TYPE
xs:integer, xs:decimal	INTEGER or NUMBER
xs:double	BINARY_DOUBLE
xs:float	BINARY_FLOAT
xs:date	DATE, TIMESTAMP WITH TIMEZONE
xs:time, xs:dateTime	TIMESTAMP, TIMESTAMP WITH TIMEZONE
xs:dayTimeDuration	INTERVAL DAY TO SECOND
xs:yearMonthDuration	INTERVAL YEAR TO MONTH

Users are recommended to cast these types properly in XQuery used within XMLExists() clause to ensure proper type-aware comparison semantics and proper XML index usage. This is illustrated in the examples below:

Example 12: Using XQuery type casting and SQL type cast to pass in the properly typed value into XMLExists()

```
SELECT … FROM purchaseOrder T
WHERE XMLEXISTS('$po/purchaseOrder[@id=$id]'
                              PASSING T.X AS "po", CAST(:1 AS NUMBER) as "id" );
```

In this example, we explicitly cast SQL bind variable :1 as SQL NUMBER type and bind that to XQuery external variable "$id" of XMLExists() operator.

If the purchaseOrder document is non-XML schema based, then @id is of type xs:untypedAtomic. The general comparison rule in XQuery states that comparing xs:untypedAtomic value with any numeric type value (xs:integer, xs:decimal, xs:float, xs:double) is done by promoting both operands to xs:double. This makes the @id comparision in XQuery use xs:double() comparison even though SQL bind variable is passed as xs:decimal typed value, it is internally casted into xs:double typed value.

On the other hand, if the purchaseOrder document is XML schema based, then @id is not of type xs:untypedAtomic, instead it is of type stated by the XML Schema. If the XML schema states that the @id is of type xs:decimal, for example, then this makes the @id comparison in XQuery use xs:decimal() comparison and the SQL bind variable passed as xs:decimal typed value no longer needs to be internally casted into xs:double typed value.

Keeping in mind that xs:decimal is for exact numeric type and xs:double is for approximate numeric type, application users need to decide what typed comparison the application needs. Once the decision is made, then write the "Example 12: Using XQuery type casting and SQL type cast to pass in the properly typed value into XMLExists()" query above as "Example 13: using xs:decimal() type exact numeric comparison" or "Example 14: Using xs:double() type approximate numeric comparison" using explicit XQuery type casting to get either xs:decimal() typed comparison or xs:double() typed comparison.

Example 13: using xs:decimal() type exact numeric comparison

```
SELECT … FROM purchaseOrder T
WHERE XMLEXISTS('$po/purchaseOrder[xs:decimal(@id)=$id]'
                       PASSING T.X AS "po", CAST(:1 AS NUMBER) as "id" );
```

Example 14: Using xs:double() type approximate numeric comparison

```
SELECT … FROM purchaseOrder T
WHERE XMLEXISTS('$po/purchaseOrder[xs:double(@id)=$id]'
                       PASSING T.X AS "po", CAST(:1 AS BINARY_DOUBLE) as "id" );
```

Using explicit type casting is required to ensure that XQuery will use proper typed value comparison independent of whether XMLType document stored in the table is XML schema based or not. Furthermore, doing so promotes the usage of XMLindex.

To make "Example 13: using xs:decimal() type exact numeric comparison" use structured XMLIndex, "/purchaseOrder/@id" must be indexed as SQL NUMBER type.

To make "Example 14: Using xs:double() type approximate numeric comparison" use structured XMLIndex, "/purchaseOrder/@id" must be indexed as SQL TO_BINARY_DOUBLE type.

For non-numeric datatypes, XQuery general comparison allows xs:untypedAtomic typed value to be cast into the type of the other value, so we just need to apply XQuery type casting on the passing parameter as shown in the 2 examples below for xs:date() and xs:dateTime() comparison.

Example 15: Using xs:date() for date datatype comparison

```
SELECT … FROM purchaseOrder T
WHERE XMLEXISTS('$po/purchaseOrder[@podate =xs:date($d)]'
                       PASSING T.X AS "po", :1 as "d" );
```

Here, :1 is expected to bind to SQL varchar of value, say '2008-07-08' .

Example 16: Using xs:dateTime() for timestamp with timezone datatype comparison

```
SELECT … FROM purchaseOrder T
WHERE XMLEXISTS('$po/purchaseOrder[@podate =xs:dateTime($d)]'

                       PASSING T.X AS "po", :1 as "d" );
```

Here, :1 is expected to bind to SQL varchar of value, say "2010-01-01T12:00:00Z' .

XQuery Guideline 8: XQuery expressions that are not optimizable with XML index

Some expressions might add performance overhead when processing large-size XML document, because these expressions typically cannot leverage the underlying XML storage or index structures. Such expressions should be avoided when querying very large XML documents. They are listed in Table 3:

TABLE 3. EXPRESSIONS TO AVOID FOR LARGE DOCUMENTS

EXPRESSIONS TO AVOID
Avoid XQuery expressions that use the following XPath step axes: • ancestor • ancestor-or-self • descendant-or-self • following • following-sibling • parent • preceding • preceding-sibling
Avoid <<, >> expressions.

XQuery Guideline 9: Use the right XQuery expression to access data within Top XQuery

Pure XQuery users prefer to write XQuery without using individual SQL/XML operators. Oracle XML DB supports this type of usage by enabling users to wrap the entire XQuery into one SQL SELECT statement using either

```
SELECT * FROM XMLTABLE('…') ;
```

or

```
SELECT XMLQuery('…') FROM DUAL;
```

depending on whether the XQuery results are consumed as a sequence or as one XML fragment. This is referred as "Top XQuery" because SQL is used here purely as a wrapping mechanism.

Prior to 11gR2 11.2.0.2 release, functions fn:collection() and fn:doc() needed to be replaced with ora:view(). In 11gR2 11.2.0.2 release, fn:collection() or fn:doc() can be used to uniformly refer to XML documents that are stored in XMLType tables, XMLType columns, or generated virtually from pure relational tables. However, you need to use the proper oradb-prefixed URL or XQuery extension expression. Examples are shown below.

Top XQuery statement goes through the same XQuery rewrite optimizations as that of regular SQL/XML statements. Just as users do performance tuning using explain plan for SQL statements, users should use explain plan to do performance tuning for Top XQuery statement as well.

- Use ora:view() to map relational table content as a collection of virtual XML documents

  ```
  SELECT *
  FROM XMLTABLE(
    'for $i in ora:view("SCOTT", "EMP")
     where $i/ROW[EMPNO = 7369 and HIREDATE=xs:date("1980-12-17")]
       return $i' );
  ```

  Here EMP is a relational table owned by user "SCOTT".

  In 11gR2 11.2.0.2 release or later, you may also use fn:collection() as shown below:

  ```
  SELECT * FROM XMLTABLE(
          'for $i in fn:collection("oradb:/SCOTT/EMP")
          where $i/ROW[EMPNO = 7369 and HIREDATE=xs:date("1980-12-17")]
        return $i' );
  ```

- Use ora:view() to map XMLType table content as a collection of XML documents

  ```
  SELECT * FROM XMLTABLE(
         'for $i in ora:view("PO", "PURCHASEORDER")
         where $i/PurchaseOrder/Id = xs:decimal(789645)
         return $i/PurchaseOrder/LineItems/LineItem[itemName="TV"]')
  ```

  Here, PURCHASEORDER is an XMLType table owned by user PO.

  In 11gR2 11.2.0.2 release, you may also use fn:collection() as shown below:

  ```
  SELECT * FROM XMLTABLE(
         'for $i in fn:collection("oradb:/PO/PURCHASEORDER")
         where $i/PurchaseOrder/Id = xs:decimal(789645)
         return $i/PurchaseOrder/LineItems/LineItem[itemName="TV"]')
  ```

- Use fn:collection() to map XMLType column of a table as a collection of XML documents

  Here, PURCHASEORDER is a relational table owned by user PO and has an XMLType column 'X'. This is available starting 11gR2 11.2.0.2 release.

```
SELECT * FROM XMLTABLE(
       'for $i in fn:collection("oradb:/PO/PURCHASEORDER/ROW/X")
       where $i/PurchaseOrder/Id = xs:decimal(789645)
       return $i/PurchaseOrder/LineItems/LineItem[itemName="TV"]')
```

- To avoid passing hard-coded search values as constants to Top-XQuery, users may use PASSING bind variable parameters as shown the example below:

  Example 17: Passing Bind Variables

```
SELECT * FROM XMLTABLE(
           'for $i in fn:collection("oradb:/SCOTT/EMP")
            where $i/ROW[EMPNO = xs:decimal($empno)]
          return $i'
   PASSING :1 as "empno")
```

XQuery Guideline 10: Gather statistics

One common problem is that user forgets to gather stats on his tables. Inaccurate stats can result in a bad execution plan. Hence it is recommended to periodically perform gather statistics on the XMLType table and relevant indexes, as listed below.

In a use case where data is loaded once and queried several times, running dbms_stats.gather_table_stats() on the affected tables (as outlined below), after data has been loaded, is sufficient. In a use case where data is loaded or updated quite frequently, running dbms_stats.gather_schema_stats() or dbms_stats.gather_table_stats (as outlined below) as a background scheduler job (package dbms_scheduler) is the best. Note that the default behavior of gather_table_stats is to propagate gathering of stats to all indexes on the table.:

- For XMLIndex, gathering stats on base table will automatically gather stats on the Structured XMLIndex tables. Hence, there is no need to gather stats on the XMLIndex separately.
- For Text Index, gathering stats on base table will automatically gather stats on the Text Index tables. Hence, there is no need to gather stats on Text Index separately.

Starting Oracle 11.2.0.3, if there are xml indexes present that use binary-double as secondary
indexes, it is recommended to set optimizer_dynamic_sampling to 3 for picking up proper secondary indexes. For example, the following 2-command script can be used to gather statistics on the schema:

```
alter session set optimizer_dynamic_sampling = 3;

exec dbms_stats.gather_schema_stats('USERNAME');
```

XQuery Guideline 11: Use SET XMLOPT[IMIZATIONCHECK] or events to determine why a query/DML is not rewritten

Just as query tuning can improve SQL performance, so it can improve XQuery performance. You tune XQuery performance by choosing appropriate indexes for your XML Storage. As with database queries generally, you can examine the execution plan for a query to determine whether tuning is required.

In general, use explain plan on your SQL statement (including Top XQuery wrapped in SQL statement) to understand and tune query performance. In particular, when there is 'COLLECTION ITERATOR' appearing in the explain plan, it usually indicate the query plan is not fully optimized.

Advanced users can use:

- XMLOPT[IMIZATIONCHECK] [ON|OFF]" mechanism (in Oracle 11gR2 release 11.2.0.2), or event 19021 with level 4096 (0x1000)(in releases prior to 11.2.0.2) to get the optimized rewritten query in the trace file to see what underlying queries are executed on the underlying internal tables created for XML storage and index models.
- Event 19027 with level 8192 (0x2000) to get a dump in the trace file indicating why a particular expression is not rewritten.

In Oracle 11gR2, release 11.2.0.2, or later:

In Oracle 11gR2 11.2.0.2 release or later, we recommend that you use the "SET XMLOPT[IMIZATIONCHECK] [ON|OFF]" mechanism to determine if parts of your query were not optimized. When it is ON, it will ensure that only XML queries or XML operations that were fully optimized will be executed. A suboptimal XML query or DML operation will be aborted with the following error message: "ORA 19022 - Unoptimized XML construct detected". In addition, the reason for the query or DML being suboptimal will be printed to the trace file. OFF will not guarantee that only fully optimized XML queries/ DML operations will be executed. The default option for this command is OFF. Please use XMLOPT[IMIZATIONCHECK] ON only when developing or debugging a query/ DML operation for performance tuning.

In releases prior to Oracle 11gR2 11.2.0.2 :

If you are on a release prior to 11gR2 11.2.0.2 release, you may set event 19021 with level 1 for a given database session using SQL statement ALTER SESSION to determine if your XML operation was rewritten. Turn on event 19021 with level 1 if you want to raise an error whenever any of the XML functions is not rewritten and is instead evaluated functionally. The error "ORA-19022 - XML XPath functions are disabled" is raised when such functions execute.

XQuery Guideline 12: Properly release resources for xmltype in client program

When XMLType result is fetched in JDBC program, please make sure to call close() method on XMLType result once it is consumed to free resources allocated by the server to track the XMLType results. The following JDBC code fragment demonstrates the call of close() method on XMLType result.

Example 18: Using the close() method to free the resources in JDBC

```
XMLType xml2;
while (rset.next())
{
    xml2 = XMLType.createXML(rset.getOPAQUE(1));
    System.out.println("Result: " + xml2.getStringVal());
    xml2.close(); // free the XMLType result tracked by the Server
}
rset.close();
```

XQuery Guideline 13: Avoid calling getObject mutilple times for xmltype in client program

In JDBC program, please avoid calling getObject() multiple times. Because XMLType object is ref counted, every call to getObject() will increase ref count by one. The call to close() method of XMLType will free the object when the ref count is 1.

Example 19: Avoud calling getObject() twice

Instead of doing this:

```
Object res = rset.getObject(j);

 if( res instanceof XMLType)
 {
      xml = (XMLType)rset.getObject(j);
 }
```

We shall do

```
Object res = rset.getObject(j);

 if( res instanceof XMLType)
 {
      xml = (XMLType)res;
 }
```

Storage dependent performance tuning

Recall that Oracle XML DB performs logic rewrite optimization followed by physical rewrite optimization based on XML storage and index by evaluating the XPath expression against the XML document without ever constructing the XML document in memory. This optimization is called XPath rewrite optimization. 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 includes XMLIndex optimizations, streaming evaluation of binary XML, and rewrite to underlying object-relational or relational structures in the case of XMLType views over relational data.

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

- When an XMLType view is built on relational data.
- When you use an XMLIndex index.
- When XMLType data is stored as binary XML – using streaming evaluation.

All of these items are discussed in the following subsections.

Binary XML

Binary XML storage is used primarily for unstructured data. The standard database indexes (B-tree, bitmap) are generally not helpful for accessing particular parts of an XML document. XMLIndex provides a general, XML-specific index that indexes the internal structure of XML data. One of its main purposes is to overcome the indexing limitation presented by binary XML storage. Sometimes when a query cannot use any index, it can still be optimized using the streaming XPath evaluation. This section provides guidance on which indexes to create, and how to write your query to use the streaming Xpath evaluation.

Binary XML Streaming Evaluation

The streaming mode of Xpath evaluation is used to efficiently evaluate the most common types of Xpaths over documents that are stored in Binary XML. This is done by first rewriting the query to collect related Xpaths together so that they can be evaluated in a single pass over the document. This type of rewrite is reflected in the output of 'explain plan' as 'XPATH EVALUATION'. For example, in the plan for the following query, the Xpaths from the SELECT list and the WHERE clause are gathered and evaluated together as columns of the 'XPATH EVALUATION' step; this is reflected in the predicate information section, which refers to the column corresponding to /PurchaseOrder/Reference.

Example 20: Xpath Evalution in Query Plan

```
SELECT XMLCAST(XMLQuery('$p/PurchaseOrder/@poDate'
                 PASSING OBJECT_VALUE AS "p" RETURNING CONTENT) as DATE)
FROM purchaseorder
WHERE  XMLExists('$p/PurchaseOrder[Reference="123456"]'
                     PASSING OBJECT_VALUE AS "p");


| Id  | Operation           | Name            | Rows  |
--------------------------------------------------------
|   0 | SELECT STATEMENT    |                 |     1 |
|   1 |  NESTED LOOPS       |                 |     1 |
|   2 |   TABLE ACCESS FULL| PURCHASEORDER |     1 |
|*  3 |   XPATH EVALUATION |                 |       |
--------------------------------------------------------


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


   3 - filter("P"."C_01$"='123456')
```

In the following query, all the columns of the XMLTable are evaluated together as part of the 'XPATH EVALUATION' step:

Example 21: Streaming Xpath Evalution of XMLTable query

```
SELECT li.description, li.lineitem
FROM
  purchaseorder T,
  XMLTable('$p/PurchaseOrder/LineItems/LineItem'
      PASSING OBJECT_VALUE AS "p"
      COLUMNS lineitem    NUMBER        PATH '@ItemNumber',
              description  VARCHAR2(30)  PATH 'Description',
              partid       NUMBER        PATH 'Part/@Id',
              unitprice    NUMBER        PATH 'Part/@UnitPrice',
              quantity     NUMBER        PATH 'Part/@Quantity') li
WHERE li.unitprice > 30 and li.quantity < 20;
```

```
| Id  | Operation            | Name            | Rows  |
|   0 | SELECT STATEMENT     |                 |     1 |
|   1 |  NESTED LOOPS        |                 |     1 |
|   2 |   TABLE ACCESS FULL| PURCHASEORDER |     1 |
|*  3 |   XPATH EVALUATION |                 |       |
Predicate Information (identified by operation id):

   3 - filter(CAST("P"."C_01$" AS NUMBER)>30 AND
              CAST("P"."C_02$" AS NUMBER)<20)
```

In general, Xpaths involving the child and descendant axes can be evaluated in this mode, but not ones involving reverse axes (like the ancestor axis). Most position-based predicates in Xpaths are evaluated in streaming mode in 11gR2. In releases prior to 11gR2, Xpaths involving position predicates cannot be evaluated in this mode. In all releases, Xpaths with predicates involving last(), as well as those with position-based and non-position-based predicates in the same step should be avoided, as these Xpaths are not evaluated in streaming mode.

Here are some guidelines on how to write queries to get the best results from streaming Xpath evaluation:

Streaming Evaluation Guideline 1: Convert reverse Xpath axes to forward axes when possible

In many cases, it is easy to convert an Xpath that uses reverse axes to an equivalent one that does not (i.e., uses forward axes only). For example, the following query uses the parent axis (the '..' step) to select nodes that have a child that is named 'a' and has an attribute id whose value is 'abc1'. It can be rewritten to an equivalent query that does not use the parent axis, by including 'a' in the predicate (rather than as a separate path step), as shown below.

Example 22: Conversion of reverse axes to forward axes

```
-- Query with reverse axis (cannot be evaluated in streaming mode)
SELECT XMLQuery('$p/PurchaseOrder/*/a[@id="abc1"]/..'
                 PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
FROM purchaseorder
WHERE XMLExists('$p/PurchaseOrder[Reference="123456"]'
                     PASSING OBJECT_VALUE AS "p");

-- Equivalent query with no reverse axes (can be evaluated in
streaming mode)
SELECT XMLQuery('$p/PurchaseOrder/*[a/@id="abc1"]'
                 PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
FROM purchaseorder
WHERE XMLExists('$p/PurchaseOrder[Reference="123456"]'
                    PASSING OBJECT_VALUE AS "p");
```

Streaming Evaluation Guideline 2: For large documents, avoid descendant axis & wild cards if exact (named) path steps can be used

Although Xpaths with descendant axis & wild cards can be evaluated in streaming mode, they are not as efficient as using just the child axis and named path steps. For example, to get all the line items in a particular purchase order that have a quantity greater than 5, use /`PurchaseOrder/LineItem` instead of //LineItem, as shown below.

Example 23: Avoiding descendant axis

```
-- Query with descendant axis
SELECT XMLQuery('$p//LineItem[@quantity > 5]'
                 PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
FROM purchaseorder
WHERE XMLExists('$p/PurchaseOrder[Reference="123456"]'
                     PASSING OBJECT_VALUE AS "p")

-- Query with named path steps (avoiding descendant axis)
SELECT XMLQuery('$p/PurchaseOrder/LineItem[@quantity > 5]'
                 PASSING OBJECT_VALUE AS "p" RETURNING CONTENT)
FROM purchaseorder
WHERE XMLExists('$p/PurchaseOrder[Reference="123456"]'
                     PASSING OBJECT_VALUE AS "p")
```

Streaming Evaluation Guideline 3: For DML-heavy workloads, enable caching for writes on the underlying lob column

Note that binary xml tables use a hidden blob column named 'xmldata' to store the encoded xml documents. For workloads that involve a significant amount of DML, enabling caching for writes on this lob column will speed up subsequent queries that use streaming evaluation on the affected documents. The following sql statement enables caching for writes on the purchaseorder table's blob column:

Example 24: Enabling caching for DMLs on binary xml tables

```
ALTER TABLE purchaseorder modify lob (xmldata) (cache);
```

Indexing Binary XML

As mentioned above, the indexing solutions in the relational world are not suitable for indexing XML, hence we have a different set of indexes for XML usecases. The different indexes supported are given below:

- XMLIndex structured component, or Table-based index. This is called the "Structured XMLIndex" for short.
- CONTEXT Index

If you are dealing with large volumes of XML data, you may want to consider taking advantage of the parallelism and partitioning features offered by Oracle. When the base XML is partitioned by range or list partitioning methods, then a corresponding XMLIndex can be created on the XML using the keyword LOCAL. When this is done, the XMLIndex is equi-partitioned with the base table – each partition of the XMLIndex has a 1-1 correspondence with a partition of the base XML. Note that XMLIndex partitioning is only supported on tables that are range or list partitioned.

You can use a PARALLEL clause (with optional degree) when creating or altering an XMLIndex index to ensure that index creation and maintenance are carried out in parallel. If the base table is partitioned or enabled for parallelism, then this can improve the performance for both DML operations (INSERT, UPDATE, DELETE) and index DDL operations (CREATE, ALTER, REBUILD). The degree-of-parallelism (DOP) value specified at the XMLIndex level is also set on each internal table of the XMLIndex.

The predicates of path expression, WHERE clause of FLWOR expression, WHERE clause of SQL/XML statement having XMLExists() or XMLTable construct are subject shown in examples below. Such predicate evaluation can be greatly speeded up by using the right XMLIndex. XMLIndex can be used to do both inter-document search (filtering XML document rows stored in the table) and intra-document search (filtering XML document fragment for XML document stored in each row of the table).

Example 25: Examples of where XMLIndex could be used

```
/* XMLExists() with predicate in path expression in SQL WHERE clause:

 * Index can be used to filter rows from table purchaseorder */
SELECT XMLQuery('$po/PurchaseOrder/Requestor'

       PASSING OBJECT_VALUE AS "po" RETURNING CONTENT)
FROM purchaseorder
WHERE

XMLExists('$poPurchaseOrder/LineItems/LineItem/Part[@Quantity = 1]'
                  PASSING OBJECT_VALUE AS "po");

/* WHERE clause of Xquery expression.

 * Index can be used to filter rows from table purchaseorder */

SELECT *
FROM XMLTABLE(
    'for $po in ora:view("purchaseorder")/PurchaseOrder

     where $po/LineItems/LineItem/@ItemNumber="1"

     return $po/Requestor);
```

```
/* XMLTable column in SQL WHERE clause. */
SELECT li.description, li.lineitem
FROM purchaseorder,

         XMLTable('/PurchaseOrder/LineItems/LineItem'
      PASSING OBJECT_VALUE
      COLUMNS lineitem    NUMBER        PATH '@ItemNumber',
             description  VARCHAR2(30)  PATH 'Description',
             partid       NUMBER        PATH 'Part/@Id',
             unitprice    NUMBER        PATH 'Part/@UnitPrice',
             quantity     NUMBER        PATH 'Part/@Quantity') li
WHERE  Lineitem = 4567;


/* Predicate in path expression.
 * Index can be used to identify document fragment.
 * Although index cannot be used to identify rows from the purchaseorder
 * because all rows are returned due to no SQL WHERE clause,
 * index can be used to identify Description fragment
 * that satisfies the path predicate from each row of purchasorder.
 * This query is analogous to scalar subquery usage in a select list
 * of a SQL statement where the scalar subquery has its own
 * where clause that can leverage index */


SELECT XMLCAST(XMLQUERY(
   '/PurchaseOrder/LineItems/LineItem[@ItemNumber=1]/Description'
   PASSING object_value RETURNING CONTENT) AS VARCHAR2(4000))
FROM purchaseorder p;
```

Here are some guidelines on which indexes to choose for your usecase.

Index choosing Guideline 1: Use the Structured XMLIndex when Xpaths are static, and to answer predicates

If you know your Xpaths in advance, the Structured Component of XMLIndex is ideal for your usecase. This will help you get relational performance on your Xqueries for Xpaths that have the Structured XMLIndex on them.

You can get optimal performance by using the structured XMLIndex to index the Xpaths that appear in the predicates. These predicates can be in the SQL statement, or the predicate of the where clause, or in the Xquery itself, as shown in "Example 25: Examples of where XMLIndex could be used".

Creating the structured index as depicted in "Example 27: Creating the XMLIndex with Structured Component" can optimize all the queries above.

Index choosing Guideline 2: Use text index for full text search requirements

If your application has requirements for full text searching, consider using the SQL contains() operator and create a text index on the base XMLType column.

Example 26: Using SQL contains() to perform full-text search

```
create table po of xmltype;
create index po_otext_ix on po (object_value) indextype is
             ctxsys.context;
call dbms_stats.gather_table_stats(USER, 'PO');

select distinct
XMLCast(XMLQuery('$p/PurchaseOrder/ShippingInstructions/address'
                 passing po.object_value as "p" returning content)

       as varchar2(256)) "Address"
from po po
where contains(po.object_value, '$(Fortieth) INPATH
                      (PurchaseOrder/ShippingInstructions/address)') > 0;
```

Index choosing Guideline 3: Fragment extraction

In the presence of queries that project out XML fragments, the indexing approach depends on the average size of documents:

- If the dataset consists of small to medium size documents, you should use one of the following:
  - Either, use the Xquery extension expression (#ora:xq_proc #) to indicate Xquery shall be functionally evaluated. *Note: ora:xq_proc gives you fine-grained control – you can make fragment extraction use xq_proc and predicates use XMLIndex, as long as the predicate Xpaths are not excluded from the XMLIndex.*
  - Or, use Binary XML streaming evaluation.

Index choosing Guideline 4: Combine different indexes as needed

You can use a combination of the different indexes. For example, if you have a table of technical documents, you can create an XMLIndex with structured component for the title, author and date fields, and create an Oracle Text index to answer text-search queries.

Once you have chosen the right indexes for your use case, please refer to the corresponding section for guidelines on how to get the best performance out of these indexes.

XMLIndex Structured Component

Even though the data in the Binary XML may be unstructured, it sometimes contains islands of predictable, structured data. An example is a technical document, with the title, author and date fields. You create and use the structured component of an XMLIndex index for queries that project fixed, structured islands of XML content, even if the surrounding data is relatively unstructured. A structured XMLIndex component organizes such islands in a relational format. It is similar to SQL/XML function XMLTable, and the syntax you use to define the structured component reflects this similarity. The relational tables used to store the indexing data are data-type aware, and each column can be of a different scalar data type. You can thus think of the act of creating the structured component of an XMLIndex index as decomposing a structured portion of your XML data into relational format.

The structured component is a targeted index, and therefore requires careful specification of the Xpaths that are to be indexed, along with their data types. But, the benefits of using such an index are significant for queries with statically known Xpaths.

Some of the advantages of using the structured component are listed below:

1. *Type-Aware, Relational-Style Searches* – The structured component of an XMLIndex has the ability to separate values by type and by path into different columns, and therefore can provide very specific relational-style statistics to the relational Cost Based Optimizer, on which the XMLIndex is built.
2. *Support for Composite B-Tree and Bitmap Indexes* – An internal table of a structured XMLIndex can store values from different XPaths in separate columns, thereby making it possible to create composite B-Tree and bitmap indexes.
3. *No Sub-Query in SQL Predicate* - When structured XMLIndex is used, a predicate in the WHERE clause becomes column-level checks on the structured XMLIndex tables.
4. *Indexing for BI-Style Queries* – SQL constructs such as order-by, group-by, window, etc., enable powerful business intelligence queries over relational data. Applications using order-by, group-by, window, etc., on values within XML data can get relational performance by using structured XMLIndex, since the queries can be rewritten to order-by, group-by, window, etc., over relational table columns. This is accomplished as follows: XMLTable allows values in XML to be projected out as a virtual table. A query that uses the XMLTable function can be rewritten to simple access of the relational tables of a structured XMLIndex. This means that order-by, group-by, window, etc., operating on columns of the virtual table are translated to order-by, group-by, window, etc., operating on the corresponding physical columns of the structured XMLIndex tables.

The example below shows how to create structured XMLIndex. It uses the Purchase Order schema, which has a collection called "LineItem". For each XML node matching the row pattern /PurchaseOrder/LineItems/LineItem, this XMLIndex projects out in its relational index table 5 columns – the values of these nodes are the values of nodes matching relative XPaths @ItemNumber, Description, Part/@Id, Part/@UnitPrice, and Part/@Quantity. The internal index table will have as many rows for each XML document as the number of LineItem nodes within the document. The index DDL specifies the name of the table (lineitem_tab in this case), the names of the 5 columns, and the SQL data types of these 5 columns.

Example 27: Creating the XMLIndex with Structured Component

```
CREATE INDEX po_struct ON purchaseorder (OBJECT_VALUE)

INDEXTYPE IS XDB.XMLIndex

PARAMETERS (

'XMLTable lineitem_tab ''/PurchaseOrder/LineItems/LineItem''

     COLUMNS lineitem    NUMBER       PATH ''@ItemNumber'',
             description VARCHAR2(30) PATH ''Description'',
             partid      NUMBER       PATH ''Part/@Id'',
             unitprice   NUMBER       PATH ''Part/@UnitPrice'',
             quantity    NUMBER       PATH ''Part/@Quantity''');
```

Below are the guidelines on how to get the best performance out of your structured XMLIndex.

Structured Index Guideline 1: Use Structured Index instead of multiple functional indexes and/or virtual columns

In XML usecases where user wants to project out several relational key columns of XML so that they can build B-tree indexes over these columns for quick search, structured XMLIndex is ideal. Structured XMLIndex projects out one relational table capturing all the key relational columns for efficient search, instead of relying on multiple virtual columns (VC) that are inefficient. These structured XMLIndex columns are efficiently populated in a single scan of the input base document - something that cannot be done with virtual columns. Also, the structured XMLIndex based approach works in cases where the XML has collections, whereas the VC based approach cannot be used when the projected value is within an XML collection.

Structured Index Guideline 2: Make Index and Query datatypes correspond

The relational tables that are used for an XMLIndex structured component use SQL data types. XQuery expressions that are used in queries use XML data types (XML Schema data types and XQuery data types). XQuery typing rules can automatically change the data type of a subexpression, to ensure coherence and type-checking. For example, if a document that is queried using XPath expression /PurchaseOrder/LineItem[@ItemNumber = 25] is not XML schema-based, then the subexpression @ItemNumber is xs:untypedAtomic, and it is then automatically cast to xs:double by the XQuery = comparison operator. To index this data using an XMLIndex structured component you must use BINARY_DOUBLE as the SQL data type.

This is a general rule. For an XMLIndex index with structured component to apply to a query, the data types must correspond. Table 2 in Guideline 7: "Using proper XQuery and SQL Typing"

If the XML and SQL data types involved do not have a built-in one-to-one correspondence, then you must make them correspond (according to Table 2), in order for the index to be picked up for your query. There are two ways you can do this:

- Make the index correspond to the query – Define (or redefine) the column in the structured index component, so that it corresponds to the XML data type. For example, if a query that you want to index uses the XML data type xs:double, then define the index to use the corresponding SQL data type, BINARY_DOUBLE.
- Make the query correspond to the index – In your query, explicitly cast the relevant parts of an XQuery expression to data types that correspond to the SQL data types used in the index content table.

Structured Index Guideline 3: Use XMLTable views with corresponding index, e.g BI style queries

Since the structured component of XMLIndex is built on the idea of an XMLTable, such an index fits nicely for usecases where this relational paradigm is applicable. For application developers who want a relational access paradigm, one or more relational views built on XMLTable should be created. The XMLTable function provides a way to expose key values from within XML as relational columns. Querying of XML in many usecases can be hidden within the definitions of relational views that use the XMLTable function, making it easier for XML to penetrate into the world of application developers who are familiar with SQL and want to be spared the complexity of XPath/XQuery. In such cases, the structured XMLIndex definition will match the definitions of the relational views.

The example below shows how XMLTable() provides a relational table abstraction over XML, and the next example shows how to create a corresponding view, and example 27 shows the corresponding index for it.

Example 28: XMLTable Provides a Virtual Table Abstraction over XML

```
SELECT lines.lineitem    ,
       lines.description,
       lines.partid      ,
       lines.unitprice   ,
       lines.quantity
FROM   purchaseorder,
       XMLTable('/PurchaseOrder/LineItems/LineItem'
PASSING OBJECT_VALUE
COLUMNS   lineitem        NUMBER        PATH '@ItemNumber',
description VARCHAR2(30) PATH 'Description',
partid      NUMBER   PATH 'Part/@Id',
unitprice   NUMBER   PATH 'Part/@UnitPrice',
quantity    NUMBER   PATH 'Part/@Quantity') lines;

LINEITEM DESCRIPTION PARTID UNITPRICE QUANTITY
-------- ----------- ------ --------- --------
11 Orphic Trilogy 37429148327 80 3
22 Dreyer Box Set 37429158425 80 4
11 Dreyer Box Set 37429158425 80 3
```

Example 29: Relational View Using XMLTable, and corresponding structured XMLIndex

```
CREATE VIEW lineitems_v
(lineitem, description, partid, unitprice, quantity)
AS SELECT
       lines.lineitem, lines.description, lines.partid,
       lines.unitprice, lines.quantity
FROM purchaseorder,
XMLTable('/PurchaseOrder/LineItems/LineItem'
PASSING OBJECT_VALUE
COLUMNS lineitem          NUMBER       PATH '@ItemNumber',
       description        VARCHAR2(30) PATH 'Description',
       partid             NUMBER       PATH 'Part/@Id',
       unitprice          NUMBER       PATH 'Part/@UnitPrice',
       quantity           NUMBER       PATH 'Part/@Quantity'
) lines;
```

One common usecase for this is that of BI-style queries. SQL constructs such as order-by, group-by, window, etc., enable powerful business intelligence queries over relational data. XMLTable allows values in XML to be projected out as a virtual table. Order-by, group-by, window, etc., can operate on columns of the virtual table. Structured XMLIndex internally organizes its storage tables in a manner that reflects the virtual table(s) exposed by XMLTable. Therefore, structured XMLIndex is well suited for indexing XML data in a way that makes such XMLTable based queries very efficient. A query that uses the XMLTable function can be rewritten to simple access of the relational tables of a structured XMLIndex. This means that order-by, group-by, window, etc., operating on columns of the virtual table are translated to order-by, group-by, window, etc., operating on the corresponding physical columns of the structured XMLIndex tables.

We recommend that the user create relational views over XML using XMLTable, where the views project all columns of interest to the BI application. Application queries should be written against these relational views. If structured XMLIndex is created in 1-1 correspondence to these views, Oracle RDBMS will make sure that queries over the views are seamlessly translated into queries over the relational tables of the structured XMLIndex, thereby giving relational performance.

Structured Index Guideline 4: Create Secondary Indexes, especially for predicates

"Example 27: Creating the XMLIndex with Structured Component" creates relational table lineitem_tab under the covers. To get good performance for value-based searches, it is important that the user create secondary indexes on the index table. This is illustrated in the example below.

Example 30: Creating Secondary Indexes on Structured XMLIndex Tables

```
CREATE INDEX li_itemnum_idx      ON lineitem_tab(lineitem);
CREATE INDEX li_desc_idx   ON lineitem_tab(description);
CREATE INDEX li_partid_idx       ON lineitem_tab(partid);
CREATE INDEX li_uprice_idx       ON lineitem_tab(unitprice);
CREATE INDEX li_quantity_idx     ON lineitem_tab(quantity);
```

Composite B-Tree indexes, bitmap indexes and domain indexes (e.g., Oracle Text) can also be created on the index table.

Example 31: Creating Oracle Text Index on Structured XMLIndex Table

```
CREATE INDEX li_desc_ctx_idx ON lineitem_tab(description)
indextype is ctxsys.context;
```

It is the responsibility of the user to create these secondary indexes. No secondary index is created automatically by the system for the structured XMLIndex component, as the user is the best judge of what secondary index best suites his needs. Once the secondary indexes are created, the user should gather statistics on the base table so that the optimizer can pick up the indexes.

If a query uses a particular XPath in a predicate, including the SQL WHERE clause, then creating a secondary index on the corresponding column of the structured XMLIndex table is highly recommended.

Structured Index Guideline 5: Check the execution plan to see if structured index is used

After creating the necessary indexes to speed up your queries, you need to verify that the execution plan is indeed picking up the index. For example, let's say you have created the structured XMLIndex as depicted in "Example 27: Creating the XMLIndex with Structured Component", and secondary indexes as depicted in Example 30. Then you should run an explain plan on your query, as illustrated in the example below:

Example 32: Using Explain Plan to determine that the index is picked up

```
EXPLAIN PLAN FOR

SELECT XMLCAST(XMLQUERY( '/PurchaseOrder/Requestor' PASSING object_value RETURNING
CONTENT) AS VARCHAR2(4000))

FROM purchaseorder p

WHERE XMLExists('/PurchaseOrder/LineItems/LineItem[xs:decimal(@ItemNumber)=1]' PASSING
object_value);

Explained.

SQL> select Id, Operation, Name from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
-------------------------------------------------------

Plan hash value: 2801523227

-------------------------------------------------------

| Id  | Operation                      | Name          |
-------------------------------------------------------

|   0 | SELECT STATEMENT               |               |
|   1 |  NESTED LOOPS SEMI             |               |
|   2 |   TABLE ACCESS FULL            | PURCHASEORDER |
|*  3 |   TABLE ACCESS BY INDEX ROWID| LINEITEM_TAB    |
|*  4 |    INDEX RANGE SCAN            | LI_ITEMNUM_IDX |
-------------------------------------------------------
```

The execution plan shows that the query gets rewritten to use the structured index storage table LINEITEM_TAB and the secondary index LI_ITEMNUM_IDX.

Structured Index Guideline 6: Indexing Master-Detail relationships

In cases where the structured islands have a master-detail kind of relationship, structured XMLIndex provides a way to capture each structured island as a relational table, with a primary-foreign key relationship between the tables. Here are definitions of such a master-detail view, and its corresponding structured XMLIndex:

Example 33: Relational View with Master-Detail Relationship

```
CREATE OR REPLACE VIEW purchaseorder_detail_view AS
       SELECT po.reference, li.*

       FROM purchaseorder 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;
```

Example 34: Structured XMLIndex to Index Master-Detail Relationship

```
CREATE INDEX po_struct ON po_tab (OBJECT_VALUE)

INDEXTYPE IS XDB.XMLIndex

PARAMETERS ('XMLTable po_ptab

         XMLNAMESPACES(DEFAULT ''http://www.example.com/po''),

                 ''/purchaseOrder''

        COLUMNS orderdate  DATE           PATH ''@orderDate'',

                Id         BINARY_DOUBLE PATH ''@id'',

                items             XMLType       PATH ''items/item'' VIRTUAL

       XMLTable li_tab

    XMLNAMESPACES(DEFAULT ''http://www.example.com/po''),

                    ''/item'' PASSING items

       COLUMNS partnum            VARCHAR2(15)  PATH ''@partNum'',

               description        CLOB           PATH ''productName'',

               usprice            BINARY_DOUBLE PATH ''USPrice'',

               shipdat            DATE          PATH ''shipDate''');
```

Structured Index Guideline 7: Split fragement extraction and value search between SELECT and WHERE clause

Instead of using a single XQuery for fragment extraction as well as for value search, use XMLQuery() in the SELECT clause for fragment extraction and use XMLExists() in the WHERE clause for value search. By doing this separation, we are able to make structured xmlindex be picked up for value search, while binary XML streaming is used for fragment extraction.

Example 35: Splitting fragment extraction and value search

In this example, Query 1 is a better formulation than Query 2 when following XMLIndex is present:

Index definition:

```
CREATE TABLE XML_TEST (XML_DOC XMLType)

       XMLType XML_DOC STORE AS BINARY XML;


CREATE INDEX XML_TEST_IX ON XML_TEST (XML_DOC)

       INDEXTYPE IS XDB.XMLIndex
PARAMETERS ('GROUP XML_TEST_G XMLTable XML_TEST_X

      XMLNAMESPACES(''http://example.com/metadata'' as "m"),

      ''/m:object'' COLUMNS

      TENANT VARCHAR(100) PATH ''m:meta/m:tenant'',
      ID VARCHAR(250) PATH ''m:meta/m:id''');
```

CREATE INDEX XML_TEST_IX_1 ON XML_TEST_X(TENANT, ID);

Query 1: Better

```
SELECT

XMLQUERY('declare namespace m="http://example.com/metadata";
          for $obj in $doc/m:object

          return <m:object>

                    {$obj/m:meta/m:id}{$obj/m:meta/m:tenant}

                 </m:object>'

         passing T.XML_DOC as "doc" returning content)
FROM XML_TEST T
WHERE

XMLEXISTS('declare namespace m="http://example.com/metadata";

           $doc/m:object[m:meta/m:tenant=$tenant

                          and m:meta/m:id=$id]'

           passing T.XML_DOC as "doc",

                   'tenant5' as "tenant",
                    'id_555' as "id");
```

Query 2: Avoid

```
SELECT X.XML_DOC

FROM XML_TEST T,
      XMLTABLE(

        XMLNAMESPACES('http://example.com/metadata' as "m"),
         'for $obj in /m:object
         where $obj/m:meta/m:tenant="tenant5" and

               $obj/m:meta/m:id="id_5555"
         return <m:object>

                    {$obj/m:meta/m:id}{$obj/m:meta/m:tenant}

                </m:object>'
        PASSING T.XML_DOC COLUMNS XML_DOC XMLTYPE PATH '.') X;
```

Structured Index Guideline 8: For ordering query results, use SQL ORDER BY along with XMLTable

Instead of using XQuery ORDER BY clause, use XMLTable to project out the key by which to order and then use SQL ORDER BY. In the example below, the query shows fragment extraction together with value search. Fragments are ordered by tenant, id which are projected out in the XMLTABLE() clause:

Example 36: Using SQL order by

```
SELECT XMLQUERY('declare namespace

       m="http://example.com/metadata";

            for $obj in $doc/m:object

            return <m:object>

                     {$obj/m:meta/m:id} {$obj/m:meta/m:tenant}

                   </m:object>'
            passing T.XML_DOC as "doc" returning content)

FROM XML_TEST T,
     XMLTABLE(XMLNAMESPACES('http://example.com/metadata'

                                as "m"),

               '$doc/m:object' PASSING T.XML_DOC as "doc"
                COLUMNS
                     tenant VARCHAR(100) PATH 'm:meta/m:tenant',
                     id  VARCHAR(250) PATH 'm:meta/m:id'
                ) tt

WHERE

       XMLEXISTS('declare namespace m="http://example.com/metadata";

             $doc/m:object[m:meta/m:tenant=$tenant]'
             passing T.XML_DOC as "doc", 'tenant5' as "tenant")

ORDER BY tt.tenant, tt.id;
```

Text Index

Besides accessing XML nodes such as elements and attributes, it is sometimes important to provide fast access to particular passages of text within XML text nodes. This is the purpose of Oracle Text indexes: they index full-text strings. Full-text indexing is particularly useful for document-centric applications, which often contain a mix of XML elements and text-node content. Full-text searching can often be made more powerful and more focused, by combining it with structural XML searching, that is, by restricting it to certain parts of an XML document, which are identified by using XPath expressions.

An Oracle Text CONTEXT index created on an XMLType column enables SQL function contains() and facilitates the XQuery function ora:contains() for full-text search over XML. The example below shows how to create an Oracle Text index on an XMLType column.

Example 37: Creating an Oracle Text Index

```
CREATE INDEX po_otext_ix ON po_clob (OBJECT_VALUE)
       INDEXTYPE IS CTXSYS.CONTEXT;
Index created.
```

Oracle Text indexing is completely orthogonal to the other types of indexing. Whenever SQL function contains() or XPath function ora:contains() is used, an Oracle Text index can be used for full-text search. The example below demonstrates this in the case where both an XMLIndex index and an Oracle Text index are defined on the same XML data. The Oracle Text index is created on the VALUE column of the XMLIndex path table.

Example 38: Using an Oracle Text Index with other indexes

```
CREATE INDEX po_otext_ix ON my_path_table (VALUE)
       INDEXTYPE IS CTXSYS.CONTEXT;
Index created.

EXPLAIN PLAN FOR
  SELECT DISTINCT XMLCAST(XMLQUERY(
             '/PurchaseOrder/ShippingInstructions/address' PASSING object_value RETURNING
CONTENT) AS VARCHAR2(4000)) "Address"
    FROM po_clob
    WHERE contains(OBJECT_VALUE, '$(Fortieth) INPATH
                   (PurchaseOrder/ShippingInstructions/address)') > 0;

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------
| Id  | Operation                    | Name                          |
---------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                               |
|*  1 |  TABLE ACCESS BY INDEX ROWID| MY_PATH_TABLE                 |
|*  2 |   INDEX RANGE SCAN           | SYS78942_PO_XMLINDE_ORDKEY_IX |
|   3 |  HASH UNIQUE                 |                               |
|*  4 |   TABLE ACCESS FULL          | PO_CLOB                       |
---------------------------------------------------------------------

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

   1 - filter("SYS_P0"."PATHID"=HEXTORAW('35EF580A')  AND
SYS_XMLI_LOC_ISNODE("SYS_P0"."LOCATOR")=1)
```

```
   2 - access("SYS_P0"."RID"=:B1)
       filter("SYS_P0"."RID"=:B1)
   4 - filter("CTXSYS"."CONTAINS"(SYS_MAKEXML("SYS_ALIAS_1"."XMLDATA"),
    '$(Fortieth) INPATH (PurchaseOrder/ShippingInstructions/address)')>0)
```

The execution plan in the example above references both the XMLIndex index and the Oracle Text index, indicating that both are used. The XMLIndex index is indicated by its path table, MY_PATH_TABLE, and its order-key index, SYS78942_PO_XMLINDE_ORDKEY_IX.

The Oracle Text index is indicated by the reference to SQL function contains in the predicate information.

Full text search on xmltype can be done using contains() function in SQL or by using ora:contains() within XPath or xquery expressions. The details of each function are outlined below.

Searching XML data using contains()

You can perform Oracle Text operations such as *contains* and *score* on XMLType columns. You will need to create Oracle Text index (ctxsys.context) on the xmltype column in order for *contains* to execute. Note that the *contains* operator is not XML-namespace aware. The example below shows an Oracle Text search using SQL function contains.

Example 39: Searching XML Data Using SQL Function CONTAINS

```
SELECT DISTINCT XMLCast(XMLQuery('$p/PurchaseOrder/ShippingInstructions/address'
          PASSING po.OBJECT_VALUE AS "p" RETURNING CONTENT)
          AS VARCHAR2(256)) "Address"
  FROM po_clob po
  WHERE contains(po.OBJECT_VALUE,
                '$(Fortieth) INPATH
        (PurchaseOrder/ShippingInstructions/address)') > 0;

Address
------------------------------
1200 East Forty Seventh Avenue
New York
NY
10024
USA
1 row selected.
```

The execution plan for this query shows two ways that the Oracle Text CONTEXT index is used:

1. It references the index explicitly, as a domain index.

2. It refers to SQL function contains in the predicate information.

```
PLAN_TABLE_OUTPUT

-------------------------------------------------------------

| Id  | Operation                   | Name        | Rows  | Bytes |

-------------------------------------------------------------

|   0 | SELECT STATEMENT            |             |     7 | 14098 |
|   1 |  HASH UNIQUE                |             |     7 | 14098 |
|   2 |   TABLE ACCESS BY INDEX ROWID| PO_CLOB     |     7 | 14098 |
|*  3 |    DOMAIN INDEX             | PO_OTEXT_IX |       |       |
------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("CTXSYS"."CONTAINS"(SYS_MAKEXML('..........',523
            3,"XMLDATA"),'$(Fortieth) INPATH
        (PurchaseOrder/ShippingInstructions/address)')>0)
```

Searching XML data using ora:contains()

The XQuery function ora:contains() lets users search for keywords within specific XPath or xquery contexts. The evaluation of ora:contains() does not need Oracle Text index (ctxsys.context) to execute functionally, but may need it for performance.

When possible, Oracle internally rewrites the ora:contains() operator to a contains() operator. This happens if both of the following conditions are satisfied:

1. The XPath or xquery context of ora:contains() can be rewritten to user-defined column of structured xmlindex.
2. There is a TRANSACTIONAL Oracle Text index on the column.

If both the conditions above are true, then ora:contains() is rewritten to a contains() on the column. If Oracle Text index on column is not TRANSACTIONAL, then ora:contains() is evaluated functionally (no index). The example below shows how to create such an index:

Example 40: Searching XML data using ora:contains()

```
create table myemp of xmltype tablespace sysaux;

create index emp_xtidx on myemp (object_value)
       indextype is xdb.xmlindex parameters('
       GROUP gp1
          XMLTABLE ETAB
          XMLNamespaces(DEFAULT ''http://www.oracle.com/tkxmsch1.xsd''),
          ''/Employee''
          columns "eid" integer PATH ''EmployeeId'',
                  "fname" varchar2(70) PATH ''FirstName'',
                  "lname" varchar2(70) PATH ''LastName'',
                  "jdesc" varchar2(70) PATH ''JobDesc''');

create index jdctxidx on ETAB (jdesc)
   indextype is ctxsys.context parameters ('transactional');

select xmlcast(xmlquery('
             declare default element namespace
             "http://www.oracle.com/tkxmsch1.xsd";(::)
             /Employee/FirstName' passing value(e) returning content) as varchar2(50) )
from myemp e
where xmlexists( '
               declare default element namespace
               "http://www.oracle.com/tkxmsch1.xsd";(::)
               /Employee[ora:contains(JobDesc, "program")>0]'
               passing value(e))
/
```

To get the best performance for your full text queries, follow the guidelines given below:

Text Index Guildeline 1: Binary XML Storage: Use contains()

If your storage is binary XML, then create Oracle Text index on xmltype and use contains(). This is the recommended approach for full-text search over binary XML. But, be aware that Oracle Text index does not understand XML namespaces.

Text Index Guildeline 2: Binary XML Storage: Creating Text Index on structured XMLIndex columns

If your storage is binary XML, look at creating Oracle Text index on user-defined column of structured XMLIndex only if guideline #2 cannot be used.

User-defined column of structured XMLIndex can be defined as CLOB to avoid any truncation of node values. But, having a CLOB column dramatically affects the load performance of structured XMLIndex.

Conclusion

Oracle XML DB support for the XQuery language is provided through a native implementation of SQL/XML functions XMLQuery, XMLTable, XMLExists, and XMLCast. This paper started out by discussing storage independent XQuery best practices, then moved on to the guidelines for getting the best performance out of various storage/indexing options.

Appendix A: Semantic differences between the deprecated mainly XPath 1.0 based functions and standard SQL/XML XQuery based functions

There are some important differences between the deprecated and the XQuery based syntax, which are listed below to make the migration easier for the users.

In the de-supported extract(), existsNode(), table(xmlsequence()), extractValue(), only XPath 1.0 can be used in the path expression. The SQL/XML standard operators XMLQuery(), XMLExists(), XMLTable, XMLCast() use XQuery 1.0 in the query expression. Other than this important difference, there are several other non-standard behavior in the de-supported operators that users must pay special attention when migrating to use the standard based operators.

- Schema based datatype comparison: When de-supported operators are applied to schema based binary XMLType column, the schema based datatype comparison semantics is applied, for example, comparing non-string type with string results in casting and datatype specific comparison. However, in the standard operators, **XQuery date type casting functions must be used**. Otherwise an error will be raised. See XQuery Guideline 7.

  Example:

  Assume @podate is xs:date type and @poid is xs:integer type and purchaseOrder is an XMLType table storing schema based purchaseOrder XML document instances.

  De-supported syntax:

  ```
  Select 1 from purchaseOrder p
  where existsNode(value(p), '/PurchaseOrder[@podate > "1998-09-02"]') = 1
  ```

  Standard based syntax:

  ```
  Select 1 from purchaseOrder p  where xmlexists( 'declare namespace po =
  http://www.po.com;/PurchaseOrder[@podate >xs:date( "1998-09-02")]' passing
  value(p))
  ```

  The following query raises type errors

  ```
  Select 1 from purchaseOrder p  where xmlexists( 'declare namespace po =
  http://www.po.com;/PurchaseOrder[@podate > "1998-09-02"]' passing value(p))
  ```

  De-supported syntax:

  ```
  Select 1 from purchaseOrder p
  where existsNode(value(p), '/PurchaseOrder[@poid = "3456"]') = 1
  ```

  Standard based syntax:

  ```
  Select 1 from purchaseOrder p  where xmlexists( 'declare namespace po =
  http://www.po.com;/PurchaseOrder[@poid = 3456]' passing value(p))
  ```

- Namespace patching: As the example shown above, **the namespace declaration must be specified unless the XML document has no namespace** whereas in the de-supported syntax, the namespace might be patched even if it is NOT specified as the third parameter of the operator.
- existsNode returns 0 or 1 while XMLExists returns Boolean, so you can use new syntax in the SQL WHERE clause directly. To use it in the SELECT list, please refer to "Example 3: Using XMLExists() with CASE Expression in select list".
- Bind variable: There is no need to use string concatenation operator || to construct XPath string to embed bind variable as in the de-supported syntax. Instead, use PASSING clauses to pass bind varaibles to XQuery based functions.

  Example:

  De-supported syntax:

  ```
  Select  value(p) from purchaseOrder p
  where existsNode(value(p), '/PurchaseOrder[@podate >' || :1: ']') = 1
  ```

  Standard syntax:

```
Select value(p) from purchaseOrder p
where xmlexists( 'declare namespace po = http://www.po.com;/PurchaseOrder[@podate >
xs:date($bindvar)]' passing value(p), :1 as "bindvar")
```

- ora:instanceof() and ora:instanceof-only() are only usable in the XPath of the de-supported syntax. Use XQuery 'instance of' expression and '@xsi:type =' respectively in the standard syntax.

  Example: ora:instanceof()

  De-supported syntax:

  ```
  select extract(value(r),'/N2:R1[ora:instanceof(.,"N1:superType1")]',
  'xmlns:N1="http://www.oracle.com/xdb/N1" xmlns:N2="http://www.oracle.com/xdb/N2"
  xmlns:ora="http://xmlns.oracle.com/xdb"')       from R1 r;
  ```

  Standard syntax:

  ```
  select XMLQuery('declare namespace N1="http://www.oracle.com/xdb/N1";
  declare namespace N2="http://www.oracle.com/xdb/N2";
  /N2:R1[. instance of element(N2:R1, N1:superType1)]'
  passing object_value returning content) from R1 r ;
  ```

  Example: ora:instanceof-only()

  De-supported syntax:

  ```
  select extract(value(r),'/N2:R1[ora:instanceof-only(.,"N1:superType1")]',
  'xmlns:N1="http://www.oracle.com/xdb/N1" xmlns:N2="http://www.oracle.com/xdb/N2"
  xmlns:ora="http://xmlns.oracle.com/xdb"') from R1 r;
  ```

  Standard syntax:

  ```
  select XMLQuery('declare namespace N1="http://www.oracle.com/xdb/N1";       declare
  namespace N2="http://www.oracle.com/xdb/N2";
  /N2:R1[@xsi:type="N1:superType1"]' passing object_value returning content)
  from R1 r ;
  ```

  Notice that xsi:type predicate is also supported in XPath, i.e., the following query works the same as the two above:

  ```
  select extract(value(r),'/N2:R1[@xsi:type="N1:superType1"]',
  'xmlns:N1="http://www.oracle.com/xdb/N1" xmlns:N2="http://www.oracle.com/xdb/N2"
  xmlns:ora="http://xmlns.oracle.com/xdb"') from R1 r;
  ```

ora:upper(), ora:lower(), ora:to_number(), ora:to_date() are only usable in the XPath of the de-supported syntax. Use corresponding XQuery F&O functions fn:upper-case(), fn:lower-case(), xs:decimal(), xs:date() respectively in the standard syntax.

Example: DBMS_XMLGEN:

De-support syntax:

```
SELECT sys_XMLGen(km_t(kid,

               kname,

               knum,

               CAST(MULTISET (SELECT kid, kdid, kdname

                       FROM ktest_d d

                      WHERE d.kid = m.kid) AS kdlist_t))).getclobval() AS detail

FROM ktest_m m;
```

Standard syntax:

```
select XMLSERIALIZE

       (

       document
```

```
        XMLELEMENT
        (
                "KD_LIST",
                XMLAGG
                (
                (
                SELECT XMLAGG
                        (
                        XMLELEMENT
                        (
                        "KD_T",
                        XMLELEMENT("KID",KID),
                        XMLELEMENT("KDID",KDID),
                        XMLELEMENT("KDNAME",KDNAME)
                        )
                        )
                from KTEST_D d
                where d.kid = m.kid
                )
        )
        )
        as clob indent size=2
        )
from KTEST_M m;
```

Connect with us

Call +**1.800.ORACLE1** or visit **oracle.com**. Outside North America, find your local office at: **oracle.com/contact**.

blogs.oracle.com facebook.com/oracle twitter.com/oracle