Mastering XML DB Queries in Oracle Database 10g Release 2

An Oracle White Paper
March 2005

Mastering XML DB Queries in Oracle Database 10g Release 2

Mastering XML DB Queries in Oracle Database 10g Release 2

INTRODUCTION

XML has become ubiquitous since W3C's official recommendation of the XML specification in 1998. As a format suitable for representing both structured data and unstructured data, XML is unifying the two traditionally separate worlds of data management and content management. Finding information hidden in this rapidly growing volume of XML data demands unprecedented query technology.

To find information in this rapidly growing volume of XML data demands unprecedented query technology.

For decades, SQL has been the trusted query language for structured data. With its solid foundation, SQL is also the industry standard for manipulating (i.e., inserting, updating, deleting) and managing (e.g., creating, altering, and deleting of constraints, tables, views, etc.) structured data. As the volume of unstructured data multiplied, SQL has kept ahead of this information revolution by defining new features in new releases of the standard over the years. The Part 14 (SQL/XML) of the latest SQL 2003 and the upcoming SQL 2005 standard defines a comprehensive and detailed specification of new SQL capabilities for managing, querying, and manipulating XML data. SQL/XML has seamlessly merged SQL and XML by building on both the industry's most enduring SQL standard and the W3C's XML standards (e.g., XML, XML Schema, XPath, XQuery, etc.).

In a parallel development, many of the same SQL committee members have devoted their collective expertise to develop the XQuery specifications within W3C since 1999. The mission of this XML Query language Working Group is to provide flexible query facilities to extract data from real and virtual documents on the World Wide Web, thereby providing the needed interaction between the Web world and the database world. Ultimately, collections of XML files will be accessed like databases using the upcoming XQuery language. Toward this end, the XML Query Working Group has produced a number of specifications, including XQuery 1.0, XML Query Use Cases, XQuery and XPath Data Model, XML Query Requirements, etc. As XQuery is expected to become a formal W3C recommendation in 2005, there is a strong demand for its commercial implementation.

As the leader in information technology products, Oracle has been a major driving force for both the SQL/XML and the XQuery standardization effort. Coinciding with the standardization effort, Oracle has introduced increasingly deeper and broader SQL/XML integration since Oracle 9i Release 2. In Oracle Database 10g Release 2, database-native XQuery will be supported.

In the rest of the paper, we will first describe the XPath Rewrite technology for XML Schema-Based structured storage. XPath Rewrite technology is the cornerstone shared by both of Oracle's SQL/XML and XQuery to achieve high performance and scalability. In the third section, we will discuss Oracle's XPath-based SQL/XML query functions. In the fourth section, we will describe Oracle's database-native XQuery implementation based on the upcoming standards of SQL 2005 SQL/XML and W3C XQuery. We will show that Oracle's SQL/XML and XQuery are complementary approaches for scalable and high performance querying of XML data in all usage scenarios.

XPATH REWRITE WITH XML SCHEMA-BASED STRUCTURED STORAGE

Structured storage has numerous advantages in managing XML documents, including optimized memory management, reduced storage requirements, B-tree indexing, and in-place updates. For complex-structured XML documents, structured storage provides a number of options for optimal storage of collections according to actual usage scenarios. Structured storage does require somewhat increased processing of the corresponding XML schema during ingestion and retrieval of entire documents.

For complex-structured XML documents, structured storage provides a number of options for optimal storage of collections according to actual usage scenarios

Structured storage of XML documents is based on decomposing the content of the document into a set of database-native objects. When an XML schema is registered with Oracle XML DB, the required database-native type definitions are automatically generated from the XML schema.

A database-native type definition is generated from each `complexType` defined by the XML schema. Each element or attribute defined by the `complexType` becomes amattribute in the corresponding database-native type. Oracle XML DB automatically maps the scalar datatypes defined by the W3C XML Schema Recommendation to the database-native scalar datatypes.

The generated database-native types allow XML content, compliant with the XML schema, to be decomposed and stored in the database as a set of objects without any loss of information. When the document is ingested, the constructs defined by the XML schema are mapped directly to the equivalent database-native types. This allows Oracle XML DB to leverage the full power of Oracle Database when managing XML and can lead to significant reductions in the amount of space required to store the document. It can also reduce the amount of memory required to query and update XML content.

Furthermore, as you can learn more in a companion white paper titled, "Mastering XML DB Storage in Oracle Database 10g", it is important to ensure that members of a collection are stored as rows in a nested table or an XMLType table for efficient query operations on members of the collection.

Understanding XPath Rewrite

XPath rewrite is the key to improving the performance of SQL statements containing XPath expressions by converting the functions into conventional relational SQL statements. This insulates the database optimizer from having to understand the XPath notation and the XML data model. The database optimizer processes the rewritten SQL statement in the same manner as any other SQL statement. In this way, it can derive an execution plan based on conventional relational algebra. This results in the execution of SQL statements with XPath expressions with near relational performance. As a result, XML DB applications can reach optimal scalability.

This enables the use of B-Tree or other indexes, if present on the column, to be used in query evaluation by the Optimizer. This XPath rewrite mechanism is used for XPaths in SQL functions such as `existsNode`, `extract`, `extractValue`, and `updateXML`. This enables the XPath to be evaluated against the XML document without having to ever construct the XML document in memory.

The rewrite of XPath expressions happen under the following conditions:

- The XML function or method is rewritable: SQL functions `extract`, `existsNode`, `extractValue`, `updateXML`, `insertChildXML`, `deleteXML`, and `XMLSequence` are rewritten. Except method `existsNode()`, none of the corresponding `XMLType` methods are rewritten.
- The XPath construct is rewritable: XPath constructs such as simple expressions, wildcards, and descendent axes get rewritten. The XPath may select attributes, elements or text nodes. Predicates also get rewritten to SQL predicates. Expressions involving parent axes, sibling axes, and so on are not rewritten.
- The `XMLSchema` constructs for these paths are rewritable: `XMLSchema` constructs such as complexTypes, enumerated values, lists, inherited types, and substitution groups are rewritten.
- The storage structure chosen during the schema registration is rewritable: Storage using the object-relational mechanism is rewritten. Storage of complex types using `CLOB`s are not rewritten

Using Indexes to Improve Performance of XPath-Based Functions

For structured storage, Oracle XML DB supports the creation of three kinds of index on XML content:

- B-Tree indexes. When the `XMLType` table or column is based on structured storage techniques, conventional B-Tree indexes can be created on underlying SQL types.

- Function-based indexes. These can be created on any `XMLType` table or column.
- Text-based indexes. These can be created on any `XMLType` table or column.

Indexes are typically created by using SQL function `ExtractValue`, although it is also possible to create indexes based on other functions such as `existsNode`. During the index creation process, Oracle XML DB uses XPath rewrite to determine whether it is possible to map between the nodes referenced in the XPath expression used in the `CREATE INDEX` statement and the attributes of the underlying SQL types. If the nodes in the XPath expression can be mapped to attributes of the SQL types, then the index is created as a conventional B-Tree index on the underlying SQL objects. If the XPath expression cannot be restated using object-relational SQL then a function-based index is created.

ORACLE SQL/XML BASED ON XPATH

As the amount of data expressed in XML grows, it becomes necessary to store, manage, and search that data in a robust, secure, and scalable environment, i.e., a database. With SQL/XML you can have all the benefits of a relational database plus all the benefits of XML. New in SQL 2003 standard as Part 14, SQL/XML defines how SQL can be used in conjunction with XML in a database. Part 14 provides detailed definition of a new XML type, the values of an XML type, mappings between SQL constructs and XML constructs, and functions for generating XML from SQL data. Since Oracle Database 9i Release 2, SQL/XML features had been supported as an integral part of the XML DB. XML DB also includes a number of additional XPath-based SQL extensions to support querying, updating, and transformation of XML data. Oracle has been working closely with the SQL standard committee to standardize these extensions in the upcoming SQL 2005 standard.

With XPath-based SQL/XML functions, additional predicates can also be included in the XPath expressions used with any SQL/XML functions. A rule of thumb is to use a SQL/XML function in the `WHERE` clause to find the documents for further extraction of node(s) by a SQL/XML function in the `SELECT` or the `FROM` clause .

Accessing XML Fragments or Nodes Using Extract

The SQL function `Extract` returns the nodes that match an XPath expression. Nodes are returned as an instance of `XMLType`. The result of `extract` can be either a complete document or an XML fragment. The functionality of SQL function `extract` is also available through `XMLType` method `extract()`.

The example below accesses XML Fragments Using the `Extract` function. The query returns an `XMLType` value containing the Reference element that matches the XPath expression.

```
SELECT Extract(OBJECT_VALUE, '/PurchaseOrder/Reference')
  FROM purchaseorder;

EXTRACT(OBJECT_VALUE, '/PURCHASEORDER/REFERENCE')
---------------------------------------------------
<Reference>SBELL-2002100912333601PDT</Reference>

1 row selected.
```

Accessing Text Nodes and Attribute Values Using ExtractValue

The SQL function `ExtractValue` returns the value of the text node or attribute value that matches the supplied XPath expression. The value is returned as a SQL scalar value. The XPath expression passed to `ExtractValue` must uniquely identify a single text node or attribute value within the document. This function supports strong typing by leveraging the associated XML schema.

The example below accesses a Text Node Value Matching an XPath Expression Using `ExtractValue.` The query returns the value of the text node associated with the `Reference` element that matches the XPath expression. The value is returned as a `VARCHAR2` value:

```
SELECT extractValue(OBJECT_VALUE,
'/PurchaseOrder/Reference')
FROM purchaseorder;

EXTRACTVALUE(OBJECT_VALUE, '/PURCHASEORDER/REFERENCE)
-------------------------------------------------------
SBELL-2002100912333601PDT

1 row selected.
```

Searching the XML Document Content Using ExistsNode

The SQL function `ExistsNode` evaluates whether or not a given document contains a node that matches a W3C XPath expression. Function `ExistsNode` returns true (`1`) if the document contains the node specified by the XPath expression supplied to the function and false (`0`) if it does not. Since XPath expressions can contain predicates, `ExistsNode` can determine whether or not a given node exists in the document, and whether or not a node with the specified value exists in the document. The functionality provided by SQL function `ExistsNode` is also available through `XMLType` method `ExistsNode`.

The example below searches XML Content Using ExistsNode. The query uses SQL function `ExistsNode` to check if the XML document contains an element named `Reference` that is a child of the root element `PurchaseOrder`:

```
SELECT COUNT(*) FROM purchaseorder WHERE ExistsNode(
OBJECT_VALUE, '/PurchaseOrder/Reference') = 1;

COUNT(*)
```

```
--------
  132

1 row selected.
```

XPath Rewrite for SQL/XML Query Functions

When `XMLType` data is stored in structured storage using an XML schema and XPath-based queries are used, they can potentially be rewritten directly to the underlying object-relational columns. This rewrite of queries can also potentially happen when queries using XPath are issued on certain non-schema-based `XMLType` views.

This enables the use of B*Tree or other indexes, if present on the column, to be used in query evaluation by the Optimizer. This XPath rewrite mechanism is used for XPaths in SQL functions such as `existsNode`, `extract`, `extractValue`, and `updateXML`. This enables the XPath to be evaluated against the XML document without having to ever construct the XML document in memory.

For example, a query such as the following tries to obtain the `Company` element and compare it with the literal '`Oracle`':

```
SELECT OBJECT_VALUE FROM mypurchaseorders p
  WHERE extractValue(OBJECT_VALUE,
'/PurchaseOrder/Company') = 'Oracle';
```

Because table `mypurchaseorders` was created with XML schema-based structured storage, `extractValue` is rewritten to the underlying relational column that stores the company information for the `purchaseOrder`. The query is rewritten to the following:

```
SELECT VALUE(p) FROM mypurchaseorders p WHERE
p.xmldata.Company = 'Oracle';
```

Rewrite for EXISTSNODE

SQL function `existsNode` returns one (`1`) if the XPath argument targets a nonempty sequence of nodes (text, element, or attribute); otherwise, it returns zero (`0`). The value is determined differently, depending on the kind of node targeted by the XPath argument:

- If the XPath argument targets a text node (using node test `text()`) or a `complexType` element node, Oracle XML DB simply checks whether the database representation of the element content is `NULL`.
- Otherwise, the XPath argument targets a `simpleType` element node or an attribute node.

In the example below, the query specifies an XPath expression with a predicate to check whether purchase order number 1001 contains a part with price greater than 2000:

```
SELECT count(*)
  FROM purchaseorder
  WHERE existsNode(OBJECT_VALUE,
'/PurchaseOrder[PONum=1001 and Item/Price > 2000]') = 1;
```

This is rewritten as something like the following:

```
SELECT count(*)
  FROM  purchaseorder p
  WHERE CASE WHEN p.XMLDATA."PONum" = 1001
    AND exists(SELECT NULL FROM table(XMLDATA."Item") p
               WHERE  p."Price" > 2000 )
           THEN 1
           ELSE 0
        END = 1;
```

This CASE expression is further optimized due to the constant relational equality expressions. The query becomes:

```
SELECT count(*)
  FROM purchaseorder p
  WHERE p.XMLDATA."PONum"=1001
   AND exists(SELECT NULL FROM table(p.XMLDATA."Item") x
              WHERE  x."Price" > 2000);
```

This rewritten query can be efficiently executed by the relational query engine. If Btree indexes are present on the Price and PONum columns, the query engine will can takes advantage of them during query execution.

Rewrite for EXTRACT and EXTRACTVALUE

SQL function extractValue is a shortcut for extracting text nodes and attributes using function extract and then using method getStringVal() or getNumberVal() to obtain the scalar content. Function extractValue returns the values of attribute nodes or the text nodes of elements with scalar values. Function extractValue cannot handle XPath expressions that return multiple values or complexType elements. If an XPath expression targets an element, then extractValue retrieves the text node of the element. For example, /PurchaseOrder/PONum and /PurchaseOrder/PONum/text() are handled identically by extractValue: both retrieve the scalar content of PONum.

In the example below, the XPath expression ni the extract function specifies a predicate to return PONum of purchase orders with Part greater than 2000.

```
SELECT extract(OBJECT_VALUE,
               '/PurchaseOrder[Item/Part > 2000]/PONum')
  FROM purchaseorder_table;
```

This query would be rewritten to the following for execution:

```
SELECT (SELECT CASE WHEN
node_exists(p.XMLDATA.SYS_XDBPD$, 'PONum')
            THEN XMLElement("PONum", p.XMLDATA."PONum")
          ELSE NULL END
```

```
         FROM DUAL
           WHERE exists(SELECT NULL FROM
table(XMLDATA."Item") p WHERE  p."Part" > 2000))
  FROM purchaseorder_table p;
```

Rewrite for XMLSequence

SQL function `XMLSequence` expose the members of a collection as a virtual table. You can use SQL function `XMLSequence` in conjunction with SQL functions `extract` and `table` to unnest XML collection values. When used with schema-based storage, these functions also get rewritten to access the underlying relational collection storage.

For example, the query below retrieves the price and part numbers of all items in a relational form:

```
SELECT extractValue(OBJECT_VALUE,
'/PurchaseOrder/PONum') AS ponum,
       extractValue(value(it), '/Item/Part') AS part,
       extractValue(value(it), '/Item/Price') AS price
  FROM purchaseorder,
       table(XMLSequence(extract(OBJECT_VALUE,
'/PurchaseOrder/Item'))) it;

PONUM PART                 PRICE
----- -------------------- ----------
      1001 9i Doc Set      2550
      1001 8i Doc Set      350
```

In this example, SQL function `extract` returns a fragment containing the list of `Item` elements. Function `XMLSequence` converts the fragment into a collection of `XMLType` values one for each `Item` element. Function `table` converts the elements of the collection into rows of `XMLType`. The XML data returned from `table` is used to extract the `Part` and the `Price` elements.

As shown in the explain plan below, during query processing, the functions `extract` and `XMLSequence` are rewritten to a simple `SELECT` operation from the `item_nested` nested table.

```
EXPLAIN PLAN
  FOR SELECT extractValue(OBJECT_VALUE,
'/PurchaseOrder/PONum') AS ponum,
     extractValue(value(it) , '/Item/Part') AS part,
     extractValue(value(it), '/Item/Price') AS price
  FROM purchaseorder,
       table(XMLSequence(extract(OBJECT_VALUE,
'/PurchaseOrder/Item'))) it;

Explained

PLAN_TABLE_OUTPUT
------------------------------------------------------
| Id  | Operation                         | Name        |
------------------------------------------------------
|   0 | SELECT STATEMENT                  |             |
|   1 |  NESTED LOOPS                     |             |
|   2 |   TABLE ACCESS FULL               | ITEM_NESTED |
|   3 |   TABLE ACCESS BY INDEX ROWID | PURCHASEORDER|
|*  4 |    INDEX UNIQUE SCAN          | SYS_C002973  |
-------------------------------------------------------

Predicate Information (identified by operation id)
--------------------------------------------------
   4 -
access("NESTED_TABLE_ID"="SYS_ALIAS_1"."SYS_NC0001100012
$")
```

The `EXPLAIN PLAN` output shows that the optimizer is able to use a simple nested-loops join between nested table `item_nested` and table `purchaseorder`.

Using Explain Plan for Performance Tuning

With Oracle's XPath rewrite implementation, XPath-based SQL/XML operations are rewritten to equivalent relational queries. This approach allows XML application developers and DBAs to use the same set of powerful tools they have been using for relational applications.

For performance tuning, the `EXPLAIN PLAN` statement displays execution plans chosen by the Oracle optimizer for `SELECT`, `UPDATE`, `INSERT`, and `DELETE` statements. A statement's execution plan is the sequence of operations Oracle performs to run the statement. The `EXPLAIN PLAN` results allow application developers and DBAs to determine whether the optimizer selects a particular execution plan, such as, nested loops join. It also helps you to understand the optimizer decisions, such as why the optimizer chose a nested loops join instead of a hash join, and lets you understand the performance of a query.

In the previous example, you can also query the `Item` values further and create appropriate indexes on the nested table to speed up such queries. For example, to search on the price to get all the expensive items, we could create an index on the `Price` column of the nested table. The following `EXPLAIN PLAN` confirms that the `price_index` is used to obtain the list of items and then joins with table `purchaseorder` to obtain the `PONum` value.

```
CREATE INDEX price_index ON item_nested ("Price");

Index created.

EXPLAIN PLAN FOR
  SELECT extractValue(OBJECT_VALUE,
'/PurchaseOrder/PONum') AS ponum,
         extractValue(value(it), '/Item/Part') AS part,
         extractValue(value(it), '/Item/Price') AS price
    FROM  purchaseorder,
          table(XMLSequence(extract(OBJECT_VALUE,
'/PurchaseOrder/Item'))) it
    WHERE extractValue(value(it), '/Item/Price') > 2000;

Explained.
```

```
PLAN_TABLE_OUTPUT
----------------------------------------------------------
| Id  | Operation                         | Name           |
----------------------------------------------------------
|   0 | SELECT STATEMENT                  |                |
|   1 |  NESTED LOOPS                     |                |
|   2 |   TABLE ACCESS BY INDEX ROWID | ITEM_NESTED    |
|*  3 |    INDEX RANGE SCAN            | PRICE_INDEX    |
|   4 |   TABLE ACCESS BY INDEX ROWID | PURCHASEORDER  |
|*  5 |    INDEX UNIQUE SCAN           | SYS_C002973    |
----------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("ITEM_NESTED"."Price">2000)
   5 -
access("NESTED_TABLE_ID"="SYS_ALIAS_1"."SYS_NC0001100012
$")
```

Usage Scenarios for XPath-Based SQL/XML Query Functions

With XPath-rewrites, query functions based on XPath can efficiently process simple path-oriented XML queries. As an extension to SQL, SQL/XML query functions can be used in conjunction with relational SQL queries to extract XML values and fragments in a complex query involving both XML and relational data.

ORACLE DATABASE-NATIVE XQUERY

XQuery 1.0 is an XML Query Language developed by W3C that will become the recommended query language to query XML from a variety of data sources. Various companies are adopting XQuery as the way to query XML stored in database rows or from WebServices and to construct new XML values.

On the SQL side, the XML datatype was introduced in SQL 2003 as a way to encapsulate XML in SQL. The SQL committee is now working to integrate the querying of XML using XQuery. This is being accomplished by introducing a new SQL function: *XMLQuery,* and a new construct: *XMLTable* both of which operate on XML and SQL values using XQuery. The former is known as **XQuery-centric** approach as it allows querying and constructing XML using XQuery. The latter is known as **SQL-centric** approach as it allows breaking apart the XQuery values into relational values.

Oracle Database 10g Release2 enables XQuery support in the database server through these SQL standard functions.

FLWOR Expressions in XQuery

At the heart of XQuery is the FLWOR expression that supports iteration and binding of variables to intermediate results. This kind of expression is often useful for computing joins between two or more documents and for restructuring data. The name FLWOR, pronounced "flower", reflects the keywords `for`, `let`, `where`, `order by`, and `return`.

Similar to the `FROM` clause in SQL, the `for` and `let` clauses in a FLWOR expression generate a sequence of tuples of bound variables called the **tuple stream**. Performing the same function as the `WHERE` clause in SQL, the `where` clause serves to filter the tuple stream, retaining some tuples and discarding others. The `order by` clause mimics the ORDER BY clause in SQL to impose an ordering on the tuple stream. Finally, the `return` clause works like the SELECT clause in SQL to construct the result of the FLWOR expression. The `return` clause is evaluated once for every tuple in the tuple stream, after filtering by the `where` clause, using the variable bindings in the respective tuples. The result of the FLWOR expression is an ordered sequence containing the concatenated results of these evaluations.

XMLQuery SQL function

The XMLQuery function takes an XQuery expression as a string literal, an optional context item and other bind variables and returns the result of evaluating the XQuery expression using these input values.

Below is the syntax that will be supported in Oracle Database 10g Release 2:

XMLQUERY (<XQuery-string-literal>

[PASSING [BY VALUE] <expression-returning-XMLType>]

RETURNING CONTENT)

The XQuery string literal is a complete XQuery expression including the prolog etc. The PASSING clause must be followed by an expression returning an XMLType that is used as the context for evaluating the XQuery expression.

To run XQuery on XMLType columns, tables, views, or expressions generated by SQL/XML functions, it is recommended that users pass the value as an argument to the XMLQuery function. However, to query any relational table or view as XML without having to first create SQL/XML views on top of them, users can use Oracle provided XQuery function: ora:view() within an XQuery expression.

XMLQuery Examples

Query on XMLType Column

You can also query on an XMLType column using the XMLQuery function. The XML column can be passed in through the PASSING clause as a context item. In this example, we are checking for the warehouses whose building area is greater than 100K. Note that since the XMLQuery function is applied to all rows in the FROM clause, we only get results for those rows where the XQuery expression matches the value.

```
SQL> connect oe/oe;
Connected
SQL> select warehouse_name,
            XMLQuery(
                'for $i in /Warehouse
                where $i/Area > 10000
                return <Details>
                          <Docks num="{$i/Docks}"/>
                          <Rail>
                             {
                                if ($i/RailAccess = "Y")
then "true" else "false"
                               }
                          </Rail>
                          </Details>' passing
warehouse_spec returning content) big_warehouses
from warehouses;

WAREHOUSE_NAME
-----------------------------------
BIG_WAREHOUSES
----------------------------------------------------
Southlake, Texas
<Details>
   <Docks num="2"/>
   <Rail>false</Rail>
</Details>

San Francisco
<Details>
   <Docks num="1"/>
   <Rail>false</Rail>
</Details>

New Jersey
<Details>
   <Docks num=""/>
   <Rail>false</Rail>
</Details>

Seattle, Washington
<Details>
   <Docks num="3"/>
   <Rail>true</Rail>
</Details>

Toronto

Sydney

Mexico City

Beijing

Bombay


9 rows selected.
```

Query XDB Repository

Assume that the XMLDB repository contains the *items.xml* XML document -

```
SQL> select any_path from resource_view where
equals_path(res,'/public/items.xml') = 1;

ANY_PATH
------------------------------------------------------
/public/items.xml

1 row selected.
```

Then the doc() function can be used inside XQuery to query this XML file. Assuming that items.xml contains information about some items that are being bid on, the following XQuery extracts out the various item tuples and orders them by their `itemno` and returns a new XML element containing the item number and it's description.

```
SQL> select XMLQuery(
  '<result> {
      for $i in doc ("/public/items.xml")//item_tuple
       order by $i/itemno
     return
       <item_tuple>
            { $i/itemno }
           { $i/description }
       </item_tuple>
 }
</result>'  returning content) as xml from dual;

XML
--------------------------------------------------------
<result>
   <item_tuple>
      <itemno>1001</itemno>
      <description>Red Bicycle</description>
   </item_tuple>
   <item_tuple>
      <itemno>1003</itemno>
      <description>Old Bicycle</description>
   </item_tuple>
   <item_tuple>
      <itemno>1007</itemno>
      <description>Racing Bicycle</description>
   </item_tuple>
   <item_tuple>
      <itemno>1008</itemno>
      <description>Broken Bicycle</description>
   </item_tuple>
</result>

1 row selected.
```

XMLTable SQL Construct

The XMLTable construct is used to map the result of an XQuery evaluation into relational rows and columns so that the user can query the XQuery result as a virtual relational table using SQL. The XMLTable construct can only be used in the from clause of SQL queries.

Below is the syntax that will be supported in Oracle Database 10g Release 2:

```
<XML table> ::=
  "XMLTable" "(" <XQuery-string-literal>
       ["PASSING" ["BY" "VALUE"] <xml-value-expression> ]
       ["COLUMNS" <XML-table-columns>]
              ")"

<XML-table-columns> ::= <XML-table-column>
                      ["," <XML-table-column>]…

<XML-table-column> ::= <column-name> [<data-type>]
 [PATH <string-literal>][ "DEFAULT" <value-expression> ]
```

In the example below, an `XMLTable` function is used to query a PurchaseOrder XMLType table. The parameter of the `XMLTable` function is an XQuery expression that finds XML documents with a specific Reference number and then returns the text values of the Description nodes.

```
SQL> SELECT x.column_value FROM purchaseorder, XMLTable
('for $i in $po where
  $i/PurchaseOrder/Reference="SBELL-2003030912333601PDT"
  return
$i/PurchaseOrder/LineItems/LineItem/Description/text()'
  passing OBJECT_VALUE as "po") x
/

COLUMN_VALUE
--------------------------------------------------------
A Night to Remember
The Unbearable Lightness Of Being
Sisters
A Night to Remember
The Unbearable Lightness Of Being
Sisters

6 rows selected.
```

Rewrite for XQuery

Similar to XPath-rewrite for XPath-based SQL/XML functions, Oracle's database-native XQuery implementation excels with extensive XQuery rewrites. XQuery rewrite allows Oracle's XQuery implementation to take full advantage of Oracle's high performance relational query engine. With XML documents stored using the structured storage approach, XQuery can be rewritten into pure relational queries to completely avoid building DOM trees of XML documents in memory. Query performance can be orders of magnitude faster with rewrites applied.

In the example below, the XQuery can be rewritten into an equivalent relational query to attain the same performance level as pure relational queries.

```
SELECT XMLQuery(
  'for $b in ora:view("SITE_TAB")/site/people/person
   where $b/@id = "person0"
   return $b/name' returning content)
FROM dual;

SELECT ( SELECT XMLAgg(XMLElement("name", p.name))
FROM SITE_TAB s, PERSON_TAB p
WHERE p.id ='person0' AND
      p.NESTED_TABLE_ID = s."SYS_NC0004700048$"
)
FROM dual;
```

Explain Plan Showing XQuery Rewrite

Similar to the benefits of rewrites for XPath-based SQL/XML functions, XQuery rewrites also allow application developers and DBAs to use the same set of powerful tools for relational application development. For XQuery performance tuning, the `EXPLAIN PLAN` statement can be used to display execution plans chosen by the Oracle optimizer for a rewritten XQuery statements.

In the example below, the XQuery uses a FLWOR construct to join the results from the ora:view() functions on the REGIONS and the COUNTRIES tables based on their REGION_ID element values.

```
SQL> SELECT XMLQuery(
     'for    $i in ora:view("REGIONS")/ROW
             $j in  ora:view("COUNTRIES")/ROW
      where  $i/REGION_ID = $j/REGION_ID and
             $i/REGION_NAME = "Asia"
      return $j' returning content) as asian_countries
     FROM DUAL;


ASIAN_COUNTRIES
--------------------------------------------------------
<ROW>
  <COUNTRY_ID>AU</COUNTRY_ID>
  <COUNTRY_NAME>Australia</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>CN</COUNTRY_ID>
  <COUNTRY_NAME>China</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>HK</COUNTRY_ID>
  <COUNTRY_NAME>HongKong</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>IN</COUNTRY_ID>
  <COUNTRY_NAME>India</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
```

```
  <COUNTRY_ID>JP</COUNTRY_ID>
  <COUNTRY_NAME>Japan</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>SG</COUNTRY_ID>
  <COUNTRY_NAME>Singapore</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
```

Since the XQuery really queries XML constructed from the underlying relational tables, this will get internally rewritten so that the entire XQuery becomes a simpler relational query (with some XML construction functions) with the XQuery predicates transformed into joins on the underlying tables. The simpler relational query is executed according to the optimal plan chosen by the cost-based optimizer. As shown below, the cost-based optimizer can take advantage of column indexes on the underlying tables.

Here is the result of using an explain plan:

```
SQL> explain plan for
  select XMLQuery(
     'for $i in ora:view("REGIONS"),
          $j in  ora:view("COUNTRIES")
      where $i/REGION_ID = $j/REGION_ID and
            $i/REGION_NAME = "Asia"
      return $j' returning content) as asian_countries
  from dual;

Explained

SQL> @?/rdbms/admin/utlxpls

PLAN_TABLE_OUTPUT
----------------------------------------------------------------------
| Id  | Operation                      | Name            | Rows | Cost|
----------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                 |    1 |    2|
|   1 |  SORT AGGREGATE                |                 |    1 |     |
|   2 |   NESTED LOOPS                 |                 |    6 |    6|
|   3 |    MERGE JOIN CARTESIAN        |                 |   25 |    5|
|   4 |     MERGE JOIN CARTESIAN       |                 |    1 |    4|
|   5 |      FAST DUAL                 |                 |    1 |    2|
|   6 |      FAST DUAL                 |                 |    1 |    2|
|   7 |     INDEX FULL SCAN            | COUNTRY_C_ID_PK |   25 |    1|
|*  8 |    TABLE ACCESS BY INDEX ROWID| REGIONS         |    1 |    1|
|*  9 |     INDEX UNIQUE SCAN          | REG_ID_PK       |    1 |    0|
|  10 |     FAST DUAL                  |                 |    1 |    2|
----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   8 - filter("REGION_NAME"='Asia')
   9 - access("REGION_ID"="REGION_ID")
```

Usage Scenarios for Database-Native XQuery

Comparing with XPath-based query functions, XQuery-based queries can use FLWOR expressions to perform complex joins between XML documents and to transform the structure of XML data. With Oracle’s implementation of the

XMLQuery SQL function and the XMLTable SQL construct based on the upcoming SQL 2005 standard, complex XQuery can join force with relational SQL to optimally perform complex queries over both XML and relational data.

CONCLUSION

XML DB query capabilities in Oracle Database 10g Release 2 provide the most comprehensive and efficient functionality for versatile, scalable, concurrent, and high performance XML applications.

With rapidly multiplying volumes of XML data, XML queries can no longer be carried out by building resource-hungry DOM trees in memory for functional evaluation of XML documents. Oracle has made a major breakthrough in its XML DB product to holistically process XML data with higher efficiency in both space and time.

Using structured storage of XML documents along with XPath- and XQuery-rewrite of SQL/XML query functions, XML DB can handle both simple XPath-based queries and complex XQuery-based queries with orders of magnitude performance improvement over functional evaluation of XML queries. Furthermore, XML queries can be seamlessly merged with SQL relational queries to handle all query scenarios. Finally, the XML query capabilities of Oracle XML DB are built on the solid foundation of industry's best object-relational database that is highly reliable, available, scalable, and secure. In short, the XML DB query capabilities in Oracle Database 10g Release 2 provide the most comprehensive and efficient functionality for versatile, scalable, concurrent, and high performance XML applications.

Mastering XML DB Queries in Oracle Database 10g Release 2
March 2005
Author: Geoff Lee

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com