# SQL 2003 Standard Support in Oracle Database 10g

*An Oracle White Paper*
*November 2003*

ORACLE

# SQL 2003 Standard Support in Oracle Database 10g

# SQL 2003 Standard Support in Oracle Database 10g

## INTRODUCTION

SQL standards have been the most popular and enduring standards in computing. Although information processing has become increasingly more sophisticated and more complex over the past decades, SQL has continually evolved to meet the growing demands.

As the latest version of SQL standards, SQL 2003 is making major improvements in a number of key areas. First, there are additional object-relational features, which were first introduced in SQL 1999. Second, SQL 2003 standard revolutionizes SQL with comprehensive OLAP features. Third, SQL 2003 delivers a brand new Part 14 for XML-Related Specifications (SQL/XML) to integrate popular XML standards into SQL. Finally, there are numerous improvements throughout the SQL 2003 standard to refine existing features.

**The latest Oracle Database 10g supports additional SQL 2003 new features as well as several new SQL capabilities beyond the current SQL 2003 standard.**

As the first commercial implementation of SQL over 25 years ago, Oracle continues to lead the database industry in implementing SQL standards. In fact, many of the SQL 2003 new features had already been supported since Oracle Database 8/8i (Multisets, OLAP functions, etc.), Oracle Database 9i (additional OLAP functions, Table functions, Nested collection types, Final structured types, etc.) or Oracle Database 9i Release 2 (SQL/XML features). The latest Oracle Database 10g supports additional SQL 2003 new features (advanced Multiset support) as well as several new SQL capabilities beyond the current SQL 2003 standard (e.g., additional statistical functions, regular expressions), making Oracle Database 10g the best implementation of SQL standards.

In the next sections, we will describe the following three key categories of SQL 2003 new features supported in Oracle databases.

- Object-Relational features
- Business Intelligence features
- SQL/XML features

## OBJECT-RELATIONAL FEATURES

With increasingly more complex eCommerce and eBusiness applications, application development adopted object-oriented approach to simplify and to manage complexity. Object-oriented programming languages (e.g., Java, C++)

Oracle has supported comprehensive Object-Relational features since Oracle8/8i Database releases.

were also chosen to implement these new applications. Traditional relational constructs fall short of the needs of object-oriented application developers. Addressing the growing demands, SQL 1999 standard introduced extensive object-relational features to simplify the storage and retrieval of complex-structured application objects. Oracle has supported comprehensive Object-Relational features since Oracle8/8i Database releases. SQL 2003 standard makes further improvements in this area with the following new features,

- Multiset support: Multiset type is a newly defined Collection type for an unordered collection. Since Oracle8, Multiset type has been supported in Oracle databases as Nested Table datatype.
- Advanced Multiset support: SQL 2003 defines advanced Multiset support with Comparison and Set operators (e.g., UNION, INTERSECTION). Oracle Database 10g provides comprehensive support of these Multiset operators for Nested Tables.
- Nested Collection Types: SQL 2003 defines two collection types, namely the Array type and the Multiset type. In addition to Nested Table datatype, Oracle supports the Array type as the Varray datatype since Oracle8. Nested collection of Varrays and Nested Tables have been supported in Oracle databases since Oracle9i.
- Final Structured Types: A User Defined Type (UDT) can be created with either FINAL or NOT FINAL option to indicate whether subtypes can inherit from it. Oracle databases have supported this feature since Oracle9i as part of its comprehensive Type Inheritance support.

In the following section, we will describe Multiset Comparison and Set operations in more detail. In depth information about other features can be found in current Oracle database documentation online at http://otn.oracle.com/documentation/index.html.

### ANSI SQL Standard Multiset Operations for Nested Tables

New in Oracle Database 10g, a number of Multiset operators are now supported for the Nested Table collection type. Real world applications use collection types to model containment relationships. Comparison and Set operators for collection types provide powerful tools for these applications. Oracle supports two collection datatypes, VARRAYs and Nested Tables.

A nested table is an unordered set of data elements, all of the same datatype. No maximum is specified in the definition of the table and the order of the elements is not preserved. Elements of a nested table are actually stored in a separate storage table that contains a column that identifies the parent table row or object to which each element belongs. A nested table has a single column, and the type of that column is a built-in type or an object type. If the column in a nested table is an object type, the table can also be viewed as a multi-column table, with a column for each attribute of the object type.

### Comparisons of Nested Tables

#### *Equal and Not Equal Comparisons*

The equal (=) and not equal (<>) conditions determine whether the input nested tables are identical or not, returning the result as a boolean value.

Two nested tables are equal if they have the same named type, have the same cardinality, and their elements are equal. Elements are equal depending on whether they are equal by the elements own equality definitions, except for object types which require a map method.

For example:

```
CREATE TYPE person_typ AS OBJECT (
  idno            NUMBER,
  name            VARCHAR2(30),
  phone           VARCHAR2(20),
  MAP MEMBER FUNCTION get_idno RETURN NUMBER );
/

CREATE TYPE BODY person_typ AS
  MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
  BEGIN
    RETURN idno;
  END;
END;
/

CREATE TYPE people_typ AS TABLE OF person_typ;
/
CREATE TABLE students (
   graduation DATE,
   math_majors people_typ,
   chem_majors people_typ,
   physics_majors people_typ)
  NESTED TABLE math_majors STORE AS math_majors_nt
  NESTED TABLE chem_majors STORE AS chem_majors_nt
  NESTED TABLE physics_majors STORE AS physics_majors_nt;

INSERT INTO students (graduation) VALUES ('01-JUN-03');
UPDATE students
  SET math_majors =
    people_typ (person_typ(12, 'Bob Jones', '111-555-1212'),
                person_typ(31, 'Sarah Chen', '111-555-2212'),
                person_typ(45, 'Chris Woods', '111-555-1213')),
   chem_majors =
    people_typ (person_typ(51, 'Joe Lane', '111-555-1312'),
                person_typ(31, 'Sarah Chen', '111-555-2212'),
                person_typ(52, 'Kim Patel', '111-555-1232')),
   physics_majors =
    people_typ (person_typ(12, 'Bob Jones', '111-555-1212'),
                person_typ(45, 'Chris Woods', '111-555-1213'))
WHERE graduation = '01-JUN-03';
```

```
SELECT p.name FROM students, TABLE(physics_majors) p
WHERE math_majors = physics_majors;

no rows selected
```

In this example, the nested tables contain `person_typ` objects which have an associated map method.

#### *In Comparisons*

The `IN` condition checks whether a nested table is in a list of nested tables, returning the result as a boolean value. NULL is returned if the nested table is a null nested table.

For example:

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors IN (math_majors, chem_majors);

no rows selected
```

#### *Subset of Multiset Comparison*

The `SUBMULTISET [OF]` condition checks whether a nested table is a subset of a another nested table, returning the result as a boolean value. The `OF` keyword is optional and does not change the functionality of `SUBMULTISET`.

This operator is implemented only for nested tables because this is a multiset function only.

For example:

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors SUBMULTISET OF math_majors;

      IDNO NAME
---------- ------------------------------
        12 Bob Jones
        45 Chris Woods
```

#### *Member of a Nested Table Comparison*

The `MEMBER [OF]` or `NOT MEMBER [OF]` condition tests whether an element is a member of a nested table, returning the result as a boolean value. The `OF` keyword is optional and has no effect on the output.

For example:

```
SELECT graduation FROM students WHERE person_typ(12, 'Bob
Jones', '1-800-555-1212') MEMBER OF math_majors;

GRADUATION
----------
01-JUN-03
```

where `person_typ (12, 'Bob Jones', '1-800-555-1212')` is an element of the same type as the elements of the nested table `math_majors`.

#### *Empty Comparison*

The `IS [NOT] EMPTY` condition checks whether a given nested table is empty or not empty, regardless of whether any of the elements are NULL. If a NULL is given for the nested table, the result is NULL. The result is returned as a boolean value.

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors IS NOT EMPTY;

      IDNO NAME
---------- ------------------------------
        12 Bob Jones
        45 Chris Woods
```

#### *Set Comparison*

The `IS [NOT] A SET` condition checks whether a given nested table is composed of unique elements, returning a boolean value.

For example:

```
SELECT p.idno, p.name
  FROM students, TABLE(physics_majors) p
WHERE physics_majors IS A SET;

      IDNO NAME
---------- ------------------------------
        12 Bob Jones
        45 Chris Woods
```

### Multisets Operations

#### *CARDINALITY*

The `CARDINALITY` function returns the number of elements in a varray or nested table. The return type is `NUMBER`. If the varray or nested table is a null collection, `NULL` is returned.

For example:

```
SELECT CARDINALITY(math_majors)
  FROM students;

CARDINALITY(MATH_MAJORS)
------------------------
                       3
```

#### *COLLECT*

The `COLLECT` function is an aggregate function which would create a multiset from a set of elements. The function would take a column of the element type as input and create a multiset from rows selected. To get the results of this function you must use it within a CAST function to specify the output type of COLLECT.

#### *MULTISET EXCEPT*

The MULTISET EXCEPT operator inputs two nested tables and returns a nested table whose elements are in the first nested table but not in the second nested table. The input nested tables and the output nested table are all type name equivalent.

The ALL or DISTINCT options can be used with the operator. The default is ALL.

With the ALL option, for ntab1 MULTISET EXCEPT ALL ntab2, all elements in ntab1 other than those in ntab2 would be part of the result. If a particular element occurs *m* times in ntab1 and *n* times in ntab2, the result will have (*m* - *n*) occurrences of the element if *m* is greater than *n* otherwise 0 occurrences of the element.

With the DISTINCT option, any element that is present in ntab1 which is also present in ntab2 would be eliminated, irrespective of the number of occurrences.

For example:

```
SELECT math_majors MULTISET EXCEPT physics_majors
  FROM students WHERE graduation = '01-JUN-03';

MATH_MAJORSMULTISETEXCEPTPHYSICS_MAJORS(IDNO, NAME, PHONE)
---------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(31, 'Sarah Chen', '111-555-2212'))
```

#### *MULTISET INTERSECTION*

The MULTISET INTERSECT operator returns a nested table whose values are common in the two input nested tables. The input nested tables and the output nested table are all type name equivalent.

There are two options associated with the operator: ALL or DISTINCT. The default is ALL. With the ALL option, if a particular value occurs *m* times in ntab1 and *n* times in ntab2, the result would contain the element MIN(*m*, *n*) times. With the DISTINCT option the duplicates from the result would be eliminated, including duplicates of NULL values if they exist.

For example:

```
SELECT math_majors MULTISET INTERSECT physics_majors
  FROM students
WHERE graduation = '01-JUN-03';

MATH_MAJORSMULTISETINTERSECTPHYSICS_MAJORS(IDNO, NAME, PHONE)
--------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '111-555-1212'),
           PERSON_TYP(45, 'Chris Woods', '111-555-1213'))
```

### *MULTISET UNION*

The `MULTISET UNION` operator returns a nested table whose values are those of the two input nested tables. The input nested tables and the output nested table are all type name equivalent.

There are two options associated with the operator: `ALL` or `DISTINCT`. The default is `ALL`. With the `ALL` option, all elements that are in `ntab1` and `ntab2` would be part of the result, including all copies of `NULL`s. If a particular element occurs *m* times in `ntab1` and *n* times in `ntab2`, the result would contain the element ($m + n$) times. With the `DISTINCT` option the duplicates from the result are eliminated, including duplicates of `NULL` values if they exist.

For example:

```
SELECT math_majors MULTISET UNION DISTINCT physics_majors
  FROM students
WHERE graduation = '01-JUN-03';

MATH_MAJORSMULTISETUNIONDISTINCTPHYSICS_MAJORS(IDNO,
NAME, PHONE)
---------------------------------------------------------

PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '111-555-1212'),
           PERSON_TYP(31, 'Sarah Chen', '111-555-2212'),
           PERSON_TYP(45, 'Chris Woods', '111-555-1213'))

SELECT math_majors MULTISET UNION ALL physics_majors
  FROM students
WHERE graduation = '01-JUN-03';

MATH_MAJORSMULTISETUNIONALLPHYSICS_MAJORS(IDNO, NAME,
PHONE)
---------------------------------------------------------

PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '111-555-1212'),
           PERSON_TYP(31, 'Sarah Chen', '111-555-2212'),
           PERSON_TYP(45, 'Chris Woods', '111-555-1213'),
           PERSON_TYP(12, 'Bob Jones', '111-555-1212'),
           PERSON_TYP(45, 'Chris Woods', '111-555-1213'))
```

### *POWERMULTISET*

The `POWERMULTISET` function generates all non-empty submultisets from a given multiset. The input to the `POWERMULTISET` function could be any expression which evaluates to a multiset. The limit on the cardinality of the multiset argument is 32.

For example:

```
SELECT * FROM TABLE(POWERMULTISET( people_typ (
           person_typ(12, 'Bob Jones', '1-800-555-1212'),
           person_typ(31, 'Sarah Chen', '1-800-555-2212'),
           person_typ(45, 'Chris Woods', '1-800-555-1213'))));
```

```
COLUMN_VALUE(IDNO, NAME, PHONE)
---------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'))
PEOPLE_TYP(PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'))
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'),
           PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'))
PEOPLE_TYP(PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'))
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'),
           PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'))
PEOPLE_TYP(PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'),
           PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'))
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'),
           PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'),
           PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'))

7 rows selected.
```

#### *POWERMULTISET_BY_CARDINALITY*

The `POWERMULTISET_BY_CARDINALITY` function returns all non-empty submultisets of a nested table of the specified cardinality. The output would be rows of nested tables.

`POWERMULTISET_BY_CARDINALITY(x, l)` is equivalent to `TABLE (POWERMULTISET(x)) p` where `CARDINALITY(value(p)) = l`, where *x* is a multiset and l is the specified cardinality.

The first input parameter to the `POWERMULTISET_BY_CARDINALITY` could be any expression which evaluates to a nested table. The length parameter should be a positive integer, otherwise an error will be returned. The limit on the cardinality of the nested table argument is 32.

For example:

```
SELECT * FROM TABLE(POWERMULTISET_BY_CARDINALITY( people_typ (
     person_typ(12, 'Bob Jones', '1-800-555-1212'),
     person_typ(31, 'Sarah Chen', '1-800-555-2212'),
     person_typ(45, 'Chris Woods', '1-800-555-1213')),2));

COLUMN_VALUE(IDNO, NAME, PHONE)
--------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'),
PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'))

PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '1-800-555-1212'),
PERSON_TYP(45, 'Chris Woods', '1-800-555-1213'))

PEOPLE_TYP(PERSON_TYP(31, 'Sarah Chen', '1-800-555-2212'),
PERSON_TYP(45, 'Chris  Woods', '1-800-555-1213'))
```

#### *SET*

The `SET` function converts a nested table into a set by eliminating duplicates, and returns a nested table whose elements are `DISTINCT` from one another. The nested table returned is of the same named type as the input nested table.

For example:

```
SELECT SET(physics_majors)
  FROM students
WHERE graduation = '01-JUN-03';

SET(PHYSICS_MAJORS)(IDNO, NAME, PHONE)
--------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(12, 'Bob Jones', '111-555-1212'),
           PERSON_TYP(45, 'Chris Woods', '111-555-1213'))
```

## BUSINESS INTELLIGENCE FEATURES

**Oracle was one of the collaborators working on defining these SQL 2003 new OLAP features for the past few years, and had provided complete and highly optimized support of these OLAP features since Oracle Database 9i.**

As information proliferated and sizes of databases continued to grow, enterprises were eager to gain precious insights from information. Data warehousing, OLAP, and Data Mining became essential tools for enterprises to extract business intelligence. In the past, OLAP functions were performed outside of databases in a separate OLAP server using non-standard APIs. It became apparent that the bottleneck of OLAP applications was in moving huge amount of data between the database server and the OLAP server. Productivity of OLAP application developers also suffered from non-standard APIs. The simple solution to these problems is to process OLAP functions inside the database server using standard SQL functions. To this end, ANSI SQL committee published in year 2000 an amendment to SQL 1999 standard defining an extensive list of OLAP functions, which are now part of the SQL 2003 new features. Oracle was one of the collaborators working on defining these SQL 2003 new OLAP features for the past few years, and had provided complete and highly optimized support of these OLAP features since Oracle Database 9i. There are additional new features in SQL 2003 defined specifically for data warehousing applications. Below is a list of these new features:

- OLAP functions: These new features have been implemented since Oracle8i or Oracle9i consisting of
  - Window functions: SQL 2003 defines aggregates computed over a window with ROW_NUMBER function, rank functions (i.e., RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST), and aggregate functions (e.g., inverse distribution, hypothetical set function)
  - NULLS FIRST, NULLS LAST in ORDER BY clause: This clause indicates the position of `NULLs` in the ordered sequence, either first or last in the sequence.
- Table functions: A table function is defined as a function that can produce a set of rows as output. Since Oracle9i, table functions have provided the support for pipelined and parallel execution of transformations implemented in PL/SQL, C, or Java. Scenarios as mentioned above can be done without requiring the use of intermediate staging tables, which interrupt the data flow through various steps of transformations.

## Examples of OLAP Functions in Oracle Databases

This section describes the key features of the analytic functions introduced in Oracle database with examples. Most of the examples involve data from a sales table, where each row contains detail or aggregated sales data. Complete information of OLAP functions in Oracle databases can be found in Oracle database documentation online at http://otn.oracle.com/documentation/index.html.

### Inverse Percentile Family

One very common analytic question is to find the value in a data set that corresponds to a specific percentile. For example, what if we ask "what value is the median (50th percentile) of my data?" This question is the inverse of the information provided by CUME_DIST function, which answers the question "what is the percentile value for each row?" Two new Oracle9*i* functions, PERCENTILE_CONT and PERCENTILE_DISC, compute inverse percentile. These functions require a sort specification and a percentile parameter value between 0 and 1. For instance, if a user needs the median value of income data, he would specify that the data set be sorted on income, and specify a percentile value of 0.5. The functions can be used as either aggregate functions or reporting aggregate functions. When used as aggregate functions, they return a single value per ordered set; and when used as reporting aggregate functions, they repeat the data on each row. PERCENTILE_DISC function returns the actual "discrete" value which is closest to the specified percentile values while the PERCENTILE_CONT function calculates a "continuous" percentile value using linear interpolation. The functions use a new WITHIN GROUP clause to specify the data ordering.

***Example of Inverse Percentile as an aggregate function:***

Consider the table HOMES which has the following data:

| Area | Address | Price |
|---|---|---|
| Uptown | 15 Peak St | 456,000 |
| Uptown | 27 Primrose Path | 349,000 |
| Uptown | 44 Shady Lane | 341,000 |
| Uptown | 23301 Highway 64 | 244,000 |
| Uptown | 34 Design Rd | 244,000 |
| Uptown | 77 Sunset Strip | 102,000 |
| Downtown | 72 Easy St | 509,000 |
| Downtown | 29 Wire Way | 402,000 |
| Downtown | 45 Diamond Lane | 203,000 |
| Downtown | 76 Blind Alley | 201,000 |
| Downtown | 15 Tern Pike | 199,000 |
| Downtown | 444 Kanga Rd | 102,000 |

To find the average and median value for each area, we could use the query below:

```
SELECT Homes.Area, AVG(Homes.Price),
PERCENTILE_DISC (0.5) WITHIN GROUP (ORDER BY Homes.Price DESC),
PERCENTILE_CONT (0.5) WITHIN GROUP (ORDER BY Homes.Price DESC)
FROM Homes  GROUP BY Area;
```

| Area | AVG | PERCENTILE_DISC | PERCENTILE_CONT |
|---|---|---|---|
| Uptown | 289,333 | 341,000 | 292,500 |
| Downtown | 269,333 | 203,000 | 202,000 |

In the example above, to compute the median price of homes, the data is ordered on home price within each area and the median price is computed using the "discrete" and "interpolation" methods. The results above show how PERCENTILE_DISC returns actual values from the table, while PERCENTILE_CONT calculates new interpolated values.

***Example of Inverse Percentile as a reporting function:***

The inverse percentile functions can be used as reporting functions. In that usage they will return a value that is repeated on multiple rows, allowing easy calculations. Consider the same HOMES table shown above. To find the median value for each area and display the results as a reporting function , we could use the query below:

```
SELECT Homes.Area, Homes.Price,
PERCENTILE_DISC (0.5) WITHIN GROUP (ORDER BY Homes.Price DESC)
OVER (PARTITION BY Area),
PERCENTILE_CONT (0.5) WITHIN GROUP (ORDER BY Homes.Price DESC)
OVER (PARTITION BY Area)
FROM Homes;
```

| Area | Price | PERCENTILE_DISC | PERCENTILE_CONT |
|---|---|---|---|
| Uptown | 456,000 | 341,000 | 292,500 |
| Uptown | 349,000 | 341,000 | 292,500 |
| Uptown | 341,000 | 341,000 | 292,500 |
| Uptown | 244,000 | 341,000 | 292,500 |
| Uptown | 244,000 | 341,000 | 292,500 |
| Uptown | 102,000 | 341,000 | 292,500 |
| Downtown | 509,000 | 203,000 | 202,000 |
| Downtown | 402,000 | 203,000 | 202,000 |
| Downtown | 203,000 | 203,000 | 202,000 |
| Downtown | 201,000 | 203,000 | 202,000 |
| Downtown | 199,000 | 203,000 | 202,000 |
| Downtown | 102,000 | 203,000 | 202,000 |

#### Hypothetical Rank and Distribution Family

In certain analyses, such as financial planning, we may wish to know how a data value would rank if it were added to our data set. For instance, if we hired a worker at a salary of $50,000, where would his salary rank compared to the other salaries at our firm? The hypothetical rank and distribution functions support this form of what-if analysis: they return the rank or percentile value which a row would be assigned if the row was hypothetically inserted into a set of other rows. The hypothetical functions can calculate RANK, DENSE_RANK, PERCENT_RANK and CUME_DIST. Like the inverse percentile functions, the hypothetical rank and distributions functions use a WITHIN GROUP clause containing an ORDER BY specification.

***Example of Hypothetical Rank function:***

Here is a query using our real estate data introduced above. It finds the hypothetical ranks and distributions for a house with a price of $400,000.

```
SELECT Area,
  RANK (400000) WITHIN GROUP (ORDER BY Price DESC),
  DENSE_RANK (400000) WITHIN GROUP (ORDER BY Price DESC),
  PERCENT_RANK (400000) WITHIN GROUP (ORDER BY Price
DESC),
  CUME_DIST (400000) WITHIN GROUP (ORDER BY Price DESC)
FROM Homes  GROUP BY Area;
```

| Area | RANK | DENSE_RANK | PERCENT _RANK | CUME_ DIST |
|---|---|---|---|---|
| Uptown | 2 | 2 | 0.166 | 0.285 |
| Downtown | 3 | 3 | 0.333 | 0.428 |

Unlike Inverse Percentile functions, Hypothetical functions cannot be used as a reporting function.

### SQL/XML FEATURES

**. Since Oracle Database 9i Release 2, SQL/XML features had been supported as an integral part of the XML DB.**

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. in 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 SQL extensions to support querying, updating, and transformation of XML data. Oracle is working closely with the SQL standard committee to standardize these extensions. Below is a list of SQL 2003 standard functions for generating XML from SQL data.

- XMLElement takes an element name, an optional collection of attributes for the element, and arguments that make up the element's content and returns an instance of type XMLType.
- XMLForest converts each of its argument parameters to XML, and then returns an XML fragment that is the concatenation of these converted arguments
- XMLConcat takes as input a series of XMLType instances, concatenates the series of elements for each row, and returns the concatenated series. XMLConcat is the inverse of XMLSequence.
- XMLAgg takes a collection of XML fragments and returns an aggregated XML document

Oracle Database 9i Release 2 XML DB extensions below provide additional capabilities to generate, query, update, and transform XML data.

- XMLColAttVal creates an XML fragment and then expands the resulting XML so that each XML fragment has the name "column" with the attribute "name".
- XMLSequence has two forms: the first form takes as input an XMLType instance and returns a varray of the top-level nodes in the XMLType. The second form takes as input a REFCURSOR instance, with an optional instance of the XMLFormat object, and returns as an XMLSequence type an XML document for each row of the cursor.
- ExtractValue takes as arguments an XMLType instance and an XPath expression and returns a scalar value of the resultant node.
- Extract (XML) takes as arguments an XMLType instance and an XPath expression and returns an XMLType instance containing an XML fragment.
- UpdateXML updates a value of the XML document and then replacing the whole document with the newly updated document.
- XMLTransform takes as arguments an XMLType instance and an XSL style sheet, which is itself a form of XMLType instance. It applies the style sheet to the instance and returns an XMLType instance.

## Generating XML from SQL Data Using SQL/XML Functions

#### XMLElement() Function

`XMLElement`() function is based on the emerging SQL XML standard. It takes an element name, an optional collection of attributes for the element, and zero or more arguments that make up the element content and returns an instance of type `XMLType`

It is similar to `SYS_XMLGEN()`, but unlike `SYS_XMLGEN()`, `XMLElement()` does not create an XML document with the prolog (the XML version information). It allows multiple arguments and can include attributes in the XML returned.

`XMLElement()` is primarily used to construct XML instances from relational data. It takes an identifier that is *partially escaped* to give the name of the root XML element to be created. The identifier does not have to be a column name, or column reference, and cannot be an expression. If the identifier specified is NULL, then no element is returned.

As part of generating a valid XML element name from a SQL identifier, characters that are disallowed in an XML element name are escaped. With p*artial escaping* the SQL identifiers other than the `":"` sign that are not representable in XML, are preceded by an escape character using the # sign followed by the unicode representation of that character in hexadecimal format. This can be used to specify namespace prefixes for the elements being generated. The *fully escaped* mapping escapes all non-XML characters in the SQL identifier name, including the `":"` character.

#### *XMLAttributes Clause*

`XMLElement()` also takes an optional `XMLAttributes()` clause, which specifies the attributes of that element. This can be followed by a list of values that make up the children of the newly created element.

In the `XMLAttributes()` clause, the value expressions are evaluated to get the values for the attributes. For a given value expression, if the AS clause is omitted, the fully escaped form of the column name is used as the name of the attribute. If the AS clause is specified, then the partially escaped form of the alias is used as the name of the attribute. If the expression evaluates to NULL, then no attribute is created for that expression. The type of the expression cannot be an object type or collection.

The list of values that follow the `XMLAttributes()` clause are converted to XML format, and are made as children of the top-level element. If the expression evaluates to NULL, then no element is created for that expression.

The following example illustrates the use of namespaces to create an XML schema-based document. Assuming that an XML schema "`http://www.oracle.com/Employee.xsd`" exists and has no target namespace, then the following query creates an `XMLType` instance conforming to that schema:

```
SELECT XMLELEMENT ( "Employee", X
MLATTRIBUTES ( 'http://www.w3.org/2001/XMLSchema' AS
"xmlns:xsi", 'http://www.oracle.com/Employee.xsd' AS
"xsi:nonamespaceSchemaLocation" ),
XMLForest(empno, ename, sal)) AS "result"
   FROM scott.emp WHERE deptno = 100;
```

This creates an XML document that conforms to the `Employee.xsd` XMLSchema, result:

```
<Employee xmlns:xsi="http://www.w3.org/2001/XMLSchema"
          xsi:nonamespaceSchemaLocation="http://www.oracl
e.com/Employee.xsd">
   <EMPNO>1769</EMPNO>
   <ENAME>John</ENAME>
   <SAL>200000</SAL>
</Employee>
```

#### XMLForest() Function

`XMLForest()` function produces a forest of XML elements from the given list of arguments. The arguments may be value expressions with optional aliases.

The list of value expressions are converted to XML format. For a given expression, if the AS clause is omitted, then the fully escaped form of the column name is used as the name of the enclosing tag of the element.

For an object type or collection, the AS clause is mandatory. For other types, the AS clause can be optionally specified. If the AS clause is specified, then the partially escaped form of the alias is used as the name of the enclosing tag. If the expression evaluates to NULL, then no element is created for that expression.

The following example generates an `Emp` element for each employee, with a `name` attribute and elements with the employee's start date and department as the content.

```
SELECT XMLELEMENT("Emp", XMLATTRIBUTES ( e.fname ||' '||
e.lname AS "name" ),  XMLForest ( e.hire, e.dept AS
"department")) AS "result" FROM employees e;
```

This query can produce the following XML result:

```
<Emp name="John Smith">
  <HIRE>2000-05-24</HIRE>
  <department>Accounting</department>
</Emp>
<Emp name="Mary Martin">
  <HIRE>1996-02-01</HIRE>
  <department>Shipping</department>
</Emp>
```

#### XMLConcat() Function

`XMLConcat()` function concatenates all the arguments passed in to create a XML fragment. `XMLConcat()` has two forms:

- The first form takes an `XMLSequenceType`, which is a VARRAY of `XMLType` and returns a single `XMLType` instance that is the concatenation of all of the elements of the varray. This form is useful to collapse lists of `XMLTypes` into a single instance.

- The second form takes an arbitrary number of `XMLType` values and concatenates them together. If one of the value is null, then it is ignored in the result. If all the values are NULL, then the result is NULL. This form is used to concatenate arbitrary number of `XMLType` instances in the same row. `XMLAgg()` can be used to concatenate `XMLType` instances across rows.

The example below shows `XMLConcat()` returning the concatenation of `XMLType`s from the `XMLSequence function`:

```
SELECT XMLConcat(XMLSequence(
   xmltype('<PartNo>1236</PartNo>'),
   xmltype('<PartName>Widget</PartName>'),
   xmltype('<PartPrice>29.99</PartPrice>'))).getClobVal()
FROM dual;
```

returns a single fragment of the form:

```
<PartNo>1236</PartNo>
<PartName>Widget</PartName>
<PartPrice>29.99</PartPrice>
```

#### XMLAgg() Function

`XMLAgg()` is an aggregate function that produces a forest of XML elements from a collection of XML elements.

As with `XMLConcat()`, any arguments that are null are dropped from the result. This function can be used to concatenate `XMLType` instances across multiple rows. It also allows an optional ORDER BY clause to order the XML values being aggregated.

`XMLAgg()` is an aggregation function and hence produces one aggregated XML result for each group. If there is no group by specified in the query, then it returns a single aggregated XML result for all the rows of the query.

The following example produces a `Department` element containing `Employee` elements with employee job ID and last name as the contents of the elements. It also orders the employee XML elements in the department by their last name.

```
SELECT XMLELEMENT("Department", XMLAGG(
XMLELEMENT("Employee", e.job_id||' '||e.last_name)
ORDER BY last_name)) as "Dept_list"
FROM employees e WHERE e.department_id = 30;

Dept_list
-------------------------------------------------------
<Department>
  <Employee>PU_CLERK Baida</Employee>
  <Employee>PU_CLERK Colmenares</Employee>
  <Employee>PU_CLERK Himuro</Employee>
  <Employee>PU_CLERK Khoo</Employee>
  <Employee>PU_MAN Raphaely</Employee>
  <Employee>PU_CLERK Tobias</Employee>
</Department>
```

The result is a single row, because `XMLAgg()` aggregates the rows. You can use the `GROUP BY` clause to group the returned set of rows into multiple groups:

```
SELECT XMLELEMENT("Department",
  XMLAttributes(department_id AS deptno),
  XMLAGG(XMLELEMENT("Employee", e.job_id||' '||
e.last_name))) AS "Dept_list"
   FROM employees e
   GROUP BY e.department_id;

Dept_list
-------------------------------------------------------
<Department deptno="1001">
  <Employee>AD_ASST Whalen</Employee>
</Department>

<Department deptno="2002">
  <Employee>MK_MAN Hartstein</Employee>
  <Employee>MK_REP Fay</Employee>
</Department>

<Department deptno="3003">
  <Employee>PU_MAN Raphaely</Employee>
  <Employee>PU_CLERK Khoo</Employee>
  <Employee>PU_CLERK Tobias</Employee>
  <Employee>PU_CLERK Baida</Employee>
  <Employee>PU_CLERK Colmenares</Employee>
  <Employee>PU_CLERK Himuro</Employee>
</Department>
```

## CONCLUSION

**As the latest version of SQL standards, SQL 2003 has made major improvements in three key areas: Object-Relational, OLAP, and SQL/XML.**

As the latest version of SQL standards, SQL 2003 has made major improvements in three key areas. First, there are additional object-relational features, which were first introduced in SQL 1999. Second, SQL 2003 standard revolutionizes SQL with comprehensive OLAP features. Third, SQL 2003 delivers a brand new Part 14 for XML-Related Specifications (SQL/XML) to integrate popular XML standards into SQL. Oracle Database 10g provides comprehensive support for these new features.

Oracle's Object-Relational Technology tracks closely with SQL standards development. It has grown to maturity over the years to provide a complete object type system, extensive language binding APIs, and a rich set of utilities and tools. This complete object type system is based on the latest ANSI SQL 2003 standard. Oracle has optimized the object type performance in the database server for object-oriented applications. Oracle's language binding APIs in Java, C/C++, and XML provide direct interfaces to database server object type system. These comprehensive APIs support the most recent standards to access database object type system services. The attendant rich set of utilities for object-relational data include import/export, SQL loader, replication, etc.

With the introduction of analytic functions, Oracle solved many problems of using SQL in business intelligence tasks. The added analytic functions enable faster query performance and greater developer productivity for even more calculations. The value of analytic functions has already been recognized, and major business intelligence tool vendors are using them in their products. The power of the analytic functions, combined with their status as international SQL standards, will make them an important tool for all SQL users.

SQL/XML features are crucial to enterprises with structured and semi-structured data, which needs to interoperate with various enterprise (SQL) reporting tools, relational data stores, transactional middleware, and so on. If your organization uses SQL today, Oracle XML DB will give you native XML capability, and SQL/XML implemented in Oracle XML DB will be the standard way to query your XML data.

In addition, Oracle Database 10g introduces enormous improvements in manageability, Grid computing infrastructure, Database Web Services, OLAP, Data Mining, and many other areas to meet the needs of global enterprises. Oracle's database technology provides the most comprehensive solution for the development, deployment, and management of enterprise applications.

Oracle has been the leader of industrial-strength SQL technology since its birth. Oracle will continue to meet the needs of our partners and customers with the best SQL technology. In short, new SQL 2003 capabilities in Oracle Database 10g provide the most comprehensive functionality for developing versatile, scalable, concurrent, and high performance database applications running in a Grid environment.

SQL 2003 Standard Support in Oracle Database 10g
November 2003
Author: Geoff Lee
Contributing Authors: Fred Zemke

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

Oracle Corporation provides the software
that powers the internet.

Oracle is a registered trademark of Oracle Corporation. Various
product and service names referenced herein may be trademarks
of Oracle Corporation. All other product and service names
mentioned may be trademarks of their respective owners.

Copyright © 2003 Oracle Corporation
All rights reserved.