7
Functions

Functions are similar to operators in that they manipulate data items and return a result. Functions differ from operators in the format of their arguments. This format enables them to operate on zero, one, two, or more arguments:

```
function(argument, argument, ...)
```

A function without any arguments is similar to a pseudocolumn (refer to Pseudocolumns). However, a pseudocolumn typically returns a different value for each row in the result set, whereas a function without any arguments typically returns the same value for each row.

This chapter contains these sections:

- About User-Defined Functions

About SQL Functions

SQL functions are built into Oracle Database and are available for use in various appropriate SQL statements. Do not confuse SQL functions with user-defined functions written in PL/SQL.

If you call a SQL function with an argument of a data type other than the data type expected by the SQL function, then Oracle attempts to convert the argument to the expected data type before performing the SQL function.

> **See Also:**
>
> About User-Defined Functions for information on user functions and Data Conversion for implicit conversion of data types

Nulls in SQL Functions

Most scalar functions return null when given a null argument. You can use the `NVL` function to return a value when a null occurs. For example, the expression `NVL(commission_pct,0)` returns 0 if `commission_pct` is null or the value of `commission_pct` if it is not null.

For information on how aggregate functions handle nulls, see Aggregate Functions .

Syntax for SQL Functions

In the syntax diagrams for SQL functions, arguments are indicated by their data types. When the parameter *`function`* appears in SQL syntax, replace it with one of the functions described in this section. Functions are grouped by the data types of their arguments and their return values.

> **Note:**
>
> When you apply SQL functions to LOB columns, Oracle Database creates temporary LOBs during SQL and PL/SQL processing. You should ensure that temporary tablespace quota is sufficient for storing these temporary LOBs for your application.

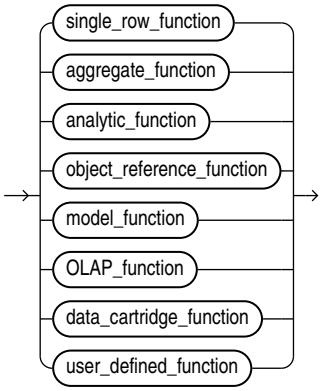
A SQL function may be collation-sensitive, which means that character value comparison or matching that it performs is controlled by a collation. The particular collation to use by the function is determined from the collations of the function's arguments.

If the result of a SQL function has a character data type, the collation derivation rules define the collation to associate with the result.

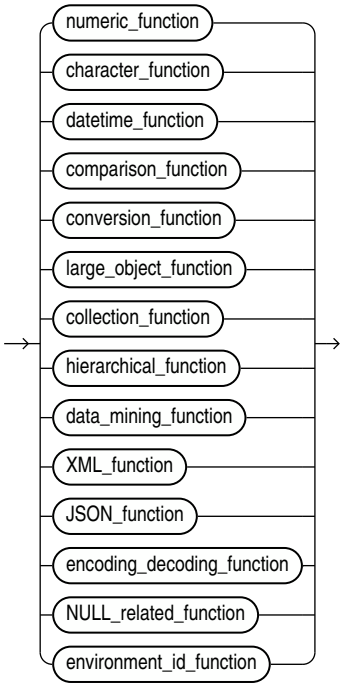
> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation and determination rules for SQL functions

The syntax showing the categories of functions follows:

***function*::=**



***single_row_function*::=**



The sections that follow list the built-in SQL functions in each of the groups illustrated in the preceding diagrams except user-defined functions. All of the built-in SQL functions are then described in alphabetical order.

> **See Also:**
>
> About User-Defined Functions and CREATE FUNCTION

Aggregate Functions

Aggregate functions return a single result row based on groups of rows, rather than on single rows. Aggregate functions can appear in select lists and in `ORDER BY` and `HAVING` clauses. They are commonly used with the `GROUP BY` clause in a `SELECT` statement, where Oracle Database divides the rows of a queried table or view into groups. In a query containing a `GROUP BY` clause, the elements of the select list can be aggregate functions, `GROUP BY` expressions, constants, or expressions involving one of these. Oracle applies the aggregate functions to each group of rows and returns a single result row for each group.

If you omit the `GROUP BY` clause, then Oracle applies aggregate functions in the select list to all the rows in the queried table or view. You use aggregate functions in the `HAVING` clause to eliminate groups from the output based on the results of the aggregate functions, rather than on the values of the individual rows of the queried table or view.

See Also:

- Using the GROUP BY Clause: Examples and the HAVING Clause for more information on the `GROUP BY` clause and `HAVING` clauses in queries and subqueries
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for expressions in the `ORDER BY` clause of an aggregate function

Many (but not all) aggregate functions that take a single argument accept these clauses:

- `DISTINCT` and `UNIQUE`, which are synonymous, cause an aggregate function to consider only distinct values of the argument expression. The syntax diagrams for aggregate functions in this chapter use the keyword `DISTINCT` for simplicity.
- `ALL` causes an aggregate function to consider all values, including all duplicates.

For example, the `DISTINCT` average of 1, 1, 1, and 3 is 2. The `ALL` average is 1.5. If you specify neither, then the default is `ALL`.

Some aggregate functions allow the *`windowing_clause`*, which is part of the syntax of analytic functions. Refer to *windowing_clause* for information about this clause.

All aggregate functions except `COUNT`(*), `GROUPING`, and `GROUPING_ID` ignore nulls. You can use the `NVL` function in the argument to an aggregate function to substitute a value for a null. `COUNT` and `REGR_COUNT` never return null, but return either a number or zero. For all the remaining aggregate functions, if the data set contains no rows, or contains only rows with nulls as arguments to the aggregate function, then the function returns null.

The aggregate functions `MIN`, `MAX`, `SUM`, `AVG`, `COUNT`, `VARIANCE`, and `STDDEV`, when followed by the `KEEP` keyword, can be used in conjunction with the `FIRST` or `LAST` function to operate on a set of values from a set of rows that rank as the `FIRST` or `LAST` with respect to a given sorting specification. Refer to FIRST for more information.

You can nest aggregate functions. For example, the following example calculates the average of the maximum salaries of all the departments in the sample schema `hr`:

```
SELECT AVG(MAX(salary))
  FROM employees
  GROUP BY department_id;

AVG(MAX(SALARY))
----------------
      10926.3333
```

This calculation evaluates the inner aggregate (`MAX(salary)`) for each group defined by the `GROUP BY` clause (`department_id`), and aggregates the results again.

- ANY_VALUE
- APPROX_COUNT
- APPROX_COUNT_DISTINCT
- APPROX_COUNT_DISTINCT_AGG
- APPROX_COUNT_DISTINCT_DETAIL
- APPROX_MEDIAN
- APPROX_PERCENTILE
- APPROX_PERCENTILE_AGG
- APPROX_PERCENTILE_DETAIL
- APPROX_RANK
- APPROX_SUM
- AVG
- BIT_AND_AGG
- BIT_OR_AGG
- BIT_XOR_AGG
- BOOLEAN_AND_AGG
- BOOLEAN_OR_AGG
- CHECKSUM
- COLLECT
- CORR
- CORR_*
- COUNT
- COVAR_POP
- COVAR_SAMP
- CUME_DIST
- DENSE_RANK
- EVERY
- FIRST
- GROUP_ID
- GROUPING
- GROUPING_ID
- JSON_ARRAYAGG
- JSON_OBJECTAGG
- KURTOSIS_POP
- KURTOSIS_SAMP
- LAST
- LISTAGG
- MAX
- MEDIAN
- MIN
- PERCENT_RANK

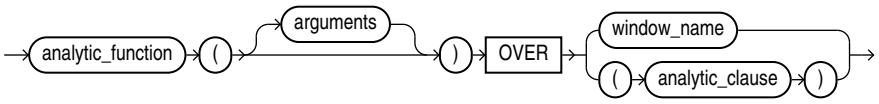
Analytic Functions

Analytic functions compute an aggregate value based on a group of rows. They differ from aggregate functions in that they return multiple rows for each group. The group of rows is called a **window** and is defined by the *analytic_clause*. For each row, a sliding window of rows is defined. The window determines the range of rows used to perform the calculations for the current row. Window sizes can be based on either a physical number of rows or a logical interval such as time.

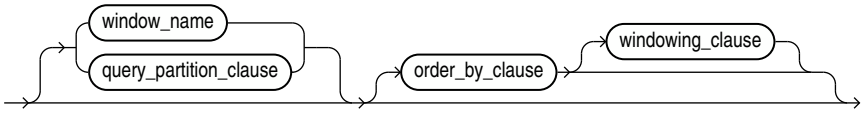
Analytic functions are the last set of operations performed in a query except for the final `ORDER BY` clause. All joins and all `WHERE`, `GROUP BY`, and `HAVING` clauses are completed before the analytic functions are processed. Therefore, analytic functions can appear only in the select list or `ORDER BY` clause.

Analytic functions are commonly used to compute cumulative, moving, centered, and reporting aggregates.

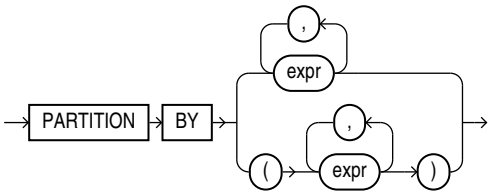
***analytic_function*::=**



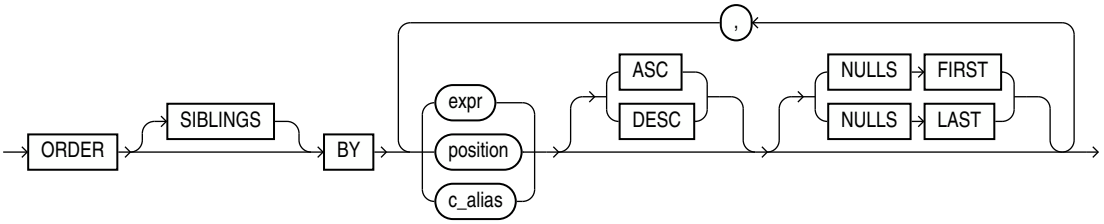
***analytic_clause*::=**



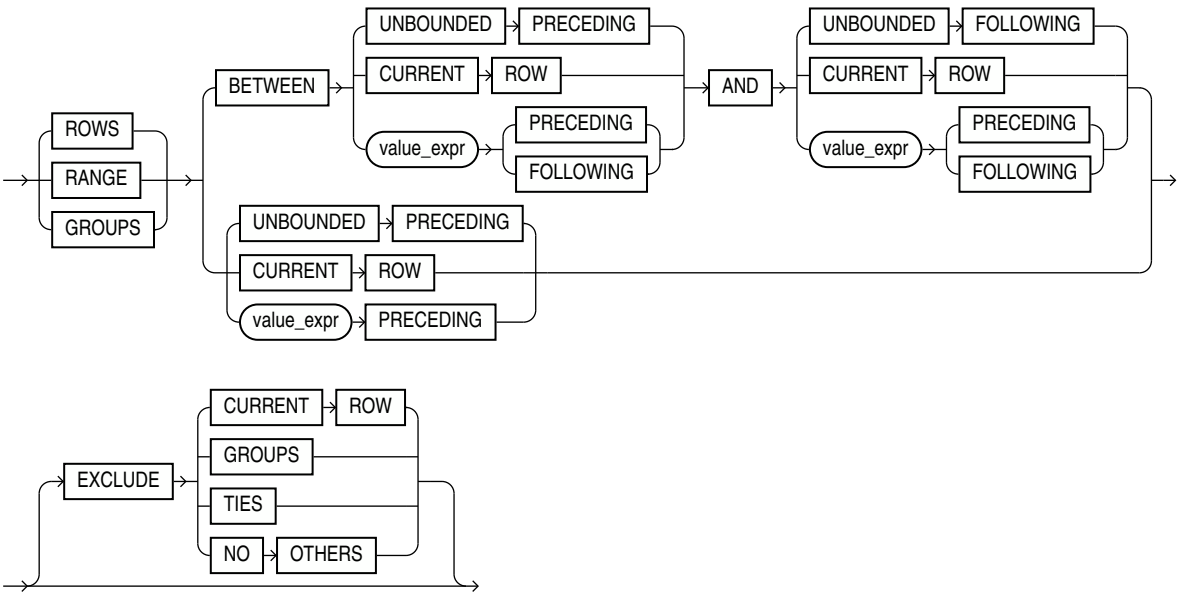
***query_partition_clause*::=**



***order_by_clause*::=**



***windowing_clause*::=**



The semantics of this syntax are discussed in the sections that follow.

analytic_function

Specify the name of an analytic function (see the listing of analytic functions following this discussion of semantics).

arguments

Analytic functions take 0 to 3 arguments. The arguments can be any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence and implicitly converts the remaining arguments to that data type. The return type is also that data type, unless otherwise noted for an individual function.

> **See Also:**
>
> Numeric Precedence for information on numeric precedence and Table 2-9 for more information on implicit conversion

analytic_clause

Use `OVER` *`analytic_clause`* to indicate that the function operates on a query result set. This clause is computed after the `FROM`, `WHERE`, `GROUP BY`, and `HAVING` clauses. You can specify analytic functions with this clause in the select list or `ORDER BY` clause. To filter the results of a query based on an analytic function, nest these functions within the parent query, and then filter the results of the nested subquery.

Notes on the *analytic_clause*:

The following notes apply to the *`analytic_clause`*:

- You cannot nest analytic functions by specifying any analytic function in any part of the *`analytic_clause`*. However, you can specify an analytic function in a subquery and compute another analytic function over it.
- You can specify `OVER` *`analytic_clause`* with user-defined analytic functions as well as built-in analytic functions. See CREATE FUNCTION .
- The `PARTITION BY` and `ORDER BY` clauses in the *`analytic_clause`* are collation-sensitive.

> **See Also:**
>
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the `OVER (PARTITION BY ... ORDER BY ... )` clause of an analytic function
> - *`window_clause`* in the `SELECT` statement

query_partition_clause

Use the PARTITION BY clause to partition the query result set into groups based on one or more *value_expr*. If you omit this clause, then the function treats all rows of the query result set as a single group.

To use the *query_partition_clause* in an analytic function, use the upper branch of the syntax (without parentheses). To use this clause in a model query (in the *model_column_clauses*) or a partitioned outer join (in the *outer_join_clause*), use the lower branch of the syntax (with parentheses).

You can specify multiple analytic functions in the same query, each with the same or different PARTITION BY keys.

If the objects being queried have the parallel attribute, and if you specify an analytic function with the *query_partition_clause*, then the function computations are parallelized as well.

Valid values of *value_expr* are constants, columns, nonanalytic functions, function expressions, or expressions involving any of these.

order_by_clause

Use the *order_by_clause* to specify how data is ordered within a partition. For all analytic functions you can order the values in a partition on multiple keys, each defined by a *value_expr* and each qualified by an ordering sequence.

Within each function, you can specify multiple ordering expressions. Doing so is especially useful when using functions that rank values, because the second expression can resolve ties between identical values for the first expression.

Whenever the *order_by_clause* results in identical values for multiple rows, the function behaves as follows:

- CUME_DIST, DENSE_RANK, NTILE, PERCENT_RANK, and RANK return the same result for each of the rows.
- ROW_NUMBER assigns each row a distinct value even if there is a tie based on the *order_by_clause*. The value is based on the order in which the row is processed, which may be nondeterministic if the ORDER BY does not guarantee a total ordering.
- For all other analytic functions, the result depends on the window specification. If you specify a logical window with the RANGE keyword, then the function returns the same result for each of the rows. If you specify a physical window with the ROWS keyword, then the result is nondeterministic.

Restrictions on the ORDER BY Clause

The following restrictions apply to the ORDER BY clause:

- When used in an analytic function, the *order_by_clause* must take an expression (*expr*). The SIBLINGS keyword is not valid (it is relevant only in hierarchical queries). Position (*position*) and column aliases (*c_alias*) are also invalid. Otherwise this *order_by_clause* is the same as that used to order the overall query or subquery.
- An analytic function that uses the RANGE keyword can use multiple sort keys in its ORDER BY clause if it specifies any of the following windows:
  - RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. The short form of this is RANGE UNBOUNDED PRECEDING.

- `RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`
- `RANGE BETWEEN CURRENT ROW AND CURRENT ROW`
- `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`

Window boundaries other than these four can have only one sort key in the `ORDER BY` clause of the analytic function. This restriction does not apply to window boundaries specified by the `ROW` keyword.

ASC | DESC

Specify the ordering sequence (ascending or descending). `ASC` is the default.

NULLS FIRST | NULLS LAST

Specify whether returned rows containing nulls should appear first or last in the ordering sequence.

`NULLS LAST` is the default for ascending order, and `NULLS FIRST` is the default for descending order.

Analytic functions always operate on rows in the order specified in the *`order_by_clause`* of the function. However, the *`order_by_clause`* of the function does not guarantee the order of the result. Use the *`order_by_clause`* of the query to guarantee the final result ordering.

> **See Also:**
>
> *order_by_clause* of SELECT for more information on this clause

windowing_clause

Some analytic functions allow the *`windowing_clause`*. In the listing of analytic functions at the end of this section, the functions that allow the *`windowing_clause`* are followed by an asterisk (*).

ROWS | RANGE | GROUPS

The keywords `ROWS`, `RANGE`, and `GROUPS` are options to define a window frame unit used for calculating the function result. The function is then applied to all the rows in the window. The window moves through the query result set or partition from top to bottom.

- Use `ROWS` to specify the window frame extent by counting rows forward or backward from the current row. `ROWS` allows any number of sort keys, of any ordered data types.
- Use `RANGE` to specify the window frame extent as a logical offset. `RANGE` allows only one sort key, and its declared data type must allow addition and subtraction operations, for example they must be numeric, datetime, or interval data types.
- Use `GROUPS` to specifiy the window frame extent with both `ROWS` and `RANGE` characteristics. Like `ROWS` a `GROUPS` window can have any number of sort keys, or any ordered types. Like `RANGE`, a `GROUPS` window does not make cutoffs between adjacent rows with the same values in the sort keys.

You cannot specify this clause unless you have specified the *`order_by_clause`*. Some window boundaries defined by the `RANGE` clause let you specify only one expression in the *`order_by_clause`*. Refer to Restrictions on the ORDER BY Clause.

The value returned by an analytic function with a logical offset is always deterministic. However, the value returned by an analytic function with a physical offset may produce nondeterministic results unless the ordering expression results in a unique ordering. You may have to specify multiple columns in the *order_by_clause* to achieve this unique ordering.

BETWEEN ... AND

Use the BETWEEN ... AND clause to specify a start point and end point for the window. The first expression (before AND) defines the start point and the second expression (after AND) defines the end point.

If you omit BETWEEN and specify only one end point, then Oracle considers it the start point, and the end point defaults to the current row.

UNBOUNDED PRECEDING

Specify UNBOUNDED PRECEDING to indicate that the window starts at the first row of the partition. This is the start point specification and cannot be used as an end point specification.

UNBOUNDED FOLLOWING

Specify UNBOUNDED FOLLOWING to indicate that the window ends at the last row of the partition. This is the end point specification and cannot be used as a start point specification.

CURRENT ROW

As a start point, CURRENT ROW specifies that the window begins at the current row or value (depending on whether you have specified ROW or RANGE, respectively). In this case the end point cannot be *value_expr* PRECEDING.

As an end point, CURRENT ROW specifies that the window ends at the current row or value (depending on whether you have specified ROW or RANGE, respectively). In this case the start point cannot be *value_expr* FOLLOWING.

value_expr PRECEDING or *value_expr* FOLLOWING

For RANGE or ROW:

- If *value_expr* FOLLOWING is the start point, then the end point must be *value_expr* FOLLOWING.
- If *value_expr* PRECEDING is the end point, then the start point must be *value_expr* PRECEDING.

If you are defining a logical window defined by an interval of time in numeric format, then you may need to use conversion functions.

> **See Also:**
>
> NUMTOYMINTERVAL and NUMTODSINTERVAL for information on converting numeric times into intervals

If you specified ROWS:

- *value_expr* is a physical offset. It must be a constant or expression and must evaluate to a positive numeric value.

- If *value_expr* is part of the start point, then it must evaluate to a row before the end point.

If you specified RANGE:

- *value_expr* is a logical offset. It must be a constant or expression that evaluates to a positive numeric value or an interval literal. Refer to Literals for information on interval literals.
- You can specify only one expression in the *order_by_clause*.
- If *value_expr* evaluates to a numeric value, then the ORDER BY *expr* must be a numeric or DATE data type.
- If *value_expr* evaluates to an interval value, then the ORDER BY *expr* must be a DATE data type.

If you omit the *windowing_clause* entirely, then the default is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

EXCLUDE

You can remove rows, groups, and ties from the window frame with the EXCLUDE options:

- If you specify EXCLUDE CURRENT ROW, and the current row in in the window frame, then the current row is removed from the window frame.
- If you specify EXCLUDE GROUP, then the current row and any peers of the current row are removed from the window frame.
- If you specify EXCLUDE TIES, then the peers of the current row are removed from the window frame. The current row is retained. Note, that if the current row is previously removed from the window frame, it remains removed.
- If you specify EXCLUDE NO OTHERS, then no additional rows are removed from the window frame. This is the default option.

Analytic functions are commonly used in data warehousing environments. In the list of analytic functions that follows, functions followed by an asterisk (*) allow the full syntax, including the *windowing_clause*.

AVG *
BIT_AND_AGG*
BIT_OR_AGG*
BIT_XOR_AGG*
BOOLEAN_AND_AGG*
BOOLEAN_OR_AGG*
CHECKSUM*
CLUSTER_DETAILS
CLUSTER_DISTANCE
CLUSTER_ID
CLUSTER_PROBABILITY
CLUSTER_SET
CORR *
COUNT *
COVAR_POP *
COVAR_SAMP *
CUME_DIST
DENSE_RANK
EVERY*

FEATURE_DETAILS
FEATURE_ID
FEATURE_SET
FEATURE_VALUE
FIRST
FIRST_VALUE *
KURTOSIS_POP*
KURTOSIS_SAMP*
LAG
LAST
LAST_VALUE *
LEAD
LISTAGG
MAX *
MEDIAN
MIN *
NTH_VALUE *
NTILE
PERCENT_RANK
PERCENTILE_CONT
PERCENTILE_DISC
PREDICTION
PREDICTION_COST
PREDICTION_DETAILS
PREDICTION_PROBABILITY
PREDICTION_SET
RANK
RATIO_TO_REPORT
REGR_ (Linear Regression) Functions *
ROW_NUMBER
STDDEV *
SKEWNESS_POP*
SKEWNESS_SAMP*
STDDEV_POP *
STDDEV_SAMP *
SUM *
VAR_POP *
VAR_SAMP *
VARIANCE *

See Also:

Oracle Database Data Warehousing Guide for more information on these functions and for scenarios illustrating their use

Data Cartridge Functions

Data Cartridge functions are useful for Data Cartridge developers. The Data Cartridge functions are:

DATAOBJ_TO_MAT_PARTITION
DATAOBJ_TO_PARTITION

Model Functions

Model functions can be used only in the *model_clause* of the SELECT statement. The model functions are:

CV
ITERATION_NUMBER
PRESENTNNV
PRESENTV
PREVIOUS

Object Reference Functions

Object reference functions manipulate REF values, which are references to objects of specified object types. The object reference functions are:

DEREF
MAKE_REF
REF
REFTOHEX
VALUE

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for more information about REF data types

OLAP Functions

OLAP functions returns data from a dimensional object in two-dimension relational format. The OLAP function is:

CUBE_TABLE

Single-Row Functions

Single-row functions return a single result row for every row of a queried table or view. These functions can appear in select lists, WHERE clauses, START WITH and CONNECT BY clauses, and HAVING clauses.

Numeric Functions

Numeric functions accept numeric input and return numeric values. Most numeric functions return `NUMBER` values that are accurate to 38 decimal digits. The transcendental functions `COS`, `COSH`, `EXP`, `LN`, `LOG`, `SIN`, `SINH`, `SQRT`, `TAN`, and `TANH` are accurate to 36 decimal digits. The transcendental functions `ACOS`, `ASIN`, `ATAN`, and `ATAN2` are accurate to 30 decimal digits. The numeric functions are:

ABS
ACOS
ASIN
ATAN
ATAN2
BITAND
CEIL (number)
COS
COSH
EXP
FLOOR (number)
LN
LOG
MOD
NANVL
POWER
REMAINDER
ROUND (number)
SIGN
SIN
SINH
SQRT
TAN
TANH
TRUNC (number)
WIDTH_BUCKET

Character Functions Returning Character Values

Character functions that return character values return values of the following data types unless otherwise documented:

- If the input argument is `CHAR` or `VARCHAR2`, then the value returned is `VARCHAR2`.
- If the input argument is `NCHAR` or `NVARCHAR2`, then the value returned is `NVARCHAR2`.

The length of the value returned by the function is limited by the maximum length of the data type returned.

- For functions that return `CHAR` or `VARCHAR2`, if the length of the return value exceeds the limit, then Oracle Database truncates it and returns the result without an error message.
- For functions that return `CLOB` values, if the length of the return values exceeds the limit, then Oracle raises an error and returns no data.

The character functions that return character values are:

CHR
CONCAT
INITCAP
LOWER
LPAD
LTRIM
NCHR
NLS_INITCAP
NLS_LOWER
NLS_UPPER
NLSSORT
REGEXP_REPLACE
REGEXP_SUBSTR
REPLACE
RPAD
RTRIM
SOUNDEX
SUBSTR
TRANSLATE
TRANSLATE ... USING
TRIM
UPPER

Character Functions Returning Number Values

Character functions that return number values can take as their argument any character data type. The character functions that return number values are:

ASCII
INSTR
LENGTH
REGEXP_COUNT
REGEXP_INSTR

Character Set Functions

The character set functions return information about the character set. The character set functions are:

NLS_CHARSET_DECL_LEN
NLS_CHARSET_ID
NLS_CHARSET_NAME

Collation Functions

The collation functions return information about collation settings. The collation functions are:

COLLATION
NLS_COLLATION_ID
NLS_COLLATION_NAME

Datetime Functions

Datetime functions operate on date (`DATE`), timestamp (`TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `TIMESTAMP WITH LOCAL TIME ZONE`), and interval (`INTERVAL DAY TO SECOND`, `INTERVAL YEAR TO MONTH`) values.

Some of the datetime functions were designed for the Oracle `DATE` data type (`ADD_MONTHS`, `CURRENT_DATE`, `LAST_DAY`, `NEW_TIME`, and `NEXT_DAY`). If you provide a timestamp value as their argument, then Oracle Database internally converts the input type to a `DATE` value and returns a `DATE` value. The exceptions are the `MONTHS_BETWEEN` function, which returns a number, and the `ROUND` and `TRUNC` functions, which do not accept timestamp or interval values at all.

The remaining datetime functions were designed to accept any of the three types of data (date, timestamp, and interval) and to return a value of one of these types.

All of the datetime functions that return current system datetime information, such as `SYSDATE`, `SYSTIMESTAMP`, `CURRENT_TIMESTAMP`, and so forth, are evaluated once for each SQL statement, regardless how many times they are referenced in that statement.

The datetime functions are:

General Comparison Functions

The general comparison functions determine the greatest and or least value from a set of values. The general comparison functions are:

GREATEST
LEAST

Conversion Functions

Conversion functions convert a value from one data type to another. Generally, the form of the function names follows the convention *datatype* `TO` *datatype*. The first data type is the input data type. The second data type is the output data type. The SQL conversion functions are:

ASCIISTR
BIN_TO_NUM
CAST
CHARTOROWID
COMPOSE
CONVERT
DECOMPOSE
HEXTORAW
NUMTODSINTERVAL
NUMTOYMINTERVAL
RAWTOHEX
RAWTONHEX
ROWIDTOCHAR
ROWIDTONCHAR
SCN_TO_TIMESTAMP
TIMESTAMP_TO_SCN
TO_BINARY_DOUBLE
TO_BINARY_FLOAT
TO_BLOB (bfile)
TO_BLOB (raw)
TO_CHAR (bfile|blob)
TO_CHAR (character)
TO_CHAR (datetime)
TO_CHAR (number)
TO_CLOB (bfile|blob)
TO_CLOB (character)
TO_DATE
TO_DSINTERVAL
TO_LOB
TO_MULTI_BYTE
TO_NCHAR (character)
TO_NCHAR (datetime)
TO_NCHAR (number)
TO_NCLOB
TO_NUMBER
TO_SINGLE_BYTE

TO_TIMESTAMP
TO_TIMESTAMP_TZ
TO_YMINTERVAL
TREAT
UNISTR
VALIDATE_CONVERSION

Large Object Functions

The large object functions operate on LOBs. The large object functions are:

BFILENAME
EMPTY_BLOB, EMPTY_CLOB

Collection Functions

The collection functions operate on nested tables and varrays. The SQL collection functions are:

CARDINALITY
COLLECT
POWERMULTISET
POWERMULTISET_BY_CARDINALITY
SET

Hierarchical Functions

Hierarchical functions applies hierarchical path information to a result set. The hierarchical function is:

SYS_CONNECT_BY_PATH

Oracle Machine Learning for SQL Functions

The Oracle Machine Learning for SQL functions use analytics to score data. The functions can apply a mining model schema object to the data, or they can dynamically mine the data by executing an analytic clause. The OML4SQL functions can be applied to models built using the native algorithms of Oracle, as well as those built using R through the extensibility mechanism.

The Oracle Machine Learning for SQL functions are:

CLUSTER_DETAILS
CLUSTER_DISTANCE
CLUSTER_ID
CLUSTER_PROBABILITY
CLUSTER_SET
FEATURE_COMPARE
FEATURE_DETAILS
FEATURE_ID
FEATURE_SET
FEATURE_VALUE
ORA_DM_PARTITION_NAME
PREDICTION

PREDICTION_BOUNDS
PREDICTION_COST
PREDICTION_DETAILS
PREDICTION_PROBABILITY
PREDICTION_SET
VECTOR_EMBEDDING

> **See Also:**
>
> - *Oracle Machine Learning for SQL Concepts* to learn about Oracle Machine Learning for SQL
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring

XML Functions

The XML functions operate on or return XML documents or fragments. These functions use arguments that are not defined as part of the ANSI/ISO/IEC SQL Standard but are defined as part of the World Wide Web Consortium (W3C) standards. The processing and operations that the functions perform are defined by the relevant W3C standards. The table below provides a link to the appropriate section of the W3C standard for the rules and guidelines that apply to each of these XML-related arguments. A SQL statement that uses one of these XML functions, where any of the arguments does not conform to the relevant W3C syntax, will result in an error. Of special note is the fact that not every character that is allowed in the value of a database column is considered legal in XML.

Syntax Element	W3C Standard URL
value_expr	http://www.w3.org/TR/2006/REC-xml-20060816
Xpath_string	http://www.w3.org/TR/1999/REC-xpath-19991116
XQuery_string	http://www.w3.org/TR/2007/REC-xquery-semantics-20070123/ http://www.w3.org/TR/xquery-update-10/
namespace_string	http://www.w3.org/TR/2006/REC-xml-names-20060816/
identifier	http://www.w3.org/TR/2006/REC-xml-20060816/#NT-Nmtoken

For more information about selecting and querying XML data using these functions, including information on formatting output, refer to *Oracle XML DB Developer's Guide*

The SQL XML functions are:

DEPTH
EXISTSNODE
EXTRACT (XML)
EXTRACTVALUE
PATH
SYS_DBURIGEN
SYS_XMLAGG
SYS_XMLGEN
XMLAGG
XMLCAST

XMLCDATA
XMLCOLATTVAL
XMLCOMMENT
XMLCONCAT
XMLDIFF
XMLELEMENT
XMLEXISTS
XMLFOREST
XMLISVALID
XMLPARSE
XMLPATCH
XMLPI
XMLQUERY
XMLSEQUENCE
XMLSERIALIZE
XMLTABLE
XMLTRANSFORM

JSON Functions

JavaScript Object Notation (JSON) functions allow you to query and generate JSON data.

The following SQL/JSON functions allow you to query JSON data:

JSON_QUERY
JSON_TABLE
JSON_VALUE

The following SQL/JSON functions allow you to generate JSON data:

JSON_ARRAY
JSON_ARRAYAGG
JSON_OBJECT
JSON_OBJECTAGG
JSON Type Constructor
JSON_SCALAR
JSON_SERIALIZE
JSON_TRANSFORM

The following Oracle SQL function creates a JSON data guide:

JSON_DATAGUIDE

Encoding and Decoding Functions

The encoding and decoding functions let you inspect and decode data in the database. The encoding and decoding functions are:

DECODE
DUMP
ORA_HASH
STANDARD_HASH
VSIZE

NULL-Related Functions

The `NULL`-related functions facilitate null handling. The `NULL`-related functions are:

COALESCE
LNNVL
NANVL
NULLIF
NVL
NVL2

Environment and Identifier Functions

The environment and identifier functions provide information about the instance and session. The environment and identifier functions are:

CON_DBID_TO_ID
CON_GUID_TO_ID
CON_NAME_TO_ID
CON_UID_TO_ID
ORA_INVOKING_USER
ORA_INVOKING_USERID
SYS_CONTEXT
SYS_GUID
SYS_TYPEID
UID
USER
USERENV

Domain Functions

Purpose

Use the following domain functions to work with usecase domains more efficiently:

- DOMAIN_DISPLAY
- DOMAIN_ORDER
- DOMAIN_NAME
- DOMAIN_CHECK
- DOMAIN_CHECK_TYPE

Vector Functions

Purpose

You can use the following vector functions in Oracle AI Vector Search to create and manipulate vectors:

Vector Distance Functions

- VECTOR_DISTANCE

- L1_DISTANCE
- L2_DISTANCE
- COSINE_DISTANCE
- INNER_PRODUCT

Vector Constructors

- TO_VECTOR
- VECTOR

Vector Serializers

- FROM_VECTOR
- VECTOR_SERIALIZE

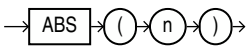
Other Common Vector Functions

- VECTOR_CHUNKS
- VECTOR_DIMS
- VECTOR_DIMENSION_COUNT
- VECTOR_DIMENSION_FORMAT
- VECTOR_EMBEDDING
- VECTOR_NORM

> **See Also:**
>
> *AI Vector Search User's Guide*

ABS

Syntax



Purpose

`ABS` returns the absolute value of *n*.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

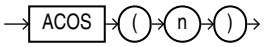
Examples

The following example returns the absolute value of -15:

```
SELECT ABS(-15) "Absolute"
  FROM DUAL;

  Absolute
----------
        15
```

ACOS

Syntax



Purpose

`ACOS` returns the arc cosine of *n*. The argument *n* must be in the range of -1 to 1, and the function returns a value in the range of 0 to *pi*, expressed in radians.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

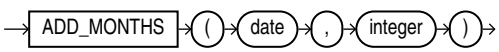
Examples

The following example returns the arc cosine of .3:

```
SELECT ACOS(.3)"Arc_Cosine"
  FROM DUAL;

Arc_Cosine
----------
1.26610367
```

ADD_MONTHS

Syntax



Purpose

ADD_MONTHS returns the date *date* plus *integer* months. A month is defined by the session parameter NLS_CALENDAR. The date argument can be a datetime value or any value that can be implicitly converted to DATE. The *integer* argument can be an integer or any value that can be implicitly converted to an integer. The return type is always DATE, regardless of the data type of *date*. If *date* is the last day of the month or if the resulting month has fewer days than the day component of *date*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *date*.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

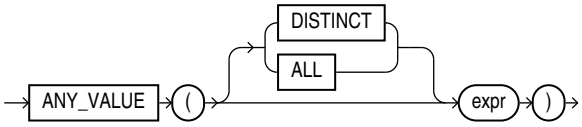
Examples

The following example returns the month after the *hire_date* in the sample table employees:

```
SELECT TO_CHAR(ADD_MONTHS(hire_date, 1), 'DD-MON-YYYY') "Next month"
  FROM employees
  WHERE last_name = 'Baer';

Next Month
-----------
07-JUL-2002
```

ANY_VALUE

Syntax



Purpose

ANY_VALUE returns a single non-deterministic value of expr. You can use it as an aggregate function.

Use ANY_VALUE to optimize a query that has a GROUP BY clause. ANY_VALUE returns a value of an expression in a group. It is optimized to return the first value.

It ensures that there are no comparisons for any incoming row and also eliminates the necessity to specify every column as part of the GROUP BY clause. Because it does not compare values, ANY_VALUE returns a value more quickly than MIN or MAX in a GROUP BY query.

Semantics

ALL, DISTINCT: These keywords are supported by ANY_VALUE although they have no effect on the result of the query.

expr: The expression can be a column, constant, bind variable, or an expression involving them.

NULL values in the expression are ignored.

Supports all of the data types, except for `LONG`, `LOB`, `FILE`, or `COLLECTION`.

If you use `LONG`, `ORA-00997` is raised.

If you use `LOB`, `FILE`, or `COLLECTION` data types, `ORA-00932` is raised.

`ANY_VALUE` follows the same rules as `MIN` and `MAX`.

Returns any value within each group based on the `GROUP BY` specification. Returns NULL if all rows in the group have NULL expression values.

The result of `ANY_VALUE` is not deterministic.

Restrictions

`XMLType` and `ANYDATA` are not supported.

Example 7-1 Using ANY_VALUE As an Aggregate Function

This example uses `ANY_VALUE` as an aggregate function in a `GROUP BY` query of the SH schema.

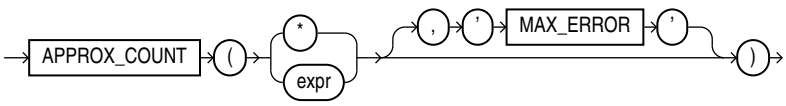
```
SELECT c.cust_id, ANY_VALUE(cust_last_name), SUM(amount_sold)
  FROM customers c, sales s
  WHERE s.cust_id = c.cust_id
  GROUP BY c.cust_id;
```

In the following result of the query, only the first eleven rows are shown.

```
CUST_ID  ANY_VALUE(CUST_LAST_NAME) SUM(AMOUNT_SOLD)
------- -------------------------- ----------------
   6950 Sandburg                               78
  17920 Oliver                               3201
  66800 Case                                 2024
  37280 Edwards                              2256
 109850 Lindegreen                            757
   3910 Oddell                                185
  84700 Marker                              164.4
  26380 Remler                                118
  11600 Oppy                                  158
  23030 Rothrock                              533
  42780 Zanis                                 182
...
630 rows selected.
```

APPROX_COUNT

Syntax



Purpose

APPROX_COUNT returns the approximate count of an expression. If you supply MAX_ERROR as the second argument, then the function returns the maximum error between the actual and approximate count.

You must use this function with a corresponding APPROX_RANK function in the HAVING clause. If a query uses APPROX_COUNT, APPROX_SUM, or APPROX_RANK, then the query must not use any other aggregation functions.

Examples

The following query returns the 10 most common jobs within every department:

```
SELECT department_id, job_id,
       APPROX_COUNT(*)
FROM   employees
GROUP BY department_id, job_id
HAVING
  APPROX_RANK (
  PARTITION BY department_id
  ORDER BY APPROX_COUNT(*)
  DESC ) <= 10;
```

APPROX_COUNT_DISTINCT

Syntax

Purpose

APPROX_COUNT_DISTINCT returns the approximate number of rows that contain a distinct value for *expr*.

This function provides an alternative to the COUNT (DISTINCT *expr*) function, which returns the exact number of rows that contain distinct values of *expr*. APPROX_COUNT_DISTINCT processes large amounts of data significantly faster than COUNT, with negligible deviation from the exact result.

For *expr*, you can specify a column of any scalar data type other than BFILE, BLOB, CLOB, LONG, LONG RAW, or NCLOB.

APPROX_COUNT_DISTINCT ignores rows that contain a null value for *expr*. This function returns a NUMBER.

> **See Also:**
>
> - COUNT for more information on the `COUNT (DISTINCT expr)` function
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `APPROX_COUNT_DISTINCT` uses to compare character values for *expr*

Examples

The following statement returns the approximate number of rows with distinct values for `manager_id`:

```
SELECT APPROX_COUNT_DISTINCT(manager_id) AS "Active Managers"
  FROM employees;

Active Managers
---------------
             18
```

The following statement returns the approximate number of distinct customers for each product:

```
SELECT prod_id, APPROX_COUNT_DISTINCT(cust_id) AS "Number of Customers"
  FROM sales
  GROUP BY prod_id
  ORDER BY prod_id;

   PROD_ID Number of Customers
---------- --------------------
        13                2516
        14                2030
        15                2105
        16                2367
        17                2093
        18                2975
        19                2630
        20                3791
. . .
```

APPROX_COUNT_DISTINCT_AGG

Syntax

Purpose

`APPROX_COUNT_DISTINCT_AGG` takes as its input a column of details containing information about approximate distinct value counts, and enables you to perform aggregations of those counts.

For *detail*, specify a column of details created by the `APPROX_COUNT_DISTINCT_DETAIL` function or the `APPROX_COUNT_DISTINCT_AGG` function. This column is of data type `BLOB`.

You can specify this function in a SELECT statement with a GROUP BY clause to aggregate the information contained in the details within each group of rows and return a single detail for each group.

This function returns a BLOB value, called a detail, which contains information about the count aggregations in a special format. You can store details returned by this function in a table or materialized view, and then again use the APPROX_COUNT_DISTINCT_AGG function to further aggregate those details, or use the TO_APPROX_COUNT_DISTINCT function to convert the detail values to human-readable NUMBER values.

See Also:

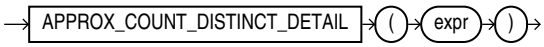
- APPROX_COUNT_DISTINCT_DETAIL
- TO_APPROX_COUNT_DISTINCT

Examples

Refer to APPROX_COUNT_DISTINCT_AGG: Examples for examples of using the APPROX_COUNT_DISTINCT_AGG function in conjunction with the APPROX_COUNT_DISTINCT_DETAIL and TO_APPROX_COUNT_DISTINCT functions.

APPROX_COUNT_DISTINCT_DETAIL

Syntax



Purpose

APPROX_COUNT_DISTINCT_DETAIL calculates information about the approximate number of rows that contain a distinct value for expr and returns a BLOB value, called a detail, which contains that information in a special format.

For *expr*, you can specify a column of any scalar data type other than BFILE, BLOB, CLOB, LONG, LONG RAW, or NCLOB. This function ignores rows for which the value of *expr* is null.

This function is commonly used with the GROUP BY clause in a SELECT statement. When used in this way, it calculates approximate distinct value count information for *expr* within each group of rows and returns a single detail for each group.

The details returned by APPROX_COUNT_DISTINCT_DETAIL can be used as input to the APPROX_COUNT_DISTINCT_AGG function, which enables you to perform aggregations of the details, or the TO_APPROX_COUNT_DISTINCT function, which converts a detail to a human-readable distinct count value. You can use these three functions together to perform resource-intensive approximate count calculations once, store the resulting details, and then perform efficient aggregations and queries on those details. For example:

1. Use the APPROX_COUNT_DISTINCT_DETAIL function to calculate approximate distinct value count information and store the resulting details in a table or materialized view. These could be highly-granular details, such as city demographic counts or daily sales counts.

2. Use the APPROX_COUNT_DISTINCT_AGG function to aggregate the details obtained in the previous step and store the resulting details in a table or materialized view. These could be details of lower granularity, such as state demographic counts or monthly sales counts.
3. Use the TO_APPROX_COUNT_DISTINCT function to convert the stored detail values to human-readable NUMBER values. You can use the TO_APPROX_COUNT_DISTINCT function to query detail values created by the APPROX_COUNT_DISTINCT_DETAIL function or the APPROX_COUNT_DISTNCT_AGG function.

See Also:

- APPROX_COUNT_DISTINCT_AGG
- TO_APPROX_COUNT_DISTINCT

Examples

The examples in this section demonstrate how to use the APPROX_COUNT_DISTINCT_DETAIL, APPROX_COUNT_DISTINCT_AGG, and TO_APPROX_COUNT_DISTINCT functions together to perform resource-intensive approximate count calculations once, store the resulting details, and then perform efficient aggregations and queries on those details.

APPROX_COUNT_DISTINCT_DETAIL: Example

The following statement queries the tables sh.times and sh.sales for the approximate number of distinct products sold each day. The APPROX_COUNT_DISTINCT_DETAIL function returns the information in a detail, called daily_detail, for each day that products were sold. The returned details are stored in a materialized view called daily_prod_count_mv.

```
CREATE MATERIALIZED VIEW daily_prod_count_mv AS
  SELECT t.calendar_year year,
         t.calendar_month_number month,
         t.day_number_in_month day,
         APPROX_COUNT_DISTINCT_DETAIL(s.prod_id) daily_detail
  FROM times t, sales s
  WHERE t.time_id = s.time_id
  GROUP BY t.calendar_year, t.calendar_month_number, t.day_number_in_month;
```

APPROX_COUNT_DISTINCT_AGG: Examples

The following statement uses the APPROX_COUNT_DISTINCT_AGG function to read the daily details stored in daily_prod_count_mv and create aggregated details that contain the approximate number of distinct products sold each month. These aggregated details are stored in a materialized view called monthly_prod_count_mv.

```
CREATE MATERIALIZED VIEW monthly_prod_count_mv AS
  SELECT year,
         month,
         APPROX_COUNT_DISTINCT_AGG(daily_detail) monthly_detail
  FROM daily_prod_count_mv
  GROUP BY year, month;
```

The following statement is similar to the previous statement, except it creates aggregated details that contain the approximate number of distinct products sold each year. These aggregated details are stored in a materialized view called annual_prod_count_mv.

```
CREATE MATERIALIZED VIEW annual_prod_count_mv AS
  SELECT year,
         APPROX_COUNT_DISTINCT_AGG(daily_detail) annual_detail
  FROM daily_prod_count_mv
  GROUP BY year;
```

TO_APPROX_COUNT_DISTINCT: Examples

The following statement uses the `TO_APPROX_COUNT_DISTINCT` function to query the daily detail information stored in `daily_prod_count_mv` and return the approximate number of distinct products sold each day:

```
SELECT year,
       month,
       day,
       TO_APPROX_COUNT_DISTINCT(daily_detail) "NUM PRODUCTS"
  FROM daily_prod_count_mv
  ORDER BY year, month, day;

      YEAR      MONTH        DAY NUM PRODUCTS
---------- ---------- ---------- ------------
      1998          1          1           24
      1998          1          2           25
      1998          1          3           11
      1998          1          4           34
      1998          1          5           10
      1998          1          6            8
      1998          1          7           37
      1998          1          8           26
      1998          1          9           25
      1998          1         10           38
. . .
```

The following statement uses the `TO_APPROX_COUNT_DISTINCT` function to query the monthly detail information stored in `monthly_prod_count_mv` and return the approximate number of distinct products sold each month:

```
SELECT year,
       month,
       TO_APPROX_COUNT_DISTINCT(monthly_detail) "NUM PRODUCTS"
  FROM monthly_prod_count_mv
  ORDER BY year, month;

      YEAR      MONTH NUM PRODUCTS
---------- ---------- ------------
      1998          1           57
      1998          2           56
      1998          3           55
      1998          4           49
      1998          5           49
      1998          6           48
      1998          7           54
      1998          8           56
      1998          9           55
```

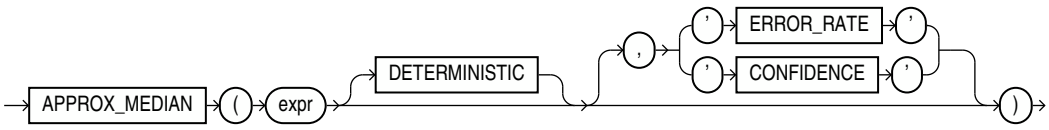
```
      1998         10           57
. . .
```

The following statement uses the `TO_APPROX_COUNT_DISTINCT` function to query the annual detail information stored in `annual_prod_count_mv` and return the approximate number of distinct products sold each year:

```
SELECT year,
       TO_APPROX_COUNT_DISTINCT(annual_detail) "NUM PRODUCTS"
  FROM annual_prod_count_mv
  ORDER BY year;

      YEAR NUM PRODUCTS
---------- ------------
      1998           60
      1999           72
      2000           72
      2001           71
```

APPROX_MEDIAN

Syntax



Purpose

`APPROX_MEDIAN` is an approximate inverse distribution function that assumes a continuous distribution model. It takes a numeric or datetime value and returns an approximate middle value or an approximate interpolated value that would be the middle value once the values are sorted. Nulls are ignored in the calculation.

This function provides an alternative to the `MEDIAN` function, which returns the exact middle value or interpolated value. `APPROX_MEDIAN` processes large amounts of data significantly faster than `MEDIAN`, with negligible deviation from the exact result.

For *`expr`*, specify the expression for which the approximate median value is being calculated. The acceptable data types for *`expr`*, and the return value data type for this function, depend on the algorithm that you specify with the `DETERMINISTIC` clause.

DETERMINISTIC

This clause lets you specify the type of algorithm this function uses to calculate the approximate median value.

- If you specify `DETERMINISTIC`, then this function calculates a deterministic approximate median value. In this case, *`expr`* must evaluate to a numeric value, or to a value that can be implicitly converted to a numeric value. The function returns the same data type as the numeric data type of its argument.

- If you omit `DETERMINSTIC`, then this function calculates a nondeterministic approximate median value. In this case, *`expr`* must evaluate to a numeric or datetime value, or to a value that can be implicitly converted to a numeric or datetime value. The function returns the same data type as the numeric or datetime data type of its argument.

ERROR_RATE | CONFIDENCE

These clauses let you determine the accuracy of the value calculated by this function. If you specify one of these clauses, then instead of returning the approximate median value for *`expr`*, the function returns a decimal value from 0 to 1, inclusive, which represents one of the following values:

- If you specify `ERROR_RATE`, then the return value represents the error rate for the approximate median value calculation for *`expr`*.
- If you specify `CONFIDENCE`, then the return value represents the confidence level for the error rate that is returned when you specify `ERROR_RATE`.

> **See Also:**
>
> - MEDIAN
> - APPROX_PERCENTILE which returns, for a given percentile, the approximate value that corresponds to that percentile by way of interpolation. `APPROX_MEDIAN` is the specific case of `APPROX_PERCENTILE` where the percentile value is 0.5.

Examples

The following query returns the deterministic approximate median salary for each department in the `hr.employees` table:

```
SELECT department_id "Department",
       APPROX_MEDIAN(salary DETERMINISTIC) "Median Salary"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department Median Salary
---------- -------------
        10          4400
        20          6000
        30          2765
        40          6500
        50          3100
        60          4800
        70         10000
        80          9003
        90         17000
       100          7739
       110          8300
                    7000
```

The following query returns the error rates for the approximate median salaries that were returned by the previous query:

```
SELECT department_id "Department",
       APPROX_MEDIAN(salary DETERMINISTIC, 'ERROR_RATE') "Error Rate"
  FROM employees
```

```
  GROUP BY department_id
  ORDER BY department_id;

Department Error Rate
---------- ----------
        10 .002718282
        20 .021746255
        30 .021746255
        40 .002718282
        50 .019027973
        60 .019027973
        70 .002718282
        80 .021746255
        90 .021746255
       100 .019027973
       110 .019027973
           .002718282
```

The following query returns the confidence levels for the error rates that were returned by the previous query:

```
SELECT department_id "Department",
       APPROX_MEDIAN(salary DETERMINISTIC, 'CONFIDENCE') "Confidence Level"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department Confidence Level
---------- ----------------
        10       .997281718
        20       .999660215
        30       .999660215
        40       .997281718
        50       .999611674
        60       .999611674
        70       .997281718
        80       .999660215
        90       .999660215
       100       .999611674
       110       .999611674
                 .997281718
```

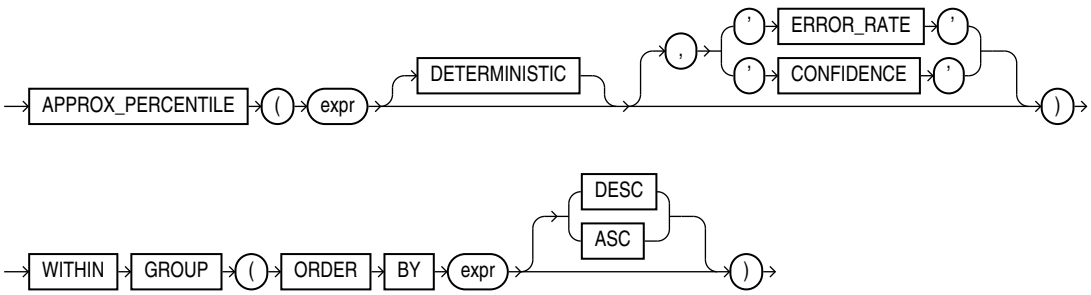
The following query returns the nondeterministic approximate median hire date for each department in the `hr.employees` table:

```
SELECT department_id "Department",
       APPROX_MEDIAN(hire_date) "Median Hire Date"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department Median Hire Date
---------- ----------------
        10        17-SEP-03
        20        17-FEB-04
        30        24-JUL-05
        40        07-JUN-02
        50        15-MAR-06
        60        05-FEB-06
        70        07-JUN-02
        80        23-MAR-06
        90        17-JUN-03
```

```
100         28-SEP-05
110         07-JUN-02
            24-MAY-07
```

APPROX_PERCENTILE

Syntax



Purpose

`APPROX_PERCENTILE` is an approximate inverse distribution function. It takes a percentile value and a sort specification, and returns the value that would fall into that percentile value with respect to the sort specification. Nulls are ignored in the calculation

This function provides an alternative to the `PERCENTILE_CONT` and `PERCENTILE_DISC` functions, which returns the exact results. `APPROX_PERCENTILE` processes large amounts of data significantly faster than `PERCENTILE_CONT` and `PERCENTILE_DISC`, with negligible deviation from the exact result.

The first *`expr`* is the percentile value, which must evaluate to a numeric value between 0 and 1.

The second *`expr`*, which is part of the `ORDER BY` clause, is a single expression over which this function calculates the result. The acceptable data types for *`expr`*, and the return value data type for this function, depend on the algorithm that you specify with the `DETERMINISTIC` clause.

DETERMINISTIC

This clause lets you specify the type of algorithm this function uses to calculate the return value.

- If you specify `DETERMINISTIC`, then this function calculates a deterministic result. In this case, the `ORDER BY` clause expression must evaluate to a numeric value, or to a value that can be implicitly converted to a numeric value, in the range -2,147,483,648 through 2,147,483,647. The function rounds numeric input to the closest integer. The function returns the same data type as the numeric data type of the `ORDER BY` clause expression. The return value is not necessarily one of the values of *`expr`*
- If you omit `DETERMINSTIC`, then this function calculates a nondeterministic result. In this case, the `ORDER BY` clause expression must evaluate to a numeric or datetime value, or to a value that can be implicitly converted to a numeric or datetime value. The function returns the same data type as the numeric or datetime data type of the `ORDER BY` clause expression. The return value is one of the values of *`expr`*.

ERROR_RATE | CONFIDENCE

These clauses let you determine the accuracy of the result calculated by this function. If you specify one of these clauses, then instead of returning the value that would fall into the

specified percentile value for *expr*, the function returns a decimal value from 0 to 1, inclusive, which represents one of the following values:

- If you specify `ERROR_RATE`, then the return value represents the error rate for calculating the value that would fall into the specified percentile value for*expr*.
- If you specify `CONFIDENCE`, then the return value represents the confidence level for the error rate that is returned when you specify `ERROR_RATE`.

DESC | ASC

Specify the sort specification for the calculating the value that would fall into the specified percentile value. Specify `DESC` to sort the `ORDER BY` clause expression values in descending order, or `ASC` to sort the values in ascending order. `ASC` is the default.

> **See Also:**
>
> - PERCENTILE_CONT and PERCENTILE_DISC
> - APPROX_MEDIAN, which is the specific case of `APPROX_PERCENTILE` where the percentile value is 0.5

Examples

The following query returns the deterministic approximate 25th percentile, 50th percentile, and 75th percentile salaries for each department in the `hr.employees` table. The salaries are sorted in ascending order for the interpolation calculation.

```
SELECT department_id "Department",
       APPROX_PERCENTILE(0.25 DETERMINISTIC)
         WITHIN GROUP (ORDER BY salary ASC) "25th Percentile Salary",
       APPROX_PERCENTILE(0.50 DETERMINISTIC)
         WITHIN GROUP (ORDER BY salary ASC) "50th Percentile Salary",
       APPROX_PERCENTILE(0.75 DETERMINISTIC)
         WITHIN GROUP (ORDER BY salary ASC) "75th Percentile Salary"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department 25th Percentile Salary 50th Percentile Salary 75th Percentile Salary
---------- ---------------------- ---------------------- ----------------------
        10                   4400                   4400                   4400
        20                   6000                   6000                  13000
        30                   2633                   2765                   3100
        40                   6500                   6500                   6500
        50                   2600                   3100                   3599
        60                   4800                   4800                   6000
        70                  10000                  10000                  10000
        80                   7400                   9003                  10291
        90                  17000                  17000                  24000
       100                   7698                   7739                   8976
       110                   8300                   8300                  12006
                             7000                   7000                   7000
```

The following query returns the error rates for the approximate 25th percentile salaries that were calculated in the previous query:

```
SELECT department_id "Department",
       APPROX_PERCENTILE(0.25 DETERMINISTIC, 'ERROR_RATE')
         WITHIN GROUP (ORDER BY salary ASC) "Error Rate"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Department Error Rate
---------- ----------
        10 .002718282
        20 .021746255
        30 .021746255
        40 .002718282
        50 .019027973
        60 .019027973
        70 .002718282
        80 .021746255
        90 .021746255
       100 .019027973
       110 .019027973
           .002718282
```

The following query returns the confidence levels for the error rates that were calculated in the previous query:

```
SELECT department_id "Department",
       APPROX_PERCENTILE(0.25 DETERMINISTIC, 'CONFIDENCE')
         WITHIN GROUP (ORDER BY salary ASC) "Confidence"
FROM employees
GROUP BY department_id
ORDER BY department_id;

Department Confidence
---------- ----------
        10 .997281718
        20 .999660215
        30 .999660215
        40 .997281718
        50 .999611674
        60 .999611674
        70 .997281718
        80 .999660215
        90 .999660215
       100 .999611674
       110 .999611674
           .997281718
```

The following query returns the nondeterministic approximate 25th percentile, 50th percentile, and 75th percentile salaries for each department in the `hr.employees` table. The salaries are sorted in ascending order for the interpolation calculation.

```
SELECT department_id "Department",
       APPROX_PERCENTILE(0.25)
         WITHIN GROUP (ORDER BY salary ASC) "25th Percentile Salary",
       APPROX_PERCENTILE(0.50)
         WITHIN GROUP (ORDER BY salary ASC) "50th Percentile Salary",
       APPROX_PERCENTILE(0.75)
         WITHIN GROUP (ORDER BY salary ASC) "75th Percentile Salary"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;
```

```
Department 25th Percentile Salary 50th Percentile Salary 75th Percentile Salary
---------- ---------------------- ---------------------- ----------------------
        10                   4400                   4400                   4400
        20                   6000                   6000                  13000
        30                   2600                   2800                   3100
        40                   6500                   6500                   6500
        50                   2600                   3100                   3600
        60                   4800                   4800                   6000
        70                  10000                  10000                  10000
        80                   7300                   8800                  10000
        90                  17000                  17000                  24000
       100                   7700                   7800                   9000
       110                   8300                   8300                  12008
                             7000                   7000                   7000
```

APPROX_PERCENTILE_AGG

Syntax

Purpose

`APPROX_PERCENTILE_AGG` takes as its input a column of details containing approximate percentile information, and enables you to perform aggregations of that information.

For *`detail`*, specify a column of details created by the `APPROX_PERCENT_DETAIL` function or the `APPROX_PERCENTILE_AGG` function. This column is of data type `BLOB`.

You can specify this function in a `SELECT` statement with a `GROUP BY` clause to aggregate the information contained in the details within each group of rows and return a single detail for each group.

This function returns a `BLOB` value, called a detail, which contains approximate percentile information in a special format. You can store details returned by this function in a table or materialized view, and then again use the `APPROX_PERCENTILE_AGG` function to further aggregate those details, or use the `TO_APPROX_PERCENTILE` function to convert the details to specified percentile values.

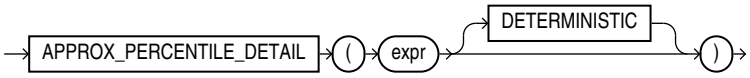
> **See Also:**
>
> - APPROX_PERCENTILE_DETAIL
> - TO_APPROX_PERCENTILE

Examples

Refer to APPROX_PERCENTILE_AGG: Examples for examples of using the `APPROX_PERCENTILE_AGG` function in conjunction with the `APPROX_PERCENTILE_DETAIL` and `TO_APPROX_PERCENTILE` functions.

APPROX_PERCENTILE_DETAIL

Syntax



Purpose

`APPROX_PERCENTILE_DETAIL` calculates approximate percentile information for the values of *`expr`* and returns a `BLOB` value, called a detail, which contains that information in a special format.

The acceptable data types for *`expr`* depend on the algorithm that you specify with the `DETERMINISTIC` clause. Refer to the DETERMINISTIC clause for more information.

This function is commonly used with the `GROUP BY` clause in a `SELECT` statement. It calculates approximate percentile information for *`expr`* within each group of rows and returns a single detail for each group.

The details returned by `APPROX_PERCENTILE_DETAIL` can be used as input to the `APPROX_PERCENTILE_AGG` function, which enables you to perform aggregations of the details, or the `TO_APPROX_PERCENTILE` function, which converts a detail to a specified percentile value. You can use these three functions together to perform resource-intensive approximate percentile calculations once, store the resulting details, and then perform efficient aggregations and queries on those details. For example:

1. Use the `APPROX_PERCENTILE_DETAIL` function to perform approximate percentile calculations and store the resulting details in a table or materialized view. These could be highly-granular percentile details, such as income percentile information for cities.
2. Use the `APPROX_PERCENTILE_AGG` function to aggregate the details obtained in the previous step and store the resulting details in a table or materialized view. These could be details of lower granularity, such as income percentile information for states.
3. Use the `TO_APPROX_PERCENTILE` function to convert the stored detail values to percentile values. You can use the `TO_APPROX_PERCENTILE` function to query detail values created by the `APPROX_PERCENTILE_DETAIL` function or the `APPROX_PERCENTILE_AGG` function.

DETERMINISTIC

This clause lets you control the type of algorithm used to calculate the approximate percentile values.

- If you specify `DETERMINISTIC`, then this function calculates deterministic approximate percentile information. In this case, *`expr`* must evaluate to a numeric value, or to a value that can be implicitly converted to a numeric value.
- If you omit `DETERMINSTIC`, then this function calculates nondeterministic approximate percentile information. In this case, *`expr`* must evaluate to a numeric or datetime value, or to a value that can be implicitly converted to a numeric or datetime value.

> **See Also:**
>
> - APPROX_PERCENTILE_AGG
> - TO_APPROX_PERCENTILE

Examples

The examples in this section demonstrate how to use the `APPROX_PERCENTILE_DETAIL`, `APPROX_PERCENTILE_AGG`, and `TO_APPROX_PERCENTILE` functions together to perform resource-intensive approximate percentile calculations once, store the resulting details, and then perform efficient aggregations and queries on those details.

APPROX_PERCENTILE_DETAIL: Example

The following statement queries the tables `sh.customers` and `sh.sales` for the monetary amounts for products sold to each customer. The `APPROX_PERCENTILE_DETAIL` function returns the information in a detail, called `city_detail`, for each city in which customers reside. The returned details are stored in a materialized view called `amt_sold_by_city_mv`.

```
CREATE MATERIALIZED VIEW amt_sold_by_city_mv
ENABLE QUERY REWRITE AS
SELECT c.country_id country,
       c.cust_state_province state,
       c.cust_city city,
       APPROX_PERCENTILE_DETAIL(s.amount_sold) city_detail
FROM customers c, sales s
WHERE c.cust_id = s.cust_id
GROUP BY c.country_id, c.cust_state_province, c.cust_city;
```

APPROX_PERCENTILE_AGG: Examples

The following statement uses the `APPROX_PERCENTILE_AGG` function to read the details stored in `amt_sold_by_city_mv` and create aggregated details that contain the monetary amounts for products sold to customers in each state. These aggregated details are stored in a materialized view called `amt_sold_by_state_mv`.

```
CREATE MATERIALIZED VIEW amt_sold_by_state_mv AS
SELECT country,
       state,
       APPROX_PERCENTILE_AGG(city_detail) state_detail
FROM amt_sold_by_city_mv
GROUP BY country, state;
```

The following statement is similar to the previous statement, except it creates aggregated details that contain the approximate monetary amounts for products sold to customers in each country. These aggregated details are stored in a materialized view called `amt_sold_by_country_mv`.

```
CREATE MATERIALIZED VIEW amt_sold_by_country_mv AS
  SELECT country,
         APPROX_PERCENTILE_AGG(city_detail) country_detail
  FROM amt_sold_by_city_mv
  GROUP BY country;
```

TO_APPROX_PERCENTILE: Examples

The following statement uses the `TO_APPROX_PERCENTILE` function to query the details stored in `amt_sold_by_city_mv` and return approximate 25th percentile, 50th percentile, and 75th percentile values for monetary amounts for products sold to customers in each city:

```
SELECT country,
       state,
       city,
       TO_APPROX_PERCENTILE(city_detail, .25, 'NUMBER') "25th Percentile",
       TO_APPROX_PERCENTILE(city_detail, .50, 'NUMBER') "50th Percentile",
       TO_APPROX_PERCENTILE(city_detail, .75, 'NUMBER') "75th Percentile"
FROM amt_sold_by_city_mv
ORDER BY country, state, city;

COUNTRY STATE        CITY           25th Percentile 50th Percentile 75th Percentile
------- ------------ -------------- --------------- --------------- ---------------
  52769 Kuala Lumpur Kuala Lumpur             19.29            38.1           53.84
  52769 Penang       Batu Ferringhi           21.51           42.09           57.26
  52769 Penang       Georgetown               19.15           33.25           56.12
  52769 Selangor     Klang                    18.08           32.06           51.29
  52769 Selangor     Petaling Jaya            19.29           35.43            60.2
. . .
```

The following statement uses the `TO_APPROX_PERCENTILE` function to query the details stored in `amt_sold_by_state_mv` and return approximate 25th percentile, 50th percentile, and 75th percentile values for monetary amounts for products sold to customers in each state:

```
SELECT country,
       state,
       TO_APPROX_PERCENTILE(state_detail, .25, 'NUMBER') "25th Percentile",
       TO_APPROX_PERCENTILE(state_detail, .50, 'NUMBER') "50th Percentile",
       TO_APPROX_PERCENTILE(state_detail, .75, 'NUMBER') "75th Percentile"
FROM amt_sold_by_state_mv
ORDER BY country, state;

COUNTRY STATE        25th Percentile 50th Percentile 75th Percentile
------- ------------ --------------- --------------- ---------------
  52769 Kuala Lumpur           19.29            38.1           53.84
  52769 Penang                 20.19           36.84           56.12
  52769 Selangor               16.97           32.41           52.69
  52770 Drenthe                16.76            31.7           53.89
  52770 Flevopolder            20.38           39.73           61.81
. . .
```

The following statement uses the `TO_APPROX_PERCENTILE` function to query the details stored in `amt_sold_by_country_mv` and return approximate 25th percentile, 50th percentile, and 75th percentile values for monetary amounts for products sold to customers in each country:

```
SELECT country,
       TO_APPROX_PERCENTILE(country_detail, .25, 'NUMBER') "25th Percentile",
       TO_APPROX_PERCENTILE(country_detail, .50, 'NUMBER') "50th Percentile",
       TO_APPROX_PERCENTILE(country_detail, .75, 'NUMBER') "75th Percentile"
FROM amt_sold_by_country_mv
ORDER BY country;

  COUNTRY 25th Percentile 50th Percentile 75th Percentile
--------- --------------- --------------- ---------------
    52769            19.1           35.43           52.78
    52770           19.29           38.99           59.58
    52771           11.99           44.99          561.47
    52772           18.08           33.72           54.16
    52773           15.67           29.61           50.65
. . .
```

APPROX_PERCENTILE_AGG takes as its input a column of details containing approximate percentile information, and enables you to perform aggregations of that information. The following statement demonstrates how approximate percentile details can interpreted by APPROX_PERCENTILE_AGG to provide an input to the TO_APPROX_PERCENTILE function. Like the previous example, this query returns approximate 25th percentile values for monetary amounts for products sold to customers in each country. Note that the results are identical to those returned for the 25th percentile in the previous example.

```
SELECT country,
       TO_APPROX_PERCENTILE(APPROX_PERCENTILE_AGG(city_detail), .25, 'NUMBER') "25th
Percentile"
FROM amt_sold_by_city_mv
GROUP BY country
ORDER BY country;

  COUNTRY 25th Percentile
---------- ---------------
     52769            19.1
     52770           19.29
     52771           11.99
     52772           18.08
     52773           15.67
. . .
```

Query Rewrite and Materialized Views Based on Approximate Queries: Example

In APPROX_PERCENTILE_DETAIL: Example, the ENABLE QUERY REWRITE clause is specified when creating the materialized view amt_sold_by_city_mv. This enables queries that contain approximation functions, such as APPROX_MEDIAN or APPROX_PERCENTILE, to be rewritten using the materialized view.

For example, ensure that query rewrite is enabled at either the database level or for the current session, and run the following query:

```
SELECT c.country_id country,
       APPROX_MEDIAN(s.amount_sold) amount_median
FROM customers c, sales s
WHERE c.cust_id = s.cust_id
GROUP BY c.country_id;
```

Explain the plan by querying DBMS_XPLAN:

```
SET LINESIZE 300
SET PAGESIZE 0
COLUMN plan_table_output FORMAT A150

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(format=>'BASIC'));
```

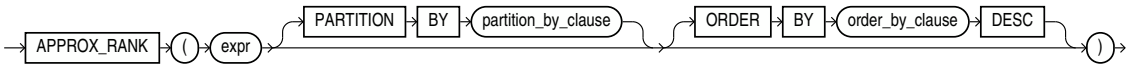
As shown in the following plan, the optimizer used the materialized view amt_sold_by_city_mv for the query:

```
EXPLAINED SQL STATEMENT:
------------------------
SELECT c.country_id country, APPROX_MEDIAN(s.amount_sold)
amount_median FROM customers c, sales s WHERE c.cust_id = s.cust_id
GROUP BY c.country_id

Plan hash value: 2232676046

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

```
|   0 | SELECT STATEMENT              |                     |
|   1 |  HASH GROUP BY APPROX         |                     |
|   2 |   MAT_VIEW REWRITE ACCESS FULL| AMT_SOLD_BY_CITY_MV |
-----------------------------------------------------------
```

APPROX_RANK

Syntax



Purpose

`APPROX_RANK` returns the approximate value in a group of values.

This function takes an optional `PARTITION BY` clause followed by a mandatory `ORDER BY ... DESC` clause. The `PARTITION BY` key must be a subset of the `GROUP BY` key. The `ORDER BY` clause must include either `APPROX_COUNT` or `APPROX_SUM`.

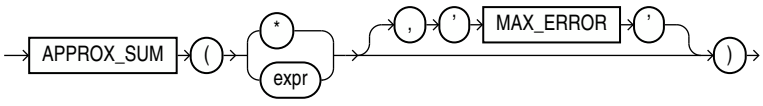
Examples

The query returns the jobs that are among the top 10 total salary per department. For each job, the total salary and ranking is also given.

```
SELECT job_id,
            APPROX_SUM(sal),
      APPROX_RANK(PARTITION BY department_id ORDER BY APPROX_SUM(salary)
DESC)
FROM employees
GROUP BY department_id, job_id
HAVING
   APPROX_RANK(
   PARTITION BY department_id
   ORDER BY APPROX_SUM (salary)
   DESC) <= 10;
```

APPROX_SUM

Syntax



Purpose

`APPROX_SUM` returns the approximate sum of an expression. If you supply `MAX_ERROR` as the second argument, then the function returns the maximum error between the actual and approximate sum.

You must use this function with a corresponding `APPROX_RANK` function in the `HAVING` clause. If a query uses `APPROX_COUNT`, `APPROX_SUM`, or `APPROX_RANK`, then the query must not use any other aggregation functions.

Note that `APPROX_SUM` returns an error when the input is a negative number.

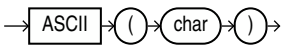
Examples

The following query returns the 10 job types within every department that have the highest aggregate salary:

```
SELECT department_id, job_id,
       APPROX_SUM(salary)
FROM   employees
GROUP BY department_id, job_id
HAVING
  APPROX_RANK (
  PARTITION BY department_id
  ORDER BY APPROX_SUM(salary)
  DESC ) <= 10;
```

ASCII

Syntax



Purpose

`ASCII` returns the decimal representation in the database character set of the first character of *`char`*.

`char` can be of data type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The value returned is of data type `NUMBER`. If your database character set is 7-bit ASCII, then this function returns an ASCII value. If your database character set is EBCDIC Code, then this function returns an EBCDIC value. There is no corresponding EBCDIC character function.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> Data Type Comparison Rules for more information

Examples

The following example returns employees whose last names begin with the letter L, whose ASCII equivalent is 76:

```
SELECT last_name
  FROM employees
  WHERE ASCII(SUBSTR(last_name, 1, 1)) = 76
```

```
  ORDER BY last_name;

LAST_NAME
-------------------------
Ladwig
Landry
Lee
Livingston
Lorentz
```

ASCIISTR

Syntax

Purpose

`ASCIISTR` takes as its argument a string, or an expression that resolves to a string, in any character set and returns an ASCII version of the string in the database character set. Non-ASCII characters are converted to the form `\xxxx`, where `xxxx` represents a UTF-16 code unit.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for information on Unicode character sets and character semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `ASCIISTR`

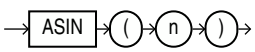
Examples

The following example returns the ASCII string equivalent of the text string "`ABÄCDE`":

```
SELECT ASCIISTR('ABÄCDE')
  FROM DUAL;

ASCIISTR('
----------
AB\00C4CDE
```

ASIN

Syntax



Purpose

`ASIN` returns the arc sine of *n*. The argument *n* must be in the range of -1 to 1, and the function returns a value in the range of -*pi*/2 to *pi*/2, expressed in radians.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

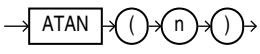
Examples

The following example returns the arc sine of .3:

```
SELECT ASIN(.3) "Arc_Sine"
  FROM DUAL;

 Arc_Sine
----------
.304692654
```

ATAN

Syntax



Purpose

`ATAN` returns the arc tangent of *n*. The argument *n* can be in an unbounded range and returns a value in the range of -*pi*/2 to *pi*/2, expressed in radians.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> ATAN2 for information about the `ATAN2` function and Table 2-9 for more information on implicit conversion

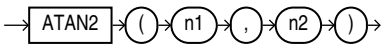
Examples

The following example returns the arc tangent of .3:

```
SELECT ATAN(.3) "Arc_Tangent"
  FROM DUAL;

Arc_Tangent
----------
.291456794
```

ATAN2

Syntax



Purpose

`ATAN2` returns the arc tangent of *n1* and *n2*. The argument *n1* can be in an unbounded range and returns a value in the range of *-pi* to *pi*, depending on the signs of *n1* and *n2*, expressed in radians.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If any argument is `BINARY_FLOAT` or `BINARY_DOUBLE`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns `NUMBER`.

> **See Also:**
>
> ATAN for information on the `ATAN` function and Table 2-9 for more information on implicit conversion

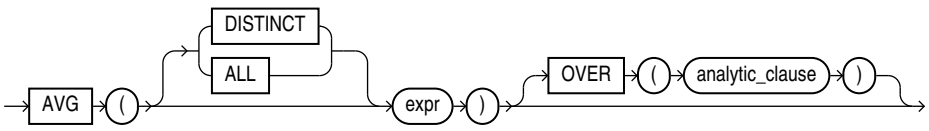
Examples

The following example returns the arc tangent of .3 and .2:

```
SELECT ATAN2(.3, .2) "Arc_Tangent2"
  FROM DUAL;

Arc_Tangent2
------------
  .982793723
```

AVG

Syntax



> **See Also:**
>
> Analytic Functions for information on syntax, semantics, and restrictions

Purpose

`AVG` returns average value of *`expr`*.

It takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type or an interval data type.

The function returns the same data type as the numeric data type of the argument. If the input is an interval, this returns an interval with the same units as the input.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

If you specify `DISTINCT`, then you can specify only the *`query_partition_clause`* of the *`analytic_clause`*. The *`order_by_clause`* and *`windowing_clause`* are not allowed.

> **See Also:**
>
> About SQL Expressions for information on valid forms of *`expr`* and Aggregate Functions

Vector Aggregate Operations

You can use `AVG` on vectors to return the average on non-null inputs.

`expr` must evaluate to `VECTOR` and must not be `BINARY` vectors. The returned vector has the same number of dimensions as the input, and the format is always `FLOAT64`. For flexible number of dimensions, all inputs must have the same number of dimensions within each aggregation group.

NULL vectors are ignored. They are not counted when calculating the average vector. If all inputs within an aggregation group are NULL, the result is NULL for that group. If a certain dimension overflows when applying arithmetic operations, an error is raised.

Rules

- `DISTINCT` syntax is not allowed.
- Only `GROUP BY` and `GROUP BY ROLLUP` are supported.
- Analytic functions are not supported for input arguments of type `VECTOR`.

See *Arithmetic Operators*of the *AI Vector Search User's Guide* for examples.

Aggregate Example

The following example calculates the average salary of all employees in the `hr.employees` table:

```
SELECT AVG(salary) "Average"
  FROM employees;

      Average
--------------
    6461.83178
```

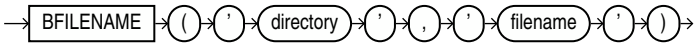
Analytic Example

The following example calculates, for each employee in the `employees` table, the average salary of the employees reporting to the same manager who were hired in the range just before through just after the employee:

```
SELECT manager_id, last_name, hire_date, salary,
       AVG(salary) OVER (PARTITION BY manager_id ORDER BY hire_date
  ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS c_mavg
  FROM employees
  ORDER BY manager_id, hire_date, salary;

MANAGER_ID LAST_NAME                 HIRE_DATE     SALARY     C_MAVG
---------- ------------------------- --------- ---------- ----------
       100 De Haan                   13-JAN-01      17000      14000
       100 Raphaely                  07-DEC-02      11000 11966.6667
       100 Kaufling                  01-MAY-03       7900 10633.3333
       100 Hartstein                 17-FEB-04      13000 9633.33333
       100 Weiss                     18-JUL-04       8000 11666.6667
       100 Russell                   01-OCT-04      14000 11833.3333
       100 Partners                  05-JAN-05      13500 13166.6667
       100 Errazuriz                 10-MAR-05      12000 11233.3333
. . .
```

BFILENAME

Syntax



Purpose

`BFILENAME` returns a `BFILE` locator that is associated with a physical LOB binary file on the server file system.

- '*directory*' is a database object that serves as an alias for a full path name on the server file system where the files are actually located.
- '*filename*' is the name of the file in the server file system.

You must create the directory object and associate a `BFILE` value with a physical file before you can use them as arguments to `BFILENAME` in a SQL or PL/SQL statement, `DBMS_LOB` package, or OCI operation.

You can use this function in two ways:

- In a DML statement to initialize a `BFILE` column
- In a programmatic interface to access `BFILE` data by assigning a value to the `BFILE` locator

The directory argument is case sensitive. You must ensure that you specify the directory object name exactly as it exists in the data dictionary. For example, if an `"Admin"` directory object was created using mixed case and a quoted identifier in the `CREATE DIRECTORY` statement, then when using the `BFILENAME` function you must refer to the directory object as `'Admin'`. You must specify the filename argument according to the case and punctuation conventions for your operating system.

> **See Also:**
>
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* and *Oracle Call Interface Developer's Guide* for more information on LOBs and for examples of retrieving `BFILE` data
> - CREATE DIRECTORY

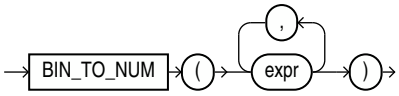
Examples

The following example inserts a row into the sample table `pm.print_media`. The example uses the `BFILENAME` function to identify a binary file on the server file system in the directory `/demo/schema/product_media`. The example shows how the directory database object `media_dir` was created in the `pm` schema.

```
CREATE DIRECTORY media_dir AS '/demo/schema/product_media';

INSERT INTO print_media (product_id, ad_id, ad_graphic)
  VALUES (3000, 31001, BFILENAME('MEDIA_DIR', 'modem_comp_ad.gif'));
```

BIN_TO_NUM

Syntax



Purpose

`BIN_TO_NUM` converts a bit vector to its equivalent number. Each argument to this function represents a bit in the bit vector. This function takes as arguments any numeric data type, or any nonnumeric data type that can be implicitly converted to `NUMBER`. Each *`expr`* must evaluate to 0 or 1. This function returns Oracle `NUMBER`.

`BIN_TO_NUM` is useful in data warehousing applications for selecting groups of interest from a materialized view using grouping sets.

> **See Also:**
>
> - *group_by_clause* for information on `GROUPING SETS` syntax
> - Table 2-9 for more information on implicit conversion
> - *Oracle Database Data Warehousing Guide* for information on data aggregation in general

Examples

The following example converts a binary value to a number:

```
SELECT BIN_TO_NUM(1,0,1,0)
  FROM DUAL;

BIN_TO_NUM(1,0,1,0)
-------------------
                 10
```

The next example converts three values into a single binary value and uses `BIN_TO_NUM` to convert that binary into a number. The example uses a PL/SQL declaration to specify the original values. These would normally be derived from actual data sources.

```
SELECT order_status
  FROM orders
  WHERE order_id = 2441;

ORDER_STATUS
------------
           5
DECLARE
  warehouse NUMBER := 1;
  ground    NUMBER := 1;
  insured   NUMBER := 1;
  result    NUMBER;
BEGIN
  SELECT BIN_TO_NUM(warehouse, ground, insured) INTO result FROM DUAL;
  UPDATE orders SET order_status = result WHERE order_id = 2441;
END;
/
PL/SQL procedure successfully completed.

SELECT order_status
  FROM orders
  WHERE order_id = 2441;

ORDER_STATUS
------------
           7
```

Refer to the examples for BITAND for information on reversing this process by extracting multiple values from a single column value.

BITAND

Syntax

Purpose

The `BITAND` function treats its inputs and its output as vectors of bits; the output is the bitwise `AND` of the inputs.

The types of *expr1* and *expr2* are `NUMBER`, and the result is of type `NUMBER`. If either argument to `BITAND` is `NULL`, the result is `NULL`.

The arguments must be in the range $-(2^{(n-1)})$ .. $((2^{(n-1)})-1)$. If an argument is out of this range, the result is undefined.

The result is computed in several steps. First, each argument A is replaced with the value `SIGN(A)*FLOOR(ABS(A))`. This conversion has the effect of truncating each argument towards zero. Next, each argument A (which must now be an integer value) is converted to an n-bit two's complement binary integer value. The two bit values are combined using a bitwise `AND` operation. Finally, the resulting n-bit two's complement value is converted back to `NUMBER`.

Notes on the BITAND Function

- The current implementation of `BITAND` defines *n* = 128.
- PL/SQL supports an overload of `BITAND` for which the types of the inputs and of the result are all `BINARY_INTEGER` and for which *n* = 32.

Examples

The following example performs an `AND` operation on the numbers 6 (binary 1,1,0) and 3 (binary 0,1,1):

```
SELECT BITAND(6,3)
  FROM DUAL;

BITAND(6,3)
-----------
          2
```

This is the same as the following example, which shows the binary values of 6 and 3. The `BITAND` function operates only on the significant digits of the binary values:

```
SELECT BITAND(
    BIN_TO_NUM(1,1,0),
    BIN_TO_NUM(0,1,1)) "Binary"
  FROM DUAL;

    Binary
----------
         2
```

Refer to the example for BIN_TO_NUM for information on encoding multiple values in a single column value.

The following example supposes that the *order_status* column of the sample table oe.orders encodes several choices as individual bits within a single numeric value. For example, an order still in the warehouse is represented by a binary value 001 (decimal 1). An order being sent by ground transportation is represented by a binary value 010 (decimal 2). An insured package is represented by a binary value 100 (decimal 4). The example uses the DECODE function to provide two values for each of the three bits in the *order_status* value, one value if the bit is turned on and one if it is turned off.

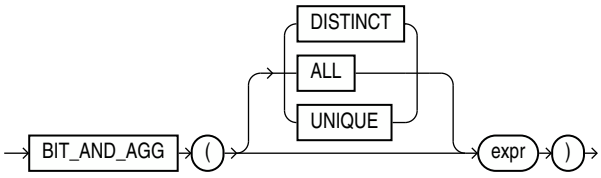
```
SELECT order_id, customer_id, order_status,
    DECODE(BITAND(order_status, 1), 1, 'Warehouse', 'PostOffice') "Location",
    DECODE(BITAND(order_status, 2), 2, 'Ground', 'Air') "Method",
    DECODE(BITAND(order_status, 4), 4, 'Insured', 'Certified') "Receipt"
  FROM orders
  WHERE sales_rep_id = 160
  ORDER BY order_id;

  ORDER_ID CUSTOMER_ID ORDER_STATUS Location   Method Receipt
---------- ----------- ------------ ---------- ------ ---------
      2416         104            6 PostOffice Ground Insured
      2419         107            3 Warehouse  Ground Certified
      2420         108            2 PostOffice Ground Certified
      2423         145            3 Warehouse  Ground Certified
      2441         106            5 Warehouse  Air    Insured
      2455         145            7 Warehouse  Ground Insured
```

For the Location column, BITAND first compares *order_status* with 1 (binary 001). Only significant bit values are compared, so any binary value with a 1 in its rightmost bit (any odd number) will evaluate positively and return 1. Even numbers will return 0. The DECODE function compares the value returned by BITAND with 1. If they are both 1, then the location is "Warehouse". If they are different, then the location is "PostOffice".

The Method and Receipt columns are calculated similarly. For Method, BITAND performs the AND operation on *order_status* and 2 (binary 010). For Receipt, BITAND performs the AND operation on *order_status* and 4 (binary 100).

BIT_AND_AGG

Syntax



Purpose

BIT_AND_AGG is a bitwise aggregation function that returns the result of a bitwise AND operation.

You can use BIT_AND_AGG as part of a GROUP BY query, window function, or as an analytical function. The return type of BIT_AND_AGG is always a number.

Semantics

The keywords DISTINCT or UNIQUE ensure that only unique values in expr are used for computation. UNIQUE is an Oracle-specific keyword and not an ANSI standard.

NULL values in the `expr` column are ignored.

Returns NULL if all rows in the group have NULL `expr` values.

Floating point values are truncated to the integer prior to aggregation. For instance, the value 4.64 is converted to 4, and the value 4.4 is also converted to 4.

Negative numbers are represented in two's complement form internally prior to performing an aggregate operation. The resultant aggregate could be a negative value.

Range of inputs supported: -2 raised to 127 to (2 raised to 127) -1

Numbers are internally converted to a 128b decimal representation prior to aggregation. The resultant aggregate is converted back into an Oracle Number.

For a given set of values, the result of a bitwise aggregate is always deterministic and independent of ordering.

Example 7-2 Use the BIT_AND_AGG Function

Select two numbers and their bitwise representation:

```
SELECT '011' num, bin_to_num(0,1,1) bits FROM dual
  UNION ALL SELECT '101' num, bin_to_num(1,0,1) bits FROM dual;

NUM       BITS
--- ----------
011          3
101          5
```

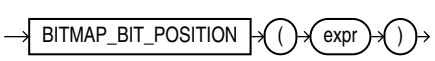
Perform the bitwise AND operation:

```
SELECT bit_and_agg(bits)
  FROM (SELECT '011' num, bin_to_num(0,1,1) bits FROM dual
        UNION ALL SELECT '101' num, bin_to_num(1,0,1) bits FROM dual);

BIT_AND_AGG(BITS)
-----------------
                1
```

Only the first bit is identical in both rows, thus the result is 001, which is the number 1.

BITMAP_BIT_POSITION

Syntax



Purpose

Use `BITMAP_BIT_POSITION` to construct the one-to-one mapping between a number and a bit position.

The argument *`expr`* is of type `NUMBER`. It is the absolute bit position in the bitmap.

`BITMAP_BIT_POSITION` returns a `NUMBER`, the relative bit position.

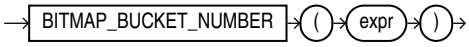
If *expr* is NULL, the function returns NULL.

If *expr* is not an integer, you will see the following error message:

```
Invalid value has been passed to a BITMAP COUNT DISTINCT related operator.
```

BITMAP_BUCKET_NUMBER

Syntax



Purpose

Use `BITMAP_BUCKET_NUMBER` to construct a one-to-one mapping between a number and a bit position in a bitmap.

The argument *expr* is of type `NUMBER`. It represents the absolute bit position in the bitmap.

`BITMAP_BUCKET_NUMBER` returns a `NUMBER`. It represents the relative bit position.

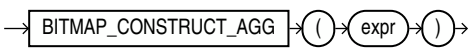
If *expr* is NULL, the function returns NULL.

If *expr* is not an integer, you will see the following error message:

```
Invalid value has been passed to a BITMAP COUNT DISTINCT related operator.
```

BITMAP_CONSTRUCT_AGG

Syntax



Purpose

`BITMAP_CONSTRUCT_AGG` is an aggregation function that operates on bit positions and returns the bitmap representation of the set of all input bit positions. It essentially maintains a bitmap and sets into it all the input bit positions. It returns the representation of the bitmap.

The argument `expr` is of type `NUMBER`.

The return type is of type `BLOB`.

If `expr` is NULL, the function returns NULL.

Restrictions

- The argument must be of `NUMBER` type. If the input value cannot be converted to a natural number, error `ORA-62575` is raised:

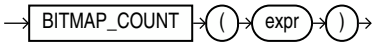
```
62575, 00000, "Invalid value has been passed to a BITMAP COUNT DISTINCT
related operator."
// *Cause: An attempt was made to pass an invalid value to a BITMAP COUNT
DISTINCT operator.
// *Action: Pass only natural number values to BITMAP_CONSTRUCT_AGG.
```

- If the bitmap exceeds the maximum value of a `BLOB`, you will see error `ORA-62577`:

```
62577, 00000, "The bitmap size exceeds maximum size of its SQL data type."
// *Cause: An attempt was made to construct a bitmap larger than its maximum SQL
type size.
// *Action: Break the input to BITMAP_CONSTRUCT_AGG into smaller ranges.
```

BITMAP_COUNT

Syntax



Purpose

`BITMAP_COUNT` is a scalar function that returns the 1-bit count for the input bitmap.

The argument `expr` is of type `BLOB`.

It returns a `NUMBER` representing the count of bits set in its input.

If `expr` is NULL, it returns 0.

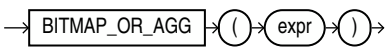
Restrictions

The argument must be of type `BLOB`type. The argument is expected to be a bitmap produced by `BITMAP_CONSTRUCT_AGG` or, recursively, by `BITMAP_OR_AGG`. Any other input results in `ORA-62578`:

```
62578, 00000, "The input is not a valid bitmap produced by BITMAP COUNT
DISTINCT related operators."
// *Cause: An attempt was made to pass a bitmap that was not produced by one
of the BITMAP COUNT DISTINCT operators.
// *Action: Only pass bitmaps constructed via BITMAP_CONSTRUCT_AGG or
BITMAP_OR_AGG to BITMAP COUNT DISTINCT related operators.
```

BITMAP_OR_AGG

Syntax



Purpose

BITMAP_OR_AGG is an aggregation function that operates on bitmaps and computes the OR of its inputs.

The argument expr must be of type BLOB.

The return type is of type BLOB. It returns the bitmap representing the OR of all the bitmaps it has aggregated.

The output of BITMAP_OR_AGG is not human-readable. It is meant to be processed by further aggregations via BITMAP_OR_AGG or by the scalar function BITMAP_COUNT.

If expr is NULL, the function returns NULL.

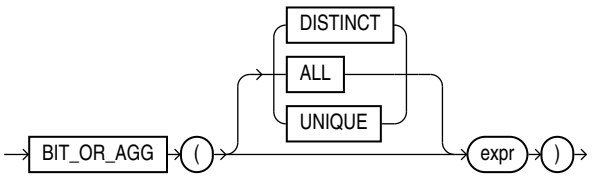
Restrictions

The argument must be of type BLOB. The argument is expected to be a bitmap produced by BITMAP_CONSTRUCT_AGG or, recursively, by BITMAP_OR_AGG. Any other input results in ORA-62578:

```
62578, 00000, "The input is not a valid bitmap produced by BITMAP COUNT
DISTINCT related operators."
// *Cause: An attempt was made to pass a bitmap that was not produced by one
of the BITMAP COUNT DISTINCT operators.
// *Action: Only pass bitmaps constructed via BITMAP_CONSTRUCT_AGG or
BITMAP_OR_AGG to BITMAP COUNT DISTINCT related operators.
```

BIT_OR_AGG

Syntax



Purpose

BIT_OR_AGG is a bitwise aggregation function that returns the result of a bitwise OR operation.

You can use BIT_OR_AGG as part of a GROUP BY query, window function, or as an analytical function. The return type of BIT_OR_AGG is always a number.

Semantics

The keywords DISTINCT or UNIQUE ensure that only unique values in expr are used for computation. UNIQUE is an Oracle-specific keyword and not an ANSI standard.

NULL values in the expr column are ignored.

Returns NULL if all rows in the group have NULL expr values.

Floating point values are truncated to the integer prior to aggregation. For instance, the value 4.64 is converted to 4 and the value 4.4 is also converted to 4.

Negative numbers are represented in two's complement form internally prior to performing an aggregate operation. The resultant aggregate could be a negative value.

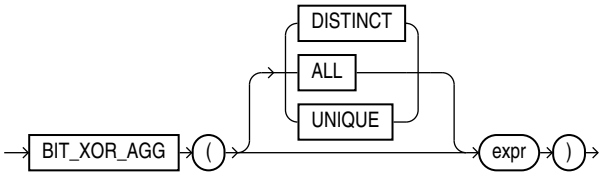
Range of inputs supported: -2 raised to 127 to (2 raised to 127) -1

Numbers are internally converted to a 128b decimal representation prior to aggregation. The resultant aggregate is converted back into an Oracle Number.

For a given set of values, the result of a bitwise aggregate is always deterministic and independent of ordering.

BIT_XOR_AGG

Syntax



Purpose

`BIT_XOR_AGG` is a bitwise aggregation function that returns the result of a bitwise XOR operation.

You can use `BIT_XOR_AGG` as part of a `GROUP BY` query, window function, or as an analytical function. The return type of `BIT_XOR_AGG` is always a number.

Semantics

The keywords `DISTINCT` or `UNIQUE` ensure that only unique values in `expr` are used for computation. `BIT_XOR_AGG` could potentially return a different value when `DISTINCT` is present. `UNIQUE` is an Oracle-specific keyword and not an ANSI standard.

NULL values in the `expr` column are ignored.

Returns NULL if all rows in the group have NULL `expr` values.

Floating point values are truncated to the integer prior to aggregation. For instance, the value 4.64 is converted to 4 and the value 4.4 is also converted to 4.

Negative numbers are represented in two's complement form internally prior to performing an aggregate operation. The resultant aggregate could be a negative value.

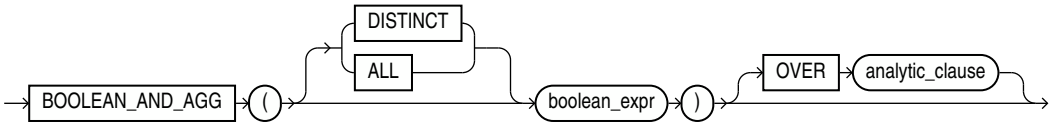
Range of inputs supported: -2 raised to 127 to (2 raised to 127) -1

Numbers are internally converted to a 128b decimal representation prior to aggregation. The resultant aggregate is converted back into an Oracle Number.

For a given set of values, the result of a bitwise aggregate is always deterministic and independent of ordering.

BOOLEAN_AND_AGG

Syntax



Purpose

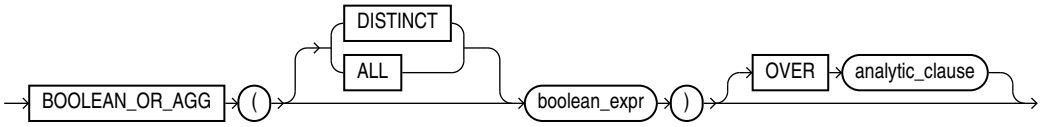
`BOOLEAN_AND_AGG` returns `'TRUE'` if the *`boolean_expr`* evaluates to true for every row that qualifies. Otherwise it returns `'FALSE'`. You can use it as an aggregate or analytic function.

Examples

```
SELECT BOOLEAN_AND_AGG(c2)
    FROM t;

SELECT BOOLEAN_AND_AGG(c2)
    FROM t
    WHERE c1 = 0;

SELECT BOOLEAN_AND_AGG(c2)
    FROM t
    WHERE c2 IS FALSE;

SELECT BOOLEAN_AND_AGG(c2)
    FROM t
    WHERE c2 IS FALSE OR c2 IS NULL;

SELECT BOOLEAN_AND_AGG(c2)
    FROM t
    WHERE c2 IS NOT TRUE OR c2 IS NULL;

SELECT BOOLEAN_AND_AGG(c2)
    FROM t
    WHERE c2 IS NOT FALSE OR c2 IS NULL;

SELECT BOOLEAN_AND_AGG(c2 OR c2 OR (c2))
    FROM t
    WHERE c2 IS NOT FALSE OR c2 IS NULL;
```

BOOLEAN_OR_AGG

Syntax



Purpose

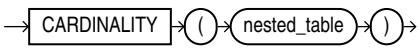
`BOOLEAN_OR_AGG` returns `'TRUE'` if the *`boolean_expr`* evaluates to true for at least one row that qualifies. Otherwise it returns `'FALSE'`. You can use it as an aggregate or analytic function.

Examples

```
SELECT BOOLEAN_OR_AGG(c2)
    FROM t;


SELECT BOOLEAN_OR_AGG(c2)
    FROM t
    WHERE c1 = 0;

SELECT BOOLEAN_OR_AGG(c2)
    FROM t
    WHERE c2 IS TRUE;

SELECT BOOLEAN_OR_AGG(c2)
    FROM t
    WHERE c2 IS TRUE OR c2 IS NULL;

SELECT BOOLEAN_OR_AGG(c2)
    FROM t
    WHERE c2 IS NOT FALSE OR c2 IS NULL;

SELECT BOOLEAN_OR_AGG(c2)
    FROM t
    WHERE c2 IS NOT TRUE OR c2 IS NULL;

SELECT BOOLEAN_OR_AGG(c2 OR c2)
    FROM t
    WHERE c2 IS NOT TRUE OR c2 IS NULL;
```

CARDINALITY

Syntax



Purpose

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

Examples

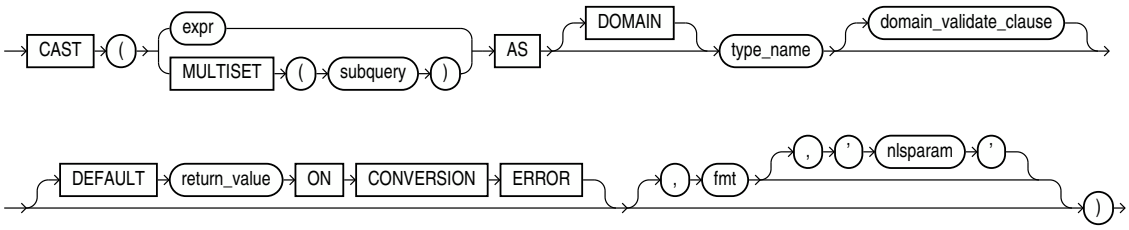
The following example shows the number of elements in the nested table column `ad_textdocs_ntab` of the sample table `pm.print_media`:

```
SELECT product_id, CARDINALITY(ad_textdocs_ntab) cardinality
  FROM print_media
  ORDER BY product_id;

PRODUCT_ID CARDINALITY
---------- -----------
```

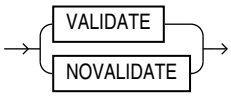
```
2056          3
2268          3
3060          3
3106          3
```

CAST

Syntax



***domain_validate_clause*::=**



Purpose

`CAST` lets you convert built-in data types or collection-typed values of one type into another built-in data type or collection type. You can cast an unnamed operand (such as a date or the result set of a subquery) or a named collection (such as a varray or a nested table) into a type-compatible data type or named collection. The *`type_name`* must be the name of a built-in data type, collection type, or domain name and the operand must be a built-in data type or must evaluate to a collection value.

For the operand, *`expr`* can be either a built-in data type, a collection type, or an instance of an `ANYDATA` type. If *`expr`* is an instance of an `ANYDATA` type, then `CAST` tries to extract the value of the `ANYDATA` instance and return it if it matches the cast target type, otherwise, null will be returned. `MULTISET` informs Oracle Database to take the result set of the subquery and return a collection value. Table 7-1 shows which built-in data types can be cast into which other built-in data types. (`CAST` does not support `LONG`, `LONG RAW`, or the Oracle-supplied types.)

`CAST` does not directly support any of the LOB data types. When you use `CAST` to convert a `CLOB` value into a character data type or a `BLOB` value into the `RAW` data type, the database implicitly converts the LOB value to character or raw data and then explicitly casts the resulting value into the target data type. If the resulting value is larger than the target type, then the database returns an error.

When you use `CAST` ... `MULTISET` to get a collection value, each select list item in the query passed to the `CAST` function is converted to the corresponding attribute type of the target collection element type.

The cells with an 'X' indicate the possible conversions from source to destination data type using `CAST`.

Table 7-1 Casting Built-In Data Types

Destination Data Type	from BINARY_FLOAT, BINARY_DOUBLE	from CHAR, VARCHAR2	from NUMBER/ INTEGER	from DATETIME / INTERVAL (Note 1)	from RAW	from ROWID, UROWID (Note 2)	from NCHAR, NVARCHAR2	from BOOLEAN
to BINARY_FLOAT, BINARY_DOUBLE	X (Note 3)	X (Note 3)	X (Note 3)	--	--	--	X (Note 3)	X(Note 4)
to CHAR, VARCHAR2	X	X	X	X	X	X	X	X (Note 5)
to NUMBER/ INTEGER	X (Note 3)	X (Note 3)	X (Note 3)	--	--	--	X (Note 3)	X(Note 4)
to DATETIME/ INTERVAL	--	X (Note 3)	--	X (Note 3)	--	--	--	--
to RAW	--	X	--	--	X	--	X	--
to ROWID, UROWID	--	X	--	--	--	X	--	--
to NCHAR, NVARCHAR2	X	--	X	X	X	X	X	X(Note 5)
to BOOLEAN	X(Note 4)	X(Note 6)	x (Note 4)	--	--	--	X(Note 6)	X

Note 1: Datetime/interval includes `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIMEZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL DAY TO SECOND`, and `INTERVAL YEAR TO MONTH`.

Note 2: You cannot cast a `UROWID` to a `ROWID` if the `UROWID` contains the value of a `ROWID` of an index-organized table.

Note 3: You can specify the `DEFAULT` *`return_value`* `ON CONVERSION ERROR` clause for this type of conversion. You can specify the *`fmt`* and *`nlsparam`* clauses for this type of conversion with the following exceptions: you cannot specify fmt when converting to `INTERVAL DAY TO SECOND`, and you cannot specify *`fmt`* or *`nlsparam`* when converting to `INTERVAL YEAR TO MONTH`.

Note 4: Casting Between Boolean and Numeric

When casting `BOOLEAN` to numeric :

- If the boolean value is true, then resulting value is 1.
- If the boolean value is false, then resulting value is 0.

When casting numeric to `BOOLEAN` :

- If the numeric value is non-zero (e.g., 1, 2, -3, 1.2), then resulting value is true.
- If the numeric value is zero, then resulting value is false.

Note 5: Casting Between Boolean and Char(n), NCHAR(n)

When casting `BOOLEAN` to `VARCHAR(n)`, `NVARCHAR(n)`

- If the boolean value is true and `n` is not less than 4, then resulting value is true.
- If the boolean value is false and `n` is not less than 5, then resulting value is false.
- Otherwise, a data exception error is raised.

Note 6: Casting Character Strings to Boolean

When casting a character string to boolean, you must trim both leading and trailing spaces of the character string first. If the resulting character string is one of the accepted literals used to determine a valid boolean value, then the result is that valid boolean value.

If you want to cast a named collection type into another named collection type, then the elements of both collections must be of the same type.

> **See Also:**
>
> - Boolean Data Type
> - Implicit Data Conversion for information on how Oracle Database implicitly converts collection type data into character data and Security Considerations for Data Conversion
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `CAST` when it is a character value

MULTISET

If the result set of *subquery* can evaluate to multiple rows, then you must specify the `MULTISET` keyword. The rows resulting from the subquery form the elements of the collection value into which they are cast. Without the `MULTISET` keyword, the subquery is treated as a scalar subquery.

Restriction on MULTISET

If you specify the `MULTISET` keyword, then you cannot specify the `DEFAULT` *return_value* `ON CONVERSION ERROR`, *fmt*, or *nlsparam* clauses.

DOMAIN

The `DOMAIN` clause specifies that `type_name` is a domain. `type_name` must be the name of a domain that the user has execute privileges on.

domain_validate_clause

This clause is only valid when casting to domain types. It controls whether domain constraints are applied when converting *expr* to *type_name*. If *type_name* is a domain with constraints, and *domain_validate_clause* is not specified, enabled constraints will be applied to *expr*. Any disabled constraints are ignored.

VALIDATE

All the domain constraints are applied to *expr*, regardless of their state.

NOVALIDATE

None of the domain constraints are applied to *expr*, regardless of their state.

DEFAULT *return_value* ON CONVERSION ERROR

This clause allows you to specify the value returned by this function if an error occurs while converting *expr* to *type_name*. This clause has no effect if an error occurs while evaluating *expr*.

This clause is valid if *expr* evaluates to a character string of type CHAR, VARCHAR2, NCHAR, or NVARCHAR2, and *type_name* is BINARY_DOUBLE, BINARY_FLOAT, DATE, INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH, NUMBER, TIMESTAMP, TIMESTAMP WITH TIME ZONE, or TIMESTAMP WITH LOCAL TIME ZONE.

The *return_value* can be a string literal, null, constant expression, or a bind variable, and must evaluate to null or a character string of type CHAR, VARCHAR2, NCHAR, or NVARCHAR2. If *return_value* cannot be converted to *type_name*, then the function returns an error.

fmt* and *nlsparam

The *fmt* argument lets you specify a format model and the *nlsparam* argument lets you specify NLS parameters. If you specify these arguments, then they are applied when converting *expr* and *return_value*, if specified, to *type_name*.

You can specify *fmt* and *nlsparam* if *type_name* is one of the following data types:

- BINARY_DOUBLE

  If you specify BINARY_DOUBLE, then the optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_BINARY_DOUBLE function. Refer to TO_BINARY_DOUBLE for more information.

- BINARY_FLOAT

  If you specify BINARY_FLOAT, then the optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_BINARY_FLOAT function. Refer to TO_BINARY_FLOAT for more information.

- DATE

  If you specify DATE, then the optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_DATE function. Refer to TO_DATE for more information.

- NUMBER

  If you specify NUMBER, then the optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_NUMBER function. Refer to TO_NUMBER for more information.

- TIMESTAMP

  If you specify TIMESTAMP, then the optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_TIMESTAMP function. If you omit *fmt*, then *expr* must be in the default format of the TIMESTAMP data type, which is determined explicitly by the NLS_TIMESTAMP_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. Refer to TO_TIMESTAMP for more information.

- TIMESTAMP WITH TIME ZONE

  If you specify TIMESTAMP WITH TIME ZONE, then the optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_TIMESTAMP_TZ function. If you omit *fmt*, then *expr* must be in the default format of the TIMESTAMP WITH TIME ZONE data type, which is determined explicitly by the NLS_TIMESTAMP_TZ_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. Refer to TO_TIMESTAMP_TZ for more information.

- TIMESTAMP WITH LOCAL TIME ZONE

  If you specify TIMESTAMP WITH LOCAL TIME ZONE then the optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_TIMESTAMP function. If you omit *fmt*, then *expr* must be in the default format of the TIMESTAMP data type, , which is determined explicitly by the NLS_TIMESTAMP_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. Refer to TO_TIMESTAMP for more information.

Built-In Data Type Examples

The following examples use the CAST function with scalar data types. The first example converts text to a timestamp value by applying the format model provided in the session parameter NLS_TIMESTAMP_FORMAT. If you want to avoid dependency on this NLS parameter, then you can use the TO_DATE as shown in the second example.

```
SELECT CAST('22-OCT-1997'
       AS TIMESTAMP WITH LOCAL TIME ZONE)
  FROM DUAL;

SELECT CAST(TO_DATE('22-Oct-1997', 'DD-Mon-YYYY')
       AS TIMESTAMP WITH LOCAL TIME ZONE)
  FROM DUAL;
```

In the preceding example, TO_DATE converts from text to DATE, and CAST converts from DATE to TIMESTAMP WITH LOCAL TIME ZONE, interpreting the date in the session time zone (SESSIONTIMEZONE).

```
SELECT product_id, CAST(ad_sourcetext AS VARCHAR2(30)) text
  FROM print_media
  ORDER BY product_id;
```

The following examples return a default value if an error occurs while converting the specified value to the specified data type. In these examples, the conversions occurs without error.

```
SELECT CAST(200
       AS NUMBER
       DEFAULT 0 ON CONVERSION ERROR)
  FROM DUAL;

SELECT CAST('January 15, 1989, 11:00 A.M.'
       AS DATE
       DEFAULT NULL ON CONVERSION ERROR,
       'Month dd, YYYY, HH:MI A.M.')
  FROM DUAL;

SELECT CAST('1999-12-01 11:00:00 -8:00'
       AS TIMESTAMP WITH TIME ZONE
       DEFAULT '2000-01-01 01:00:00 -8:00' ON CONVERSION ERROR,
       'YYYY-MM-DD HH:MI:SS TZH:TZM',
       'NLS_DATE_LANGUAGE = American')
  FROM DUAL;
```

In the following example, an error occurs while converting 'N/A' to a NUMBER value. Therefore, the CAST function returns the default value of 0.

```
SELECT CAST('N/A'
       AS NUMBER
       DEFAULT '0' ON CONVERSION ERROR)
  FROM DUAL;
```

The following example converts data types VARCHAR2, NUMBER as BOOLEAN:

```
SELECT
  CAST ( 'yes' AS BOOLEAN ),
  CAST ( true AS NUMBER ),
  CAST ( false AS VARCHAR2(10) ) ;

CAST('YES'ASBOOLEAN) CAST(TRUEASNUMBER) CAST(FALSE
```

```
------------------- ----------------- ----------
               TRUE                 1 FALSE
```

Collection Examples

The `CAST` examples that follow build on the `cust_address_typ` found in the sample order entry schema, `oe`.

```
CREATE TYPE address_book_t AS TABLE OF cust_address_typ;

CREATE TYPE address_array_t AS VARRAY(3) OF cust_address_typ;

CREATE TABLE cust_address (
  custno            NUMBER,
  street_address    VARCHAR2(40),
  postal_code       VARCHAR2(10),
  city              VARCHAR2(30),
  state_province    VARCHAR2(10),
  country_id        CHAR(2));

CREATE TABLE cust_short (custno NUMBER, name VARCHAR2(31));

CREATE TABLE states (state_id NUMBER, addresses address_array_t);
```

This example casts a subquery:

```
SELECT s.custno, s.name,
       CAST(MULTISET(SELECT ca.street_address,
                            ca.postal_code,
                            ca.city,
                            ca.state_province,
                            ca.country_id
                       FROM cust_address ca
                       WHERE s.custno = ca.custno)
       AS address_book_t)
  FROM cust_short s
  ORDER BY s.custno;
```

`CAST` converts a varray type column into a nested table:

```
SELECT CAST(s.addresses AS address_book_t)
  FROM states s
  WHERE s.state_id = 111;
```

The following objects create the basis of the example that follows:

```
CREATE TABLE projects
  (employee_id NUMBER, project_name VARCHAR2(10));

CREATE TABLE emps_short
  (employee_id NUMBER, last_name VARCHAR2(10));

CREATE TYPE project_table_typ AS TABLE OF VARCHAR2(10);
```

The following example of a `MULTISET` expression uses these objects:

```
SELECT e.last_name,
       CAST(MULTISET(SELECT p.project_name
                       FROM projects p
```

```
                    WHERE p.employee_id = e.employee_id
                    ORDER BY p.project_name)
      AS project_table_typ)
  FROM emps_short e
  ORDER BY e.last_name;
```

The following example casts the string `'yes'` to a boolean value, the boolean value `true` to a `NUMBER` and the boolean value `false` to `VARCHAR2(10)`:

```
SELECT
    CAST ( 'yes' AS BOOLEAN ),
    CAST ( true AS NUMBER ),
    CAST ( false AS VARCHAR2(10) );

   CAST('YES'ASBOOLEAN) CAST(TRUEASNUMBER) CAST(FALSE
   -------------------- ------------------ ----------
                TRUE                  1      FALSE
```

Domain Examples

The following example creates the domain `DAY_OF_WEEK` with a disabled check constraint. The first query omits the *`domain_validate_clause`*, so uses the constraint state to determine whether to verify the value. As this is disabled, the database does not check the value.

The second query uses the `VALIDATE` clause. This applies the constraint to "`N/A`", even though it's disabled. The value "`N/A`" is not in the list permitted by the constraints, so `CAST` raises an exception.

```
CREATE DOMAIN day_of_week AS VARCHAR2(3 CHAR)
  CONSTRAINT CHECK (day_of_week IN('MON','TUE','WED','THU','FRI','SAT','SUN'))
  DISABLE;


SELECT CAST ( 'N/A' AS day_of_week ) use_constraint_state;

USE_CONSTRAINT_STATE
--------------------
N/A


SELECT CAST ( 'N/A' AS day_of_week VALIDATE ) apply_constraints;

ORA-11513: CAST AS DOMAIN has failed due to domain constraints.
```

The following example creates the domain `DAY_OF_WEEK` with an enabled check constraint. The first query omits the *`domain_validate_clause`*, so uses the constraint state to determine whether to verify the value. As this is enabled, the database applies the constraint to "`N/A`". This is not in the list of permitted values so `CAST` raises an error.

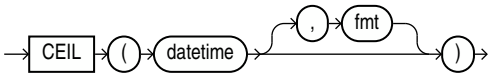
The second query uses the `NOVALIDATE` clause. This ignores the constraint even though it is enabled and the statement completes without error.

```
CREATE DOMAIN day_of_week AS VARCHAR2(3 CHAR)
  CONSTRAINT CHECK (day_of_week IN('MON','TUE','WED','THU','FRI','SAT','SUN'))
  ENABLE;


SELECT CAST ( 'N/A' AS day_of_week ) use_constraint_state;
```

```
ORA-11513: CAST AS DOMAIN has failed due to domain constraints.


SELECT CAST ( 'N/A' AS DOMAIN day_of_week NOVALIDATE ) ignore_constraints;

IGNORE_CONSTRAINTS
------------------
N/A
```

CEIL (datetime)

Syntax



Purpose

`CEIL(datetime)` returns the date or the timestamp rounded up to the unit specified by the second argument `fmt`, the format model. If the input value is already truncated to the specified unit, then the return value is the same as the input. That is, if `datetime` = `TRUNC(datetime, fmt)`, then `CEIL(datetime, fmt)` = `datetime`. For example, `CEIL(DATE '2023-02-01', 'MONTH')` returns February 1 2023.

This function is not sensitive to the `NLS_CALENDAR` session parameter. It operates according to the rules of the Gregorian calendar. The value returned is always of data type `DATE`, even if you specify a different datetime data type for the argument. If you do not specify the second argument, the default format model '`DD`' is used.

> **See Also:**
>
> Refer to CEIL, FLOOR, ROUND, and TRUNC Date Functions for the permitted format models to use in *`fmt`*.

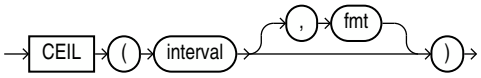
Examples

For these examples `NLS_DATE_FORMAT` is set:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';

SELECT CEIL(TO_DATE ('28-FEB-2023','DD-MON-YYYY'), 'MM') AS month_ceiling;

MONTH_CEILING
--------------------
01-MAR-2023 00:00:00

SELECT CEIL(TO_TIMESTAMP ('28-FEB-2023 14:10:10','DD-MON-YYYY HH24:MI:SS'),'HH24') AS
hour_ceiling;

HOUR_CEILING
--------------------
28-FEB-2023 15:00:00
```

CEIL (interval)

Syntax



Purpose

`CEIL(interval)` returns the interval rounded up to the unit specified by the second argument *`fmt`*, the format model. If the first argument is truncated to the units of *`fmt`*, the output equals the input. For example, `CEIL(INTERVAL '+123-0' YEAR(3) TO MONTH)` returns 123 years and no months (+123-00).

The result of `CEIL(interval)` is never smaller than *`interval`*. The result precision for year and day is the input precision for year plus one and day plus one, since `CEIL(interval)` can have overflow. If an interval already has the maximum precision for year and day, the statement compiles but errors at runtime.

For `INTERVAL YEAR TO MONTH`, *`fmt`* can only be year. The default *`fmt`* is year.

For `INTERVAL DAY TO SECOND`, *`fmt`* can be day, hour, and minute. The default *`fmt`* is day. Note that *`fmt`* does not support second.

`CEIL(interval)` supports the format models of `ROUND` and `TRUNC`.

> **See Also:**
>
> Refer to CEIL, FLOOR, ROUND, and TRUNC Date Functions for the permitted format models to use in *`fmt`*.

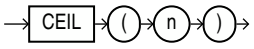
Examples

```
SELECT CEIL(INTERVAL '+123-5' YEAR(3) TO MONTH) AS year_ceil;

YEAR_CEIL
---------
+124-00

SELECT CEIL(INTERVAL '+99-11' YEAR(2) TO MONTH, 'YEAR');

YEAR_CEIL
---------
+100-00

SELECT CEIL(INTERVAL '+999999999-11' YEAR(9) TO MONTH, 'YEAR') AS year_ceil;

ORA-01873: the leading precision of the interval is too small

SELECT CEIL(INTERVAL '+4 12:42:10.222' DAY(2) TO SECOND(3), 'DD') AS day_ceil;

DAY_CEIL
```

```
-------------------
+05 00:00:00.000000
```

CEIL (number)

Syntax



Purpose

`CEIL` returns the smallest integer that is greater than or equal to *n*. The number *n* can always be written as the difference of an integer *k* and a positive fraction *f* such that 0 <= *f* < 1 and *n* = *k* - *f*. The value of `CEIL` is the integer *k*. Thus, the value of `CEIL` is *n* itself if and only if *n* is precisely an integer.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and FLOOR (number)

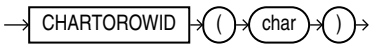
Examples

The following example returns the smallest integer greater than or equal to the order total of a specified order:

```
SELECT order_total, CEIL(order_total)
  FROM orders
  WHERE order_id = 2434;

ORDER_TOTAL CEIL(ORDER_TOTAL)
----------- -----------------
   268651.8            268652
```

CHARTOROWID

Syntax



Purpose

`CHARTOROWID` converts a value from `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type to `ROWID` data type.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> Data Type Comparison Rules for more information.

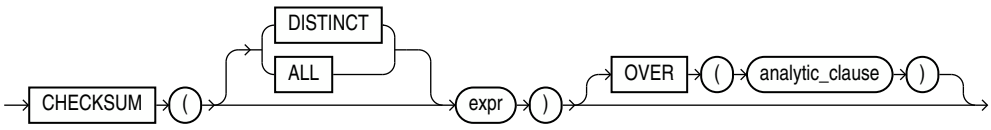
Examples

The following example converts a character rowid representation to a rowid. (The actual rowid is different for each database instance.)

```
SELECT last_name
  FROM employees
  WHERE ROWID = CHARTOROWID('AAAFd1AAFAAAABSAA/');

LAST_NAME
-------------------------
Greene
```

CHECKSUM

Syntax



Purpose

Use CHECKSUM to detect changes in a table. The order of the rows in the table does not affect the result. You can use CHECKSUM with DISTINCT, as part of a GROUP BY query, as a window function, or an analytical function.

Semantics

ALL: Applies the aggregate function to all values. ALL is the default option.

DISTINCT or UNIQUE: Returns the checksum of unique values. UNIQUE is an Oracle-specific keyword and not an ANSI standard.

expr: Can be a column, constant, bind variable, or an expression involving them. All data types except ADT and JSON are supported.

The return data type is an Oracle number (converted from an (8-byte) signed long long) regardless of the data type of expr.

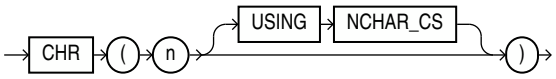
NULL values in expr column are ignored.

It returns NULL if expr is NULL.

The output of the CHECKSUM function is deterministic and independent of the ordering of the input rows.

CHR

Syntax



Purpose

`CHR` returns the character having the binary equivalent to *n* as a `VARCHAR2` value in either the database character set or, if you specify `USING NCHAR_CS`, the national character set.

For single-byte character sets, if `n` > 256, then Oracle Database returns the binary equivalent of `n mod 256`. For multibyte character sets, `n` must resolve to one entire code point. Invalid code points are not validated, and the result of specifying invalid code points is indeterminate.

This function takes as an argument a `NUMBER` value, or any value that can be implicitly converted to `NUMBER`, and returns a character.

> **Note:**
>
> Use of the `CHR` function (either with or without the optional `USING NCHAR_CS` clause) results in code that is not portable between ASCII- and EBCDIC-based machine architectures.

> **See Also:**
>
> - NCHR and Table 2-9 for more information on implicit conversion
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `CHR`

Examples

The following example is run on an ASCII-based machine with the database character set defined as WE8ISO8859P1:

```
SELECT CHR(67)||CHR(65)||CHR(84) "Dog"
  FROM DUAL;

Dog
---
CAT
```

To produce the same results on an EBCDIC-based machine with the WE8EBCDIC1047 character set, the preceding example would have to be modified as follows:

```
SELECT CHR(195)||CHR(193)||CHR(227) "Dog"
  FROM DUAL;
```

```
Dog
---
CAT
```

For multibyte character sets, this sort of concatenation gives different results. For example, given a multibyte character whose hexadecimal value is `a1a2` (`a1` representing the first byte and `a2` the second byte), you must specify for `n` the decimal equivalent of '`a1a2`', or 41378:

```
SELECT CHR(41378)
  FROM DUAL;
```

You cannot specify the decimal equivalent of a1 concatenated with the decimal equivalent of a2, as in the following example:

```
SELECT CHR(161)||CHR(162)
  FROM DUAL;
```

However, you can concatenate whole multibyte code points, as in the following example, which concatenates the multibyte characters whose hexadecimal values are `a1a2` and `a1a3`:

```
SELECT CHR(41378)||CHR(41379)
  FROM DUAL;
```

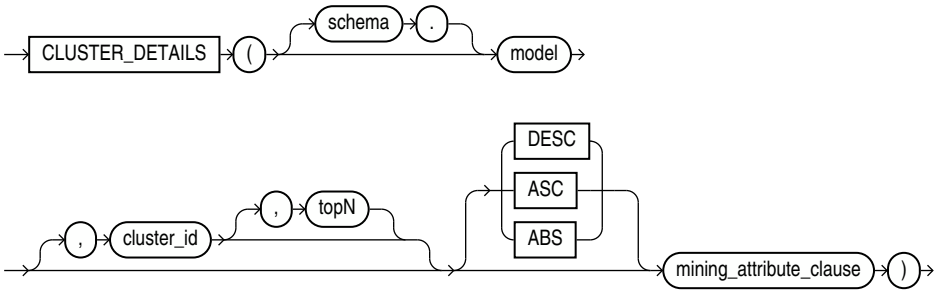
The following example assumes that the national character set is UTF16:

```
SELECT CHR (196 USING NCHAR_CS)
  FROM DUAL;

CH
--
Ä
```

CLUSTER_DETAILS

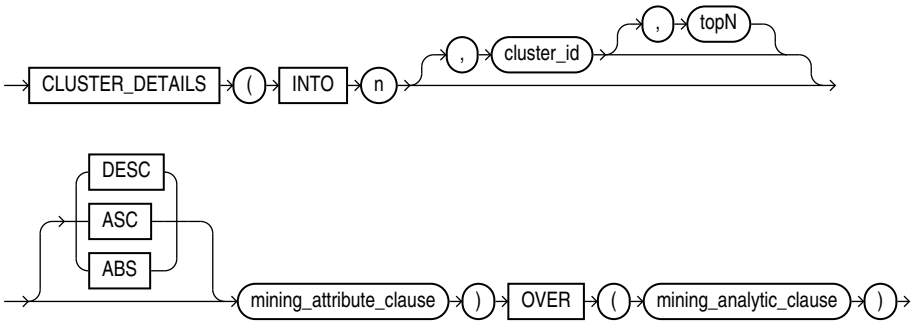
Syntax

***cluster_details*::=**

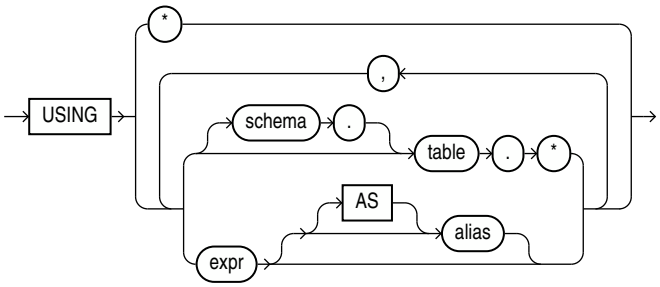


Analytic Syntax

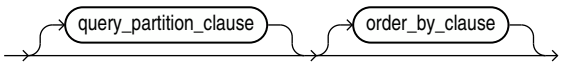
***cluster_details_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> Analytic Functions for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`CLUSTER_DETAILS` returns cluster details for each row in the selection. The return value is an XML string that describes the attributes of the highest probability cluster or the specified *cluster_id*.

topN

If you specify a value for *topN*, the function returns the *N* attributes that most influence the cluster assignment (the score). If you do not specify *topN*, the function returns the 5 most influential attributes.

DESC, ASC, or ABS

The returned attributes are ordered by weight. The weight of an attribute expresses its positive or negative impact on cluster assignment. A positive weight indicates an increased likelihood of assignment. A negative weight indicates a decreased likelihood of assignment.

By default, CLUSTER_DETAILS returns the attributes with the highest positive weights (DESC). If you specify ASC, the attributes with the highest negative weights are returned. If you specify ABS, the attributes with the greatest weights, whether negative or positive, are returned. The results are ordered by absolute value from highest to lowest. Attributes with a zero weight are not included in the output.

Syntax Choice

CLUSTER_DETAILS can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a clustering model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include INTO *n*, where *n* is the number of clusters to compute, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See *analytic_clause*::=.)

The syntax of the CLUSTER_DETAILS function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the PREDICTION function. (See *mining_attribute_clause*::=.)

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring.
> - *Oracle Machine Learning for SQL Concepts* for information about clustering.

> **Note:**
>
> The following examples are excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

This example lists the attributes that have the greatest impact (more that 20% probability) on cluster assignment for customer ID 100955. The query invokes the `CLUSTER_DETAILS` and `CLUSTER_SET` functions, which apply the clustering model `em_sh_clus_sample`.

```
SELECT S.cluster_id, probability prob,
       CLUSTER_DETAILS(em_sh_clus_sample, S.cluster_id, 5 USING T.*) det
FROM
  (SELECT v.*, CLUSTER_SET(em_sh_clus_sample, NULL, 0.2 USING *) pset
    FROM mining_data_apply_v v
   WHERE cust_id = 100955) T,
  TABLE(T.pset) S
ORDER BY 2 DESC;

CLUSTER_ID  PROB DET
---------- ----- --------------------------------------------------------------------------------
        14 .6761 <Details algorithm="Expectation Maximization" cluster="14">
                 <Attribute name="AGE" actualValue="51" weight=".676" rank="1"/>
                 <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".557" rank="2"/>
                 <Attribute name="FLAT_PANEL_MONITOR" actualValue="0" weight=".412" rank="3"/>
                 <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".171" rank="4"/>
                 <Attribute name="BOOKKEEPING_APPLICATION" actualValue="1" weight="-.003"rank="5"/>
                 </Details>

         3 .3227 <Details algorithm="Expectation Maximization" cluster="3">
                 <Attribute name="YRS_RESIDENCE" actualValue="3" weight=".323" rank="1"/>
                 <Attribute name="BULK_PACK_DISKETTES" actualValue="1" weight=".265" rank="2"/>
                 <Attribute name="EDUCATION" actualValue="HS-grad" weight=".172" rank="3"/>
                 <Attribute name="AFFINITY_CARD" actualValue="0" weight=".125" rank="4"/>
                 <Attribute name="OCCUPATION" actualValue="Crafts" weight=".055" rank="5"/>
                 </Details>
```

Analytic Example

This example divides the customer database into four segments based on common characteristics. The clustering functions compute the clusters and return the score without a predefined clustering model.

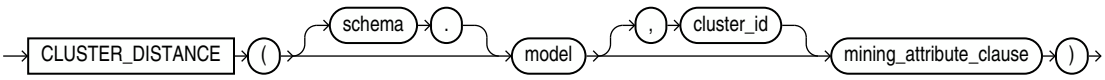
```
SELECT * FROM (
     SELECT cust_id,
          CLUSTER_ID(INTO 4 USING *) OVER () cls,
          CLUSTER_DETAILS(INTO 4 USING *) OVER () cls_details
     FROM mining_data_apply_v)
WHERE cust_id <= 100003
ORDER BY 1;

CUST_ID CLS CLS_DETAILS
------- --- --------------------------------------------------------------------------------
 100001   5 <Details algorithm="K-Means Clustering" cluster="5">
            <Attribute name="FLAT_PANEL_MONITOR" actualValue="0" weight=".349" rank="1"/>
            <Attribute name="BULK_PACK_DISKETTES" actualValue="0" weight=".33" rank="2"/>
            <Attribute name="CUST_INCOME_LEVEL" actualValue="G: 130\,000 - 149\,999" weight=".291"
```

```
           rank="3"/>
          <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".268" rank="4"/>
          <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".179" rank="5"/>
          </Details>

100002    6 <Details algorithm="K-Means Clustering" cluster="6">
          <Attribute name="CUST_GENDER" actualValue="F" weight=".945" rank="1"/>
          <Attribute name="CUST_MARITAL_STATUS" actualValue="NeverM" weight=".856" rank="2"/>
          <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".468" rank="3"/>
          <Attribute name="AFFINITY_CARD" actualValue="0" weight=".012" rank="4"/>
          <Attribute name="CUST_INCOME_LEVEL" actualValue="L: 300\,000 and above" weight=".009"
           rank="5"/>
          </Details>

100003    7 <Details algorithm="K-Means Clustering" cluster="7">
          <Attribute name="CUST_MARITAL_STATUS" actualValue="NeverM" weight=".862" rank="1"/>
          <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".423" rank="2"/>
          <Attribute name="HOME_THEATER_PACKAGE" actualValue="0" weight=".113" rank="3"/>
          <Attribute name="AFFINITY_CARD" actualValue="0" weight=".007" rank="4"/>
          <Attribute name="CUST_ID" actualValue="100003" weight=".006" rank="5"/>
          </Details>
```

CLUSTER_DISTANCE

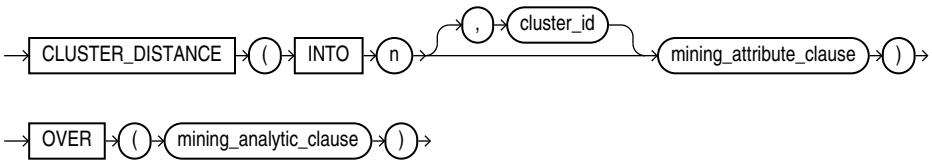
Syntax

***cluster_distance*::=**

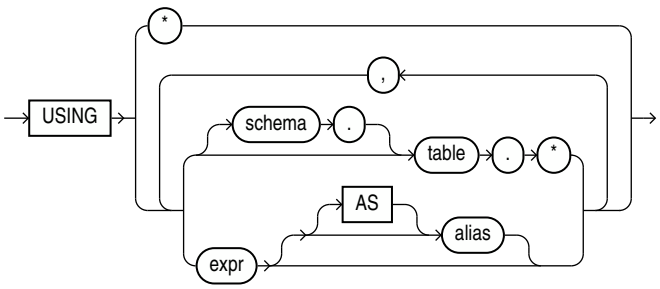


Analytic Syntax

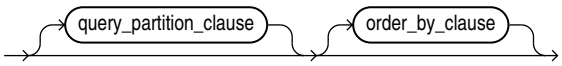
***cluster_distance_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> Analytic Functions for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`CLUSTER_DISTANCE` returns a cluster distance for each row in the selection. The cluster distance is the distance between the row and the centroid of the highest probability cluster or the specified *cluster_id*. The distance is returned as `BINARY_DOUBLE`.

Syntax Choice

`CLUSTER_DISTANCE` can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a clustering model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include `INTO` *n*, where *n* is the number of clusters to compute, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See *analytic_clause*::=.)

The syntax of the `CLUSTER_DISTANCE` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, this data is also used for building the transient models. The *mining_attribute_clause* behaves as described for the `PREDICTION` function. (See *mining_attribute_clause*::=.)

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring.
> - *Oracle Machine Learning for SQL Concepts* for information about clustering.

> **Note:**
>
> The following example is excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

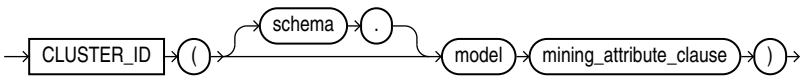
This example finds the 10 rows that are most anomalous as measured by their distance from their nearest cluster centroid.

```
SELECT cust_id
  FROM (
    SELECT cust_id,
           rank() over
             (order by CLUSTER_DISTANCE(km_sh_clus_sample USING *) desc) rnk
      FROM mining_data_apply_v)
  WHERE rnk <= 11
  ORDER BY rnk;

   CUST_ID
----------
    100579
    100050
    100329
    100962
    101251
    100179
    100382
    100713
    100629
    100787
    101478
```

CLUSTER_ID

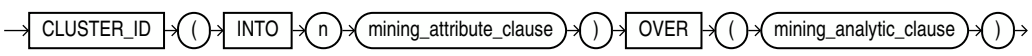
Syntax

***cluster_id*::=**

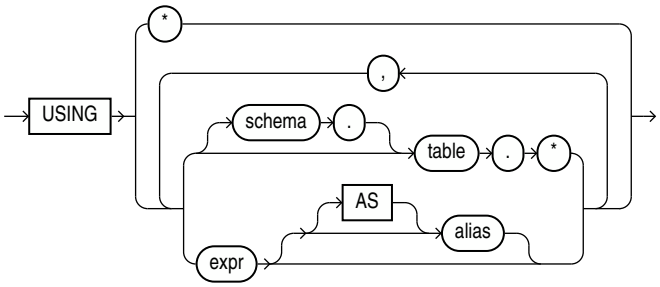


Analytic Syntax

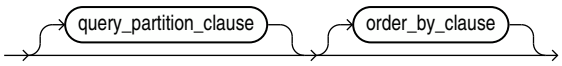
***cluster_id_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> Analytic Functions for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`CLUSTER_ID` returns the identifier of the highest probability cluster for each row in the selection. The cluster identifier is returned as an Oracle `NUMBER`.

Syntax Choice

`CLUSTER_ID` can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a clustering model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include `INTO` *n*, where *n* is the number of clusters to compute, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See *analytic_clause*::=.)

The syntax of the `CLUSTER_ID` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the `PREDICTION` function. (See *mining_attribute_clause*::=.)

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring.
> - *Oracle Machine Learning for SQL Concepts* for information about clustering.

> **Note:**
>
> The following examples are excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

The following example lists the clusters into which the customers in `mining_data_apply_v` have been grouped.

```
SELECT CLUSTER_ID(km_sh_clus_sample USING *) AS clus, COUNT(*) AS cnt
  FROM mining_data_apply_v
  GROUP BY CLUSTER_ID(km_sh_clus_sample USING *)
  ORDER BY cnt DESC;

      CLUS        CNT
---------- ----------
         2        580
        10        216
         6        186
         8        115
        19        110
        12        101
        18         81
        16         39
        17         38
        14         34
```

Analytic Example

This example divides the customer database into four segments based on common characteristics. The clustering functions compute the clusters and return the score without a predefined clustering model.

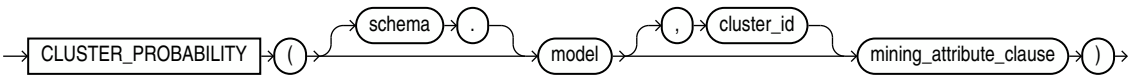
```
SELECT * FROM (
     SELECT cust_id,
          CLUSTER_ID(INTO 4 USING *) OVER () cls,
          CLUSTER_DETAILS(INTO 4 USING *) OVER () cls_details
     FROM mining_data_apply_v)
WHERE cust_id <= 100003
ORDER BY 1;

CUST_ID CLS CLS_DETAILS
------- --- --------------------------------------------------------------------------------
 100001   5 <Details algorithm="K-Means Clustering" cluster="5">
            <Attribute name="FLAT_PANEL_MONITOR" actualValue="0" weight=".349" rank="1"/>
            <Attribute name="BULK_PACK_DISKETTES" actualValue="0" weight=".33" rank="2"/>
            <Attribute name="CUST_INCOME_LEVEL" actualValue="G: 130\,000 - 149\,999"
               weight=".291" rank="3"/>
```

```
          <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".268" rank="4"/>
          <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".179" rank="5"/>
          </Details>

100002    6 <Details algorithm="K-Means Clustering" cluster="6">
          <Attribute name="CUST_GENDER" actualValue="F" weight=".945" rank="1"/>
          <Attribute name="CUST_MARITAL_STATUS" actualValue="NeverM" weight=".856" rank="2"/>
          <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".468" rank="3"/>
          <Attribute name="AFFINITY_CARD" actualValue="0" weight=".012" rank="4"/>
          <Attribute name="CUST_INCOME_LEVEL" actualValue="L: 300\,000 and above"
             weight=".009" rank="5"/>
          </Details>

100003    7 <Details algorithm="K-Means Clustering" cluster="7">
          <Attribute name="CUST_MARITAL_STATUS" actualValue="NeverM" weight=".862" rank="1"/>
          <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".423" rank="2"/>
          <Attribute name="HOME_THEATER_PACKAGE" actualValue="0" weight=".113" rank="3"/>
          <Attribute name="AFFINITY_CARD" actualValue="0" weight=".007" rank="4"/>
          <Attribute name="CUST_ID" actualValue="100003" weight=".006" rank="5"/>
          </Details>
```

CLUSTER_PROBABILITY

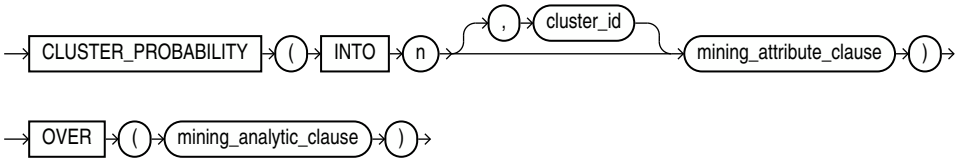
Syntax

***cluster_probability*::=**

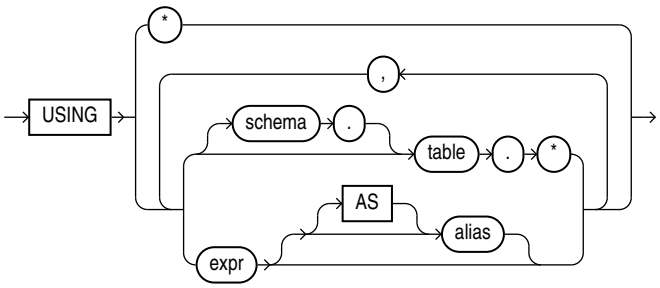


Analytic Syntax

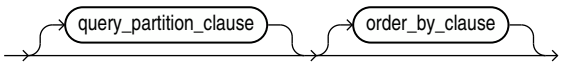
***cluster_prob_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> Analytic Functions for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

CLUSTER_PROBABILITY returns a probability for each row in the selection. The probability refers to the highest probability cluster or to the specified *cluster_id*. The cluster probability is returned as BINARY_DOUBLE.

Syntax Choice

CLUSTER_PROBABILITY can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a clustering model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include INTO *n*, where *n* is the number of clusters to compute, and mining_analytic_clause, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See *analytic_clause*::=.)

The syntax of the CLUSTER_PROBABILITY function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the PREDICTION function. (See *mining_attribute_clause*::=.)

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring.
> - *Oracle Machine Learning for SQL Concepts* for information about clustering.

> **Note:**
>
> The following example is excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

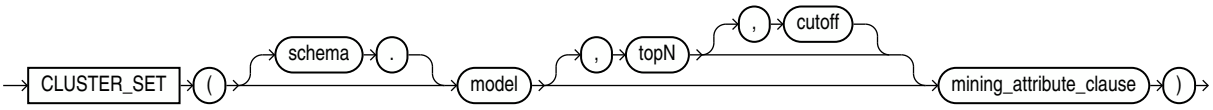
The following example lists the ten most representative customers, based on likelihood, of cluster 2.

```
SELECT cust_id
  FROM (SELECT cust_id, rank() OVER (ORDER BY prob DESC, cust_id) rnk_clus2
    FROM (SELECT cust_id, CLUSTER_PROBABILITY(km_sh_clus_sample, 2 USING *) prob
          FROM mining_data_apply_v))
WHERE rnk_clus2 <= 10
ORDER BY rnk_clus2;

   CUST_ID
----------
    100256
    100988
    100889
    101086
    101215
    100390
    100985
    101026
    100601
    100672
```

CLUSTER_SET

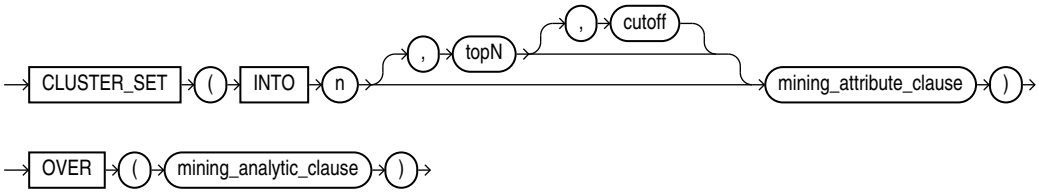
Syntax

***cluster_set*::=**

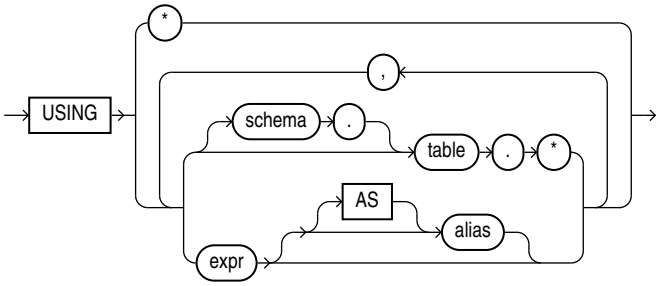


Analytic Syntax

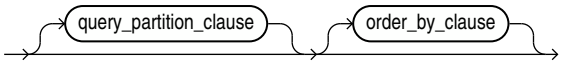
***cluster_set_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> Analytic Functions for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`CLUSTER_SET` returns a set of cluster ID and probability pairs for each row in the selection. The return value is a varray of objects with field names `CLUSTER_ID` and `PROBABILITY`. The cluster identifier is an Oracle `NUMBER`; the probability is `BINARY_DOUBLE`.

topN and cutoff

You can specify *topN* and *cutoff* to limit the number of clusters returned by the function. By default, both *topN* and *cutoff* are null and all clusters are returned.

- *topN* is the *N* most probable clusters. If multiple clusters share the *N*th probability, then the function chooses one of them.
- *cutoff* is a probability threshold. Only clusters with probability greater than or equal to *cutoff* are returned. To filter by *cutoff* only, specify `NULL` for *topN*.

To return up to the *N* most probable clusters that are greater than or equal to *cutoff*, specify both *topN* and *cutoff*.

Syntax Choice

`CLUSTER_SET` can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a clustering model.

- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include `INTO n`, where *n* is the number of clusters to compute, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See *analytic_clause*::=.)

The syntax of the `CLUSTER_SET` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the `PREDICTION` function. (See *mining_attribute_clause*::=.)

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring.
> - *Oracle Machine Learning for SQL Concepts* for information about clustering.

> **Note:**
>
> The following example is excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

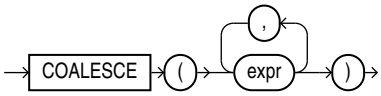
This example lists the attributes that have the greatest impact (more that 20% probability) on cluster assignment for customer ID 100955. The query invokes the `CLUSTER_DETAILS` and `CLUSTER_SET` functions, which apply the clustering model `em_sh_clus_sample`.

```
SELECT S.cluster_id, probability prob,
       CLUSTER_DETAILS(em_sh_clus_sample, S.cluster_id, 5 USING T.*) det
FROM
  (SELECT v.*, CLUSTER_SET(em_sh_clus_sample, NULL, 0.2 USING *) pset
    FROM mining_data_apply_v v
   WHERE cust_id = 100955) T,
  TABLE(T.pset) S
ORDER BY 2 DESC;

CLUSTER_ID  PROB DET
---------- ----- --------------------------------------------------------------------------------
        14 .6761 <Details algorithm="Expectation Maximization" cluster="14">
                 <Attribute name="AGE" actualValue="51" weight=".676" rank="1"/>
                 <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".557" rank="2"/>
                 <Attribute name="FLAT_PANEL_MONITOR" actualValue="0" weight=".412" rank="3"/>
                 <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".171" rank="4"/>
                 <Attribute name="BOOKKEEPING_APPLICATION" actualValue="1" weight="-.003"rank="5"/>
                 </Details>

         3 .3227 <Details algorithm="Expectation Maximization" cluster="3">
```

```
<Attribute name="YRS_RESIDENCE" actualValue="3" weight=".323" rank="1"/>
<Attribute name="BULK_PACK_DISKETTES" actualValue="1" weight=".265" rank="2"/>
<Attribute name="EDUCATION" actualValue="HS-grad" weight=".172" rank="3"/>
<Attribute name="AFFINITY_CARD" actualValue="0" weight=".125" rank="4"/>
<Attribute name="OCCUPATION" actualValue="Crafts" weight=".055" rank="5"/>
</Details>
```

COALESCE

Syntax



Purpose

`COALESCE` returns the first non-null *`expr`* in the expression list. You must specify at least two expressions. If all occurrences of *`expr`* evaluate to null, then the function returns null.

Oracle Database uses **short-circuit evaluation**. The database evaluates each *`expr`* value and determines whether it is `NULL`, rather than evaluating all of the *`expr`* values before determining whether any of them is `NULL`.

If all occurrences of *`expr`* are numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type, then Oracle Database determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> - Table 2-9 for more information on implicit conversion and Numeric Precedence for information on numeric precedence
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `COALESCE` when it is a character value

This function is a generalization of the `NVL` function.

You can also use `COALESCE` as a variety of the `CASE` expression. For example,

```
COALESCE(expr1, expr2)
```

is equivalent to:

```
CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END
```

Similarly,

```
COALESCE(expr1, expr2, ..., exprn)
```

where `n` >= 3, is equivalent to:

```
CASE WHEN expr1 IS NOT NULL THEN expr1
   ELSE COALESCE (expr2, ..., exprn) END
```

> **See Also:**
>
> NVL and CASE Expressions

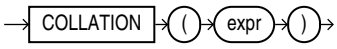
Examples

The following example uses the sample `oe.product_information` table to organize a clearance sale of products. It gives a 10% discount to all products with a list price. If there is no list price, then the sale price is the minimum price. If there is no minimum price, then the sale price is "5":

```
SELECT product_id, list_price, min_price,
       COALESCE(0.9*list_price, min_price, 5) "Sale"
  FROM product_information
  WHERE supplier_id = 102050
  ORDER BY product_id;

PRODUCT_ID LIST_PRICE  MIN_PRICE       Sale
---------- ---------- ---------- ----------
      1769         48                  43.2
      1770                    73         73
      2378        305        247      274.5
      2382        850        731        765
      3355                                5
```

COLLATION

Syntax



Purpose

`COLLATION` returns the name of the derived collation for *`expr`*. This function returns named collations and pseudo-collations. If the derived collation is a Unicode Collation Algorithm (UCA) collation, then the function returns the long form of its name. This function is evaluated during compilation of the SQL statement that contains it. If the derived collation is undefined due to a collation conflict while evaluating *`expr`*, then the function returns null.

`expr` must evaluate to a character string of type `CHAR`, `VARCHAR2`, `LONG`, `NCHAR`, or `NVARCHAR2`.

This function returns a `VARCHAR2` value.

> **Note:**
>
> The `COLLATION` function returns only the data-bound collation, and not the dynamic collation set by the `NLS_SORT` parameter. Thus, for a column declared as `COLLATE USING_NLS_SORT`, the function returns the character value `'USING_NLS_SORT'`, not the actual value of the session parameter `NLS_SORT`. You can use the built-in function `SYS_CONTEXT('USERENV','NLS_SORT')` to get the actual value of the session parameter `NLS_SORT`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `COLLATION`

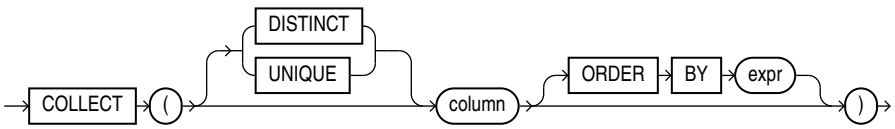
Examples

The following example returns the derived collation of columns `name` and `id` in table `id_table`:

```
CREATE TABLE id_table
  (name VARCHAR2(64) COLLATE BINARY_AI,
   id VARCHAR2(8) COLLATE BINARY_CI);

INSERT INTO id_table VALUES('Christopher', 'ABCD1234');

SELECT COLLATION(name), COLLATION(id)
  FROM id_table;

COLLATION COLLATION
--------- ---------
BINARY_AI BINARY_CI
```

COLLECT

Syntax



Purpose

`COLLECT` is an aggregate function that takes as its argument a column of any type and creates a nested table of the input type out of the rows selected. To get accurate results from this function you must use it within a `CAST` function.

If *`column`* is itself a collection, then the output of `COLLECT` is a nested table of collections. If *`column`* is of a user-defined type, then *`column`* must have a `MAP` or `ORDER` method defined on it in order for you to use the optional `DISTINCT`, `UNIQUE`, and `ORDER BY` clauses.

> **See Also:**
>
> - CAST and Aggregate Functions
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `COLLECT` uses to compare character values for the `DISTINCT` and `ORDER BY` clauses

Examples

The following example creates a nested table from the varray column of phone numbers in the sample table `oe.customers`. The nested table includes only the phone numbers of customers with an income level of `L: 300,000 and above`.

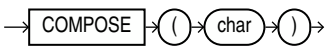
```
CREATE TYPE phone_book_t AS TABLE OF phone_list_typ;
/

SELECT CAST(COLLECT(phone_numbers) AS phone_book_t) "Income Level L Phone Book"
  FROM customers
  WHERE income_level = 'L: 300,000 and above';

Income Level L Phone Book
--------------------------------------------------------------------------------
PHONE_BOOK_T(PHONE_LIST_TYP('+1 414 123 4307'), PHONE_LIST_TYP('+1 608 123 4344'
), PHONE_LIST_TYP('+1 814 123 4696'), PHONE_LIST_TYP('+1 215 123 4721'), PHONE_L
IST_TYP('+1 814 123 4755'), PHONE_LIST_TYP('+91 11 012 4817', '+91 11 083 4817')
, PHONE_LIST_TYP('+91 172 012 4837'), PHONE_LIST_TYP('+41 31 012 3569', '+41 31
083 3569'))
```

The following example creates a nested table from the column of warehouse names in the sample table `oe.warehouses`. It uses `ORDER BY` to order the warehouse names.

```
CREATE TYPE warehouse_name_t AS TABLE OF VARCHAR2(35);
/

SELECT CAST(COLLECT(warehouse_name ORDER BY warehouse_name)
       AS warehouse_name_t) "Warehouses"
   FROM warehouses;

Warehouses
--------------------------------------------------------------------------------
WAREHOUSE_NAME_TYP('Beijing', 'Bombay', 'Mexico City', 'New Jersey', 'San Franci
sco', 'Seattle, Washington', 'Southlake, Texas', 'Sydney', 'Toronto')
```

COMPOSE

Syntax



Purpose

`COMPOSE` takes as its argument a character value *`char`* and returns the result of applying the Unicode canonical composition, as described in the Unicode Standard definition D117, to it. If

the character set of the argument is not one of the Unicode character sets, COMPOSE returns its argument unmodified.

`COMPOSE` does not directly return strings in any of the Unicode normalization forms. To get a string in the NFC form, first call `DECOMPOSE` with the `CANONICAL` setting and then `COMPOSE` . To get a string in the NFKC form, first call `DECOMPOSE` with the `COMPATIBILITY` setting and then `COMPOSE` .

`char` can be of any of the data types: `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. Other data types are allowed if they can be implicitly converted to `VARCHAR2` or `NVARCHAR2`. The return value of `COMPOSE` is in the same character set as its argument.

`CLOB` and `NCLOB` values are supported through implicit conversion. If *`char`* is a character LOB value, then it is converted to a `VARCHAR2` value before the `COMPOSE` operation. The operation will fail if the size of the LOB value exceeds the supported length of the `VARCHAR2` in the particular execution environment.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for information on Unicode character sets and character semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `COMPOSE`
> - DECOMPOSE

Examples

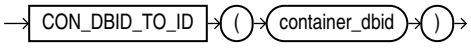
The following example returns the o-umlaut code point:

```
SELECT COMPOSE( 'o' || UNISTR('\0308') )
  FROM DUAL;

CO
--
ö
```

> **See Also:**
>
> UNISTR

CON_DBID_TO_ID

Syntax



Purpose

`CON_DBID_TO_ID` takes as its argument a container DBID and returns the container ID. For *`container_dbid`*, specify a `NUMBER` value or any value that can be implicitly converted to `NUMBER`. The function returns a `NUMBER` value.

This function is useful in a multitenant container database (CDB). If you use this function in a non-CDB, then it returns 0.

Example

The following query displays the ID and DBID for all containers in a CDB. The sample output shown is for the purpose of this example.

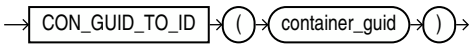
```
SELECT CON_ID, DBID
  FROM V$CONTAINERS;

    CON_ID       DBID
---------- ----------
         1 1930093401
         2 4054529501
         4 2256797992
```

The following statement returns the ID for the container with DBID 2256797992:

```
SELECT CON_DBID_TO_ID(2256797992) "Container ID"
  FROM DUAL;

Container ID
------------
           4
```

CON_GUID_TO_ID

Syntax



Purpose

`CON_GUID_TO_ID` takes as its argument a container GUID (globally unique identifier) and returns the container ID. For *`container_guid`*, specify a raw value. The function returns a `NUMBER` value.

This function is useful in a multitenant container database (CDB). If you use this function in a non-CDB, then it returns 0.

Example

The following query displays the ID and GUID for all containers in a CDB. The GUID is stored as a 16-byte `RAW` value in the `V$CONTAINERS` view. The query returns the 32-character hexadecimal representation of the GUID. The sample output shown is for the purpose of this example.

```
SELECT CON_ID, GUID
  FROM V$CONTAINERS;
```

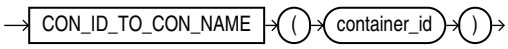
```
    CON_ID GUID
---------- --------------------------------
         1 DB0A9F33DF99567FE04305B4F00A667D
         2 D990C280C309591EE04305B4F00A593E
         4 D990F4BD938865C1E04305B4F00ACA18
```

The following statement returns the ID for the container whose GUID is represented by the hexadecimal value `D990F4BD938865C1E04305B4F00ACA18`. The `HEXTORAW` function converts the GUID's hexadecimal representation to a raw value.

```
SELECT CON_GUID_TO_ID(HEXTORAW('D990F4BD938865C1E04305B4F00ACA18')) "Container ID"
  FROM DUAL;

Container ID
------------
           4
```

CON_ID_TO_CON_NAME

Syntax



Purpose

`CON_ID_TO_CON_NAME` takes as an argument a container `CON_ID` and returns the container `NAME`.

For `CON_ID` you must specify a number or an expression that resolves to a number. The function returns a `NUMBER` value.

This function is useful in a multitentant container database (CDB). If you use this function in a non-CDB, then it returns 0.

Example

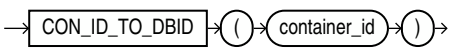
```
SELECT CON_ID, NAME FROM V$CONTAINERS;
    CON_ID         NAME
   ---------       -------------
       1           CDB$ROOT
       2           PDB$SEED
       3           CDB1_PDB1
       4           SALESPDB
```

The following statement returns the container `NAME` given the container `CON_ID` 4:

```
SELECT CON_ID_TO_CON_NAME(4) "CON_NAME" FROM DUAL;
     CON_NAME
     --------
     SALESDB
```

CON_ID_TO_DBID

Syntax



Purpose

CON_ID_TO_DBID takes as an argument a container CON_ID and returns the container DBID. For CON_ID you must specify a number or an expression that resolves to a number. The function returns a NUMBER value.

This function is useful in a multitentant container database (CDB). If you use this function in a non-CDB, then it returns 0.

Example

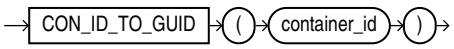
```
SELECT CON_ID, NAME, DBID FROM V$CONTAINERS;

CON_ID      NAME           DBID
-------     ------------   ---------------
   1        CDB$ROOT       2048400776
   2        PDB$SEED       2929762556
   3        CDB1_PDB1      3483444080
   4        SALESPDB       2221053340
```

The following statement returns the container DBID given the container CON_ID 4:

```
SELECT CON_ID_TO_DBID(4) FROM DUAL;
    DBID
    -------------
    2221053340
```

CON_ID_TO_GUID

Syntax



Purpose

CON_ID_TO_GUID takes as an argument a container CON_ID and returns the container's GLOBAL UNIQUE ID (GUID). For CON_ID you must specify a number or an expression that resolves to a number. The function returns a NUMBER value.

This function is useful in a multitentant container database (CDB).

Example

The following query displays the CON_ID, NAME and GUID for all containers in a CDB:

```
SELECT CON_ID, NAME, GUID FROM V$CONTAINERS;

CON_ID      NAME           GUID
-------     ------------   ---------------
   1        CDB$ROOT       A8C0E03CB11A132FE0532684E80A96B3
   2        PDB$SEED       A8DA5D32F8F5590DE053C4E15A0A6EED
   3        CDB1_PDB1      A8DA63CEAD385A5BE053C4E15A0A774A
   4        SALESPDB       A8DA9AB18CE85BD0E053C4E15A0AE2C3
```

The following statement returns the container GUID given the container CON_ID 4:

```
SELECT CON_ID_TO_GUID(4) "CON_GUID" FROM DUAL;
    CON_GUID
    -----------------
    A8DA9AB18CE85BD0E053C4E15A0AE2C3
```

CON_ID_TO_UID

Syntax

Purpose

CON_ID_TO_UID takes as an argument a container CON_ID and returns the container's UNIQUE ID (UID). For CON_ID you must specify a number or an expression that resolves to a number. The function returns a NUMBER value.

This function is useful in a multitentant container database (CDB).

Example

The following query displays the CON_ID, NAME and CON_UID for all containers in a CDB:

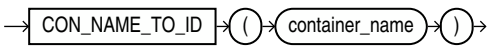
```
SELECT CON_ID, NAME, CON_UID FROM V$CONTAINERS;

CON_ID      NAME           CON_UID
-------     ------------   ---------------
   1        CDB$ROOT       1
   2        PDB$SEED       2929762556
   3        CDB1_PDB1      3483444080
   4        SALESPDB       2221053340
```

The following statement returns the container CON_UID given the container CON_ID 4:

```
SELECT CON_ID_TO_UID(4) "PDB_UID" FROM DUAL;
    PDB_UID
    -------------
    2221053340
```

CON_NAME_TO_ID

Syntax



Purpose

CON_NAME_TO_ID takes as its argument a container name and returns the container ID. For *container_name*, specify a string, or an expression that resolves to a string, in any data type. The function returns a NUMBER value.

This function is useful in a multitenant container database (CDB). If you use this function in a non-CDB, then it returns 0.

Example

The following query displays the ID and name for all containers in a CDB. The sample output shown is for the purpose of this example.

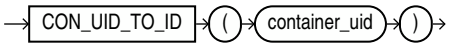
```
SELECT CON_ID, NAME
  FROM V$CONTAINERS;

    CON_ID NAME
---------- ----------
         1 CDB$ROOT
         2 PDB$SEED
         4 SALESPDB
```

The following statement returns the ID for the container named `SALESPDB`:

```
SELECT CON_NAME_TO_ID('SALESPDB') "Container ID"
  FROM DUAL;

Container ID
------------
           4
```

CON_UID_TO_ID

Syntax



Purpose

`CON_UID_TO_ID` takes as its argument a container UID (unique identifier) and returns the container ID. For *`container_uid`*, specify a `NUMBER` value or any value that can be implicitly converted to `NUMBER`. The function returns a `NUMBER` value.

This function is useful in a multitenant container database (CDB). If you use this function in a non-CDB, then it returns 0.

Example

The following query displays the ID and UID for all containers in a CDB. The sample output shown is for the purpose of this example.

```
SELECT CON_ID, CON_UID
  FROM V$CONTAINERS;

    CON_ID    CON_UID
---------- ----------
         1          1
         2 4054529501
         4 2256797992
```

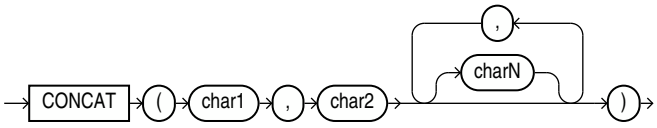
The following query returns the ID for the container with UID 2256797992:

```
SELECT CON_UID_TO_ID(2256797992) "Container ID"
  FROM DUAL;
```

```
Container ID
------------
           4
```

CONCAT

Syntax



Purpose

`CONCAT` takes as input two or more arguments and returns the concatenation of all arguments.

The arguments can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. Arguments of other data types are implicitly converted to `VARCHAR2` before concatenation.

The string returned is in the same character set as *`char1`*. Its data type depends on the data types of the arguments.

In concatenations of two or more different data types, Oracle Database returns the data type that results in a lossless conversion. Therefore, if one of the arguments is a LOB, then the returned value is a LOB. If one of the arguments is a national data type, then the returned value is a national data type.

Rules for the Data Types of Return Values

Among all arguments:

- if there is a NCLOB, or if there is a CLOB and a NVARCHAR2 /NCHAR, then the return type is NCLOB.
- otherwise, if there is CLOB, then the return type is CLOB
- otherwise, if there is NVARCHAR2, or if there is a VARCHAR2 and a NCHAR, then the return type is NVARCHAR2
- otherwise, if there is VARCHAR2, then the return type is VARCHAR2
- otherwise, if there is NCHAR, then the return type is NCHAR
- otherwise, the return type is CHAR

Examples of Data Types Returned

`CONCAT(CLOB, NCLOB)` returns `NCLOB`

`CONCAT(CLOB, NCHAR)` returns `NCLOB`

`CONCAT(CLOB, CHAR)` returns `CLOB`

`CONCAT(VARCHAR2, NCHAR)` returns `NVARCHAR2`

`CONCAT(CHAR, VARCHAR2)` returns `VARCHAR2`

`CONCAT(CHAR, VARCHAR2, CLOB)` returns `CLOB`

CONCAT(CHAR, NVARCHAR2, CLOB) returns NCLOB

This function is equivalent to the concatenation operator (||).

> **See Also:**
>
> - Concatenation Operator for information on the CONCAT operator
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of CONCAT

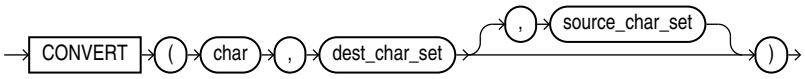
Examples

This example concatenates three character strings:

```
SELECT CONCAT( last_name, '''s job category is ', job_id) "Job"
  FROM employees
  WHERE employee_id = 152;

Job
----------------------------------------------------
Hall's job category is SA_REP
```

CONVERT

Syntax



Purpose

CONVERT converts a character string from one character set to another.

- The *char* argument is the value to be converted. It can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB.
- The *dest_char_set* argument is the name of the character set to which *char* is converted.
- The *source_char_set* argument is the name of the character set in which *char* is stored in the database. The default value is the database character set.

The return value for CHAR and VARCHAR2 is VARCHAR2. For NCHAR and NVARCHAR2, it is NVARCHAR2. For CLOB, it is CLOB, and for NCLOB, it is NCLOB.

Both the destination and source character set arguments can be either literals or columns containing the name of the character set.

For complete correspondence in character conversion, it is essential that the destination character set contains a representation of all the characters defined in the source character set. Where a character does not exist in the destination character set, a replacement character appears. Replacement characters can be defined as part of a character set definition.

> **Note:**
>
> Oracle discourages the use of the `CONVERT` function in the current Oracle Database release. The return value of `CONVERT` has a character data type, so it should be either in the database character set or in the national character set, depending on the data type. Any *`dest_char_set`* that is not one of these two character sets is unsupported. The *`char`* argument and the *`source_char_set`* have the same requirements. Therefore, the only practical use of the function is to correct data that has been stored in a wrong character set.
>
> Values that are in neither the database nor the national character set should be processed and stored as `RAW` or `BLOB`. Procedures in the PL/SQL packages `UTL_RAW` and `UTL_I18N`—for example, `UTL_RAW.CONVERT`—allow limited processing of such values. Procedures accepting a `RAW` argument in the packages `UTL_FILE`, `UTL_TCP`, `UTL_HTTP`, and `UTL_SMTP` can be used to output the processed data.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `CONVERT`

Examples

The following example illustrates character set conversion by converting a Latin-1 string to ASCII. The result is the same as importing the same string from a WE8ISO8859P1 database to a US7ASCII database.

```
SELECT CONVERT('Ä Ê Í Õ Ø A B C D E ', 'US7ASCII', 'WE8ISO8859P1')
   FROM DUAL;

CONVERT('ÄÊÍÕØABCDE'
--------------------
A E I ? ? A B C D E ?
```

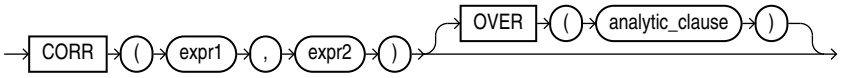
You can query the `V$NLS_VALID_VALUES` view to get a listing of valid character sets, as follows:

```
SELECT * FROM V$NLS_VALID_VALUES WHERE parameter = 'CHARACTERSET';
```

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for the list of character sets that Oracle Database supports and *Oracle Database Reference* for information on the `V$NLS_VALID_VALUES` view

CORR

Syntax



> **See Also:**
>
> Analytic Functions for information on syntax, semantics, and restrictions

Purpose

`CORR` returns the coefficient of correlation of a set of number pairs. You can use it as an aggregate or analytic function.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and Numeric Precedence for information on numeric precedence

Oracle Database applies the function to the set of (*expr1*, *expr2*) after eliminating the pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
COVAR_POP(expr1, expr2) / (STDDEV_POP(expr1) * STDDEV_POP(expr2))
```

The function returns a value of type `NUMBER`. If the function is applied to an empty set, then it returns null.

> **Note:**
>
> The `CORR` function calculates the Pearson's correlation coefficient, which requires numeric expressions as arguments. Oracle also provides the `CORR_S` (Spearman's rho coefficient) and `CORR_K` (Kendall's tau-b coefficient) functions to support nonparametric or rank correlation.

> **See Also:**
>
> Aggregate Functions , About SQL Expressions for information on valid forms of *expr*, and CORR_* for information on the CORR_S and CORR_K functions

Aggregate Example

The following example calculates the coefficient of correlation between the list prices and minimum prices of products by weight class in the sample table oe.product_information:

```
SELECT weight_class, CORR(list_price, min_price) "Correlation"
  FROM product_information
  GROUP BY weight_class
  ORDER BY weight_class, "Correlation";

WEIGHT_CLASS Correlation
------------ -----------
           1  .999149795
           2  .999022941
           3  .998484472
           4  .999359909
           5  .999536087
```

Analytic Example

The following example shows the correlation between duration at the company and salary by the employee's position. The result set shows the same correlation for each employee in a given job:

```
SELECT employee_id, job_id,
       TO_CHAR((SYSDATE - hire_date) YEAR TO MONTH ) "Yrs-Mns",     salary,
       CORR(SYSDATE-hire_date, salary)
       OVER(PARTITION BY job_id) AS "Correlation"
  FROM employees
  WHERE department_id in (50, 80)
  ORDER BY job_id, employee_id;

EMPLOYEE_ID JOB_ID     Yrs-Mns     SALARY Correlation
----------- ---------- ------- ---------- -----------
        145 SA_MAN     +04-09       14000  .912385598
        146 SA_MAN     +04-06       13500  .912385598
        147 SA_MAN     +04-04       12000  .912385598
        148 SA_MAN     +01-08       11000  .912385598
        149 SA_MAN     +01-05       10500  .912385598
        150 SA_REP     +04-05       10000   .80436755
        151 SA_REP     +04-03        9500   .80436755
        152 SA_REP     +03-10        9000   .80436755
        153 SA_REP     +03-03        8000   .80436755
        154 SA_REP     +02-07        7500   .80436755
        155 SA_REP     +01-07        7000   .80436755
. . .
```

CORR_*

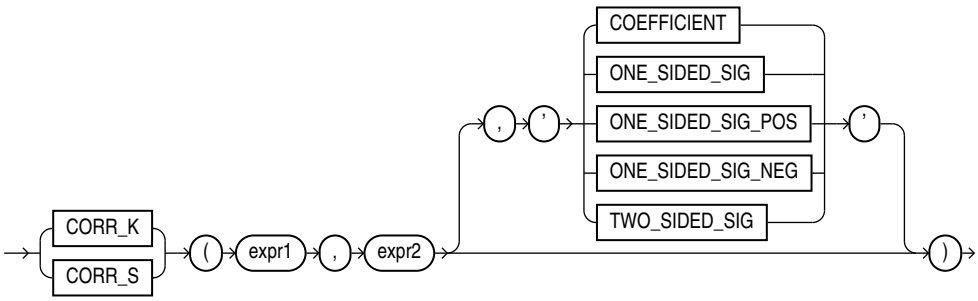
The CORR_* functions are:

- CORR_S

- CORR_K

Syntax

***correlation*::=**



Purpose

The CORR function (see CORR) calculates the Pearson's correlation coefficient and requires numeric expressions as input. The CORR_* functions support nonparametric or rank correlation. They let you find correlations between expressions that are ordinal scaled (where a ranking of the values is possible). Correlation coefficients take on a value ranging from -1 to 1, where 1 indicates a perfect relationship, -1 a perfect inverse relationship (when one variable increases as the other decreases), and a value close to 0 means no relationship.

These functions take two mandatory arguments, *expr1* and *expr2*, and an optional third argument. The return value of the functions is a NUMBER.

The mandatory arguments are the two variables being analyzed. They can be of any data type that is comparable other than LONG, CLOB, BLOB, BFILE, or VECTOR. If the data type is a user-defined type (UDT), it must have a MAP or ORDER method to be comparable.

The third argument specifies the variant of the result returned by the functions. It is of type VARCHAR2 and must be a constant expression, for example, a character literal. If you omit the third argument, then the default is 'COEFFICIENT'. The allowed argument values and their meaning are shown in Table 7-2 Table 7-2:

> **See Also:**
>
> - Table 2-9 for more information on implicit conversion and Numeric Precedence for information on numeric precedence
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation CORR_K and CORR_S use to compare characters from *expr1* with characters from *expr2*

Table 7-2 CORR_* Return Values

Return Value	Meaning
'COEFFICIENT'	Coefficient of correlation

Table 7-2 (Cont.) CORR_* Return Values

Return Value	Meaning
'ONE_SIDED_SIG'	Positive one-tailed significance of the correlation
'ONE_SIDED_SIG_POS'	Same as ONE_SIDED_SIG
'ONE_SIDED_SIG_NEG'	Negative one-tailed significance of the correlation
'TWO_SIDED_SIG'	Two-tailed significance of the correlation

CORR_S

CORR_S calculates the Spearman's rho correlation coefficient. The input expressions should be a set of (x_i, y_i) pairs of observations. The function first replaces each value with a rank. Each value of x_i is replaced with its rank among all the other x_is in the sample, and each value of y_i is replaced with its rank among all the other y_is. Thus, each x_i and y_i take on a value from 1 to *n*, where *n* is the total number of pairs of values. Ties are assigned the average of the ranks they would have had if their values had been slightly different. Then the function calculates the linear correlation coefficient of the ranks.

CORR_S Example

Using Spearman's rho correlation coefficient, the following example derives a coefficient of correlation for each of two different comparisons -- salary and commission_pct, and salary and employee_id:

```
SELECT COUNT(*) count,
       CORR_S(salary, commission_pct) commission,
       CORR_S(salary, employee_id) empid
  FROM employees;

     COUNT COMMISSION      EMPID
---------- ---------- ----------
       107 .735837022 -.04473016
```

CORR_K

CORR_K calculates the Kendall's tau-b correlation coefficient. As for CORR_S, the input expressions are a set of (x_i, y_i) pairs of observations. To calculate the coefficient, the function counts the number of concordant and discordant pairs. A pair of observations is concordant if the observation with the larger x also has a larger value of y. A pair of observations is discordant if the observation with the larger x has a smaller y.

The significance of tau-b is the probability that the correlation indicated by tau-b was due to chance—a value of 0 to 1. A small value indicates a significant correlation for positive values of tau-b (or anticorrelation for negative values of tau-b).

CORR_K Example

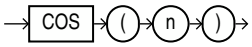
Using Kendall's tau-b correlation coefficient, the following example determines whether a correlation exists between an employee's salary and commission percent:

```
SELECT CORR_K(salary, commission_pct, 'COEFFICIENT') coefficient,
       CORR_K(salary, commission_pct, 'TWO_SIDED_SIG') two_sided_p_value
  FROM employees;
```

```
COEFFICIENT TWO_SIDED_P_VALUE
----------- -----------------
 .603079768        3.4702E-07
```

COS

Syntax



Purpose

`COS` returns the cosine of *n* (an angle expressed in radians).

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

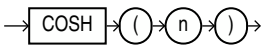
Examples

The following example returns the cosine of 180 degrees:

```
SELECT COS(180 * 3.14159265359/180) "Cosine of 180 degrees"
  FROM DUAL;

Cosine of 180 degrees
---------------------
                   -1
```

COSH

Syntax



Purpose

`COSH` returns the hyperbolic cosine of *n*.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

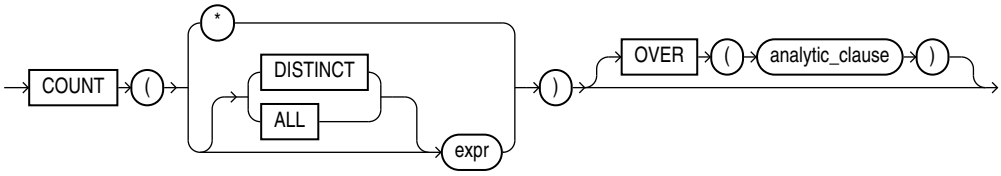
Examples

The following example returns the hyperbolic cosine of zero:

```
SELECT COSH(0) "Hyperbolic cosine of 0"
  FROM DUAL;

Hyperbolic cosine of 0
----------------------
                     1
```

COUNT

Syntax



> **See Also:**
>
> Analytic Functions for information on syntax, semantics, and restrictions

Purpose

`COUNT` returns the number of rows returned by the query. You can use it as an aggregate or analytic function.

If you specify `DISTINCT`, then you can specify only the *`query_partition_clause`* of the *`analytic_clause`*. The *`order_by_clause`* and *`windowing_clause`* are not allowed.

If you specify *`expr`*, then `COUNT` returns the number of rows where *`expr`* is not null. You can count either all rows, or only distinct values of *`expr`*.

If you specify the asterisk (*), then this function returns all rows, including duplicates and nulls. `COUNT` never returns null.

> **Note:**
>
> Before performing a `COUNT (DISTINCT expr)` operation on a large amount of data, consider using one of the following methods to obtain approximate results more quickly than exact results:
>
> - Set the `APPROX_FOR_COUNT_DISTINCT` initialization parameter to true before using the `COUNT (DISTINCT expr)` function. Refer to *Oracle Database Reference* for more information on this parameter.
> - Use the `APPROX_COUNT_DISTINCT` function instead of the `COUNT (DISTINCT expr)` function. Refer to APPROX_COUNT_DISTINCT.

> **See Also:**
>
> - "About SQL Expressions " for information on valid forms of *expr* and Aggregate Functions
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `COUNT` uses to compare character values for the `DISTINCT` clause

Aggregate Examples

The following examples use `COUNT` as an aggregate function:

```
SELECT COUNT(*) "Total"
  FROM employees;

     Total
----------
       107

SELECT COUNT(*) "Allstars"
  FROM employees
  WHERE commission_pct > 0;

 Allstars
---------
       35

SELECT COUNT(commission_pct) "Count"
  FROM employees;

     Count
----------
        35

SELECT COUNT(DISTINCT manager_id) "Managers"
  FROM employees;

  Managers
----------
        18
```

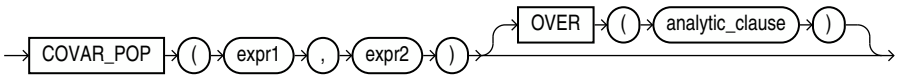
Analytic Example

The following example calculates, for each employee in the `employees` table, the moving count of employees earning salaries in the range 50 less than through 150 greater than the employee's salary.

```
SELECT last_name, salary,
       COUNT(*) OVER (ORDER BY salary RANGE BETWEEN 50 PRECEDING AND
                      150 FOLLOWING) AS mov_count
  FROM employees
  ORDER BY salary, last_name;

LAST_NAME                     SALARY  MOV_COUNT
------------------------- ---------- ----------
Olson                           2100          3
Markle                          2200          2
Philtanker                      2200          2
Gee                             2400          8
Landry                          2400          8
Colmenares                      2500         10
Marlow                          2500         10
Patel                           2500         10
. . .
```

COVAR_POP

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`COVAR_POP` returns the population covariance of a set of number pairs. You can use it as an aggregate or analytic function.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and Numeric Precedence for information on numeric precedence

Oracle Database applies the function to the set of (*expr1*, *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
(SUM(expr1 * expr2) - SUM(expr2) * SUM(expr1) / n) / n
```

where *n* is the number of (*expr1*, *expr2*) pairs where neither *expr1* nor *expr2* is null.

The function returns a value of type `NUMBER`. If the function is applied to an empty set, then it returns null.

> **See Also:**
>
> About SQL Expressions for information on valid forms of *expr* and Aggregate Functions

Aggregate Example

The following example calculates the population covariance and sample covariance for time employed (`SYSDATE - hire_date`) and salary using the sample table `hr.employees`:

```
SELECT job_id,
       COVAR_POP(SYSDATE-hire_date, salary) AS covar_pop,
       COVAR_SAMP(SYSDATE-hire_date, salary) AS covar_samp
  FROM employees
  WHERE department_id in (50, 80)
  GROUP BY job_id
  ORDER BY job_id, covar_pop, covar_samp;

JOB_ID       COVAR_POP  COVAR_SAMP
---------- ----------- -----------
SA_MAN          660700      825875
SA_REP      579988.466   600702.34
SH_CLERK      212432.5  223613.158
ST_CLERK     176577.25  185870.789
ST_MAN          436092      545115
```

Analytic Example

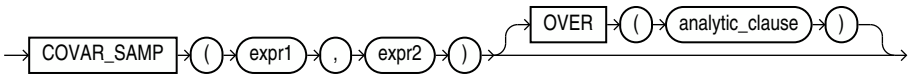
The following example calculates cumulative sample covariance of the list price and minimum price of the products in the sample schema `oe`:

```
SELECT product_id, supplier_id,
       COVAR_POP(list_price, min_price)
         OVER (ORDER BY product_id, supplier_id)
         AS CUM_COVP,
       COVAR_SAMP(list_price, min_price)
         OVER (ORDER BY product_id, supplier_id)
         AS CUM_COVS
  FROM product_information p
  WHERE category_id = 29
  ORDER BY product_id, supplier_id;

PRODUCT_ID SUPPLIER_ID   CUM_COVP   CUM_COVS
---------- ----------- ---------- ----------
      1774      103088          0
      1775      103087    1473.25     2946.5
      1794      103096 1702.77778 2554.16667
      1825      103093    1926.25 2568.33333
```

```
      2004      103086     1591.4    1989.25
      2005      103086     1512.5       1815
      2416      103088 1475.97959 1721.97619
. . .
```

COVAR_SAMP

Syntax



> **See Also:**
>
> Analytic Functions for information on syntax, semantics, and restrictions

Purpose

`COVAR_SAMP` returns the sample covariance of a set of number pairs. You can use it as an aggregate or analytic function.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and Numeric Precedence for information on numeric precedence

Oracle Database applies the function to the set of (*expr1*, *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
(SUM(expr1 * expr2) - SUM(expr1) * SUM(expr2) / n) / (n-1)
```

where *n* is the number of (*expr1*, *expr2*) pairs where neither *expr1* nor *expr2* is null.

The function returns a value of type `NUMBER`. If the function is applied to an empty set, then it returns null.

> **See Also:**
>
> About SQL Expressions for information on valid forms of *expr* and Aggregate Functions

Aggregate Example

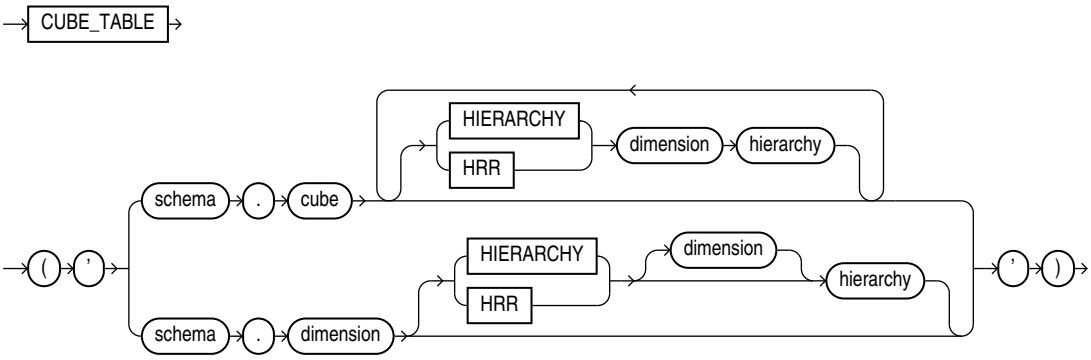
Refer to the aggregate example for COVAR_POP .

Analytic Example

Refer to the analytic example for COVAR_POP .

CUBE_TABLE

Syntax



Purpose

`CUBE_TABLE` extracts data from a cube or dimension and returns it in the two-dimensional format of a relational table, which can be used by SQL-based applications.

The function takes a single `VARCHAR2` argument. The optional hierarchy clause enables you to specify a dimension hierarchy. A cube can have multiple hierarchy clauses, one for each dimension.

You can generate these different types of tables:

- A cube table contains a key column for each dimension and a column for each measure and calculated measure in the cube. To create a cube table, you can specify the cube with or without a cube hierarchy clause. For a dimension with multiple hierarchies, this clause limits the return values to the dimension members and levels in the specified hierarchy. Without a hierarchy clause, all dimension members and all levels are included.
- A dimension table contains a key column, and a column for each level and each attribute. It also contains a `MEMBER_TYPE` column, which identifies each member with one of the following codes:
  - `L` - Loaded from a table, view, or synonym
  - `A` - Loaded member and the single root of all hierarchies in the dimension, that is, the "all" aggregate member
  - `C` - Calculated member

  All dimension members and all levels are included in the table. To create a dimension table, specify the dimension **without** a dimension hierarchy clause.
- A hierarchy table contains all the columns of a dimension table plus a column for the parent member and a column for each source level. It also contains a `MEMBER_TYPE`

column, as described for dimension tables. Any dimension members and levels that are not part of the named hierarchy are excluded from the table. To create a hierarchy table, specify the dimension **with** a dimension hierarchy clause.

CUBE_TABLE is a table function and is always used in the context of a SELECT statement with this syntax:

```
SELECT ... FROM TABLE(CUBE_TABLE('arg'));
```

> **See Also:**
>
> - *Oracle OLAP User's Guide* for information about dimensional objects and about the tables generated by CUBE_TABLE.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to each character data type column in the table generated by CUBE_TABLE

Examples

The following examples require Oracle Database with the OLAP option and the GLOBAL sample schema. Refer to *Oracle OLAP User's Guide* for information on downloading and installing the GLOBAL sample schema.

The following SELECT statement generates a dimension table of CHANNEL in the GLOBAL schema.

```
SELECT dim_key, level_name, long_description, channel_total_id tot_id,
       channel_channel_id chan_id, channel_long_description chan_desc,
       total_long_description tot_desc
  FROM TABLE(CUBE_TABLE('global.channel'));

DIM_KEY      LEVEL_NAME LONG_DESCRIPTION TOT_ID CHAN_ID CHAN_DESC    TOT_DESC
-----------  ---------- ---------------- ------ ------- ------------ -------------
CHANNEL_CAT  CHANNEL    Catalog          TOTAL  CAT     Catalog      Total Channel
CHANNEL_DIR  CHANNEL    Direct Sales     TOTAL  DIR     Direct Sales Total Channel
CHANNEL_INT  CHANNEL    Internet         TOTAL  INT     Internet     Total Channel
TOTAL_TOTAL  TOTAL      Total Channel    TOTAL                       Total Channel
```

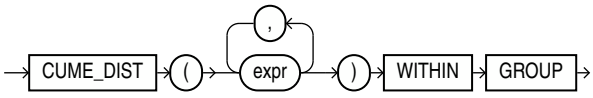
The next statement generates a cube table of UNITS_CUBE. It restricts the table to the MARKET and CALENDAR hierarchies.

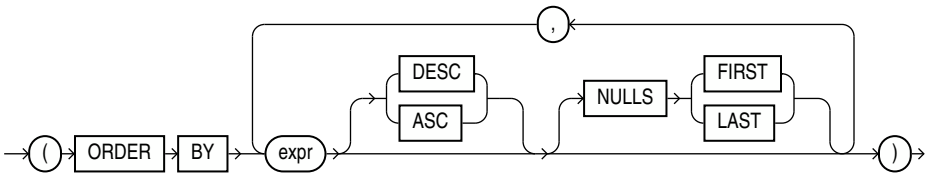
```
SELECT sales, units, cost, time, customer, product, channel
  FROM TABLE(CUBE_TABLE('global.units_cube HIERARCHY customer market HIERARCHY time calendar'))
  WHERE rownum < 20;

     SALES      UNITS       COST TIME                      CUSTOMER       PRODUCT     CHANNEL
---------- ---------- ---------- ------------------------- -------------- ----------- -----------
24538587.9      61109 22840853.7 CALENDAR_QUARTER_CY1998.Q1 TOTAL_TOTAL    TOTAL_TOTAL TOTAL_TOTAL
24993273.3      61320   23147171 CALENDAR_QUARTER_CY1998.Q2 TOTAL_TOTAL    TOTAL_TOTAL TOTAL_TOTAL
25080541.4      65265 23242535.4 CALENDAR_QUARTER_CY1998.Q3 TOTAL_TOTAL    TOTAL_TOTAL TOTAL_TOTAL
  26258474      66122 24391020.6 CALENDAR_QUARTER_CY1998.Q4 TOTAL_TOTAL    TOTAL_TOTAL TOTAL_TOTAL
  32785170      77589 30607218.1 CALENDAR_QUARTER_CY1999.Q1 TOTAL_TOTAL    TOTAL_TOTAL TOTAL_TOTAL
. . .
```

CUME_DIST

Aggregate Syntax

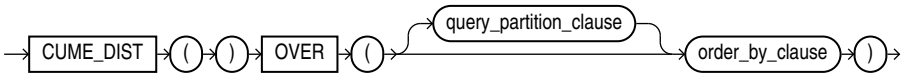
***cume_dist_aggregate*::=**





Analytic Syntax

***cume_dist_analytic*::=**



> **See Also:**
>
> Analytic Functions for information on syntax, semantics, and restrictions

Purpose

`CUME_DIST` calculates the cumulative distribution of a value in a group of values. The range of values returned by `CUME_DIST` is >0 to <=1. Tie values always evaluate to the same cumulative distribution value.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle Database determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, makes the calculation, and returns `NUMBER`.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and Numeric Precedence for information on numeric precedence

- As an aggregate function, `CUME_DIST` calculates, for a hypothetical row *r* identified by the arguments of the function and a corresponding sort specification, the relative position of

row *r* among the rows in the aggregation group. Oracle makes this calculation as if the hypothetical row *r* were inserted into the group of rows to be aggregated over. The arguments of the function identify a single hypothetical row within each aggregate group. Therefore, they must all evaluate to constant expressions within each aggregate group. The constant argument expressions and the expressions in the `ORDER BY` clause of the aggregate match by position. Therefore, the number of arguments must be the same and their types must be compatible.

- As an analytic function, `CUME_DIST` computes the relative position of a specified value in a group of values. For a row *r*, assuming ascending ordering, the `CUME_DIST` of *r* is the number of rows with values lower than or equal to the value of *r*, divided by the number of rows being evaluated (the entire query result set or a partition).

Aggregate Example

The following example calculates the cumulative distribution of a hypothetical employee with a salary of $15,500 and commission rate of 5% among the employees in the sample table `oe.employees`:

```
SELECT CUME_DIST(15500, .05) WITHIN GROUP
  (ORDER BY salary, commission_pct) "Cume-Dist of 15500"
  FROM employees;

Cume-Dist of 15500
------------------
        .972222222
```

Analytic Example

The following example calculates the salary percentile for each employee in the purchasing division. For example, 40% of clerks have salaries less than or equal to Himuro.

```
SELECT job_id, last_name, salary, CUME_DIST()
  OVER (PARTITION BY job_id ORDER BY salary) AS cume_dist
  FROM employees
  WHERE job_id LIKE 'PU%'
  ORDER BY job_id, last_name, salary, cume_dist;

JOB_ID     LAST_NAME                 SALARY  CUME_DIST
---------- ------------------------- ---------- ----------
PU_CLERK   Baida                       2900         .8
PU_CLERK   Colmenares                  2500         .2
PU_CLERK   Himuro                      2600         .4
PU_CLERK   Khoo                        3100          1
PU_CLERK   Tobias                      2800         .6
PU_MAN     Raphaely                   11000          1
```

CURRENT_DATE

Syntax

Purpose

`CURRENT_DATE` returns the current date in the session time zone, in a value in the Gregorian calendar of data type `DATE`.

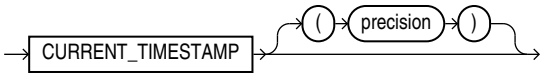
Examples

The following example illustrates that CURRENT_DATE is sensitive to the session time zone:

```
ALTER SESSION SET TIME_ZONE = '-5:0';
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;

SESSIONTIMEZONE CURRENT_DATE
--------------- --------------------
-05:00          29-MAY-2000 13:14:03

ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;

SESSIONTIMEZONE CURRENT_DATE
--------------- --------------------
-08:00          29-MAY-2000 10:14:33
```

CURRENT_TIMESTAMP

Syntax



Purpose

CURRENT_TIMESTAMP returns the current date and time in the session time zone, in a value of data type TIMESTAMP WITH TIME ZONE. The time zone offset reflects the current local time of the SQL session. If you omit precision, then the default is 6. The difference between this function and LOCALTIMESTAMP is that CURRENT_TIMESTAMP returns a TIMESTAMP WITH TIME ZONE value while LOCALTIMESTAMP returns a TIMESTAMP value.

In the optional argument, *precision* specifies the fractional second precision of the time value returned.

> **See Also:**
>
> LOCALTIMESTAMP

Examples

The following example illustrates that CURRENT_TIMESTAMP is sensitive to the session time zone:

```
ALTER SESSION SET TIME_ZONE = '-5:0';
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;

SESSIONTIMEZONE CURRENT_TIMESTAMP
--------------- ---------------------------------------------------
-05:00          04-APR-00 01.17.56.917550 PM -05:00
```

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;

SESSIONTIMEZONE CURRENT_TIMESTAMP
--------------- ---------------------------------------------------
-08:00          04-APR-00 10.18.21.366065 AM -08:00
```

When you use the `CURRENT_TIMESTAMP` with a format mask, take care that the format mask matches the value returned by the function. For example, consider the following table:

```
CREATE TABLE current_test (col1 TIMESTAMP WITH TIME ZONE);
```

The following statement fails because the mask does not include the `TIME ZONE` portion of the type returned by the function:

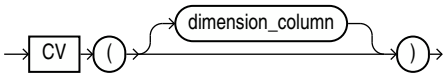
```
INSERT INTO current_test VALUES
  (TO_TIMESTAMP_TZ(CURRENT_TIMESTAMP, 'DD-MON-RR HH.MI.SSXFF PM'));
```

The following statement uses the correct format mask to match the return type of `CURRENT_TIMESTAMP`:

```
INSERT INTO current_test VALUES
  (TO_TIMESTAMP_TZ(CURRENT_TIMESTAMP, 'DD-MON-RR HH.MI.SSXFF PM TZH:TZM'));
```

CV

Syntax



Purpose

The `CV` function can be used only in the *`model_clause`* of a `SELECT` statement and then only on the right-hand side of a model rule. It returns the current value of a dimension column or a partitioning column carried from the left-hand side to the right-hand side of a rule. This function is used in the *`model_clause`* to provide relative indexing with respect to the dimension column. The return type is that of the data type of the dimension column. If you omit the argument, then it defaults to the dimension column associated with the relative position of the function within the cell reference.

The `CV` function can be used outside a cell reference. In this case, *`dimension_column`* is required.

> **See Also:**
>
> - *model_clause* and Model Expressions for the syntax and semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `CV` when it is a character value

Examples

The following example assigns the sum of the sales of the product represented by the current value of the dimension column (Mouse Pad or Standard Mouse) for years 1999 and 2000 to the sales of that product for year 2001:

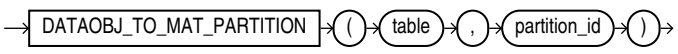
```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s[FOR prod IN ('Mouse Pad', 'Standard Mouse'), 2001] =
        s[CV( ), 1999] + s[CV( ), 2000]
    )
  ORDER BY country, prod, year;
```

```
COUNTRY       PROD                                    YEAR          S
----------    ----------------------------------      --------   ---------
France        Mouse Pad                                  1998     2509.42
France        Mouse Pad                                  1999     3678.69
France        Mouse Pad                                  2000     3000.72
France        Mouse Pad                                  2001     6679.41
France        Standard Mouse                             1998     2390.83
France        Standard Mouse                             1999     2280.45
France        Standard Mouse                             2000     1274.31
France        Standard Mouse                             2001     3554.76
Germany       Mouse Pad                                  1998     5827.87
Germany       Mouse Pad                                  1999     8346.44
Germany       Mouse Pad                                  2000     7375.46
Germany       Mouse Pad                                  2001     15721.9
Germany       Standard Mouse                             1998     7116.11
Germany       Standard Mouse                             1999     6263.14
Germany       Standard Mouse                             2000     2637.31
Germany       Standard Mouse                             2001     8900.45

16 rows selected.
```

The preceding example requires the view `sales_view_ref`. Refer to The MODEL clause: Examples to create this view.

DATAOBJ_TO_MAT_PARTITION

Syntax



Purpose

`DATAOBJ_TO_MAT_PARTITION` is useful only to Data Cartridge developers who are performing data maintenance or query operations on system-partitioned tables that are used to store domain index data. The DML or query operations are triggered by corresponding operations on the base table of the domain index.

This function takes as arguments the name of the base table and the partition ID of the base table partition, both of which are passed to the function by the appropriate ODCIIndex method. The function returns the materialized partition number of the corresponding system-partitioned table, which can be used to perform the operation (DML or query) on that partition of the system-partitioned table.

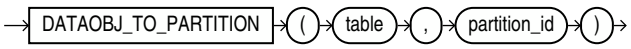
If the base table is interval partitioned, then Oracle recommends that you use this function instead of the `DATAOBJ_TO_PARTITION` function. The `DATAOBJ_TO_PARTITION` function determines the absolute partition number, given the physical partition identifier. However, if the base table is interval partitioned, then there might be holes in the partition numbers corresponding to unmaterialized partitions. Because the system partitioned table only has materialized partitions, `DATAOBJ_TO_PARTITION` numbers can cause a mis-match between the partitions of the base table and the partitions of the underlying system partitioned index storage tables. The `DATAOBJ_TO_MAT_PARTITION` function returns the materialized partition number (as opposed to the absolute partition number) and helps keep the two tables in sync. Indextypes planning to support local domain indexes on interval partitioned tables should migrate to the use of this function.

See Also:

- DATAOBJ_TO_PARTITION
- *Oracle Database Data Cartridge Developer's Guide* for information on the use of the `DATAOBJ_TO_MAT_PARTITION` function, including examples

DATAOBJ_TO_PARTITION

Syntax



Purpose

`DATAOBJ_TO_PARTITION` is useful only to Data Cartridge developers who are performing data maintenance or query operations on system-partitioned tables that are used to store domain index data. The DML or query operations are triggered by corresponding operations on the base table of the domain index.

This function takes as arguments the name of the base table and the partition ID of the base table partition, both of which are passed to the function by the appropriate ODCIIndex method. The function returns the absolute partition number of the corresponding system-partitioned table, which can be used to perform the operation (DML or query) on that partition of the system-partitioned table.

Note:

If the base table is interval partitioned, then Oracle recommends that you instead use the `DATAOBJ_TO_MAT_PARTITION` function. Refer to DATAOBJ_TO_MAT_PARTITION for more information.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for information on the use of the `DATAOBJ_TO_PARTITION` function, including examples

DBTIMEZONE

Syntax

Purpose

`DBTIMEZONE` returns the value of the database time zone. The return type is a time zone offset (a character type in the format `'[+|-]TZH:TZM'`) or a time zone region name, depending on how the user specified the database time zone value in the most recent `CREATE DATABASE` or `ALTER DATABASE` statement.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `DBTIMEZONE`

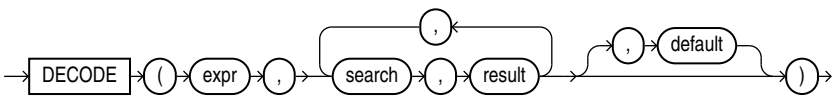
Examples

The following example assumes that the database time zone is set to UTC time zone:

```
SELECT DBTIMEZONE
  FROM DUAL;

DBTIME
------
+00:00
```

DECODE

Syntax



Purpose

DECODE compares *expr* to each *search* value one by one. If *expr* is equal to a *search*, then Oracle Database returns the corresponding *result*. If no match is found, then Oracle returns *default*. If *default* is omitted, then Oracle returns null.

- If *expr* and *search* are character data, then Oracle compares them using nonpadded comparison semantics. *expr*, *search*, and *result* can be any of the data types CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as the first *result* parameter.
- If the first *search-result* pair are numeric, then Oracle compares all *search-result* expressions and the first *expr* to determine the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

The *search*, *result*, and *default* values can be derived from expressions. Oracle Database uses **short-circuit evaluation**. The database evaluates each *search* value only before comparing it to *expr*, rather than evaluating all *search* values before comparing any of them with *expr*. Consequently, Oracle never evaluates a *search* if a previous *search* is equal to *expr*.

Oracle automatically converts *expr* and each *search* value to the data type of the first *search* value before comparing. Oracle automatically converts the return value to the same data type as the first *result*. If the first *result* has the data type CHAR or if the first *result* is null, then Oracle converts the return value to the data type VARCHAR2.

In a DECODE function, Oracle considers two nulls to be equivalent. If *expr* is null, then Oracle returns the *result* of the first *search* that is also null.

The maximum number of components in the DECODE function, including *expr*, *searches*, *results*, and *default*, is 255.

See Also:

- Data Type Comparison Rules for information on comparison semantics
- Data Conversion for information on data type conversion in general
- Floating-Point Numbers for information on floating-point comparison semantics
- Implicit and Explicit Data Conversion for information on the drawbacks of implicit conversion
- COALESCE and CASE Expressions , which provide functionality similar to that of DECODE
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation DECODE uses to compare characters from *expr* with characters from *search*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

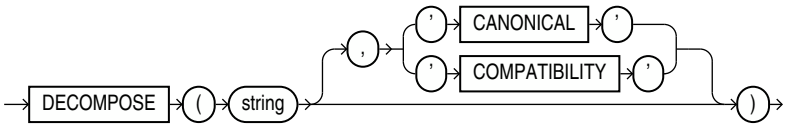
Examples

This example decodes the value `warehouse_id`. If `warehouse_id` is 1, then the function returns 'Southlake'; if `warehouse_id` is 2, then it returns 'San Francisco'; and so forth. If `warehouse_id` is not 1, 2, 3, or 4, then the function returns 'Non domestic'.

```
SELECT product_id,
       DECODE (warehouse_id, 1, 'Southlake',
                             2, 'San Francisco',
                             3, 'New Jersey',
                             4, 'Seattle',
                                'Non domestic') "Location"
  FROM inventories
  WHERE product_id < 1775
  ORDER BY product_id, "Location";
```

DECOMPOSE

Syntax



Purpose

`DECOMPOSE` takes as its first argument a character value *`string`* and returns the result of applying one of the Unicode decompositions to it. The decomposition to apply is determined by the second, optional parameter. If the character set of the first argument is not one of the Unicode character sets, `DECOMPOSE` returns the argument unmodified.

If the second argument to `DECOMPOSE` is the string `CANONICAL` (case-insensitively), `DECOMPOSE` applies canonical decomposition, as described in the Unicode Standard definition D68, and returns a string in the NFD normalization form. If the second argument is the string `COMPATIBILITY`, `DECOMPOSE` applies compatibility decomposition, as described in the Unicode Standard definition D65, and returns a string in the NFKD normalization form. The default behavior is to apply the canonical decomposition.

In a pessimistic case, the return value of `DECOMPOSE` may be a few times longer than *`string`*. If a string to be returned is longer than the maximum length `VARCHAR2` value in a given runtime environment, the value is silently truncated to the maximum `VARCHAR2` length.

Both arguments to `DECOMPOSE` can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. Other data types are allowed if they can be implicitly converted to `VARCHAR2`or `NVARCHAR2`. The return value of `DECOMPOSE` is in the same character set as its first argument.

`CLOB` and `NCLOB` values are supported through implicit conversion. If *`string`* is a character LOB value, then it is converted to a `VARCHAR2` value before the `DECOMPOSE` operation. The operation will fail if the size of the LOB value exceeds the supported length of the `VARCHAR2` in the particular execution environment.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for information on Unicode character sets and character semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `DECOMPOSE`
> - COMPOSE

Examples

The following example decomposes the string "`Châteaux`" into its component code points:

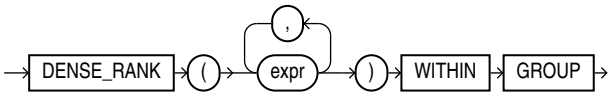
```
SELECT DECOMPOSE ('Châteaux')
  FROM DUAL;

DECOMPOSE
---------
Châteaux
```

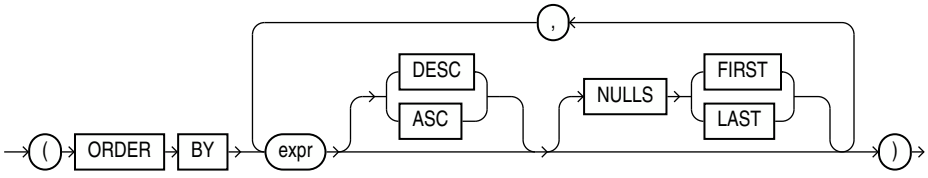
> **Note:**
>
> The results of this example can vary depending on the character set of your operating system.

DENSE_RANK

Aggregate Syntax

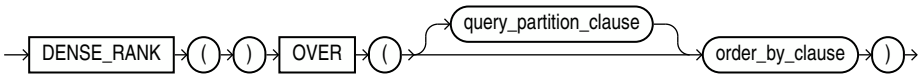
***dense_rank_aggregate*::=**





Analytic Syntax

***dense_rank_analytic*::=**



> **See Also:**
>
> Analytic Functions for information on syntax, semantics, and restrictions

Purpose

DENSE_RANK computes the rank of a row in an ordered group of rows and returns the rank as a NUMBER. The ranks are consecutive integers beginning with 1. The largest rank value is the number of unique values returned by the query. Rank values are not skipped in the event of ties. Rows with equal values for the ranking criteria receive the same rank. This function is useful for top-N and bottom-N reporting.

This function accepts as arguments any numeric data type and returns NUMBER.

- As an aggregate function, DENSE_RANK calculates the dense rank of a hypothetical row identified by the arguments of the function with respect to a given sort specification. The arguments of the function must all evaluate to constant expressions within each aggregate group, because they identify a single row within each group. The constant argument expressions and the expressions in the *order_by_clause* of the aggregate match by position. Therefore, the number of arguments must be the same and types must be compatible.
- As an analytic function, DENSE_RANK computes the rank of each row returned from a query with respect to the other rows, based on the values of the *value_exprs* in the *order_by_clause*.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation DENSE_RANK uses to compare character values for the ORDER BY clause

Aggregate Example

The following example computes the ranking of a hypothetical employee with the salary $15,500 and a commission of 5% in the sample table oe.employees:

```
SELECT DENSE_RANK(15500, .05) WITHIN GROUP
  (ORDER BY salary DESC, commission_pct) "Dense Rank"
  FROM employees;

Dense Rank
----------
         3
```

Analytic Example

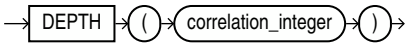
The following statement ranks the employees in the sample hr schema in department 60 based on their salaries. Identical salary values receive the same rank. However, no rank values are skipped. Compare this example with the analytic example for RANK .

```
SELECT department_id, last_name, salary,
       DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary) DENSE_RANK
  FROM employees WHERE department_id = 60
```

```
  ORDER BY DENSE_RANK, last_name;

DEPARTMENT_ID LAST_NAME                 SALARY DENSE_RANK
------------- ------------------------- ---------- ----------
           60 Lorentz                         4200          1
           60 Austin                          4800          2
           60 Pataballa                       4800          2
           60 Ernst                           6000          3
           60 Hunold                          9000          4
```

DEPTH

Syntax



Purpose

`DEPTH` is an ancillary function used only with the `UNDER_PATH` and `EQUALS_PATH` conditions. It returns the number of levels in the path specified by the `UNDER_PATH` condition with the same correlation variable.

The *`correlation_integer`* can be any `NUMBER` integer. Use it to correlate this ancillary function with its primary condition if the statement contains multiple primary conditions. Values less than 1 are treated as 1.

> **See Also:**
>
> EQUALS_PATH Condition , UNDER_PATH Condition , and the related function PATH

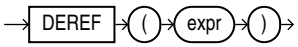
Examples

The `EQUALS_PATH` and `UNDER_PATH` conditions can take two ancillary functions, `DEPTH` and `PATH`. The following example shows the use of both ancillary functions. The example assumes the existence of the XMLSchema `warehouses.xsd` (created in Using XML in SQL Statements).

```
SELECT PATH(1), DEPTH(2)
  FROM RESOURCE_VIEW
  WHERE UNDER_PATH(res, '/sys/schemas/OE', 1)=1
    AND UNDER_PATH(res, '/sys/schemas/OE', 2)=1;

PATH(1)                         DEPTH(2)
------------------------------- --------
. . .
www.example.com                        1
www.example.com/xwarehouses.xsd        2
. . .
```

DEREF

Syntax



Purpose

`DEREF` returns the object reference of argument *`expr`*, where *`expr`* must return a `REF` to an object. If you do not use this function in a query, then Oracle Database returns the object ID of the `REF` instead, as shown in the example that follows.

> **See Also:**
>
> MAKE_REF

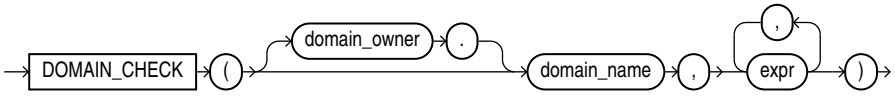
Examples

The sample schema `oe` contains an object type `cust_address_typ`. The REF Constraint Examples create a similar type, `cust_address_typ_new`, and a table with one column that is a `REF` to the type. The following example shows how to insert into such a column and how to use `DEREF` to extract information from the column:

```
INSERT INTO address_table VALUES
  ('1 First', 'G45 EU8', 'Paris', 'CA', 'US');

INSERT INTO customer_addresses
  SELECT 999, REF(a) FROM address_table a;

SELECT address
  FROM customer_addresses
  ORDER BY address;

ADDRESS
--------------------------------------------------------------------------------
000022020876B2245DBE325C5FE03400400B40DCB176B2245DBE305C5FE03400400B40DCB1

SELECT DEREF(address)
  FROM customer_addresses;

DEREF(ADDRESS)(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
--------------------------------------------------------------------------------
CUST_ADDRESS_TYP_NEW('1 First', 'G45 EU8', 'Paris', 'CA', 'US')
```

DOMAIN_CHECK

Syntax



Purpose

DOMAIN_CHECK first converts the data type of the arguments in *expr* to the data type of their corresponding domain columns. It then applies the constraint conditions (not null or check constraint) on *domain_name* to *expr*.

If the domain's constraint is deferred or unvalidated, DOMAIN_CHECK still applies the conditions to *expr*. If the domain's constraint is disabled, it is not checked as part of DOMAIN_CHECK.

> **See Also:**
>
> Domain Functions

- *domain_name* must be an identifier and can be specified using *domain_owner.domain_name*. If you specify it without *domain_owner*, it resolves first to the current user then as a public synonym. If the name cannot be resolved, an error is raised.
- If *domain_name* refers to a non-existent domain or one that you do not have EXECUTE privileges on, then DOMAIN_CHECK will raise an error.
- If the domain column data type is STRICT, then the value is converted to the domain column's data type. For example, if the domain column data type is VARCHAR2(100) STRICT, then the value is converted to VARCHAR2(100). Note that the conversion will not automatically trim the input to the maximum length. If the value evaluates to 'abc' for some row and the domain data type is CHAR(2 CHAR), the conversion will fail instead of returning 'ab'.

  If the domain column data type is not STRICT, then the value is converted to the most permissive variant of the domain column's data type in terms of length, scale, and precision. For example, if the input value is a VARCHAR2(30), it is converted to a VARCHAR2(100) because it is shorter than the domain length. If the input value is a VARCHAR2(200), it remains a VARCHAR2(200) because this is larger than the domain length.
- If the data type conversion fails, the error is masked and DOMAIN_CHECK returns FALSE. You can use DOMAIN_CHECK to filter out values that cannot be inserted into a column of the given domain.

  If the data type conversion succeeds and *domain_name* does not have any enabled constraint associated with it, DOMAIN_CHECK returns TRUE.
- If the data type conversion succeeds and *domain_name* has enabled constraints that are all satisfied for a given converted value, DOMAIN_CHECK returns TRUE. If any of the domain constraints are not satisfied, it returns FALSE .

MULTI-COLUMN Domains

When calling DOMAIN_CHECK for multicolumn domains, the number if arguments for *expr* must match the number of columns in the domain. If there is a mismatch, DOMAIN_CHECK raises an error.

If domain D has n columns, then you should call DOMAIN_CHECK should be called with D+1 arguments, like DOMAIN_CHECK(D, arg1, ..., argn).

If D does not exist or you have no privilege to access D, then an error is raised. If all the checks return true, TRUE is returned. This means that:

- `arg1` is successfully converted to the data type of column 1 in `D`, `arg2` is successfully converted to the data type of column 2 in `D`and so on to `argn` is successfully converted to the data type of column `n` in `D` .
- All of `D`'s enabled constraints are all satisfied with column 1 substituted by `arg1` converted to `D`'s column 1 data type, column 2 substituted by `arg2` converted to `D`'s column 2 data type, and so on to column `n` substituted by `argn` converted to `D`'s column `n` data type .

Example

The following example creates a domain `dgreater` with two columns `c1` and `c2` of type `NUMBER` and a check constraint that `c1` be greater than `c2`:

```
CREATE DOMAIN dgreater AS (c1 AS NUMBER, c2 AS NUMBER ) CHECK (c1 > c2);
```

Then `DOMAIN_CHECK (dgreater, 1, 2)` returns `FALSE` because `c1` is less than `c2` (the check condition fails). `DOMAIN_CHECK (dgreater, 2, 1)` returns `TRUE` because because `c1` is greater than `c2` (the check condition passes).

Flexible Domains

When calling `DOMAIN_CHECK` for flexible domains, the number of arguments for *`expr`* must match the number of domain columns plus discriminant columns. If there is a mismatch `DOMAIN_CHECK` raises an error.

Checking flexible domain constraints is equivalent to checking constraints of the corresponding subdomain.

You must have the `EXECUTE` privilege on the flexible domain in order to use `DOMAIN_CHECK`.

Operations that require `EXECUTE` privilege on a flexible domain (such as when associating columns with the flexible domain, or during `DOMAIN_CHECK` with the first argument the flexible domain name) require `EXECUTE` privilege on the sub-domains. This is because a flexible domain is translated during its creation to a multi-column domain. Therefore the following rules apply:

- Associating columns to a flex domain is equivalent to associating them to the corresponding multi-column domain.
- Checking flexible domain constraints is equivalent to checking constraints of the corresponding multi-column domain.
- Evaluating flexible domain display and order properties is equivalent to evaluating properties on the corresponding multi-column domain.

Examples

Example 1

The following example creates a strict domain of data type `CHAR(3 CHAR)`:

```
CREATE DOMAIN three_chars AS CHAR(3 CHAR) STRICT;
```

Calling `DOMAIN_CHECK` returns true for strings three characters or shorter. For strings four characters or more long it returns false:

```
SELECT DOMAIN_CHECK (three_chars, 'ab') two_chars,
       DOMAIN_CHECK (three_chars, 'abc') three_chars,
       DOMAIN_CHECK (three_chars, 'abcd') four_chars;
```

```
TWO_CHARS   THREE_CHARS FOUR_CHARS
----------- ----------- -----------
TRUE        TRUE        FALSE
```

Example 2

The following example creates a domain dgreater with two columns `c1` and `c2` of type `NUMBER` and a check constraint that `c1` be greater than `c2`:

```
CREATE DOMAIN dgreater AS (
  c1 AS NUMBER, c2 AS NUMBER
)
  CHECK (c1 > c2);
```

The first query passes one expression value. This raises an error because there are two columns in the domain:

```
SELECT DOMAIN_CHECK (dgreater, 1) one_expr;

ORA-11515: incorrect number of columns in domain association list
```

In the second query:

- `first_lower` is `FALSE` because this fails the domain constraint
- `first_higher` is `TRUE` because it passes the domain constraint
- `letters` is `FALSE` because the values cannot be converted to numbers

```
SELECT DOMAIN_CHECK (dgreater, 1, 2) first_lower,
       DOMAIN_CHECK (dgreater, 2, 1) first_higher,
       DOMAIN_CHECK (dgreater, 'b', 'a') letters;

FIRST_LOWER FIRST_HIGHER LETTERS
----------- -----------  -----------
FALSE       TRUE         FALSE
```

Example 3

The following example creates the domain `DAY_OF_WEEK` with no domain constraints. All calls to `DOMAIN_CHECK` return true because all the input values can be converted to `CHAR`. It is a non-strict domain, so there is no length check.

```
CREATE DOMAIN day_of_week AS CHAR(3 CHAR);

CREATE TABLE calendar_dates (
  calendar_date    DATE,
  day_of_week_abbr day_of_week
);

INSERT INTO calendar_dates
VALUES(DATE'2023-05-01', 'MON'),
      (DATE'2023-05-02', 'tue'),
      (DATE'2023-05-05', 'fRI');

SELECT day_of_week_abbr,
       DOMAIN_CHECK(day_of_week, day_of_week_abbr) domain_column,
```

```
       DOMAIN_CHECK(day_of_week, calendar_date) nondomain_column,
       DOMAIN_CHECK(day_of_week, CAST('MON' AS day_of_week)) domain_value,
       DOMAIN_CHECK(day_of_week, 'mon') nondomain_value
  FROM calendar_dates;

DAY DOMAIN_COLUMN   NONDOMAIN_COLUMN   DOMAIN_VALUE   NONDOMAIN_VALUE
--- --------------- ------------------ -------------- -----------------
MON TRUE            TRUE               TRUE           TRUE
TUE TRUE            TRUE               TRUE           TRUE
FRI TRUE            TRUE               TRUE           TRUE
mon TRUE            TRUE               TRUE           TRUE
MON TRUE            TRUE               TRUE           TRUE
```

Example 4

The following example creates the domain `DAY_OF_WEEK` with a constraint to ensure the values are the uppercase day name abbreviations (`MON`, `TUE`, etc.). Validating this constraint is deferred until commit, so you can insert invalid values.

Using `DOMAIN_CHECK` to test the values for the domain column `DAY_OF_WEEK_ABBR` returns `TRUE` for the value that conforms to the constraint (`MON`) and `FALSE` for those that do not (`tue`, `fRI`):

```
CREATE DOMAIN day_of_week AS CHAR(3 CHAR)
  CONSTRAINT CHECK(day_of_week IN ('MON','TUE','WED','THU','FRI','SAT','SUN'))
  INITIALLY DEFERRED;

CREATE TABLE calendar_dates (
  calendar_date    DATE,
  day_of_week_abbr day_of_week
);

INSERT INTO calendar_dates
VALUES(DATE'2023-05-01', 'MON'),
      (DATE'2023-05-02', 'tue'),
      (DATE'2023-05-05', 'fRI');

SELECT day_of_week_abbr,
       DOMAIN_CHECK(day_of_week, day_of_week_abbr) domain_column,
       DOMAIN_CHECK(day_of_week, calendar_date) nondomain_column,
       DOMAIN_CHECK(day_of_week, CAST('MON' AS day_of_week)) domain_value,
       DOMAIN_CHECK(day_of_week, 'mon') nondomain_value
  FROM calendar_dates;

DAY DOMAIN_COLUMN NONDOMAIN_COLUMN DOMAIN_VALUE NONDOMAIN_VALUE
--- ------------- ---------------- ------------ -----------
MON TRUE          FALSE            TRUE         FALSE
tue FALSE         FALSE            TRUE         FALSE
fRI FALSE         FALSE            TRUE         FALSE
```

Example 5

The following example creates the multicolumn domain currency with two deferred constraints:

```
CREATE DOMAIN currency AS (
  amount        AS NUMBER(10, 2)
  currency_code AS CHAR(3 CHAR)
)
CONSTRAINT supported_currencies_c
  CHECK ( currency_code IN ( 'USD', 'GBP', 'EUR', 'JPY' ) )
  DEFERRABLE INITIALLY DEFERRED
```

```
CONSTRAINT non_negative_amounts_c
  CHECK ( amount >= 0 )
  DEFERRABLE INITIALLY DEFERRED;
```

The columns AMOUNT and CURRENCY_CODE in the table ORDER_ITEMS are associated with domain currency:

```
CREATE TABLE order_items (
  order_id      INTEGER,
  product_id    INTEGER,
  amount        NUMBER(10, 2),
  currency_code CHAR(3 CHAR),
  DOMAIN currency(amount, currency_code)
);
INSERT INTO order_items
VALUES (1, 1,    9.99, 'USD'),
       (2, 2, 1234.56, 'GBP'),
       (3, 3, -999999, 'JPY'),
       (4, 4, 3141592, 'XXX') ,
       (5, 5, 2718281, '123');
```

The query makes four calls to DOMAIN_CHECK:

```
SELECT order_id,
       product_id,
       amount,
       currency_code,
       DOMAIN_CHECK(currency, order_id, product_id) order_product,
       DOMAIN_CHECK(currency, amount, currency_code) amount_currency,
       DOMAIN_CHECK(currency, currency_code, amount) currency_amount,
       DOMAIN_CHECK(currency, order_id, currency_code) order_currency
  FROM order_items;

  ORDER_ID PRODUCT_ID     AMOUNT CUR ORDER_PRODUCT AMOUNT_CURRENCY CURRENCY_AMOUNT
ORDER_CURRENCY
---------- ---------- ---------- --- ------------- --------------- ---------------
-----------
         1          1       9.99 USD FALSE         TRUE            FALSE           TRUE
         2          2    1234.56 GBP FALSE         TRUE            FALSE           TRUE
         3          3    -999999 JPY FALSE         FALSE           FALSE           TRUE
         4          4    3141592 XXX FALSE         FALSE           FALSE           FALSE
         5          5    2718281 123 FALSE         FALSE           FALSE           FALSE
```

In the example above:

- ORDER_PRODUCT is FALSE for all rows because the values for PRODUCT_ID do not conform to the supported_currencies_c constraint.
- AMOUNT_CURRENCY is FALSE for the rows with values that violate the constraints (AMOUNT = -999999, and CURRENCY_CODE = "XXX" and "123"). It is TRUE for the valid values.
- CURRENCY_AMOUNT is FALSE for all rows. For the first four rows this is because the values for the first argument, CURRENCY_CODE are all letters. These cannot be converted to the type of the first column in the domain (NUMBER), leading to a type error. For the fifth row, the amount (2718281) does not conform to the supported_currencies_c constraint.
- ORDER_CURRENCY is FALSE for the row with values that violate the constraints (CURRENCY_CODE = "XXX" and "123"). It is TRUE for the valid values.

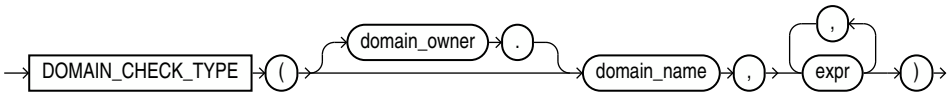
Example 6

The following statement tries to validate the string "raises an error" against the non-existent domain `NOT_A_DOMAIN`. This raises an exception:

```
SELECT DOMAIN_CHECK(not_a_domain, 'raises an error');
ORA-11504: The domain specified does not exist or the user does not have privileges on
the domain for the operation.
```

DOMAIN_CHECK_TYPE

Syntax



Purpose

Use `DOMAIN_CHECK_TYPE` to convert the value expression to the data type of the domain column without checking domain constraints. If you want to check constraints, you must use DOMAIN_CHECK.

`DOMAIN_CHECK_TYPE` takes the same arguments as `DOMAIN_CHECK` and returns `TRUE` if the data type of the arguments match the data types of the corresponding domain columns. If the data type match fails, it returns `FALSE`.

> **See Also:**
>
> Domain Functions

- *`domain_name`* must be an identifier and can be specified using *`domain_owner.domain_name`*. If you specify it without *`domain_owner`*, it resolves first to the current user then as a public synonym. If the name cannot be resolved, an error is raised.
- If *`domain_name`* refers to a non-existent domain or one that you do not have `EXECUTE` privileges on, then `DOMAIN_CHECK` will raise an error.
- If the domain column data type is `STRICT`, then the value is converted to the domain column's data type. For example, if the domain column data type is `VARCHAR2(100) STRICT`, then the value is converted to `VARCHAR2(100)`. Note that the conversion will not automatically trim the input to the maximum length. If the value evaluates to 'abc' for some row and the domain data type is `CHAR(2 CHAR)`, the conversion will fail instead of returning 'ab'.

  If the domain column data type is not `STRICT`, then the value is converted to the most permissive variant of the domain column's data type in terms of length, scale and precision. For example, if the input value is a `VARCHAR2(30)`, it is converted to a `VARCHAR2(100)` because it is shorter than the domain length. If the input value is a `VARCHAR2(200)`, it remains a`VARCHAR2(200)` because this is larger than the domain length.
- If the data type conversion fails, the error is masked and `DOMAIN_CHECK_TYPE` returns `FALSE`. You can use `DOMAIN_CHECK_TYPE` to filter out values that cannot be inserted into a column of the given domain..

MULTI-COLUMN Domains

When calling DOMAIN_CHECK_TYPE for multicolumn domains, the number of arguments for *expr* must match the number of columns in the domain. If there is a mismatch DOMAIN_CHECK_TYPE raises an error.

Flexible Domains

When calling DOMAIN_CHECK_TYPE for flexible domains, the number of arguments for *expr* must match the number of domain columns plus discriminant columns. If there is a mismatch DOMAIN_CHECK_TYPE raises an error.

Examples

Example 1

The following example creates a strict domain of data type CHAR(3 CHAR):

```
CREATE DOMAIN three_chars AS CHAR(3 CHAR) STRICT;
```

Calling DOMAIN_CHECK_TYPE returns true for strings three characters or shorter. For strings four characters or more long it returns false:

```
SELECT DOMAIN_CHECK_TYPE (three_chars, 'ab') two_chars,
       DOMAIN_CHECK_TYPE (three_chars, 'abc') three_chars,
       DOMAIN_CHECK_TYPE (three_chars, 'abcd') four_chars;

TWO_CHARS   THREE_CHARS FOUR_CHARS
----------- ----------- -----------
TRUE        TRUE        FALSE
```

Example 2

The following example creates a domain dgreater with two columns c1 and c2 of type NUMBER and a check constraint that c1 be greater than c2:

```
CREATE DOMAIN dgreater AS (
  c1 AS NUMBER, c2 AS NUMBER
)
  CHECK (c1 > c2);
```

The first query passes one expression value. This raises an error because there are two columns in the domain.

```
SELECT DOMAIN_CHECK_TYPE (dgreater, 1) one_expr;

ORA-11515: incorrect number of columns in domain association list
```

In the second query:

- first_lower and first_higher are both TRUE because the values are numbers. The domain constraint is not checked.
- letters is FALSE because the values cannot be converted to numbers.

```
SELECT DOMAIN_CHECK_TYPE (dgreater, 1, 2) first_lower,
       DOMAIN_CHECK_TYPE (dgreater, 2, 1) first_higher,
       DOMAIN_CHECK_TYPE (dgreater, 'b', 'a') letters;
```

```
FIRST_LOWER FIRST_HIGHER LETTERS
----------- -----------  -----------
TRUE        TRUE         FALSE
```

Example 3

The following example creates the domain `DAY_OF_WEEK` with no domain constraints. All calls to `DOMAIN_CHECK_TYPE` return true because all the input values can be converted to `CHAR`. It's a non-strict domain, so there is no length check.

```
CREATE DOMAIN day_of_week AS CHAR(3 CHAR);

CREATE TABLE calendar_dates (
  calendar_date    DATE,
  day_of_week_abbr day_of_week
);

INSERT INTO calendar_dates
VALUES(DATE'2023-05-01', 'MON'),
      (DATE'2023-05-02', 'tue'),
      (DATE'2023-05-05', 'fRI');

SELECT day_of_week_abbr,
       DOMAIN_CHECK_TYPE(day_of_week, day_of_week_abbr) domain_column,
       DOMAIN_CHECK_TYPE(day_of_week, calendar_date) nondomain_column,
       DOMAIN_CHECK_TYPE(day_of_week, CAST('MON' AS day_of_week)) domain_value,
       DOMAIN_CHECK_TYPE(day_of_week, 'mon') nondomain_value
  FROM calendar_dates;

DAY DOMAIN_COLUMN   NONDOMAIN_COLUMN   DOMAIN_VALUE   NONDOMAIN_VALUE
--- --------------- ------------------ -------------- -----------------
MON TRUE            TRUE               TRUE           TRUE
TUE TRUE            TRUE               TRUE           TRUE
FRI TRUE            TRUE               TRUE           TRUE
mon TRUE            TRUE               TRUE           TRUE
MON TRUE            TRUE               TRUE           TRUE
```

Example 4

The following example creates the domain DAY_OF_WEEK with a constraint to ensure the values are the uppercase day name abbreviations (MON, TUE, etc.).

Validating this constraint is deferred until commit, so you can insert invalid values.

Using `DOMAIN_CHECK_TYPE` returns `TRUE` for all values because they all pass the type check:

```
CREATE DOMAIN day_of_week AS CHAR(3 CHAR)
  CONSTRAINT CHECK(day_of_week IN ('MON','TUE','WED','THU','FRI','SAT','SUN'))
  INITIALLY DEFERRED;

CREATE TABLE calendar_dates (
  calendar_date    DATE,
  day_of_week_abbr day_of_week
);

INSERT INTO calendar_dates
VALUES(DATE'2023-05-01', 'MON'),
      (DATE'2023-05-02', 'tue'),
      (DATE'2023-05-05', 'fRI');
```

```
SELECT day_of_week_abbr,
       DOMAIN_CHECK_TYPE(day_of_week, day_of_week_abbr) domain_column,
       DOMAIN_CHECK_TYPE(day_of_week, calendar_date) nondomain_column,
       DOMAIN_CHECK_TYPE(day_of_week, CAST('MON' AS day_of_week)) domain_value,
       DOMAIN_CHECK_TYPE(day_of_week, 'mon') nondomain_value
  FROM calendar_dates;

DAY DOMAIN_COLUMN NONDOMAIN_COLUMN DOMAIN_VALUE NONDOMAIN_VALUE
--- ------------- ---------------- ------------ -----------
MON TRUE          TRUE             TRUE         TRUE
tue FALSE         TRUE             TRUE         TRUE
fRI FALSE         TRUE             TRUE         TRUE
```

Example 5

The following example creates the multicolumn domain currency with two deferred constraints:

```
CREATE DOMAIN currency AS (
  amount        AS NUMBER(10, 2)
  currency_code AS CHAR(3 CHAR)
)
CONSTRAINT supported_currencies_c
  CHECK ( currency_code IN ( 'USD', 'GBP', 'EUR', 'JPY' ) )
  DEFERRABLE INITIALLY DEFERRED
CONSTRAINT non_negative_amounts_c
  CHECK ( amount >= 0 )
  DEFERRABLE INITIALLY DEFERRED;
```

The columns `AMOUNT` and `CURRENCY_CODE` in the table `ORDER_ITEMS` are associated with domain `currency`:

```
CREATE TABLE order_items (
  order_id      INTEGER,
  product_id    INTEGER,
  amount        NUMBER(10, 2),
  currency_code CHAR(3 CHAR),
  DOMAIN currency(amount, currency_code)
);
INSERT INTO order_items
VALUES (1, 1,    9.99, 'USD'),
       (2, 2, 1234.56, 'GBP'),
       (3, 3, -999999, 'JPY'),
       (4, 4, 3141592, 'XXX') ,
       (5, 5, 2718281, '123');
```

The query makes four calls to `DOMAIN_CHECK_TYPE`:

```
SELECT order_id,
       product_id,
       amount,
       currency_code,
       DOMAIN_CHECK_TYPE(currency, order_id, product_id) order_product,
       DOMAIN_CHECK_TYPE(currency, amount, currency_code) amount_currency,
       DOMAIN_CHECK_TYPE(currency, currency_code, amount) currency_amount,
       DOMAIN_CHECK_TYPE(currency, order_id, currency_code) order_currency
  FROM order_items;

  ORDER_ID PRODUCT_ID     AMOUNT CUR ORDER_PRODUCT AMOUNT_CURRENCY CURRENCY_AMOUNT
ORDER_CURRENCY
---------- ---------- ---------- --- ------------- --------------- ---------------
```

```
-----------
         1          1       9.99 USD TRUE          TRUE           FALSE          TRUE
         2          2    1234.56 GBP TRUE          TRUE           FALSE          TRUE
         3          3    -999999 JPY TRUE          TRUE           FALSE          TRUE
         4          4    3141592 XXX TRUE          TRUE           FALSE          TRUE
         5          5    2718281 123 TRUE          TRUE           TRUE           TRUE
```

In the example above:

- `ORDER_PRODUCT` is `TRUE` for all rows because the values for `ORDER_ID` and `PRODUCT_ID` can be converted to the corresponding column types in the domain (`NUMBER` and `CHAR`).
- `AMOUNT_CURRENCY` is `TRUE` for all rows because the table columns match the domain columns.
- `CURRENCY_AMOUNT` is `FALSE` for the first four rows because the values for the first argument, `CURRENCY_CODE` are all letters. These cannot be converted to the type of the first column in the domain (`NUMBER`), leading to a type error. The fifth row is `TRUE` because the amount (`2718281`) can be converted to `CHAR`.
- `ORDER_CURRENCY` is `TRUE` for all rows because the types for `ORDER_ID` and `CURRENCY_CODE` match the corresponding domain column types (`NUMBER` and `CHAR`).

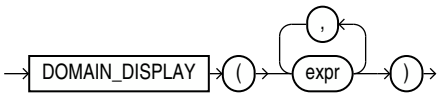
Example 6

The following statement tries to validate the string `"raises an error"` against the non-existent domain `NOT_A_DOMAIN`. This raises an exception:

```
SELECT DOMAIN_CHECK_TYPE(not_a_domain, 'raises an error');
ORA-11504: The domain specified does not exist or the user does not have privileges on
the domain for the operation.
```

DOMAIN_DISPLAY

Syntax



Purpose

`DOMAIN_DISPLAY` returns *`expr`* formatted according to the domain's display expression. This returns `NULL` if the arguments are not associated with a domain or the domain has no display expression.

When calling `DOMAIN_DISPLAY` for multicolumn domains, all values of *`expr`* should be from the same domain. It returns `NULL` if the number of *`expr`* arguments are different from the number of domain columns or they are in a different order in the domain.

To get the display expression for a non-domain value, cast *`expr`* to the domain type. This is only possible for single column domains.

> **See Also:**
>
> - Domain Functions

Examples

The following example creates the domain `DAY_OF_WEEK` and associates it with the column `CALENDAR_DATES.DAY_OF_WEEK_ABBR`. Passing this column to `DOMAIN_DISPLAY` returns it with the first letter of each word capitalized and all other letters in lowercase. `DOMAIN_DISPLAY` also returns this format when casting a string to the domain.

All other calls to `DOMAIN_DISPLAY` pass non-domain values, so return `NULL`.

```
CREATE DOMAIN day_of_week AS CHAR(3 CHAR)
  DISPLAY INITCAP(day_of_week);


CREATE TABLE calendar_dates (
  calendar_date DATE,
  day_of_week_abbr day_of_week
);


INSERT INTO calendar_dates
VALUES(DATE'2023-05-01', 'MON'),
      (DATE'2023-05-02', 'tue'),
      (DATE'2023-05-05', 'fRI');


SELECT day_of_week_abbr,
       DOMAIN_DISPLAY(day_of_week_abbr) domain_column,
       DOMAIN_DISPLAY(calendar_date) nondomain_column,
       DOMAIN_DISPLAY(CAST('MON' AS day_of_week)) domain_value,
       DOMAIN_DISPLAY('MON') nondomain_value
  FROM calendar_dates;

DAY_OF_WEEK_ABBR DOMAIN_COLUMN NONDOMAIN_COLUMN DOMAIN_VALUE NONDOMAIN_VALUE
---------------- ------------- ---------------- ------------ ---------------
MON              Mon           <null>           Mon          <null>
tue              Tue           <null>           Mon          <null>
fRI              Fri           <null>           Mon          <null>
```

The following example creates the multicolumn domain `CURRENCY` with a display expression. The columns `AMOUNT` and `CURRENCY_CODE` in `ORDER_ITEMS` are associated with this domain.

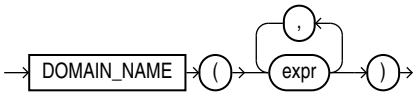
In the query, only the domain_cols expression formats the columns according to the domain expression. All other calls to `DOMAIN_DISPLAY` have a mismatch between its arguments and the domain columns so return `NULL`:

```
CREATE DOMAIN currency AS (
  amount        AS NUMBER(10, 2)
  currency_code AS CHAR(3 CHAR)
)
DISPLAY CASE currency_code
  WHEN 'USD' THEN '$'
  WHEN 'GBP' THEN '£'
```

```
  WHEN 'EUR' THEN '€'
  WHEN 'JPY' THEN '¥'
END || TO_CHAR(amount, '999,999,999.00');


CREATE TABLE order_items (
  order_id      INTEGER,
  product_id    INTEGER,
  amount        NUMBER(10, 2),
  currency_code CHAR(3 CHAR),
  DOMAIN currency(amount, currency_code)
);


INSERT INTO order_items
VALUES (1, 1,    9.99, 'USD'),
       (2, 2, 1234.56, 'GBP'),
       (3, 3, 4321,    'EUR'),
       (4, 4, 3141592, 'JPY');


SELECT order_id,
       product_id,
       DOMAIN_DISPLAY(amount, currency_code) domain_cols,
       DOMAIN_DISPLAY(currency_code, amount) domain_cols_wrong_order,
       DOMAIN_DISPLAY(order_id, product_id) nondomain_cols,
       DOMAIN_DISPLAY(amount) domain_cols_subset
  FROM order_items;

  ORDER_ID PRODUCT_ID DOMAIN_COLS      DOMAIN_COLS_WRONG_ORDER NONDOMAIN_COLS
DOMAIN_COLS_SUBSET
---------- ---------- ---------------- ----------------------- --------------
------------------
         1          1 $           9.99 <null>                  <null>
<null>
         2          2 £       1,234.56 <null>                  <null>
<null>
         3          3 €       4,321.00 <null>                  <null>
<null>
         4          4 ¥   3,141,592.00 <null>                  <null>         <null>
```

DOMAIN_NAME

Syntax



Purpose

`DOMAIN_NAME` returns the fully qualified name of the domain associated with *`expr`*. This returns `NULL` if expr is associated with a domain.

When calling `DOMAIN_NAME` for multicolumn domains, all values of *`expr`* should be from the same domain. It returns `NULL` if the number of *`expr`* arguments are different from the number of domain columns or they are in a different order in the domain.

> **See Also:**
>
> - Domain Functions

Examples

The following example creates the domain `DAY_OF_WEEK` in the schema `HR` and associates it with the column `HR.CALENDAR_DATES.DAY_OF_WEEK_ABBR`. Passing this column to `DOMAIN_NAME` returns the fully qualified name of the domain.

The query casts the string "`MON`" to `DAY_OF_WEEK` to get the domain name.

All other calls to `DOMAIN_NAME` return `NULL`.

```
CREATE DOMAIN hr.day_of_week AS CHAR(3 CHAR);


CREATE TABLE hr.calendar_dates (
  calendar_date    DATE,
  day_of_week_abbr hr.day_of_week
);


INSERT INTO hr.calendar_dates
VALUES(DATE'2023-05-01', 'MON');


SELECT day_of_week_abbr,
       DOMAIN_NAME(day_of_week_abbr) domain_column,
       DOMAIN_NAME(calendar_date) nondomain_column,
       DOMAIN_NAME(CAST('MON' AS hr.day_of_week)) domain_value,
       DOMAIN_NAME('MON') nondomain_value
  FROM hr.calendar_dates;

DAY_OF_WEEK_ABBR DOMAIN_COLUMN  NONDOMAIN_COLUMN DOMAIN_VALUE   NONDOMAIN_VALUE
---------------- -------------- ---------------- -------------- ---------------
MON              HR.DAY_OF_WEEK <null>           HR.DAY_OF_WEEK <null>
```

The following example creates the multicolumn domain `CURRENCY` in the schema `CO`. The columns `AMOUNT` and `CURRENCY_CODE` in `CO.ORDER_ITEMS` are associated with this domain.

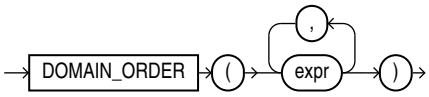
In the query, the arguments for `DOMAIN_NAME` only match the domain definition for the domain_cols expression. This returns the fully qualified name of the domain. All other calls to `DOMAIN_NAME` have a mismatch between its arguments and the domain columns so return `NULL`:

```
CREATE DOMAIN co.currency AS (
  amount        AS NUMBER(10, 2)
  currency_code AS CHAR(3 CHAR)
);


CREATE TABLE co.order_items (
  order_id      INTEGER,
  product_id    INTEGER,
  amount        NUMBER(10, 2),
  currency_code CHAR(3 CHAR),
  DOMAIN co.currency(amount, currency_code)
);
```

```
INSERT INTO co.order_items
VALUES (1, 1, 9.99, 'USD');


SELECT order_id,
       product_id,
       DOMAIN_NAME(amount, currency_code) domain_cols,
       DOMAIN_NAME(currency_code, amount) domain_cols_wrong_order,
       DOMAIN_NAME(order_id, product_id) nondomain_cols,
       DOMAIN_NAME(amount) domain_cols_subset
  FROM co.order_items
  ORDER BY domain_cols;

  ORDER_ID PRODUCT_ID DOMAIN_COLS     DOMAIN_COLS_WRONG_ORDER   NONDOMAIN_COLS
DOMAIN_COLS_SUBSET
---------- ---------- --------------- ------------------------ ---------------
--------------------
         1          1 CO.CURRENCY     <null>                   <null>          <null>
```

DOMAIN_ORDER

Syntax



Purpose

`DOMAIN_ORDER` returns expr formatted according to the domain's order expression. This returns `NULL` if the arguments are not associated with a domain or the domain has no order expression.

When calling `DOMAIN_ORDER` for multicolumn domains, all values of *`expr`* should be from the same domain. It returns `NULL` if the number expr arguments are different from the number of domain columns or they are in a different order in the domain.

To get the display expression for a non-domain value, cast *`expr`* to the domain type. This is only possible for single column domains.

> **See Also:**
>
> - Domain Functions

Examples

The following example creates the domain `DAY_OF_WEEK` and associates it with the column `CALENDAR_DATES.DAY_OF_WEEK_ABBR`. Passing this column to `DOMAIN_ORDER` returns the result of the `ORDER` expression (`MON = 0`, `TUE = 1`, etc.). Using this in the `ORDER BY` returns the rows sorted by their position in the week instead of alphabetically.

The query casts the string `"MON"` to `DAY_OF_WEEK` to get its sort value.

All other calls to DOMAIN_ORDER pass non-domain values, so return NULL.

```
CREATE DOMAIN day_of_week AS CHAR(3 CHAR)
  ORDER CASE UPPER(day_of_week)
     WHEN 'MON' THEN 0
     WHEN 'TUE' THEN 1
     WHEN 'WED' THEN 2
     WHEN 'THU' THEN 3
     WHEN 'FRI' THEN 4
     WHEN 'SAT' THEN 5
     WHEN 'SUN' THEN 6
     ELSE 7
  END;


CREATE TABLE calendar_dates (
  calendar_date    DATE,
  day_of_week_abbr day_of_week
);


INSERT INTO calendar_dates
VALUES(DATE'2023-05-01', 'MON'),
      (DATE'2023-05-02', 'TUE'),
      (DATE'2023-05-05', 'FRI'),
      (DATE'2023-05-08', 'mon');


SELECT day_of_week_abbr,
       DOMAIN_ORDER(day_of_week_abbr) domain_column,
       DOMAIN_ORDER(calendar_date) nondomain_column,
       DOMAIN_ORDER(CAST('MON' AS day_of_week)) domain_value,
       DOMAIN_ORDER('MON') nondomain_value
  FROM calendar_dates
  ORDER BY DOMAIN_ORDER(day_of_week_abbr);

DAY_OF_WEEK_ABBR DOMAIN_COLUMN NONDOMAIN_COLUMN DOMAIN_VALUE NONDOMAIN_VALUE
---------------- ------------- ---------------- ------------ ---------------
MON                          0 <null>                      0 <null>
mon                          0 <null>                      0 <null>
TUE                          1 <null>                      0 <null>
FRI                          4 <null>                      0 <null>
```

The following example creates the multicolumn domain CURRENCY with an order expression. This sorts the values by currency then value. The columns AMOUNT and CURRENCY_CODE in ORDER_ITEMS are associated with this domain.

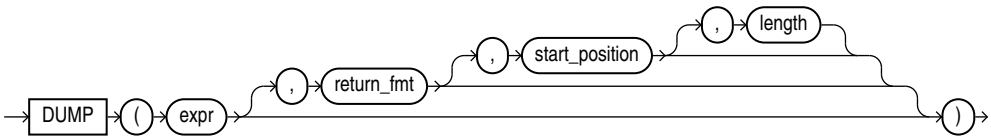
In the query, only the domain_cols expression formats the columns according to the order expression. All other calls to DOMAIN_ORDER have a mismatch between its arguments and the domain columns so return NULL:

```
CREATE DOMAIN currency AS (
  amount        AS NUMBER(10, 2)
  currency_code AS CHAR(3 CHAR)
)
ORDER currency_code || TO_CHAR(amount, '999999999.00');


CREATE TABLE order_items (
  order_id      INTEGER,
```

```
  product_id    INTEGER,
  amount        NUMBER(10, 2),
  currency_code CHAR(3 CHAR),
  DOMAIN currency(amount, currency_code)
);


INSERT INTO order_items
VALUES (1, 1,    9.99, 'USD'),
       (2, 2, 1234.56, 'USD'),
       (3, 3, 4321,    'EUR'),
       (4, 4, 3141592, 'JPY'),
       (5, 5, 2718281, 'JPY');


SELECT order_id,
       product_id,
       DOMAIN_ORDER(amount, currency_code) domain_cols,
       DOMAIN_ORDER(currency_code, amount) domain_cols_wrong_order,
       DOMAIN_ORDER(order_id, product_id) nondomain_cols,
       DOMAIN_ORDER(amount) domain_cols_subset
  FROM order_items
  ORDER BY domain_cols;

  ORDER_ID PRODUCT_ID DOMAIN_COLS      DOMAIN_COLS_WRONG_ORDER NONDOMAIN_COLS
DOMAIN_COLS_SUBSET
---------- ---------- ---------------- ----------------------- --------------
------------------
         3          3 EUR      4321.00 <null>                  <null>
<null>
         5          5 JPY   2718281.00 <null>                  <null>
<null>
         4          4 JPY   3141592.00 <null>                  <null>
<null>
         1          1 USD         9.99 <null>                  <null>
<null>
         2          2 USD      1234.56 <null>                  <null>         <null>
```

DUMP

Syntax



Purpose

`DUMP` returns a `VARCHAR2` value containing the data type code, length in bytes, and internal representation of *`expr`*. The returned result is always in the database character set. For the data type corresponding to each code, see Table 2-1.

The argument *`return_fmt`* specifies the format of the return value and can have any of the following values:

- 8 returns result in octal notation.

- 10 returns result in decimal notation.
- 16 returns result in hexadecimal notation.
- 17 returns each byte printed as a character if and only if it can be interpreted as a printable character in the character set of the compiler—typically ASCII or EBCDIC. Some ASCII control characters may be printed in the form ^X as well. Otherwise the character is printed in hexadecimal notation. All NLS parameters are ignored. Do not depend on any particular output format for `DUMP` with *`return_fmt`* 17.

By default, the return value contains no character set information. To retrieve the character set name of *`expr`*, add 1000 to any of the preceding format values. For example, a *`return_fmt`* of 1008 returns the result in octal and provides the character set name of *`expr`*.

The arguments *`start_position`* and *`length`* combine to determine which portion of the internal representation to return. The default is to return the entire internal representation in decimal notation.

If *`expr`* is null, then this function returns `NULL`.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - Data Type Comparison Rules for more information
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `DUMP`

Examples

The following examples show how to extract dump information from a string expression and a column:

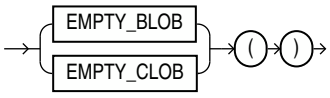
```
SELECT DUMP('abc', 1016)
  FROM DUAL;

DUMP('ABC',1016)
----------------------------------------
Typ=96 Len=3 CharacterSet=WE8DEC: 61,62,63

SELECT DUMP(last_name, 8, 3, 2) "OCTAL"
  FROM employees
  WHERE last_name = 'Hunold'
  ORDER BY employee_id;

OCTAL
----------------------------------------------------------------------
Typ=1 Len=6: 156,157

SELECT DUMP(last_name, 10, 3, 2) "ASCII"
  FROM employees
  WHERE last_name = 'Hunold'
  ORDER BY employee_id;

ASCII
```

```
-------------------------------------------------------------------
Typ=1 Len=6: 110,111
```

EMPTY_BLOB, EMPTY_CLOB

Syntax

***empty_LOB*::=**



Purpose

`EMPTY_BLOB` and `EMPTY_CLOB` return an empty LOB locator that can be used to initialize a LOB variable or, in an `INSERT` or `UPDATE` statement, to initialize a LOB column or attribute to `EMPTY`. `EMPTY` means that the LOB is initialized, but not populated with data.

> **Note:**
>
> An empty LOB is not the same as a null LOB, and an empty `CLOB` is not the same as a LOB containing a string of 0 length. For more information, see *Oracle Database SecureFiles and Large Objects Developer's Guide*.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `EMPTY_CLOB`

Restriction on LOB Locators

You cannot use the locator returned from this function as a parameter to the `DBMS_LOB` package or the OCI.

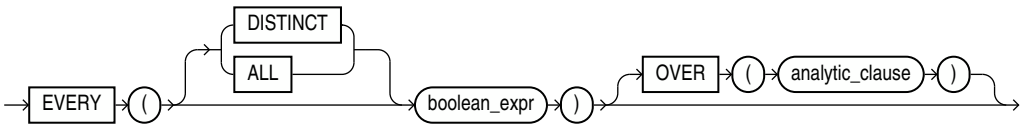
Examples

The following example initializes the `ad_photo` column of the sample `pm.print_media` table to `EMPTY`:

```
UPDATE print_media
  SET ad_photo = EMPTY_BLOB();
```

EVERY

Syntax



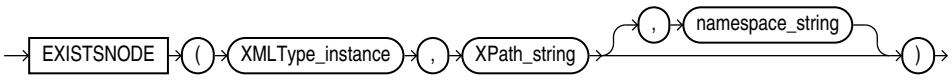
Purpose

`EVERY` returns `'TRUE'` if the *`boolean_expr`* evaluates to true for every row that qualifies. Otherwise it returns `'FALSE'`. You can use it as an aggregate or analytic function. It is the same as the function `BOOLEAN_AND_AGG`.

EXISTSNODE

> **Note:**
>
> The `EXISTSNODE` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use the `XMLEXISTS` function instead. See XMLEXISTS for more information.

Syntax



Purpose

`EXISTSNODE` determines whether traversal of an XML document using a specified path results in any nodes. It takes as arguments the `XMLType` instance containing an XML document and a `VARCHAR2` XPath string designating a path. The optional *`namespace_string`* must resolve to a `VARCHAR2` value that specifies a default mapping or namespace mapping for prefixes, which Oracle Database uses when evaluating the XPath expression(s).

The *`namespace_string`* argument defaults to the namespace of the root element. If you refer to any subelement in *`Xpath_string`*, then you must specify *`namespace_string`*, and you must specify the "who" prefix in both of these arguments.

> **See Also:**
>
> Using XML in SQL Statements for examples that specify *`namespace_string`* and use the "who" prefix.

The return value is NUMBER:

- 0 if no nodes remain after applying the XPath traversal on the document
- 1 if any nodes remain

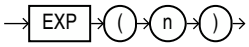
Examples

The following example tests for the existence of the /Warehouse/Dock node in the XML path of the warehouse_spec column of the sample table oe.warehouses:

```
SELECT warehouse_id, warehouse_name
  FROM warehouses
  WHERE EXISTSNODE(warehouse_spec, '/Warehouse/Docks') = 1
  ORDER BY warehouse_id;

WAREHOUSE_ID WAREHOUSE_NAME
------------ ----------------------------------
           1 Southlake, Texas
           2 San Francisco
           4 Seattle, Washington
```

EXP

Syntax



Purpose

EXP returns e raised to the *n*th power, where e = 2.71828183... . The function returns a value of the same type as the argument.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is BINARY_FLOAT, then the function returns BINARY_DOUBLE. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

Examples

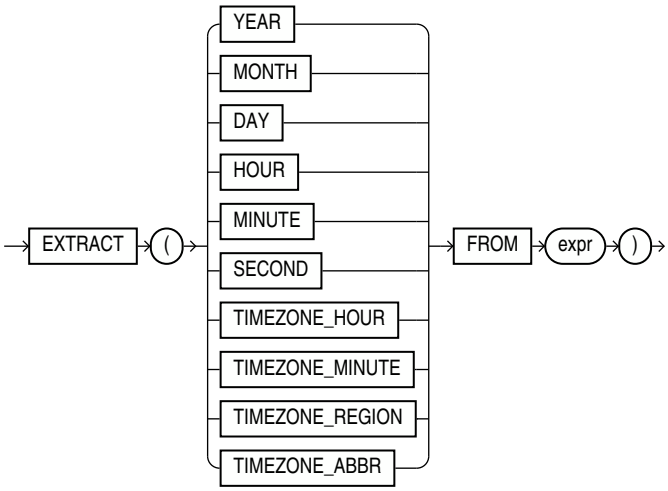
The following example returns e to the 4th power:

```
SELECT EXP(4) "e to the 4th power"
  FROM DUAL;

e to the 4th power
------------------
          54.59815
```

EXTRACT (datetime)

Syntax

***extract_datetime*::=**



Purpose

`EXTRACT` extracts and returns the value of a specified datetime field from a datetime or interval expression. The *`expr`* can be any expression that evaluates to a datetime or interval data type compatible with the requested field:

- If `YEAR` or `MONTH` is requested, then *`expr`* must evaluate to an expression of data type `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, or `INTERVAL YEAR TO MONTH`.
- If `DAY` is requested, then *`expr`* must evaluate to an expression of data type `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, or `INTERVAL DAY TO SECOND`.
- If `HOUR`, `MINUTE`, or `SECOND` is requested, then *`expr`* must evaluate to an expression of data type `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, or `INTERVAL DAY TO SECOND`. `DATE` is not valid here, because Oracle Database treats it as ANSI `DATE` data type, which has no time fields.
- If `TIMEZONE_HOUR`, `TIMEZONE_MINUTE`, `TIMEZONE_ABBR`, `TIMEZONE_REGION`, or `TIMEZONE_OFFSET` is requested, then *`expr`* must evaluate to an expression of data type `TIMESTAMP WITH TIME ZONE` or `TIMESTAMP WITH LOCAL TIME ZONE`.

`EXTRACT` interprets *`expr`* as an ANSI datetime data type. For example, `EXTRACT` treats `DATE` not as legacy Oracle `DATE` but as ANSI `DATE`, without time elements. Therefore, you can extract only `YEAR`, `MONTH`, and `DAY` from a `DATE` value. Likewise, you can extract `TIMEZONE_HOUR` and `TIMEZONE_MINUTE` only from the `TIMESTAMP WITH TIME ZONE` data type.

When you specify `TIMEZONE_REGION` or `TIMEZONE_ABBR` (abbreviation), the value returned is a `VARCHAR2` string containing the appropriate time zone region name or abbreviation. When you specify any of the other datetime fields, the value returned is an integer value of `NUMBER` data type representing the datetime value in the Gregorian calendar. When extracting from a

datetime with a time zone value, the value returned is in UTC. For a listing of time zone region names and their corresponding abbreviations, query the `V$TIMEZONE_NAMES` dynamic performance view.

This function can be very useful for manipulating datetime field values in very large tables, as shown in the first example below.

> **Note:**
>
> Time zone region names are needed by the daylight saving feature. These names are stored in two types of time zone files: one large and one small. One of these files is the default file, depending on your environment and the release of Oracle Database you are using. For more information regarding time zone files and names, see *Oracle Database Globalization Support Guide*.

Some combinations of datetime field and datetime or interval value expression result in ambiguity. In these cases, Oracle Database returns `UNKNOWN` (see the examples that follow for additional information).

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for a complete listing of the time zone region names in both files
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `EXTRACT`
> - Datetime/Interval Arithmetic for a description of *`datetime_value_expr`* and *`interval_value_expr`*
> - *Oracle Database Reference* for information on the dynamic performance views

Examples

The following example returns from the `oe.orders` table the number of orders placed in each month:

```
SELECT EXTRACT(month FROM order_date) "Month", COUNT(order_date) "No. of Orders"
  FROM orders
  GROUP BY EXTRACT(month FROM order_date)
  ORDER BY "No. of Orders" DESC, "Month";

     Month No. of Orders
---------- -------------
        11            15
         6            14
         7            14
         3            11
         5            10
         2             9
         9             9
         8             7
        10             6
```

```
         1             5
        12             4
         4             1

12 rows selected.
```

The following example returns the year 1998.

```
SELECT EXTRACT(YEAR FROM DATE '1998-03-07')
  FROM DUAL;

EXTRACT(YEARFROMDATE'1998-03-07')
---------------------------------
                             1998
```

The following example selects from the sample table `hr.employees` all employees who were hired after 2007:

```
SELECT last_name, employee_id, hire_date
  FROM employees
  WHERE EXTRACT(YEAR FROM hire_date) > 2007
  ORDER BY hire_date;

LAST_NAME                 EMPLOYEE_ID HIRE_DATE
------------------------- ----------- ---------
Johnson                           179 04-JAN-08
Grant                             199 13-JAN-08
Marvins                           164 24-JAN-08
. . .
```

The following example results in ambiguity, so Oracle returns `UNKNOWN`:

```
SELECT EXTRACT(TIMEZONE_REGION FROM TIMESTAMP '1999-01-01 10:00:00 -08:00')
  FROM DUAL;

EXTRACT(TIMEZONE_REGIONFROMTIMESTAMP'1999-01-0110:00:00-08:00')
---------------------------------------------------------------
UNKNOWN
```

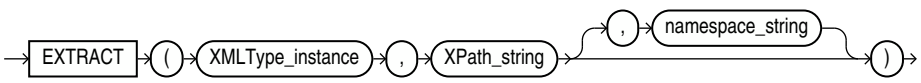
The ambiguity arises because the time zone numerical offset is provided in the expression, and that numerical offset may map to more than one time zone region name.

EXTRACT (XML)

> **Note:**
>
> The `EXTRACT` (XML) function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use the `XMLQUERY` function instead. See XMLQUERY for more information.

Syntax

***extract_xml*::=**



Purpose

`EXTRACT` (XML) is similar to the `EXISTSNODE` function. It applies a `VARCHAR2` XPath string and returns an `XMLType` instance containing an XML fragment. You can specify an absolute *`XPath_string`* with an initial slash or a relative *`XPath_string`* by omitting the initial slash. If you omit the initial slash, then the context of the relative path defaults to the root node. The optional *`namespace_string`* is required if the XML you are handling uses a namespace prefix. This argument must resolve to a `VARCHAR2` value that specifies a default mapping or namespace mapping for prefixes, which Oracle Database uses when evaluating the XPath expression(s).

Examples

The following example extracts the value of the `/Warehouse/Dock` node of the XML path of the `warehouse_spec` column in the sample table `oe.warehouses`:

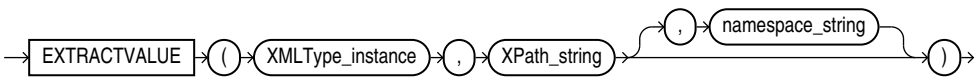
```
SELECT warehouse_name,
       EXTRACT(warehouse_spec, '/Warehouse/Docks') "Number of Docks"
  FROM warehouses
  WHERE warehouse_spec IS NOT NULL
  ORDER BY warehouse_name;

WAREHOUSE_NAME            Number of Docks
------------------------- -------------------------
New Jersey
San Francisco             <Docks>1</Docks>
Seattle, Washington       <Docks>3</Docks>
Southlake, Texas          <Docks>2</Docks>
```

Compare this example with the example for EXTRACTVALUE , which returns the scalar value of the XML fragment.

EXTRACTVALUE

> **Note:**
>
> The `EXTRACTVALUE` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use the `XMLTABLE` function, or the `XMLCAST` and `XMLQUERY` functions instead. See XMLTABLE , XMLCAST, and XMLQUERY for more information.

Syntax



The `EXTRACTVALUE` function takes as arguments an `XMLType` instance and an XPath expression and returns a scalar value of the resultant node. The result must be a single node and be either a text node, attribute, or element. If the result is an element, then the element must have a single text node as its child, and it is this value that the function returns. You can specify an absolute *`XPath_string`* with an initial slash or a relative *`XPath_string`* by omitting the initial slash. If you omit the initial slash, the context of the relative path defaults to the root node.

If the specified XPath points to a node with more than one child, or if the node pointed to has a non-text node child, then Oracle returns an error. The optional *namespace_string* must resolve to a VARCHAR2 value that specifies a default mapping or namespace mapping for prefixes, which Oracle uses when evaluating the XPath expression(s).

For documents based on XML schemas, if Oracle can infer the type of the return value, then a scalar value of the appropriate type is returned. Otherwise, the result is of type VARCHAR2. For documents that are not based on XML schemas, the return type is always VARCHAR2.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of EXTRACTVALUE

Examples

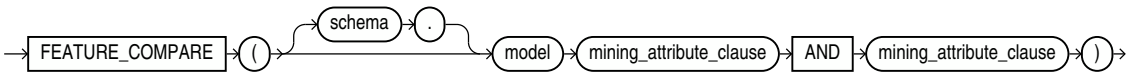
The following example takes as input the same arguments as the example for EXTRACT (XML) . Instead of returning an XML fragment, as does the EXTRACT function, it returns the scalar value of the XML fragment:

```
SELECT warehouse_name, EXTRACTVALUE(e.warehouse_spec, '/Warehouse/Docks') "Docks"
  FROM warehouses e
  WHERE warehouse_spec IS NOT NULL
  ORDER BY warehouse_name;

WAREHOUSE_NAME       Docks
-------------------- ------------
New Jersey
San Francisco        1
Seattle, Washington  3
Southlake, Texas     2
```

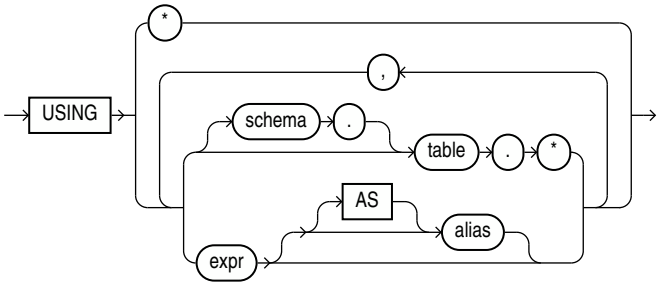
FEATURE_COMPARE

Syntax

***feature_compare*::=**



***mining_attribute_clause*::=**



Purpose

The `FEATURE_COMPARE` function uses a Feature Extraction model to compare two different documents, including short ones such as keyword phrases or two attribute lists, for similarity or dissimilarity. The `FEATURE_COMPARE` function can be used with Feature Extraction algorithms such as Singular Value Decomposition (SVD), Principal Component Analysis PCA), Non-Negative Matrix Factorization (NMF), and Explicit Semantic Analysis (ESA). This function is applicable not only to documents, but also to numeric and categorical data.

The input to the `FEATURE_COMPARE` function is a single feature model built using the Feature Extraction algorithms of Oracle Machine Learning for SQL, such as NMF, SVD, and ESA. The double `USING` clause provides a mechanism to compare two different documents or constant keyword phrases, or any combination of the two, for similarity or dissimilarity using the extracted features in the model.

The syntax of the `FEATURE_COMPARE` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

The *`mining_attribute_clause`* identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *`mining_attribute_clause`* behaves as described for the `PREDICTION` function. See mining_attribute_clause.

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring
> - *Oracle Machine Learning for SQL Concepts* for information about clustering

> **Note:**
>
> The following examples are excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Examples

An ESA model is built against a 2005 Wiki dataset rendering over 200,000 features. The documents are mined as text and the document titles are considered as the Feature IDs.

The examples show the `FEATURE_COMPARE` function with the ESA algorithm, which compares a similar set of texts and then a dissimilar set of texts.

Similar texts

```
SELECT 1-FEATURE_COMPARE(esa_wiki_mod USING 'There are several PGA tour golfers from South
Africa' text AND USING 'Nick Price won the 2002 Mastercard Colonial Open' text) similarity
FROM DUAL;

SIMILARITY
----------
      .258
```

The output metric shows the results of a distance calculation. Therefore, a smaller number represents more similar texts. So 1 minus the distance in the queries represents a document similarity metric.

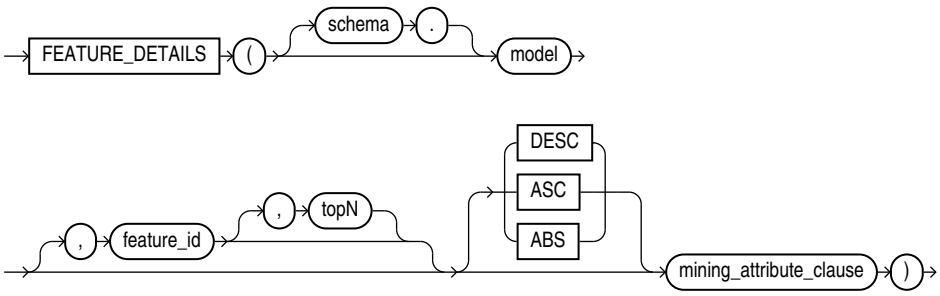
Dissimilar texts

```
SELECT 1-FEATURE_COMPARE(esa_wiki_mod USING 'There are several PGA tour golfers from South
Africa' text AND USING 'John Elway played quarterback for the Denver Broncos' text)
similarity FROM DUAL;

SIMILARITY
----------
      .007
```

FEATURE_DETAILS

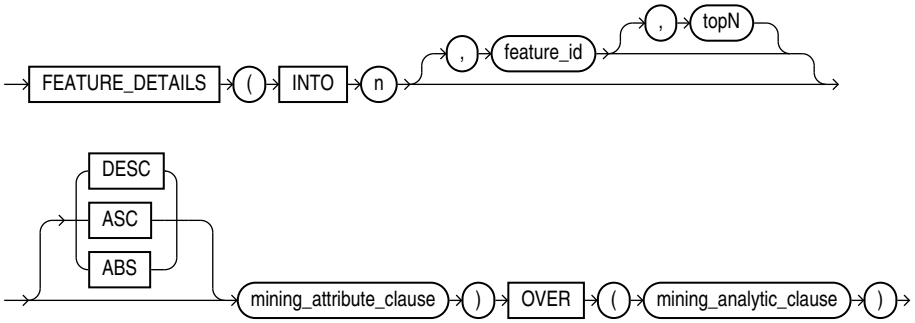
Syntax

***feature_details*::=**

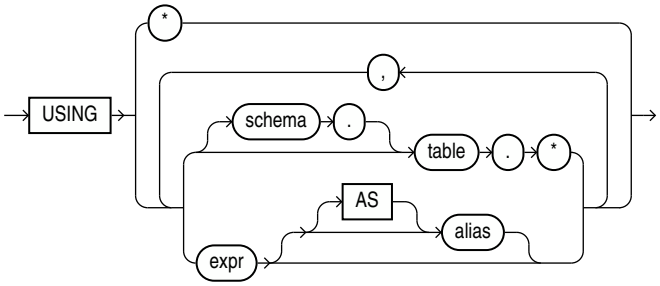


Analytic Syntax

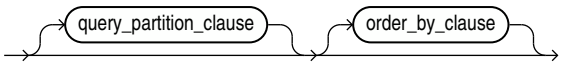
***feature_details_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`FEATURE_DETAILS` returns feature details for each row in the selection. The return value is an XML string that describes the attributes of the highest value feature or the specified *feature_id*.

topN

If you specify a value for *topN*, the function returns the *N* attributes that most influence the feature value. If you do not specify *topN*, the function returns the 5 most influential attributes.

DESC, ASC, or ABS

The returned attributes are ordered by weight. The weight of an attribute expresses its positive or negative impact on the value of the feature. A positive weight indicates a higher feature value. A negative weight indicates a lower feature value.

By default, `FEATURE_DETAILS` returns the attributes with the highest positive weight (`DESC`). If you specify `ASC`, the attributes with the highest negative weight are returned. If you specify `ABS`, the attributes with the greatest weight, whether negative or positive, are returned. The results are ordered by absolute value from highest to lowest. Attributes with a zero weight are not included in the output.

Syntax Choice

`FEATURE_DETAILS` can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a feature extraction model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include `INTO` *`n`*, where *`n`* is the number of features to extract, and *`mining_analytic_clause`*, which specifies if the data should be partitioned for multiple model builds. The *`mining_analytic_clause`* supports a *`query_partition_clause`* and an *`order_by_clause`*. (See "*analytic_clause*::=".)

The syntax of the `FEATURE_DETAILS` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

`mining_attribute_clause` identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *`mining_attribute_clause`* behaves as described for the `PREDICTION` function. (See "*mining_attribute_clause*::=".)

See Also:

- *Oracle Machine Learning for SQL User's Guide* for information about scoring.
- *Oracle Machine Learning for SQL Concepts* for information about feature extraction.

Note:

The following examples are excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

This example uses the feature extraction model `nmf_sh_sample` to score the data. The query returns the three features that best represent customer 100002 and the attributes that most affect those features.

```
SELECT S.feature_id fid, value val,
       FEATURE_DETAILS(nmf_sh_sample, S.feature_id, 5 using T.*) det
   FROM
     (SELECT v.*, FEATURE_SET(nmf_sh_sample, 3 USING *) fset
        FROM mining_data_apply_v v
        WHERE cust_id = 100002) T,
   TABLE(T.fset) S
ORDER BY 2 DESC;

 FID    VAL  DET
---- ------  --------------------------------------------------------------------------------
   5  3.492  <Details algorithm="Non-Negative Matrix Factorization" feature="5">
             <Attribute name="BULK_PACK_DISKETTES" actualValue="1" weight=".077" rank="1"/>
             <Attribute name="OCCUPATION" actualValue="Prof." weight=".062" rank="2"/>
             <Attribute name="BOOKKEEPING_APPLICATION" actualValue="1" weight=".001" rank="3"/>
             <Attribute name="OS_DOC_SET_KANJI" actualValue="0" weight="0" rank="4"/>
             <Attribute name="YRS_RESIDENCE" actualValue="4" weight="0" rank="5"/>
             </Details>
   3  1.928  <Details algorithm="Non-Negative Matrix Factorization" feature="3">
             <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".239" rank="1"/>
             <Attribute name="CUST_INCOME_LEVEL" actualValue="L: 300\,000 and above"
              weight=".051" rank="2"/>
             <Attribute name="FLAT_PANEL_MONITOR" actualValue="1" weight=".02" rank="3"/>
             <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".006" rank="4"/>
             <Attribute name="AGE" actualValue="41" weight=".004" rank="5"/>
             </Details>
   8   .816  <Details algorithm="Non-Negative Matrix Factorization" feature="8">
             <Attribute name="EDUCATION" actualValue="Bach." weight=".211" rank="1"/>
             <Attribute name="CUST_MARITAL_STATUS" actualValue="NeverM" weight=".143" rank="2"/>
             <Attribute name="FLAT_PANEL_MONITOR" actualValue="1" weight=".137" rank="3"/>
             <Attribute name="CUST_GENDER" actualValue="F" weight=".044" rank="4"/>
             <Attribute name="BULK_PACK_DISKETTES" actualValue="1" weight=".032" rank="5"/>
             </Details>
```

Analytic Example

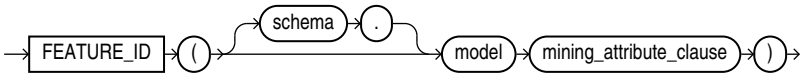
This example dynamically maps customer attributes into six features and returns the feature mapping for customer 100001.

```
SELECT feature_id, value
  FROM (
     SELECT cust_id, feature_set(INTO 6 USING *) OVER () fset
        FROM mining_data_apply_v),
  TABLE (fset)
  WHERE cust_id = 100001
  ORDER BY feature_id;

FEATURE_ID    VALUE
---------- --------
         1    2.670
         2     .000
         3    1.792
         4     .000
         5     .000
         6    3.379
```

FEATURE_ID

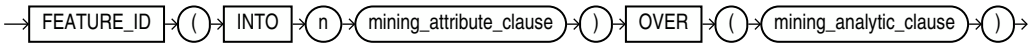
Syntax

***feature_id*::=**

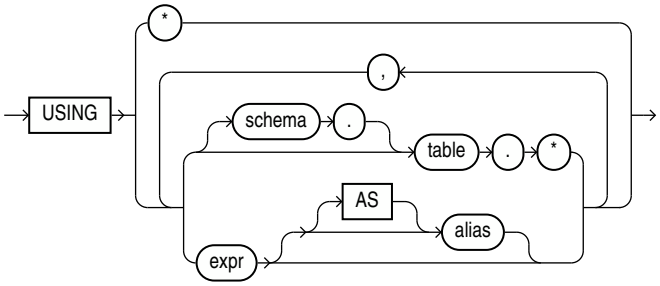


Analytic Syntax

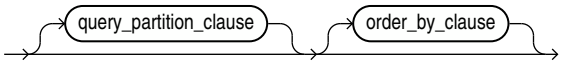
***feature_id_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`FEATURE_ID` returns the identifier of the highest value feature for each row in the selection. The feature identifier is returned as an Oracle `NUMBER`.

Syntax Choice

FEATURE_ID can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a feature extraction model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include INTO *n*, where *n* is the number of features to extract, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)

The syntax of the FEATURE_ID function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the PREDICTION function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring.
> - *Oracle Machine Learning for SQL Concepts* for information about feature extraction.

> **Note:**
>
> The following example is excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

This example lists the features and corresponding count of customers in a data set.

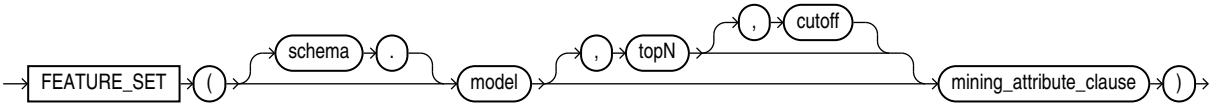
```
SELECT FEATURE_ID(nmf_sh_sample USING *) AS feat, COUNT(*) AS cnt
  FROM nmf_sh_sample_apply_prepared
  GROUP BY FEATURE_ID(nmf_sh_sample USING *)
  ORDER BY cnt DESC, feat DESC;

      FEAT        CNT
---------- ----------
         7       1443
         2         49
         3          6
```

```
                6           1
                1           1
```

FEATURE_SET

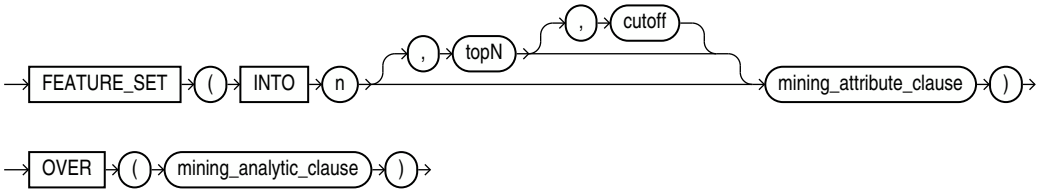
Syntax

***feature_set*::=**

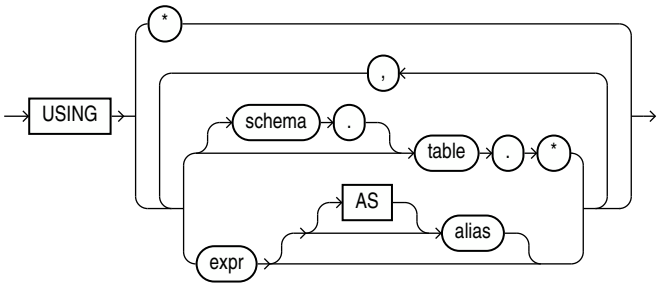


Analytic Syntax

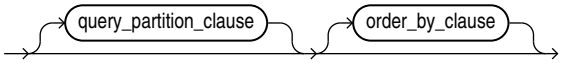
***feature_set_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

FEATURE_SET returns a set of feature ID and feature value pairs for each row in the selection. The return value is a varray of objects with field names FEATURE_ID and VALUE. The data type of both fields is NUMBER.

topN and cutoff

You can specify *topN* and *cutoff* to limit the number of features returned by the function. By default, both *topN* and *cutoff* are null and all features are returned.

- *topN* is the *N* highest value features. If multiple features have the *N*th value, then the function chooses one of them.
- *cutoff* is a value threshold. Only features that are greater than or equal to *cutoff* are returned. To filter by *cutoff* only, specify NULL for *topN*.

To return up to *N* features that are greater than or equal to *cutoff*, specify both *topN* and *cutoff*.

Syntax Choice

FEATURE_SET can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a feature extraction model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include INTO *n*, where *n* is the number of features to extract, and *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)

The syntax of the FEATURE_SET function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the PREDICTION function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring.
> - *Oracle Machine Learning for SQL Concepts* for information about feature extraction.

> **Note:**
>
> The following example is excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

This example lists the top features corresponding to a given customer record and determines the top attributes for each feature (based on coefficient > 0.25).

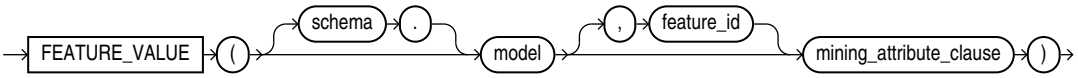
```
WITH
feat_tab AS (
SELECT F.feature_id fid,
       A.attribute_name attr,
       TO_CHAR(A.attribute_value) val,
       A.coefficient coeff
  FROM TABLE(DBMS_DATA_MINING.GET_MODEL_DETAILS_NMF('nmf_sh_sample')) F,
       TABLE(F.attribute_set) A
 WHERE A.coefficient > 0.25
),
feat AS (
SELECT fid,
       CAST(COLLECT(Featattr(attr, val, coeff))
         AS Featattrs) f_attrs
  FROM feat_tab
GROUP BY fid
),
cust_10_features AS (
SELECT T.cust_id, S.feature_id, S.value
  FROM (SELECT cust_id, FEATURE_SET(nmf_sh_sample, 10 USING *) pset
          FROM nmf_sh_sample_apply_prepared
         WHERE cust_id = 100002) T,
       TABLE(T.pset) S
)
SELECT A.value, A.feature_id fid,
       B.attr, B.val, B.coeff
  FROM cust_10_features A,
       (SELECT T.fid, F.*
          FROM feat T,
               TABLE(T.f_attrs) F) B
 WHERE A.feature_id = B.fid
ORDER BY A.value DESC, A.feature_id ASC, coeff DESC, attr ASC, val ASC;

   VALUE  FID ATTR                      VAL                       COEFF
-------- ---- ------------------------ ------------------------ -------
  6.8409    7 YRS_RESIDENCE                                    1.3879
  6.8409    7 BOOKKEEPING_APPLICATION                           .4388
  6.8409    7 CUST_GENDER              M                        .2956
  6.8409    7 COUNTRY_NAME             United States of America  .2848
  6.4975    3 YRS_RESIDENCE                                    1.2668
  6.4975    3 BOOKKEEPING_APPLICATION                           .3465
  6.4975    3 COUNTRY_NAME             United States of America  .2927
  6.4886    2 YRS_RESIDENCE                                    1.3285
  6.4886    2 CUST_GENDER              M                        .2819
  6.4886    2 PRINTER_SUPPLIES                                  .2704
  6.3953    4 YRS_RESIDENCE                                    1.2931
  5.9640    6 YRS_RESIDENCE                                    1.1585
  5.9640    6 HOME_THEATER_PACKAGE                              .2576
```

```
    5.2424     5 YRS_RESIDENCE                                  1.0067
    2.4714     8 YRS_RESIDENCE                                   .3297
    2.3559     1 YRS_RESIDENCE                                   .2768
    2.3559     1 FLAT_PANEL_MONITOR                              .2593
```

FEATURE_VALUE

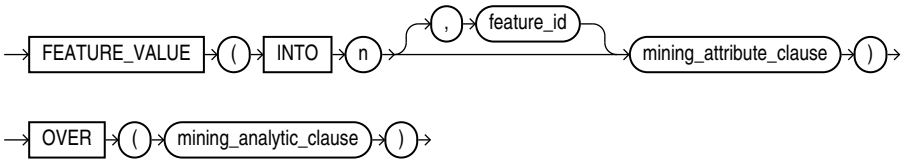
Syntax

***feature_value*::=**

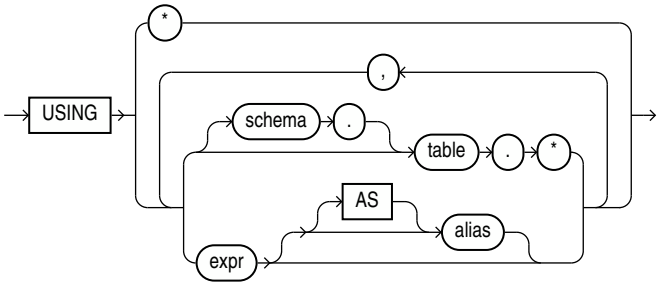


Analytic Syntax

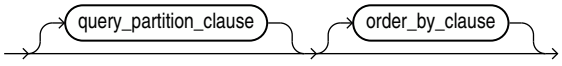
***feature_value_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`FEATURE_VALUE` returns a feature value for each row in the selection. The value refers to the highest value feature or to the specified *`feature_id`*. The feature value is returned as `BINARY_DOUBLE`.

Syntax Choice

`FEATURE_VALUE` can score the data in one of two ways: It can apply a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax** — Use the first syntax to score the data with a pre-defined model. Supply the name of a feature extraction model.
- **Analytic Syntax** — Use the analytic syntax to score the data without a pre-defined model. Include `INTO` *`n`*, where *`n`* is the number of features to extract, and *`mining_analytic_clause`*, which specifies if the data should be partitioned for multiple model builds. The *`mining_analytic_clause`* supports a *`query_partition_clause`* and an *`order_by_clause`*. (See "*analytic_clause*::=".)

The syntax of the `FEATURE_VALUE` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

`mining_attribute_clause` identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, this data is also used for building the transient models. The *`mining_attribute_clause`* behaves as described for the `PREDICTION` function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring.
> - *Oracle Machine Learning for SQL Concepts* for information about feature extraction.

> **Note:**
>
> The following example is excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

The following example lists the customers that correspond to feature 3, ordered by match quality.

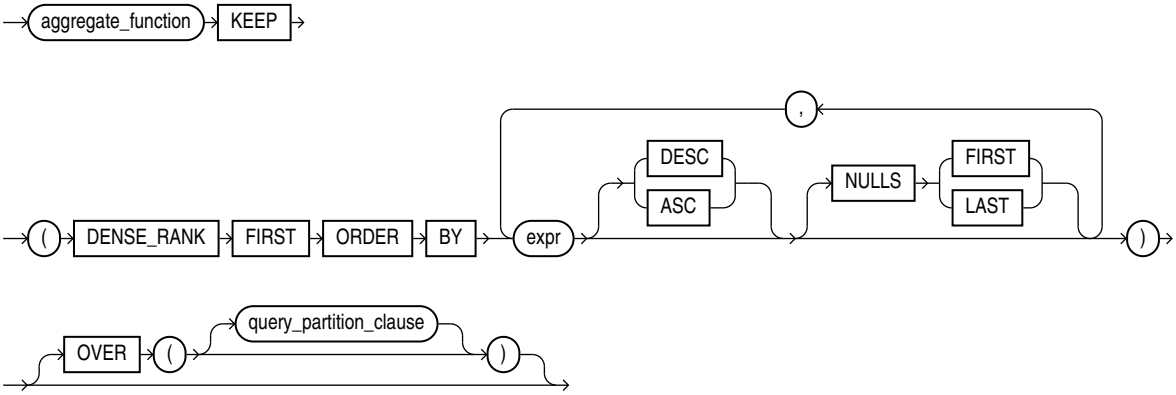
```
SELECT *
  FROM (SELECT cust_id, FEATURE_VALUE(nmf_sh_sample, 3 USING *) match_quality
          FROM nmf_sh_sample_apply_prepared
          ORDER BY match_quality DESC)
```

```
  WHERE ROWNUM < 11;

   CUST_ID MATCH_QUALITY
---------- -------------
    100210    19.4101627
    100962    15.2482251
    101151    14.5685197
    101499    14.4186292
    100363    14.4037396
    100372    14.3335148
    100982    14.1716545
    101039    14.1079914
    100759    14.0913761
    100953    14.0799737
```

FIRST

Syntax

***first*::=**



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions of the `ORDER BY` clause and `OVER` clause

Purpose

`FIRST` and `LAST` are very similar functions. Both are aggregate and analytic functions that operate on a set of values from a set of rows that rank as the `FIRST` or `LAST` with respect to a given sorting specification. If only one row ranks as `FIRST` or `LAST`, then the aggregate operates on the set with only one element.

If you omit the `OVER` clause, then the `FIRST` and `LAST` functions are treated as aggregate functions. You can use these functions as analytic functions by specifying the `OVER` clause. The *`query_partition_clause`* is the only part of the `OVER` clause valid with these functions. If you include the `OVER` clause but omit the *`query_partition_clause`*, then the function is treated as an analytic function, but the window defined for analysis is the entire table.

These functions take as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

When you need a value from the first or last row of a sorted group, but the needed value is not the sort key, the `FIRST` and `LAST` functions eliminate the need for self-joins or views and enable better performance.

- The *`aggregate_function`* argument is any one of the `MIN`, `MAX`, `SUM`, `AVG`, `COUNT`, `VARIANCE`, or `STDDEV` functions. It operates on values from the rows that rank either `FIRST` or `LAST`. If only one row ranks as `FIRST` or `LAST`, then the aggregate operates on a singleton (nonaggregate) set.
- The `KEEP` keyword is for semantic clarity. It qualifies *`aggregate_function`*, indicating that only the `FIRST` or `LAST` values of *`aggregate_function`* will be returned.
- `DENSE_RANK FIRST` or `DENSE_RANK LAST` indicates that Oracle Database will aggregate over only those rows with the minimum (`FIRST`) or the maximum (`LAST`) dense rank (also called olympic rank).

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and LAST

Aggregate Example

The following example returns, within each department of the sample table `hr.employees`, the minimum salary among the employees who make the lowest commission and the maximum salary among the employees who make the highest commission:

```
SELECT department_id,
       MIN(salary) KEEP (DENSE_RANK FIRST ORDER BY commission_pct) "Worst",
       MAX(salary) KEEP (DENSE_RANK LAST ORDER BY commission_pct) "Best"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

DEPARTMENT_ID      Worst       Best
------------- ---------- ----------
           10       4400       4400
           20       6000      13000
           30       2500      11000
           40       6500       6500
           50       2100       8200
           60       4200       9000
           70      10000      10000
           80       6100      14000
           90      17000      24000
          100       6900      12008
          110       8300      12008
                    7000       7000
```

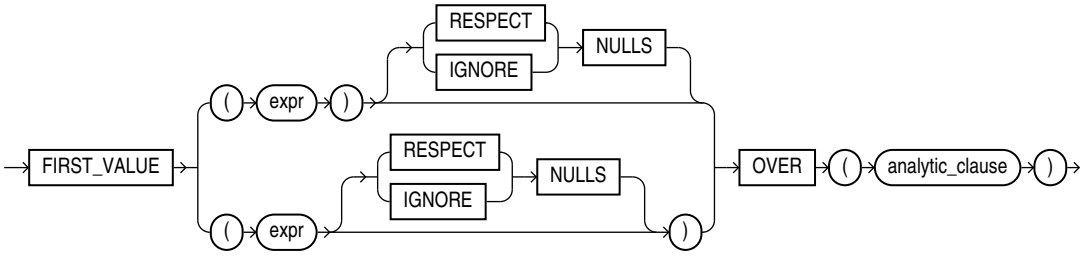
Analytic Example

The next example makes the same calculation as the previous example but returns the result for each employee within the department:

```
SELECT last_name, department_id, salary,
       MIN(salary) KEEP (DENSE_RANK FIRST ORDER BY commission_pct)
         OVER (PARTITION BY department_id) "Worst",
       MAX(salary) KEEP (DENSE_RANK LAST ORDER BY commission_pct)
         OVER (PARTITION BY department_id) "Best"
   FROM employees
   ORDER BY department_id, salary, last_name;

LAST_NAME           DEPARTMENT_ID     SALARY      Worst       Best
------------------- ------------- ---------- ---------- ----------
Whalen                         10       4400       4400       4400
Fay                            20       6000       6000      13000
Hartstein                      20      13000       6000      13000
. . .
Gietz                         110       8300       8300      12008
Higgins                       110      12008       8300      12008
Grant                                   7000       7000       7000
```

FIRST_VALUE

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *expr*

Purpose

`FIRST_VALUE` is an analytic function. It returns the first value in an ordered set of values. If the first value in the set is null, then the function returns `NULL` unless you specify `IGNORE NULLS`. This setting is useful for data densification.

> **Note:**
>
> The two forms of this syntax have the same behavior. The top branch is the ANSI format, which Oracle recommends for ANSI compatibility.

{`RESPECT` | `IGNORE`} `NULLS` determines whether null values of *expr* are included in or eliminated from the calculation. The default is `RESPECT NULLS`. If you specify `IGNORE NULLS`, then

FIRST_VALUE returns the first non-null value in the set, or NULL if all values are null. Refer to "Using Partitioned Outer Joins: Examples" for an example of data densification.

You cannot nest analytic functions by using FIRST_VALUE or any other analytic function for *expr*. However, you can use other built-in function expressions for *expr*. Refer to "About SQL Expressions " for information on valid forms of *expr*.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of FIRST_VALUE when it is a character value

Examples

The following example selects, for each employee in Department 90, the name of the employee with the lowest salary.

```
SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
         OVER (ORDER BY salary ASC ROWS UNBOUNDED PRECEDING) AS fv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);


EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
        102 De Haan                    17000 13-JAN-01 De Haan
        101 Kochhar                    17000 21-SEP-05 De Haan
        100 King                       24000 17-JUN-03 De Haan
```

The example illustrates the nondeterministic nature of the FIRST_VALUE function. Kochhar and DeHaan have the same salary, so are in adjacent rows. Kochhar appears first because the rows returned by the subquery are ordered by hire_date. However, if the rows returned by the subquery are ordered by hire_date in descending order, as in the next example, then the function returns a different value:

```
SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
         OVER (ORDER BY salary ASC ROWS UNBOUNDED PRECEDING) AS fv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER by hire_date DESC);

EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
        101 Kochhar                    17000 21-SEP-05 Kochhar
        102 De Haan                    17000 13-JAN-01 Kochhar
        100 King                       24000 17-JUN-03 Kochhar
```

The following two examples show how to make the FIRST_VALUE function deterministic by ordering on a unique key. By ordering within the function by both salary and the unique key employee_id, you can ensure the same result regardless of the ordering in the subquery.

```
SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
```

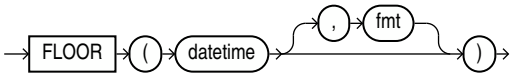
```
        OVER (ORDER BY salary ASC, employee_id ROWS UNBOUNDED PRECEDING) AS fv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);

EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
        101 Kochhar                    17000 21-SEP-05 Kochhar
        102 De Haan                    17000 13-JAN-01 Kochhar
        100 King                       24000 17-JUN-03 Kochhar


SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
         OVER (ORDER BY salary ASC, employee_id ROWS UNBOUNDED PRECEDING) AS fv
   FROM (SELECT * FROM employees
           WHERE department_id = 90
           ORDER BY hire_date DESC);

EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
        101 Kochhar                    17000 21-SEP-05 Kochhar
        102 De Haan                    17000 13-JAN-01 Kochhar
        100 King                       24000 17-JUN-03 Kochhar
```

The following two examples show that the `FIRST_VALUE` function is deterministic when you use a logical offset (`RANGE` instead of `ROWS`). When duplicates are found for the `ORDER BY` expression, the `FIRST_VALUE` is the lowest value of *`expr`*:

```
SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
         OVER (ORDER BY salary ASC RANGE UNBOUNDED PRECEDING) AS fv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);

EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
        102 De Haan                    17000 13-JAN-01 De Haan
        101 Kochhar                    17000 21-SEP-05 De Haan
        100 King                       24000 17-JUN-03 De Haan


SELECT employee_id, last_name, salary, hire_date,
       FIRST_VALUE(last_name)
         OVER (ORDER BY salary ASC RANGE UNBOUNDED PRECEDING) AS fv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date DESC);

EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE FV
----------- ------------------------- ---------- --------- -------
        102 De Haan                    17000 13-JAN-01 De Haan
        101 Kochhar                    17000 21-SEP-05 De Haan
        100 King                       24000 17-JUN-03 De Haan
```

FLOOR (datetime)

Syntax



Purpose

`FLOOR(datetime)` returns the date or the timestamp rounded down to the unit specified by the second argument *`fmt`*, the format model. This function is not sensitive to the `NLS_CALENDAR` session parameter. It operates according to the rules of the Gregorian calendar. The value returned is always of data type `DATE`, even if you specify a different datetime data type for the first argument. If you do not specify the second argument, the default format model '`DD`' is used.

The `FLOOR` and `TRUNC` functions are synonymous for dates and timestamps.

> **See Also:**
>
> Refer to CEIL, FLOOR, ROUND, and TRUNC Date Functions for the permitted format models to use in *`fmt`*.

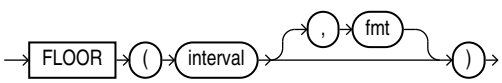
Examples

For these examples `NLS_DATE_FORMAT` is set:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';

SELECT FLOOR(TO_DATE ('28-FEB-2023','DD-MON-YYYY'), 'MM') AS month_floor;

MONTH_FLOOR
--------------------
01-FEB-2023 00:00:00

SELECT FLOOR(TO_TIMESTAMP ('28-FEB-2023 14:10:10','DD-MON-YYYY HH24:MI:SS'),'HH24') AS
hour_floor;

HOUR_FLOOR
--------------------
28-FEB-2023 14:00:00
```

FLOOR (interval)

Syntax



Purpose

`FLOOR(interval)` returns the interval rounded down to the unit specified by the second argument *`fmt`*, the format model .

The result of `FLOOR(interval)` is never larger than *`interval`* . The result precision for year and day is the input precision for year plus one and day plus one, since `FLOOR(interval)` can have overflow . If an interval already has the maximum precision for year and day, the statement compiles but errors at runtime.

For `INTERVAL YEAR TO MONTH`, *`fmt`* can only be year. The default *`fmt`* is year.

For `INTERVAL DAY TO SECOND`, *`fmt`* can be day, hour and minute. The default *`fmt`* is day. Note that *`fmt`* does not support second.

`FLOOR(interval)` supports the format models of `ROUND` and `TRUNC`.

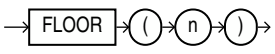
> **See Also:**
>
> Refer to CEIL, FLOOR, ROUND, and TRUNC Date Functions for the permitted format models to use in *`fmt`*.

Examples

```
SELECT FLOOR(INTERVAL '+123-5' YEAR(3) TO MONTH) as year_floor;

YEAR_FLOOR
------------------------------------------------------------------------------
+000000123-00


SELECT FLOOR(INTERVAL '+99-11' YEAR(2) TO MONTH, 'YEAR') as year_floor;

YEAR_FLOOR
------------------------------------------------------------------------------
+000000099-00


SELECT FLOOR(INTERVAL '+4 12:42:10.222' DAY(2) TO SECOND(3), 'DD') as year_floor;

YEAR_FLOOR
------------------------------------------------------------------------------
+000000004 00:00:00.000000000
```

FLOOR (number)

Syntax



Purpose

`FLOOR` returns the largest integer equal to or less than *n*. The number *n* can always be written as the sum of an integer *k* and a positive fraction *f* such that 0 <= *f* < 1 and *n* = *k* + *f*. The value of `FLOOR` is the integer *k*. Thus, the value of `FLOOR` is *n* itself if and only if *n* is precisely an integer.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and CEIL (number)

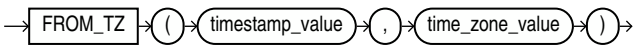
Examples

The following example returns the largest integer equal to or less than 15.7:

```
SELECT FLOOR(15.7) "Floor"
  FROM DUAL;

     Floor
----------
        15
```

FROM_TZ

Syntax



Purpose

`FROM_TZ` converts a timestamp value and a time zone to a `TIMESTAMP WITH TIME ZONE` value. *time_zone_value* is a character string in the format `'TZH:TZM'` or a character expression that returns a string in `TZR` with optional `TZD` format.

Examples

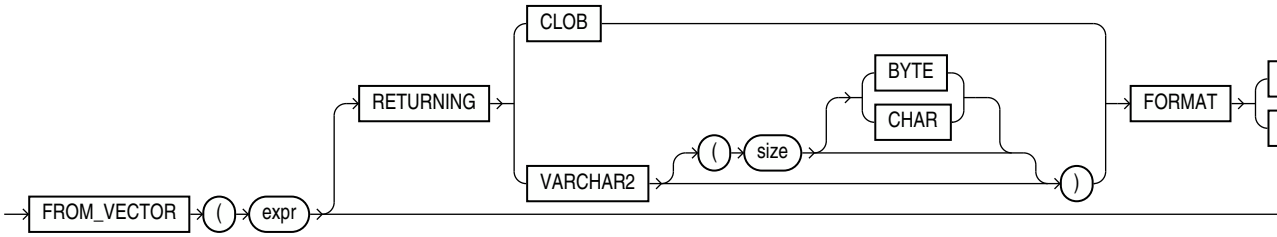
The following example returns a timestamp value to `TIMESTAMP WITH TIME ZONE`:

```
SELECT FROM_TZ(TIMESTAMP '2000-03-28 08:00:00', '3:00')
  FROM DUAL;

FROM_TZ(TIMESTAMP'2000-03-2808:00:00','3:00')
---------------------------------------------------------------
28-MAR-00 08.00.000000000 AM +03:00
```

FROM_VECTOR

FROM_VECTOR takes a vector as input and returns a string of type VARCHAR2 or CLOB as output.

Syntax



Purpose

FROM_VECTOR optionally takes a RETURNING clause to specify the data type of the returned value.

If VARCHAR2 is specified without size, the size of the returned value size is 32767.

You can optionally specify the text format of the output in the FORMAT clause, using the tokens SPARSE or DENSE. Note that the input vector storage format does not need to match the specified output format.

There is no support to convert to CHAR, NCHAR, and NVARCHAR2.

FROM_VECTOR is synonymous with VECTOR_SERIALIZE.

Parameters

expr must evaluate to a vector. The function returns NULL if *expr* is NULL.

Examples

```
SELECT FROM_VECTOR(TO_VECTOR('[1, 2, 3]') );


FROM_VECTOR(TO_VECTOR('[1,2,3]'))
------------------------------------------------------------
[1.0E+000,2.0E+000,3.0E+000]

1 row selected.


SELECT FROM_VECTOR(TO_VECTOR('[1.1, 2.2, 3.3]', 3, FLOAT32) );


FROM_VECTOR(TO_VECTOR('[1.1,2.2,3.3]',3,FLOAT32))
------------------------------------------------------------------
[1.10000002E+000,2.20000005E+000,3.29999995E+000]

1 row selected.


SELECT FROM_VECTOR( TO_VECTOR('[1.1, 2.2, 3.3]', 3, FLOAT32) RETURNING VARCHAR2(1000));
```

```
FROM_VECTOR(TO_VECTOR('[1.1,2.2,3.3]',3,FLOAT32)RETURNINGVARCHAR2(1000))
--------------------------------------------------------------------------------
[1.10000002E+000,2.20000005E+000,3.29999995E+000]

1 row selected.


SELECT FROM_VECTOR(TO_VECTOR('[1.1, 2.2, 3.3]', 3, FLOAT32) RETURNING CLOB );


FROM_VECTOR(TO_VECTOR('[1.1,2.2,3.3]',3,FLOAT32)RETURNINGCLOB)
--------------------------------------------------------------------------------
[1.10000002E+000,2.20000005E+000,3.29999995E+000]

1 row selected.


SELECT FROM_VECTOR(TO_VECTOR('[5,[2,4],[1.0,2.0]]', 5, FLOAT64, SPARSE) RETURNING CLOB
FORMAT SPARSE);

FROM_VECTOR(TO_VECTOR('[5,[2,4],[1.0,2.0]]',5,FLOAT64,SPARSE)RETURNINGCLOBFORMAT
--------------------------------------------------------------------------------
[5,[2,4],[1.0E+000,2.0E+000]]

1 row selected.


SELECT FROM_VECTOR(TO_VECTOR('[5,[2,4],[1.0,2.0]]', 5, FLOAT64, SPARSE) RETURNING CLOB
FORMAT DENSE);

FROM_VECTOR(TO_VECTOR('[5,[2,4],[1.0,2.0]]',5,FLOAT64,SPARSE)RETURNINGCLOBFORMAT
--------------------------------------------------------------------------------
[0,1.0E+000,0,2.0E+000,0]

1 row selected.
```

Note:

- Applications using Oracle Client 23ai libraries or Thin mode drivers can fetch vector data directly, as shown in the following example:

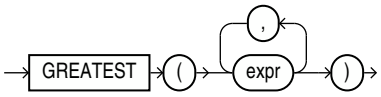
  ```
  SELECT dataVec FROM vecTab;
  ```

- For applications using Oracle Client 23ai libraries prior to 23ai connected to Oracle Database 23ai, use the `FROM_VECTOR` to fetch vector data, as shown by the following example:

  ```
  SELECT FROM_VECTOR(dataVec) FROM vecTab;
  ```

GREATEST

Syntax



Purpose

`GREATEST` returns the greatest of a list of one or more expressions. Oracle Database uses the first *`expr`* to determine the return type. If the first *`expr`* is numeric, then Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type before the comparison, and returns that data type. If the first *`expr`* is not numeric, then each *`expr`* after the first is implicitly converted to the data type of the first *`expr`* before the comparison.

Oracle Database compares each *`expr`* using nonpadded comparison semantics. The comparison is binary by default and is linguistic if the `NLS_COMP` parameter is set to `LINGUISTIC` and the `NLS_SORT` parameter has a setting other than `BINARY`. Character comparison is based on the numerical codes of the characters in the database character set and is performed on whole strings treated as one sequence of bytes, rather than character by character. If the value returned by this function is character data, then its data type is `VARCHAR2` if the first *`expr`* is a character data type and `NVARCHAR2` if the first *`expr`* is a national character data type.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information on character comparison
> - Table 2-9 for more information on implicit conversion and "Floating-Point Numbers " for information on binary-float comparison semantics
> - "LEAST ", which returns the least of a list of one or more expressions
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `GREATEST` uses to compare character values for *`expr`*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Examples

The following statement selects the string with the greatest value:

```
SELECT GREATEST('HARRY', 'HARRIOT', 'HAROLD') "Greatest"
  FROM DUAL;

Greatest
--------
HARRY
```

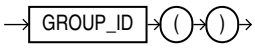
In the following statement, the first argument is numeric. Oracle Database determines that the argument with the highest numeric precedence is the second argument, converts the

remaining arguments to the data type of the second argument, and returns the greatest value as that data type:

```
SELECT GREATEST (1, '3.925', '2.4') "Greatest"
  FROM DUAL;

Greatest
--------
   3.925
```

GROUP_ID

Syntax



Purpose

`GROUP_ID` distinguishes duplicate groups resulting from a `GROUP BY` specification. It is useful in filtering out duplicate groupings from the query result. It returns an Oracle `NUMBER` to uniquely identify duplicate groups. This function is applicable only in a `SELECT` statement that contains a `GROUP BY` clause.

If *n* duplicates exist for a particular grouping, then `GROUP_ID` returns numbers in the range 0 to *n*-1.

Examples

The following example assigns the value `1` to the duplicate `co.country_region` grouping from a query on the sample tables `sh.countries` and `sh.sales`:

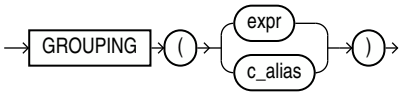
```
SELECT co.country_region, co.country_subregion,
       SUM(s.amount_sold) "Revenue", GROUP_ID() g
  FROM sales s, customers c, countries co
  WHERE s.cust_id = c.cust_id
    AND c.country_id = co.country_id
    AND s.time_id = '1-JAN-00'
    AND co.country_region IN ('Americas', 'Europe')
  GROUP BY GROUPING SETS ( (co.country_region, co.country_subregion),
                           (co.country_region, co.country_subregion) )
  ORDER BY co.country_region, co.country_subregion, "Revenue", g;

COUNTRY_REGION       COUNTRY_SUBREGION                 Revenue          G
-------------------- ------------------------------ ---------- ----------
Americas             Northern America                    944.6          0
Americas             Northern America                    944.6          1
Europe               Western Europe                     566.39          0
Europe               Western Europe                     566.39          1
```

To ensure that only rows with `GROUP_ID` < 1 are returned, add the following `HAVING` clause to the end of the statement :

```
HAVING GROUP_ID() < 1
```

GROUPING

Syntax



Purpose

GROUPING distinguishes superaggregate rows from regular grouped rows. GROUP BY extensions such as ROLLUP and CUBE produce superaggregate rows where the set of all values is represented by null. Using the GROUPING function, you can distinguish a null representing the set of all values in a superaggregate row from a null in a regular row.

The *expr* in the GROUPING function must match one of the expressions in the GROUP BY clause. The function returns a value of 1 if the value of *expr* in the row is a null representing the set of all values. Otherwise, it returns zero. The data type of the value returned by the GROUPING function is Oracle NUMBER. Refer to the SELECT *group_by_clause* for a discussion of these terms.

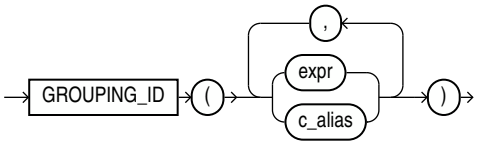
Examples

In the following example, which uses the sample tables hr.departments and hr.employees, if the GROUPING function returns 1 (indicating a superaggregate row rather than a regular row from the table), then the string "All Jobs" appears in the "JOB" column instead of the null that would otherwise appear:

```
SELECT
    DECODE(GROUPING(department_name), 1, 'ALL DEPARTMENTS', department_name)
      AS department,
    DECODE(GROUPING(job_id), 1, 'All Jobs', job_id) AS job,
    COUNT(*) "Total Empl",
    AVG(salary) * 12 "Average Sal"
  FROM employees e, departments d
  WHERE d.department_id = e.department_id
  GROUP BY ROLLUP (department_name, job_id)
  ORDER BY department, job;

DEPARTMENT                     JOB        Total Empl Average Sal
------------------------------ ---------- ---------- -----------
ALL DEPARTMENTS                All Jobs          106  77481.0566
Accounting                     AC_ACCOUNT          1       99600
Accounting                     AC_MGR              1      144096
Accounting                     All Jobs            2      121848
Administration                 AD_ASST             1       52800
Administration                 All Jobs            1       52800
Executive                      AD_PRES             1      288000
Executive                      AD_VP               2      204000
Executive                      All Jobs            3      232000
Finance                        All Jobs            6      103216
Finance                        FI_ACCOUNT          5       95040
. . .
```

GROUPING_ID

Syntax



Purpose

`GROUPING_ID` returns a number corresponding to the `GROUPING` bit vector associated with a row. `GROUPING_ID` is applicable only in a `SELECT` statement that contains a `GROUP BY` extension, such as `ROLLUP` or `CUBE`, and a `GROUPING` function. In queries with many `GROUP BY` expressions, determining the `GROUP BY` level of a particular row requires many `GROUPING` functions, which leads to cumbersome SQL. `GROUPING_ID` is useful in these cases.

`GROUPING_ID` is functionally equivalent to taking the results of multiple `GROUPING` functions and concatenating them into a bit vector (a string of ones and zeros). By using `GROUPING_ID` you can avoid the need for multiple `GROUPING` functions and make row filtering conditions easier to express. Row filtering is easier with `GROUPING_ID` because the desired rows can be identified with a single condition of `GROUPING_ID` = *n*. The function is especially useful when storing multiple levels of aggregation in a single table.

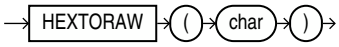
Examples

The following example shows how to extract grouping IDs from a query of the sample table `sh.sales`:

```
SELECT channel_id, promo_id, sum(amount_sold) s_sales,
       GROUPING(channel_id) gc,
       GROUPING(promo_id) gp,
       GROUPING_ID(channel_id, promo_id) gcp,
       GROUPING_ID(promo_id, channel_id) gpc
  FROM sales
  WHERE promo_id > 496
  GROUP BY CUBE(channel_id, promo_id)
  ORDER BY channel_id, promo_id, s_sales, gc;

CHANNEL_ID   PROMO_ID    S_SALES         GC         GP        GCP        GPC
---------- ---------- ---------- ---------- ---------- ---------- ----------
         2        999 25797563.2          0          0          0          0
         2            25797563.2          0          1          1          2
         3        999 55336945.1          0          0          0          0
         3            55336945.1          0          1          1          2
         4        999 13370012.5          0          0          0          0
         4            13370012.5          0          1          1          2
                  999 94504520.8          1          0          2          1
                      94504520.8          1          1          3          3
```

HEXTORAW

Syntax



Purpose

`HEXTORAW` converts *`char`* containing hexadecimal digits in the `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type to a raw value.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> "Data Type Comparison Rules " for more information.

Examples

The following example creates a simple table with a raw column, and inserts a hexadecimal value that has been converted to `RAW`:

```
CREATE TABLE test (raw_col RAW(10));

INSERT INTO test VALUES (HEXTORAW('7D'));
```

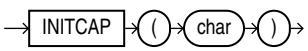
The following example converts hexadecimal digits to a raw value and casts the raw value to `VARCHAR2`:

```
SELECT UTL_RAW.CAST_TO_VARCHAR2(HEXTORAW('4041424344'))
  FROM DUAL;

UTL_RAW.CAST_TO_VARCHAR2(HEXTORAW('4041424344'))
------------------------------------------------
@ABCD
```

> **See Also:**
>
> "RAW and LONG RAW Data Types " and RAWTOHEX

INITCAP

Syntax



Purpose

`INITCAP` returns *`char`*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

`char` can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The return value is the same data type as *`char`*. The database sets the case of the initial characters based on the binary mapping defined for the underlying character set. For linguistic-sensitive uppercase and lowercase, refer to NLS_INITCAP .

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `INITCAP`

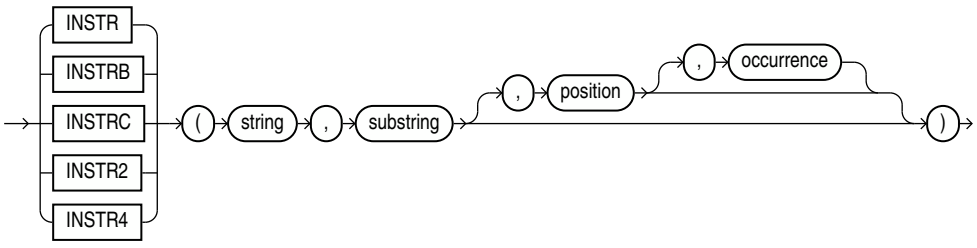
Examples

The following example capitalizes each word in the string:

```
SELECT INITCAP('the soap') "Capitals"
  FROM DUAL;

Capitals
---------
The Soap
```

INSTR

Syntax



Purpose

The `INSTR` functions search *`string`* for *`substring`*. The search operation is defined as comparing the *`substring`* argument with substrings of *`string`* of the same length for equality until a match is found or there are no more substrings left. Each consecutive compared substring of *`string`* begins one character to the right (for forward searches) or one character to the left (for backward searches) from the first character of the previous compared substring. If

a substring that is equal to *substring* is found, then the function returns an integer indicating the position of the first character of this substring. If no such substring is found, then the function returns zero.

- *position* is an nonzero integer indicating the character of *string* where Oracle Database begins the search—that is, the position of the first character of the first substring to compare with *substring*. If *position* is negative, then Oracle counts backward from the end of *string* and then searches backward from the resulting position.
- *occurrence* is an integer indicating which occurrence of *substring* in *string* Oracle should search for. The value of *occurrence* must be positive. If *occurrence* is greater than 1, then the database does not return on the first match but continues comparing consecutive substrings of *string*, as described above, until match number *occurrence* has been found.

INSTR accepts and returns positions in characters as defined by the input character set, with the first character of string having position 1. INSTRB uses bytes instead of characters. INSTRC uses Unicode complete characters. INSTR2 uses UCS2 code points. INSTR4 uses UCS4 code points.

string can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The exceptions are INSTRC, INSTR2, and INSTR4, which do not allow *string* to be a CLOB or NCLOB.

substring can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB.

The value returned is of NUMBER data type.

Both *position* and *occurrence* must be of data type NUMBER, or any data type that can be implicitly converted to NUMBER, and must resolve to an integer. The default values of both *position* and *occurrence* are 1, meaning Oracle begins searching at the first character of *string* for the first occurrence of *substring*. The return value is relative to the beginning of *string*, regardless of the value of *position*.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for more on character length.
> - *Oracle Database SecureFiles and Large Objects Developer's Guide* for more on character length.
> - Table 2-9 for more information on implicit conversion
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation the INSTR functions use to compare the *substring* argument with substrings of *string*

Examples

The following example searches the string CORPORATE FLOOR, beginning with the third character, for the string "OR". It returns the position in CORPORATE FLOOR at which the second occurrence of "OR" begins:

```
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2) "Instring"
  FROM DUAL;

  Instring
```

```
----------
        14
```

In the next example, Oracle counts backward from the last character to the third character from the end, which is the first `O` in `FLOOR`. Oracle then searches backward for the second occurrence of `OR`, and finds that this second occurrence begins with the second character in the search string :

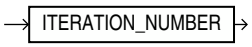
```
SELECT INSTR('CORPORATE FLOOR','OR', -3, 2) "Reversed Instring"
  FROM DUAL;

Reversed Instring
-----------------
                2
```

The next example assumes a double-byte database character set.

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2) "Instring in bytes"
  FROM DUAL;

Instring in bytes
-----------------
               27
```

ITERATION_NUMBER

Syntax



Purpose

The `ITERATION_NUMBER` function can be used only in the *`model_clause`* of the `SELECT` statement and then only when `ITERATE(`*`number`*`)` is specified in the *`model_rules_clause`*. It returns an integer representing the completed iteration through the model rules. The `ITERATION_NUMBER` function returns 0 during the first iteration. For each subsequent iteration, the `ITERATION_NUMBER` function returns the equivalent of *`iteration_number`* plus one.

> **See Also:**
>
> *model_clause* and "Model Expressions" for the syntax and semantics

Examples

The following example assigns the sales of the Mouse Pad for the years 1998 and 1999 to the sales of the Mouse Pad for the years 2001 and 2002 respectively:

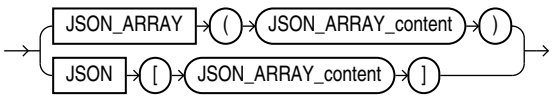
```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
```

```
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER ITERATE(2)
      (
        s['Mouse Pad', 2001 + ITERATION_NUMBER] =
          s['Mouse Pad', 1998 + ITERATION_NUMBER]
      )
  ORDER BY country, prod, year;

COUNTRY       PROD                                     YEAR          S
----------    ----------------------------------    --------   ---------
France        Mouse Pad                                 1998     2509.42
France        Mouse Pad                                 1999     3678.69
France        Mouse Pad                                 2000     3000.72
France        Mouse Pad                                 2001     2509.42
France        Mouse Pad                                 2002     3678.69
France        Standard Mouse                            1998     2390.83
France        Standard Mouse                            1999     2280.45
France        Standard Mouse                            2000     1274.31
France        Standard Mouse                            2001     2164.54
Germany       Mouse Pad                                 1998     5827.87
Germany       Mouse Pad                                 1999     8346.44
Germany       Mouse Pad                                 2000     7375.46
Germany       Mouse Pad                                 2001     5827.87
Germany       Mouse Pad                                 2002     8346.44
Germany       Standard Mouse                            1998     7116.11
Germany       Standard Mouse                            1999     6263.14
Germany       Standard Mouse                            2000     2637.31
Germany       Standard Mouse                            2001     6456.13

18 rows selected.
```

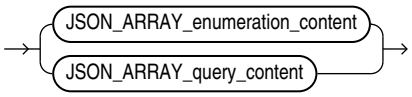
The preceding example requires the view `sales_view_ref`. Refer to "The MODEL clause: Examples" to create this view.

JSON_ARRAY

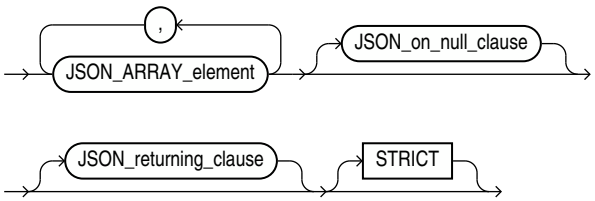
Syntax



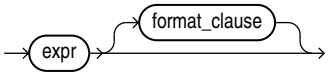
JSON_ARRAY_content



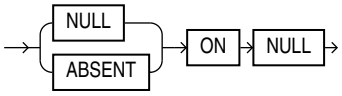
***JSON_ARRAY_enumeration_content*::=**



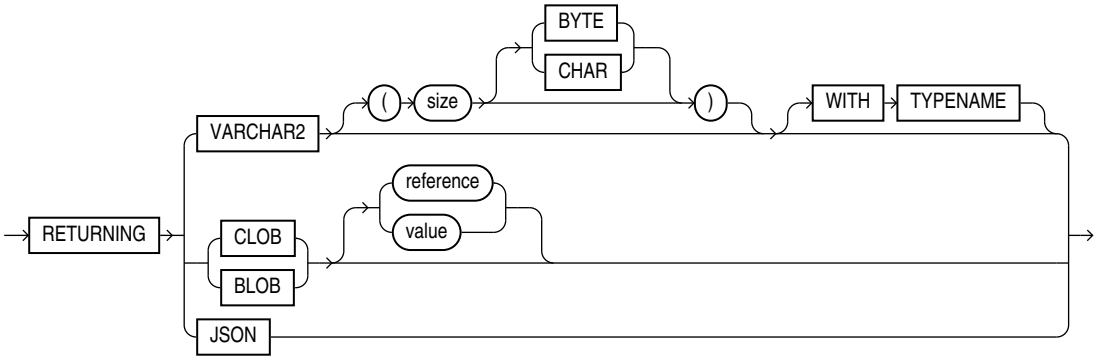
JSON_ARRAY_element



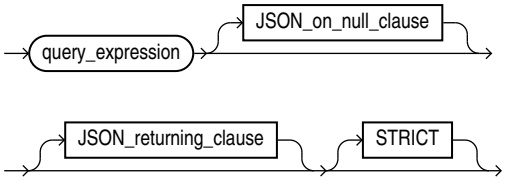
***JSON_on_null_clause*::=**



***JSON_returning_clause*::=**



***JSON_ARRAY_query_content*::=**



Purpose

The SQL/JSON function `JSON_ARRAY` takes as its input a sequence of SQL scalar expressions or *one* collection type instance, `VARRAY` or `NESTED TABLE`.

It converts each expression to a JSON value, and returns a JSON array that contains those JSON values.

If an ADT has a member which is a collection than the type mapping creates a JSON object for the ADT with a nested JSON array for the collection member.

If a collection contains ADT instances then the type mapping will create a JSON array of JSON objects.

> **Note:**
>
> *Generation of JSON Data Using SQL* of the *JSON Developer's Guide*.

JSON_ARRAY_content

Use this clause to define the input to the `JSON_ARRAY` function.

JSON_ARRAY_element

- ***expr***
  For *`expr`*, you can specify any SQL expression that evaluates to a JSON object, a JSON array, a numeric literal, a text literal, date, timestamp, or null. This function converts a numeric literal to a JSON number value, and a text literal to a JSON string value. The date and timestamp data types are printed in the generated JSON object or array as JSON Strings following the ISO 8601 date format.
- ***format_clause***
  You can specify `FORMAT JSON` to indicate that the input string is JSON, and will therefore not be quoted in the output.

JSON_on_null_clause

Use this clause to specify the behavior of this function when *`expr`* evaluates to null.

- `NULL ON NULL` - If you specify this clause, then the function returns the JSON null value.
- `ABSENT ON NULL` - If you specify this clause, then the function omits the value from the JSON array. This is the default.

JSON_returning_clause

Use this clause to specify the type of return value. One of :

- `BLOB` to return a binary large object of the `AL32UTF8` character set.
- `CLOB` to return a character large object containing single-byte or multi-byte characters.
- `VARCHAR2` specifying the size as a number of bytes or characters. The default is bytes. If you omit this clause, or specify the clause without specifying the *`size`* value, then `JSON_ARRAY` returns a character string of type `VARCHAR2(4000)`. Refer to VARCHAR2 Data Type for more information. Note that when specifying the `VARCHAR2` data type elsewhere in

SQL, you are required to specify a size. However, in the *JSON_returning_clause* you can omit the size.

- BOOLEAN
- JSON
- VECTOR

STRICT

Specify the STRICT clause to verify that the output of the JSON generation function is correct JSON. If the check fails, a syntax error is raised.

Refer to JSON_OBJECT for examples.

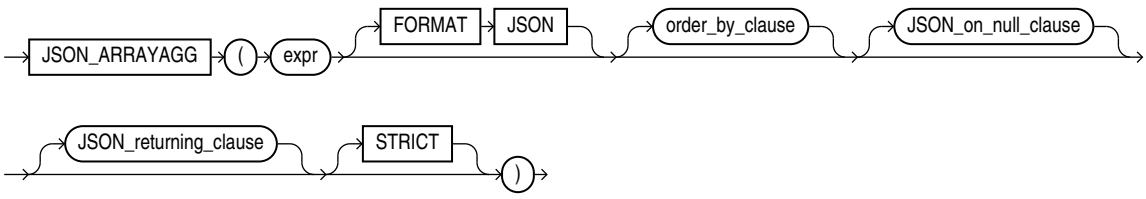
Examples

The following example constructs a JSON array from a JSON object, a JSON array, a numeric literal, a text literal, and null:

```
SELECT JSON_ARRAY (
    JSON_OBJECT('percentage' VALUE .50),
    JSON_ARRAY(1,2,3),
    100,
    'California',
    null
    NULL ON NULL
    ) "JSON Array Example"
  FROM DUAL;

JSON Array Example
--------------------------------------------------------------------------------
[{"percentage":0.5},[1,2,3],100,"California",null]
```

JSON_ARRAYAGG

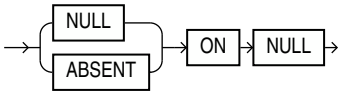
Syntax



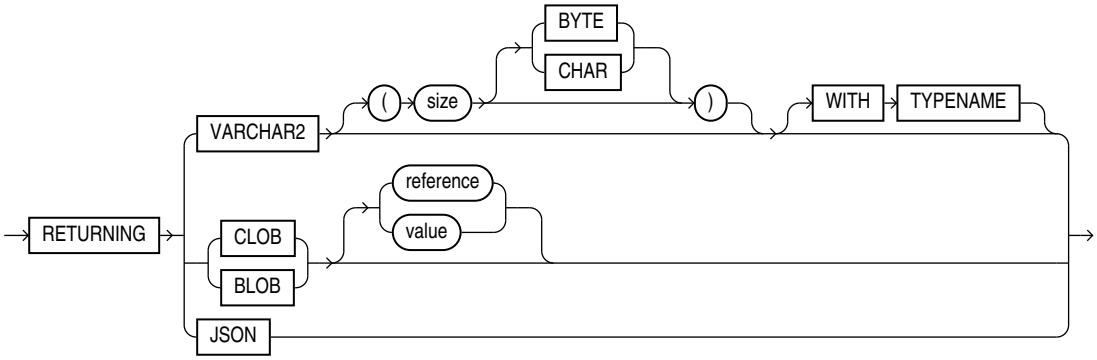
(See *order_by_clause*::= in the documentation on SELECT for the syntax of this clause)

> **Note:**
>
> For JSON_ARRAYAGG, the *order_by* clause must refer to columns. Positional orders are not supported.

***JSON_on_null_clause*::=**



***JSON_returning_clause*::=**



Purpose

The SQL/JSON function `JSON_ARRAYAGG` is an aggregate function. It takes as its input a column of SQL expressions, converts each expression to a JSON value, and returns a single JSON array that contains those JSON values.

expr

For *`expr`*, you can specify any SQL expression that evaluates to a JSON object, a JSON array, a numeric literal, a text literal, or null. This function converts a numeric literal to a JSON number value and a text literal to a JSON string value.

FORMAT JSON

Use this optional clause to indicate that the input string is JSON, and will therefore not be quoted in the output.

order_by_clause

This clause allows you to order the JSON values within the JSON array returned by the statement. Refer to the *order_by_clause* in the documentation on `SELECT` for the full semantics of this clause.

JSON_on_null_clause

Use this clause to specify the behavior of this function when *`expr`* evaluates to null.

- `NULL ON NULL` - If you specify this clause, then the function returns the JSON null value.
- `ABSENT ON NULL` - If you specify this clause, then the function omits the value from the JSON array. This is the default.

JSON_returning_clause

Use this clause to specify the data type of the character string returned by this function. You can specify the following data types:

- `VARCHAR2[(size [BYTE,CHAR])]`

  When specifying the `VARCHAR2` data type elsewhere in SQL, you are required to specify a size. However, in this clause you can omit the size.
- `CLOB` to return a character large object containing single-byte or multi-byte characters.
- `BLOB` to return a binary large object of the `AL32UTF8` character set.
- `JSON` to return JSON data.

  You must set the database initialization parameter compatible to `20` or greater to use the `JSON` type.

If you omit this clause, or if you specify `VARCHAR2` but omit the *`size`* value, then `JSON_ARRAYAGG` returns a character string of type `VARCHAR2(4000)`.

Refer to "Data Types " for more information on the preceding data types.

STRICT

Specify the `STRICT` clause to verify that the output of the JSON generation function is correct JSON. If the check fails, a syntax error is raised.

Refer to JSON_OBJECT for examples.

WITH UNIQUE KEYS

Specify `WITH UNIQUE KEYS` to guarantee that generated JSON objects have unique keys.

Examples

The following statements creates a table `id_table`, which contains ID numbers:

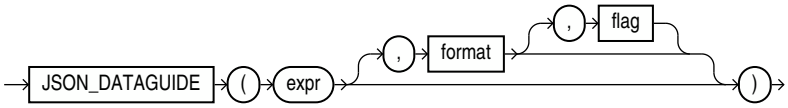
```
CREATE TABLE id_table (id NUMBER);
INSERT INTO id_table VALUES(624);
INSERT INTO id_table VALUES(null);
INSERT INTO id_table VALUES(925);
INSERT INTO id_table VALUES(585);
```

The following example constructs a JSON array from the ID numbers in table `id_table`:

```
SELECT JSON_ARRAYAGG(id ORDER BY id RETURNING VARCHAR2(100)) ID_NUMBERS
  FROM id_table;

ID_NUMBERS
-------------
[585,624,925]
```

JSON_DATAGUIDE

Syntax



Purpose

The aggregate function `JSON_DATAGUIDE` computes the data guide of a set of JSON data. The data guide is returned as a `CLOB` which can be in either flat or hierarchical format depending on the passing format parameter.

expr

`expr` is a SQL expression that evaluates to a JSON object or a JSON array. It can also be a JSON column in a table.

format options

Use the format options to specify the format of the data guide that will be returned. It must be one of the following values:

- `dbms_json.format_flat` for a flat format.
- `dbms_json.format_hierarchical` for a hierarchical format.
- `dbms_json.format_schema` for a data guide of a JSON schema that you can use to validate JSON documents.

If the parameter is the absent, the default is `dbms_json.format_flat`.

See *Data-Guide Formats and Ways of Creating a Data Guide* of the *JSON Developer's Guide*.

flag options

`flag` can have the following values:

- Specify `DBMS_JSON.PRETTY` to improve readability of the returned data guide with appropriate indentation.
- Specify `DBMS_JSON.GEOJSON` for the data guide to auto detect the `GeoJSON` type. The corresponding view column created by the data guide will be of `sdo_geometry` type.
- Specify `DBMS_JSON.GATHER_STATS` for the data guide to collect statistical information. The data guide report generated with `DBMS_JSON.GATHER_STATS` has a new field `o:sample_size`, in addition to all of the other statistical fields that you get with `DBMS_JSON.get_index_dataguide` .
- Specify `DBMS_JSON.DETECT_DATETIME` for the data guide to detect temporal types. The data guide reports a JSON field value that conforms to the ISO 8601 format as a timestamp type, not a string type.
- All values `DBMS_JSON.PRETTY`, `DBMS_JSON.GEOJSON`, and `DBMS_JSON.GATHER_STATS`, and `DBMS_JSON.DETECT_DATETIME` can be combined with a plus sign. For example,

DBMS_JSON.GEOJSON+DBMS_JSON.PRETTY, or DBMS_JSON.GEOJSON+DBMS_JSON.PRETTY+DBMS_JSON.GATHER_STATS.

> **See Also:**
>
> *JSON Data Guide*

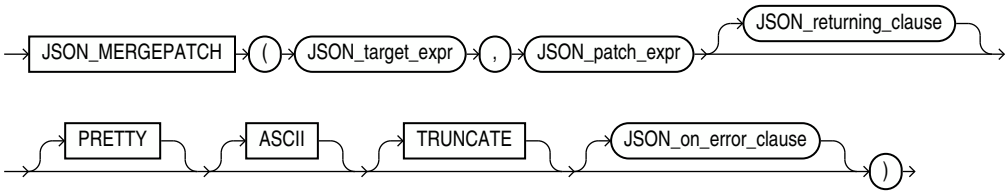
Examples

The following example uses the `j_purchaseorder` table, which is created in "Creating a Table That Contains a JSON Document: Example". This table contains a column of JSON data called `po_document`. This example returns a flat data guide for each year group.

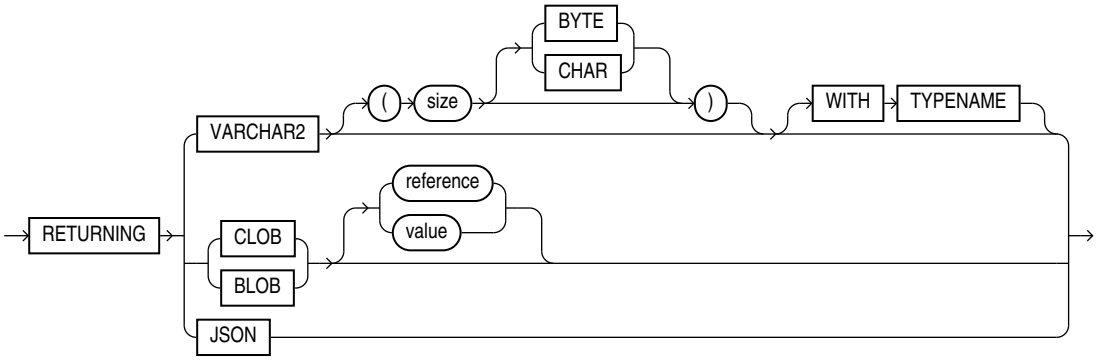
```
SELECT EXTRACT(YEAR FROM date_loaded) YEAR,
       JSON_DATAGUIDE(po_document) "DATA GUIDE"
  FROM j_purchaseorder
  GROUP BY extract(YEAR FROM date_loaded)
  ORDER BY extract(YEAR FROM date_loaded) DESC;


YEAR DATA GUIDE
---- -----------------------------------------
2016 [
       {
         "o:path" : "$.PO_ID",
         "type" : "number",
         "o:length" : 4
       },
       {
         "o:path" : "$.PO_Ref",
         "type" : "string",
         "o:length" : 16
       },
       {
         "o:path" : "$.PO_Items",
         "type" : "array",
         "o:length" : 64
       },
       {
         "o:path" : "$.PO_Items.Part_No",
         "type" : "number",
         "o:length" : 16
       },
       {
         "o:path" : "$.PO_Items.Item_Quantity",
         "type" : "number",
         "o:length" : 2
       }
     ]
. . .
```

JSON_MERGEPATCH

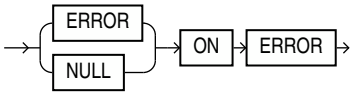
Syntax



***JSON_returning_clause*::=**



***json_on_error_clause*::=**



Purpose

You can use the `JSON_MERGEPATCH` function to update specific portions of a JSON document. You pass it a JSON Merge Patch document in `JSON_patch_expr`, which specifies the changes to make to a specified JSON document, the `JSON_target_expr`.

`JSON_MERGEPATCH` evaluates the patch document against the target document to produce the result document. If the target or the patch document is NULL, then the result is also NULL.

You can input any SQL data type that supports JSON data: `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`. The function returns any of the SQL data types as output.

Data type `JSON` is available only if database initialization parameter compatible is `20` or greater.

The default return type depends on the input data type. If the input type is `JSON`, then `JSON` is also the default return type. Otherwise, `VARCHAR2` is the default return type.

The *`JSON_returning_clause`* specifies the return type of the operator. The default return type is `VARCHAR2(4000)`.

The `PRETTY` keyword specifies that the result should be formatted for human readability.

The `ASCII` keyword specifies that non-ASCII characters should be output using JSON escape sequences.

The `TRUNCATE` keyword specifies that the result document should be truncated to fit in the specified return type.

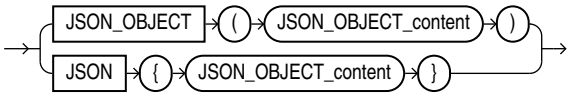
The *`JSON_on_error_clause`* optionally controls the handling of errors that occur during the processing of the target and patch documents.

- `NULL ON ERROR` - Returns null when an error occurs. This is the default.
- `ERROR ON ERROR` - Returns the appropriate Oracle error when an error occurs.

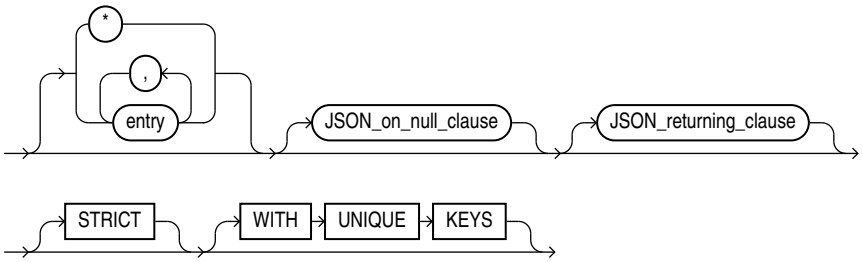
> **See Also:**
>
> - RFC 7396 JSON Merge Patch
> - Updating a JSON Document with JSON Merge Patch

JSON_OBJECT

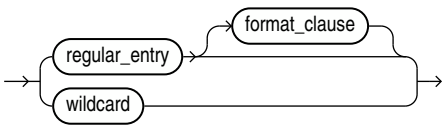
Syntax



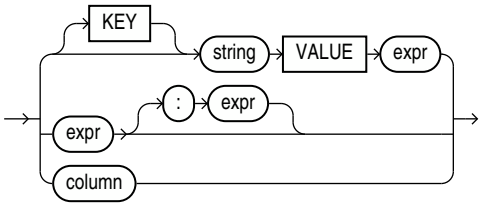
***json_object_content*::=**



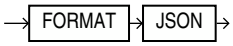
***entry*::=**



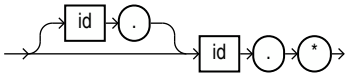
***regular_entry*::=**



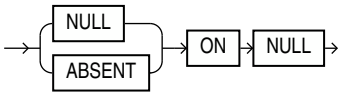
***format_clause*::=**



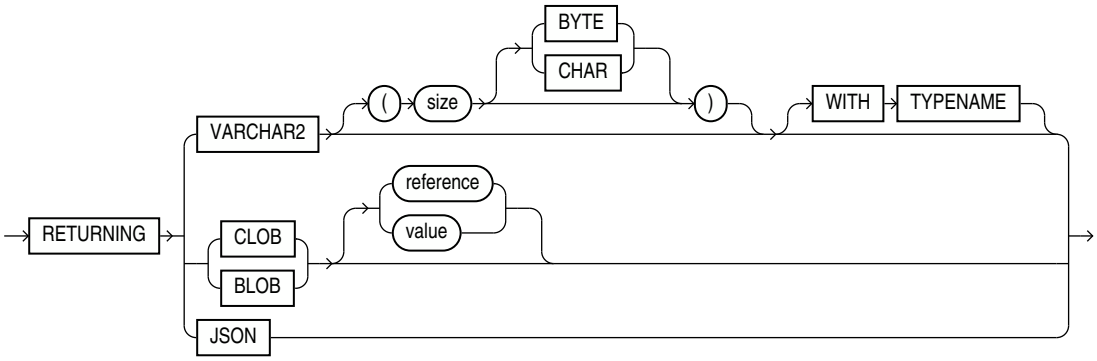
***wildcard*::=**



***JSON_on_null_clause*::=**



***JSON_returning_clause*::=**



Purpose

The SQL/JSON function `JSON_OBJECT` takes as its input either a sequence of key-value pairs or one object type instance. A collection type cannot be passed to `JSON_OBJECT`.

It returns a JSON object that contains an object member for each of those key-value pairs.

entry

`regular_entry`: Use this clause to specify a property key-value pair.

regular_entry

- `KEY` is optional and is provided for semantic clarity.
- Use the optional *`expr`* to specify the property key name as a case-sensitive text literal.
- Use *`expr`* to specify the property value. For *`expr`*, you can specify any expression that evaluates to a SQL numeric literal, text literal, date, or timestamp. The date and timestamp data types are printed in the generated JSON object or array as JSON strings following the ISO date format. If *`expr`* evaluates to a numeric literal, then the resulting property value is a JSON number value; otherwise, the resulting property value is a case-sensitive JSON string value enclosed in double quotation marks.

  You can use the colon to separate `JSON_OBJECT` entries.

  Example

```
SELECT JSON_OBJECT(
'name' : first_name || ' ' || last_name,
'email' : email,
'phone' : phone_number,
'hire_date' : hire_date
)
FROM employees
WHERE employee_id = 140;
```

format_clause

Specify `FORMAT JSON` after an input expression to declare that the value that results from it represents JSON data, and will therefore not be quoted in the output.

wildcard

Wildcard entries select multiple columns and can take the form of *, table.*, view.*, or t_alias.*. Use wildcard entries to map all the columns from a table, subquery, or view to a JSON object without explicitly naming all of the columns in the query. In this case wildcard entries are used in the same way that they are used directly in a select_list.

Example 1

In the resulting JSON object, the key names are equal to the names of the corresponding columns.

```
SELECT JSON_OBJECT(*)
FROM employees
WHERE employee_id = 140;
```

Output 1

```
{"EMPLOYEE_ID":140,"FIRST_NAME":"Joshua","LAST_NAME":"Patel","EMAIL":"JPAT
EL","PHONE_NUMBER":"650.121.1834","HIRE_DATE":"2006-04-
06T00:00:00","JOB_ID":"ST_CLERK","SALARY":2500,"COMMISSION_PCT":null,"MAN
AGER_ID":123,"DEPARTMENT_ID":50}
```

Example 2

This query selects columns from a specific table in a join query.

```
SELECT JSON_OBJECT('NAME' VALUE first_name, d.*)
FROM employees e, departments d
WHERE e.department_id = d.department_id
AND e.employee_id =140
```

Example 3

This query converts the departments table to a single JSON array value.

```
SELECT JSON_ARRAYAGG(JSON_OBJECT(*))
FROM departments
```

JSON_on_null_clause

Use this clause to specify the behavior of this function when *expr* evaluates to null.

- `NULL ON NULL` - When NULL ON NULL is specified, then a JSON NULL value is used as a value for the given key.

  ```
  SELECT JSON_OBJECT('key1' VALUE NULL)  evaluates to  {"key1" :  null}
  ```

- `ABSENT ON NULL` - If you specify this clause, then the function omits the property key-value pair from the JSON object.

JSON_returning_clause

Use this clause to specify the type of return value. One of :

- `VARCHAR2` specifying the size as a number of bytes or characters. The default is bytes. If you omit this clause, or specify the clause without specifying the *size* value, then `JSON_ARRAY` returns a character string of type `VARCHAR2(4000)`. Refer to VARCHAR2 Data Type for more information. Note that when specifying the `VARCHAR2` data type elsewhere in SQL, you are required to specify a size. However, in the *JSON_returning_clause* you can omit the size.
- `CLOB` to return a character large object containing single-byte or multi-byte characters.
- `BLOB` to return a binary large object of the `AL32UTF8` character set.
- `WITH TYPENAME`

STRICT

Specify the `STRICT` clause to verify that the output of the JSON generation function is correct JSON. If the check fails, a syntax error is raised.

Example 1: Output string appears within quotes, because `FORMAT JSON` is not used

```
SELECT JSON_OBJECT ('name' value 'Foo') FROM DUAL
Output:
JSON_OBJECT('NAME'VALUE'FOO'FORMATJSON)
--------------------------------------------------
{"name":"Foo"}
```

Example 2: No quotes around output string when `FORMAT JSON` is used.

```
SELECT JSON_OBJECT ('name' value 'Foo' FORMAT JSON ) FROM DUAL
Output:
JSON_OBJECT('NAME'VALUE'FOO'FORMATJSON)
-----------------------------------------------
{"name":Foo}
```

Example 3: JSON Syntax error when `FORMAT JSON STRICT` is used.

```
SELECT JSON_OBJECT ('name' value 'Foo' FORMAT JSON STRICT ) FROM DUAL
Output:
ORA-40441: JSON syntax error
```

WITH UNIQUE KEYS

Specify `WITH UNIQUE KEYS` to guarantee that generated JSON objects have unique keys.

Example

The following example returns JSON objects that each contain two property key-value pairs:

```
SELECT JSON_OBJECT (
    KEY 'deptno' VALUE d.department_id,
    KEY 'deptname' VALUE d.department_name
    ) "Department Objects"
  FROM departments d
  ORDER BY d.department_id;

Department Objects
---------------------------------------
{"deptno":10,"deptname":"Administration"}
{"deptno":20,"deptname":"Marketing"}
{"deptno":30,"deptname":"Purchasing"}
{"deptno":40,"deptname":"Human Resources"}
{"deptno":50,"deptname":"Shipping"}
. . .
```

JSON_OBJECT Column Entries

In some cases you might want to have JSON object key names match the names of the table columns to avoid repeating the column name in the key value expression. For example:

```
SELECT JSON_OBJECT(
'first_name' VALUE first_name,
'last_name' VALUE last_name,
'email' VALUE email,
'hire_date' VALUE hire_date
)
FROM employees
WHERE employee_id = 140;

{"first_name":"Joshua","last_name":"Patel","email":"JPATEL","hire_date":"2006-
```

```
04-
06T00:00:00"}
```

In such cases you can use a shortcut, where a single column value may be specified as input and the corresponding object entry key is inferred from the name of the column. For example:

```
SELECT JSON_OBJECT(first_name, last_name, email, hire_date)
FROM employees
WHERE employee_id = 140;

{"first_name":"Joshua","last_name":"Patel","email":"JPATEL","hire_date":"2006-
04-
06T00:00:00"}
```

You can use quoted or non-quoted identifiers for column names. If you use non-quoted identifiers, then the case-sensitive value of the identifier, as written in the query, is used to generate the corresponding object key value. However for the purpose of referencing the column value, the identifier is still case-insensitive. For example:

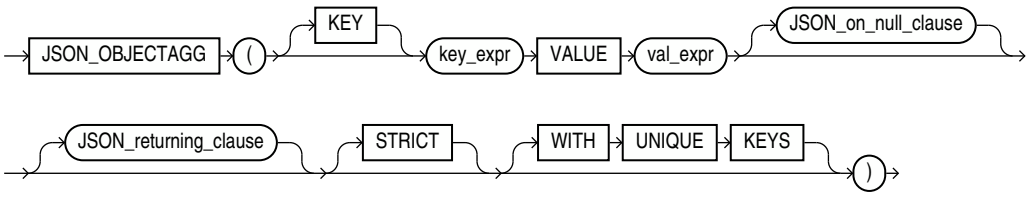
```
SELECT JSON_OBJECT(eMail)
FROM employees
WHERE employee_id = 140

{"eMail":"JPATEL"}
```

Notice that the capital 'M' as typed in the column name is preserved.

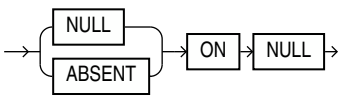
> **See Also:**
>
> Generation of JSON Data Using SQL

JSON_OBJECTAGG

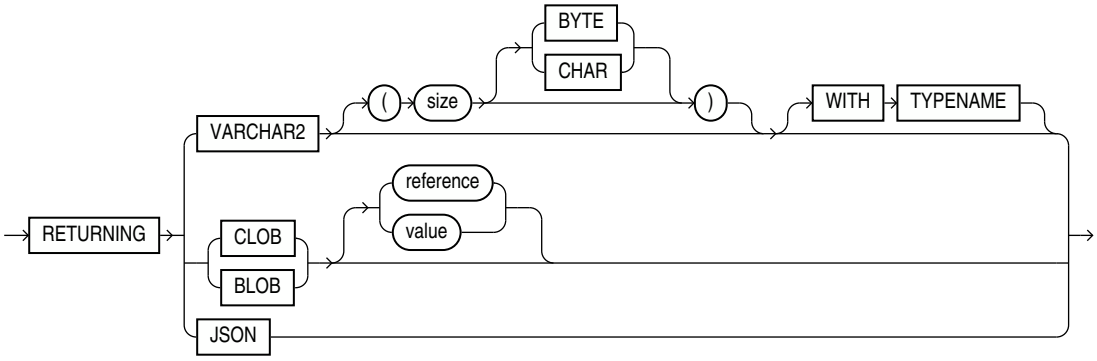
Syntax



***JSON_on_null_clause*::=**



***JSON_returning_clause*::=**



Purpose

The SQL/JSON function `JSON_OBJECTAGG` is an aggregate function. It takes as its input a property key-value pair. Typically, the property key, the property value, or both are columns of SQL expressions. This function constructs an object member for each key-value pair and returns a single JSON object that contains those object members.

[KEY] *string* VALUE *expr*

Use this clause to specify property key-value pairs.

- `KEY` is optional and is provided for semantic clarity.
- Use *`string`* to specify the property key name as a case-sensitive text literal.
- Use *`expr`* to specify the property value. For *`expr`*, you can specify any expression that evaluates to a SQL numeric literal, text literal, date, or timestamp. The date and timestamp data types are printed in the generated JSON object or array as JSON Strings following the ISO 8601 date format. If *`expr`* evaluates to a numeric literal, then the resulting property value is a JSON number value; otherwise, the resulting property value is a case-sensitive JSON string value enclosed in double quotation marks.

FORMAT JSON

Use this optional clause to indicate that the input string is JSON, and will therefore not be quoted in the output.

JSON_on_null_clause

Use this clause to specify the behavior of this function when *`expr`* evaluates to null.

- `NULL ON NULL` - When NULL ON NULL is specified, then a JSON NULL value is used as a value for the given key.
- `ABSENT ON NULL` - If you specify this clause, then the function omits the property key-value pair from the JSON object.

JSON_returning_clause

Use this clause to specify the data type of the character string returned by this function. You can specify the following data types:

- `VARCHAR2[(`*`size`* `[BYTE,CHAR])]`

When specifying the `VARCHAR2` data type elsewhere in SQL, you are required to specify a size. However, in this clause you can omit the size.

- `CLOB` to return a character large object containing single-byte or multi-byte characters.
- `BLOB` to return a binary large object of the `AL32UTF8` character set.
- `JSON` to return JSON data.

  You must set the database initialization parameter compatible to `20` or greater to use the `JSON` data type.

If you omit this clause, or if you specify `VARCHAR2` but omit the *`size`* value, then `JSON_OBJECTAGG` returns a character string of type `VARCHAR2(4000)`.

Refer to "Data Types " for more information on the preceding data types.

STRICT

Specify the `STRICT` clause to verify that the output of the JSON generation function is correct JSON. If the check fails, a syntax error is raised.

Refer to JSON_OBJECT for examples.

WITH UNIQUE KEYS

Specify `WITH UNIQUE KEYS` to guarantee that generated JSON objects have unique keys.

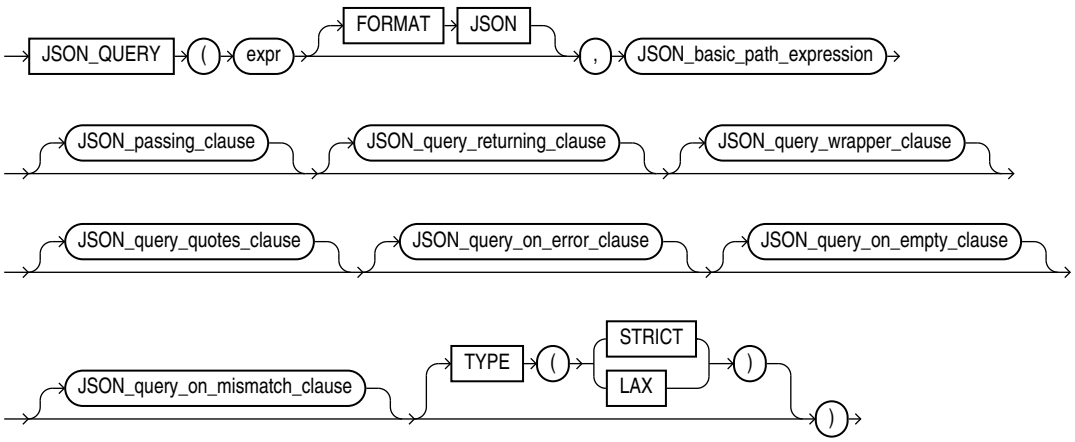
Examples

The following example constructs a JSON object whose members contain department names and department numbers:

```
SELECT JSON_OBJECTAGG(KEY department_name VALUE department_id) "Department Numbers"
  FROM departments
  WHERE department_id <= 30;

Department Numbers
--------------------------------------------------
{"Administration":10,"Marketing":20,"Purchasing":30}
```

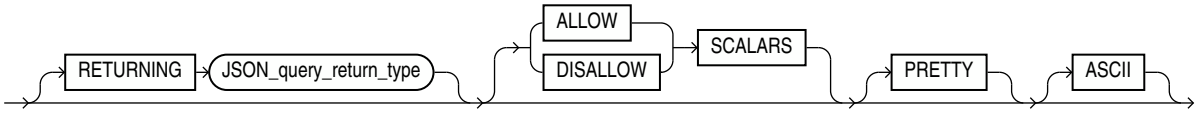
JSON_QUERY

Syntax

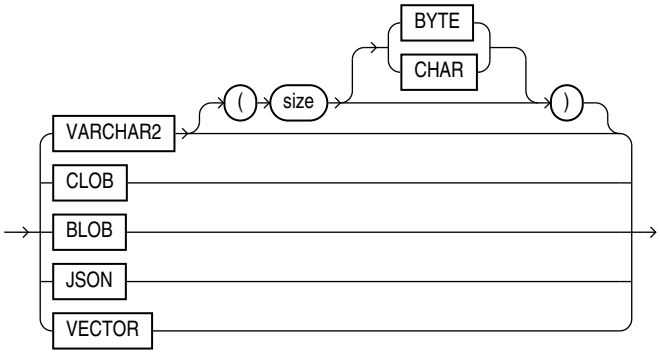


(*JSON_basic_path_expression*: See *Oracle Database JSON Developer's Guide*)

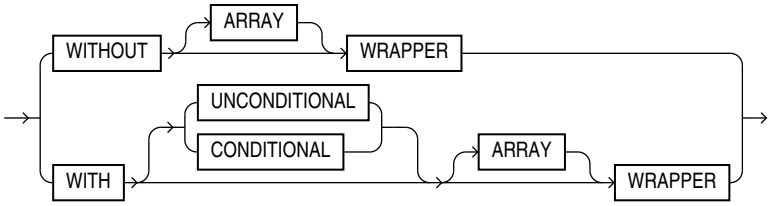
***JSON_query_returning_clause*::=**



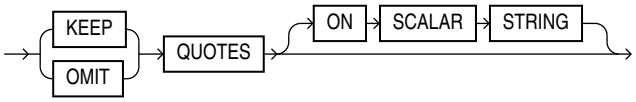
***JSON_query_return_type*::=**



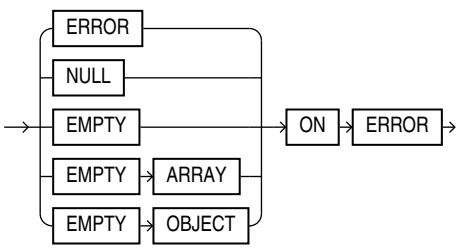
***JSON_query_wrapper_clause*::=**



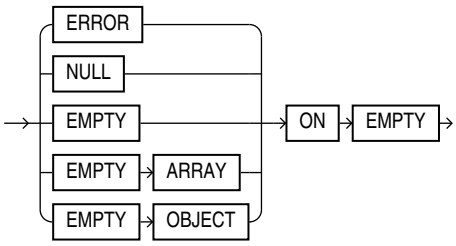
***JSON_query_quotes_clause*::=**



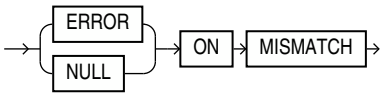
***JSON_query_on_error_clause*::=**



***JSON_query_on_empty_clause*::=**



***JSON_query_on_mismatch_clause*::=**



Purpose

`JSON_QUERY` selects and returns one or more values from JSON data and returns those values. You can use `JSON_QUERY` to retrieve fragments of a JSON document.

> **See Also:**
>
> - *Query JSON Data*
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character value returned by `JSON_QUERY`

expr

Use *`expr`* to specify the JSON data you want to query.

`expr` is a SQL expression that returns an instance of a SQL data type, one of `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`. It can be a table or view column value, a PL/SQLvariable, or a bind variable with proper casting.

If *`expr`* is null, then the function returns null.

If *`expr`* is not a text literal of well-formed JSON data using strict or lax syntax, then the function returns null by default. You can use the *`JSON_query_on_error_clause`* to override this default behavior. Refer to *JSON_query_on_error_clause*.

FORMAT JSON

You must specify `FORMAT JSON` if *`expr`* is a column of data type `BLOB`.

JSON_basic_path_expression

Use this clause to specify a SQL/JSON path expression. The function uses the path expression to evaluate *`expr`* and find one or more JSON values that match, or satisfy the path

expression. The path expression must be a text literal. See *Oracle Database JSON Developer's Guide* for the full semantics of *JSON_basic_path_expression*.

JSON_query_returning_clause

Use this clause to specify the data type and format of the character string returned by this function.

RETURNING

You can use the RETURNING clause to specify the data type of the returned instance, one of JSON, VARCHAR2, CLOB, or BLOB.

The default return type depends on the input data type. If the input type is JSON, then JSON is also the default return type. Otherwise VARCHAR2(4000) is the default return type.

When specifying the VARCHAR2 data type elsewhere in SQL, you are required to specify a size. However, in this clause you can omit the size. In this case, JSON_QUERY returns a character string of type VARCHAR2(4000).

Refer to "VARCHAR2 Data Type " for more information.

If the data type is not large enough to hold the return character string, then JSON_QUERY returns null by default. You can use the *JSON_query_on_error_clause* to override this default behavior. Refer to the *JSON_query_on_error_clause*.

PRETTY

Specify PRETTY to pretty-print the return character string by inserting newline characters and indenting.

ASCII

Specify ASCII to automatically escape any non-ASCII Unicode characters in the return character string, using standard ASCII Unicode escape sequences.

JSON_query_wrapper_clause

Use this clause to control whether this function wraps the values matched by the path expression in an array wrapper—that is, encloses the sequence of values in square brackets ([]).

- Specify WITHOUT WRAPPER to omit the array wrapper. You can specify this clause only if the path expression matches a single JSON object or JSON array. This is the default.
- Specify WITH WRAPPER to include the array wrapper. You must specify this clause if the path expression matches a single scalar value (a value that is not a JSON object or JSON array) or multiple values of any type.
- Specifying the WITH UNCONDITIONAL WRAPPER clause is equivalent to specifying the WITH WRAPPER clause. The UNCONDITIONAL keyword is provided for semantic clarity.
- Specify WITH CONDITIONAL WRAPPER to include the array wrapper only if the path expression matches a single scalar value or multiple values of any type. If the path expression matches a single JSON object or JSON array, then the array wrapper is omitted.

The ARRAY keyword is optional and is provided for semantic clarity.

If the function returns a single scalar value, or multiple values of any type, and you do not specify WITH [UNCONDITIONAL | CONDITIONAL] WRAPPER, then the function returns null by

default. You can use the *JSON_query_on_error_clause* to override this default behavior. Refer to the *JSON_query_on_error_clause*.

JSON_query_on_error_clause

Use this clause to specify the value returned by this function when the following errors occur:

- *expr* is not well-formed JSON data using strict or lax JSON syntax
- No match is found when the JSON data is evaluated using the SQL/JSON path expression. You can override the behavior for this type of error by specifying the *JSON_query_on_empty_clause*.
- The return value data type is not large enough to hold the return character string
- The function matches a single scalar value or, multiple values of any type, and the `WITH [UNCONDITIONAL | CONDITIONAL] WRAPPER` clause is not specified

You can specify the following clauses:

- `NULL ON ERROR` - Returns null when an error occurs. This is the default.
- `ERROR ON ERROR` - Returns the appropriate Oracle error when an error occurs.
- `EMPTY ON ERROR` - Specifying this clause is equivalent to specifying `EMPTY ARRAY ON ERROR`.
- `EMPTY ARRAY ON ERROR` - Returns an empty JSON array (`[]`) when an error occurs.
- `EMPTY OBJECT ON ERROR` - Returns an empty JSON object (`{}`) when an error occurs.

JSON_query_on_empty_clause

Use this clause to specify the value returned by this function if no match is found when the JSON data is evaluated using the SQL/JSON path expression. This clause allows you to specify a different outcome for this type of error than the outcome specified with the *JSON_query_on_error_clause*.

You can specify the following clauses:

- `NULL ON EMPTY` - Returns null when no match is found.
- `ERROR ON EMPTY` - Returns the appropriate Oracle error when no match is found.
- `EMPTY ON EMPTY` - Specifying this clause is equivalent to specifying `EMPTY ARRAY ON EMPTY`.
- `EMPTY ARRAY ON EMPTY` - Returns an empty JSON array (`[]`) when no match is found.
- `EMPTY OBJECT ON EMPTY` - Returns an empty JSON object (`{}`) when no match is found.

If you omit this clause, then the *JSON_query_on_error_clause* determines the value returned when no match is found.

TYPE Clause

For a full discussion of `STRICT` and `LAX` syntax see *About Strict and Lax JSON Syntax*, and *TYPE Clause for SQL Functions and Conditions*

Examples

The following query returns the context item, or the specified string of JSON data. The path expression matches a single JSON object, which does not require an array wrapper. Note that the JSON data is converted to strict JSON syntax in the returned value—that is, the object property names are enclosed in double quotation marks.

```
SELECT JSON_QUERY('{a:100, b:200, c:300}', '$') AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
{"a":100,"b":200,"c":300}
```

The following query returns the value of the member with property name `a`. The path expression matches a scalar value, which must be enclosed in an array wrapper. Therefore, the `WITH WRAPPER` clause is specified.

```
SELECT JSON_QUERY('{a:100, b:200, c:300}', '$.a' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[100]
```

The following query returns the values of all object members. The path expression matches multiple values, which together must be enclosed in an array wrapper. Therefore, the `WITH WRAPPER` clause is specified.

```
SELECT JSON_QUERY('{a:100, b:200, c:300}', '$.*' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[100,200,300]
```

The following query returns the context item, or the specified string of JSON data. The path expression matches a single JSON array, which does not require an array wrapper.

```
SELECT JSON_QUERY('[0,1,2,3,4]', '$') AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[0,1,2,3,4]
```

The following query is similar to the previous query, except the `WITH WRAPPER` clause is specified. Therefore, the JSON array is wrapped in an array wrapper.

```
SELECT JSON_QUERY('[0,1,2,3,4]', '$' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[[0,1,2,3,4]]
```

The following query returns all elements in a JSON array. The path expression matches multiple values, which together must be enclosed in an array wrapper. Therefore, the `WITH WRAPPER` clause is specified.

```
SELECT JSON_QUERY('[0,1,2,3,4]', '$[*]' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[0,1,2,3,4]
```

The following query returns the elements at indexes 0, 3 through 5, and 7 in a JSON array. The path expression matches multiple values, which together must be enclosed in an array wrapper. Therefore, the `WITH WRAPPER` clause is specified.

```
SELECT JSON_QUERY('[0,1,2,3,4,5,6,7,8]', '$[0, 3 to 5, 7]' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[0,3,4,5,7]
```

The following query returns the fourth element in a JSON array. The path expression matches a scalar value, which must be enclosed in an array wrapper. Therefore, the `WITH WRAPPER` clause is specified.

```
SELECT JSON_QUERY('[0,1,2,3,4]', '$[3]' WITH WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[3]
```

The following query returns the first element in a JSON array. The `WITH CONDITIONAL WRAPPER` clause is specified and the path expression matches a single JSON object. Therefore, the value returned is not wrapped in an array. Note that the JSON data is converted to strict JSON syntax in the returned value—that is, the object property name is enclosed in double quotation marks.

```
SELECT JSON_QUERY('[{a:100},{b:200},{c:300}]', '$[0]'
       WITH CONDITIONAL WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
{"a":100}
```

The following query returns all elements in a JSON array. The `WITH CONDITIONAL WRAPPER` clause is specified and the path expression matches multiple JSON objects. Therefore, the value returned is wrapped in an array.

```
SELECT JSON_QUERY('[{"a":100},{"b":200},{"c":300}]', '$[*]'
       WITH CONDITIONAL WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[{"a":100},{"b":200},{"c":300}]
```

The following query is similar to the previous query, except that the value returned is of data type `VARCHAR2(100)`.

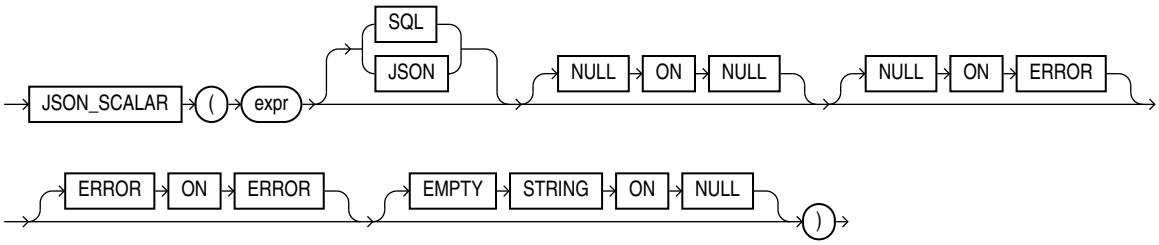
```
SELECT JSON_QUERY('[{"a":100},{"b":200},{"c":300}]', '$[*]'
       RETURNING VARCHAR2(100) WITH CONDITIONAL WRAPPER) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[{"a":100},{"b":200},{"c":300}]
```

The following query returns the fourth element in a JSON array. However, the supplied JSON array does not contain a fourth element, which results in an error. The `EMPTY ON ERROR` clause is specified. Therefore, the query returns an empty JSON array.

```
SELECT JSON_QUERY('[{"a":100},{"b":200},{"c":300}]', '$[3]'
       EMPTY ON ERROR) AS value
  FROM DUAL;

VALUE
--------------------------------------------------------------------------------
[]
```

JSON_SCALAR

Syntax



Purpose

`JSON_SCALAR` accepts a SQL scalar value as input and returns a corresponding JSON scalar value as a `JSON` type instance. The value can be an Oracle-specific JSON-language type, such as a date, which is not part of the JSON standard.

To use `JSON_SCALAR` you must set the database initialization parameter `compatible` is at least `20`. Otherwise it raises an error.

The argument to `JSON_SCALAR` can be an instance of any of these SQL data types: `BINARY_DOUBLE`, `BINARY_FLOAT`, `BLOB`, `CLOB`, `DATE`, `INTERVAL YEAR TO MONTH`, `INTERVAL DAY TO SECOND`, `JSON`, `NUMBER`, `RAW`, `TIMESTAMP`, `VARCHAR`, `VARCHAR2`, or `VECTOR`.

The returned `JSON` type instance is a JSON-language scalar value supported by Oracle.

If the argument to `JSON_SCALAR` is a SQL NULL value, then you can obtain a return value as follows:

- SQL `NULL`, the default behavior
- JSON `null`, using keywords `NULL ON NULL`
- An empty JSON string, " ", using keywords `EMPTY STRING ON NULL`

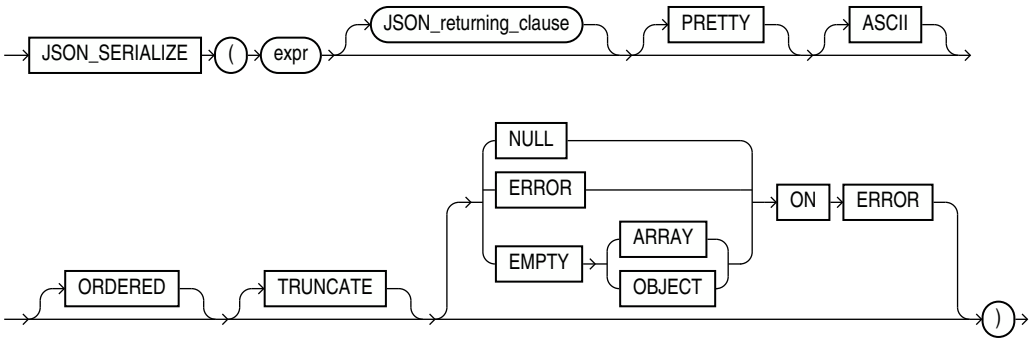
The default behavior of returning SQL `NULL` is the only exception to the rule that a JSON scalar value is returned.

> **See Also:**
>
> See Oracle SQL Function JSON_SCALAR of the *JSON Developer's Guide*.

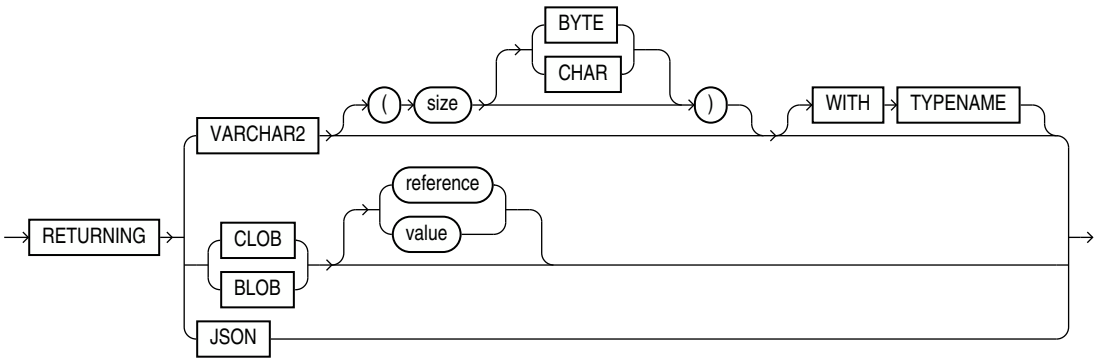
JSON_SERIALIZE

Syntax

json_serialize



json_returning_clause



Purpose

`json_serialize` takes JSON data of any SQL data type (`BLOB`,`CLOB`, `JSON`, or `VARCHAR2`) as input and returns a textual representation of it. You typically use it to transform the result of a query.

You can use `json_serialize` to convert binary JSON data to textual form (`CLOB` or `VARCHAR2`), or to transform textual JSON data by pretty-printing it or escaping non-ASCII Unicode characters in it.

When Oracle SQL function *`vector_serialize`* is applied to a JSON type instance, any non-standard Oracle scalar JSON value is returned as a standard JSON scalar value.

When you apply *`vector_serialize`* to a `VECTOR` type instance, it returns a textual JSON array of numbers.

> **Note:**
>
> You can serialize a `VECTOR` instance to a textual JSON array of numbers using SQL function *`vector_serialize`*. (Function json_serialize serializes only JSON data.) See VECTOR_SERIALIZE

> **See Also:**
>
> Oracle SQL Function JSON_SERIALIZE of the *JSON Developer's Guide.*

expr

`expr` is the input expression. Can be any one of type `JSON`, `VARCHAR2`, `CLOB`, or `BLOB`.

***JSON_returning_clause*::=**

You can use the *`JSON_returning_clause`* to specify the return type of the function. One of `BOOLEAN`, `BLOB`, `CLOB`, `JSON`, or `VARCHAR2`.

The default return type is `VARCHAR2(4000)`.

If the return type is `RAW` or `BLOB`, it contains `UTF8` encoded JSON text.

PRETTY

Specify `PRETTY` if you want the result to be formatted for human readability.

ASCII

Specify `ASCII` if you want non-ASCII characters to be output using JSON escape sequences.

ORDERED

Specify `ORDERED` if you want to reorder key-value pairs alphabetically in ascending order. You can combine `ORDERED` with `PRETTY` and `ASCII`.

Example

```
SELECT JSON_SERIALIZE('{price:20, currency:" €"}' ASCII PRETTY ORDERED) from dual;
{
  "currency" : "\u20AC",
  "price" : 20
}
```

TRUNCATE

Specify `TRUNCATE`, if you want the textual output in the result document to fit into the buffer of the specified return type .

***JSON_on_error_clause*::=**

Specify *`JSON_on_error_clause`* to control the handling of processing errors.

`ERROR ON ERROR` is the default.

`EMPTY ON ERROR` is not supported.

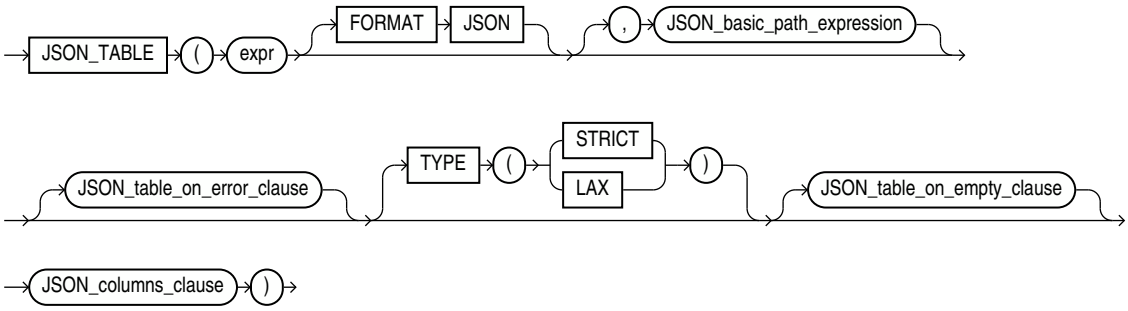
If you specify `TRUNCATE` with `JSON_on_error_clause`, then a value too large for the return type will be truncated to fit into the buffer instead of raising an error.

Example

```
SELECT JSON_SERIALIZE ('{a:[1,2,3,4]}' RETURNING VARCHAR2(3) TRUNCATE ERROR ON ERROR)
from dual
--------
{"a
```

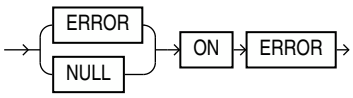
JSON_TABLE

Syntax

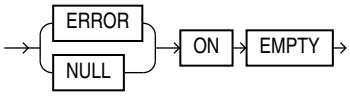


(*JSON_basic_path_expression*: See *Oracle Database JSON Developer's Guide*, *JSON_table_on_error_clause*::=, *JSON_columns_clause*::=)

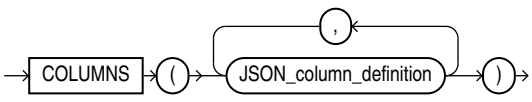
***JSON_table_on_error_clause*::=**



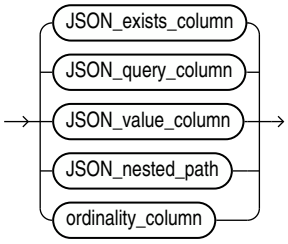
***JSON_table_on_empty_clause*::=**



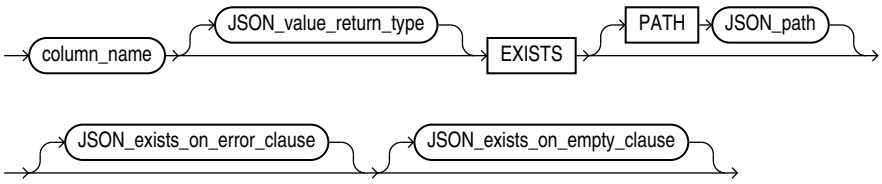
***JSON_columns_clause*::=**



***JSON_column_definition*::=**

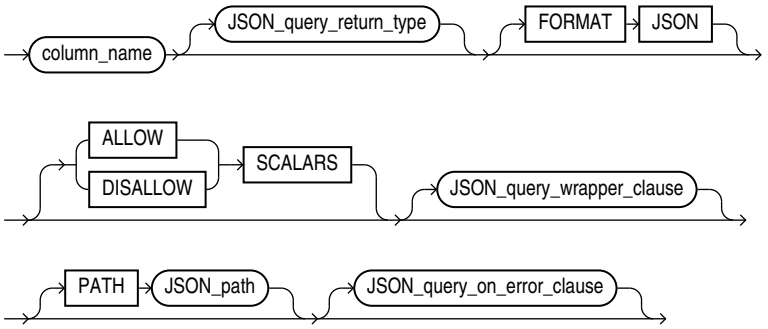


***JSON_exists_column*::=**



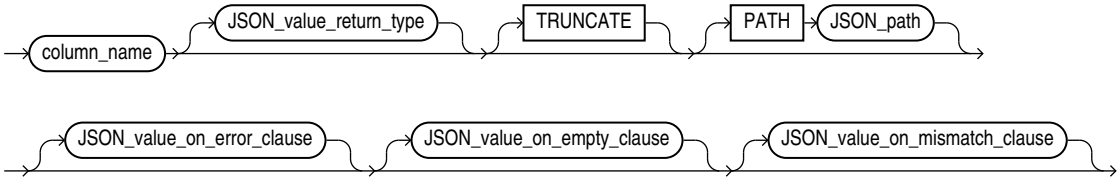
(*JSON_value_return_type*::=—part of `JSON_VALUE`, *`JSON_basic_path_expression`*: See *Oracle Database JSON Developer's Guide*, *JSON_exists_on_error_clause*::=—part of `JSON_EXISTS`)

***JSON_query_column*::=**



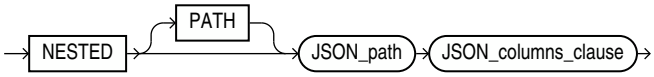
(*JSON_query_return_type*::=, *JSON_query_wrapper_clause*::=, and *JSON_query_on_error_clause*::=—part of `JSON_QUERY`, *`JSON_basic_path_expression`*: See *Oracle Database JSON Developer's Guide*)

***JSON_value_column*::=**



(*JSON_value_return_type*::= and *JSON_value_on_error_clause*::=—part of `JSON_VALUE`, *`JSON_basic_path_expression`*: See *Oracle Database JSON Developer's Guide*)

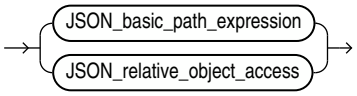
***JSON_nested_path*::=**



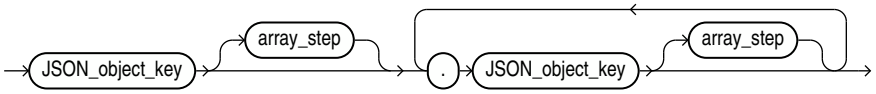
(*`JSON_basic_path_expression`*: See *Oracle Database JSON Developer's Guide*, *JSON_columns_clause*::=)

***ordinality_column*::=**

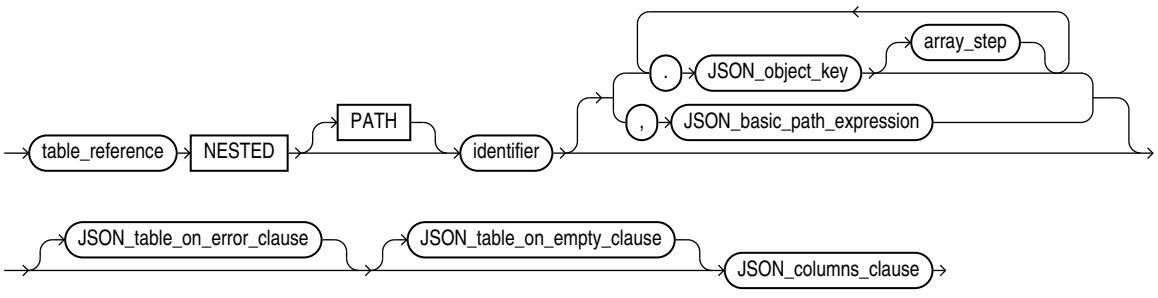
***JSON_path* ::=**



***JSON_relative_object_access* ::=**



***nested_clause* ::=**



Purpose

The SQL/JSON function `JSON_TABLE` creates a relational view of JSON data. It maps the result of a JSON data evaluation into relational rows and columns. You can query the result returned by the function as a virtual relational table using SQL. The main purpose of `JSON_TABLE` is to

create a row of relational data for each object inside a JSON array and output JSON values from within that object as individual SQL column values.

You must specify `JSON_TABLE` only in the `FROM` clause of a `SELECT` statement. The function first applies a path expression, called a **SQL/JSON row path expression**, to the supplied JSON data. The JSON value that matches the row path expression is called a **row source** in that it generates a row of relational data. The `COLUMNS` clause evaluates the row source, finds specific JSON values within the row source, and returns those JSON values as SQL values in individual columns of a row of relational data.

The `COLUMNS` clause enables you to search for JSON values in different ways by using the following clauses:

- *JSON_exists_column* - Evaluates JSON data in the same manner as the `JSON_EXISTS` condition, that is, determines if a specified JSON value exists, and returns either a `VARCHAR2` column of values '`true`' or '`false`', or a `NUMBER` column of values 1 or 0.
- *JSON_query_column* - Evaluates JSON data in the same manner as the `JSON_QUERY` function, that is, finds one or more specified JSON values, and returns a column of character strings that contain those JSON values.
- *JSON_value_column* - Evaluates JSON data in the same manner as the `JSON_VALUE` function, that is, finds a specified scalar JSON value, and returns a column of those JSON values as SQL values.
- *JSON_nested_path* - Allows you to flatten JSON values in a nested JSON object or JSON array into individual columns in a single row along with JSON values from the parent object or array. You can use this clause recursively to project data from multiple layers of nested objects or arrays into a single row.
- *ordinality_column* - Returns a column of generated row numbers.

The column definition clauses allow you to specify a name for each column of data that they return. You can reference these column names elsewhere in the `SELECT` statement, such as in the `SELECT` list and the `WHERE` clause.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to each character data type column in the table generated by `JSON_TABLE`

expr

Use this clause to specify the JSON data to be evaluated. For *expr*, specify an expression that evaluates to a text literal. If *expr* is a column, then the column must be of data type `VARCHAR2`, `CLOB`, or `BLOB`. If *expr* is null, then the function returns null.

If *expr* is not a text literal of well-formed JSON data using strict or lax syntax, then the function returns null by default. You can use the *JSON_table_on_error_clause* to override this default behavior. Refer to *JSON_table_on_error_clause*.

FORMAT JSON

You must specify `FORMAT JSON` if *expr* is a column of data type `BLOB`.

PATH

Use the `PATH` clause to delineate a portion of the row that you want to use as the column content. The absence of the `PATH` clause does not change the behavior with a path of *'$.<column-name>'*, where *<column-name>* is the column name. The name of the object field that is targeted is taken implicitly as the column name. See *Oracle Database JSON Developer's Guide* for the full semantics of `PATH`.

JSON_basic_path_expression

The *JSON_basic_path_expression* is a text literal. See *Oracle Database JSON Developer's Guide* for the full semantics of this clause.

JSON_relative_object_access

Specify this row path expression to enable simple dot notation. The value of `JSON_relative_object_access` is evaluated as a JSON/Path expression relative to the current row item.

For more information on the *JSON_object_key clause*, refer to JSON Object Access Expressions .

JSON_table_on_error_clause

Use this clause to specify the value returned by the function when errors occur:

- *NULL ON ERROR*
  - If the input is not well-formed *JSON* text, no more rows will be returned as soon as the error is detected. Note that since *JSON_TABLE* supports streaming evaluation, rows may be returned prior to encountering the portion of the input with the error.
  - If no match is found when the row path expression is evaluated, no rows are returned.
  - Sets the default error behavior for all column expressions to *NULL ON ERROR*
- *ERROR ON ERROR*
  - If the input is not well-formed *JSON* text, an error will be raised.
  - If no match is found when the row path expression is evaluated, an error will be raised
  - Sets the default error behavior for all column expressions to *ERROR ON ERROR*

TYPE Clause

For a full discussion of `STRICT` and `LAX` syntax see *About Strict and Lax JSON Syntax*, and *TYPE Clause for SQL Functions and Conditions*

JSON_table_on_empty_clause

Use this clause to specify the value returned by this function if no match is found when the JSON data is evaluated using the SQL/JSON path expression. This clause allows you to specify a different outcome for this type of error than the outcome specified with the *JSON_table_on_error_clause.*

You can specify the following clauses:

- `NULL ON EMPTY` - Returns null when no match is found.
- `ERROR ON EMPTY` - Returns the appropriate Oracle error when no match is found.

- DEFAULT *literal* ON EMPTY - Returns *literal* when no match is found. The data type of *literal* must match the data type of the value returned by this function.

If you omit this clause, then the *JSON_table_on_error_clause* determines the value returned when no match is found.

JSON_columns_clause

Use the COLUMNS clause to define the columns in the virtual relational table returned by the JSON_TABLE function.

JSON_exists_column

This clause evaluates JSON data in the same manner as the JSON_EXISTS condition, that is, it determines if a specified JSON value exists. It returns either a VARCHAR2 column of values 'true' or 'false', or a NUMBER column of values 1 or 0.

A value of 'true' or 1 indicates that the JSON value exists and a value of 'false' or 0 indicates that the JSON value does not exist.

You can use the *JSON_value_return_type* clause to control the data type of the returned column. If you omit this clause, then the data type is VARCHAR2(4000). Use *column_name* to specify the name of the returned column. The rest of the clauses of *JSON_exists_column* have the same semantics here as they have for the JSON_EXISTS condition. For full information on these clauses, refer to "JSON_EXISTS Condition". Also see "Using *JSON_exists_column*: Examples" for an example.

JSON_query_column

This clause evaluates JSON data in the same manner as the JSON_QUERY function, that is, it finds one or more specified JSON values, and returns a column of character strings that contain those JSON values.

Use *column_name* to specify the name of the returned column. The rest of the clauses of *JSON_query_column* have the same semantics here as they have for the JSON_QUERY function. For full information on these clauses, refer to JSON_QUERY. Also see "Using *JSON_query_column*: Examples" for an example.

JSON_value_column

This clause evaluates JSON data in the same manner as the JSON_VALUE function, that is, it finds a specified scalar JSON value, and returns a column of those JSON values as SQL values.

Use *column_name* to specify the name of the returned column. The rest of the clauses of *JSON_value_column* have the same semantics here as they have for the JSON_VALUE function. For full information on these clauses, refer to JSON_VALUE. Also see "Using *JSON_value_column*: Examples" for an example.

JSON_nested_path

Use this clause to flatten JSON values in a nested JSON object or JSON array into individual columns in a single row along with JSON values from the parent object or array. You can use this clause recursively to project data from multiple layers of nested objects or arrays into a single row.

Specify the *JSON_basic_path_expression* clause to match the nested object or array. This path expression is relative to the SQL/JSON row path expression specified in the JSON_TABLE function.

Use the `COLUMNS` clause to define the columns of the nested object or array to be returned. This clause is recursive—you can specify the *`JSON_nested_path`* clause within another *`JSON_nested_path`* clause. Also see "Using *JSON_nested_path*: Examples" for an example.

ordinality_column

This clause returns a column of generated row numbers of data type `NUMBER`. You can specify at most one *`ordinality_column`*. Also see "Using *JSON_value_column*: Examples" for an example of using the *`ordinality_column`* clause.

nested_clause

Use the `nested_clause` as a short-hand syntax for mapping JSON values to relational columns. It reuses the syntax of the `JSON_TABLE` columns clause and is essentially equivalent to a left-outer ANSI join with `JSON_TABLE`.

Example 1 using the nested_clause is equivalent to Example 2 using the left-outer join with `JSON_TABLE` .

Example 1 Nested_Clause

```
SELECT t.*
FROM j_purchaseOrder
NESTED po_document COLUMNS(PONumber, Reference, Requestor) t;
PONUMBER REFERENCE REQUESTOR
-------------- ---------------------------- ----------------------------
1600 ABULL-20140421 Alexis Bull
```

Example 2 Left-Outer Join With JSON_TABLE

```
SELECT t.*
FROM j_purchaseOrder LEFT OUTER JOIN
JSON_TABLE(po_document COLUMNS(PONumber, Reference, Requestor)) t ON 1=1;
```

When using the `nested_clause`, the JSON column name following the `NESTED` keyword will not be included in `SELECT *` expansion. For example:

```
SELECT *
FROM j_purchaseOrder
NESTED po_document.LineItems[*]
COLUMNS(ItemNumber, Quantity NUMBER);
ID DATE_LOADED ITEMN QUANTITY
----------------------- ------------------------------------- -------
-----------
6C5589E9A9156… 16-MAY-18 08.40.30.397688 AM -07:00 1 9
6C5589E9A9156… 16-MAY-18 08.40.30.397688 AM -07:00 2 5
```

The result does not include the JSON column name `po_document` as one of the columns in the result.

When unnesting JSON column data, the recommendation is to use `LEFT OUTER JOIN` semantics, so that JSON columns that produce no rows will not filter other non-JSON data from the result. For example,a `j_purchaseOrder` row with a NULL `po_document` column will not filter the possibly non-null relational `columns id` and `date_loaded` from the result.

The columns clause supports all the same features defined for `JSON_TABLE` including nested columns. For example:

```
SELECT t.*
FROM j_purchaseorder
NESTED po_document COLUMNS(PONumber, Reference,
NESTED LineItems[*] COLUMNS(ItemNumber, Quantity)
) t
PONUMBER REFERENCE ITEMN QUANTITY
---------- ------------------------------ ----- ----------
1600 ABULL-20140421 1 9
1600 ABULL-20140421 2 5
```

Examples

Creating a Table That Contains a JSON Document: Example

This example shows how to create and populate table `j_purchaseorder`, which is used in the rest of the `JSON_TABLE` examples in this section.

The following statement creates table `j_purchaseorder`. Column `po_document` is for storing JSON data and, therefore, has an `IS JSON` check constraint to ensure that only well-formed JSON is stored in the column.

```
CREATE TABLE j_purchaseorder
  (id RAW (16) NOT NULL,
   date_loaded TIMESTAMP(6) WITH TIME ZONE,
   po_document CLOB CONSTRAINT ensure_json CHECK (po_document IS JSON));
```

The following statement inserts one row, or one JSON document, into table `j_purchaseorder`:

```
INSERT INTO j_purchaseorder
  VALUES (
    SYS_GUID(),
    SYSTIMESTAMP,
    '{"PONumber"             : 1600,
      "Reference"            : "ABULL-20140421",
       "Requestor"           : "Alexis Bull",
       "User"                : "ABULL",
       "CostCenter"          : "A50",
       "ShippingInstructions" : {"name"   : "Alexis Bull",
                                 "Address": {"street"   : "200 Sporting Green",
                                             "city"     : "South San Francisco",
                                             "state"    : "CA",
                                             "zipCode"  : 99236,
                                             "country" : "United States of America"},
                                 "Phone" : [{"type" : "Office", "number" : "909-555-7307"},
                                            {"type" : "Mobile", "number" : "415-555-1234"}]},
       "Special Instructions" : null,
       "AllowPartialShipment" : true,
       "LineItems" : [{"ItemNumber" : 1,
                       "Part" : {"Description" : "One Magic Christmas",
                                 "UnitPrice"   : 19.95,
                                 "UPCCode"     : 13131092899},
                       "Quantity" : 9.0},
                      {"ItemNumber" : 2,
                       "Part" : {"Description" : "Lethal Weapon",
                                 "UnitPrice"   : 19.95,
                                 "UPCCode"     : 85391628927},
                       "Quantity" : 5.0}]}');
```

Using *JSON_query_column*: Examples

The statement in this example queries JSON data for a specific JSON property using the *JSON_query_column* clause, and returns the property value in a column.

The statement first applies a SQL/JSON row path expression to column `po_document`, which results in a match to the `ShippingInstructions` property. The `COLUMNS` clause then uses the *JSON_query_column* clause to return the `Phone` property value in a `VARCHAR2(100)` column.

```
SELECT jt.phones
FROM j_purchaseorder,
JSON_TABLE(po_document, '$.ShippingInstructions'
COLUMNS
  (phones VARCHAR2(100) FORMAT JSON PATH '$.Phone')) AS jt;
```

```
PHONES
--------------------------------------------------------------------------------
[{"type":"Office","number":"909-555-7307"},{"type":"Mobile","number":"415-555-1234"}]
```

Using *JSON_value_column*: Examples

The statement in this example refines the statement in the previous example by querying JSON data for specific JSON values using the *JSON_value_column* clause, and returns the JSON values as SQL values in relational rows and columns.

The statement first applies a SQL/JSON row path expression to column `po_document`, which results in a match to the elements in the JSON array `Phone`. These elements are JSON objects that contain two members named `type` and `number`. The statement uses the `COLUMNS` clause to return the `type` value for each object in a `VARCHAR2(10)` column called `phone_type`, and the `number` value for each object in a `VARCHAR2(20)` column called `phone_num`. The statement also returns an ordinal column named `row_number`.

```
SELECT jt.*
FROM j_purchaseorder,
JSON_TABLE(po_document, '$.ShippingInstructions.Phone[*]'
COLUMNS (row_number FOR ORDINALITY,
         phone_type VARCHAR2(10) PATH '$.type',
         phone_num VARCHAR2(20) PATH '$.number'))
AS jt;

ROW_NUMBER PHONE_TYPE PHONE_NUM
---------- ---------- --------------------
         1 Office     909-555-7307
         2 Mobile     415-555-1234
```

Using *JSON_exists_column*: Examples

The statements in this example test whether a JSON value exists in JSON data using the *JSON_exists_column* clause. The first example returns the result of the test as a '`true`' or '`false`' value in a column. The second example uses the result of the test in the `WHERE` clause.

The following statement first applies a SQL/JSON row path expression to column `po_document`, which results in a match to the entire context item, or JSON document. It then uses the `COLUMNS` clause to return the requestor's name and a string value of '`true`' or '`false`' indicating whether the JSON data for that requestor contains a zip code. The `COLUMNS` clause first uses the *JSON_value_column* clause to return the `Requestor` value in a `VARCHAR2(32)` column called `requestor`. It then uses the *JSON_exists_column* clause to determine if the `zipCode` object exists and returns the result in a `VARCHAR2(5)` column called `has_zip`.

```
SELECT requestor, has_zip
FROM j_purchaseorder,
JSON_TABLE(po_document, '$'
COLUMNS
  (requestor VARCHAR2(32) PATH '$.Requestor',
   has_zip VARCHAR2(5) EXISTS PATH '$.ShippingInstructions.Address.zipCode'));

REQUESTOR                       HAS_ZIP
------------------------------- -------
Alexis Bull                     true
```

The following statement is similar to the previous statement, except that it uses the value of `has_zip` in the `WHERE` clause to determine whether to return the `Requestor` value:

```
SELECT requestor
FROM j_purchaseorder,
JSON_TABLE(po_document, '$'
COLUMNS
  (requestor VARCHAR2(32) PATH '$.Requestor',
   has_zip VARCHAR2(5) EXISTS PATH '$.ShippingInstructions.Address.zipCode'))
WHERE (has_zip = 'true');

REQUESTOR
-------------------------------
Alexis Bull
```

Using *JSON_nested_path*: Examples

The following two simple statements demonstrate the functionality of the *JSON_nested_path* clause. They operate on a simple JSON array that contains three elements. The first two elements are numbers. The third element is a nested JSON array that contains two string value elements.

The following statement does not use the *JSON_nested_path* clause. It returns the three elements in the array in a single row. The nested array is returned in its entirety.

```
SELECT *
FROM JSON_TABLE('[1,2,["a","b"]]', '$'
COLUMNS (outer_value_0 NUMBER PATH '$[0]',
         outer_value_1 NUMBER PATH '$[1]',
         outer_value_2 VARCHAR2(20) FORMAT JSON PATH '$[2]'));

OUTER_VALUE_0 OUTER_VALUE_1 OUTER_VALUE_2
------------- ------------- --------------------
            1             2 ["a","b"]
```

The following statement is different from the previous statement because it uses the *JSON_nested_path* clause to return the individual elements of the nested array in individual columns in a single row along with the parent array elements.

```
SELECT *
FROM JSON_TABLE('[1,2,["a","b"]]', '$'
COLUMNS (outer_value_0 NUMBER PATH '$[0]',
         outer_value_1 NUMBER PATH '$[1]',
         NESTED PATH '$[2]'
         COLUMNS (nested_value_0 VARCHAR2(1) PATH '$[0]',
                  nested_value_1 VARCHAR2(1) PATH '$[1]')));

OUTER_VALUE_0 OUTER_VALUE_1 NESTED_VALUE_0 NESTED_VALUE_1
------------- ------------- -------------- --------------
            1             2 a              b
```

The previous example shows how to use *JSON_nested_path* with a nested JSON array. The following example shows how to use the *JSON_nested_path* clause with a nested JSON object by returning the individual elements of the nested object in individual columns in a single row along with the parent object elements.

```
SELECT *
FROM JSON_TABLE('{a:100, b:200, c:{d:300, e:400}}', '$'
COLUMNS (outer_value_0 NUMBER PATH '$.a',
         outer_value_1 NUMBER PATH '$.b',
         NESTED PATH '$.c'
         COLUMNS (nested_value_0 NUMBER PATH '$.d',
                  nested_value_1 NUMBER PATH '$.e')));

OUTER_VALUE_0 OUTER_VALUE_1 NESTED_VALUE_0 NESTED_VALUE_1
------------- ------------- -------------- --------------
          100           200            300            400
```

The following statement uses the *JSON_nested_path* clause when querying the `j_purchaseorder` table. It first applies a row path expression to column `po_document`, which results in a match to the entire context item, or JSON document. It then uses the `COLUMNS` clause to return the `Requestor` value in a `VARCHAR2(32)` column called `requestor`. It then uses the *JSON_nested_path* clause to return the property values of the individual objects in each member of the nested `Phone` array. Note that a row is generated for each member of the nested array, and each row contains the corresponding `Requestor` value.

```
SELECT jt.*
FROM j_purchaseorder,
JSON_TABLE(po_document, '$'
COLUMNS
  (requestor VARCHAR2(32) PATH '$.Requestor',
   NESTED PATH '$.ShippingInstructions.Phone[*]'
     COLUMNS (phone_type VARCHAR2(32) PATH '$.type',
              phone_num VARCHAR2(20) PATH '$.number')))
AS jt;


REQUESTOR            PHONE_TYPE           PHONE_NUM
-------------------- -------------------- ---------------
Alexis Bull          Office               909-555-7307
Alexis Bull          Mobile               415-555-1234
```

The following example shows the use of simple dot-notation in `JSON_nested_path` and its equivalent without dot notation.

```
SELECT c.*
FROM customer t,
JSON_TABLE(t.json COLUMNS(
id, name, phone, address,
NESTED orders[*] COLUMNS(
updated, status,
NESTED lineitems[*] COLUMNS(
description, quantity NUMBER, price NUMBER
)
)
)) c;
```

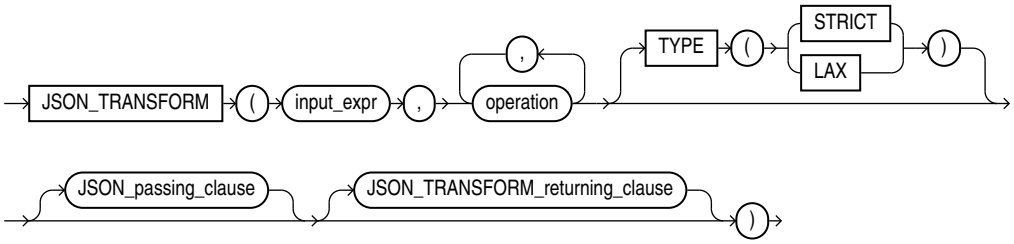
The above statement in dot notation is equivalent to the following one without dot notation:

```
SELECT c.*
FROM customer t,
JSON_TABLE(t.json, '$' COLUMNS(
```

```
id PATH '$.id',
name PATH '$.name',
phone PATH '$.phone',
address PATH '$.address',
NESTED PATH '$.orders[*]' COLUMNS(
updated PATH '$.updated',
status PATH '$.status',
NESTED PATH '$.lineitems[*]' COLUMNS(
description PATH '$.description',
quantity NUMBER PATH '$.quantity',
price NUMBER PATH '$.price'
)
)
)) c;
```

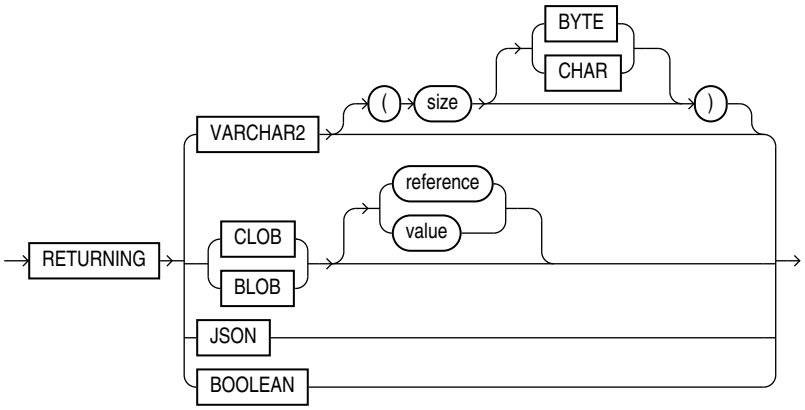
JSON_TRANSFORM

Syntax



(operation::=, JSON_TRANSFORM_returning_clause::=, JSON_passing_clause::=)

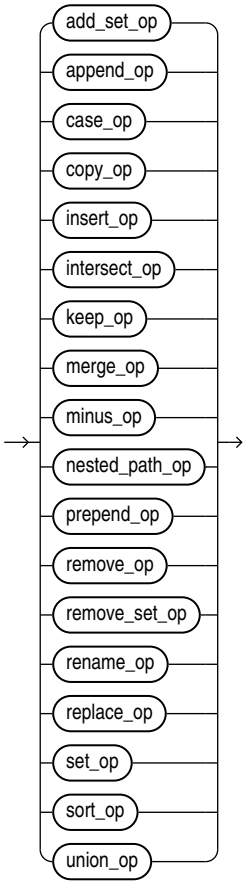
***JSON_TRANSFORM_returning_clause*::=**



***JSON_passing_clause*::=**

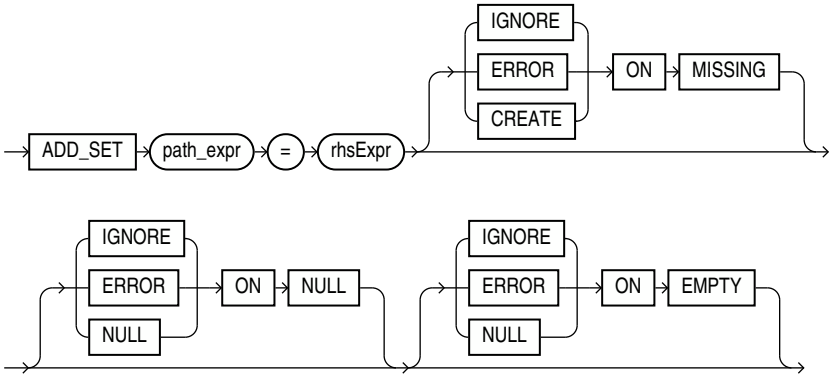
For details on *JSON_passing_clause* see JSON_EXISTS Condition.

***operation* ::=**

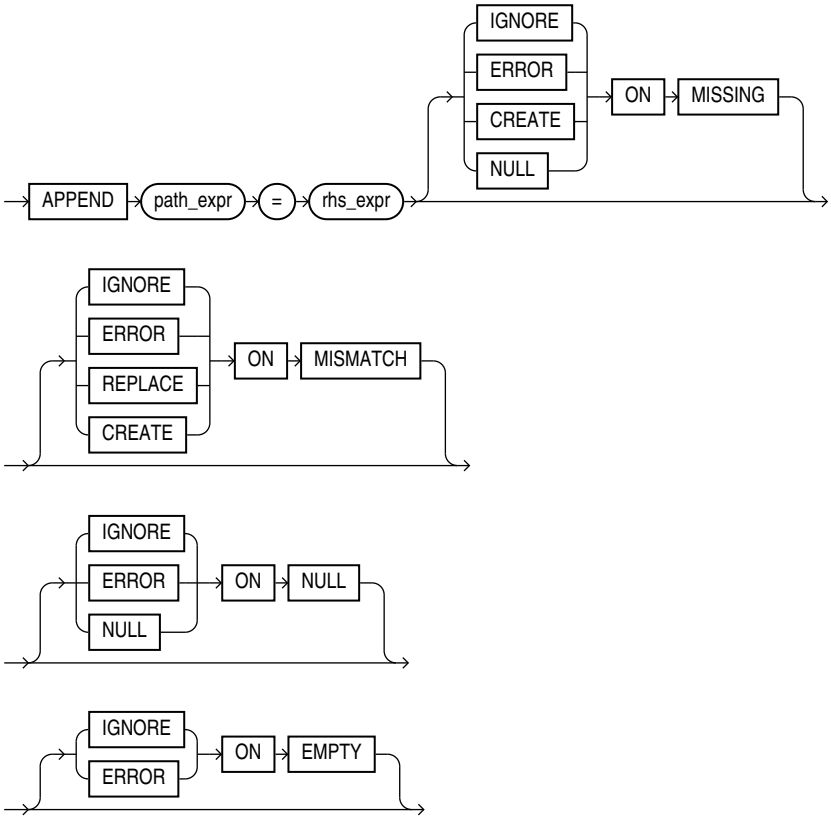


(add_set_op::=,append_op::=,
case_op::=,copy_op::=,insert_op::=,intersect_op::=,keep_op::=,merge_op::=,minus_op::=,nested_path_op::=,prepend_op::=,remove_op::=,rename_op::=,remove_set_op::=,replace_op::=,set_op,sort_op,union_op,)

***add_set_op*::=**

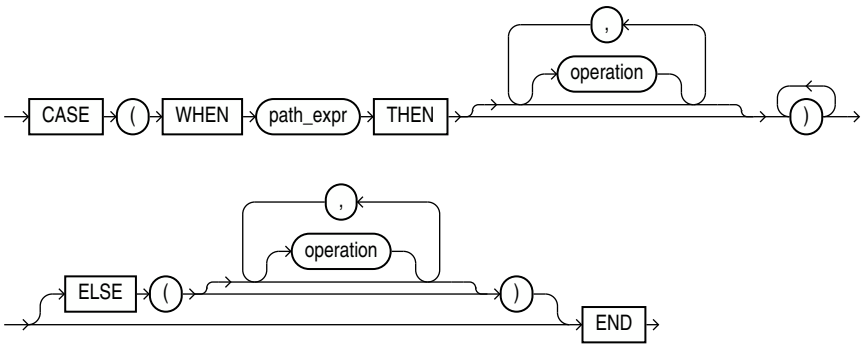


***append_op* ::=**

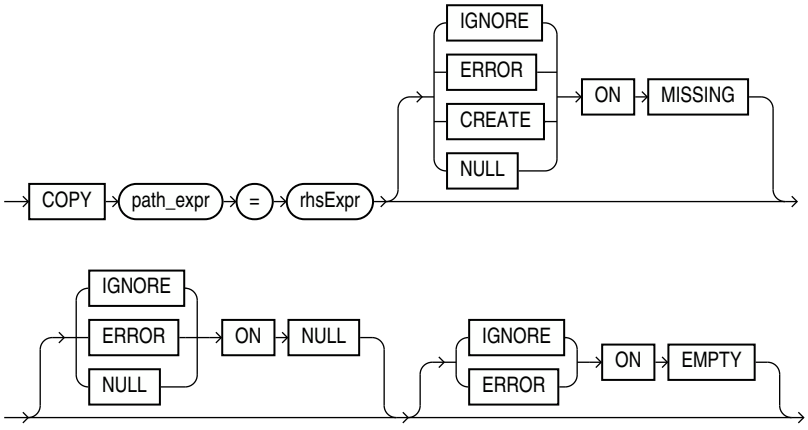


rhs_expr::=

***case_op*::=**

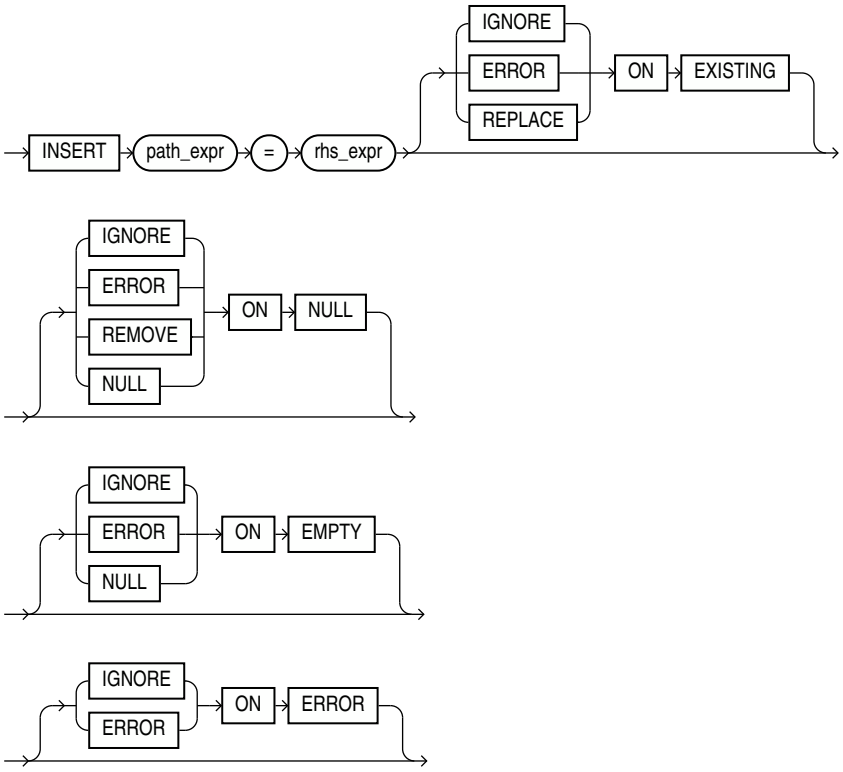


***copy_op*::=**



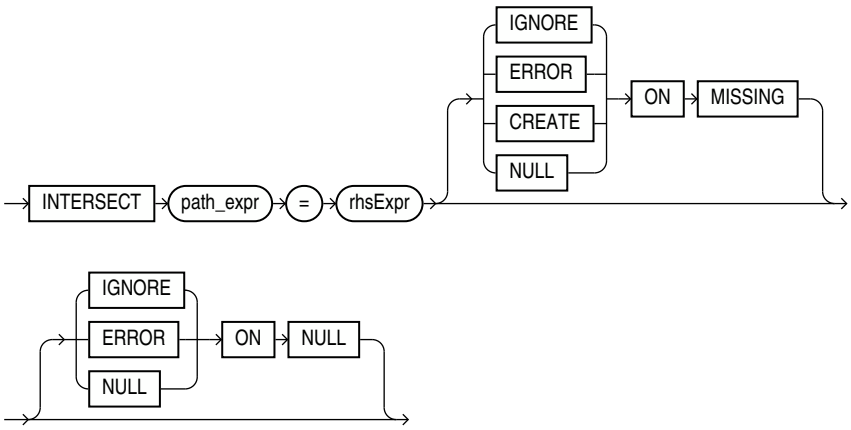
rhs_expr::=

***insert_op* ::=**



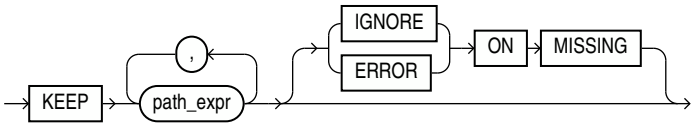
rhs_expr::=

***intersect_op*::=**

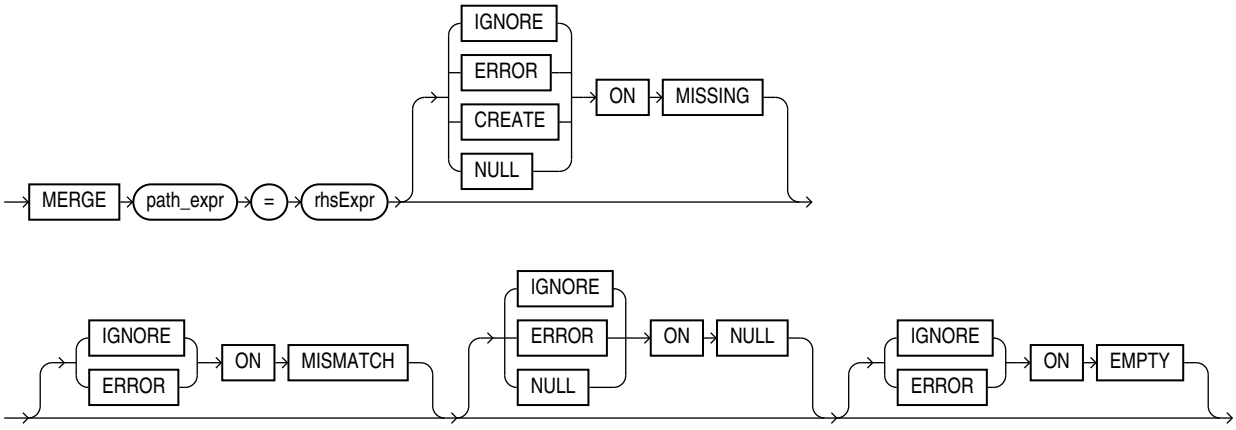


rhs_expr::=

***keep_op* ::=**

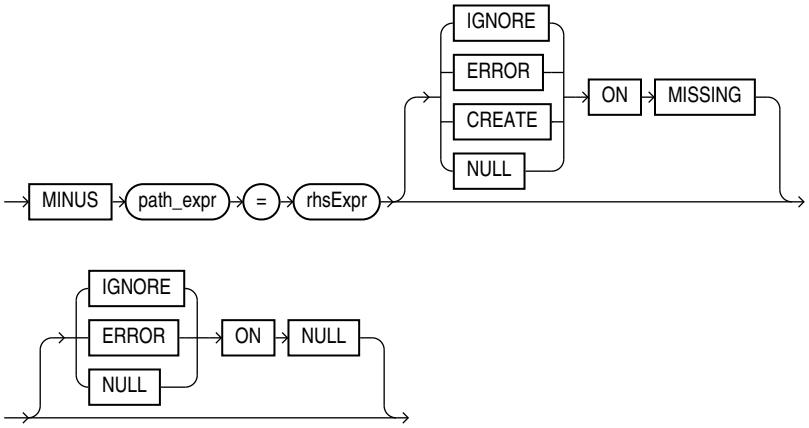


***merge_op*::=**



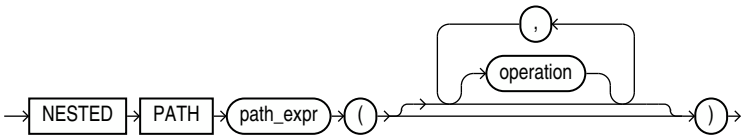
rhs_expr::=

***minus_op*::=**

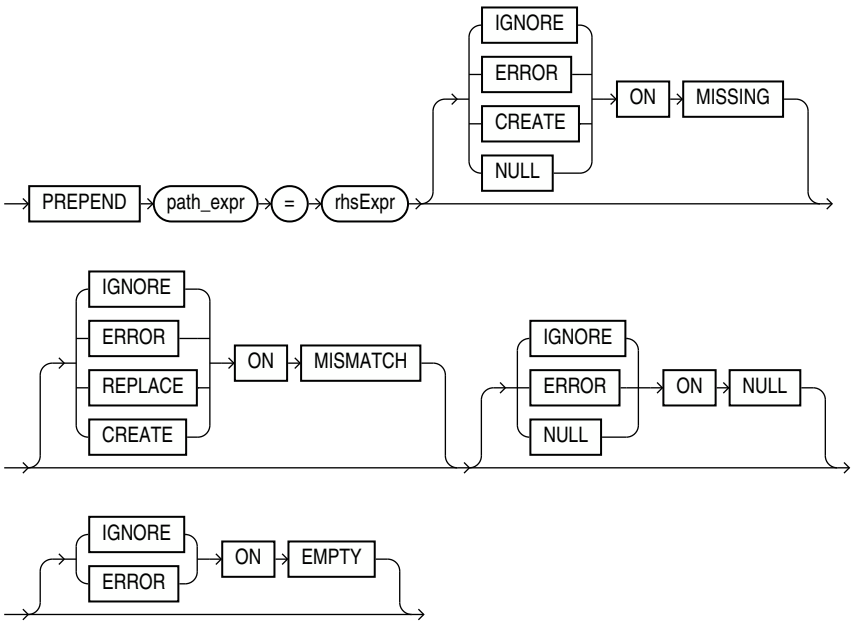


rhs_expr::=

***nested_path_op*::=**

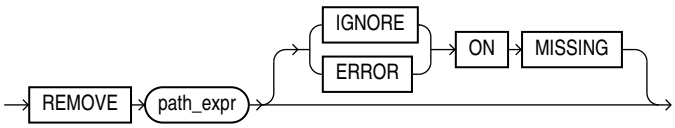


***prepend_op*::=**

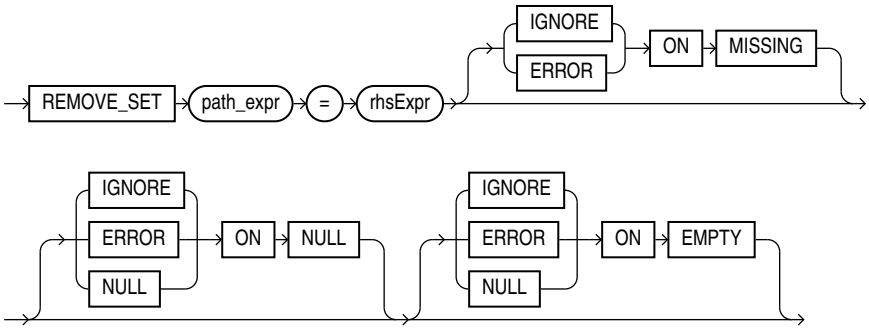


rhs_expr::=

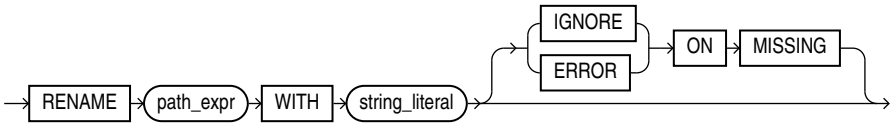
***remove_op* ::=**



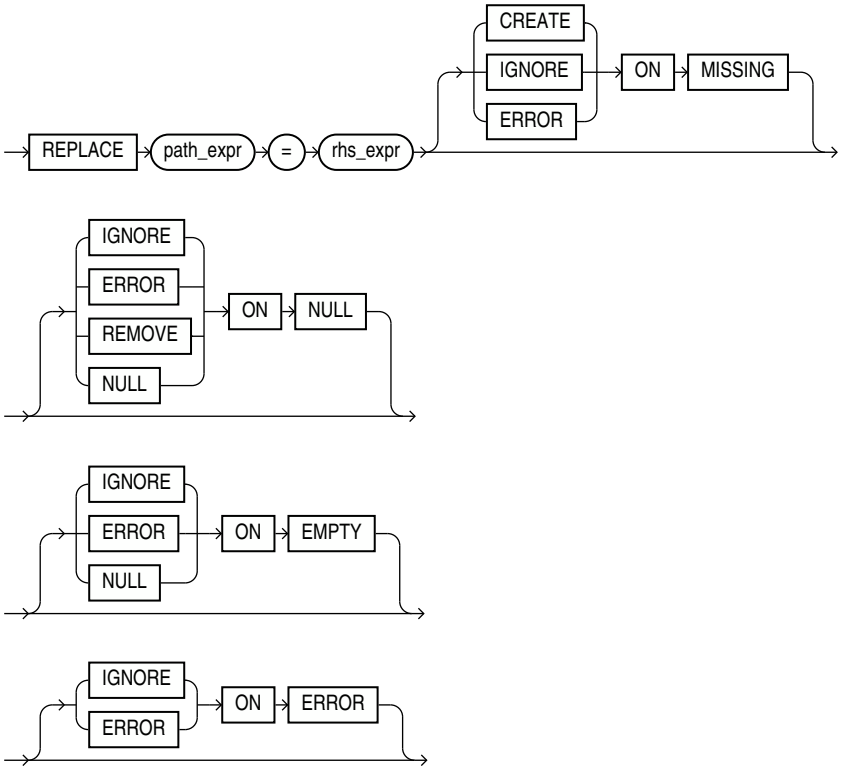
***remove_set_op*::=**



***rename_op* ::=**

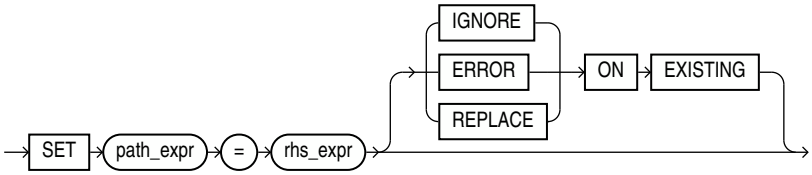


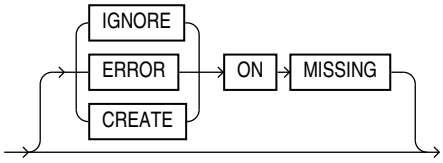
***replace_op* ::=**

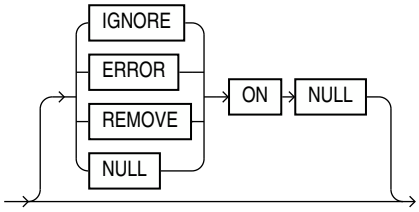


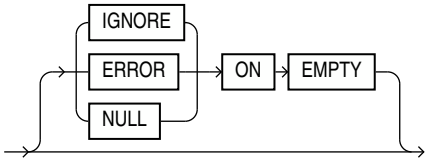
rhs_expr::=

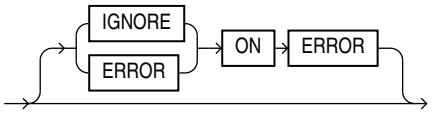
***set_op* ::=**





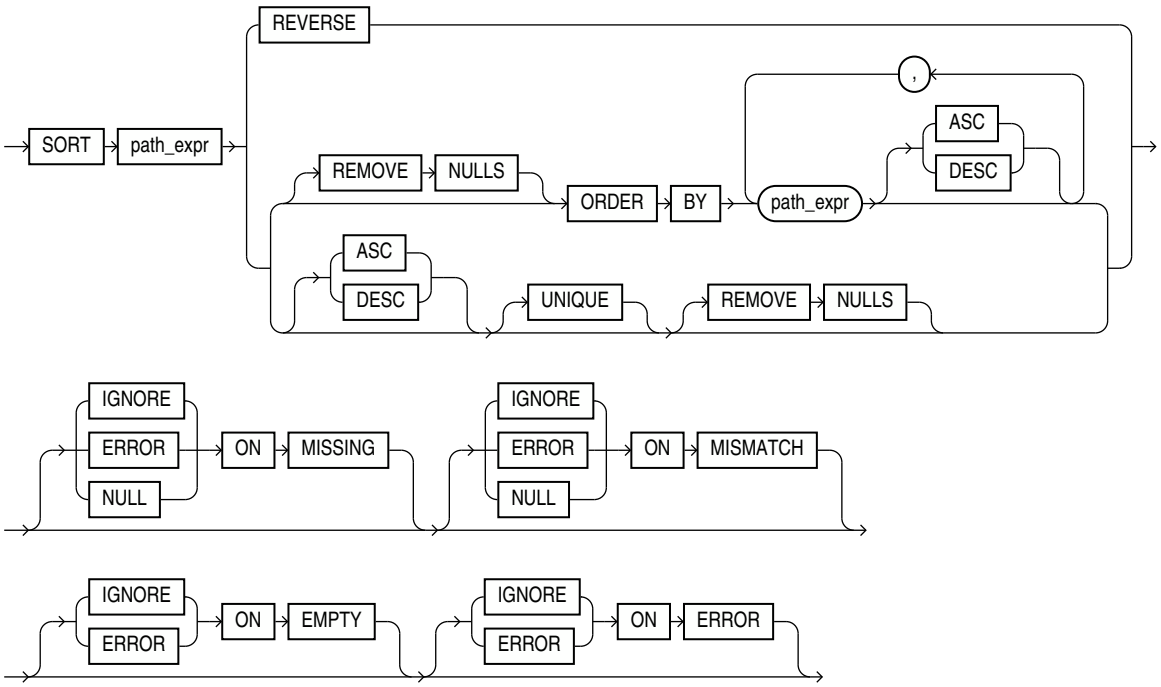




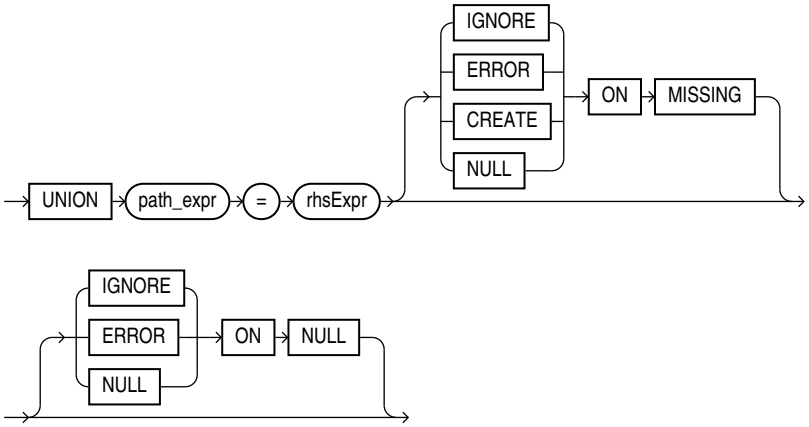


rhs_expr::=

***sort_op*::=**

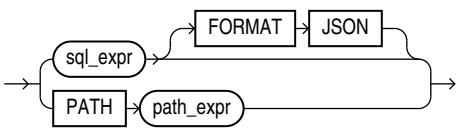


***union_op*::=**



rhs_expr::=

***rhs_expr* ::=**



Purpose

JSON_TRANSFORM modifies JSON documents. You specify operations to perform and SQL/JSON path expressions that target the places to modify. The operations are applied to the input data in the order specified: each operation acts on the data that results from applying all of the preceding operations.

JSON_TRANSFORM either succeeds completely or not at all. If any of the specified operations raises an error, then none of the operations take effect. JSON_TRANSFORM returns the original data changed according to the operations specified.

You can use the JSON_TRANSFORM within the UPDATE statement to modify documents in a JSON column.

You can use it in a SELECT list, to modify the selected documents. The modified documents can be returned or processed further.

JSON_TRANSFORM can accept as input, and return as output, any SQL data type that supports JSON data: JSON, VARCHAR2, CLOB, or BLOB. Note that data type JSON is available only if database initialization parameter compatible is 20 or greater.

The default return (output) data type is the same as the input data type.

> **See Also:**
>
> *Oracle SQL Function JSON_TRANSFORM of the JSON Developer's Guide* for a full discussion with examples.

JSON_TRANSFORM Operations

- Use ADD_SET to add missing value to an array, as if adding an element to a set.
- Use APPEND to append the values that are specified by the RHS to the array that is targeted by the LHS path expression.

  APPEND has the effect of INSERT for an array position of last+1.

  An error is raised if the LHS path expression targets an existing field whose value is not an array.

  If the RHS targets an array then the LHS array is updated by appending the elements of the RHS array to it, in order.
- Use CASE to set conditions to perform a sequence of JSON_TRANSFORM operations.

  This is a control operation that conditionally applies other operations, which in turn can modify data.

  The syntax is keyword CASE followed by one or more WHEN clauses, followed optionally by an ELSE clause, followed by END.

  A WHEN clause is keyword WHEN followed by a path expression, followed by a THEN clause.

  The path expression contains a filter condition, which checks for the existence of some data.

  A THEN or an ELSE clause is keyword THEN or ELSE, respectively, followed by parentheses (()) containing zero or more JSON_TRANSFORM operations.

The operations of a `THEN` clause are performed if the condition of its `WHEN` clause is satisfied. The operations of the optional `ELSE` clause are performed if the condition of no `WHEN` clause is satisfied.

The syntax of the `JSON_TRANSFORM CASE` operation is thus essentially the same as an Oracle SQL searched `CASE` expression, except that it is the predicate that is tested and the resulting effect of each `THEN`/`ELSE` branch.

For SQL, the predicate tested is a SQL comparison. For `JSON_TRANSFORM`, the predicate is a path expression that checks for the existence of some data. (The check is essentially done using `JSON_EXISTS`.)

For SQL, each `THEN`/`ELSE` branch holds a SQL expression to evaluate, and its value is returned as the result of the `CASE` expression. For json_transform, each `THEN`/`ELSE` branch holds a (parenthesized) sequence of `JSON_TRANSFORM` operations, which are performed in order.

The conditional path expressions of the `WHEN` clauses are tested in order, until one succeeds (those that follow are not tested). The `THEN` operations for the successful `WHEN` test are then performed, in order.

- Use `COPY` to replace the elements of the array that is targeted by the LHS path expression with the values that are specified by the RHS. An error is raised if the LHS path expression does not target an array. The operation can accept a sequence of multiple values matched by the RHS path expression.
- `INSERT` Insert the value of the specified SQL expression at the location that's targeted by the specified path expression that follows the equal sign (=), which must be either the field of an object or an array position (otherwise, an error is raised). By default, an error is raised if a targeted object field already exists.

  `INSERT` for an object field has the effect of `SET` with clause `CREATE ON MISSING` (default for `SET`), except that the default behavior for `ON EXISTING` is `ERROR`, not `REPLACE`.)

  You can specify an array position past the current end of an array. In that case, the array is lengthened to accommodate insertion of the value at the indicated position, and the intervening positions are filled with JSON null values.

  For example, if the input JSON data is `{"a":["b"]}` then `INSERT '$.a[3]'=42` returns `{"a":["b", null, null 42]}` as the modified data. The elements at array positions 1 and 2 are null.
- Use `INTERSECT` to remove all elements of the array that is targeted by the LHS path expression that are not equal to any value specified by the RHS. Remove any duplicate elements. Note that this is a set operation. The order of all array elements is undefined after the operation.
- Use `MERGE` to add specified fields (name and value) matched by the RHS path expression to the object that is targeted by the LHS path expression. Ignore any fields specified by the RHS that are already in the targeted LHS object. If the same field is specified more than once by the RHS then use only the last one in the sequence of matches.
- Use `MINUS` to remove all elements of the array that is targeted by the LHS path expression that are equal to a value specified by the RHS. Remove any duplicate elements. Note that this is a set operation. The order of all array elements is undefined after the operation.
- `KEEP` removes all parts of the input data that are not targeted by at least one of the specified path expressions. A topmost object or array is not removed, it is emptied and becomes an empty object ({}) or array ([]).
- Use `NESTED PATH` to define a scope (a particular part of your data) in which to apply a sequence of operations.

- Use `PREPEND` to prepend the values that are specified by the RHS to the array that is targeted by the LHS path expression. The operation can accept a sequence of multiple values matched by the RHS path expression.

  An error is raised if the LHS path expression targets an existing field whose value is not an array.

  When prepending a single value, `PREPEND` has the effect of `INSERT` for an array position of 0.

  If the RHS targets an array then the LHS array is updated by prepending the elements of the RHS array to it, in order.

- `REMOVE` Remove the input data that's targeted by the specified path expression. An error is raised if you try to remove all of the data, for example you cannot use `REMOVE '$'`. By default, no error is raised if the targeted data does not exist (`IGNORE ON MISSING`).

- Use `REMOVE_SET` to remove all occurrences of a value from an array, as if removing an element from a set.

- `RENAME` renames the field that is targeted by the specified path expression to the value of the SQL expression that follows the equal sign (=). By default, no error is raised if the targeted field does not exist (`IGNORE ON MISSING`).

- `REPLACE` replaces the data that's targeted by the specified path expression with the value of the specified SQL expression that follows the equal sign (=). By default, no error is raised if the targeted data does not exist (`IGNORE ON MISSING`).

  `REPLACE` has the effect of `SET` with clause `IGNORE ON MISSING`.

- `SET` Set what the LHS specifies to the value specified by what follows the equal sign (=). The LHS can be either a SQL/JSON variable or a path expression that targets data. If the RHS is a SQL expression then its value is assigned to the LHS variable. When the LHS specifies a path expression, the default behavior is to replace existing targeted data with the new value, or insert the new value at the targeted location if the path expression matches nothing. (See operator `INSERT` about inserting an array element past the end of the array.)

- When the LHS specifies a SQL/JSON variable, the variable is dynamically assigned to whatever is specified by the RHS. (The variable is created if it does not yet exist.) The variable continues to have that value until it is set to a different value by a subsequent `SET` operation (in the same `JSON_TRANSFORM` invocation).

  If the RHS is a path expression then its targeted data is assigned to the variable.

  Setting a variable is a control operation; it can affect how subsequent operations modify data, but it does not, itself, directly modify data.

  When the LHS specifies a path expression, the default behavior is like that of SQL `UPSERT`: replace existing targeted data with the new value, or insert the new value at the targeted location if the path expression matches nothing. (See operator `INSERT` about inserting an array element past the end of the array.)

- `SORT` sorts the elements of the array targeted by the specified path. The result includes all elements of the array (none are dropped); the only possible change is that they are reordered.

- Use `UNION` to add the values specified by the RHS to the array that is targeted by the LHS path expression. Remove any duplicate elements. The operation can accept a sequence of multiple values matched by the RHS path expression. Note that this is a set operation. The order of all array elements is undefined after the operation.

TYPE Clause

For a full discussion of `STRICT` and `LAX` syntax see *About Strict and Lax JSON Syntax*, and *TYPE Clause for SQL Functions and Conditions*

JSON_passing_clause

You can use *`JSON_passing_clause`* to specify SQL bindings of bind variables to SQL/JSON variables similar to the `JSON_EXISTS` condition and the SQL/JSON query functions.

JSON_TRANSFORM_returning_clause

After you specify the operations you can use *`JSON_TRANSFORM_returning_clause`* to specify the return data type.

Examples

Example 1 : Update a JSON Column with a Timestamp

```
UPDATE t SET jcol = JSON_TRANSFORM(jcol, SET '$.lastUpdated' = SYSTIMESTAMP)
```

Example 2 : Remove a Social Security Number before Shipping JSON to a Client

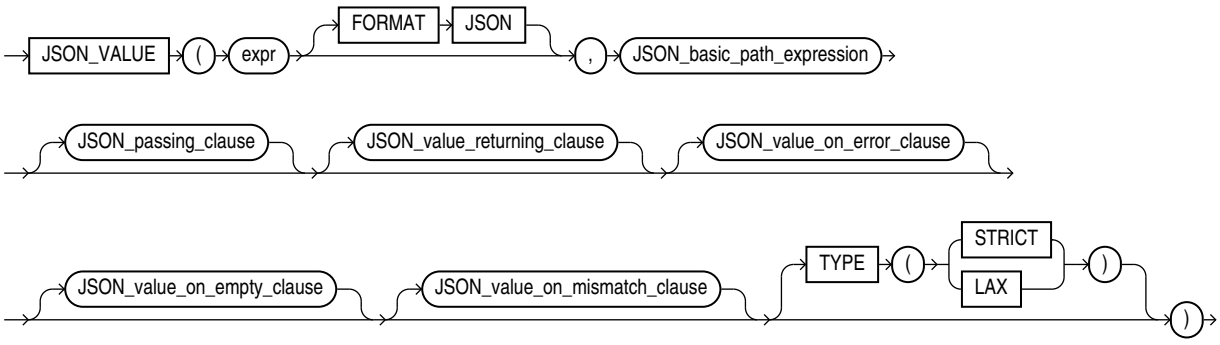
```
SELECT JSON_TRANSFORM (jcol, REMOVE '$.ssn') FROM t WHERE …
```

JSON_TRANSFORM_returning_clause

If the input data is JSON, then the output data type is also JSON. For all other input types, the default output data type is `VARCHAR2(4000)`.

JSON_VALUE

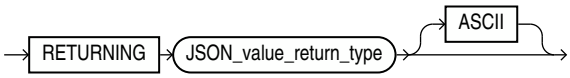
Syntax



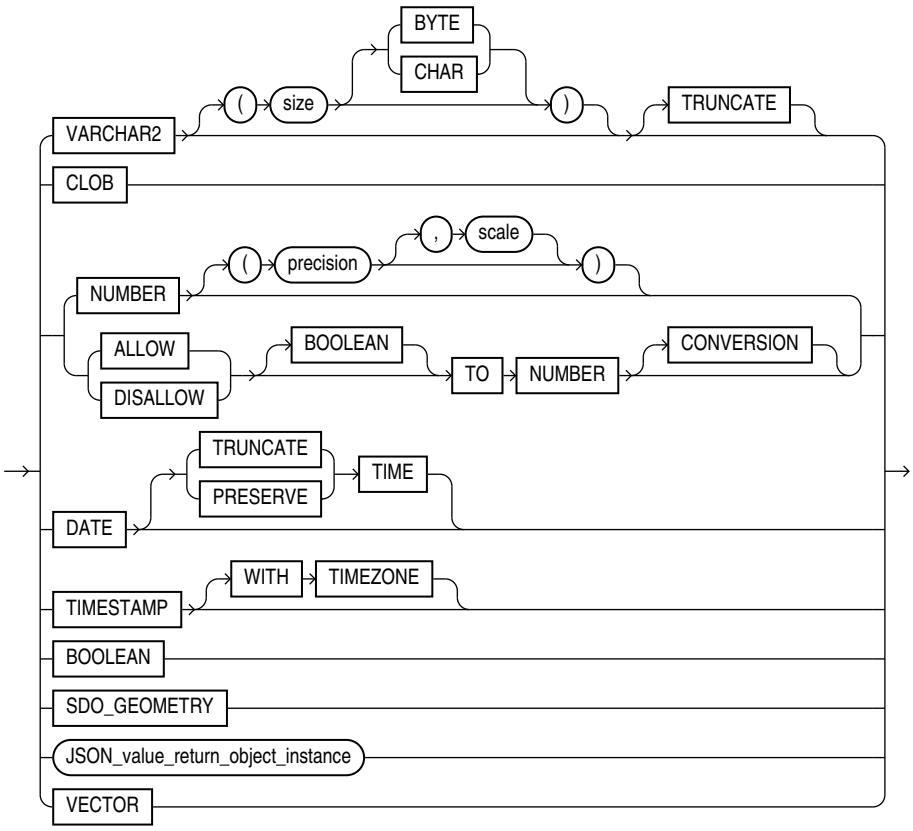
JSON_basic_path_expression::=

(*`JSON_basic_path_expression`*: See *SQL/JSON Path Expressions*)

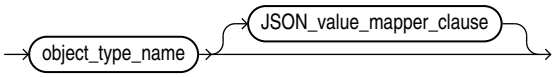
JSON_value_returning_clause::=



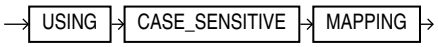
***JSON_value_return_type*::=**



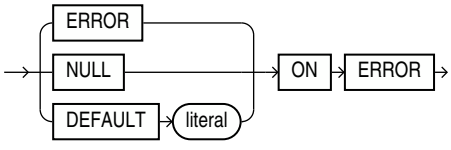
***JSON_value_return_object_instance* ::=**



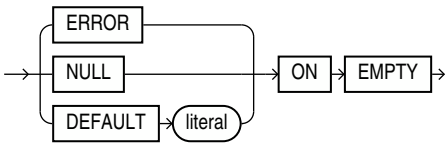
***JSON_value_mapper_clause* ::=**



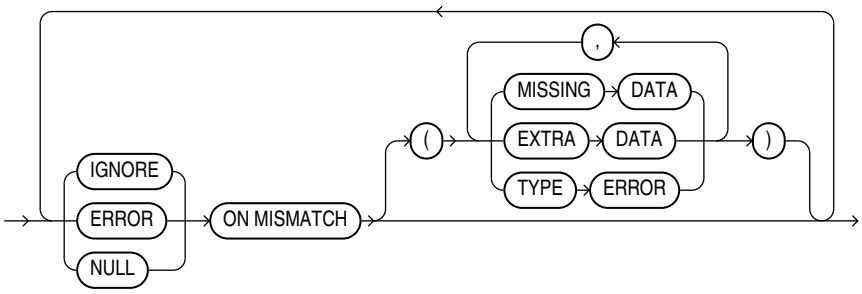
***JSON_value_on_error_clause*::=**



***JSON_value_on_empty_clause*::=**



***JSON_value_on_mismatch_clause*::=**



Purpose

SQL/JSON function `JSON_VALUE` selects JSON data and returns a SQL scalar or an instance of a user-defined SQL object type or SQL collection type (varray, nested table)

> **See Also:**
>
> - *JSON TRANSFORM* of the *JSON Developer's Guide.*
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the value returned by this function when it is a character value

expr

The first argument to `JSON_VALUE` *`expr`* is a SQL expression that returns an instance of a scalar SQL data type (that is, not an object or collection data type). A scalar value returned from`JSON_VALUE` can be of any of these data types: `BINARY_DOUBLE`, `BINARY_FLOAT`, `BOOLEAN`, `CHAR`, `CLOB`, `DATE`, `INTERVAL DAY TO SECOND`, `INTERVAL YEAR TO MONTH`, `NCHAR`, `NCLOB`, `NVARCHAR2`, `NUMBER`, `RAW1`, `SDO_GEOMETRY`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `VARCHAR2`, and `VECTOR`.

If *`expr`* is not a text literal of well-formed JSON data using strict or lax syntax, then the function returns null by default. You can use the *`JSON_value_on_error_clause`* to override this default behavior. Refer to the *JSON_value_on_error_clause*.

FORMAT JSON

You must specify `FORMAT JSON` if *`expr`* is a column of data type `BLOB`.

JSON_basic_path_expression

Use this clause to specify a SQL/JSON path expression. The function uses the path expression to evaluate *expr* and find a scalar JSON value that matches, or satisfies, the path expression. The path expression must be a text literal. See *Oracle Database JSON Developer's Guide* for the full semantics of *JSON_basic_path_expression*.

JSON_value_returning_clause

Use this clause to specify the data type and format of the value returned by this function.

RETURNING

Use the RETURNING clause to specify the data type of the return value. If you omit this clause, then JSON_VALUE returns a value of type VARCHAR2(4000).

JSON_value_return_type ::=

You can use *JSON_value_return_type* to specify the following data types:

- VARCHAR2[(*size* [BYTE,CHAR])]

  If you specify this data type, then the scalar value returned by this function can be a character or number value. A number value will be implicitly converted to a VARCHAR2. When specifying the VARCHAR2 data type elsewhere in SQL, you are required to specify a size. However, in this clause you can omit the size. In this case, JSON_VALUE returns a value of type VARCHAR2(4000).

  Specify the optional TRUNCATE clause immediately after VARCHAR2(N) to truncate the return value to N characters, if the return value is greater than N characters.

  Notes on the TRUNCATE clause :

  – If the string value is too long, then ORA-40478 is raised.

  – If TRUNCATE is present, and the return value is not a character type, then a compile time error is raised.

  – If TRUNCATE is present with FORMAT JSON, then the return value may contain data that is not syntactically correct JSON.

  – TRUNCATE does not work with EXISTS.

- CLOB

  Specify this data type to return a character large object containing single-byte or multi-byte characters.

- NUMBER[(*precision* [, *scale*])]

  If you specify this data type, then the scalar value returned by this function must be a number value. The scalar value returned can also be a JSON Boolean value. Note however, that returning NUMBER for a JSON Boolean value is deprecated.

- DATE

  If you specify this data type, then the scalar value returned by this function must be a character value that can be implicitly converted to a DATE data type. If the JSON input represents a date with a time component, specify DATE PRESERVE TIME to retain the time component. If you do not want to retain the time component, specify DATE TRUNCATE TIME.

If you specify neither `PRESERVE TIME` nor `TRUNCATE TIME`, the time component is not preserved.

- `TIMESTAMP`

  If you specify this data type, then the scalar value returned by this function must be a character value that can be implicitly converted to a `TIMESTAMP` data type.

- `TIMESTAMP WITH TIME ZONE`

  If you specify this data type, then the scalar value returned by this function must be a character value that can be implicitly converted to a `TIMESTAMP WITH TIME ZONE` data type.

- `BOOLEAN`

  Specify `BOOLEAN` to return true, false, or unknown.

- `SDO_GEOMETRY`

  This data type is used for Oracle Spatial and Graph data. If you specify this data type, then *`expr`* must evaluate to a text literal containing GeoJSON data, which is a format for encoding geographic data in JSON. If you specify this data type, then the scalar value returned by this function must be an object of type `SDO_GEOMETRY`.

- *JSON_value_return_object_instance*

  If `JSON_VALUE` targets a JSON object, and you specify a user-defined SQL object type as the return type, then `JSON_VALUE` returns an instance of that object type in *object_type_name*.

  For examples see Using *JSON_VALUE To Instantiate a User-Defined Object Type Instance* of the *JSON Developer's Guide.*

> **See Also:**
>
> - *SQL/JSON Function JSON_VALUE* for a conceptual understanding in the *JSON Developer's Guide.*
> - Refer to "Data Types " for more information on the preceding data types.
> - If the data type is not large enough to hold the return value, then this function returns null by default. You can use the *`JSON_value_on_error_clause`* to override this default behavior. Refer to the *JSON_value_on_error_clause*.

ASCII

Specify `ASCII` to automatically escape any non-ASCII Unicode characters in the return value, using standard ASCII Unicode escape sequences.

JSON_value_on_error_clause

Use this clause to specify the value returned by this function when the following errors occur:

- *`expr`* is not well-formed JSON data using strict or lax JSON syntax
- A nonscalar value is found when the JSON data is evaluated using the SQL/JSON path expression
- No match is found when the JSON data is evaluated using the SQL/JSON path expression. You can override the behavior for this type of error by specifying the *`JSON_value_on_empty_clause`*.

- The return value data type is not large enough to hold the return value

You can specify the following clauses:

- `NULL ON ERROR` - Returns null when an error occurs. This is the default.
- `ERROR ON ERROR` - Returns the appropriate Oracle error when an error occurs.
- `DEFAULT literal ON ERROR` - Returns *literal* when an error occurs. The data type of *literal* must match the data type of the value returned by this function.

JSON_value_on_empty_clause

Use this clause to specify the value returned by this function if no match is found when the JSON data is evaluated using the SQL/JSON path expression. This clause allows you to specify a different outcome for this type of error than the outcome specified with the *JSON_value_on_error_clause*.

You can specify the following clauses:

- `NULL ON EMPTY` - Returns null when no match is found.
- `ERROR ON EMPTY` - Returns the appropriate Oracle error when no match is found.
- `DEFAULT literal ON EMPTY` - Returns *literal* when no match is found. The data type of *literal* must match the data type of the value returned by this function.

If you omit this clause, then the *JSON_value_on_error_clause* determines the value returned when no match is found.

JSON_value_on_mismatch_clause

The *JSON_value_on_mismatch_clause* applies when a type conversion fails, for example when you try to convert a JSON number to a SQL date.

If the return type of `JSON_VALUE` is a SQL scalar like `NUMBER` or `DATE` , then `ON MISMATCH` applies for all type conversion errors - no further specification is required. `ERROR` and `NULL` are valid options.

Example 1

```
select json_value( '{a:"cat"}','$.a.number()'  NULL ON
EMPTY
 ERROR ON  MISMATCH  DEFAULT -1 ON ERROR ) from dual;
 ORA-01722: invalid number
```

If the return type is an object type, then `ON MISMATCH` can be further specified with `MISSING DATA`, `EXTRA DATA` and `TYPE ERROR`. You can use it generally to apply to all error cases, or you can use it case by case by specifying different `ON MISMATCH` clauses for each case.

Example 2

```
IGNORE ON MISMATCH (EXTRA DATA)


ERROR ON MISMATCH ( MISSING DATA, TYPE ERROR)
```

The option `IGNORE` is only valid when the return type is an object type.

TYPE Clause

For a full discussion of `STRICT` and `LAX` syntax see *About Strict and Lax JSON Syntax*, and *TYPE Clause for SQL Functions and Conditions*

Examples

The following query returns the value of the member with property name `a`. Because the `RETURNING` clause is not specified, the value is returned as a `VARCHAR2(4000)` data type:

```
SELECT JSON_VALUE('{a:100}', '$.a') AS value
  FROM DUAL;

VALUE
-----
100
```

The following query returns the value of the member with property name `a`. Because the `RETURNING NUMBER` clause is specified, the value is returned as a `NUMBER` data type:

```
SELECT JSON_VALUE('{a:100}', '$.a' RETURNING NUMBER) AS value
  FROM DUAL;

     VALUE
----------
       100
```

The following query returns the value of the member with property name `b`, which is in the value of the member with property name `a`:

```
SELECT JSON_VALUE('{a:{b:100}}', '$.a.b') AS value
  FROM DUAL;

VALUE
-----
100
```

The following query returns the value of the member with property name `d` in any object:

```
SELECT JSON_VALUE('{a:{b:100}, c:{d:200}, e:{f:300}}', '$.*.d') AS value
  FROM DUAL;

VALUE
-----
200
```

The following query returns the value of the first element in an array:

```
SELECT JSON_VALUE('[0, 1, 2, 3]', '$[0]') AS value
  FROM DUAL;

VALUE
-----
0
```

The following query returns the value of the third element in an array. The array is the value of the member with property name `a`.

```
SELECT JSON_VALUE('{a:[5, 10, 15, 20]}', '$.a[2]') AS value
  FROM DUAL;
```

```
VALUE
-----
15
```

The following query returns the value of the member with property name `a` in the second object in an array:

```
SELECT JSON_VALUE('[{a:100}, {a:200}, {a:300}]', '$[1].a') AS value
  FROM DUAL;

VALUE
-----
200
```

The following query returns the value of the member with property name `c` in any object in an array:

```
SELECT JSON_VALUE('[{a:100}, {b:200}, {c:300}]', '$[*].c') AS value
  FROM DUAL;

VALUE
-----
300
```

The following query attempts to return the value of the member that has property name `lastname`. However, such a member does not exist in the specified JSON data, resulting in no match. Because the `ON ERROR` clause is not specified, the statement uses the default `NULL ON ERROR` and returns null.

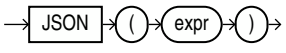
```
SELECT JSON_VALUE('{firstname:"John"}', '$.lastname') AS "Last Name"
  FROM DUAL;

Last Name
---------
```

The following query results in an error because it attempts to return the value of the member with property name `lastname`, which does not exist in the specified JSON. Because the `ON ERROR` clause is specified, the statement returns the specified text literal.

```
SELECT JSON_VALUE('{firstname:"John"}', '$.lastname'
                  DEFAULT 'No last name found' ON ERROR) AS "Last Name"
  FROM DUAL;

Last Name
---------
No last name found
```

JSON Type Constructor

Syntax



Purpose

The JSON data type constructor, JSON, takes as input a textual JSON value (a scalar, object, or array), parses it, and returns the value as an instance of JSON type. Alternatively, the input can be an instance of SQL type VECTOR, a user-defined PL/SQL type, or a SQL aggregate type.

You can use the JSON data type constructor `JSON` to parse textual JSON input (a scalar, object, or array), and return it as an instance of type `JSON`.

Input values must pass the `IS JSON` test. Input values that fail the `IS JSON` test are rejected with a syntax error.

To filter out duplicate input values, you must run the `IS JSON (WITH UNIQUE KEYS)` check on the textual JSON input before using the `JSON` constructor.

Prerequisites

You can use the constructor `JSON` only if database initialization parameter `compatible` is atleast `20`.

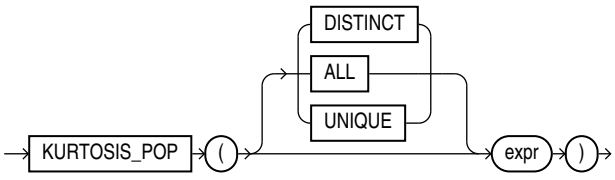
> **See Also:**
>
> JSON Data Type Constructor of the *JSON Developer's Guide*.

expr

The input in `expr` must be a syntactically valid textual representation of type `VARCHAR2`, `CLOB` and `BLOB`. It can also be a literal SQL string. A SQL `NULL` input value results in a JSON type instance of SQL `NULL`.

KURTOSIS_POP

Syntax



Purpose

The population kurtosis function `KURTOSIS_POP` is primarily used to determine the characteristics of outliers in a given distribution.

NULL values in `expr` are ignored.

Returns NULL if all rows in the group have NULL `expr` values.

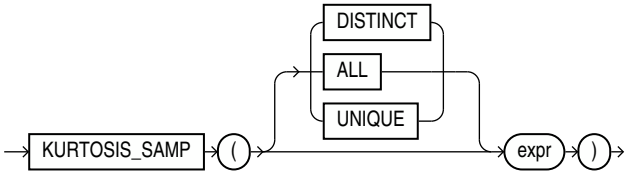
Returns 0 if there are one or two rows in `expr`.

For a given set of values, the result of population kurtosis (`KURTOSIS_POP`) and sample kurtosis (`KURTOSIS_SAMP`) are always deterministic. However, the values of `KURTOSIS_POP` and

KURTOSIS_SAMP differ. As the number of values in the data set increases, the difference between the computed values of KURTOSIS_SAMP and KURTOSIS_POP decreases.

KURTOSIS_SAMP

Syntax



Purpose

The sample kurtosis function KURTOSIS_SAMP is primarily used to determine the characteristics of outliers in a given distribution.

NULL values in expr are ignored.

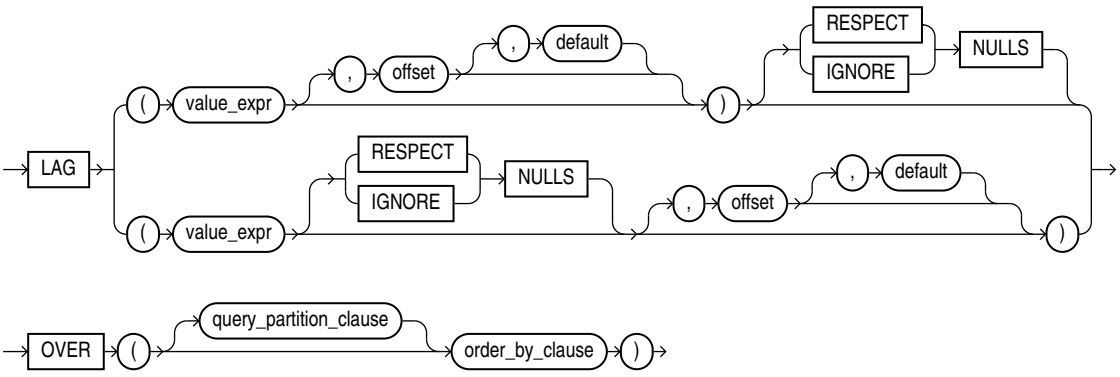
Returns NULL if all rows in the group have NULL expr values.

Returns 0 if there are one or two rows in expr.

For a given set of values, the result of sample kurtosis (KURTOSIS_SAMP) and population kurtosis (KURTOSIS_POP) are always deterministic. However, the values of KURTOSIS_SAMP and KURTOSIS_POP differ. As the number of values in the data set increases, the difference between the computed values of KURTOSIS_SAMP and KURTOSIS_POP decreases.

LAG

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *value_expr*

Purpose

LAG is an analytic function. It provides access to more than one row of a table at the same time without a self join. Given a series of rows returned from a query and a position of the cursor, LAG provides access to a row at a given physical offset prior to that position.

For the optional *offset* argument, specify an integer that is greater than zero. If you do not specify *offset*, then its default is 1. The optional *default* value is returned if the offset goes beyond the scope of the window. If you do not specify *default*, then its default is null.

{RESPECT | IGNORE} NULLS determines whether null values of *value_expr* are included in or eliminated from the calculation. The default is RESPECT NULLS.

You cannot nest analytic functions by using LAG or any other analytic function for *value_expr*. However, you can use other built-in function expressions for *value_expr*.

> **See Also:**
>
> - "About SQL Expressions " for information on valid forms of *expr* and LEAD
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of LAG when it is a character value

Examples

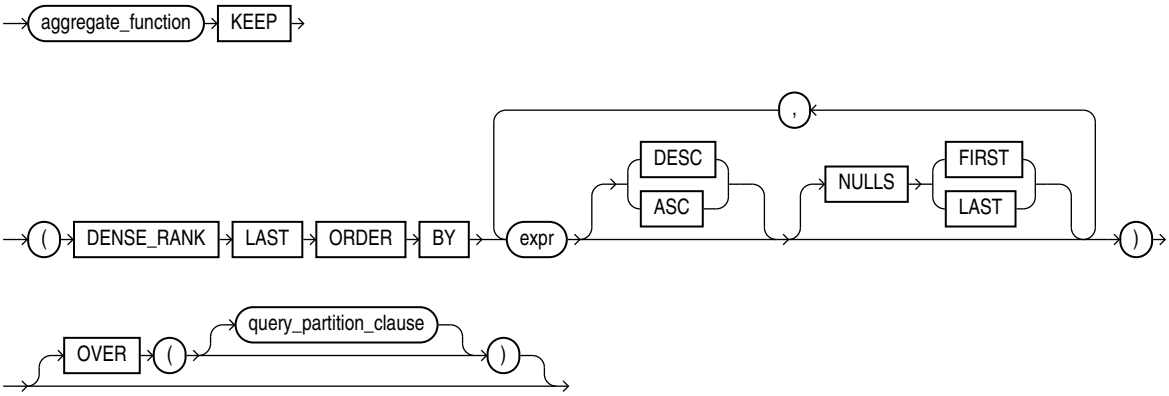
The following example provides, for each purchasing clerk in the employees table, the salary of the employee hired just before:

```
SELECT hire_date, last_name, salary,
       LAG(salary, 1, 0) OVER (ORDER BY hire_date) AS prev_sal
  FROM employees
  WHERE job_id = 'PU_CLERK'
  ORDER BY hire_date;

HIRE_DATE LAST_NAME                     SALARY   PREV_SAL
--------- ------------------------- ---------- ----------
18-MAY-03 Khoo                            3100          0
24-JUL-05 Tobias                          2800       3100
24-DEC-05 Baida                           2900       2800
15-NOV-06 Himuro                          2600       2900
10-AUG-07 Colmenares                      2500       2600
```

LAST

Syntax

***last*::=**



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions of the *query_partitioning_clause*

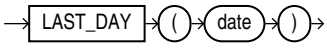
Purpose

`FIRST` and `LAST` are very similar functions. Both are aggregate and analytic functions that operate on a set of values from a set of rows that rank as the `FIRST` or `LAST` with respect to a given sorting specification. If only one row ranks as `FIRST` or `LAST`, then the aggregate operates on the set with only one element.

Refer to FIRST for complete information on this function and for examples of its use.

LAST_DAY

Syntax



Purpose

`LAST_DAY` returns the date of the last day of the month that contains *date*. The last day of the month is defined by the session parameter `NLS_CALENDAR`. The return type is always `DATE`, regardless of the data type of *date*.

Examples

The following statement determines how many days are left in the current month.

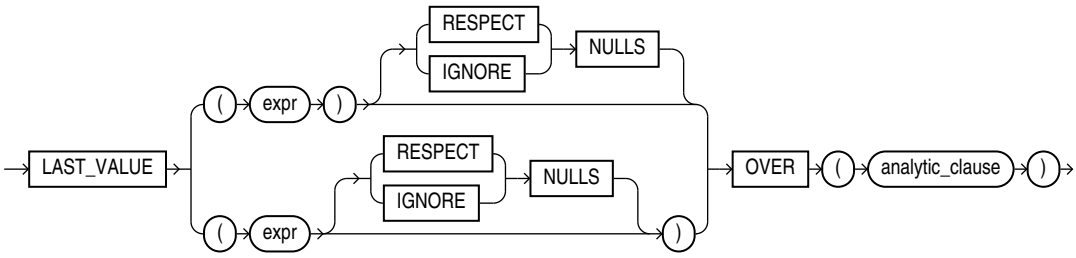
```
SELECT SYSDATE,
       LAST_DAY(SYSDATE) "Last",
       LAST_DAY(SYSDATE) - SYSDATE "Days Left"
  FROM DUAL;

SYSDATE   Last       Days Left
--------- --------- ----------
30-MAY-09 31-MAY-09          1
```

The following example adds `5` months to the hire date of each employee to give an evaluation date:

```
SELECT last_name, hire_date,
       TO_CHAR(ADD_MONTHS(LAST_DAY(hire_date), 5)) "Eval Date"
  FROM employees
  ORDER BY last_name, hire_date;

LAST_NAME                 HIRE_DATE Eval Date
------------------------- --------- ---------
Abel                      11-MAY-04 31-OCT-04
Ande                      24-MAR-08 31-AUG-08
Atkinson                  30-OCT-05 31-MAR-06
Austin                    25-JUN-05 30-NOV-05
Baer                      07-JUN-02 30-NOV-02
Baida                     24-DEC-05 31-MAY-06
Banda                     21-APR-08 30-SEP-08
Bates                     24-MAR-07 31-AUG-07
. . .
```

LAST_VALUE

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *expr*

Purpose

LAST_VALUE is an analytic function that is useful for data densification. It returns the last value in an ordered set of values.

> **Note:**
>
> The two forms of this syntax have the same behavior. The top branch is the ANSI format, which Oracle recommends for ANSI compatibility.

{RESPECT | IGNORE} NULLS determines whether null values of *expr* are included in or eliminated from the calculation. The default is RESPECT NULLS. If the last value in the set is null, then the function returns NULL unless you specify IGNORE NULLS. If you specify IGNORE NULLS, then LAST_VALUE returns the last non-null value in the set, or NULL if all values are null. Refer to "Using Partitioned Outer Joins: Examples" for an example of data densification.

You cannot nest analytic functions by using LAST_VALUE or any other analytic function for *expr*. However, you can use other built-in function expressions for *expr*. Refer to "About SQL Expressions " for information on valid forms of *expr*.

If you omit the *windowing_clause* of the *analytic_clause*, it defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. This default sometimes returns an unexpected value, because the last value in the window is at the bottom of the window, which is not fixed. It keeps changing as the current row changes. For expected results, specify the *windowing_clause* as RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. Alternatively, you can specify the *windowing_clause* as RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Examples

The following example returns, for each row, the hire date of the employee earning the lowest salary:

```
SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
         OVER (ORDER BY salary DESC ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
               FOLLOWING) AS lv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);

EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                       24000 17-JUN-03 13-JAN-01
        101 Kochhar                    17000 21-SEP-05 13-JAN-01
        102 De Haan                    17000 13-JAN-01 13-JAN-01
```

This example illustrates the nondeterministic nature of the LAST_VALUE function. Kochhar and De Haan have the same salary, so they are in adjacent rows. Kochhar appears first because the rows in the subquery are ordered by hire_date. However, if the rows are ordered by hire_date in descending order, as in the next example, then the function returns a different value:

```
SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
         OVER (ORDER BY salary DESC ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
               FOLLOWING) AS lv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date DESC);

EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                       24000 17-JUN-03 21-SEP-05
        102 De Haan                    17000 13-JAN-01 21-SEP-05
        101 Kochhar                    17000 21-SEP-05 21-SEP-05
```

The following two examples show how to make the LAST_VALUE function deterministic by ordering on a unique key. By ordering within the function by both salary and the unique key employee_id, you can ensure the same result regardless of the ordering in the subquery.

```
SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
         OVER (ORDER BY salary DESC, employee_id ROWS BETWEEN UNBOUNDED PRECEDING
               AND UNBOUNDED FOLLOWING) AS lv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);

EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                       24000 17-JUN-03 13-JAN-01
        101 Kochhar                    17000 21-SEP-05 13-JAN-01
        102 De Haan                    17000 13-JAN-01 13-JAN-01


SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
         OVER (ORDER BY salary DESC, employee_id ROWS BETWEEN UNBOUNDED PRECEDING
               AND UNBOUNDED FOLLOWING) AS lv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date DESC);

EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                       24000 17-JUN-03 13-JAN-01
        101 Kochhar                    17000 21-SEP-05 13-JAN-01
        102 De Haan                    17000 13-JAN-01 13-JAN-01
```

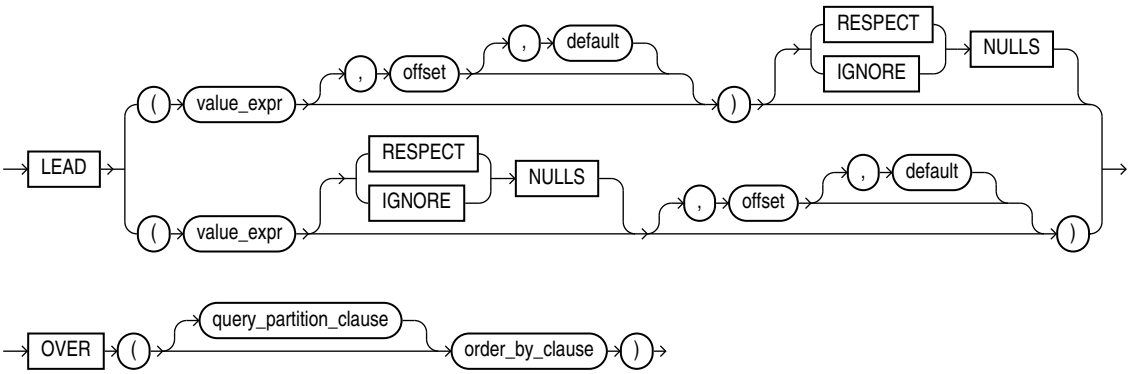
The following two examples show that the LAST_VALUE function is deterministic when you use a logical offset (RANGE instead of ROWS). When duplicates are found for the ORDER BY expression, the LAST_VALUE is the highest value of *expr*:

```
SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
         OVER (ORDER BY salary DESC RANGE BETWEEN UNBOUNDED PRECEDING AND
               UNBOUNDED FOLLOWING) AS lv
```

```
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date);

EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                       24000 17-JUN-03 21-SEP-05
        102 De Haan                    17000 13-JAN-01 21-SEP-05
        101 Kochhar                    17000 21-SEP-05 21-SEP-05


SELECT employee_id, last_name, salary, hire_date,
       LAST_VALUE(hire_date)
         OVER (ORDER BY salary DESC RANGE BETWEEN UNBOUNDED PRECEDING AND
               UNBOUNDED FOLLOWING) AS lv
  FROM (SELECT * FROM employees
          WHERE department_id = 90
          ORDER BY hire_date DESC);

EMPLOYEE_ID LAST_NAME                 SALARY HIRE_DATE LV
----------- ------------------------- ---------- --------- ---------
        100 King                       24000 17-JUN-03 21-SEP-05
        102 De Haan                    17000 13-JAN-01 21-SEP-05
        101 Kochhar                    17000 21-SEP-05 21-SEP-05
```

LEAD

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *value_expr*

Purpose

`LEAD` is an analytic function. It provides access to more than one row of a table at the same time without a self join. Given a series of rows returned from a query and a position of the cursor, `LEAD` provides access to a row at a given physical offset beyond that position.

If you do not specify *offset*, then its default is 1. The optional *default* value is returned if the offset goes beyond the scope of the table. If you do not specify *default*, then its default value is null.

{RESPECT | IGNORE} NULLS determines whether null values of *value_expr* are included in or eliminated from the calculation. The default is RESPECT NULLS.

You cannot nest analytic functions by using LEAD or any other analytic function for *value_expr*. However, you can use other built-in function expressions for *value_expr*.

> **See Also:**
>
> - "About SQL Expressions " for information on valid forms of *expr* and LAG
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of LEAD when it is a character value

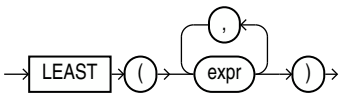
Examples

The following example provides, for each employee in Department 30 in the employees table, the hire date of the employee hired just after:

```
SELECT hire_date, last_name,
       LEAD(hire_date, 1) OVER (ORDER BY hire_date) AS "NextHired"
  FROM employees
  WHERE department_id = 30
  ORDER BY hire_date;

HIRE_DATE LAST_NAME                 Next Hired
--------- ------------------------- ----------
07-DEC-02 Raphaely                  18-MAY-03
18-MAY-03 Khoo                      24-JUL-05
24-JUL-05 Tobias                    24-DEC-05
24-DEC-05 Baida                     15-NOV-06
15-NOV-06 Himuro                    10-AUG-07
10-AUG-07 Colmenares
```

LEAST

Syntax



Purpose

LEAST returns the least of a list of one or more expressions. Oracle Database uses the first *expr* to determine the return type. If the first *expr* is numeric, then Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type before the comparison, and returns that data type. If the first *expr* is not numeric, then each *expr* after the first is implicitly converted to the data type of the first *expr* before the comparison.

Oracle Database compares each *expr* using nonpadded comparison semantics. The comparison is binary by default and is linguistic if the NLS_COMP parameter is set to LINGUISTIC and the NLS_SORT parameter has a setting other than BINARY. Character comparison is based on the numerical codes of the characters in the database character set and is performed on whole strings treated as one sequence of bytes, rather than character by character. If the value returned by this function is character data, then its data type is VARCHAR2 if the first *expr* is a character data type and NVARCHAR2 if the first *expr* is a national character data type.

See Also:

- "Data Type Comparison Rules " for more information on character comparison
- Table 2-9 for more information on implicit conversion and "Floating-Point Numbers " for information on binary-float comparison semantics
- "GREATEST ", which returns the greatest of a list of one or more expressions
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation LEAST uses to compare character values for *expr*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Examples

The following statement selects the string with the least value:

```
SELECT LEAST('HARRY','HARRIOT','HAROLD') "Least"
  FROM DUAL;

Least
------
HAROLD
```

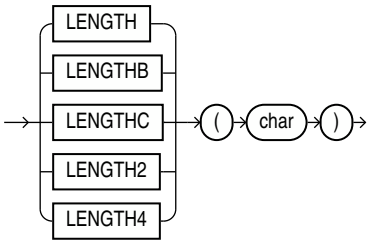
In the following statement, the first argument is numeric. Oracle Database determines that the argument with the highest numeric precedence is the third argument, converts the remaining arguments to the data type of the third argument, and returns the least value as that data type:

```
SELECT LEAST (1, '2.1', '.000832') "Least"
  FROM DUAL;

Least
-------
.000832
```

LENGTH

Syntax

***length*::=**



Purpose

The `LENGTH` functions return the length of *`char`*. `LENGTH` calculates length using characters as defined by the input character set. `LENGTHB` uses bytes instead of characters. `LENGTHC` uses Unicode complete characters. `LENGTH2` uses UCS2 code points. `LENGTH4` uses UCS4 code points.

`char` can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The exceptions are `LENGTHC`, `LENGTH2`, and `LENGTH4`, which do not allow *`char`* to be a `CLOB` or `NCLOB`. The return value is of data type `NUMBER`. If *`char`* has data type `CHAR`, then the length includes all trailing blanks. If *`char`* is null, then this function returns null.

For more on character length see the following:

- *Oracle Database Globalization Support Guide*
- *Oracle Database SecureFiles and Large Objects Developer's Guide*

Restriction on LENGTHB

The `LENGTHB` function is supported for single-byte LOBs only. It cannot be used with `CLOB` and `NCLOB` data in a multibyte character set.

Examples

The following example uses the `LENGTH` function using a single-byte database character set:

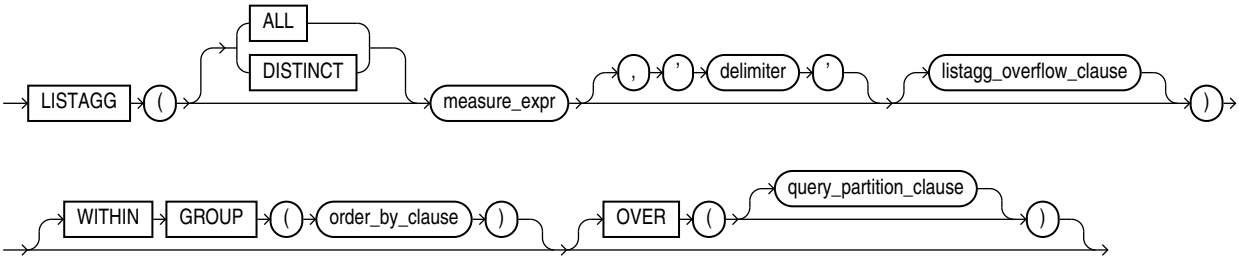
```
SELECT LENGTH('CANDIDE') "Length in characters"
  FROM DUAL;

Length in characters
--------------------
                   7
```

The next example assumes a double-byte database character set.

```
SELECT LENGTHB ('CANDIDE') "Length in bytes"
  FROM DUAL;

Length in bytes
---------------
             14
```

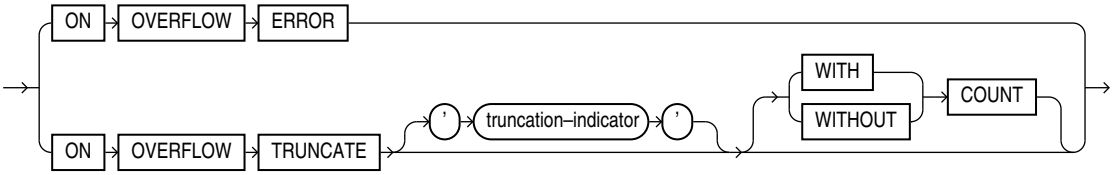
LISTAGG

Syntax



(*listagg_overflow_clause*::=, *order_by_clause*::=, *query_partition_clause*::=)

***listagg_overflow_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions of the `ORDER BY` clause and `OVER` clause

Purpose

For a specified measure, `LISTAGG` orders data within each group specified in the `ORDER BY` clause and then concatenates the values of the measure column.

- As a single-set aggregate function, `LISTAGG` operates on all rows and returns a single output row.
- As a group-set aggregate, the function operates on and returns an output row for each group defined by the `GROUP BY` clause.
- As an analytic function, `LISTAGG` partitions the query result set into groups based on one or more expression in the *`query_partition_clause`*.

The arguments to the function are subject to the following rules:

- The `ALL` keyword is optional and is provided for semantic clarity.
- The *`measure_expr`* is the measure column and can be any expression. Null values in the measure column are ignored.

- The *delimiter* designates the string that is to separate the measure column values. This clause is optional and defaults to NULL.

  If measure_expr is of type RAW, then the delimiter must be of type RAW. You can achieve this by specifying the delimiter as a character string that can be implicitly converted to RAW, or by explicitly converting the delimiter to RAW, for example, using the UTL_RAW.CAST_TO_RAW function.

- The *order_by_clause* determines the order in which the concatenated values are returned. The function is deterministic only if the ORDER BY column list achieved unique ordering.
- If you specify *order_by_clause*, you must also specify WITHIN GROUP and vice versa. These two clauses must be specified together or not at all.

The DISTINCT keyword removes duplicate values from the list.

If the measure column is of type RAW, then the return data type is RAW. Otherwise, the return data type is VARCHAR2.

The maximum length of the return data type depends on the value of the MAX_STRING_SIZE initialization parameter. If MAX_STRING_SIZE = EXTENDED, then the maximum length is 32767 bytes for the VARCHAR2 and RAW data types. If MAX_STRING_SIZE = STANDARD, then the maximum length is 4000 bytes for the VARCHAR2 data type and 2000 bytes for the RAW data type. A final delimiter is not included when determining if the return value fits in the return data type.

> **See Also:**
>
> - Extended Data Types for more information on the MAX_STRING_SIZE initialization parameter
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of LISTAGG
> - *Database Data Warehousing Guide* for details.

listagg_overflow_clause

This clause controls how the function behaves when the return value exceeds the maximum length of the return data type.

ON OVERFLOW ERROR If you specify this clause, then the function returns an ORA-01489 error. This is the default.

ON OVERFLOW TRUNCATE If you specify this clause, then the function returns a truncated list of measure values.

- The *truncation_indicator* designates the string that is to be appended to the truncated list of measure values. If you omit this clause, then the truncation indicator is an ellipsis (. . .).

  If *measure_expr* is of type RAW, then the truncation indicator must be of type RAW. You can achieve this by specifying the truncation indicator as a character string that can be implicitly converted to RAW, or by explicitly converting the truncation indicator to RAW, for example, using the UTL_RAW.CAST_TO_RAW function.

- If you specify WITH COUNT, then after the truncation indicator, the database appends the number of truncated values, enclosed in parentheses. In this case, the database truncates

enough measure values to allow space in the return value for a final delimiter, the truncation indicator, and 24 characters for the number value enclosed in parentheses.

- If you specify `WITHOUT COUNT`, then the database omits the number of truncated values from the return value. In this case, the database truncates enough measure values to allow space in the return value for a final delimiter and the truncation indicator.

If you do not specify `WITH COUNT` or `WITHOUT COUNT`, then the default is `WITH COUNT`.

Aggregate Examples

The following single-set aggregate example lists all of the employees in Department 30 in the `hr.employees` table, ordered by hire date and last name:

```
SELECT LISTAGG(last_name, '; ')
         WITHIN GROUP (ORDER BY hire_date, last_name) "Emp_list",
       MIN(hire_date) "Earliest"
  FROM employees
  WHERE department_id = 30;

Emp_list                                                    Earliest
----------------------------------------------------------- ---------
Raphaely; Khoo; Tobias; Baida; Himuro; Colmenares           07-DEC-02
```

The following group-set aggregate example lists, for each department ID in the `hr.employees` table, the employees in that department in order of their hire date:

```
SELECT department_id "Dept.",
       LISTAGG(last_name, '; ') WITHIN GROUP (ORDER BY hire_date) "Employees"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

Dept. Employees
------ ------------------------------------------------------------
    10 Whalen
    20 Hartstein; Fay
    30 Raphaely; Khoo; Tobias; Baida; Himuro; Colmenares
    40 Mavris
    50 Kaufling; Ladwig; Rajs; Sarchand; Bell; Mallin; Weiss; Davie
       s; Marlow; Bull; Everett; Fripp; Chung; Nayer; Dilly; Bissot
       ; Vollman; Stiles; Atkinson; Taylor; Seo; Fleaur; Matos; Pat
       el; Walsh; Feeney; Dellinger; McCain; Vargas; Gates; Rogers;
        Mikkilineni; Landry; Cabrio; Jones; Olson; OConnell; Sulliv
       an; Mourgos; Gee; Perkins; Grant; Geoni; Philtanker; Markle
    60 Austin; Hunold; Pataballa; Lorentz; Ernst
    70 Baer
. . .
```

The following example is identical to the previous example, except it contains the `ON OVERFLOW TRUNCATE` clause. For the purpose of this example, assume that the maximum length of the return value is an artificially small number of 200 bytes. Because the list of employees for department 50 exceeds 200 bytes, the list is truncated and appended with a final delimiter '`; `', the specified truncation indicator '`. . .`', and the number of truncated values '`(23)`'.

```
SELECT department_id "Dept.",
       LISTAGG(last_name, '; ' ON OVERFLOW TRUNCATE '...')
               WITHIN GROUP (ORDER BY hire_date) "Employees"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;
```

```
Dept. Employees
------ ------------------------------------------------------------
    10 Whalen
    20 Hartstein; Fay
    30 Raphaely; Khoo; Tobias; Baida; Himuro; Colmenares
    40 Mavris
    50 Kaufling; Ladwig; Rajs; Sarchand; Bell; Mallin; Weiss; Davie
       s; Marlow; Bull; Everett; Fripp; Chung; Nayer; Dilly; Bissot
       ; Vollman; Stiles; Atkinson; Taylor; Seo; Fleaur; ... (23)
    70 Baer
. . .
```

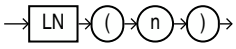
Analytic Example

The following analytic example shows, for each employee hired earlier than September 1, 2003, the employee's department, hire date, and all other employees in that department also hired before September 1, 2003:

```
SELECT department_id "Dept", hire_date "Date", last_name "Name",
       LISTAGG(last_name, '; ') WITHIN GROUP (ORDER BY hire_date, last_name)
         OVER (PARTITION BY department_id) as "Emp_list"
  FROM employees
  WHERE hire_date < '01-SEP-2003'
  ORDER BY "Dept", "Date", "Name";

 Dept Date      Name            Emp_list
----- --------- --------------- --------------------------------------------
   30 07-DEC-02 Raphaely        Raphaely; Khoo
   30 18-MAY-03 Khoo            Raphaely; Khoo
   40 07-JUN-02 Mavris          Mavris
   50 01-MAY-03 Kaufling        Kaufling; Ladwig
   50 14-JUL-03 Ladwig          Kaufling; Ladwig
   70 07-JUN-02 Baer            Baer
   90 13-JAN-01 De Haan         De Haan; King
   90 17-JUN-03 King            De Haan; King
  100 16-AUG-02 Faviet          Faviet; Greenberg
  100 17-AUG-02 Greenberg       Faviet; Greenberg
  110 07-JUN-02 Gietz           Gietz; Higgins
  110 07-JUN-02 Higgins         Gietz; Higgins
```

LN

Syntax



Purpose

`LN` returns the natural logarithm of *n*, where *n* is greater than 0.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

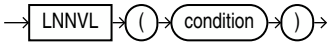
Examples

The following example returns the natural logarithm of 95:

```
SELECT LN(95) "Natural log of 95"
  FROM DUAL;

Natural log of 95
-----------------
       4.55387689
```

LNNVL

Syntax



Purpose

`LNNVL` provides a concise way to evaluate a condition when one or both operands of the condition may be null. The function can be used in the `WHERE` clause of a query, or as the `WHEN` condition in a searched `CASE` expression. It takes as an argument a condition and returns `TRUE` if the condition is `FALSE` or `UNKNOWN` and `FALSE` if the condition is `TRUE`. `LNNVL` can be used anywhere a scalar expression can appear, even in contexts where the `IS` [`NOT`] `NULL`, `AND`, or `OR` conditions are not valid but would otherwise be required to account for potential nulls.

Oracle Database sometimes uses the `LNNVL` function internally in this way to rewrite `NOT IN` conditions as `NOT EXISTS` conditions. In such cases, output from `EXPLAIN PLAN` shows this operation in the plan table output. The *`condition`* can evaluate any scalar values but cannot be a compound condition containing `AND`, `OR`, or `BETWEEN`.

The table that follows shows what `LNNVL` returns given that `a` = 2 and `b` is null.

Condition	Truth of Condition	LNNVL Return Value
a = 1	FALSE	TRUE
a = 2	TRUE	FALSE
a IS NULL	FALSE	TRUE
b = 1	UNKNOWN	TRUE
b IS NULL	TRUE	FALSE
a = b	UNKNOWN	TRUE

Examples

Suppose that you want to know the number of employees with commission rates of less than 20%, including employees who do not receive commissions. The following query returns only employees who actually receive a commission of less than 20%:

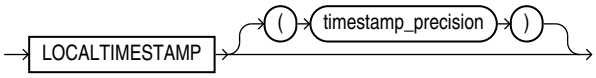
```
SELECT COUNT(*)
  FROM employees
  WHERE commission_pct < .2;

  COUNT(*)
----------
        11
```

To include the 72 employees who receive no commission at all, you could rewrite the query using the `LNNVL` function as follows:

```
SELECT COUNT(*)
  FROM employees
  WHERE LNNVL(commission_pct >= .2);

  COUNT(*)
----------
        83
```

LOCALTIMESTAMP

Syntax



Purpose

`LOCALTIMESTAMP` returns the current date and time in the session time zone in a value of data type `TIMESTAMP`. The difference between this function and `CURRENT_TIMESTAMP` is that `LOCALTIMESTAMP` returns a `TIMESTAMP` value while `CURRENT_TIMESTAMP` returns a `TIMESTAMP WITH TIME ZONE` value.

The optional argument *`timestamp_precision`* specifies the fractional second precision of the time value returned.

> **See Also:**
>
> CURRENT_TIMESTAMP , "TIMESTAMP Data Type ", and "TIMESTAMP WITH TIME ZONE Data Type "

Examples

This example illustrates the difference between `LOCALTIMESTAMP` and `CURRENT_TIMESTAMP`:

```
ALTER SESSION SET TIME_ZONE = '-5:00';
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;

CURRENT_TIMESTAMP                   LOCALTIMESTAMP
---------------------------------------------------------------------------
04-APR-00 01.27.18.999220 PM -05:00  04-APR-00 01.27.19 PM

ALTER SESSION SET TIME_ZONE = '-8:00';
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;
```

```
CURRENT_TIMESTAMP                         LOCALTIMESTAMP
--------------------------------------    ----------------------------
04-APR-00 10.27.45.132474 AM -08:00       04-APR-00 10.27.451 AM
```

When you use the `LOCALTIMESTAMP` with a format mask, take care that the format mask matches the value returned by the function. For example, consider the following table:

```
CREATE TABLE local_test (col1 TIMESTAMP WITH LOCAL TIME ZONE);
```

The following statement fails because the mask does not include the `TIME ZONE` portion of the return type of the function:

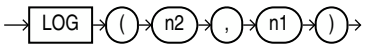
```
INSERT INTO local_test
  VALUES (TO_TIMESTAMP(LOCALTIMESTAMP, 'DD-MON-RR HH.MI.SSXFF'));
```

The following statement uses the correct format mask to match the return type of `LOCALTIMESTAMP`:

```
INSERT INTO local_test
  VALUES (TO_TIMESTAMP(LOCALTIMESTAMP, 'DD-MON-RR HH.MI.SSXFF PM'));
```

LOG

Syntax



Purpose

`LOG` returns the logarithm, base *n2*, of *n1*. The base *n2* can be any positive value other than 0 or 1 and *n1* can be any positive value.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If any argument is `BINARY_FLOAT` or `BINARY_DOUBLE`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns `NUMBER`.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

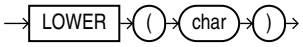
Examples

The following example returns the log of 100:

```
SELECT LOG(10,100) "Log base 10 of 100"
  FROM DUAL;

Log base 10 of 100
------------------
                 2
```

LOWER

Syntax



Purpose

`LOWER` returns *`char`*, with all letters lowercase. *`char`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The return value is the same data type as *`char`*. The database sets the case of the characters based on the binary mapping defined for the underlying character set. For linguistic-sensitive lowercase, refer to NLS_LOWER .

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `LOWER`

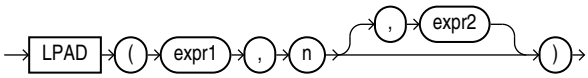
Examples

The following example returns a string in lowercase:

```
SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase"
  FROM DUAL;

Lowercase
--------------------
mr. scott mcmillan
```

LPAD

Syntax



Purpose

`LPAD` returns *`expr1`*, left-padded to length *`n`* characters with the sequence of characters in *`expr2`*. This function is useful for formatting the output of a query.

Both *`expr1`* and *`expr2`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` data type if *`expr1`* is a character data type, `NVARCHAR2` if *`expr1`* is a national character data type, and a LOB if *`expr1`* is a LOB data type. The string returned is in the same character set as *`expr1`*. The argument *`n`* must be a `NUMBER` integer or a value that can be implicitly converted to a `NUMBER` integer.

If you do not specify *expr2*, then the default is a single blank. If *expr1* is longer than *n*, then this function returns the portion of *expr1* that fits in *n*.

The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of LPAD

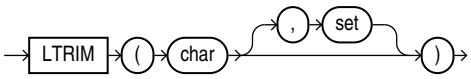
Examples

The following example left-pads a string with the asterisk (*) and period (.) characters:

```
SELECT LPAD('Page 1',15,'*.') "LPAD example"
  FROM DUAL;

LPAD example
---------------
*.*.*.*.*Page 1
```

LTRIM

Syntax



Purpose

LTRIM removes from the left end of *char* all of the characters contained in *set*. If you do not specify *set*, then it defaults to a single blank. Oracle Database begins scanning *char* from its first character and removes all characters that appear in *set* until reaching a character not in *set* and then returns the result.

Both *char* and *set* can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is of VARCHAR2 data type if *char* is a character data type, NVARCHAR2 if *char* is a national character data type, and a LOB if *char* is a LOB data type.

> **See Also:**
>
> - RTRIM
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `LTRIM` uses to compare characters from *`set`* with characters from *`char`*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

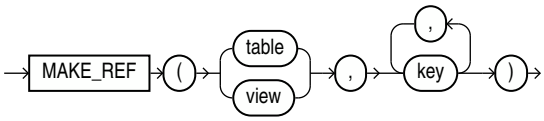
Examples

The following example trims all the left-most occurrences of less than sign (<), greater than sign (>) , and equal sign (=) from a string:

```
SELECT LTRIM('<=====>BROWNING<=====>', '<>=') "LTRIM Example"
  FROM DUAL;

LTRIM Example
---------------
BROWNING<=====>
```

MAKE_REF

Syntax



Purpose

`MAKE_REF` creates a `REF` to a row of an object view or a row in an object table whose object identifier is primary key based. This function is useful, for example, if you are creating an object view

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for more information about object views and DEREF

Examples

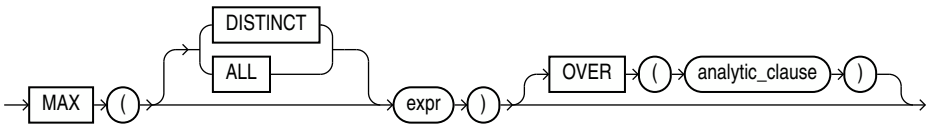
The sample schema `oe` contains an object view `oc_inventories` based on `inventory_typ`. The object identifier is `product_id`. The following example creates a `REF` to the row in the `oc_inventories` object view with a `product_id` of 3003:

```
SELECT MAKE_REF (oc_inventories, 3003)
  FROM DUAL;

MAKE_REF(OC_INVENTORIES,3003)
```

```
-------------------------------------------------------------------
00004A038A0046857C14617141109EE0340800208254360000001426010001000 1
00290090606002A00078401FE0000000B03C21F040000000000000000000000000 0
0000000000
```

MAX

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`MAX` returns maximum value of *`expr`*. You can use it as an aggregate or analytic function.

> **See Also:**
>
> - "About SQL Expressions " for information on valid forms of *`expr`*, "Floating-Point Numbers " for information on binary-float comparison semantics, and "Aggregate Functions "
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `MAX` uses to compare character values for *`expr`*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Aggregate Example

The following example determines the highest salary in the `hr.employees` table:

```
SELECT MAX(salary) "Maximum"
  FROM employees;

   Maximum
----------
     24000
```

Analytic Examples

The following example calculates, for each employee, the highest salary of the employees reporting to the same manager as the employee.

```
SELECT manager_id, last_name, salary,
       MAX(salary) OVER (PARTITION BY manager_id) AS mgr_max
```

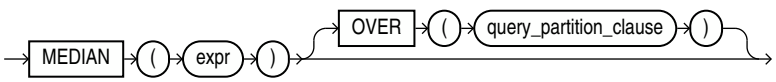
```
  FROM employees
  ORDER BY manager_id, last_name, salary;

MANAGER_ID LAST_NAME                 SALARY    MGR_MAX
---------- ------------------------- ---------- ----------
       100 Cambrault                  11000      17000
       100 De Haan                    17000      17000
       100 Errazuriz                  12000      17000
       100 Fripp                       8200      17000
       100 Hartstein                  13000      17000
       100 Kaufling                    7900      17000
       100 Kochhar                    17000      17000
. . .
```

If you enclose this query in the parent query with a predicate, then you can determine the employee who makes the highest salary in each department:

```
SELECT manager_id, last_name, salary
  FROM (SELECT manager_id, last_name, salary,
               MAX(salary) OVER (PARTITION BY manager_id) AS rmax_sal
          FROM employees)
  WHERE salary = rmax_sal
  ORDER BY manager_id, last_name, salary;

MANAGER_ID LAST_NAME                 SALARY
---------- ------------------------- ----------
       100 De Haan                    17000
       100 Kochhar                    17000
       101 Greenberg                  12008
       101 Higgins                    12008
       102 Hunold                      9000
       103 Ernst                       6000
       108 Faviet                      9000
       114 Khoo                        3100
       120 Nayer                       3200
       120 Taylor                      3200
       121 Sarchand                    4200
       122 Chung                       3800
       123 Bell                        4000
       124 Rajs                        3500
       145 Tucker                     10000
       146 King                       10000
       147 Vishney                    10500
       148 Ozer                       11500
       149 Abel                       11000
       201 Fay                         6000
       205 Gietz                       8300
           King                       24000

22 rows selected.
```

MEDIAN

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`MEDIAN` is an inverse distribution function that assumes a continuous distribution model. It takes a numeric or datetime value and returns the middle value or an interpolated value that would be the middle value once the values are sorted. Nulls are ignored in the calculation.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If you specify only *expr*, then the function returns the same data type as the numeric data type of the argument. If you specify the `OVER` clause, then Oracle Database determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

The result of `MEDIAN` is computed by first ordering the rows. Using `N` as the number of rows in the group, Oracle calculates the row number (`RN`) of interest with the formula `RN` = (`1` + (`0.5`*(`N`-`1`)). The final result of the aggregate function is computed by linear interpolation between the values from rows at row numbers `CRN` = `CEILING`(`RN`) and `FRN` = `FLOOR`(`RN`).

The final result will be:

```
if (CRN = FRN = RN) then
   (value of expression from row at RN)
else
   (CRN - RN) * (value of expression for row at FRN) +
   (RN - FRN) * (value of expression for row at CRN)
```

You can use `MEDIAN` as an analytic function. You can specify only the *query_partition_clause* in its `OVER` clause. It returns, for each row, the value that would fall in the middle among a set of values within each partition.

Compare this function with these functions:

- PERCENTILE_CONT , which returns, for a given percentile, the value that corresponds to that percentile by way of interpolation. `MEDIAN` is the specific case of `PERCENTILE_CONT` where the percentile value defaults to 0.5.
- PERCENTILE_DISC , which is useful for finding values for a given percentile without interpolation.

Aggregate Example

The following query returns the median salary for each department in the `hr.employees` table:

```
SELECT department_id, MEDIAN(salary)
  FROM employees
  GROUP BY department_id
```

```
  ORDER BY department_id;

DEPARTMENT_ID MEDIAN(SALARY)
------------- --------------
           10           4400
           20           9500
           30           2850
           40           6500
           50           3100
           60           4800
           70          10000
           80           8900
           90          17000
          100           8000
          110          10154
                        7000
```

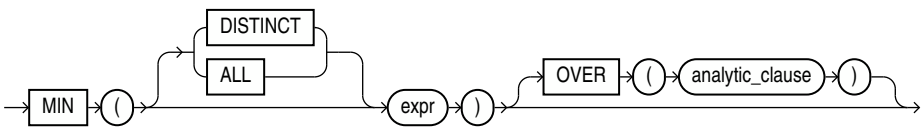
Analytic Example

The following query returns the median salary for each manager in a subset of departments in the `hr.employees` table:

```
SELECT manager_id, employee_id, salary,
       MEDIAN(salary) OVER (PARTITION BY manager_id) "Median by Mgr"
  FROM employees
  WHERE department_id > 60
  ORDER BY manager_id, employee_id;

MANAGER_ID EMPLOYEE_ID     SALARY Median by Mgr
---------- ----------- ---------- -------------
       100         101      17000         13500
       100         102      17000         13500
       100         145      14000         13500
       100         146      13500         13500
       100         147      12000         13500
       100         148      11000         13500
       100         149      10500         13500
       101         108      12008         12008
       101         204      10000         12008
       101         205      12008         12008
       108         109       9000          7800
       108         110       8200          7800
       108         111       7700          7800
       108         112       7800          7800
       108         113       6900          7800
       145         150      10000          8500
       145         151       9500          8500
       145         152       9000          8500
. . .
```

MIN

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`MIN` returns minimum value of *`expr`*. You can use it as an aggregate or analytic function.

> **See Also:**
>
> - "About SQL Expressions " for information on valid forms of *`expr`*, "Floating-Point Numbers " for information on binary-float comparison semantics, and "Aggregate Functions "
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `MIN` uses to compare character values for *`expr`*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Aggregate Example

The following statement returns the earliest hire date in the `hr.employees` table:

```
SELECT MIN(hire_date) "Earliest"
  FROM employees;

Earliest
---------
13-JAN-01
```

Analytic Example

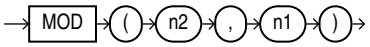
The following example determines, for each employee, the employees who were hired on or before the same date as the employee. It then determines the subset of employees reporting to the same manager as the employee, and returns the lowest salary in that subset.

```
SELECT manager_id, last_name, hire_date, salary,
       MIN(salary) OVER(PARTITION BY manager_id ORDER BY hire_date
         RANGE UNBOUNDED PRECEDING) AS p_cmin
  FROM employees
  ORDER BY manager_id, last_name, hire_date, salary;

MANAGER_ID LAST_NAME                 HIRE_DATE     SALARY     P_CMIN
---------- ------------------------- --------- ---------- ----------
       100 Cambrault                 15-OCT-07      11000       6500
       100 De Haan                   13-JAN-01      17000      17000
       100 Errazuriz                 10-MAR-05      12000       7900
       100 Fripp                     10-APR-05       8200       7900
       100 Hartstein                 17-FEB-04      13000       7900
       100 Kaufling                  01-MAY-03       7900       7900
       100 Kochhar                   21-SEP-05      17000       7900
       100 Mourgos                   16-NOV-07       5800       5800
       100 Partners                  05-JAN-05      13500       7900
       100 Raphaely                  07-DEC-02      11000      11000
       100 Russell                   01-OCT-04      14000       7900
```

. . .

MOD

Syntax



Purpose

`MOD` returns the remainder of *n2* divided by *n1*. Returns *n2* if *n1* is 0.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

Examples

The following example returns the remainder of 11 divided by 4:

```
SELECT MOD(11,4) "Modulus"
  FROM DUAL;

   Modulus
----------
         3
```

This function behaves differently from the classical mathematical modulus function, if the product of *n1* and *n2* is negative. The classical modulus can be expressed using the `MOD` function with this formula:

```
n2 - n1 * FLOOR(n2/n1)
```

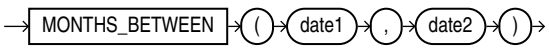
The following table illustrates the difference between the `MOD` function and the classical modulus:

n2	n1	MOD(n2,n1)	Classical Modulus
11	4	3	3
11	-4	3	-1
-11	4	-3	1
-11	-4	-3	-3

> **See Also:**
>
> FLOOR (number) and REMAINDER , which is similar to `MOD`, but uses `ROUND` in its formula instead of `FLOOR`

MONTHS_BETWEEN

Syntax



Purpose

`MONTHS_BETWEEN` returns number of months between dates *`date1`* and *`date2`*. The month and the last day of the month are defined by the parameter `NLS_CALENDAR`. If *`date1`* is later than *`date2`*, then the result is positive. If *`date1`* is earlier than *`date2`*, then the result is negative. If *`date1`* and *`date2`* are either the same days of the month or both last days of months, then the result is always an integer. Otherwise Oracle Database calculates the fractional portion of the result based on a 31-day month and considers the difference in time components *`date1`* and *`date2`*.

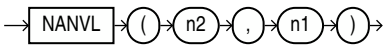
Examples

The following example calculates the months between two dates:

```
SELECT MONTHS_BETWEEN
       (TO_DATE('02-02-1995','MM-DD-YYYY'),
        TO_DATE('01-01-1995','MM-DD-YYYY') ) "Months"
  FROM DUAL;

    Months
----------
1.03225806
```

NANVL

Syntax



Purpose

The `NANVL` function is useful only for floating-point numbers of type `BINARY_FLOAT` or `BINARY_DOUBLE`. It instructs Oracle Database to return an alternative value *`n1`* if the input value *`n2`* is `NaN` (not a number). If *`n2`* is **not** `NaN`, then Oracle returns *`n2`*.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the

highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion, "Floating-Point Numbers " for information on binary-float comparison semantics, and "Numeric Precedence " for information on numeric precedence

Examples

Using table `float_point_demo` created for TO_BINARY_DOUBLE , insert a second entry into the table:

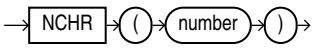
```
INSERT INTO float_point_demo
  VALUES (0,'NaN','NaN');

SELECT *
  FROM float_point_demo;

   DEC_NUM BIN_DOUBLE  BIN_FLOAT
---------- ---------- ----------
   1234.56 1.235E+003 1.235E+003
         0        Nan        Nan
```

The following example returns *bin_float* if it is a number. Otherwise, 0 is returned.

```
SELECT bin_float, NANVL(bin_float,0)
  FROM float_point_demo;

 BIN_FLOAT NANVL(BIN_FLOAT,0)
---------- ------------------
1.235E+003         1.235E+003
       Nan                  0
```

NCHR

Syntax



Purpose

`NCHR` returns the character having the binary equivalent to *number* in the national character set. The value returned is always `NVARCHAR2`. This function is equivalent to using the `CHR` function with the `USING NCHAR_CS` clause.

This function takes as an argument a `NUMBER` value, or any value that can be implicitly converted to `NUMBER`, and returns a character.

> **See Also:**
>
> - CHR
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of NCHR

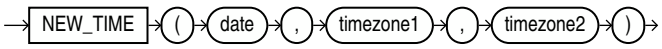
Examples

The following examples return the nchar character 187:

```
SELECT NCHR(187)
  FROM DUAL;

N
-
>

SELECT CHR(187 USING NCHAR_CS)
  FROM DUAL;

C
-
>
```

NEW_TIME

Syntax



Purpose

NEW_TIME returns the date and time in time zone *timezone2* when date and time in time zone *timezone1* are *date*. Before using this function, you must set the NLS_DATE_FORMAT parameter to display 24-hour time. The return type is always DATE, regardless of the data type of *date*.

> **Note:**
>
> This function takes as input only a limited number of time zones. You can have access to a much greater number of time zones by combining the FROM_TZ function and the datetime expression. See FROM_TZ and the example for "Datetime Expressions ".

The arguments *timezone1* and *timezone2* can be any of these text strings:

- AST, ADT: Atlantic Standard or Daylight Time
- BST, BDT: Bering Standard or Daylight Time

- CST, CDT: Central Standard or Daylight Time
- EST, EDT: Eastern Standard or Daylight Time
- GMT: Greenwich Mean Time
- HST, HDT: Alaska-Hawaii Standard Time or Daylight Time.
- MST, MDT: Mountain Standard or Daylight Time
- NST: Newfoundland Standard Time
- PST, PDT: Pacific Standard or Daylight Time
- YST, YDT: Yukon Standard or Daylight Time

Examples

The following example returns an Atlantic Standard time, given the Pacific Standard time equivalent:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';

SELECT NEW_TIME(TO_DATE('11-10-09 01:23:45', 'MM-DD-YY HH24:MI:SS'), 'AST', 'PST')
         "New Date and Time"
  FROM DUAL;

New Date and Time
--------------------
09-NOV-2009 21:23:45
```

NEXT_DAY

Syntax

Purpose

`NEXT_DAY` returns the date of the first weekday named by *`char`* that is later than the date *`date`*. The return type is always `DATE`, regardless of the data type of *`date`*. The argument *`char`* must be a day of the week in the date language of your session, either the full name or the abbreviation. The minimum number of letters required is the number of letters in the abbreviated version. Any characters immediately following the valid abbreviation are ignored. The return value has the same hours, minutes, and seconds component as the argument *`date`*.

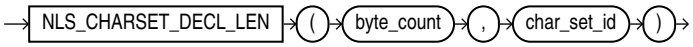
Examples

This example returns the date of the next Tuesday after October 15, 2009:

```
SELECT NEXT_DAY('15-OCT-2009','TUESDAY') "NEXT DAY"
  FROM DUAL;

NEXT DAY
--------------------
20-OCT-2009 00:00:00
```

NLS_CHARSET_DECL_LEN

Syntax



Purpose

`NLS_CHARSET_DECL_LEN` returns the declaration length (in number of characters) of an `NCHAR` column. The *`byte_count`* argument is the width of the column. The *`char_set_id`* argument is the character set ID of the column.

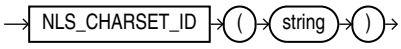
Examples

The following example returns the number of characters that are in a 200-byte column when you are using a multibyte character set:

```
SELECT NLS_CHARSET_DECL_LEN(200, nls_charset_id('ja16eucfixed'))
  FROM DUAL;

NLS_CHARSET_DECL_LEN(200,NLS_CHARSET_ID('JA16EUCFIXED'))
--------------------------------------------------------
                                                     100
```

NLS_CHARSET_ID

Syntax



Purpose

`NLS_CHARSET_ID` returns the character set ID number corresponding to character set name *`string`*. The *`string`* argument is a run-time `VARCHAR2` value. The *`string`* value '`CHAR_CS`' returns the database character set ID number of the server. The *`string`* value '`NCHAR_CS`' returns the national character set ID number of the server.

Invalid character set names return null.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for a list of character sets

Examples

The following example returns the character set ID of a character set:

```
SELECT NLS_CHARSET_ID('ja16euc')
  FROM DUAL;
```

```
NLS_CHARSET_ID('JA16EUC')
-------------------------
                      830
```

NLS_CHARSET_NAME

Syntax

Purpose

`NLS_CHARSET_NAME` returns the name of the character set corresponding to ID number *`number`*. The character set name is returned as a `VARCHAR2` value in the database character set. If *`number`* is not recognized as a valid character set ID, then this function returns null.

This function returns a `VARCHAR2` value.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `NLS_CHARSET_NAME`

Examples

The following example returns the character set corresponding to character set ID number 2:

```
SELECT NLS_CHARSET_NAME(2)
  FROM DUAL;

NLS_CH
------
WE8DEC
```

NLS_COLLATION_ID

Syntax

Purpose

`NLS_COLLATION_ID` takes as its argument a collation name and returns the corresponding collation ID number. Collation IDs are used in the data dictionary tables and in Oracle Call Interface (OCI). Collation names are used in SQL statements and data dictionary views

For *`expr`*, specify the collation name as a `VARCHAR2` value. You can specify a valid named collation or a pseudo-collation, in any combination of uppercase and lowercase letters.

This function returns a `NUMBER` value. If you specify an invalid collation name, then this function returns null.

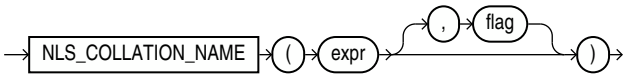
Examples

The following example returns the collation ID of collation `BINARY_CI`:

```
SELECT NLS_COLLATION_ID('BINARY_CI')
  FROM DUAL;

NLS_COLLATION_ID('BINARY_CI')
-----------------------------
                       147455
```

NLS_COLLATION_NAME

Syntax



Purpose

`NLS_COLLATION_NAME` takes as its argument a collation ID number and returns the corresponding collation name. Collation IDs are used in the data dictionary tables and in Oracle Call Interface (OCI). Collation names are used in SQL statements and data dictionary views

For *`expr`*, specify the collation ID as a `NUMBER` value.

This function returns a `VARCHAR2` value. If you specify an invalid collation ID, then this function returns null.

The optional *`flag`* parameter applies only to Unicode Collation Algorithm (UCA) collations. This parameter determines whether the function returns the short form or long form of the collation name. The parameter must be a character expression evaluating to the value `'S'`, `'s'`, `'L'`, or `'l'`, with the following meaning:

- `'S'` or `'s'` – Returns the short form of the collation name
- `'L'` or `'l'` – Returns the long form of the collation name

If you omit *`flag`*, then the default is `'L'`.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for more information on UCA collations
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `NLS_COLLATION_NAME`

Examples

The following example returns the name of the collation corresponding to collation ID number 81919:

```
SELECT NLS_COLLATION_NAME(81919)
  FROM DUAL;

NLS_COLLA
---------
BINARY_AI
```

The following example returns the short form of the name of the UCA collation corresponding to collation ID number 208897:

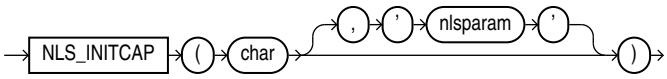
```
SELECT NLS_COLLATION_NAME(208897,'S')
  FROM DUAL;

NLS_COLLATION
-------------
UCA0610_DUCET
```

The following example returns the long form of the name of the UCA collation corresponding to collation ID number 208897:

```
SELECT NLS_COLLATION_NAME(208897,'L')
  FROM DUAL;

NLS_COLLATION_NAME(208897,'L')
--------------------------------------
UCA0610_DUCET_S4_VS_BN_NY_EN_FN_HN_DN_MN
```

NLS_INITCAP

Syntax



Purpose

`NLS_INITCAP` returns *`char`*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

Both *`char`* and *`'nlsparam'`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The string returned is of `VARCHAR2` data type and is in the same character set as *`char`*.

The value of *`'nlsparam'`* can have this form:

```
'NLS_SORT = sort'
```

where *`sort`* is a named collation. The collation handles special linguistic requirements for case conversions. These requirements can result in a return value of a different length than the *`char`*. If you omit `'nlsparam'`, then this function uses the determined collation of the function.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `NLS_INITCAP` , and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

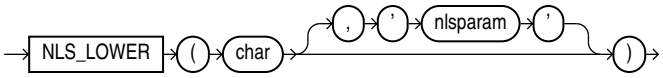
The following examples show how the linguistic sort sequence results in a different return value from the function:

```
SELECT NLS_INITCAP('ijsland') "InitCap"
  FROM DUAL;

InitCap
-------
Ijsland

SELECT NLS_INITCAP('ijsland', 'NLS_SORT = XDutch') "InitCap"
  FROM DUAL;

InitCap
-------
IJsland
```

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information on collations

NLS_LOWER

Syntax



Purpose

`NLS_LOWER` returns *`char`*, with all letters lowercase.

Both *`char`* and *`'nlsparam'`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` data type if *`char`* is a character data type and a LOB if *`char`* is a LOB data type. The return string is in the same character set as *`char`*.

The *`'nlsparam'`* can have the same form and serve the same purpose as in the `NLS_INITCAP` function.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `NLS_LOWER`, and for the collation derivation rules, which define the collation assigned to the character return value of this function

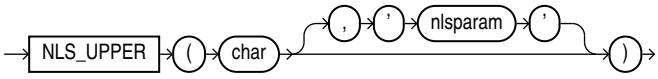
Examples

The following statement returns the lowercase form of the character string 'NOKTASINDA' using the XTurkish linguistic sort sequence. The Turkish uppercase I becoming a small, dotless i.

```
SELECT NLS_LOWER('NOKTASINDA', 'NLS_SORT = XTurkish') "Lowercase"
  FROM DUAL;
```

NLS_UPPER

Syntax



Purpose

`NLS_UPPER` returns *char*, with all letters uppercase.

Both *char* and *'nlsparam'* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` data type if *char* is a character data type and a LOB if *char* is a LOB data type. The return string is in the same character set as *char*.

The *'nlsparam'* can have the same form and serve the same purpose as in the `NLS_INITCAP` function.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `NLS_UPPER`, and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

The following example returns a string with all the letters converted to uppercase:

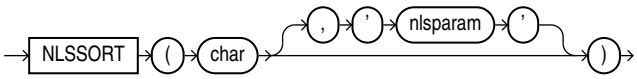
```
SELECT NLS_UPPER('große') "Uppercase"
  FROM DUAL;

Upper
-----
GROßE

SELECT NLS_UPPER('große', 'NLS_SORT = XGerman') "Uppercase"
  FROM DUAL;
```

```
Upperc
------
GROSSE
```

> **See Also:**
>
> NLS_INITCAP

NLSSORT

Syntax



Purpose

`NLSSORT` returns a collation key for the character value *`char`* and an explicitly or implicitly specified collation. A collation key is a string of bytes used to sort *`char`* according to the specified collation. The property of the collation keys is that mutual ordering of two such keys generated for the given collation when compared according to their binary order is the same as mutual ordering of the source character values when compared according to the given collation.

Both *`char`* and '*`nlsparam`*' can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`.

The value of '*`nlsparam`*' must have the form

```
'NLS_SORT = collation'
```

where *`collation`* is the name of a linguistic collation or `BINARY`. `NLSSORT` uses the specified collation to generate the collation key. If you omit '*`nlsparam`*', then this function uses the derived collation of the argument *`char`*. If you specify `BINARY`, then this function returns the *`char`* value itself cast to `RAW` and possibly truncated as described below.

If you specify '*`nlsparam`*', then you can append to the linguistic collation name the suffix `_ai` to request an accent-insensitive collation or `_ci` to request a case-insensitive collation. Refer to *Oracle Database Globalization Support Guide* for more information on accent- and case-insensitive sorting. Using accent-insensitive or case-insensitive collations with the `ORDER BY` query clause is not recommended as it leads to a nondeterministic sort order.

The returned collation key is of `RAW` data type. The length of the collation key resulting from a given *`char`* value for a given collation may exceed the maximum length of the `RAW` value returned by `NLSSORT`. In this case, the behavior of `NLSSORT` depends on the value of the initialization parameter `MAX_STRING_SIZE`. If `MAX_STRING_SIZE = EXTENDED`, then the maximum length of the return value is 32767 bytes. If the collation key exceeds this limit, then the function fails with the error "ORA-12742: unable to create the collation key". This error may also be reported for short input strings if they contain a high percentage of Unicode characters with very high decomposition ratios.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for details of when the ORA-12742 error is reported and how to prevent application availability issues that the error could cause

If MAX_STRING_SIZE = STANDARD, then the maximum length of the return value is 2000 bytes. If the value to be returned exceeds the limit, then NLSSORT calculates the collation key for a maximum prefix, or initial substring, of *char* so that the calculated result does not exceed the maximum length. For monolingual collations, for example FRENCH, the prefix length is typically 1000 characters. For multilingual collations, for example GENERIC_M, the prefix is typically 500 characters. For Unicode Collation Algorithm (UCA) collations, for example UCA0610_DUCET, the prefix is typically 285 characters. The exact length may be lower or higher depending on the collation and the characters contained in *char*.

The behavior when MAX_STRING_SIZE = STANDARD implies that two character values whose collation keys (NLSSORT results) are compared to find the linguistic ordering are considered equal if they do not differ in the prefix even though they may differ at some further character position. Because the NLSSORT function is used implicitly to find linguistic ordering for comparison conditions, the BETWEEN condition, the IN condition, ORDER BY, GROUP BY, and COUNT(DISTINCT), those operations may return results that are only approximate for long character values. If you want guarantee that the results of those operations are exact, then migrate your database to use MAX_STRING_SIZE = EXTENDED.

Refer to "Extended Data Types" for more information on the MAX_STRING_SIZE initialization parameter.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for NLSSORT

Examples

This function can be used to specify sorting and comparison operations based on a linguistic sort sequence rather than on the binary value of a string. The following example creates a test table containing two values and shows how the values returned can be ordered by the NLSSORT function:

```
CREATE TABLE test (name VARCHAR2(15));
INSERT INTO test VALUES ('Gaardiner');
INSERT INTO test VALUES ('Gaberd');
INSERT INTO test VALUES ('Gaasten');

SELECT *
  FROM test
  ORDER BY name;
```

```
NAME
---------------
Gaardiner
Gaasten
Gaberd

SELECT *
  FROM test
  ORDER BY NLSSORT(name, 'NLS_SORT = XDanish');

NAME
---------------
Gaberd
Gaardiner
Gaasten
```

The following example shows how to use the `NLSSORT` function in comparison operations:

```
SELECT *
  FROM test
  WHERE name > 'Gaberd'
  ORDER BY name;

no rows selected

SELECT *
  FROM test
  WHERE NLSSORT(name, 'NLS_SORT = XDanish') >
        NLSSORT('Gaberd', 'NLS_SORT = XDanish')
  ORDER BY name;

NAME
---------------
Gaardiner
Gaasten
```

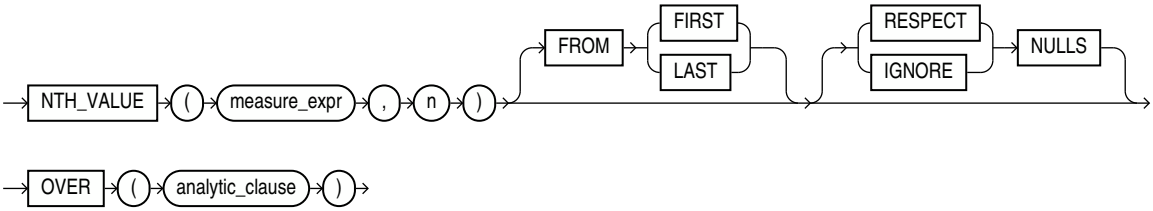
If you frequently use `NLSSORT` in comparison operations with the same linguistic sort sequence, then consider this more efficient alternative: Set the `NLS_COMP` parameter (either for the database or for the current session) to `LINGUISTIC`, and set the `NLS_SORT` parameter for the session to the desired sort sequence. Oracle Database will use that sort sequence by default for all sorting and comparison operations during the current session:

```
ALTER SESSION SET NLS_COMP = 'LINGUISTIC';
ALTER SESSION SET NLS_SORT = 'XDanish';

SELECT *
  FROM test
  WHERE name > 'Gaberd'
  ORDER BY name;

NAME
---------------
Gaardiner
Gaasten
```

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information on sort sequences

NTH_VALUE

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions of the *analytic_clause*

Purpose

`NTH_VALUE` returns the *measure_expr* value of the *n*th row in the window defined by the *analytic_clause*. The returned value has the data type of the *measure_expr*.

- {`RESPECT` | `IGNORE`} `NULLS` determines whether null values of *measure_expr* are included in or eliminated from the calculation. The default is `RESPECT NULLS`.
- *n* determines the *n*th row for which the measure value is to be returned. *n* can be a constant, bind variable, column, or an expression involving them, as long as it resolves to a positive integer. The function returns `NULL` if the data source window has fewer than *n* rows. If *n* is null, then the function returns an error.
- `FROM` {`FIRST` | `LAST`} determines whether the calculation begins at the first or last row of the window. The default is `FROM FIRST`.

If you omit the *windowing_clause* of the *analytic_clause*, it defaults to `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. This default sometimes returns an unexpected value for `NTH_VALUE ... FROM LAST ...` , because the last value in the window is at the bottom of the window, which is not fixed. It keeps changing as the current row changes. For expected results, specify the *windowing_clause* as `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`. Alternatively, you can specify the *windowing_clause* as `RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for more information on the use of this function
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `NTH_VALUE` when it is a character value

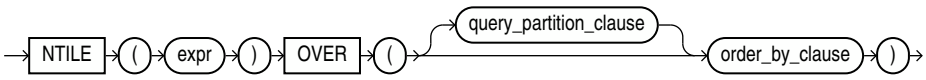
Examples

The following example shows the minimum *amount_sold* value for the second *channel_id* in ascending order for each *prod_id* between 13 and 16:

```
SELECT prod_id, channel_id, MIN(amount_sold),
    NTH_VALUE(MIN(amount_sold), 2) OVER (PARTITION BY prod_id ORDER BY channel_id
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) nv
  FROM sales
  WHERE prod_id BETWEEN 13 and 16
  GROUP BY prod_id, channel_id;

   PROD_ID CHANNEL_ID MIN(AMOUNT_SOLD)         NV
---------- ---------- ---------------- ----------
        13          2           907.34      906.2
        13          3            906.2      906.2
        13          4           842.21      906.2
        14          2          1015.94    1036.72
        14          3          1036.72    1036.72
        14          4           935.79    1036.72
        15          2           871.19     871.19
        15          3           871.19     871.19
        15          4           871.19     871.19
        16          2           266.84     266.84
        16          3           266.84     266.84
        16          4           266.84     266.84
        16          9            11.99     266.84

13 rows selected.
```

NTILE

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *expr*

Purpose

`NTILE` is an analytic function. It divides an ordered data set into a number of buckets indicated by *expr* and assigns the appropriate bucket number to each row. The buckets are numbered 1 through *expr*. The *expr* value must resolve to a positive constant for each partition. Oracle Database expects an integer, and if *expr* is a noninteger constant, then Oracle truncates the value to an integer. The return value is `NUMBER`.

The number of rows in the buckets can differ by at most 1. The remainder values (the remainder of number of rows divided by buckets) are distributed one for each bucket, starting with bucket 1.

If *expr* is greater than the number of rows, then a number of buckets equal to the number of rows will be filled, and the remaining buckets will be empty.

You cannot nest analytic functions by using NTILE or any other analytic function for *expr*. However, you can use other built-in function expressions for *expr*.

> **See Also:**
>
> "About SQL Expressions " for information on valid forms of *expr* and Table 2-9 for more information on implicit conversion

Examples

The following example divides into 4 buckets the values in the salary column of the oe.employees table from Department 100. The salary column has 6 values in this department, so the two extra values (the remainder of 6 / 4) are allocated to buckets 1 and 2, which therefore have one more value than buckets 3 or 4.

```
SELECT last_name, salary, NTILE(4) OVER (ORDER BY salary DESC) AS quartile
  FROM employees
  WHERE department_id = 100
  ORDER BY last_name, salary, quartile;

LAST_NAME                 SALARY   QUARTILE
------------------------- ---------- ----------
Chen                        8200          2
Faviet                      9000          1
Greenberg                  12008          1
Popp                        6900          4
Sciarra                     7700          3
Urman                       7800          2
```

NULLIF

Syntax

Purpose

NULLIF compares *expr1* and *expr2*. If they are equal, then the function returns null. If they are not equal, then the function returns *expr1*. You cannot specify the literal NULL for *expr1*.

If both arguments are numeric data types, then Oracle Database determines the argument with the higher numeric precedence, implicitly converts the other argument to that data type, and returns that data type. If the arguments are not numeric, then they must be of the same data type, or Oracle returns an error.

The NULLIF function is logically equivalent to the following CASE expression:

```
CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END
```

> **See Also:**
>
> - "CASE Expressions "
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `NULLIF` uses to compare characters from *`expr1`* with characters from *`expr2`*, and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

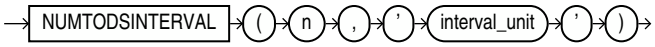
Examples

The following example selects those employees from the sample schema `hr` who have changed jobs since they were hired, as indicated by a `job_id` in the `job_history` table different from the current `job_id` in the `employees` table:

```
SELECT e.last_name, NULLIF(j.job_id, e.job_id) "Old Job ID"
  FROM employees e, job_history j
  WHERE e.employee_id = j.employee_id
  ORDER BY last_name, "Old Job ID";

LAST_NAME                 Old Job ID
------------------------- ----------
De Haan                   IT_PROG
Hartstein                 MK_REP
Kaufling                  ST_CLERK
Kochhar                   AC_ACCOUNT
Kochhar                   AC_MGR
Raphaely                  ST_CLERK
Taylor                    SA_MAN
Taylor
Whalen                    AC_ACCOUNT
Whalen
```

NUMTODSINTERVAL

Syntax



Purpose

`NUMTODSINTERVAL` converts *`n`* to an `INTERVAL DAY TO SECOND` literal. The argument *`n`* can be any `NUMBER` value or an expression that can be implicitly converted to a `NUMBER` value. The argument *`interval_unit`* can be of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type. The value for *`interval_unit`* specifies the unit of *`n`* and must resolve to one of the following string values:

- `'DAY'`
- `'HOUR'`
- `'MINUTE'`
- `'SECOND'`

interval_unit is case insensitive. Leading and trailing values within the parentheses are ignored. By default, the precision of the return is 9.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

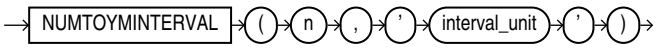
Examples

The following example uses NUMTODSINTERVAL in a COUNT analytic function to calculate, for each employee, the number of employees hired by the same manager within the past 100 days from his or her hire date. Refer to "Analytic Functions " for more information on the syntax of the analytic functions.

```
SELECT manager_id, last_name, hire_date,
       COUNT(*) OVER (PARTITION BY manager_id ORDER BY hire_date
       RANGE NUMTODSINTERVAL(100, 'day') PRECEDING) AS t_count
  FROM employees
  ORDER BY last_name, hire_date;

MANAGER_ID LAST_NAME                 HIRE_DATE    T_COUNT
---------- ------------------------- --------- ----------
       149 Abel                      11-MAY-04          1
       147 Ande                      24-MAR-08          3
       121 Atkinson                  30-OCT-05          2
       103 Austin                    25-JUN-05          1
. . .
       124 Walsh                     24-APR-06          2
       100 Weiss                     18-JUL-04          1
       101 Whalen                    17-SEP-03          1
       100 Zlotkey                   29-JAN-08          2
```

NUMTOYMINTERVAL

Syntax



Purpose

NUMTOYMINTERVAL converts number *n* to an INTERVAL YEAR TO MONTH literal. The argument *n* can be any NUMBER value or an expression that can be implicitly converted to a NUMBER value. The argument *interval_unit* can be of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The value for *interval_unit* specifies the unit of *n* and must resolve to one of the following string values:

- 'YEAR'
- 'MONTH'

interval_unit is case insensitive. Leading and trailing values within the parentheses are ignored. By default, the precision of the return is 9.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

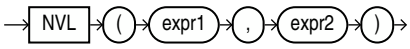
Examples

The following example uses `NUMTOYMINTERVAL` in a `SUM` analytic function to calculate, for each employee, the total salary of employees hired in the past one year from his or her hire date. Refer to "Analytic Functions " for more information on the syntax of the analytic functions.

```
SELECT last_name, hire_date, salary,
       SUM(salary) OVER (ORDER BY hire_date
       RANGE NUMTOYMINTERVAL(1,'year') PRECEDING) AS t_sal
  FROM employees
  ORDER BY last_name, hire_date;

LAST_NAME                 HIRE_DATE     SALARY      T_SAL
------------------------- --------- ---------- ----------
Abel                      11-MAY-04      11000      90300
Ande                      24-MAR-08       6400     112500
Atkinson                  30-OCT-05       2800     177000
Austin                    25-JUN-05       4800     134700
. . .
Walsh                     24-APR-06       3100     186200
Weiss                     18-JUL-04       8000      70900
Whalen                    17-SEP-03       4400      54000
Zlotkey                   29-JAN-08      10500     119000
```

NVL

Syntax



Purpose

`NVL` lets you replace null (returned as a blank) with a string in the results of a query. If *expr1* is null, then `NVL` returns *expr2*. If *expr1* is not null, then `NVL` returns *expr1*.

The arguments *expr1* and *expr2* can have any data type. If their data types are different, then Oracle Database implicitly converts one to the other. If they cannot be converted implicitly, then the database returns an error. The implicit conversion is implemented as follows:

- If *expr1* is character data, then Oracle Database converts *expr2* to the data type of *expr1* before comparing them and returns `VARCHAR2` in the character set of *expr1*.
- If *expr1* is numeric, then Oracle Database determines which argument has the highest numeric precedence, implicitly converts the other argument to that data type, and returns that data type.

> **See Also:**
>
> - Table 2-9 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence
> - "COALESCE " and "CASE Expressions ", which provide functionality similar to that of `NVL`
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `NVL` when it is a character value

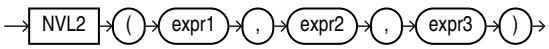
Examples

The following example returns a list of employee names and commissions, substituting "Not Applicable" if the employee receives no commission:

```
SELECT last_name, NVL(TO_CHAR(commission_pct), 'Not Applicable') commission
  FROM employees
  WHERE last_name LIKE 'B%'
  ORDER BY last_name;

LAST_NAME                 COMMISSION
------------------------- ----------------------------------------
Baer                      Not Applicable
Baida                     Not Applicable
Banda                     .1
Bates                     .15
Bell                      Not Applicable
Bernstein                 .25
Bissot                    Not Applicable
Bloom                     .2
Bull                      Not Applicable
```

NVL2

Syntax



Purpose

`NVL2` lets you determine the value returned by a query based on whether a specified expression is null or not null. If *expr1* is not null, then `NVL2` returns *expr2*. If *expr1* is null, then `NVL2` returns *expr3*.

The argument *expr1* can have any data type. The arguments *expr2* and *expr3* can have any data types except `LONG`.

If the data types of *expr2* and *expr3* are different, then Oracle Database implicitly converts one to the other. If they cannot be converted implicitly, then the database returns an error. If *expr2* is character or numeric data, then the implicit conversion is implemented as follows:

- If *expr2* is character data, then Oracle Database converts *expr3* to the data type of *expr2* before returning a value unless *expr3* is a null constant. In that case, a data type

conversion is not necessary, and the database returns `VARCHAR2` in the character set of *expr2*.

- If *expr2* is numeric data, then Oracle Database determines which argument has the highest numeric precedence, implicitly converts the other argument to that data type, and returns that data type.

> **See Also:**
>
> - Table 2-9 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `NVL2` when it is a character value

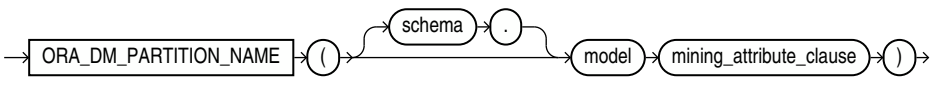
Examples

The following example shows whether the income of some employees is made up of salary plus commission, or just salary, depending on whether the `commission_pct` column of `employees` is null or not.

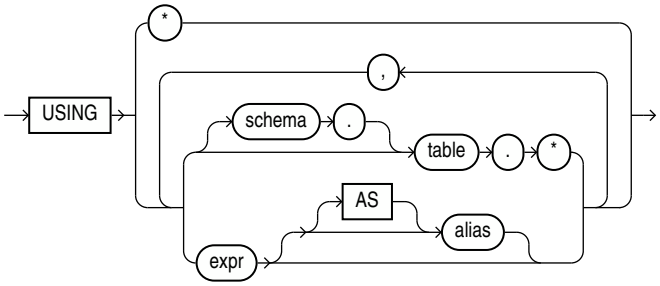
```
SELECT last_name, salary,
       NVL2(commission_pct, salary + (salary * commission_pct), salary) income
  FROM employees
  WHERE last_name like 'B%'
  ORDER BY last_name;

LAST_NAME                 SALARY     INCOME
------------------------- ---------- ----------
Baer                       10000      10000
Baida                       2900       2900
Banda                       6200       6820
Bates                       7300       8395
Bell                        4000       4000
Bernstein                   9500      11875
Bissot                      3300       3300
Bloom                      10000      12000
Bull                        4100       4100
```

ORA_DM_PARTITION_NAME

Syntax



***mining_attribute_clause*::=**



Purpose

`ORA_DM_PARTITION_NAME` is a single row function that works along with other existing functions. This function returns the name of the partition associated with the input row. When `ORA_DM_PARTITION_NAME` is used on a non-partitioned model, the result is `NULL`.

The syntax of the `ORA_DM_PARTITION_NAME` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

The *`mining_attribute_clause`* identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *`mining_attribute_clause`* behaves as described for the `PREDICTION` function. See mining_attribute_clause.

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring
> - *Oracle Machine Learning for SQL Concepts* for information about clustering

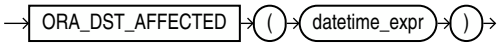
> **Note:**
>
> The following examples are excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

```
SELECT prediction(mymodel using *) pred, ora_dm_partition_name(mymodel USING
*) pname FROM customers;
```

ORA_DST_AFFECTED

Syntax



Purpose

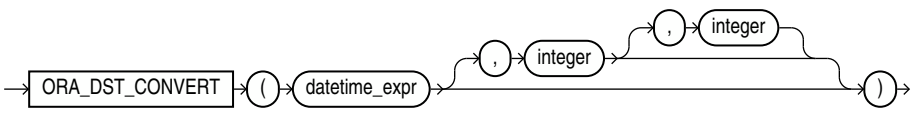
`ORA_DST_AFFECTED` is useful when you are changing the time zone data file for your database. The function takes as an argument a datetime expression that resolves to a `TIMESTAMP WITH TIME ZONE` value or a `VARRAY` object that contains `TIMESTAMP WITH TIME ZONE` values. The function returns `1` if the datetime value is affected by or will result in a "nonexisting time" or "duplicate time" error with the new time zone data. Otherwise, it returns 0.

This function can be issued only when changing the time zone data file of the database and upgrading the timestamp with the time zone data, and only between the execution of the `DBMS_DST.BEGIN_PREPARE` and the `DBMS_DST.END_PREPARE` procedures or between the execution of the `DBMS_DST.BEGIN_UPGRADE` and the `DBMS_DST.END_UPGRADE` procedures.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on time zone data files and on how Oracle Database handles daylight saving time, and *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_DST` package

ORA_DST_CONVERT

Syntax



Purpose

`ORA_DST_CONVERT` is useful when you are changing the time zone data file for your database. The function lets you specify error handling for a specified datetime expression.

- For *`datetime_expr`*, specify a datetime expression that resolves to a `TIMESTAMP WITH TIME ZONE` value or a `VARRAY` object that contains `TIMESTAMP WITH TIME ZONE` values.
- The optional second argument specifies handling of "duplicate time" errors. Specify `0` (false) to suppress the error by returning the source datetime value. This is the default. Specify `1` (true) to allow the database to return the duplicate time error.

- The optional third argument specifies handling of "nonexisting time" errors. Specify `0` (false) to suppress the error by returning the source datetime value. This is the default. Specify `1` (true) to allow the database to return the nonexisting time error.

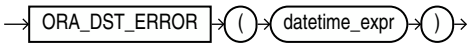
If no error occurs, this function returns a value of the same data type as *datetime_expr* (a `TIMESTAMP WITH TIME ZONE` value or a `VARRAY` object that contains `TIMESTAMP WITH TIME ZONE` values). The returned datetime value when interpreted with the new time zone file corresponds to *datetime_expr* interpreted with the old time zone file.

This function can be issued only when changing the time zone data file of the database and upgrading the timestamp with the time zone data, and only between the execution of the `DBMS_DST.BEGIN_UPGRADE` and the `DBMS_DST.END_UPGRADE` procedures.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on time zone data files and on how Oracle Database handles daylight saving time, and *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_DST` package

ORA_DST_ERROR

Syntax



Purpose

`ORA_DST_ERROR` is useful when you are changing the time zone data file for your database. The function takes as an argument a datetime expression that resolves to a `TIMESTAMP WITH TIME ZONE` value or a `VARRAY` object that contains `TIMESTAMP WITH TIME ZONE` values, and indicates whether the datetime value will result in an error with the new time zone data. The return values are:

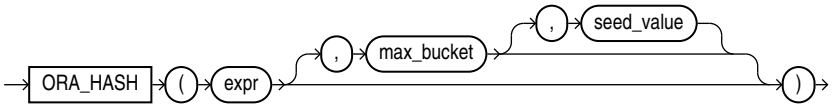
- `0`: the datetime value does not result in an error with the new time zone data.
- `1878`: the datetime value results in a "nonexisting time" error.
- `1883`: the datetime value results in a "duplicate time" error.

This function can be issued only when changing the time zone data file of the database and upgrading the timestamp with the time zone data, and only between the execution of the `DBMS_DST.BEGIN_PREPARE` and the `DBMS_DST.END_PREPARE` procedures or between the execution of the `DBMS_DST.BEGIN_UPGRADE` and the `DBMS_DST.END_UPGRADE` procedures.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on time zone data files and on how Oracle Database handles daylight saving time, and *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_DST` package

ORA_HASH

Syntax



Purpose

`ORA_HASH` is a function that computes a hash value for a given expression. This function is useful for operations such as analyzing a subset of data and generating a random sample.

- The *expr* argument determines the data for which you want Oracle Database to compute a hash value. There are no restrictions on the length of data represented by *expr*, which commonly resolves to a column name. The *expr* cannot be a `LONG` or LOB type. It cannot be a user-defined object type unless it is a nested table type. The hash value for nested table types does not depend on the order of elements in the collection. All other data types are supported for *expr*.
- The optional *max_bucket* argument determines the maximum bucket value returned by the hash function. You can specify any value between 0 and 4294967295. The default is 4294967295.
- The optional *seed_value* argument enables Oracle to produce many different results for the same set of data. Oracle applies the hash function to the combination of *expr* and *seed_value*. You can specify any value between 0 and 4294967295. The default is 0.

The function returns a `NUMBER` value.

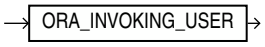
Examples

The following example creates a hash value for each combination of customer ID and product ID in the `sh.sales` table, divides the hash values into a maximum of 100 buckets, and returns the sum of the `amount_sold` values in the first bucket (bucket 0). The third argument (5) provides a seed value for the hash function. You can obtain different hash results for the same query by changing the seed value.

```
SELECT SUM(amount_sold)
  FROM sales
  WHERE ORA_HASH(CONCAT(cust_id, prod_id), 99, 5) = 0;

SUM(AMOUNT_SOLD)
----------------
       989431.14
```

ORA_INVOKING_USER

Syntax



Purpose

`ORA_INVOKING_USER` returns the name of the database user who invoked the current statement or view. This function takes into account the `BEQUEATH` property of intervening views referenced in the statement. If this function is invoked from within a definer's rights context, then it returns the name of the owner of the definer's rights object. If the invoking user is a Real Application Security user, then it returns user `XS$NULL`.

This function returns a `VARCHAR2` value.

> **See Also:**
>
> - BEQUEATH clause of the `CREATE VIEW` statement
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `ORA_INVOKING_USER`

Examples

The following example returns the name of the database user who invoked the statement:

```
SELECT ORA_INVOKING_USER FROM DUAL;
```

ORA_INVOKING_USERID

Syntax

Purpose

`ORA_INVOKING_USERID` returns the identifier of the database user who invoked the current statement or view. This function takes into account the `BEQUEATH` property of intervening views referenced in the statement.

This function returns a `NUMBER` value.

> **See Also:**
>
> - ORA_INVOKING_USER to learn how Oracle Database determines the database user who invoked the current statement or view
> - BEQUEATH clause of the `CREATE VIEW` statement

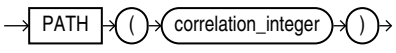
Examples

The following example returns the identifier of the database user who invoked the statement:

```
SELECT ORA_INVOKING_USERID FROM DUAL;
```

PATH

Syntax



Purpose

`PATH` is an ancillary function used only with the `UNDER_PATH` and `EQUALS_PATH` conditions. It returns the relative path that leads to the resource specified in the parent condition.

The *`correlation_integer`* can be any `NUMBER` integer and is used to correlate this ancillary function with its primary condition. Values less than 1 are treated as 1.

> **See Also:**
>
> - EQUALS_PATH Condition and UNDER_PATH Condition
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `PATH`

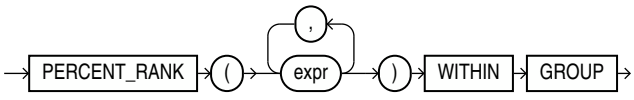
Examples

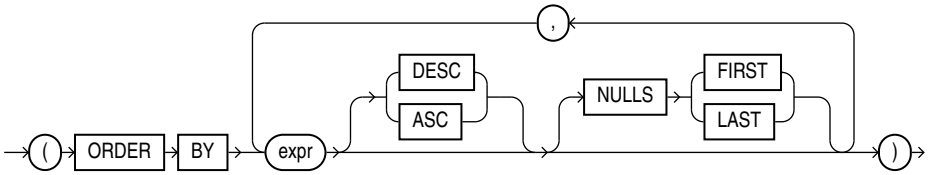
Refer to the related function DEPTH for an example using both of these ancillary functions of the `EQUALS_PATH` and `UNDER_PATH` conditions.

PERCENT_RANK

Aggregate Syntax

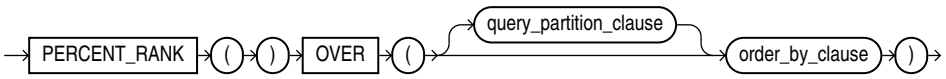
***percent_rank_aggregate*::=**





Analytic Syntax

***percent_rank_analytic*::=**



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`PERCENT_RANK` is similar to the `CUME_DIST` (cumulative distribution) function. The range of values returned by `PERCENT_RANK` is 0 to 1, inclusive. The first row in any set has a `PERCENT_RANK` of 0. The return value is `NUMBER`.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

- As an aggregate function, `PERCENT_RANK` calculates, for a hypothetical row *`r`* identified by the arguments of the function and a corresponding sort specification, the rank of row *`r`* minus 1 divided by the number of rows in the aggregate group. This calculation is made as if the hypothetical row *`r`* were inserted into the group of rows over which Oracle Database is to aggregate.

  The arguments of the function identify a single hypothetical row within each aggregate group. Therefore, they must all evaluate to constant expressions within each aggregate

group. The constant argument expressions and the expressions in the `ORDER BY` clause of the aggregate match by position. Therefore the number of arguments must be the same and their types must be compatible.

- As an analytic function, for a row *r*, `PERCENT_RANK` calculates the rank of *r* minus 1, divided by 1 less than the number of rows being evaluated (the entire query result set or a partition).

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `PERCENT_RANK` uses to compare character values for the `ORDER BY` clause

Aggregate Example

The following example calculates the percent rank of a hypothetical employee in the sample table `hr.employees` with a salary of $15,500 and a commission of 5%:

```
SELECT PERCENT_RANK(15000, .05) WITHIN GROUP
       (ORDER BY salary, commission_pct) "Percent-Rank"
  FROM employees;

Percent-Rank
------------
  .971962617
```

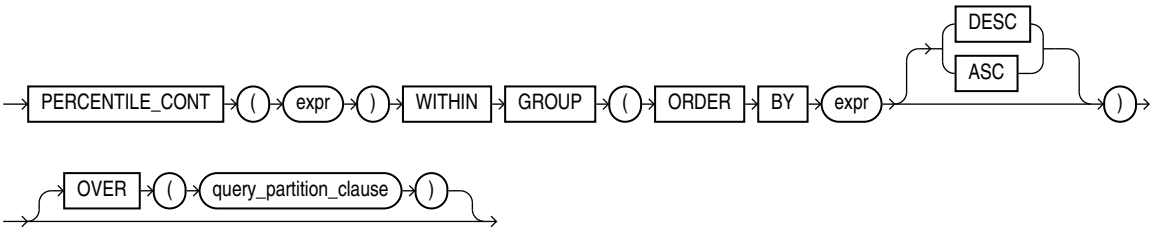
Analytic Example

The following example calculates, for each employee, the percent rank of the employee's salary within the department:

```
SELECT department_id, last_name, salary, PERCENT_RANK()
       OVER (PARTITION BY department_id ORDER BY salary DESC) AS pr
  FROM employees
  ORDER BY pr, salary, last_name;

DEPARTMENT_ID LAST_NAME                    SALARY         PR
------------- ------------------------- ---------- ----------
           10 Whalen                          4400          0
           40 Mavris                          6500          0
              Grant                           7000          0
. . .
           80 Vishney                        10500 .181818182
           80 Zlotkey                        10500 .181818182
           30 Khoo                            3100         .2
. . .
           50 Markle                          2200 .954545455
           50 Philtanker                      2200 .954545455
           50 Olson                           2100          1
. . .
```

PERCENTILE_CONT

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions of the `OVER` clause

Purpose

`PERCENTILE_CONT` is an inverse distribution function that assumes a continuous distribution model. It takes a percentile value and a sort specification, and returns an interpolated value that would fall into that percentile value with respect to the sort specification. Nulls are ignored in the calculation.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

The first *expr* must evaluate to a numeric value between 0 and 1, because it is a percentile value. This *expr* must be constant within each aggregation group. The `ORDER BY` clause takes a single expression that must be a numeric or datetime value, as these are the types over which Oracle can perform interpolation.

The result of `PERCENTILE_CONT` is computed by linear interpolation between values after ordering them. Using the percentile value (P) and the number of rows (N) in the aggregation group, you can compute the row number you are interested in after ordering the rows with respect to the sort specification. This row number (RN) is computed according to the formula `RN = (1+(P*(N-1))`. The final result of the aggregate function is computed by linear interpolation between the values from rows at row numbers `CRN = CEILING(RN)` and `FRN = FLOOR(RN)`.

The final result will be:

```
If (CRN = FRN = RN) then the result is
  (value of expression from row at RN)
```

```
  Otherwise the result is
    (CRN - RN) * (value of expression for row at FRN) +
    (RN - FRN) * (value of expression for row at CRN)
```

You can use the PERCENTILE_CONT function as an analytic function. You can specify only the *query_partitioning_clause* in its OVER clause. It returns, for each row, the value that would fall into the specified percentile among a set of values within each partition.

The MEDIAN function is a specific case of PERCENTILE_CONT where the percentile value defaults to 0.5. For more information, refer to MEDIAN .

> **Note:**
>
> Before processing a large amount of data with the PERCENTILE_CONT function, consider using one of the following methods to obtain approximate results more quickly than exact results:
>
> - Set the APPROX_FOR_PERCENTILE initialization parameter to PERCENTILE_CONT or ALL before using the PERCENTILE_CONT function. Refer to *Oracle Database Reference* for more information on this parameter.
> - Use the APPROX_PERCENTILE function instead of the PERCENTILE_CONT function. Refer to APPROX_PERCENTILE.

Aggregate Example

The following example computes the median salary in each department:

```
SELECT department_id,
       PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary DESC) "Median cont",
       PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY salary DESC) "Median disc"
  FROM employees
  GROUP BY department_id
  ORDER BY department_id;

DEPARTMENT_ID Median cont Median disc
------------- ----------- -----------
           10        4400        4400
           20        9500       13000
           30        2850        2900
           40        6500        6500
           50        3100        3100
           60        4800        4800
           70       10000       10000
           80        8900        9000
           90       17000       17000
          100        8000        8200
          110       10154       12008
                     7000        7000
```

PERCENTILE_CONT and PERCENTILE_DISC may return different results. PERCENTILE_CONT returns a computed result after doing linear interpolation. PERCENTILE_DISC simply returns a value from the set of values that are aggregated over. When the percentile value is 0.5, as in this example, PERCENTILE_CONT returns the average of the two middle values for groups with even number of elements, whereas PERCENTILE_DISC returns the value of the first one among the two middle values. For aggregate groups with an odd number of elements, both functions return the value of the middle element.

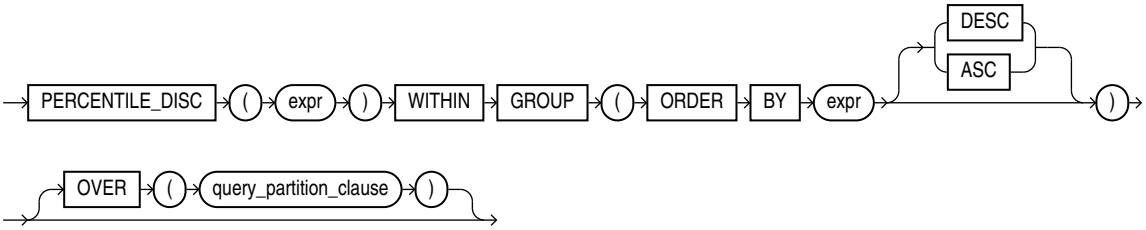
Analytic Example

In the following example, the median for Department 60 is 4800, which has a corresponding percentile (`Percent_Rank`) of 0.5. None of the salaries in Department 30 have a percentile of 0.5, so the median value must be interpolated between 2900 (percentile 0.4) and 2800 (percentile 0.6), which evaluates to 2850.

```
SELECT last_name, salary, department_id,
       PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary DESC)
         OVER (PARTITION BY department_id) "Percentile_Cont",
       PERCENT_RANK()
        OVER (PARTITION BY department_id ORDER BY salary DESC) "Percent_Rank"
  FROM employees
  WHERE department_id IN (30, 60)
  ORDER BY last_name, salary, department_id;

LAST_NAME                     SALARY DEPARTMENT_ID Percentile_Cont Percent_Rank
------------------------- ---------- ------------- --------------- ------------
Austin                          4800            60            4800           .5
Baida                           2900            30            2850           .4
Colmenares                      2500            30            2850            1
Ernst                           6000            60            4800          .25
Himuro                          2600            30            2850           .8
Hunold                          9000            60            4800            0
Khoo                            3100            30            2850           .2
Lorentz                         4200            60            4800            1
Pataballa                       4800            60            4800           .5
Raphaely                       11000            30            2850            0
Tobias                          2800            30            2850           .6
```

PERCENTILE_DISC

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions of the `OVER` clause

Purpose

`PERCENTILE_DISC` is an inverse distribution function that assumes a discrete distribution model. It takes a percentile value and a sort specification and returns an element from the set. Nulls are ignored in the calculation.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

The first *expr* must evaluate to a numeric value between 0 and 1, because it is a percentile value. This expression must be constant within each aggregate group. The `ORDER BY` clause takes a single expression that can be of any type that can be sorted.

For a given percentile value `P`, `PERCENTILE_DISC` sorts the values of the expression in the `ORDER BY` clause and returns the value with the smallest `CUME_DIST` value (with respect to the same sort specification) that is greater than or equal to `P`.

> **Note:**
>
> Before processing a large amount of data with the `PERCENTILE_DISC` function, consider using one of the following methods to obtain approximate results more quickly than exact results:
>
> - Set the `APPROX_FOR_PERCENTILE` initialization parameter to `PERCENTILE_DISC` or `ALL` before using the `PERCENTILE_DISC` function. Refer to *Oracle Database Reference* for more information on this parameter.
> - Use the `APPROX_PERCENTILE` function instead of the `PERCENTILE_DISC` function. Refer to APPROX_PERCENTILE.

Aggregate Example

See aggregate example for PERCENTILE_CONT .

Analytic Example

The following example calculates the median discrete percentile of the salary of each employee in the sample table `hr.employees`:

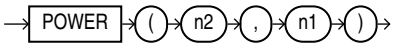
```
SELECT last_name, salary, department_id,
       PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY salary DESC)
         OVER (PARTITION BY department_id) "Percentile_Disc",
       CUME_DIST() OVER (PARTITION BY department_id
         ORDER BY salary DESC) "Cume_Dist"
  FROM employees
  WHERE department_id in (30, 60)
  ORDER BY last_name, salary, department_id;

LAST_NAME                 SALARY DEPARTMENT_ID Percentile_Disc  Cume_Dist
------------------------- ---------- ------------- --------------- ----------
Austin                      4800            60            4800         .8
Baida                       2900            30            2900         .5
Colmenares                  2500            30            2900          1
Ernst                       6000            60            4800         .4
Himuro                      2600            30            2900 .833333333
```

```
Hunold                    9000         60         4800         .2
Khoo                      3100         30         2900 .333333333
Lorentz                   4200         60         4800          1
Pataballa                 4800         60         4800         .8
Raphaely                 11000         30         2900 .166666667
Tobias                    2800         30         2900 .666666667
```

The median value for Department 30 is 2900, which is the value whose corresponding percentile (`Cume_Dist`) is the smallest value greater than or equal to 0.5. The median value for Department 60 is 4800, which is the value whose corresponding percentile is the smallest value greater than or equal to 0.5.

POWER

Syntax



Purpose

`POWER` returns *n2* raised to the *n1* power. The base *n2* and the exponent *n1* can be any numbers, but if *n2* is negative, then *n1* must be an integer.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If any argument is `BINARY_FLOAT` or `BINARY_DOUBLE`, then the function returns `BINARY_DOUBLE`. Otherwise, the function returns `NUMBER`.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

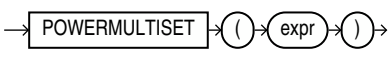
Examples

The following example returns 3 squared:

```
SELECT POWER(3,2) "Raised"
  FROM DUAL;

    Raised
----------
         9
```

POWERMULTISET

Syntax



Purpose

`POWERMULTISET` takes as input a nested table and returns a nested table of nested tables containing all nonempty subsets (called submultisets) of the input nested table.

- *expr* can be any expression that evaluates to a nested table.
- If *expr* resolves to null, then Oracle Database returns `NULL`.
- If *expr* resolves to a nested table that is empty, then Oracle returns an error.
- The element types of the nested table must be comparable. Refer to "Comparison Conditions " for information on the comparability of nonscalar types.

> **Note:**
>
> This function is not supported in PL/SQL.

Examples

First, create a data type that is a nested table of the `cust_address_tab_type` data type:

```
CREATE TYPE cust_address_tab_tab_typ
  AS TABLE OF cust_address_tab_typ;
/
```

Now, select the nested table column `cust_address_ntab` from the `customers_demo` table using the `POWERMULTISET` function:

```
SELECT CAST(POWERMULTISET(cust_address_ntab) AS cust_address_tab_tab_typ)
  FROM customers_demo;

CAST(POWERMULTISET(CUST_ADDRESS_NTAB) AS CUST_ADDRESS_TAB_TAB_TYP)
  (STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
--------------------------------------------------------------------
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
  ('514 W Superior St', '46901', 'Kokomo', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
  ('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
  ('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP
  ('6445 Bay Harbor Ln', '46254', 'Indianapolis', 'IN', 'US')))
. . .
```

The preceding example requires the `customers_demo` table and a nested table column containing data. Refer to "Multiset Operators " to create this table and nested table columns.

POWERMULTISET_BY_CARDINALITY

Syntax

Purpose

`POWERMULTISET_BY_CARDINALITY` takes as input a nested table and a cardinality and returns a nested table of nested tables containing all nonempty subsets (called submultisets) of the nested table of the specified cardinality.

- *`expr`* can be any expression that evaluates to a nested table.
- *`cardinality`* can be any positive integer.
- If *`expr`* resolves to null, then Oracle Database returns `NULL`.
- If *`expr`* resolves to a nested table that is empty, then Oracle returns an error.
- The element types of the nested table must be comparable. Refer to "Comparison Conditions " for information on the comparability of nonscalar types.

> **Note:**
>
> This function is not supported in PL/SQL.

Examples

First, create a data type that is a nested table of the `cust_address_tab_type` data type:

```
CREATE TYPE cust_address_tab_tab_typ
  AS TABLE OF cust_address_tab_typ;
/
```

Next, duplicate the elements in all the nested table rows to increase the cardinality of the nested table rows to 2:

```
UPDATE customers_demo
  SET cust_address_ntab = cust_address_ntab MULTISET UNION cust_address_ntab;
```

Now, select the nested table column `cust_address_ntab` from the `customers_demo` table using the `POWERMULTISET_BY_CARDINALITY` function:

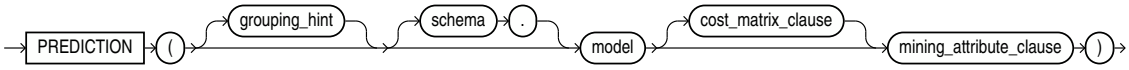
```
SELECT CAST(POWERMULTISET_BY_CARDINALITY(cust_address_ntab, 2)
         AS cust_address_tab_tab_typ)
  FROM customers_demo;

CAST(POWERMULTISET_BY_CARDINALITY(CUST_ADDRESS_NTAB,2) AS CUST_ADDRESS_TAB_TAB_TYP)
  (STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
--------------------------------------------------------------------------------
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP
  (CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US'),
   CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP
  (CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US'),
   CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US')))
CUST_ADDRESS_TAB_TAB_TYP(CUST_ADDRESS_TAB_TYP
  (CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'),
   CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US')))
. . .
```

The preceding example requires the `customers_demo` table and a nested table column containing data. Refer to "Multiset Operators " to create this table and nested table columns.

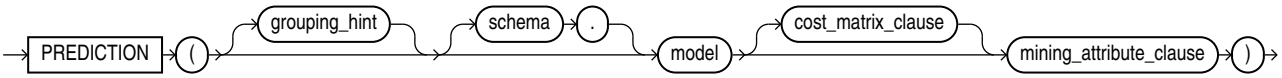
PREDICTION

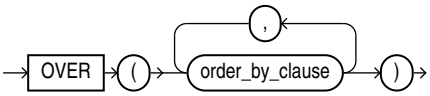
Syntax

***prediction*::=**



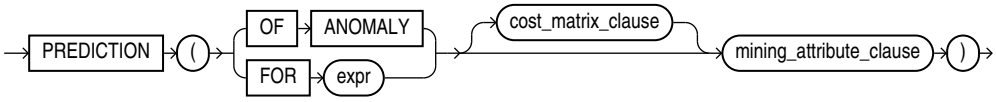
***prediction_ordered*::=**

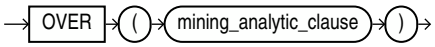




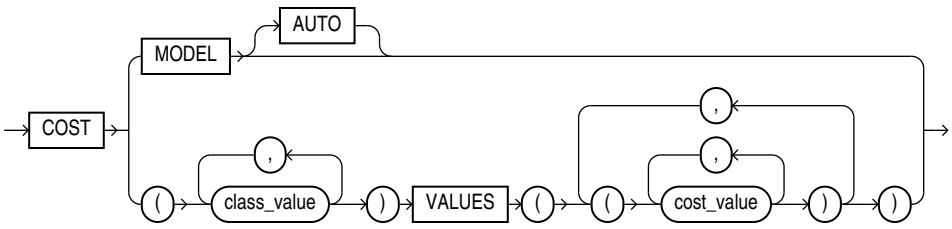
Analytic Syntax

***prediction_analytic*::=**

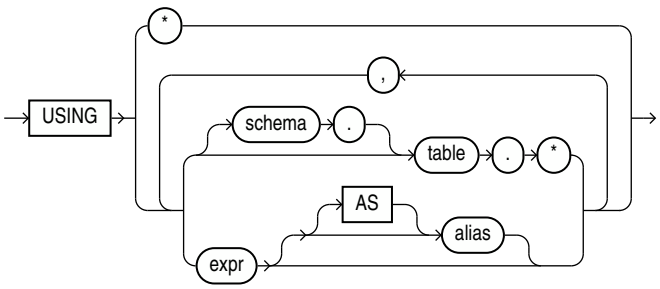




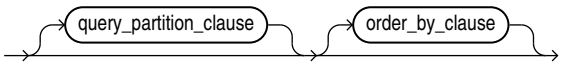
***cost_matrix_clause*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`PREDICTION` returns a prediction for each row in the selection. The data type of the returned prediction depends on whether the function performs Regression, Classification, or Anomaly Detection.

- **Regression**: Returns the expected target value for each row. The data type of the return value is the data type of the target.
- **Classification**: Returns the most probable target class (or lowest cost target class, if costs are specified) for each row. The data type of the return value is the data type of the target.
- **Anomaly Detection**: Returns `1` or `0` for each row. Typical rows are classified as 1. Rows that differ significantly from the rest of the data are classified as 0.

cost_matrix_clause

Costs are a biasing factor for minimizing the most harmful kinds of misclassifications. You can specify *cost_matrix_clause* for Classification or Anomaly Detection. Costs are not relevant for Regression. The *cost_matrix_clause* behaves as described for "PREDICTION_COST ".

Syntax Choice

`PREDICTION` can score data by applying a mining model object to the data, or it can dynamically score the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax**: Use the *prediction* syntax to score the data with a pre-defined model. Supply the name of a model that performs classification, regression, or anomaly detection.

  Use the *prediction_ordered* syntax for a model that requires ordered data, such as an MSET-SPRT model. The *prediction_ordered* syntax requires an *order_by_clause* clause.

  Restrictions on the *prediction_ordered* syntax are that you cannot use it in the `WHERE` clause of a query. Also, you cannot use a *query_partition_clause* or a *windowing_clause* with the *prediction_ordered* syntax.

  For details about the *order_by_clause*, see "Analytic Functions ".
- **Analytic Syntax**: Use the analytic syntax to score the data without a pre-defined model. The analytic syntax uses *mining_analytic_clause* , which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)

  - For Regression, specify `FOR expr`, where *`expr`* is an expression that identifies a target column that has a numeric data type.
  - For Classification, specify `FOR expr`, where *`expr`* is an expression that identifies a target column that has a character data type.
  - For Anomaly Detection, specify the keywords `OF ANOMALY`.

The syntax of the `PREDICTION` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

`mining_attribute_clause` identifies the column attributes to use as predictors for scoring.

- If you specify `USING *`, all the relevant attributes present in the input row are used.
- If you invoke the function with the analytic syntax, the *`mining_attribute_clause`* is used both for building the transient models and for scoring.
- It you invoke the function with a pre-defined model, the *`mining_attribute_clause`* should include all or some of the attributes that were used to create the model. The following conditions apply:
  - If *`mining_attribute_clause`* includes an attribute with the same name but a different data type from the one that was used to create the model, then the data type is converted to the type expected by the model.
  - If you specify more attributes for scoring than were used to create the model, then the extra attributes are silently ignored.
  - If you specify fewer attributes for scoring than were used to create the model, then scoring is performed on a best-effort basis.

See Also:

- *Oracle Machine Learning for SQL User's Guide* for information about scoring.
- *Oracle Machine Learning for SQL Concepts* for information about predictive Oracle Machine Learning for SQL.
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `PREDICTION` when it is a character value

Note:

The following examples are excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

In this example, the model `dt_sh_clas_sample` predicts the gender and age of customers who are most likely to use an affinity card (target = 1). The `PREDICTION` function takes into account

the cost matrix associated with the model and uses marital status, education, and household size as predictors.

```
SELECT cust_gender, COUNT(*) AS cnt, ROUND(AVG(age)) AS avg_age
   FROM mining_data_apply_v
   WHERE PREDICTION(dt_sh_clas_sample COST MODEL
      USING cust_marital_status, education, household_size) = 1
   GROUP BY cust_gender
   ORDER BY cust_gender;

CUST_GENDER         CNT    AVG_AGE
------------ ---------- ----------
F                   170         38
M                   685         42
```

The cost matrix associated with the model `dt_sh_clas_sample` is stored in the table `dt_sh_sample_costs`. The cost matrix specifies that the misclassification of 1 is 8 times more costly than the misclassification of 0.

```
SQL> select * from dt_sh_sample_cost;

ACTUAL_TARGET_VALUE PREDICTED_TARGET_VALUE         COST
------------------- ---------------------- ------------
                  0                      0   .000000000
                  0                      1  1.000000000
                  1                      0  8.000000000
                  1                      1   .000000000
```

Analytic Example

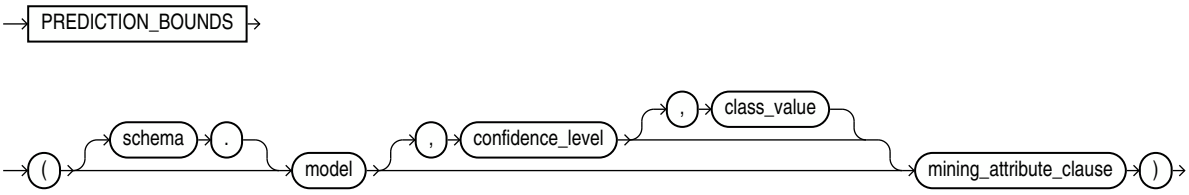
In this example, dynamic regression is used to predict the age of customers who are likely to use an affinity card. The query returns the 3 customers whose predicted age is most different from the actual. The query includes information about the predictors that have the greatest influence on the prediction.

```
SELECT cust_id, age, pred_age, age-pred_age age_diff, pred_det FROM
   (SELECT cust_id, age, pred_age, pred_det,
          RANK() OVER (ORDER BY ABS(age-pred_age) desc) rnk FROM
   (SELECT cust_id, age,
           PREDICTION(FOR age USING *) OVER () pred_age,
           PREDICTION_DETAILS(FOR age ABS USING *) OVER () pred_det
    FROM mining_data_apply_v))
  WHERE rnk <= 3;

CUST_ID  AGE PRED_AGE AGE_DIFF PRED_DET
------- ---- -------- -------- --------------------------------------------------------------
 100910   80    40.67    39.33 <Details algorithm="Support Vector Machines">
                               <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".059"
                                rank="1"/>
                               <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                                rank="2"/>
                               <Attribute name="AFFINITY_CARD" actualValue="0" weight=".059"
                                rank="3"/>
                               <Attribute name="FLAT_PANEL_MONITOR" actualValue="1" weight=".059"
                                rank="4"/>
                               <Attribute name="YRS_RESIDENCE" actualValue="4" weight=".059"
                                rank="5"/>
                               </Details>

 101285    79   42.18    36.82 <Details algorithm="Support Vector Machines">
                               <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".059"
                                rank="1"/>
```

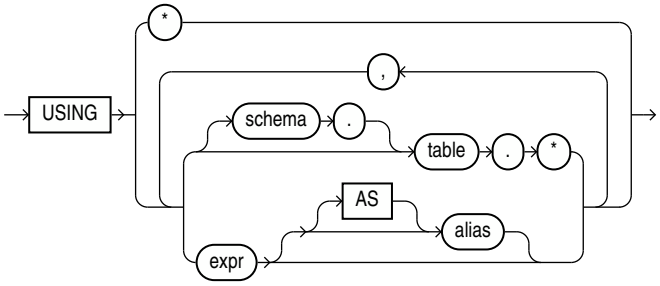
```
                          <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".059"
                           rank="2"/>
                          <Attribute name="CUST_MARITAL_STATUS" actualValue="Mabsent"
                           weight=".059" rank="3"/>
                          <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                           rank="4"/>
                          <Attribute name="OCCUPATION" actualValue="Prof." weight=".059"
                           rank="5"/>
                          </Details>

100694     77  41.04    35.96 <Details algorithm="Support Vector Machines">
                          <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".059"
                           rank="1"/>
                          <Attribute name="EDUCATION" actualValue="&lt; Bach." weight=".059"
                           rank="2"/>
                          <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                           rank="3"/>
                          <Attribute name="CUST_ID" actualValue="100694" weight=".059"
                           rank="4"/>
                          <Attribute name="COUNTRY_NAME" actualValue="United States of
                           America" weight=".059" rank="5"/>
                          </Details>
```

PREDICTION_BOUNDS

Syntax



***mining_attribute_clause*::=**



Purpose

`PREDICTION_BOUNDS` applies a Generalized Linear Model (GLM) to predict a class or a value for each row in the selection. The function returns the upper and lower bounds of each prediction in a varray of objects with fields `UPPER` and `LOWER`.

GLM can perform either regression or binary classification:

- The bounds for regression refer to the predicted target value. The data type of `UPPER` and `LOWER` is the data type of the target.
- The bounds for binary classification refer to the probability of either the predicted target class or the specified *`class_value`*. The data type of `UPPER` and `LOWER` is `BINARY_DOUBLE`.

If the model was built using ridge regression, or if the covariance matrix is found to be singular during the build, then `PREDICTION_BOUNDS` returns `NULL` for both bounds.

`confidence_level` is a number in the range (0,1). The default value is 0.95. You can specify *`class_value`* while leaving *`confidence_level`* at its default by specifying `NULL` for *`confidence_level`*.

The syntax of the `PREDICTION_BOUNDS` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

`mining_attribute_clause` identifies the column attributes to use as predictors for scoring. This clause behaves as described for the `PREDICTION` function. (Note that the reference to analytic syntax does not apply.) See "*mining_attribute_clause*::=".

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring
> - *Oracle Machine Learning for SQL Concepts* for information about Generalized Linear Models

> **Note:**
>
> The following example is excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

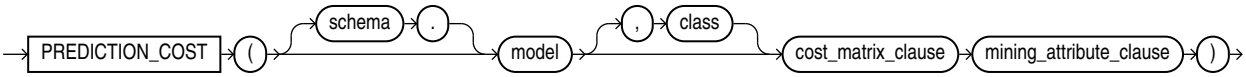
The following example returns the distribution of customers whose ages are predicted with 98% confidence to be greater than 24 and less than 46.

```
SELECT count(cust_id) cust_count, cust_marital_status
  FROM (SELECT cust_id, cust_marital_status
    FROM mining_data_apply_v
    WHERE PREDICTION_BOUNDS(glmr_sh_regr_sample,0.98 USING *).LOWER > 24 AND
          PREDICTION_BOUNDS(glmr_sh_regr_sample,0.98 USING *).UPPER < 46)
    GROUP BY cust_marital_status;

    CUST_COUNT CUST_MARITAL_STATUS
-------------- --------------------
            46 NeverM
             7 Mabsent
             5 Separ.
            35 Divorc.
            72 Married
```

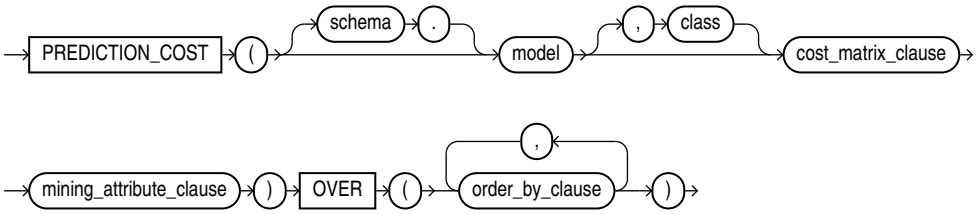
PREDICTION_COST

Syntax

***prediction_cost*::=**

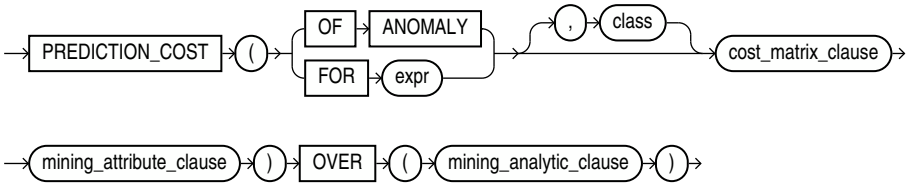


***prediction_cost_ordered*::=**

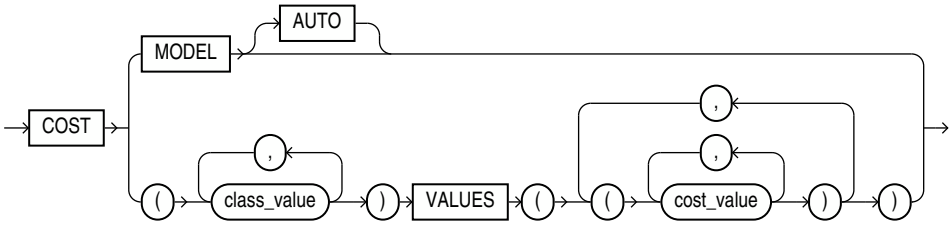


Analytic Syntax

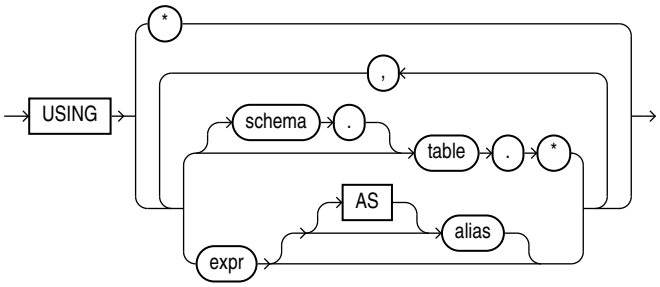
***prediction_cost_analytic*::=**



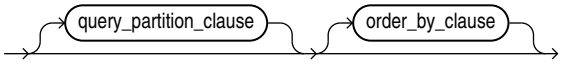
***cost_matrix_clause*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`PREDICTION_COST` returns a cost for each row in the selection. The cost refers to the lowest cost class or to the specified *class*. The cost is returned as `BINARY_DOUBLE`.

`PREDICTION_COST` can perform classification or anomaly detection. For classification, the returned cost refers to a predicted target class. For anomaly detection, the returned cost refers to a classification of `1` (for typical rows) or `0` (for anomalous rows).

You can use `PREDICTION_COST` in conjunction with the `PREDICTION` function to obtain the prediction and the cost of the prediction.

cost_matrix_clause

Costs are a biasing factor for minimizing the most harmful kinds of misclassifications. For example, false positives might be considered more costly than false negatives. Costs are specified in a cost matrix that can be associated with the model or defined inline in a `VALUES` clause. All classification algorithms can use costs to influence scoring.

Decision Tree is the only algorithm that can use costs to influence the model build. The cost matrix used to build a Decision Tree model is also the default scoring cost matrix for the model.

The following cost matrix table specifies that the misclassification of `1` is five times more costly than the misclassification of `0`.

```
ACTUAL_TARGET_VALUE PREDICTED_TARGET_VALUE       COST
------------------- ---------------------- ----------
                  0                      0          0
                  0                      1          1
                  1                      0          5
                  1                      1          0
```

In *cost_matrix_clause*:

- `COST MODEL` indicates that scoring should be performed by taking into account the scoring cost matrix associated with the model. If the cost matrix does not exist, then the function returns an error.
- `COST MODEL AUTO` indicates that the existence of a cost matrix is unknown. If a cost matrix exists, then the function uses it to return the lowest cost prediction. Otherwise the function returns the highest probability prediction.
- The `VALUES` clause specifies an inline cost matrix for *class_value*. For example, you could specify that the misclassification of `1` is five times more costly than the misclassification of `0` as follows:

```
PREDICTION (nb_model COST (0,1) VALUES ((0, 1),(1, 5)) USING *)
```

If a model that has a scoring cost matrix is invoked with an inline cost matrix, then the inline costs are used.

> **See Also:**
>
> *Oracle Machine Learning for SQL User's Guide* for more information about cost-sensitive prediction.

Syntax Choice

`PREDICTION_COST` can score the data by applying a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax**: Use the *`prediction_cost`* syntax to score the data with a pre-defined model. Supply the name of a model that performs classification or anomaly detection.

  Use the *`prediction_cost_ordered`* syntax for a model that requires ordered data, such as an MSET-SPRT model. The *`prediction_cost_ordered`* syntax requires an *`order_by_clause`* clause.

  Restrictions on the *`prediction_cost_ordered`* syntax are that you cannot use it in the `WHERE` clause of a query. Also, you cannot use a *`query_partition_clause`* or a *`windowing_clause`* with the *`prediction_ordered`* syntax.

- **Analytic Syntax**: Use the analytic syntax to score the data without a pre-defined model. The analytic syntax uses *`mining_analytic_clause`* , which specifies if the data should be partitioned for multiple model builds. The *`mining_analytic_clause`* supports a *`query_partition_clause`* and an *`order_by_clause`*. (See "*analytic_clause*::=".)
  - For classification, specify `FOR` *`expr`*, where *`expr`* is an expression that identifies a target column that has a character data type.
  - For anomaly detection, specify the keywords `OF ANOMALY`.

The syntax of the `PREDICTION_COST` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

`mining_attribute_clause` identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *`mining_attribute_clause`* behaves as described for the `PREDICTION` function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring.
> - *Oracle Machine Learning for SQL Concepts* for information about classification with costs

> **Note:**
>
> The following example is excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

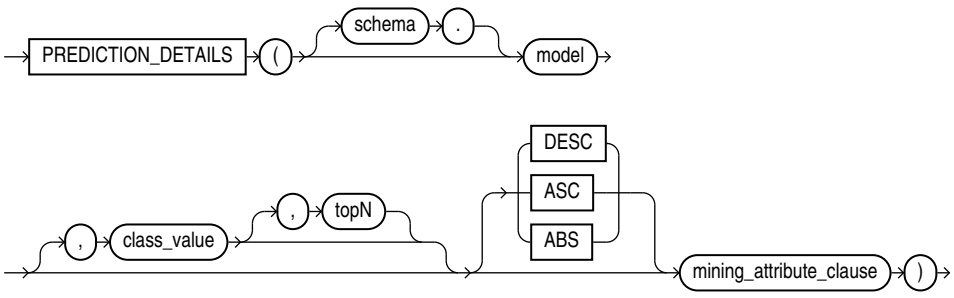
This example predicts the ten customers in Italy who would respond to the least expensive sales campaign (offering an affinity card).

```
SELECT cust_id
FROM (SELECT cust_id,rank()
       OVER (ORDER BY PREDICTION_COST(DT_SH_Clas_sample, 1 COST MODEL USING *)
            ASC, cust_id) rnk
        FROM mining_data_apply_v
        WHERE country_name = 'Italy')
  WHERE rnk <= 10
  ORDER BY rnk;

   CUST_ID
----------
    100081
    100179
    100185
    100324
    100344
    100554
    100662
    100733
    101250
    101306
```

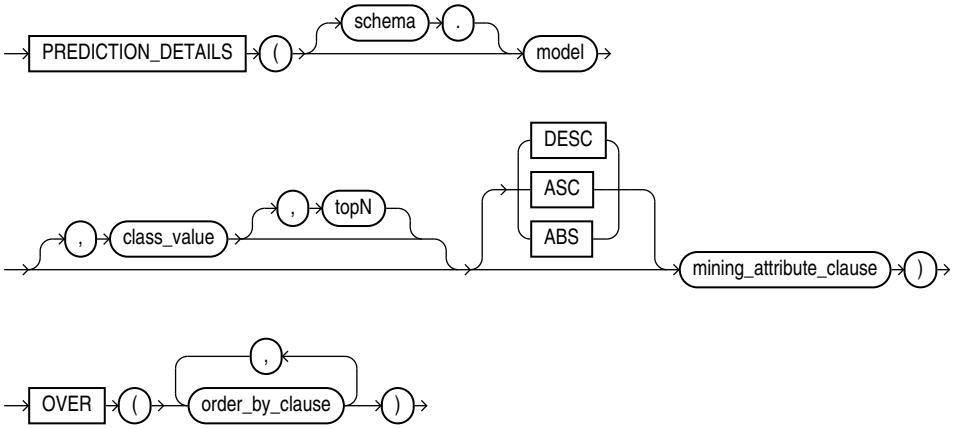
PREDICTION_DETAILS

Syntax

***prediction_details*::=**

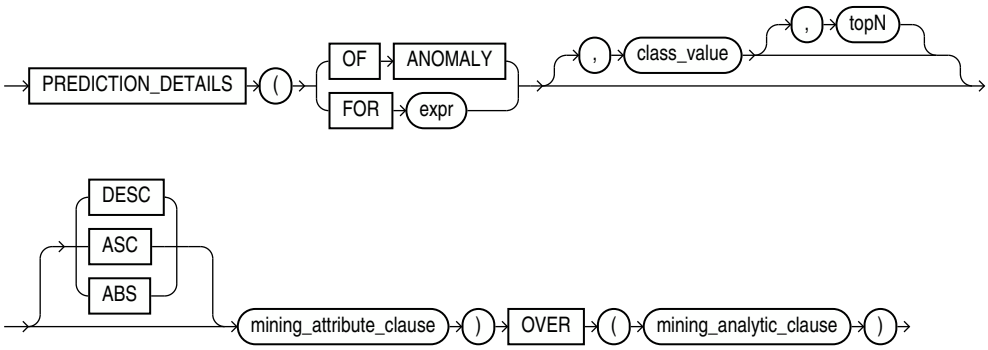


***prediction_details_ordered*::=**

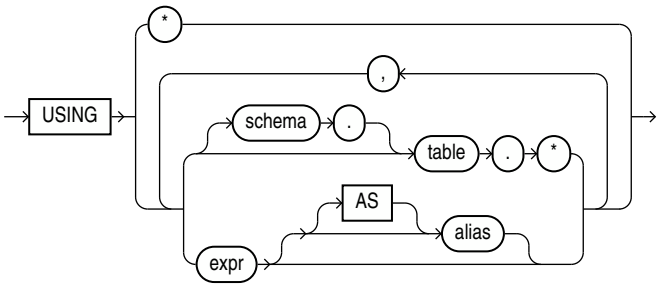


Analytic Syntax

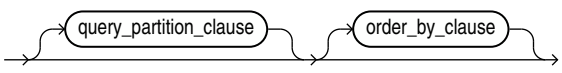
***prediction_details_analytic*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

PREDICTION_DETAILS returns prediction details for each row in the selection. The return value is an XML string that describes the attributes of the prediction.

For regression, the returned details refer to the predicted target value. For classification and anomaly detection, the returned details refer to the highest probability class or the specified *class_value*.

topN

If you specify a value for *topN*, the function returns the *N* attributes that have the most influence on the prediction (the score). If you do not specify *topN*, the function returns the 5 most influential attributes.

DESC, ASC, or ABS

The returned attributes are ordered by weight. The weight of an attribute expresses its positive or negative impact on the prediction. For regression, a positive weight indicates a higher value prediction; a negative weight indicates a lower value prediction. For classification and anomaly detection, a positive weight indicates a higher probability prediction; a negative weight indicates a lower probability prediction.

By default, PREDICTION_DETAILS returns the attributes with the highest positive weight (DESC). If you specify ASC, the attributes with the highest negative weight are returned. If you specify ABS, the attributes with the greatest weight, whether negative or positive, are returned. The results are ordered by absolute value from highest to lowest. Attributes with a zero weight are not included in the output.

Syntax Choice

PREDICTION_DETAILS can score the data by applying a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax**: Use the *prediction_details* syntax to score the data with a pre-defined model. Supply the name of a model that performs classification, regression, or anomaly detection.

  Use the *prediction_details_ordered* syntax for a model that requires ordered data, such as an MSET-SPRT model. The *prediction_details_ordered* syntax requires an *order_by_clause* clause.

  Restrictions on the *prediction_details_ordered* syntax are that you cannot use it in the WHERE clause of a query. Also, you cannot use a *query_partition_clause* or a *windowing_clause* with the *prediction_details_ordered* syntax.

- **Analytic Syntax**: Use the analytic syntax to score the data without a pre-defined model. The analytic syntax uses *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)

- For classification, specify `FOR expr`, where *expr* is an expression that identifies a target column that has a character data type.
- For regression, specify `FOR expr`, where *expr* is an expression that identifies a target column that has a numeric data type.
- For anomaly detection, specify the keywords `OF ANOMALY`.

The syntax of the `PREDICTION_DETAILS` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the `PREDICTION` function. (See "*mining_attribute_clause*::=".)

See Also:

- *Oracle Machine Learning for SQL User's Guide* for information about scoring.
- *Oracle Machine Learning for SQL Concepts* for information about predictive Oracle Machine Learning for SQL.

Note:

The following examples are excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

This example uses the model `svmr_sh_regr_sample` to score the data. The query returns the three attributes that have the greatest influence on predicting a higher value for customer age.

```
SELECT PREDICTION_DETAILS(svmr_sh_regr_sample, null, 3 USING *) prediction_details
    FROM mining_data_apply_v
    WHERE cust_id = 100001;

PREDICTION_DETAILS
--------------------------------------------------------------------------------
<Details algorithm="Support Vector Machines">
<Attribute name="CUST_MARITAL_STATUS" actualValue="Widowed" weight=".361" rank="1"/>
<Attribute name="CUST_GENDER" actualValue="F" weight=".14" rank="2"/>
<Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".135" rank="3"/>
</Details>
```

Analytic Syntax

This example dynamically identifies customers whose age is not typical for the data. The query returns the attributes that predict or detract from a typical age.

```
SELECT cust_id, age, pred_age, age-pred_age age_diff, pred_det
    FROM (SELECT cust_id, age, pred_age, pred_det,
```

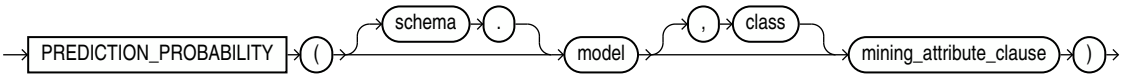
```
          RANK() OVER (ORDER BY ABS(age-pred_age) DESC) rnk
          FROM (SELECT cust_id, age,
             PREDICTION(FOR age USING *) OVER () pred_age,
             PREDICTION_DETAILS(FOR age ABS USING *) OVER () pred_det
             FROM mining_data_apply_v))
    WHERE rnk <= 5;

CUST_ID AGE PRED_AGE AGE_DIFF  PRED_DET
------- --- -------- -------- --------------------------------------------------------------------
 100910  80    40.67    39.33 <Details algorithm="Support Vector Machines">
                              <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".059"
                               rank="1"/>
                              <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                               rank="2"/>
                              <Attribute name="AFFINITY_CARD" actualValue="0" weight=".059"
                               rank="3"/>
                              <Attribute name="FLAT_PANEL_MONITOR" actualValue="1" weight=".059"
                               rank="4"/>
                              <Attribute name="YRS_RESIDENCE" actualValue="4" weight=".059"
                               rank="5"/>
                              </Details>

 101285  79    42.18    36.82  <Details algorithm="Support Vector Machines">
                               <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".059"
                                rank="1"/>
                               <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".059"
                                rank="2"/>
                               <Attribute name="CUST_MARITAL_STATUS" actualValue="Mabsent"
                                weight=".059" rank="3"/>
                               <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                                rank="4"/>
                               <Attribute name="OCCUPATION" actualValue="Prof." weight=".059"
                                rank="5"/>
                               </Details>

 100694   77    41.04    35.96  <Details algorithm="Support Vector Machines">
                                <Attribute name="HOME_THEATER_PACKAGE" actualValue="1"
                                 weight=".059" rank="1"/>
                                <Attribute name="EDUCATION" actualValue="&lt; Bach." weight=".059"
                                 rank="2"/>
                                <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                                 rank="3"/>
                                <Attribute name="CUST_ID" actualValue="100694" weight=".059"
                                 rank="4"/>
                                <Attribute name="COUNTRY_NAME" actualValue="United States of
                                 America" weight=".059" rank="5"/>
                                </Details>

 100308  81    45.33    35.67  <Details algorithm="Support Vector Machines">
                               <Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".059"
                                rank="1"/>
                               <Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
                                rank="2"/>
                               <Attribute name="HOUSEHOLD_SIZE" actualValue="2" weight=".059"
                                rank="3"/>
                               <Attribute name="FLAT_PANEL_MONITOR" actualValue="1" weight=".059"
                                rank="4"/>
                               <Attribute name="CUST_GENDER" actualValue="F" weight=".059"
                                rank="5"/>
                               </Details>

 101256  90    54.39    35.61  <Details algorithm="Support Vector Machines">
```

```
<Attribute name="YRS_RESIDENCE" actualValue="9" weight=".059"
 rank="1"/>
<Attribute name="HOME_THEATER_PACKAGE" actualValue="1" weight=".059"
 rank="2"/>
<Attribute name="EDUCATION" actualValue="&lt; Bach." weight=".059"
 rank="3"/>
<Attribute name="Y_BOX_GAMES" actualValue="0" weight=".059"
 rank="4"/>
<Attribute name="COUNTRY_NAME" actualValue="United States of
 America" weight=".059" rank="5"/>
</Details>
```

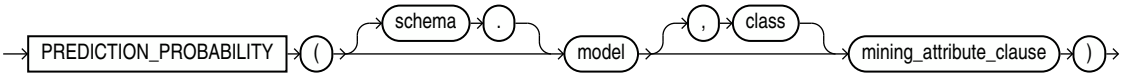
PREDICTION_PROBABILITY

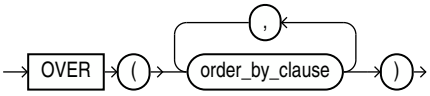
Syntax

***prediction_probability*::=**



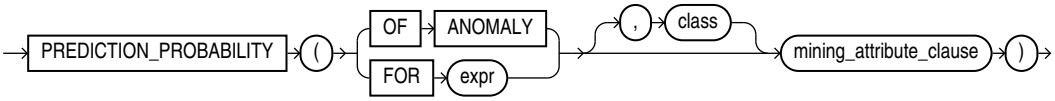
***prediction_probability_ordered*::=**

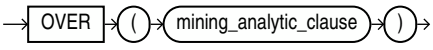




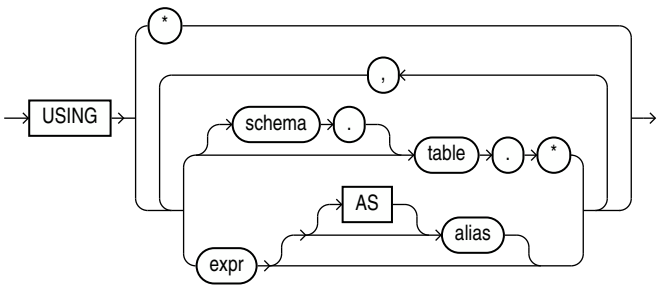
Analytic Syntax

***prediction_prob_analytic*::=**

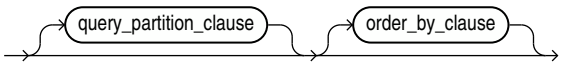




***mining_attribute_clause*::=**



***mining_analytic_clause*::=**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`PREDICTION_PROBABILITY` returns a probability for each row in the selection. The probability refers to the highest probability class or to the specified *class*. The data type of the returned probability is `BINARY_DOUBLE`.

`PREDICTION_PROBABILITY` can perform classification or anomaly detection. For classification, the returned probability refers to a predicted target class. For anomaly detection, the returned probability refers to a classification of `1` (for typical rows) or `0` (for anomalous rows).

You can use `PREDICTION_PROBABILITY` in conjunction with the `PREDICTION` function to obtain the prediction and the probability of the prediction.

Syntax Choice

`PREDICTION_PROBABILITY` can score the data by applying a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax**: Use the *prediction_probability* syntax to score the data with a pre-defined model. Supply the name of a model that performs classification or anomaly detection.

  Use the *prediction_probability_ordered* syntax for a model that requires ordered data, such as an MSET-SPRT model. The *prediction_probability_ordered* syntax requires an *order_by_clause* clause.

  Restrictions on the *prediction_probability_ordered* syntax are that you cannot use it in the `WHERE` clause of a query. Also, you cannot use a *query_partition_clause* or a *windowing_clause* with the *prediction_probability_ordered* syntax.

- **Analytic Syntax**: Use the analytic syntax to score the data without a pre-defined model. The analytic syntax uses *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)
  - For classification, specify `FOR` *expr*, where *expr* is an expression that identifies a target column that has a character data type.
  - For anomaly detection, specify the keywords `OF ANOMALY`.

The syntax of the `PREDICTION_PROBABILITY` function can use an optional `GROUPING` hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the PREDICTION function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring.
> - *Oracle Machine Learning for SQL Concepts* for information about predictive Oracle Machine Learning for SQL.

> **Note:**
>
> The following examples are excerpted from the Oracle Machine Learning for SQL sample programs. For information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

Example

The following example returns the 10 customers living in Italy who are most likely to use an affinity card.

```
SELECT cust_id FROM (
   SELECT cust_id
   FROM mining_data_apply_v
   WHERE country_name = 'Italy'
   ORDER BY PREDICTION_PROBABILITY(DT_SH_Clas_sample, 1 USING *)
      DESC, cust_id)
   WHERE rownum < 11;

   CUST_ID
----------
    100081
    100179
    100185
    100324
    100344
    100554
    100662
    100733
    101250
    101306
```

Analytic Example

This example identifies rows that are most atypical in the data in mining_data_one_class_v. Each type of marital status is considered separately so that the most anomalous rows per marital status group are returned.

The query returns three attributes that have the most influence on the determination of anomalous rows. The PARTITION BY clause causes separate models to be built and applied for

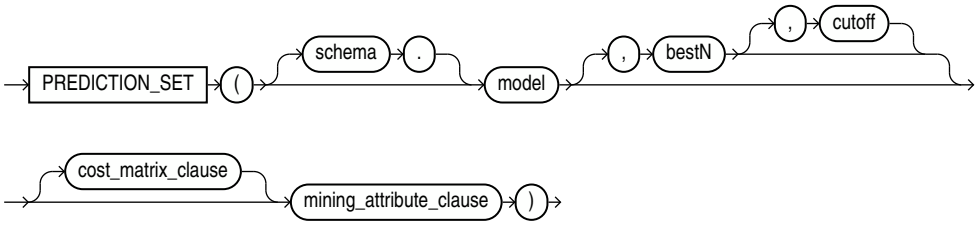
each marital status. Because there is only one record with status Mabsent, no model is created for that partition (and no details are provided).

```
SELECT cust_id, cust_marital_status, rank_anom, anom_det FROM
    (SELECT cust_id, cust_marital_status, anom_det,
            rank() OVER (PARTITION BY CUST_MARITAL_STATUS
                         ORDER BY ANOM_PROB DESC,cust_id) rank_anom FROM
     (SELECT cust_id, cust_marital_status,
            PREDICTION_PROBABILITY(OF ANOMALY, 0 USING *)
              OVER (PARTITION BY CUST_MARITAL_STATUS) anom_prob,
            PREDICTION_DETAILS(OF ANOMALY, 0, 3 USING *)
              OVER (PARTITION BY CUST_MARITAL_STATUS) anom_det
     FROM mining_data_one_class_v
    ))
   WHERE rank_anom < 3 order by 2, 3;

CUST_ID CUST_MARITAL_STATUS RANK_ANOM  ANOM_DET
------- ------------------- ---------- ------------------------------------------------------------
102366  Divorc.             1          <Details algorithm="Support Vector Machines" class="0">
                                       <Attribute name="COUNTRY_NAME" actualValue="United Kingdom"
                                        weight=".069" rank="1"/>
                                       <Attribute name="AGE" actualValue="28" weight=".013"
                                        rank="2"/>
                                       <Attribute name="YRS_RESIDENCE" actualValue="4"
                                        weight=".006" rank="3"/>
                                       </Details>

101817  Divorc.             2          <Details algorithm="Support Vector Machines" class="0">
                                       <Attribute name="YRS_RESIDENCE" actualValue="8"
                                        weight=".018" rank="1"/>
                                       <Attribute name="EDUCATION" actualValue="PhD" weight=".007"
                                        rank="2"/>
                                       <Attribute name="CUST_INCOME_LEVEL" actualValue="K:
                                        250\,000 - 299\,999" weight=".006" rank="3"/>
                                       </Details>

101713  Mabsent             1

101790  Married             1          <Details algorithm="Support Vector Machines" class="0">
                                       <Attribute name="COUNTRY_NAME" actualValue="Canada"
                                        weight=".063" rank="1"/>
                                       <Attribute name="EDUCATION" actualValue="7th-8th"
                                        weight=".011" rank="2"/>
                                       <Attribute name="HOUSEHOLD_SIZE" actualValue="4-5"
                                        weight=".011" rank="3"/>
                                       </Details>
. . .
```

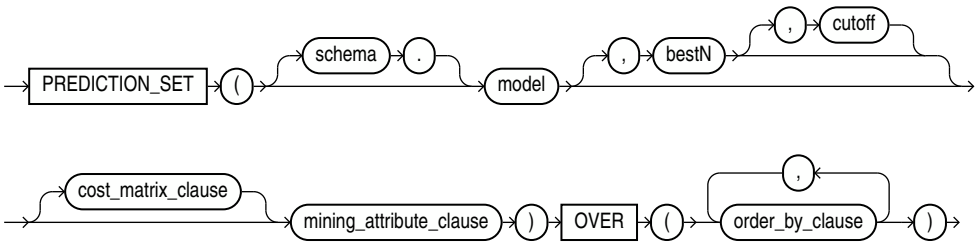
PREDICTION_SET

Syntax

***prediction_set*::=**

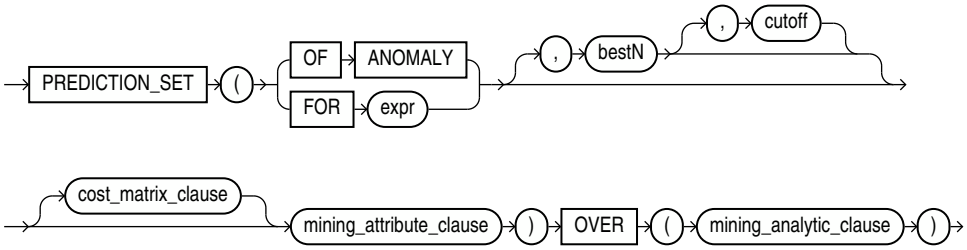


***prediction_set_ordered*::=**

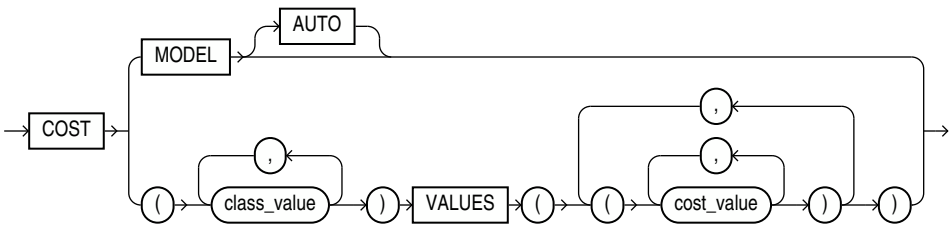


Analytic Syntax

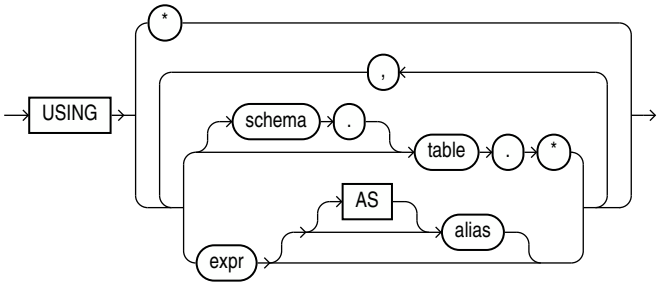
***prediction_set_analytic*::=**



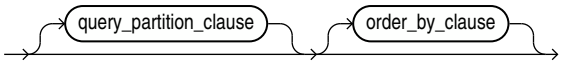
***cost_matrix_clause*::=**



***mining_attribute_clause*::=**



***mining_analytic_clause*::-**



> **See Also:**
>
> "Analytic Functions " for information on the syntax, semantics, and restrictions of *mining_analytic_clause*

Purpose

`PREDICTION_SET` returns a set of predictions with either probabilities or costs for each row in the selection. The return value is a varray of objects with field names `PREDICTION_ID` and `PROBABILITY` or `COST`. The prediction identifier has the data type of the target; the probability and cost fields are `BINARY_DOUBLE`.

`PREDICTION_SET` can perform classification or anomaly detection. For classification, the return value refers to a predicted target class. For anomaly detection, the return value refers to a classification of `1` (for typical rows) or `0` (for anomalous rows).

bestN and cutoff

You can specify *bestN* and *cutoff* to limit the number of predictions returned by the function. By default, both *bestN* and *cutoff* are null and all predictions are returned.

- *bestN* is the *N* predictions that are either the most probable or the least costly. If multiple predictions share the *N*th probability or cost, then the function chooses one of them.
- *cutoff* is a value threshold. Only predictions with probability greater than or equal to *cutoff*, or with cost less than or equal to *cutoff*, are returned. To filter by *cutoff* only, specify `NULL` for *bestN*. If the function uses a *cost_matrix_clause* with `COST MODEL AUTO`, then *cutoff* is ignored.

You can specify *bestN* with *cutoff* to return up to the *N* most probable predictions that are greater than or equal to *cutoff*. If costs are used, specify *bestN* with *cutoff* to return up to the *N* least costly predictions that are less than or equal to cutoff.

cost_matrix_clause

You can specify *cost_matrix_clause* as a biasing factor for minimizing the most harmful kinds of misclassifications. *cost_matrix_clause* behaves as described for "PREDICTION_COST ".

Syntax Choice

PREDICTION_SET can score the data by applying a mining model object to the data, or it can dynamically mine the data by executing an analytic clause that builds and applies one or more transient mining models. Choose **Syntax** or **Analytic Syntax**:

- **Syntax**: Use the *prediction_set* syntax to score the data with a pre-defined model. Supply the name of a model that performs classification or anomaly detection.

  Use the *prediction_set_ordered* syntax for a model that requires ordered data, such as an MSET-SPRT model. The *prediction_set_ordered* syntax requires an *order_by_clause* clause.

  Restrictions on the *prediction_set_ordered* syntax are that you cannot use it in the WHERE clause of a query. Also, you cannot use a *query_partition_clause* or a *windowing_clause* with the *prediction_set_ordered* syntax.

- **Analytic Syntax**: Use the analytic syntax to score the data without a pre-defined model. The analytic syntax uses *mining_analytic_clause*, which specifies if the data should be partitioned for multiple model builds. The *mining_analytic_clause* supports a *query_partition_clause* and an *order_by_clause*. (See "*analytic_clause*::=".)

  – For classification, specify FOR *expr*, where *expr* is an expression that identifies a target column that has a character data type.

  – For anomaly detection, specify the keywords OF ANOMALY.

The syntax of the PREDICTION_SET function can use an optional GROUPING hint when scoring a partitioned model. See GROUPING Hint.

mining_attribute_clause

mining_attribute_clause identifies the column attributes to use as predictors for scoring. When the function is invoked with the analytic syntax, these predictors are also used for building the transient models. The *mining_attribute_clause* behaves as described for the PREDICTION function. (See "*mining_attribute_clause*::=".)

> **See Also:**
>
> - *Oracle Machine Learning for SQL User's Guide* for information about scoring.
> - *Oracle Machine Learning for SQL Concepts* for information about predictive Oracle Machine Learning for SQL.

> **Note:**
>
> The following example is excerpted from the Oracle Machine Learning for SQL sample programs. For more information about the sample programs, see Appendix A in *Oracle Machine Learning for SQL User's Guide*.

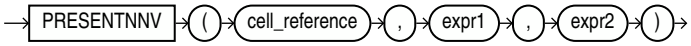
Example

This example lists the probability and cost that customers with ID less than 100006 will use an affinity card. This example has a binary target, but such a query is also useful for multiclass classification such as low, medium, and high.

```
SELECT T.cust_id, S.prediction, S.probability, S.cost
  FROM (SELECT cust_id,
               PREDICTION_SET(dt_sh_clas_sample COST MODEL USING *) pset
          FROM mining_data_apply_v
         WHERE cust_id < 100006) T,
       TABLE(T.pset) S
ORDER BY cust_id, S.prediction;

   CUST_ID PREDICTION  PROBABILITY         COST
---------- ---------- ------------ ------------
    100001          0   .966183575   .270531401
    100001          1   .033816425   .966183575
    100002          0   .740384615  2.076923077
    100002          1   .259615385   .740384615
    100003          0   .909090909   .727272727
    100003          1   .090909091   .909090909
    100004          0   .909090909   .727272727
    100004          1   .090909091   .909090909
    100005          0   .272357724  5.821138211
    100005          1   .727642276   .272357724
```

PRESENTNNV

Syntax



Purpose

The `PRESENTNNV` function can be used only in the *`model_clause`* of the `SELECT` statement and then only on the right-hand side of a model rule. It returns *`expr1`* when *`cell_reference`* exists prior to the execution of the *`model_clause`* and is not null when `PRESENTNNV` is evaluated. Otherwise it returns *`expr2`*. This function differs from `NVL2` in that `NVL2` evaluates the data at the time it is executed, rather than evaluating the data as it was prior to the execution of the *`model_clause`*.

> **See Also:**
>
> - *model_clause* and "Model Expressions" for the syntax and semantics
> - NVL2 for comparison
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `PRESENTNNV` when it is a character value

Examples

In the following example, if a row containing sales for the Mouse Pad for the year 2002 exists, and the sales value is not null, then the sales value remains unchanged. If the row exists and the sales value is null, then the sales value is set to 10. If the row does not exist, then the row is created with the sales value set to 10.

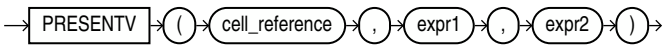
```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    ( s['Mouse Pad', 2002] =
        PRESENTNNV(s['Mouse Pad', 2002], s['Mouse Pad', 2002], 10)
    )
  ORDER BY country, prod, year;
```

```
COUNTRY       PROD                                     YEAR          S
----------    ----------------------------------    --------   ---------
France        Mouse Pad                                1998     2509.42
France        Mouse Pad                                1999     3678.69
France        Mouse Pad                                2000     3000.72
France        Mouse Pad                                2001     3269.09
France        Mouse Pad                                2002          10
France        Standard Mouse                           1998     2390.83
France        Standard Mouse                           1999     2280.45
France        Standard Mouse                           2000     1274.31
France        Standard Mouse                           2001     2164.54
Germany       Mouse Pad                                1998     5827.87
Germany       Mouse Pad                                1999     8346.44
Germany       Mouse Pad                                2000     7375.46
Germany       Mouse Pad                                2001     9535.08
Germany       Mouse Pad                                2002          10
Germany       Standard Mouse                           1998     7116.11
Germany       Standard Mouse                           1999     6263.14
Germany       Standard Mouse                           2000     2637.31
Germany       Standard Mouse                           2001     6456.13

18 rows selected.
```

The preceding example requires the view `sales_view_ref`. Refer to "Examples" to create this view.

PRESENTV

Syntax



Purpose

The `PRESENTV` function can be used only within the *`model_clause`* of the `SELECT` statement and then only on the right-hand side of a model rule. It returns *`expr1`* when, prior to the execution of the *`model_clause`*, *`cell_reference`* exists. Otherwise it returns *`expr2`*.

> **See Also:**
>
> - *model_clause* and "Model Expressions" for the syntax and semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `PRESENTV` when it is a character value

Examples

In the following example, if a row containing sales for the Mouse Pad for the year 2000 exists, then the sales value for the Mouse Pad for the year 2001 is set to the sales value for the Mouse Pad for the year 2000. If the row does not exist, then a row is created with the sales value for the Mouse Pad for year 20001 set to 0.

```
SELECT country, prod, year, s
  FROM sales_view_ref
  MODEL
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale s)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES UPSERT SEQUENTIAL ORDER
    (
      s['Mouse Pad', 2001] =
        PRESENTV(s['Mouse Pad', 2000], s['Mouse Pad', 2000], 0)
    )
  ORDER BY country, prod, year;
```

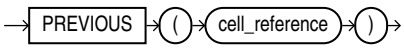
```
COUNTRY       PROD                                    YEAR          S
----------    ----------------------------------    --------   ---------
France        Mouse Pad                                 1998     2509.42
France        Mouse Pad                                 1999     3678.69
France        Mouse Pad                                 2000     3000.72
France        Mouse Pad                                 2001     3000.72
France        Standard Mouse                            1998     2390.83
France        Standard Mouse                            1999     2280.45
France        Standard Mouse                            2000     1274.31
France        Standard Mouse                            2001     2164.54
Germany       Mouse Pad                                 1998     5827.87
```

```
Germany       Mouse Pad                                 1999     8346.44
Germany       Mouse Pad                                 2000     7375.46
Germany       Mouse Pad                                 2001     7375.46
Germany       Standard Mouse                            1998     7116.11
Germany       Standard Mouse                            1999     6263.14
Germany       Standard Mouse                            2000     2637.31
Germany       Standard Mouse                            2001     6456.13

16 rows selected.
```

The preceding example requires the view `sales_view_ref`. Refer to "The MODEL clause: Examples" to create this view.

PREVIOUS

Syntax



Purpose

The `PREVIOUS` function can be used only in the *`model_clause`* of the `SELECT` statement and then only in the `ITERATE` ... [`UNTIL`] clause of the *`model_rules_clause`*. It returns the value of *`cell_reference`* at the beginning of each iteration.

> **See Also:**
>
> - *model_clause* and "Model Expressions" for the syntax and semantics
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `PREVIOUS` when it is a character value

Examples

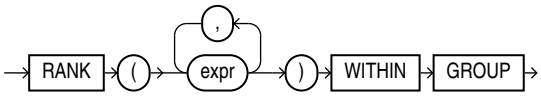
The following example repeats the rules, up to 1000 times, until the difference between the values of `cur_val` at the beginning and at the end of an iteration is less than one:

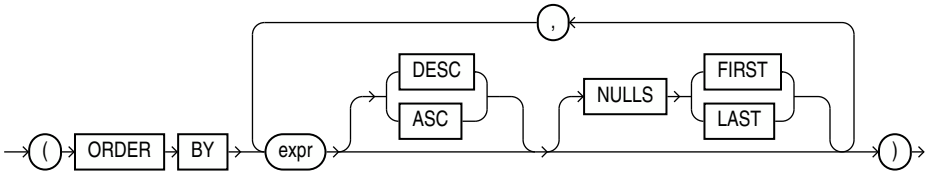
```
SELECT dim_col, cur_val, num_of_iterations
  FROM (SELECT 1 AS dim_col, 10 AS cur_val FROM dual)
  MODEL
    DIMENSION BY (dim_col)
    MEASURES (cur_val, 0 num_of_iterations)
    IGNORE NAV
    UNIQUE DIMENSION
    RULES ITERATE (1000) UNTIL (PREVIOUS(cur_val[1]) - cur_val[1] < 1)
    (
      cur_val[1] = cur_val[1]/2,
      num_of_iterations[1] = num_of_iterations[1] + 1
    );

   DIM_COL    CUR_VAL NUM_OF_ITERATIONS
---------- ---------- -----------------
         1       .625                 4
```

RANK

Aggregate Syntax

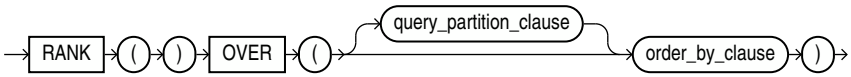
***rank_aggregate*::=**





Analytic Syntax

***rank_analytic*::=**



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`RANK` calculates the rank of a value in a group of values. The return type is `NUMBER`.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

Rows with equal values for the ranking criteria receive the same rank. Oracle Database then adds the number of tied rows to the tied rank to calculate the next rank. Therefore, the ranks may not be consecutive numbers. This function is useful for top-N and bottom-N reporting.

- As an aggregate function, `RANK` calculates the rank of a hypothetical row identified by the arguments of the function with respect to a given sort specification. The arguments of the function must all evaluate to constant expressions within each aggregate group, because they identify a single row within each group. The constant argument expressions and the

expressions in the `ORDER BY` clause of the aggregate match by position. Therefore, the number of arguments must be the same and their types must be compatible.

- As an analytic function, `RANK` computes the rank of each row returned from a query with respect to the other rows returned by the query, based on the values of the *`value_exprs`* in the *`order_by_clause`*.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `RANK` uses to compare character values for the `ORDER BY` clause

Aggregate Example

The following example calculates the rank of a hypothetical employee in the sample table `hr.employees` with a salary of $15,500 and a commission of 5%:

```
SELECT RANK(15500, .05) WITHIN GROUP
   (ORDER BY salary, commission_pct) "Rank"
   FROM employees;

      Rank
----------
       105
```

Similarly, the following query returns the rank for a $15,500 salary among the employee salaries:

```
SELECT RANK(15500) WITHIN GROUP
   (ORDER BY salary DESC) "Rank of 15500"
   FROM employees;

Rank of 15500
--------------
             4
```

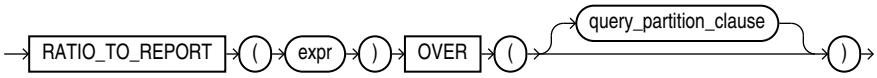
Analytic Example

The following statement ranks the employees in the sample `hr` schema in department 60 based on their salaries. Identical salary values receive the same rank and cause nonconsecutive ranks. Compare this example with the analytic example for DENSE_RANK .

```
SELECT department_id, last_name, salary,
       RANK() OVER (PARTITION BY department_id ORDER BY salary) RANK
  FROM employees WHERE department_id = 60
  ORDER BY RANK, last_name;

DEPARTMENT_ID LAST_NAME                     SALARY       RANK
------------- ------------------------- ---------- ----------
           60 Lorentz                         4200          1
           60 Austin                          4800          2
           60 Pataballa                       4800          2
           60 Ernst                           6000          4
           60 Hunold                          9000          5
```

RATIO_TO_REPORT

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions, including valid forms of *expr*

Purpose

`RATIO_TO_REPORT` is an analytic function. It computes the ratio of a value to the sum of a set of values. If *expr* evaluates to null, then the ratio-to-report value also evaluates to null.

The set of values is determined by the *query_partition_clause*. If you omit that clause, then the ratio-to-report is computed over all rows returned by the query.

You cannot nest analytic functions by using `RATIO_TO_REPORT` or any other analytic function for *expr*. However, you can use other built-in function expressions for *expr*. Refer to "About SQL Expressions " for information on valid forms of *expr*.

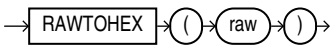
Examples

The following example calculates the ratio-to-report value of each purchasing clerk's salary to the total of all purchasing clerks' salaries:

```
SELECT last_name, salary, RATIO_TO_REPORT(salary) OVER () AS rr
   FROM employees
   WHERE job_id = 'PU_CLERK'
   ORDER BY last_name, salary, rr;

LAST_NAME                 SALARY         RR
------------------------- ---------- ----------
Baida                           2900 .208633094
Colmenares                      2500 .179856115
Himuro                          2600  .18705036
Khoo                            3100 .223021583
Tobias                          2800 .201438849
```

RAWTOHEX

Syntax



Purpose

`RAWTOHEX` converts *raw* to a character value containing its hexadecimal representation.

As a SQL built-in function, `RAWTOHEX` accepts an argument of any scalar data type other than `LONG`, `LONG RAW`, `CLOB`, `NCLOB`, `BLOB`, or `BFILE`. If the argument is of a data type other than `RAW`, then this function converts the argument value, which is represented using some number of data bytes, into a `RAW` value with the same number of data bytes. The data itself is not modified in any way, but the data type is recast to a `RAW` data type.

This function returns a `VARCHAR2` value with the hexadecimal representation of bytes that make up the value of *raw*. Each byte is represented by two hexadecimal digits.

> **Note:**
>
> `RAWTOHEX` functions differently when used as a PL/SQL built-in function. Refer to *Oracle Database Development Guide* for more information.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `RAWTOHEX`

Examples

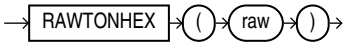
The following hypothetical example returns the hexadecimal equivalent of a `RAW` column value:

```
SELECT RAWTOHEX(raw_column) "Graphics"
   FROM graphics;

Graphics
--------
7D
```

> **See Also:**
>
> "RAW and LONG RAW Data Types " and HEXTORAW

RAWTONHEX

Syntax



Purpose

`RAWTONHEX` converts *raw* to a character value containing its hexadecimal representation. `RAWTONHEX(raw)` is equivalent to `TO_NCHAR(RAWTOHEX(raw))`. The value returned is always in the national character set.

> **Note:**
>
> `RAWTONHEX` functions differently when used as a PL/SQL built-in function. Refer to *Oracle Database Development Guide* for more information.

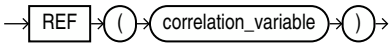
Examples

The following hypothetical example returns the hexadecimal equivalent of a `RAW` column value:

```
SELECT RAWTONHEX(raw_column),
   DUMP ( RAWTONHEX (raw_column) ) "DUMP"
   FROM graphics;

RAWTONHEX(RA)            DUMP
------------------------ ------------------------------
7D                       Typ=1 Len=4: 0,55,0,68
```

REF

Syntax



Purpose

`REF` takes as its argument a correlation variable (table alias) associated with a row of an object table or an object view. A `REF` value is returned for the object instance that is bound to the variable or row.

Examples

The sample schema `oe` contains a type called `cust_address_typ`, described as follows:

```
 Attribute                      Type
 ------------------------------ ----------------
 STREET_ADDRESS                 VARCHAR2(40)
 POSTAL_CODE                    VARCHAR2(10)
 CITY                           VARCHAR2(30)
 STATE_PROVINCE                 VARCHAR2(10)
 COUNTRY_ID                          CHAR(2)
```

The following example creates a table based on the sample type `oe.cust_address_typ`, inserts a row into the table, and retrieves a `REF` value for the object instance of the type in the addresses table:

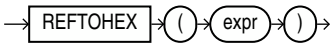
```
CREATE TABLE addresses OF cust_address_typ;
```

```
INSERT INTO addresses VALUES (
   '123 First Street', '4GF H1J', 'Our Town', 'Ourcounty', 'US');

SELECT REF(e) FROM addresses e;

REF(E)
--------------------------------------------------------------------------------
00002802097CD1261E51925B60E0340800208254367CD1261E51905B60E0340800208254360101018200000
```

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for information on `REF`s

REFTOHEX

Syntax



Purpose

`REFTOHEX` converts argument *`expr`* to a character value containing its hexadecimal equivalent. *`expr`* must return a `REF`.

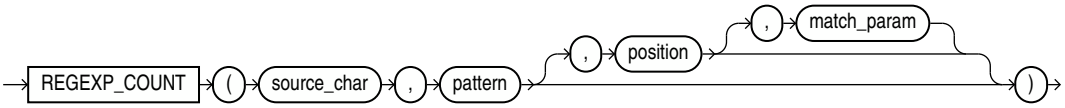
Examples

The sample schema `oe` contains a `warehouse_typ`. The following example builds on that type to illustrate how to convert the `REF` value of a column to a character value containing its hexadecimal equivalent:

```
CREATE TABLE warehouse_table OF warehouse_typ
   (PRIMARY KEY (warehouse_id));

CREATE TABLE location_table
   (location_number NUMBER, building REF warehouse_typ
   SCOPE IS warehouse_table);

INSERT INTO warehouse_table VALUES (1, 'Downtown', 99);

INSERT INTO location_table SELECT 10, REF(w) FROM warehouse_table w;

SELECT REFTOHEX(building) FROM location_table;

REFTOHEX(BUILDING)
--------------------------------------------------------------------------------
0000220208859B5E9255C31760E0340800208254368 59B5E9255C21760E034080020825436
```

REGEXP_COUNT

Syntax



Purpose

`REGEXP_COUNT` complements the functionality of the `REGEXP_INSTR` function by returning the number of times a pattern occurs in a source string. The function evaluates strings using characters as defined by the input character set. It returns an integer indicating the number of occurrences of *`pattern`*. If no match is found, then the function returns 0.

- *`source_char`* is a character expression that serves as the search value. It is commonly a character column and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`.
- *`pattern`* is the regular expression. It is usually a text literal and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. It can contain up to 512 bytes. If the data type of *`pattern`* is different from the data type of *`source_char`*, then Oracle Database converts *`pattern`* to the data type of *`source_char`*.

  `REGEXP_COUNT` ignores subexpression parentheses in *`pattern`*. For example, the pattern `'(123(45))'` is equivalent to `'12345'`. For a listing of the operators you can specify in *`pattern`*, refer to Oracle Regular Expression Support.
- *`position`* is a positive integer indicating the character of *`source_char`* where Oracle should begin the search. The default is 1, meaning that Oracle begins the search at the first character of *`source_char`*. After finding the first occurrence of *`pattern`*, the database searches for a second occurrence beginning with the first character following the first occurrence.
- *`match_param`* is a character expression of the data type `VARCHAR2` or `CHAR` that lets you change the default matching behavior of the function.

  The value of *`match_param`* can include one or more of the following characters:

  - `'i'` specifies case-insensitive matching, even if the determined collation of the condition is case-sensitive.
  - `'c'` specifies case-sensitive and accent-sensitive matching, even if the determined collation of the condition is case-insensitive or accent-insensitive.
  - `'n'` allows the period (.), which is the match-any-character character, to match the newline character. If you omit this parameter, then the period does not match the newline character.
  - `'m'` treats the source string as multiple lines. Oracle interprets the caret (`^`) and dollar sign (`$`) as the start and end, respectively, of any line anywhere in the source string, rather than only at the start or end of the entire source string. If you omit this parameter, then Oracle treats the source string as a single line.
  - `'x'` ignores whitespace characters. By default, whitespace characters match themselves.

If the value of *match_param* contains multiple contradictory characters, then Oracle uses the last character. For example, if you specify `'ic'`, then Oracle uses case-sensitive and accent-sensitive matching. If the value contains a character other than those shown above, then Oracle returns an error.

If you omit *match_param*, then:

- The default case and accent sensitivity are determined by the determined collation of the `REGEXP_COUNT` function.
- A period (.) does not match the newline character.
- The source string is treated as a single line.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `REGEXP_COUNT` uses to compare characters from *source_char* with characters from *pattern*

Examples

The following example shows that subexpressions parentheses in pattern are ignored:

```
SELECT REGEXP_COUNT('123123123123123', '(12)3', 1, 'i') REGEXP_COUNT
   FROM DUAL;

REGEXP_COUNT
------------
           5
```

In the following example, the function begins to evaluate the source string at the third character, so skips over the first occurrence of pattern:

```
SELECT REGEXP_COUNT('123123123123', '123', 3, 'i') COUNT FROM DUAL;

     COUNT
----------
         3
```

REGEXP_COUNT simple matching: Examples

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters:

```
select regexp_count('ABC123', '[A-Z]'), regexp_count('A1B2C3', '[A-Z]') from dual;

REGEXP_COUNT('ABC123','[A-Z]') REGEXP_COUNT('A1B2C3','[A-Z]')
------------------------------ ------------------------------
                             3                              3
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number:

```
select regexp_count('ABC123', '[A-Z][0-9]'), regexp_count('A1B2C3', '[A-Z][0-9]') from
dual;

REGEXP_COUNT('ABC123','[A-Z][0-9]') REGEXP_COUNT('A1B2C3','[A-Z][0-9]')
```

```
---------------------------------- ----------------------------------
                 1                                  3
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number only at the beginning of the string:

```
select regexp_count('ABC123', '[A-Z][0-9]'), regexp_count('A1B2C3', '[A-Z][0-9]') from
dual;

REGEXP_COUNT('ABC123','^[A-Z][0-9]') REGEXP_COUNT('A1B2C3','^[A-Z][0-9]')
---------------------------------- ----------------------------------
                  0                                  1
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by two digits of number only contained within the string:

```
select regexp_count('ABC123', '[A-Z][0-9]{2}'), regexp_count('A1B2C3', '[A-Z][0-9]{2}')
from dual;

REGEXP_COUNT('ABC123','[A-Z][0-9]{2}') REGEXP_COUNT('A1B2C3','[A-Z][0-9]{2}')
------------------------------------- -------------------------------------
                     1                                   0
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number within the first two occurrences from the beginning of the string:

```
select regexp_count('ABC123', '([A-Z][0-9]){2}'), regexp_count('A1B2C3', '([A-Z][0-9])
{2}') from dual;

REGEXP_COUNT('ABC123','([A-Z][0-9]){2}') REGEXP_COUNT('A1B2C3','([A-Z][0-9]){2}')
---------------------------------------- ----------------------------------------
                                       0                                        1
```

> **Live SQL:**
>
> View and run related examples on Oracle Live SQL at *REGEXP_COUNT simple matching*

REGEXP_COUNT advanced matching: Examples

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters:

```
select regexp_count('ABC123', '[A-Z]') Match_char_ABC_count,
regexp_count('A1B2C3', '[A-Z]') Match_char_ABC_count from dual;

MATCH_CHAR_ABC_COUNT MATCH_CHAR_ABC_COUNT
-------------------- --------------------
           3                    3
```

In the following example, `REGEXP_COUNT` validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number:

```
select regexp_count('ABC123', '[A-Z][0-9]') Match_string_C1_count,
regexp_count('A1B2C3', '[A-Z][0-9]')  Match_strings_A1_B2_C3_count from dual;
```

```
MATCH_STRING_C1_COUNT MATCH_STRINGS_A1_B2_C3_COUNT
--------------------- ----------------------------
                    1                            3
```

In the following example, REGEXP_COUNT validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number only at the beginning of the string:

```
select regexp_count('ABC123A5', '^[A-Z][0-9]') Char_num_like_A1_at_start,
regexp_count('A1B2C3', '^[A-Z][0-9]') Char_num_like_A1_at_start from dual;

CHAR_NUM_LIKE_A1_AT_START CHAR_NUM_LIKE_A1_AT_START
------------------------- -------------------------
                        0                         1
```

In the following example, REGEXP_COUNT validates the supplied string for the given pattern and returns the number of alphabetic letters followed by two digits of number only contained within the string:

```
select regexp_count('ABC123', '[A-Z][0-9]{2}') Char_num_like_A12_anywhere,
regexp_count('A1B2C34', '[A-Z][0-9]{2}') Char_num_like_A12_anywhere from dual;

CHAR_NUM_LIKE_A12_ANYWHERE CHAR_NUM_LIKE_A12_ANYWHERE
-------------------------- --------------------------
                         1                          1
```

In the following example, REGEXP_COUNT validates the supplied string for the given pattern and returns the number of alphabetic letters followed by a single digit number within the first two occurrences from the beginning of the string:

```
select regexp_count('ABC12D3', '([A-Z][0-9]){2}') Char_num_within_2_places,
regexp_count('A1B2C3', '([A-Z][0-9]){2}') Char_num_within_2_places from dual;

CHAR_NUM_WITHIN_2_PLACES CHAR_NUM_WITHIN_2_PLACES
------------------------ ------------------------
                       0                        1
```

> **Live SQL:**
>
> View and run related examples on Oracle Live SQL at *REGEXP_COUNT advanced matching*

REGEXP_COUNT case-sensitive matching: Examples

The following statements create a table regexp_temp and insert values into it:

```
CREATE TABLE regexp_temp(empName varchar2(20));

INSERT INTO regexp_temp (empName) VALUES ('John Doe');
INSERT INTO regexp_temp (empName) VALUES ('Jane Doe');
```

In the following example, the statement queries the employee name column and searches for the lowercase of character ‘E’:

```
SELECT empName, REGEXP_COUNT(empName, 'e', 1, 'c') "CASE_SENSITIVE_E" From regexp_temp;

EMPNAME          CASE_SENSITIVE_E
-------------------- ----------------
```

```
John Doe                1
Jane Doe                2
```

In the following example, the statement queries the employee name column and searches for the lowercase of character ‘O’:

```
SELECT empName, REGEXP_COUNT(empName, 'o', 1, 'c') "CASE_SENSITIVE_O" From regexp_temp;

EMPNAME          CASE_SENSITIVE_O
-------------------- ----------------
John Doe                2
Jane Doe                1
```

In the following example, the statement queries the employee name column and searches for the lowercase or uppercase of character ‘E’:

```
SELECT empName, REGEXP_COUNT(empName, 'E', 1, 'i') "CASE_INSENSITIVE_E" From regexp_temp;

EMPNAME          CASE_INSENSITIVE_E
-------------------- ------------------
John Doe                  1
Jane Doe                  2
```

In the following example, the statement queries the employee name column and searches for the lowercase of string ‘DO’:

```
SELECT empName, REGEXP_COUNT(empName, 'do', 1, 'i') "CASE_INSENSITIVE_STRING" From
regexp_temp;

EMPNAME          CASE_INSENSITIVE_STRING
-------------------- ----------------------
John Doe                  1
Jane Doe                  1
```

In the following example, the statement queries the employee name column and searches for the lowercase or uppercase of string ‘AN’:

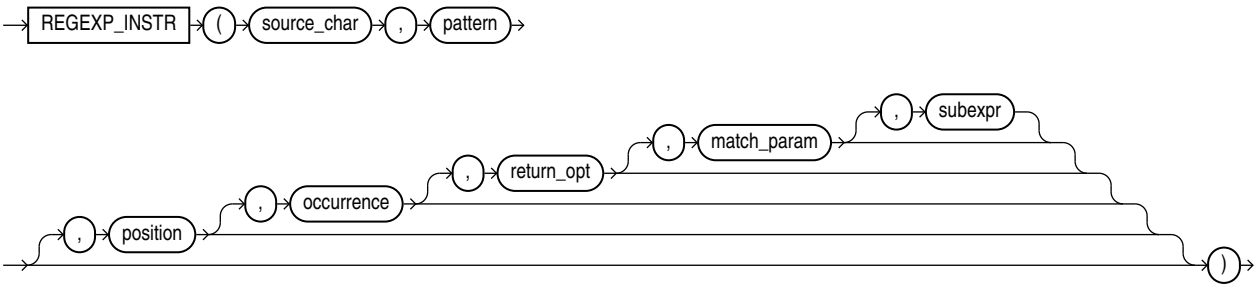
```
SELECT empName, REGEXP_COUNT(empName, 'an', 1, 'c') "CASE_SENSITIVE_STRING" From
regexp_temp;

EMPNAME          CASE_SENSITIVE_STRING
-------------------- --------------------
John Doe                  0
Jane Doe                  1
```

Live SQL:

View and run related examples on Oracle Live SQL at *REGEXP_COUNT case-sensitive matching*

REGEXP_INSTR

Syntax



Purpose

`REGEXP_INSTR` extends the functionality of the `INSTR` function by letting you search a string for a regular expression pattern. The function evaluates strings using characters as defined by the input character set. It returns an integer indicating the beginning or ending position of the matched substring, depending on the value of the *`return_option`* argument. If no match is found, then the function returns 0.

This function complies with the POSIX regular expression standard and the Unicode Regular Expression Guidelines. For more information, refer to Oracle Regular Expression Support.

- *`source_char`* is a character expression that serves as the search value. It is commonly a character column and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`.
- *`pattern`* is the regular expression. It is usually a text literal and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. It can contain up to 512 bytes. If the data type of *`pattern`* is different from the data type of *`source_char`*, then Oracle Database converts *`pattern`* to the data type of *`source_char`*. For a listing of the operators you can specify in *`pattern`*, refer to Oracle Regular Expression Support.
- *`position`* is a positive integer indicating the character of *`source_char`* where Oracle should begin the search. The default is 1, meaning that Oracle begins the search at the first character of *`source_char`*.
- *`occurrence`* is a positive integer indicating which occurrence of *`pattern`* in *`source_char`* Oracle should search for. The default is 1, meaning that Oracle searches for the first occurrence of *`pattern`*. If *`occurrence`* is greater than 1, then the database searches for the second occurrence beginning with the first character following the first occurrence of *`pattern`*, and so forth. This behavior is different from the `INSTR` function, which begins its search for the second occurrence at the second character of the first occurrence.
- *`return_option`* lets you specify what Oracle should return in relation to the occurrence:
  - If you specify 0, then Oracle returns the position of the first character of the occurrence. This is the default.
  - If you specify 1, then Oracle returns the position of the character following the occurrence.
- *`match_param`* is a character expression of the data type `VARCHAR2` or `CHAR` that lets you change the default matching behavior of the function. The behavior of this parameter is the

same for this function as for REGEXP_COUNT. Refer to REGEXP_COUNT for detailed information.

- For a *pattern* with subexpressions, *subexpr* is an integer from 0 to 9 indicating which subexpression in *pattern* is the target of the function. The *subexpr* is a fragment of pattern enclosed in parentheses. Subexpressions can be nested. Subexpressions are numbered in order in which their left parentheses appear in pattern. For example, consider the following expression:

  ```
  0123(((abc)(de)f)ghi)45(678)
  ```

  This expression has five subexpressions in the following order: "abcdefghi" followed by "abcdef", "abc", "de" and "678".

  If *subexpr* is zero, then the position of the entire substring that matches the *pattern* is returned. If *subexpr* is greater than zero, then the position of the substring fragment that corresponds to subexpression number *subexpr* in the matched substring is returned. If *pattern* does not have at least *subexpr* subexpressions, the function returns zero. A null *subexpr* value returns NULL. The default value for *subexpr* is zero.

> **See Also:**
>
> - INSTR and REGEXP_SUBSTR
> - REGEXP_REPLACE and REGEXP_LIKE Condition
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation REGEXP_INSTR uses to compare characters from *source_char* with characters from *pattern*

Examples

The following example examines the string, looking for occurrences of one or more non-blank characters. Oracle begins searching at the first character in the string and returns the starting position (default) of the sixth occurrence of one or more non-blank characters.

```
SELECT
  REGEXP_INSTR('500 Oracle Parkway, Redwood Shores, CA',
               '[^ ]+', 1, 6) "REGEXP_INSTR"
  FROM DUAL;

REGEXP_INSTR
------------
          37
```

The following example examines the string, looking for occurrences of words beginning with `s`, `r`, or `p`, regardless of case, followed by any six alphabetic characters. Oracle begins searching at the third character in the string and returns the position in the string of the character following the second occurrence of a seven-letter word beginning with `s`, `r`, or `p`, regardless of case.

```
SELECT
  REGEXP_INSTR('500 Oracle Parkway, Redwood Shores, CA',
               '[s|r|p][[:alpha:]]{6}', 3, 2, 1, 'i') "REGEXP_INSTR"
  FROM DUAL;

REGEXP_INSTR
```

```
------------
          28
```

The following examples use the *subexpr* argument to search for a particular subexpression in *pattern*. The first statement returns the position in the source string of the first character in the first subexpression, which is '123':

```
SELECT REGEXP_INSTR('1234567890', '(123)(4(56)(78))', 1, 1, 0, 'i', 1)
"REGEXP_INSTR" FROM DUAL;

REGEXP_INSTR
--------------------
1
```

The next statement returns the position in the source string of the first character in the second subexpression, which is '45678':

```
SELECT REGEXP_INSTR('1234567890', '(123)(4(56)(78))', 1, 1, 0, 'i', 2)
"REGEXP_INSTR" FROM DUAL;

REGEXP_INSTR
--------------------
4
```

The next statement returns the position in the source string of the first character in the fourth subexpression, which is '78':

```
SELECT REGEXP_INSTR('1234567890', '(123)(4(56)(78))', 1, 1, 0, 'i', 4)
"REGEXP_INSTR" FROM DUAL;

REGEXP_INSTR
--------------------
7
```

REGEXP_INSTR pattern matching: Examples

The following statements create a table regexp_temp and insert values into it:

```
CREATE TABLE regexp_temp(empName varchar2(20), emailID varchar2(20));

INSERT INTO regexp_temp (empName, emailID) VALUES ('John Doe', 'johndoe@example.com');
INSERT INTO regexp_temp (empName, emailID) VALUES ('Jane Doe', 'janedoe');
```

In the following example, the statement queries the email column and searches for valid email addresses:

```
SELECT emailID, REGEXP_INSTR(emailID, '\w+@\w+(\.\w+)+') "IS_A_VALID_EMAIL" FROM
regexp_temp;

EMAILID           IS_A_VALID_EMAIL
-------------------- ----------------
johndoe@example.com             1
example.com                0
```

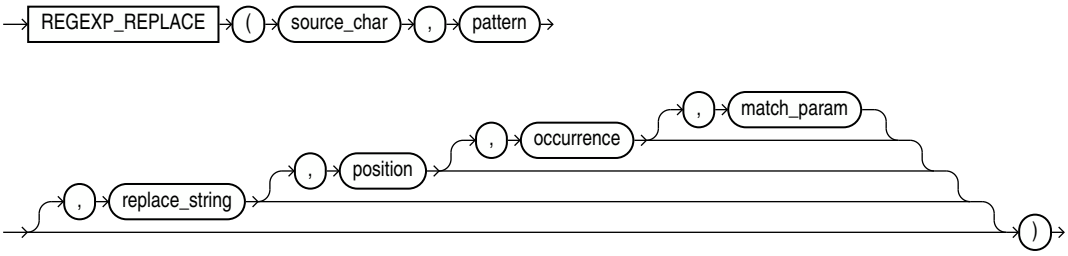
In the following example, the statement queries the email column and returns the count of valid email addresses:

```
EMPNAME        Valid Email            FIELD_WITH_VALID_EMAIL
--------    -------------------    ----------------------
John Doe    johndoe@example.com    1
Jane Doe
```

> **Live SQL:**
>
> View and run related examples on Oracle Live SQL at *REGEXP_INSTR pattern matching*

REGEXP_REPLACE

Syntax



Purpose

`REGEXP_REPLACE` extends the functionality of the `REPLACE` function by letting you search a string for a regular expression pattern. By default, the function returns *`source_char`* with every occurrence of the regular expression pattern replaced with *`replace_string`*. The string returned is in the same character set as *`source_char`*. The function returns `VARCHAR2` if the first argument is not a LOB and returns `CLOB` if the first argument is a LOB.

This function complies with the POSIX regular expression standard and the Unicode Regular Expression Guidelines. For more information, refer to Oracle Regular Expression Support.

- *`source_char`* is a character expression that serves as the search value. It is commonly a character column and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB` or `NCLOB`.
- *`pattern`* is the regular expression. It is usually a text literal and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. It can contain up to 512 bytes. If the data type of *`pattern`* is different from the data type of *`source_char`*, then Oracle Database converts *`pattern`* to the data type of *`source_char`*. For a listing of the operators you can specify in *`pattern`*, refer to Oracle Regular Expression Support.
- *`replace_string`* can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. If *`replace_string`* is a `CLOB` or `NCLOB`, then Oracle truncates *`replace_string`* to 32K. The *`replace_string`* can contain up to 500 backreferences to subexpressions in the form `\n`, where `n` is a number from 1 to 9. If you want to include a backslash (`\`) in *`replace_string`*, then you must precede it with the escape character, which is also a backslash. For example, to replace `\2` you would enter `\\2`. For more information on backreference expressions, refer to the notes to " Oracle Regular Expression Support", Table D-1.
- *`position`* is a positive integer indicating the character of *`source_char`* where Oracle should begin the search. The default is 1, meaning that Oracle begins the search at the first character of *`source_char`*.

- *occurrence* is a nonnegative integer indicating the occurrence of the replace operation:
  - If you specify 0, then Oracle replaces all occurrences of the match.
  - If you specify a positive integer *n*, then Oracle replaces the *n*th occurrence.

  If *occurrence* is greater than 1, then the database searches for the second occurrence beginning with the first character following the first occurrence of *pattern*, and so forth. This behavior is different from the INSTR function, which begins its search for the second occurrence at the second character of the first occurrence.
- *match_param* is a character expression of the data type VARCHAR2 or CHAR that lets you change the default matching behavior of the function. The behavior of this parameter is the same for this function as for REGEXP_COUNT. Refer to REGEXP_COUNT for detailed information.

> **See Also:**
>
> - REPLACE
> - REGEXP_INSTR , REGEXP_SUBSTR , and REGEXP_LIKE Condition
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation REGEXP_REPLACE uses to compare characters from *source_char* with characters from *pattern*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

The following example examines phone_number, looking for the pattern *xxx.xxx.xxxx*. Oracle reformats this pattern with (*xxx*) *xxx*-*xxxx*.

```
SELECT
  REGEXP_REPLACE(phone_number,
                 '([[:digit:]]{3})\.([[:digit:]]{3})\.([[:digit:]]{4})',
                 '(\1) \2-\3') "REGEXP_REPLACE"
  FROM employees
  ORDER BY "REGEXP_REPLACE";

REGEXP_REPLACE
--------------------------------------------------------------------------------
(515) 123-4444
(515) 123-4567
(515) 123-4568
(515) 123-4569
(515) 123-5555
. . .
```

The following example examines country_name. Oracle puts a space after each non-null character in the string.

```
SELECT
  REGEXP_REPLACE(country_name, '(.)', '\1 ') "REGEXP_REPLACE"
  FROM countries;

REGEXP_REPLACE
--------------------------------------------------------------------------------
A r g e n t i n a
```

```
A u s t r a l i a
B e l g i u m
B r a z i l
C a n a d a
. . .
```

The following example examines the string, looking for two or more spaces. Oracle replaces each occurrence of two or more spaces with a single space.

```
SELECT
  REGEXP_REPLACE('500   Oracle     Parkway,    Redwood  Shores, CA',
                 '( ){2,}', ' ') "REGEXP_REPLACE"
  FROM DUAL;

REGEXP_REPLACE
-------------------------------------
500 Oracle Parkway, Redwood Shores, CA
```

REGEXP_REPLACE pattern matching: Examples

The following statements create a table regexp_temp and insert values into it:

```
CREATE TABLE regexp_temp(empName varchar2(20), emailID varchar2(20));

INSERT INTO regexp_temp (empName, emailID) VALUES ('John Doe', 'johndoe@example.com');
INSERT INTO regexp_temp (empName, emailID) VALUES ('Jane Doe', 'janedoe@example.com');
```

The following statement replaces the string ‘Jane’ with ‘John’:

```
SELECT empName, REGEXP_REPLACE (empName, 'Jane', 'John') "STRING_REPLACE" FROM
regexp_temp;

EMPNAME        STRING_REPLACE
--------    --------------
John Doe    John Doe
Jane Doe    John Doe
```

The following statement replaces the string ‘John’ with ‘Jane’:

```
SELECT empName, REGEXP_REPLACE (empName, 'Jane', 'John') "STRING_REPLACE" FROM
regexp_temp;

EMPNAME        STRING_REPLACE
--------    --------------
John Doe    Jane Doe
Jane Doe    Jane Doe
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *REGEXP_REPLACE - Pattern Matching*

REGEXP_REPLACE: Examples

The following statement replaces all the numbers in a string:

```
WITH strings AS (
  SELECT 'abc123' s FROM dual union all
  SELECT '123abc' s FROM dual union all
```

```
  SELECT 'a1b2c3' s FROM dual
)
  SELECT s "STRING", regexp_replace(s, '[0-9]', '') "MODIFIED_STRING"
  FROM strings;

  STRING               MODIFIED_STRING
-------------------- --------------------
abc123               abc
123abc               abc
a1b2c3               abc
```

The following statement replaces the first numeric occurrence in a string:

```
WITH strings AS (
  SELECT 'abc123' s from DUAL union all
  SELECT '123abc' s from DUAL union all
  SELECT 'a1b2c3' s from DUAL
)
  SELECT s "STRING", REGEXP_REPLACE(s, '[0-9]', '', 1, 1) "MODIFIED_STRING"
  FROM   strings;

 STRING               MODIFIED_STRING
-------------------- --------------------
abc123               abc23
123abc               23abc
a1b2c3               ab2c3
```

The following statement replaces the second numeric occurrence in a string:

```
WITH strings AS (
  SELECT 'abc123' s from DUAL union all
  SELECT '123abc' s from DUAL union all
  SELECT 'a1b2c3' s from DUAL
)
  SELECT s "STRING", REGEXP_REPLACE(s, '[0-9]', '', 1, 2) "MODIFIED_STRING"
  FROM   strings;

STRING               MODIFIED_STRING
-------------------- --------------------
abc123               abc13
123abc               13abc
a1b2c3               a1bc3
```

The following statement replaces multiple spaces in a string with a single space:

```
WITH strings AS (
  SELECT 'Hello  World' s FROM dual union all
  SELECT 'Hello        World' s FROM dual union all
  SELECT 'Hello,   World  !' s FROM dual
)
  SELECT s "STRING", regexp_replace(s, ' {2,}', ' ') "MODIFIED_STRING"
  FROM   strings;

 STRING               MODIFIED_STRING
```

```
------------------- -------------------
Hello  World        Hello World
Hello       World   Hello World
Hello,   World  !   Hello, World !
```

The following statement converts camel case strings to a string containing lower case words separated by an underscore:

```
WITH strings as (
  SELECT 'AddressLine1' s FROM dual union all
  SELECT 'ZipCode' s FROM dual union all
  SELECT 'Country' s FROM dual
)
  SELECT s "STRING",
         lower(regexp_replace(s, '([A-Z0-9])', '_\1', 2)) "MODIFIED_STRING"
  FROM strings;

  STRING              MODIFIED_STRING
------------------- -------------------
AddressLine1        address_line_1
ZipCode             zip_code
Country             country
```

The following statement converts the format of a date:

```
WITH date_strings AS (
  SELECT  '2015-01-01' d from dual union all
  SELECT '2000-12-31' d from dual union all
  SELECT '900-01-01' d from dual
)
  SELECT d "STRING",
         regexp_replace(d, '([[:digit:]]+)-([[:digit:]]{2})-([[:digit:]]
{2})', '\3.\2.\1') "MODIFIED_STRING"
  FROM date_strings;

  STRING              MODIFIED_STRING
------------------- -------------------
2015-01-01          01.01.2015
2000-12-31          31.12.2000
900-01-01           01.01.900
```

The following statement replaces all the letters in a string with ‘1’:

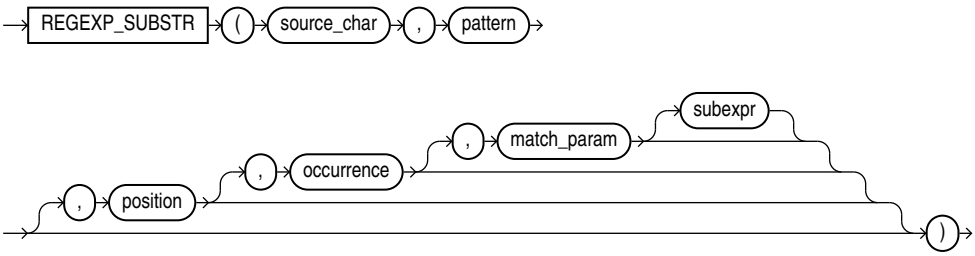
```
WITH strings as (
  SELECT 'NEW YORK' s FROM dual union all
  SELECT 'New York' s FROM dual union all
  SELECT 'new york' s FROM dual
)
  SELECT s "STRING",
        regexp_replace(s, '[a-z]', '1', 1, 0, 'i') "CASE_INSENSITIVE",
        regexp_replace(s, '[a-z]', '1', 1, 0, 'c') "CASE_SENSITIVE",
        regexp_replace(s, '[a-zA-Z]', '1', 1, 0, 'c')
"CASE_SENSITIVE_MATCHING"
  FROM  strings;
```

```
  STRING     CASE_INSEN CASE_SENSI CASE_SENSI
---------- ---------- ---------- ----------
NEW YORK   111 1111   NEW YORK   111 1111
New York   111 1111   N11 Y111   111 1111
new york   111 1111   111 1111   111 1111
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *REGEXP_REPLACE*

REGEXP_SUBSTR

Syntax



Purpose

`REGEXP_SUBSTR` extends the functionality of the `SUBSTR` function by letting you search a string for a regular expression pattern. It is also similar to `REGEXP_INSTR`, but instead of returning the position of the substring, it returns the substring itself. This function is useful if you need the contents of a match string but not its position in the source string. The function returns the string as `VARCHAR2` or `CLOB` data in the same character set as *`source_char`*.

This function complies with the POSIX regular expression standard and the Unicode Regular Expression Guidelines. For more information, refer to Oracle Regular Expression Support.

- *`source_char`* is a character expression that serves as the search value. It is commonly a character column and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`.
- *`pattern`* is the regular expression. It is usually a text literal and can be of any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. It can contain up to 512 bytes. If the data type of *`pattern`* is different from the data type of *`source_char`*, then Oracle Database converts *`pattern`* to the data type of *`source_char`*. For a listing of the operators you can specify in *`pattern`*, refer to Oracle Regular Expression Support.
- *`position`* is a positive integer indicating the character of *`source_char`* where Oracle should begin the search. The default is 1, meaning that Oracle begins the search at the first character of *`source_char`*.
- *`occurrence`* is a positive integer indicating which occurrence of *`pattern`* in *`source_char`* Oracle should search for. The default is 1, meaning that Oracle searches for the first occurrence of *`pattern`*.

If *occurrence* is greater than 1, then the database searches for the second occurrence beginning with the first character following the first occurrence of *pattern*, and so forth. This behavior is different from the SUBSTR function, which begins its search for the second occurrence at the second character of the first occurrence.

- *match_param* is a character expression of the data type VARCHAR2 or CHAR that lets you change the default matching behavior of the function. The behavior of this parameter is the same for this function as for REGEXP_COUNT. Refer to REGEXP_COUNT for detailed information.
- For a *pattern* with subexpressions, *subexpr* is a nonnegative integer from 0 to 9 indicating which subexpression in *pattern* is to be returned by the function. This parameter has the same semantics that it has for the REGEXP_INSTR function. Refer to REGEXP_INSTR for more information.

See Also:

- SUBSTR and REGEXP_INSTR
- REGEXP_REPLACE , and REGEXP_LIKE Condition
- Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation REGEXP_SUBSTR uses to compare characters from *source_char* with characters from *pattern*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

The following example examines the string, looking for the first substring bounded by commas. Oracle Database searches for a comma followed by one or more occurrences of non-comma characters followed by a comma. Oracle returns the substring, including the leading and trailing commas.

```
SELECT
  REGEXP_SUBSTR('500 Oracle Parkway, Redwood Shores, CA',
                ',[^,]+,') "REGEXPR_SUBSTR"
  FROM DUAL;

REGEXPR_SUBSTR
-----------------
, Redwood Shores,
```

The following example examines the string, looking for `http://` followed by a substring of one or more alphanumeric characters and optionally, a period (`.`). Oracle searches for a minimum of three and a maximum of four occurrences of this substring between `http://` and either a slash (`/`) or the end of the string.

```
SELECT
  REGEXP_SUBSTR('http://www.example.com/products',
                'http://([[:alnum:]]+\.?){3,4}/?') "REGEXP_SUBSTR"
  FROM DUAL;

REGEXP_SUBSTR
----------------------
http://www.example.com/
```

The next two examples use the *subexpr* argument to return a specific subexpression of *pattern*. The first statement returns the first subexpression in *pattern*:

```
SELECT REGEXP_SUBSTR('1234567890', '(123)(4(56)(78))', 1, 1, 'i', 1)
"REGEXP_SUBSTR" FROM DUAL;

REGEXP_SUBSTR
-------------------
123
```

The next statement returns the fourth subexpression in *pattern*:

```
SELECT REGEXP_SUBSTR('1234567890', '(123)(4(56)(78))', 1, 1, 'i', 4)
"REGEXP_SUBSTR" FROM DUAL;

REGEXP_SUBSTR
-------------------
78
```

REGEXP_SUBSTR pattern matching: Examples

The following statements create a table regexp_temp and insert values into it:

```
CREATE TABLE regexp_temp(empName varchar2(20), emailID varchar2(20));

INSERT INTO regexp_temp (empName, emailID) VALUES ('John Doe', 'johndoe@example.com');
INSERT INTO regexp_temp (empName, emailID) VALUES ('Jane Doe', 'janedoe');
```

In the following example, the statement queries the email column and searches for valid email addresses:

```
SELECT empName, REGEXP_SUBSTR(emailID, '[[:alnum:]]+\@[[:alnum:]]+\.[[:alnum:]]+')
"Valid Email" FROM regexp_temp;

EMPNAME  Valid Email
-------- -------------------
John Doe johndoe@example.com
Jane Doe
```

In the following example, the statement queries the email column and returns the count of valid email addresses:

```
SELECT empName, REGEXP_SUBSTR(emailID, '[[:alnum:]]+\@[[:alnum:]]+\.[[:alnum:]]+')
"Valid Email", REGEXP_INSTR(emailID, '\w+@\w+(\.\w+)+') "FIELD_WITH_VALID_EMAIL" FROM
regexp_temp;

EMPNAME        Valid Email            FIELD_WITH_VALID_EMAIL
--------    -------------------    ----------------------
John Doe    johndoe@example.com    1
Jane Doe
```

> **Live SQL:**
>
> View and run related examples on Oracle Live SQL at *REGEXP_SUBSTR pattern matching*

In the following example, numbers and alphabets are extracted from a string:

```
with strings as (
  select 'ABC123' str from dual union all
  select 'A1B2C3' str from dual union all
  select '123ABC' str from dual union all
  select '1A2B3C' str from dual
)
  select regexp_substr(str, '[0-9]') First_Occurrence_of_Number,
         regexp_substr(str, '[0-9].*') Num_Followed_by_String,
         regexp_substr(str, '[A-Z][0-9]') Letter_Followed_by_String
  from strings;

FIRST_OCCURRENCE_OF_NUMB NUM_FOLLOWED_BY_STRING   LETTER_FOLLOWED_BY_STRIN
------------------------ ------------------------ ------------------------
1             123              C1
1             1B2C3              A1
1             123ABC
1             1A2B3C           A2
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *REGEXP_SUBSTR - Extract Numbers and Alphabets*

In the following example, passenger names and flight information are extracted from a string:

```
with strings as (
  select 'LHRJFK/010315/JOHNDOE' str from dual union all
  select 'CDGLAX/050515/JANEDOE' str from dual union all
  select 'LAXCDG/220515/JOHNDOE' str from dual union all
  select 'SFOJFK/010615/JANEDOE' str from dual
)
  SELECT regexp_substr(str, '[A-Z]{6}') String_of_6_characters,
         regexp_substr(str, '[0-9]+') First_Matching_Numbers,
         regexp_substr(str, '[A-Z].*$') Letter_by_other_characters,
         regexp_substr(str, '/[A-Z].*$') Slash_letter_and_characters
  FROM strings;

STRING_OF_6_CHARACTERS    FIRST_MATCHING_NUMBERS    LETTER_BY_OTHER_CHARACTERS
SLASH_LETTER_AND_CHARACTERS
----------------------    ----------------------    --------------------------
---------------------------
LHRJFK                    010315                    LHRJFK/010315/JOHNDOE              /
JOHNDOE
CDGLAX                    050515                    CDGLAX/050515/JANEDOE              /
JANEDOE
LAXCDG                    220515                    LAXCDG/220515/JOHNDOE              /
JOHNDOE
SFOJFK                    010615                    SFOJFK/010615/JANEDOE              /
JANEDOE
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *REGEXP_SUBSTR - Extract Passenger Names and Flight Information*

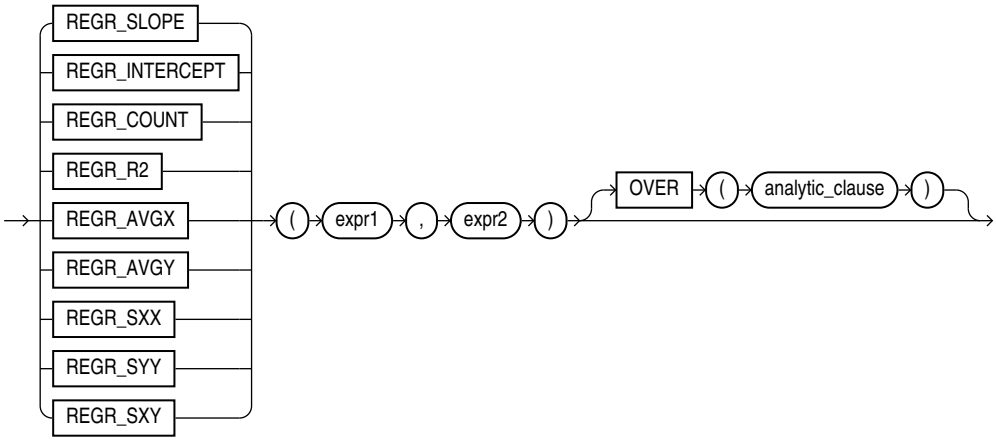
REGR_ (Linear Regression) Functions

The linear regression functions are:

- REGR_SLOPE
- REGR_INTERCEPT
- REGR_COUNT
- REGR_R2
- REGR_AVGX
- REGR_AVGY
- REGR_SXX
- REGR_SYY
- REGR_SXY

Syntax

***linear_regr*::=**



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

The linear regression functions fit an ordinary-least-squares regression line to a set of number pairs. You can use them as both aggregate and analytic functions.

> **See Also:**
>
> "Aggregate Functions " and "About SQL Expressions " for information on valid forms of *expr*

These functions take as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

Oracle applies the function to the set of (*expr1*, *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Oracle computes all the regression functions simultaneously during a single pass through the data.

expr1 is interpreted as a value of the dependent variable (a `y` value), and *expr2* is interpreted as a value of the independent variable (an `x` value).

- `REGR_SLOPE` returns the slope of the line. The return value is a numeric data type and can be null. After the elimination of null (*expr1*, *expr2*) pairs, it makes the following computation:

  ```
  COVAR_POP(expr1, expr2) / VAR_POP(expr2)
  ```

- `REGR_INTERCEPT` returns the y-intercept of the regression line. The return value is a numeric data type and can be null. After the elimination of null (*expr1*, *expr2*) pairs, it makes the following computation:

  ```
  AVG(expr1) - REGR_SLOPE(expr1, expr2) * AVG(expr2)
  ```

- `REGR_COUNT` returns an integer that is the number of non-null number pairs used to fit the regression line.
- `REGR_R2` returns the coefficient of determination (also called R-squared or goodness of fit) for the regression. The return value is a numeric data type and can be null. `VAR_POP`(*expr1*) and `VAR_POP`(*expr2*) are evaluated after the elimination of null pairs. The return values are:

  ```
                      NULL if VAR_POP(expr2)  = 0

                         1 if VAR_POP(expr1)  = 0 and
                              VAR_POP(expr2) != 0

  POWER(CORR(expr1,expr),2) if VAR_POP(expr1)  > 0 and
                              VAR_POP(expr2  != 0
  ```

All of the remaining regression functions return a numeric data type and can be null:

- `REGR_AVGX` evaluates the average of the independent variable (*expr2*) of the regression line. It makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  AVG(expr2)
  ```

- REGR_AVGY evaluates the average of the dependent variable (*expr1*) of the regression line. It makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  AVG(expr1)
  ```

REGR_SXY, REGR_SXX, REGR_SYY are auxiliary functions that are used to compute various diagnostic statistics.

- REGR_SXX makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  REGR_COUNT(expr1, expr2) * VAR_POP(expr2)
  ```

- REGR_SYY makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  REGR_COUNT(expr1, expr2) * VAR_POP(expr1)
  ```

- REGR_SXY makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  REGR_COUNT(expr1, expr2) * COVAR_POP(expr1, expr2)
  ```

The following examples are based on the sample tables sh.sales and sh.products.

General Linear Regression Example

The following example provides a comparison of the various linear regression functions used in their analytic form. The analytic form of these functions can be useful when you want to use regression statistics for calculations such as finding the salary predicted for each employee by the model. The sections that follow on the individual linear regression functions contain examples of the aggregate form of these functions.

```
SELECT job_id, employee_id ID, salary,
REGR_SLOPE(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) slope,
REGR_INTERCEPT(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) intcpt,
REGR_R2(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) rsqr,
REGR_COUNT(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) count,
REGR_AVGX(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) avgx,
REGR_AVGY(SYSDATE-hire_date, salary)
   OVER (PARTITION BY job_id) avgy
   FROM employees
   WHERE department_id in (50, 80)
   ORDER BY job_id, employee_id;

JOB_ID        ID     SALARY SLOPE    INTCPT  RSQR  COUNT       AVGX      AVGY
---------- ----- ---------- ----- --------- ----- ------ ---------- ---------
SA_MAN       145      14000  .355 -1707.035  .832      5  12200.000  2626.589
SA_MAN       146      13500  .355 -1707.035  .832      5  12200.000  2626.589
SA_MAN       147      12000  .355 -1707.035  .832      5  12200.000  2626.589
SA_MAN       148      11000  .355 -1707.035  .832      5  12200.000  2626.589
SA_MAN       149      10500  .355 -1707.035  .832      5  12200.000  2626.589
SA_REP       150      10000  .257   404.763  .647     29   8396.552  2561.244
SA_REP       151       9500  .257   404.763  .647     29   8396.552  2561.244
SA_REP       152       9000  .257   404.763  .647     29   8396.552  2561.244
SA_REP       153       8000  .257   404.763  .647     29   8396.552  2561.244
SA_REP       154       7500  .257   404.763  .647     29   8396.552  2561.244
SA_REP       155       7000  .257   404.763  .647     29   8396.552  2561.244
SA_REP       156      10000  .257   404.763  .647     29   8396.552  2561.244
. . .
```

REGR_SLOPE and REGR_INTERCEPT Examples

The following example calculates the slope and regression of the linear regression model for time employed (`SYSDATE - hire_date`) and salary using the sample table `hr.employees`. Results are grouped by `job_id`.

```
SELECT job_id,
REGR_SLOPE(SYSDATE-hire_date, salary) slope,
REGR_INTERCEPT(SYSDATE-hire_date, salary) intercept
   FROM employees
   WHERE department_id in (50,80)
   GROUP BY job_id
   ORDER BY job_id;

JOB_ID     SLOPE    INTERCEPT
---------- ----- ------------
SA_MAN      .355 -1707.030762
SA_REP      .257   404.767151
SH_CLERK    .745   159.015293
ST_CLERK    .904   134.409050
ST_MAN      .479  -570.077291
```

REGR_COUNT Examples

The following example calculates the count of by `job_id` for time employed (`SYSDATE - hire_date`) and salary using the sample table `hr.employees`. Results are grouped by `job_id`.

```
SELECT job_id,
REGR_COUNT(SYSDATE-hire_date, salary) count
   FROM employees
   WHERE department_id in (30, 50)
   GROUP BY job_id
   ORDER BY job_id, count;

JOB_ID          COUNT
---------- ----------
PU_CLERK             5
PU_MAN               1
SH_CLERK            20
ST_CLERK            20
ST_MAN               5
```

REGR_R2 Examples

The following example calculates the coefficient of determination the linear regression of time employed (`SYSDATE - hire_date`) and salary using the sample table `hr.employees`:

```
SELECT job_id,
REGR_R2(SYSDATE-hire_date, salary) Regr_R2
   FROM employees
   WHERE department_id in (80, 50)
   GROUP by job_id
  ORDER BY job_id, Regr_R2;

JOB_ID        REGR_R2
---------- ----------
SA_MAN      .83244748
SA_REP     .647007156
SH_CLERK   .879799698
ST_CLERK   .742808493
ST_MAN      .69418508
```

REGR_AVGY and REGR_AVGX Examples

The following example calculates the average values for time employed (`SYSDATE - hire_date`) and salary using the sample table `hr.employees`. Results are grouped by `job_id`:

```
SELECT job_id,
REGR_AVGY(SYSDATE-hire_date, salary) avgy,
REGR_AVGX(SYSDATE-hire_date, salary) avgx
   FROM employees
   WHERE department_id in (30,50)
   GROUP BY job_id
   ORDER BY job_id, avgy, avgx;

JOB_ID           AVGY       AVGX
---------- ---------- ----------
PU_CLERK    2950.3778       2780
PU_MAN      4026.5778      11000
SH_CLERK    2773.0778       3215
ST_CLERK    2872.7278       2785
ST_MAN      3140.1778       7280
```

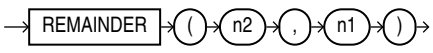
REGR_SXY, REGR_SXX, and REGR_SYY Examples

The following example calculates three types of diagnostic statistics for the linear regression of time employed (`SYSDATE - hire_date`) and salary using the sample table `hr.employees`:

```
SELECT job_id,
REGR_SXY(SYSDATE-hire_date, salary) regr_sxy,
REGR_SXX(SYSDATE-hire_date, salary) regr_sxx,
REGR_SYY(SYSDATE-hire_date, salary) regr_syy
   FROM employees
   WHERE department_id in (80, 50)
   GROUP BY job_id
   ORDER BY job_id;

JOB_ID       REGR_SXY    REGR_SXX   REGR_SYY
---------- ---------- ----------- ----------
SA_MAN        3303500   9300000.0    1409642
SA_REP     16819665.5  65489655.2 6676562.55
SH_CLERK      4248650   5705500.0    3596039
ST_CLERK      3531545   3905500.0 4299084.55
ST_MAN        2180460   4548000.0  1505915.2
```

REMAINDER

Syntax



Purpose

`REMAINDER` returns the remainder of *n2* divided by *n1*.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

The MOD function is similar to REMAINDER except that it uses FLOOR in its formula, whereas REMAINDER uses ROUND. Refer to MOD .

> **See Also:**
>
> Table 2-9 for more information on implicit conversion and "Numeric Precedence " for information on numeric precedence

- If *n1* = 0 or *n2* = infinity, then Oracle returns
  - An error if the arguments are of type NUMBER
  - NaN if the arguments are BINARY_FLOAT or BINARY_DOUBLE.
- If *n1* != 0, then the remainder is *n2* - (*n1***N*) where *N* is the integer nearest *n2*/*n1*. If *n2*/*n1* equals x.5, then *N* is the nearest *even* integer.
- If *n2* is a floating-point number, and if the remainder is 0, then the sign of the remainder is the sign of *n2*. Remainders of 0 are unsigned for NUMBER values.

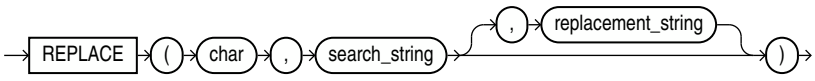
Examples

Using table float_point_demo, created for the TO_BINARY_DOUBLE "Examples", the following example divides two floating-point numbers and returns the remainder of that operation:

```
SELECT bin_float, bin_double, REMAINDER(bin_float, bin_double)
  FROM float_point_demo;

 BIN_FLOAT BIN_DOUBLE REMAINDER(BIN_FLOAT,BIN_DOUBLE)
---------- ---------- ------------------------------
1.235E+003 1.235E+003                     5.859E-005
```

REPLACE

Syntax



Purpose

REPLACE returns *char* with every occurrence of *search_string* replaced with *replacement_string*. If *replacement_string* is omitted or null, then all occurrences of *search_string* are removed. If *search_string* is null, then *char* is returned.

Both *search_string* and *replacement_string*, as well as *char*, can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is in the same character set as *char*. The function returns VARCHAR2 if the first argument is not a LOB and returns CLOB if the first argument is a LOB.

REPLACE provides functionality related to that provided by the TRANSLATE function. TRANSLATE provides single-character, one-to-one substitution. REPLACE lets you substitute one string for another as well as to remove character strings.

> **See Also:**
>
> - TRANSLATE
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `REPLACE` uses to compare characters from *`char`* with characters from *`search_string`*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

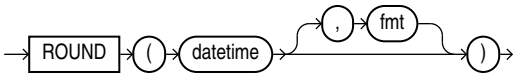
The following example replaces occurrences of `J` with `BL`:

```
SELECT REPLACE('JACK and JUE','J','BL') "Changes"
     FROM DUAL;

Changes
--------------
BLACK and BLUE
```

ROUND (datetime)

Syntax

***round_datetime*::=**



Purpose

`ROUND` returns *`datetime`* rounded to the unit specified by the format model *`fmt`*.

This function is not sensitive to the `NLS_CALENDAR` session parameter. It operates according to the rules of the Gregorian calendar. The value returned is always of data type `DATE`, even if you specify a different datetime data type for *`date`*. If you omit *`fmt`*, then *`date`* is rounded to the nearest day. The *`date`* expression must resolve to a `DATE` value.

> **See Also:**
>
> "CEIL, FLOOR, ROUND, and TRUNC Date Functions" for the permitted format models to use in *`fmt`*

Examples

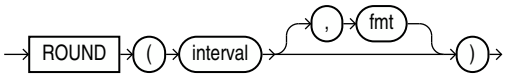
The following example rounds a date to the first day of the following year:

```
SELECT ROUND (TO_DATE ('27-OCT-00'),'YEAR')
   "New Year" FROM DUAL;
```

```
New Year
---------
01-JAN-01
```

ROUND (interval)

Syntax



Purpose

`ROUND(interval)` returns the interval rounded up to the unit specified by the second argument *`fmt`*, the format model .

For `INTERVAL YEAR TO MONTH`, *`fmt`* can only be year. The default *`fmt`* is year.

For `INTERVAL DAY TO SECOND`, *`fmt`* can be day, hour and minute. The default *`fmt`* is day. Note that *`fmt`* does not support second.

`ROUND(interval)` rounds up on the mid value of next part of *`fmt`* as follows:

- If *`fmt`* is year, `ROUND(interval)` rounds up on the mid value of month which is 6.
- If *`fmt`* is day, `ROUND(interval)` rounds up on the mid value of hour which is 12.
- If *`fmt`* is hour, `ROUND(interval)` rounds up on the mid value of minute which is 30.
- If *`fmt`* is minute, `ROUND(interval)` rounds up on the mid value of second which is 30.

The result precision for year and day is the input precision for year plus one and day plus one respectively, since `ROUND(interval)` can have overflow. If an interval already has the maximum precision for year and day, the statement compiles but errors at runtime.

> **See Also:**
>
> Refer to CEIL, FLOOR, ROUND, and TRUNC Date Functions for the permitted format models to use in *`fmt`*.

Examples

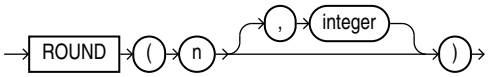
```
SELECT ROUND(INTERVAL '+123-06' YEAR(3) TO MONTH) AS year_round;

YEAR_ROUND
----------
+124-00


SELECT ROUND(INTERVAL '+99-11' YEAR(2) TO MONTH, 'YEAR') AS year_round;

YEAR_ROUND
----------
+100-00
```

```
SELECT ROUND(INTERVAL '-999999999-11' YEAR(9) TO MONTH, 'YEAR')AS year_round;

ORA-01873: the leading precision of the interval is too small

SELECT ROUND(INTERVAL '+4 12:42:10.222' DAY(2) TO SECOND(3), 'DD') AS day_round;

DAY_ROUND
-------------------
+05 00:00:00.000000
```

ROUND (number)

Syntax

***round_number*::=**



Purpose

`ROUND` returns *`n`* rounded to *`integer`* places to the right of the decimal point. If you omit *`integer`*, then *`n`* is rounded to zero places. If *`integer`* is negative, then *`n`* is rounded off to the left of the decimal point.

`n` can be any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If you omit *`integer`*, then the function returns the value `ROUND`(n, 0) in the same data type as the numeric data type of *`n`*. If you include *`integer`*, then the function returns `NUMBER`.

`ROUND` is implemented using the following rules:

1. If *`n`* is 0, then `ROUND` always returns 0 regardless of *`integer`*.
2. If *`n`* is negative, then `ROUND`(n, integer) returns -`ROUND`(-n, integer).
3. If *`n`* is positive, then

```
ROUND(n, integer) = FLOOR(n * POWER(10, integer) + 0.5) * POWER(10, -integer)
```

`ROUND` applied to a `NUMBER` value may give a slightly different result from `ROUND` applied to the same value expressed in floating-point. The different results arise from differences in internal representations of `NUMBER` and floating point values. The difference will be 1 in the rounded digit if a difference occurs.

> **See Also:**
>
> - Table 2-9 for more information on implicit conversion
> - "Floating-Point Numbers " for more information on how Oracle Database handles `BINARY_FLOAT` and `BINARY_DOUBLE` values
> - FLOOR (number) and CEIL (number), TRUNC (number) and MOD for information on functions that perform related operations

Examples

The following example rounds a number to one decimal point:

```
SELECT ROUND(15.193,1) "Round" FROM DUAL;

     Round
----------
      15.2
```

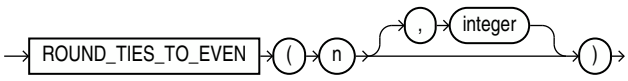
The following example rounds a number one digit to the left of the decimal point:

```
SELECT ROUND(15.193,-1) "Round" FROM DUAL;

     Round
----------
        20
```

ROUND_TIES_TO_EVEN (number)

Syntax

***round_ties_to_even* ::=**



Purpose

`ROUND_TIES_TO_EVEN` is a rounding function that takes two parameters: `n` and `integer`. The function returns `n` rounded to `integer` places according to the following rules:

1. If `integer` is positive, `n` is rounded to `integer` places to the right of the decimal point.
2. If `integer` is not specified, then `n` is rounded to 0 places.
3. If `integer` is negative, then `n` is rounded to `integer` places to the left of the decimal point.

Restrictions

The function does not support the following types: `BINARY_FLOAT` and `BINARY_DOUBLE`.

Examples

The following example rounds a number to one decimal point to the right:

```
SELECT ROUND_TIES_TO_EVEN (0.05, 1) from DUAL

ROUND_TIES_TO_EVEN(0.05,1)
--------------------------
                         0
```

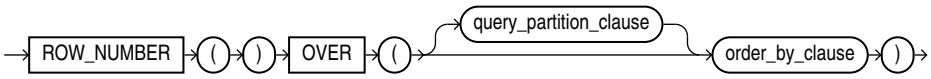
The following example rounds a number to one decimal point to the left:

```
SELECT ROUND_TIES_TO_EVEN(45.177,-1) "ROUND_EVEN" FROM DUAL;
```

```
ROUND_TIES_TO_EVEN(45.177,-1)
-----------------------------
                           50
```

ROW_NUMBER

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`ROW_NUMBER` is an analytic function. It assigns a unique number to each row to which it is applied (either each row in the partition or each row returned by the query), in the ordered sequence of rows specified in the *`order_by_clause`*, beginning with 1.

By nesting a subquery using `ROW_NUMBER` inside a query that retrieves the `ROW_NUMBER` values for a specified range, you can find a precise subset of rows from the results of the inner query. This use of the function lets you implement top-N, bottom-N, and inner-N reporting. For consistent results, the query must ensure a deterministic sort order.

Examples

The following example finds the three highest paid employees in each department in the `hr.employees` table. Fewer than three rows are returned for departments with fewer than three employees.

```
SELECT department_id, first_name, last_name, salary
FROM
(
  SELECT
    department_id, first_name, last_name, salary,
    ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary desc) rn
  FROM employees
)
WHERE rn <= 3
ORDER BY department_id, salary DESC, last_name;
```

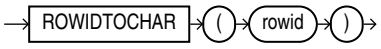
The following example is a join query on the `sh.sales` table. It finds the sales amounts in 2000 of the five top-selling products in 1999 and compares the difference between 2000 and 1999. The ten top-selling products are calculated within each distribution channel.

```
SELECT sales_2000.channel_desc, sales_2000.prod_name,
       sales_2000.amt amt_2000,  top_5_prods_1999_year.amt amt_1999,
       sales_2000.amt  - top_5_prods_1999_year.amt amt_diff
FROM
/* The first subquery finds the 5 top-selling products per channel in year 1999. */
  (SELECT channel_desc, prod_name, amt
```

```
   FROM
   (
     SELECT channel_desc, prod_name, sum(amount_sold) amt,
       ROW_NUMBER () OVER (PARTITION BY channel_desc
                           ORDER BY SUM(amount_sold) DESC) rn
     FROM sales, times, channels, products
     WHERE sales.time_id = times.time_id
       AND times.calendar_year = 1999
       AND channels.channel_id = sales.channel_id
       AND products.prod_id = sales.prod_id
     GROUP BY channel_desc, prod_name
   )
   WHERE rn <= 5
  ) top_5_prods_1999_year,
/* The next subquery finds sales per product and per channel in 2000. */
  (SELECT channel_desc, prod_name, sum(amount_sold) amt
     FROM sales, times, channels, products
     WHERE sales.time_id = times.time_id
       AND times.calendar_year = 2000
       AND channels.channel_id = sales.channel_id
       AND products.prod_id = sales.prod_id
     GROUP BY channel_desc, prod_name
  ) sales_2000
WHERE sales_2000.channel_desc = top_5_prods_1999_year.channel_desc
  AND sales_2000.prod_name = top_5_prods_1999_year.prod_name
ORDER BY sales_2000.channel_desc, sales_2000.prod_name
;
CHANNEL_DESC    PROD_NAME                                    AMT_2000   AMT_1999   AMT_DIFF
--------------- --------------==------------------------------ ---------- ---------- ----------
Direct Sales     17" LCD w/built-in HDTV Tuner                628855.7 1163645.78 -534790.08
Direct Sales     Envoy 256MB - 40GB                           502938.54  843377.88 -340439.34
Direct Sales     Envoy Ambassador                            2259566.96 1770349.25  489217.71
Direct Sales     Home Theatre Package with DVD-Audio/Video Play 1235674.15 1260791.44  -25117.29
Direct Sales     Mini DV Camcorder with 3.5" Swivel LCD       775851.87 1326302.51 -550450.64
Internet         17" LCD w/built-in HDTV Tuner                31707.48   160974.7 -129267.22
Internet         8.3 Minitower Speaker                        404090.32  155235.25  248855.07
Internet         Envoy 256MB - 40GB                           28293.87  154072.02 -125778.15
Internet         Home Theatre Package with DVD-Audio/Video Play 155405.54  153175.04     2230.5
Internet         Mini DV Camcorder with 3.5" Swivel LCD       39726.23  189921.97 -150195.74
Partners         17" LCD w/built-in HDTV Tuner                269973.97  325504.75  -55530.78
Partners         Envoy Ambassador                            1213063.59  614857.93  598205.66
Partners         Home Theatre Package with DVD-Audio/Video Play 700266.58  520166.26  180100.32
Partners         Mini DV Camcorder with 3.5" Swivel LCD       404265.85  520544.11 -116278.26
Partners         Unix/Windows 1-user pack                     374002.51  340123.02   33879.49

15 rows selected.
```

ROWIDTOCHAR

Syntax



Purpose

`ROWIDTOCHAR` converts a rowid value to `VARCHAR2` data type. The result of this conversion is always 18 characters long.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `ROWIDTOCHAR`

Examples

The following example converts a rowid value in the `employees` table to a character value. (Results vary for each build of the sample database.)

```
SELECT ROWID FROM employees
   WHERE ROWIDTOCHAR(ROWID) LIKE '%JAAB%'
   ORDER BY ROWID;

ROWID
------------------
AAAFfIAAFAAAABSAAb
```

ROWIDTONCHAR

Syntax

Purpose

`ROWIDTONCHAR` converts a rowid value to `NVARCHAR2` data type. The result of this conversion is always in the national character set and is 18 characters long.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `ROWIDTONCHAR`

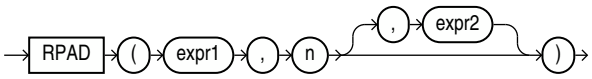
Examples

The following example converts a rowid value to an `NVARCHAR2` string:

```
SELECT LENGTHB( ROWIDTONCHAR(ROWID) ) Length, ROWIDTONCHAR(ROWID)
   FROM employees
   ORDER BY length;

    LENGTH ROWIDTONCHAR(ROWID
---------- ------------------
        36 AAAL52AAFAAAABSABD
        36 AAAL52AAFAAAABSABV
. . .
```

RPAD

Syntax



Purpose

`RPAD` returns *`expr1`*, right-padded to length *`n`* characters with *`expr2`*, replicated as many times as necessary. This function is useful for formatting the output of a query.

Both *`expr1`* and *`expr2`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` data type if *`expr1`* is a character data type, `NVARCHAR2` if *`expr1`* is a national character data type, and a LOB if *`expr1`* is a LOB data type. The string returned is in the same character set as *`expr1`*. The argument *`n`* must be a `NUMBER` integer or a value that can be implicitly converted to a `NUMBER` integer.

`expr1` cannot be null. If you do not specify *`expr2`*, then it defaults to a single blank. If *`expr1`* is longer than *`n`*, then this function returns the portion of *`expr1`* that fits in *`n`*.

The argument *`n`* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `RPAD`

Examples

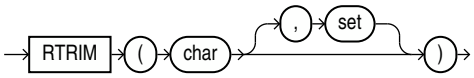
The following example creates a simple chart of salary amounts by padding a single space with asterisks:

```
SELECT last_name, RPAD(' ', salary/1000/1, '*') "Salary"
   FROM employees
   WHERE department_id = 80
   ORDER BY last_name, "Salary";

LAST_NAME                 Salary
------------------------- ---------------
Abel                       **********
Ande                       *****
Banda                      *****
Bates                      ******
Bernstein                  ********
Bloom                      *********
Cambrault                  **********
Cambrault                  ******
Doran                      ******
```

```
Errazuriz                 ***********
Fox                       ********
Greene                    ********
Hall                      ********
Hutton                    *******
Johnson                   *****
King                      *********
. . .
```

RTRIM

Syntax



Purpose

`RTRIM` removes from the right end of *`char`* all of the characters that appear in *`set`*. This function is useful for formatting the output of a query.

If you do not specify *`set`*, then it defaults to a single blank. `RTRIM` works similarly to `LTRIM`.

Both *`char`* and *`set`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` data type if *`char`* is a character data type, `NVARCHAR2` if *`char`* is a national character data type, and a LOB if *`char`* is a LOB data type.

> **See Also:**
>
> - LTRIM
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `RTRIM` uses to compare characters from *`set`* with characters from *`char`*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

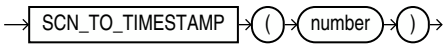
Examples

The following example trims all the right-most occurrences of less than sign (<), greater than sign (>) , and equal sign (=) from a string:

```
SELECT RTRIM('<=====>BROWNING<=====>', '<>=') "RTRIM Example"
  FROM DUAL;

RTRIM Example
---------------
<=====>BROWNING
```

SCN_TO_TIMESTAMP

Syntax



Purpose

`SCN_TO_TIMESTAMP` takes as an argument a number that evaluates to a system change number (SCN), and returns the approximate timestamp associated with that SCN. The returned value is of `TIMESTAMP` data type. This function is useful any time you want to know the timestamp associated with an SCN. For example, it can be used in conjunction with the `ORA_ROWSCN` pseudocolumn to associate a timestamp with the most recent change to a row.

Notes:

- The usual precision of the result value is 3 seconds.
- The association between an SCN and a timestamp when the SCN is generated is remembered by the database for a limited period of time. This period is the maximum of the auto-tuned undo retention period, if the database runs in the Automatic Undo Management mode, and the retention times of all flashback archives in the database, but no less than 120 hours. The time for the association to become obsolete elapses only when the database is open. An error is returned if the SCN specified for the argument to `SCN_TO_TIMESTAMP` is too old.

See Also:

ORA_ROWSCN Pseudocolumn and TIMESTAMP_TO_SCN

Examples

The following example uses the `ORA_ROWSCN` pseudocolumn to determine the system change number of the last update to a row and uses `SCN_TO_TIMESTAMP` to convert that SCN to a timestamp:

```
SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM employees
   WHERE employee_id = 188;
```

You could use such a query to convert a system change number to a timestamp for use in an Oracle Flashback Query:

```
SELECT salary FROM employees WHERE employee_id = 188;
    SALARY
----------
      3800

UPDATE employees SET salary = salary*10 WHERE employee_id = 188;
```

```
COMMIT;

SELECT salary FROM employees WHERE employee_id = 188;
    SALARY
----------
     38000

SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM employees
   WHERE employee_id = 188;
SCN_TO_TIMESTAMP(ORA_ROWSCN)
---------------------------------------------------------------------------
28-AUG-03 01.58.01.000000000 PM

FLASHBACK TABLE employees TO TIMESTAMP
   TO_TIMESTAMP('28-AUG-03 01.00.00.000000000 PM');

SELECT salary FROM employees WHERE employee_id = 188;
    SALARY
----------
      3800
```

SESSIONTIMEZONE

Syntax

Purpose

`SESSIONTIMEZONE` returns the time zone of the current session. The return type is a time zone offset (a character type in the format `'[+|-]TZH:TZM'`) or a time zone region name, depending on how the user specified the session time zone value in the most recent `ALTER SESSION` statement.

> **Note:**
>
> The default client session time zone is an offset even if the client operating system uses a named time zone. If you want the default session time zone to use a named time zone, then set the `ORA_SDTZ` variable in the client environment to an Oracle time zone region name. Refer to *Oracle Database Globalization Support Guide* for more information on this variable.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `SESSIONTIMEZONE`

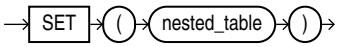
Examples

The following example returns the time zone of the current session:

```
SELECT SESSIONTIMEZONE FROM DUAL;

SESSION
-------
-08:00
```

SET

Syntax



Purpose

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

The element types of the nested table must be comparable. Refer to "Comparison Conditions " for information on the comparability of nonscalar types.

Examples

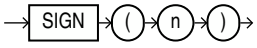
The following example selects from the `customers_demo` table the unique elements of the `cust_address_ntab` nested table column:

```
SELECT customer_id, SET(cust_address_ntab) address
  FROM customers_demo
  ORDER BY customer_id;


CUSTOMER_ID ADDRESS(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
----------- --------------------------------------------------------------------
        101 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('514 W Superior St', '46901', 'Kokomo', 'IN', 'US'))
        102 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('2515 Bloyd Ave', '46218', 'Indianapolis', 'IN', 'US'))
        103 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('8768 N State Rd 37', '47404', 'Bloomington', 'IN', 'US'))
        104 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('6445 Bay Harbor Ln', '46254', 'Indianapolis', 'IN', 'US'))
        105 CUST_ADDRESS_TAB_TYP(CUST_ADDRESS_TYP('4019 W 3Rd St', '47404', 'Bloomington', 'IN', 'US'))
. . .
```

The preceding example requires the table `customers_demo` and a nested table column containing data. Refer to "Multiset Operators " to create this table and nested table column.

SIGN

Syntax



Purpose

`SIGN` returns the sign of *n*. This function takes as an argument any numeric data type, or any nonnumeric data type that can be implicitly converted to `NUMBER`, and returns `NUMBER`.

For value of NUMBER type, the sign is:

- -1 if *n*<0
- 0 if *n*=0
- 1 if *n*>0

For binary floating-point numbers (BINARY_FLOAT and BINARY_DOUBLE), this function returns the sign bit of the number. The sign bit is:

- -1 if *n*<0
- +1 if *n*>=0 or *n*=NaN

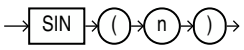
Examples

The following example indicates that the argument of the function (-15) is <0:

```
SELECT SIGN(-15) "Sign" FROM DUAL;

      Sign
----------
        -1
```

SIN

Syntax



Purpose

SIN returns the sine of *n* (an angle expressed in radians).

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is BINARY_FLOAT, then the function returns BINARY_DOUBLE. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

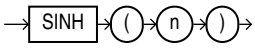
Examples

The following example returns the sine of 30 degrees:

```
SELECT SIN(30 * 3.14159265359/180)
 "Sine of 30 degrees" FROM DUAL;

Sine of 30 degrees
------------------
                .5
```

SINH

Syntax



Purpose

`SINH` returns the hyperbolic sine of *n*.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

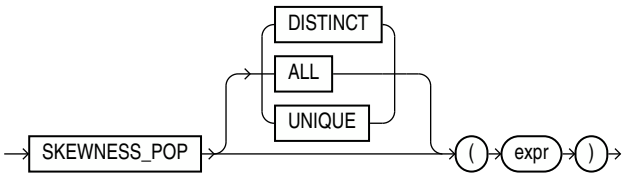
Examples

The following example returns the hyperbolic sine of 1:

```
SELECT SINH(1) "Hyperbolic sine of 1" FROM DUAL;

Hyperbolic sine of 1
--------------------
          1.17520119
```

SKEWNESS_POP

Syntax



Purpose

`SKEWNESS_POP` is an aggregate function that is primarily used to determine symmetry in a given distribution.

NULL values in `expr` are ignored.

Returns NULL if all rows in the group have NULL `expr` values.

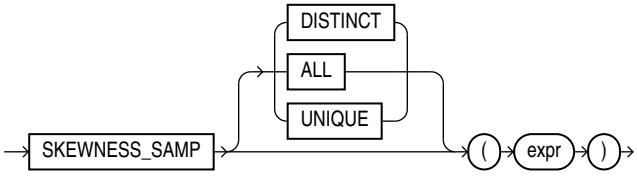
Returns 0 if there are one or two rows in `expr`.

For a given set of values, the result of population skewness (`SKEWNESS_POP`) and sample skewness (`SKEWNESS_SAMP`) are always deterministic. However, the values of `SKEWNESS_POP`

and SKEWNESS_SAMP differ. As the number of values in the data set increases, the difference between the computed values of SKEWNESS_SAMP and SKEWNESS_POP decreases.

SKEWNESS_SAMP

Syntax



Purpose

SKEWNESS_SAMP is an aggregate function that is primarily used to determine symmetry in a given distribution.

NULL values in expr are ignored.

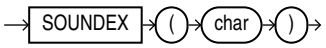
Returns NULL if all rows in the group have NULL expr values.

Returns 0 if there are one or two rows in expr.

For a given set of values, the result of population skewness (SKEWNESS_POP) and sample skewness (SKEWNESS_SAMP) are always deterministic. However, the values of SKEWNESS_POP and SKEWNESS_SAMP differ. As the number of values in the data set increases, the difference between the computed values of SKEWNESS_SAMP and SKEWNESS_POP decreases.

SOUNDEX

Syntax



Purpose

SOUNDEX returns a character string containing the phonetic representation of *char*. This function lets you compare words that are spelled differently, but sound alike in English.

The phonetic representation is defined in *The Art of Computer Programming*, Volume 3: Sorting and Searching, by Donald E. Knuth, as follows:

1. Retain the first letter of the string and remove all other occurrences of the following letters: a, e, h, i, o, u, w, y.
2. Assign numbers to the remaining letters (after the first) as follows:

```
b, f, p, v = 1
c, g, j, k, q, s, x, z = 2
d, t = 3
l = 4
m, n = 5
r = 6
```

3. If two or more letters with the same number were adjacent in the original name (before step 1), or adjacent except for any intervening h and w, then retain the first letter and omit rest of all the adjacent letters with same number.
4. Return the first four bytes padded with 0.

char can be of any of the data types CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The return value is the same data type as *char*.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of SOUNDEX

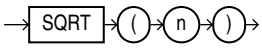
Examples

The following example returns the employees whose last names are a phonetic representation of "Smyth":

```
SELECT last_name, first_name
     FROM hr.employees
     WHERE SOUNDEX(last_name)
         = SOUNDEX('SMYTHE')
     ORDER BY last_name, first_name;

LAST_NAME  FIRST_NAME
---------- ----------
Smith      Lindsey
Smith      William
```

SQRT

Syntax



Purpose

SQRT returns the square root of *n*.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

- If *n* resolves to a NUMBER, then the value *n* cannot be negative. SQRT returns a real number.
- If *n* resolves to a binary floating-point number (BINARY_FLOAT or BINARY_DOUBLE):
  - If *n* >= 0, then the result is positive.
  - If *n* = -0, then the result is -0.
  - If *n* < 0, then the result is NaN.

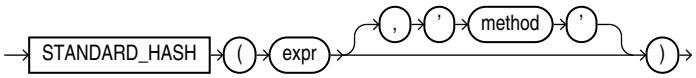
Examples

The following example returns the square root of 26:

```
SELECT SQRT(26) "Square root" FROM DUAL;

Square root
-----------
5.09901951
```

STANDARD_HASH

Syntax



Purpose

STANDARD_HASH computes a hash value for a given expression using one of several hash algorithms that are defined and standardized by the National Institute of Standards and Technology. This function is useful for performing authentication and maintaining data integrity in security applications such as digital signatures, checksums, and fingerprinting.

You can use the STANDARD_HASH function to create an index on an extended data type column. Refer to "Creating an Index on an Extended Data Type Column" for more information.

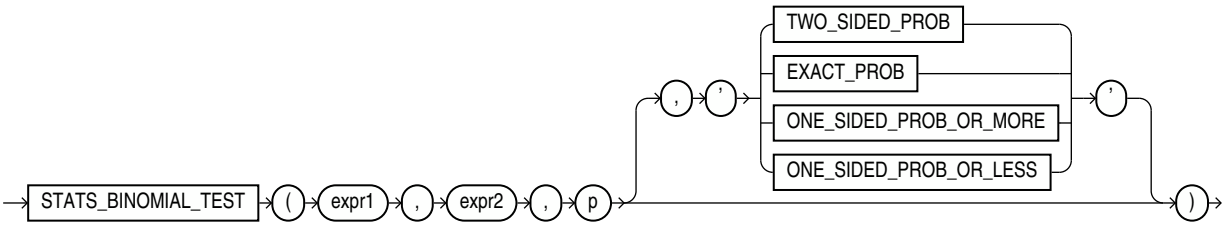
- The *expr* argument determines the data for which you want Oracle Database to compute a hash value. There are no restrictions on the length of data represented by *expr*, which commonly resolves to a column name. The *expr* cannot be a LONG or LOB type. It cannot be a user-defined object type. All other data types are supported for *expr*.
- The optional *method* argument lets you specify the name of the hash algorithm to be used. Valid algorithms are SHA1, SHA256, SHA384, SHA512 and MD5. If you omit this argument, then SHA1 is used.

The function returns a RAW value.

> **Note:**
>
> The `STANDARD_HASH` function is not identical to the one used internally by Oracle Database for hash partitioning.

STATS_BINOMIAL_TEST

Syntax



Purpose

`STATS_BINOMIAL_TEST` is an exact probability test used for dichotomous variables, where only two possible values exist. It tests the difference between a sample proportion and a given proportion. The sample size in such tests is usually small.

This function takes three required arguments: *expr1* is the sample being examined, *expr2* contains the values for which the proportion is expected to be, and *p* is a proportion to test against. The optional fourth argument lets you specify the meaning of the `NUMBER` value returned by this function, as shown in Table 7-3. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the fourth argument, then the default is `'TWO_SIDED_PROB'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `STATS_BINOMIAL_TEST`

Table 7-3 STATS_BINOMIAL Return Values

Argument	Return Value Meaning
`'TWO_SIDED_PROB'`	The probability that the given population proportion, *p*, could result in the observed proportion or a more extreme one.
`'EXACT_PROB'`	The probability that the given population proportion, *p*, could result in exactly the observed proportion.
`'ONE_SIDED_PROB_OR_MORE'`	The probability that the given population proportion, *p*, could result in the observed proportion or a larger one.
`'ONE_SIDED_PROB_OR_LESS'`	The probability that the given population proportion, *p*, could result in the observed proportion or a smaller one.

'EXACT_PROB' gives the probability of getting exactly proportion *p*. In cases where you want to test whether the proportion found in the sample is significantly different from a 50-50 split, *p* would normally be 0.50. If you want to test only whether the proportion is different, then use the return value 'TWO_SIDED_PROB'. If your test is whether the proportion is more than the value of *expr2*, then use the return value 'ONE_SIDED_PROB_OR_MORE'. If the test is to determine whether the proportion of *expr2* is less, then use the return value 'ONE_SIDED_PROB_OR_LESS'.

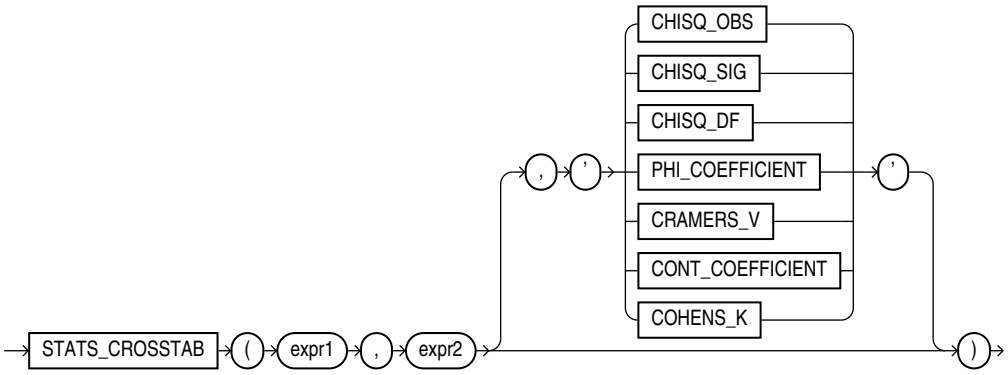
STATS_BINOMIAL_TEST Example

The following example determines the probability that reality exactly matches the number of men observed under the assumption that 69% of the population is composed of men:

```
SELECT AVG(DECODE(cust_gender, 'M', 1, 0)) real_proportion,
       STATS_BINOMIAL_TEST
         (cust_gender, 'M', 0.68, 'EXACT_PROB') exact,
       STATS_BINOMIAL_TEST
         (cust_gender, 'M', 0.68, 'ONE_SIDED_PROB_OR_LESS') prob_or_less
  FROM sh.customers;
```

STATS_CROSSTAB

Syntax



Purpose

Crosstabulation (commonly called crosstab) is a method used to analyze two nominal variables. The STATS_CROSSTAB function takes two required arguments: *expr1* and *expr2* are the two variables being analyzed. The optional third argument lets you specify the meaning of the NUMBER value returned by this function, as shown in Table 7-4. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is 'CHISQ_SIG'.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for STATS_CROSSTAB

Table 7-4 STATS_CROSSTAB Return Values

Argument	Return Value Meaning
'CHISQ_OBS'	Observed value of chi-squared
'CHISQ_SIG'	Significance of observed chi-squared
'CHISQ_DF'	Degree of freedom for chi-squared
'PHI_COEFFICIENT'	Phi coefficient
'CRAMERS_V'	Cramer's V statistic
'CONT_COEFFICIENT'	Contingency coefficient
'COHENS_K'	Cohen's kappa

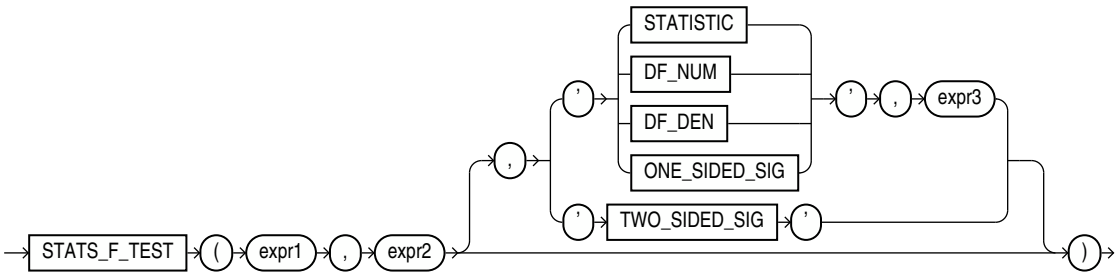
STATS_CROSSTAB Example

The following example determines the strength of the association between gender and income level:

```
SELECT STATS_CROSSTAB
         (cust_gender, cust_income_level, 'CHISQ_OBS') chi_squared,
       STATS_CROSSTAB
         (cust_gender, cust_income_level, 'CHISQ_SIG') p_value,
       STATS_CROSSTAB
         (cust_gender, cust_income_level, 'PHI_COEFFICIENT') phi_coefficient
  FROM sh.customers;

CHI_SQUARED    P_VALUE PHI_COEFFICIENT
----------- ---------- ---------------
 251.690705 1.2364E-47      .067367056
```

STATS_F_TEST

Syntax



Purpose

STATS_F_TEST tests whether two variances are significantly different. The observed value of *f* is the ratio of one variance to the other, so values very different from 1 usually indicate significant differences.

This function takes two required arguments: *expr1* is the grouping or independent variable and *expr2* is the sample of values. The optional third argument lets you specify the meaning of the NUMBER value returned by this function, as shown in Table 7-5. For this argument, you can

specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is `'TWO_SIDED_SIG'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `STATS_F_TEST`

Table 7-5 STATS_F_TEST Return Values

Argument	Return Value Meaning
`'STATISTIC'`	The observed value of *f*
`'DF_NUM'`	Degree of freedom for the numerator
`'DF_DEN'`	Degree of freedom for the denominator
`'ONE_SIDED_SIG'`	One-tailed significance of *f*
`'TWO_SIDED_SIG'`	Two-tailed significance of *f*

The one-tailed significance is always in relation to the upper tail. The final argument, *expr3*, indicates which of the two groups specified by *expr1* is the high value or numerator (the value whose rejection region is the upper tail).

The observed value of *f* is the ratio of the variance of one group to the variance of the second group. The significance of the observed value of *f* is the probability that the variances are different just by chance—a number between 0 and 1. A small value for the significance indicates that the variances are significantly different. The degree of freedom for each of the variances is the number of observations in the sample minus 1.

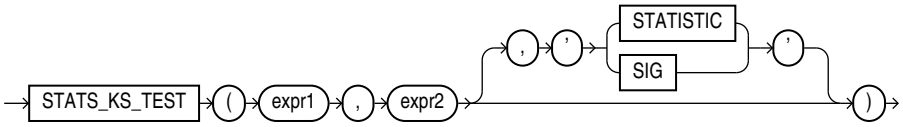
STATS_F_TEST Example

The following example determines whether the variance in credit limit between men and women is significantly different. The results, a p_value not close to zero, and an f_statistic close to 1, indicate that the difference between credit limits for men and women are not significant.

```
SELECT VARIANCE(DECODE(cust_gender, 'M', cust_credit_limit, null)) var_men,
       VARIANCE(DECODE(cust_gender, 'F', cust_credit_limit, null)) var_women,
       STATS_F_TEST(cust_gender, cust_credit_limit, 'STATISTIC', 'F') f_statistic,
       STATS_F_TEST(cust_gender, cust_credit_limit) two_sided_p_value
  FROM sh.customers;

   VAR_MEN  VAR_WOMEN F_STATISTIC TWO_SIDED_P_VALUE
---------- ---------- ----------- -----------------
12879896.7   13046865  1.01296348        .311928071
```

STATS_KS_TEST

Syntax



Purpose

`STATS_KS_TEST` is a Kolmogorov-Smirnov function that compares two samples to test whether they are from the same population or from populations that have the same distribution. It does not assume that the population from which the samples were taken is normally distributed.

This function takes two required arguments: *`expr1`* classifies the data into the two samples and *`expr2`* contains the values for each of the samples. If *`expr1`* classifies the data into only one sample or into more than two samples, then an error is raised. The optional third argument lets you specify the meaning of the `NUMBER` value returned by this function, as shown in Table 7-6. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is `'SIG'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `STATS_KS_TEST`

Table 7-6 STATS_KS_TEST Return Values

Argument	Return Value Meaning
`'STATISTIC'`	Observed value of *D*
`'SIG'`	Significance of *D*

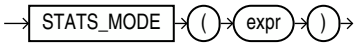
STATS_KS_TEST Example

Using the Kolmogorov Smirnov test, the following example determines whether the distribution of sales between men and women is due to chance:

```
SELECT stats_ks_test(cust_gender, amount_sold, 'STATISTIC') ks_statistic,
       stats_ks_test(cust_gender, amount_sold) p_value
  FROM sh.customers c, sh.sales s
  WHERE c.cust_id = s.cust_id;

KS_STATISTIC    P_VALUE
------------ ----------
  .003841396 .004080006
```

STATS_MODE

Syntax



Purpose

`STATS_MODE` takes as its argument a set of values and returns the value that occurs with the greatest frequency. If more than one mode exists, then Oracle Database chooses one and returns only that one value.

To obtain multiple modes (if multiple modes exist), you must use a combination of other functions, as shown in the hypothetical query:

```
SELECT x FROM (SELECT x, COUNT(x) AS cnt1
   FROM t GROUP BY x)
   WHERE cnt1 =
      (SELECT MAX(cnt2) FROM (SELECT COUNT(x) AS cnt2 FROM t GROUP BY x));
```

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `STATS_MODE` uses to compare character values for *`expr`* , and for the collation derivation rules, which define the collation assigned to the return value of this function when it is a character value

Examples

The following example returns the mode of salary per department in the `hr.employees` table:

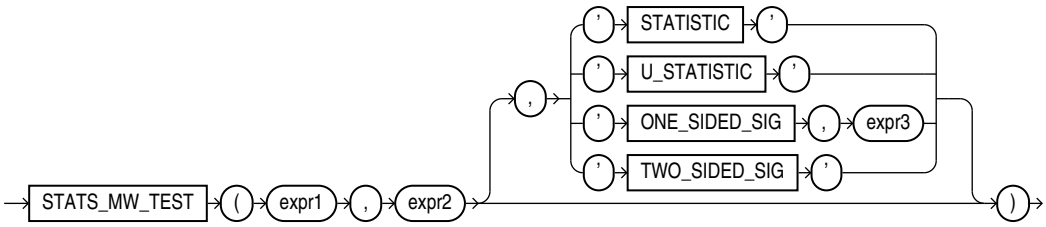
```
SELECT department_id, STATS_MODE(salary) FROM employees
   GROUP BY department_id
   ORDER BY department_id, stats_mode(salary);

DEPARTMENT_ID STATS_MODE(SALARY)
------------- ------------------
           10               4400
           20               6000
           30               2500
           40               6500
           50               2500
           60               4800
           70              10000
           80               9500
           90              17000
          100               6900
          110               8300
                            7000
```

If you need to retrieve all of the modes (in cases with multiple modes), you can do so using a combination of other functions, as shown in the next example:

```
SELECT commission_pct FROM
   (SELECT commission_pct, COUNT(commission_pct) AS cnt1 FROM employees
      GROUP BY commission_pct)
   WHERE cnt1 =
      (SELECT MAX (cnt2) FROM
         (SELECT COUNT(commission_pct) AS cnt2
         FROM employees GROUP BY commission_pct))
   ORDER BY commission_pct;

COMMISSION_PCT
--------------
            .2
            .3
```

STATS_MW_TEST

Syntax



Purpose

A Mann Whitney test compares two independent samples to test the null hypothesis that two populations have the same distribution function against the alternative hypothesis that the two distribution functions are different.

The `STATS_MW_TEST` does not assume that the differences between the samples are normally distributed, as do the `STATS_T_TEST_*` functions. This function takes two required arguments: *`expr1`* classifies the data into groups and *`expr2`* contains the values for each of the groups. The optional third argument lets you specify the meaning of the `NUMBER` value returned by this function, as shown in Table 7-7. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is `'TWO_SIDED_SIG'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for `STATS_MW_TEST`

Table 7-7 STATS_MW_TEST Return Values

Argument	Return Value Meaning
`'STATISTIC'`	The observed value of *Z*
`'U_STATISTIC'`	The observed value of *U*

Table 7-7 (Cont.) STATS_MW_TEST Return Values

Argument	Return Value Meaning
'ONE_SIDED_SIG'	One-tailed significance of *Z*
'TWO_SIDED_SIG'	Two-tailed significance of *Z*

The significance of the observed value of *Z* or *U* is the probability that the variances are different just by chance—a number between 0 and 1. A small value for the significance indicates that the variances are significantly different. The degree of freedom for each of the variances is the number of observations in the sample minus 1.

The one-tailed significance is always in relation to the upper tail. The final argument, *expr3*, indicates which of the two groups specified by expr1 is the high value (the value whose rejection region is the upper tail).

STATS_MW_TEST computes the probability that the samples are from the same distribution by checking the differences in the sums of the ranks of the values. If the samples come from the same distribution, then the sums should be close in value.

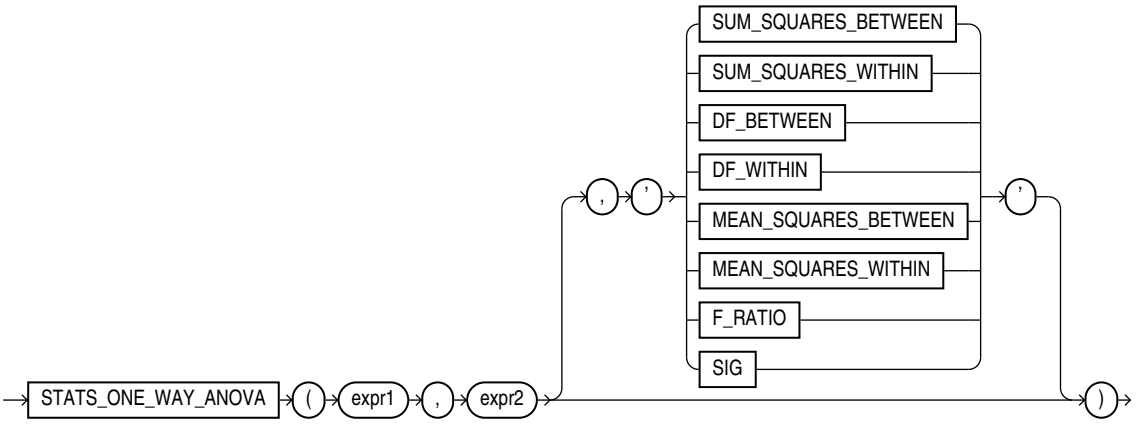
STATS_MW_TEST Example

Using the Mann Whitney test, the following example determines whether the distribution of sales between men and women is due to chance:

```
SELECT STATS_MW_TEST
         (cust_gender, amount_sold, 'STATISTIC') z_statistic,
       STATS_MW_TEST
         (cust_gender, amount_sold, 'ONE_SIDED_SIG', 'F') one_sided_p_value
  FROM sh.customers c, sh.sales s
  WHERE c.cust_id = s.cust_id;

Z_STATISTIC ONE_SIDED_P_VALUE
----------- -----------------
 -1.4011509        .080584471
```

STATS_ONE_WAY_ANOVA

Syntax



Purpose

The one-way analysis of variance function (STATS_ONE_WAY_ANOVA) tests differences in means (for groups or variables) for statistical significance by comparing two different estimates of variance. One estimate is based on the variances within each group or category. This is known as the **mean squares within** or **mean square error**. The other estimate is based on the variances among the means of the groups. This is known as the **mean squares between**. If the means of the groups are significantly different, then the mean squares between will be larger than expected and will not match the mean squares within. If the mean squares of the groups are consistent, then the two variance estimates will be about the same.

STATS_ONE_WAY_ANOVA takes two required arguments: *expr1* is an independent or grouping variable that divides the data into a set of groups and *expr2* is a dependent variable (a numeric expression) containing the values corresponding to each member of a group. The optional third argument lets you specify the meaning of the NUMBER value returned by this function, as shown in Table 7-8. For this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is 'SIG'.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for STATS_ONE_WAY_ANOVA

Table 7-8 STATS_ONE_WAY_ANOVA Return Values

Argument	Return Value Meaning
'SUM_SQUARES_BETEEN'	Sum of squares between groups
'SUM_SQUARES_WITHIN'	Sum of squares within groups
'DF_BETWEEN'	Degree of freedom between groups
'DF_WITHIN'	Degree of freedom within groups
'MEAN_SQUARES_BETWEEN'	Mean squares between groups
'MEAN_SQUARES_WITHIN'	Mean squares within groups
'F_RATIO'	Ratio of the mean squares between to the mean squares within (MSB/MSW)
'SIG'	Significance

The significance of one-way analysis of variance is determined by obtaining the one-tailed significance of an *f*-test on the ratio of the mean squares between and the mean squares within. The *f*-test should use one-tailed significance, because the mean squares between can be only equal to or larger than the mean squares within. Therefore, the significance returned by STATS_ONE_WAY_ANOVA is the probability that the differences between the groups happened by chance—a number between 0 and 1. The smaller the number, the greater the significance of the difference between the groups. Refer to the STATS_F_TEST for information on performing an *f*-test.

STATS_ONE_WAY_ANOVA Example

The following example determines the significance of the differences in mean sales within an income level and differences in mean sales between income levels. The results, p_values close to zero, indicate that, for both men and women, the difference in the amount of goods sold across different income levels is significant.

```
SELECT cust_gender,
       STATS_ONE_WAY_ANOVA(cust_income_level, amount_sold, 'F_RATIO') f_ratio,
       STATS_ONE_WAY_ANOVA(cust_income_level, amount_sold, 'SIG') p_value
  FROM sh.customers c, sh.sales s
  WHERE c.cust_id = s.cust_id
  GROUP BY cust_gender
  ORDER BY cust_gender;

C    F_RATIO    P_VALUE
- ---------- ----------
F 5.59536943 4.7840E-09
M  9.2865001 6.7139E-17
```

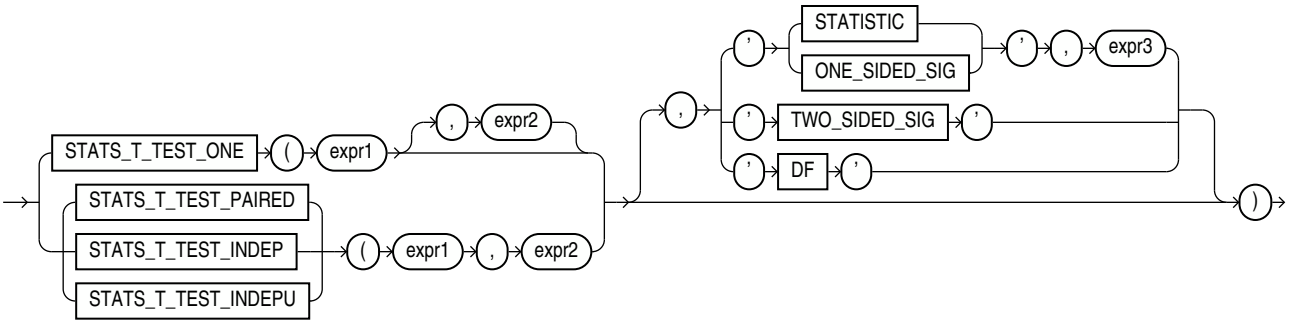
STATS_T_TEST_*

The *t*-test functions are:

- `STATS_T_TEST_ONE`: A one-sample *t*-test
- `STATS_T_TEST_PAIRED`: A two-sample, paired *t*-test (also known as a crossed *t*-test)
- `STATS_T_TEST_INDEP`: A *t*-test of two independent groups with the same variance (pooled variances)
- `STATS_T_TEST_INDEPU`: A *t*-test of two independent groups with unequal variance (unpooled variances)

Syntax

***stats_t_test*::=**



Purpose

The *t*-test measures the significance of a difference of means. You can use it to compare the means of two groups or the means of one group with a constant. Each *t*-test function takes two expression arguments, although the second expression is optional for the one-sample function (`STATS_T_TEST_ONE`). Each *t*-test function takes an optional third argument, which lets you specify the meaning of the `NUMBER` value returned by the function, as shown in Table 7-9. For

this argument, you can specify a text literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is `'TWO_SIDED_SIG'`.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules for the `STATS_T_TEST_*` functions

Table 7-9 STATS_T_TEST_* Return Values

Argument	Return Value Meaning
`'STATISTIC'`	The observed value of *t*
`'DF'`	Degree of freedom
`'ONE_SIDED_SIG'`	One-tailed significance of *t*
`'TWO_SIDED_SIG'`	Two-tailed significance of *t*

The two independent `STATS_T_TEST_*` functions can take a fourth argument (*expr3*) if the third argument is specified as `'STATISTIC'` or `'ONE_SIDED_SIG'`. In this case, *expr3* indicates which value of *expr1* is the high value, or the value whose rejection region is the upper tail.

The significance of the observed value of *t* is the probability that the value of *t* would have been obtained by chance—a number between 0 and 1. The smaller the value, the more significant the difference between the means. One-sided significance is always respect to the upper tail. For one-sample and paired *t*-test, the high value is the first expression. For independent *t*-test, the high value is the one specified by *expr3*.

The degree of freedom depends on the type of *t*-test that resulted in the observed value of *t*. For example, for a one-sample *t*-test (`STATS_T_TEST_ONE`), the degree of freedom is the number of observations in the sample minus 1.

STATS_T_TEST_ONE

In the `STATS_T_TEST_ONE` function, *expr1* is the sample and *expr2* is the constant mean against which the sample mean is compared. For this *t*-test only, *expr2* is optional; the constant mean defaults to 0. This function obtains the value of *t* by dividing the difference between the sample mean and the known mean by the standard error of the mean (rather than the standard error of the difference of the means, as for `STATS_T_TEST_PAIRED`).

STATS_T_TEST_ONE Example

The following example determines the significance of the difference between the average list price and the constant value 60:

```
SELECT AVG(prod_list_price) group_mean,
       STATS_T_TEST_ONE(prod_list_price, 60, 'STATISTIC') t_observed,
       STATS_T_TEST_ONE(prod_list_price, 60) two_sided_p_value
  FROM sh.products;

GROUP_MEAN T_OBSERVED TWO_SIDED_P_VALUE
---------- ---------- -----------------
139.545556 2.32107746        .023158537
```

STATS_T_TEST_PAIRED

In the STATS_T_TEST_PAIRED function, *expr1* and *expr2* are the two samples whose means are being compared. This function obtains the value of *t* by dividing the difference between the sample means by the standard error of the difference of the means (rather than the standard error of the mean, as for STATS_T_TEST_ONE).

STATS_T_TEST_INDEP and STATS_T_TEST_INDEPU

In the STATS_T_TEST_INDEP and STATS_T_TEST_INDEPU functions, *expr1* is the grouping column and *expr2* is the sample of values. The pooled variances version (STATS_T_TEST_INDEP) tests whether the means are the same or different for two distributions that have similar variances. The unpooled variances version (STATS_T_TEST_INDEPU) tests whether the means are the same or different even if the two distributions are known to have significantly different variances.

Before using these functions, it is advisable to determine whether the variances of the samples are significantly different. If they are, then the data may come from distributions with different shapes, and the difference of the means may not be very useful. You can perform an *f*-test to determine the difference of the variances. If they are not significantly different, use STATS_T_TEST_INDEP. If they are significantly different, use STATS_T_TEST_INDEPU. Refer to STATS_F_TEST for information on performing an *f*-test.

STATS_T_TEST_INDEP Example

The following example determines the significance of the difference between the average sales to men and women where the distributions are assumed to have similar (pooled) variances:

```
SELECT SUBSTR(cust_income_level, 1, 22) income_level,
      AVG(DECODE(cust_gender, 'M', amount_sold, null)) sold_to_men,
      AVG(DECODE(cust_gender, 'F', amount_sold, null)) sold_to_women,
      STATS_T_TEST_INDEP(cust_gender, amount_sold, 'STATISTIC', 'F') t_observed,
      STATS_T_TEST_INDEP(cust_gender, amount_sold) two_sided_p_value
  FROM sh.customers c, sh.sales s
  WHERE c.cust_id = s.cust_id
  GROUP BY ROLLUP(cust_income_level)
  ORDER BY income_level, sold_to_men, sold_to_women, t_observed;

INCOME_LEVEL           SOLD_TO_MEN SOLD_TO_WOMEN T_OBSERVED TWO_SIDED_P_VALUE
---------------------- ----------- ------------- ---------- -----------------
A: Below 30,000          105.28349    99.4281447 -1.9880629        .046811482
B: 30,000 - 49,999       102.59651    109.829642 3.04330875        .002341053
C: 50,000 - 69,999      105.627588    110.127931 2.36148671        .018204221
D: 70,000 - 89,999      106.630299     110.47287 2.28496443        .022316997
E: 90,000 - 109,999     103.396741    101.610416 -1.2544577        .209677823
F: 110,000 - 129,999     106.76476    105.981312 -.60444998        .545545304
G: 130,000 - 149,999    108.877532     107.31377 -.85298245        .393671218
H: 150,000 - 169,999    110.987258    107.152191 -1.9062363        .056622983
I: 170,000 - 189,999    102.808238     107.43556 2.18477851        .028908566
J: 190,000 - 249,999    108.040564    115.343356 2.58313425        .009794516
K: 250,000 - 299,999    112.377993    108.196097 -1.4107871        .158316973
L: 300,000 and above    120.970235    112.216342 -2.0642868        .039003862
                        107.121845     113.80441 .686144393        .492670059
                        106.663769    107.276386 1.08013499        .280082357
14 rows selected.
```

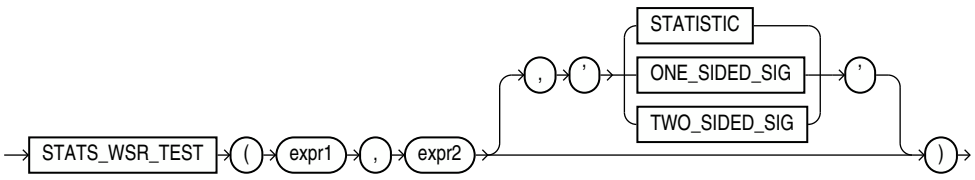
STATS_T_TEST_INDEPU Example

The following example determines the significance of the difference between the average sales to men and women where the distributions are known to have significantly different (unpooled) variances:

```
SELECT SUBSTR(cust_income_level, 1, 22) income_level,
       AVG(DECODE(cust_gender, 'M', amount_sold, null)) sold_to_men,
       AVG(DECODE(cust_gender, 'F', amount_sold, null)) sold_to_women,
       STATS_T_TEST_INDEPU(cust_gender, amount_sold, 'STATISTIC', 'F') t_observed,
       STATS_T_TEST_INDEPU(cust_gender, amount_sold) two_sided_p_value
  FROM sh.customers c, sh.sales s
  WHERE c.cust_id = s.cust_id
  GROUP BY ROLLUP(cust_income_level)
  ORDER BY income_level, sold_to_men, sold_to_women, t_observed;

INCOME_LEVEL           SOLD_TO_MEN SOLD_TO_WOMEN T_OBSERVED TWO_SIDED_P_VALUE
---------------------- ----------- ------------- ---------- -----------------
A: Below 30,000          105.28349    99.4281447 -2.0542592        .039964704
B: 30,000 - 49,999       102.59651    109.829642 2.96922332        .002987742
C: 50,000 - 69,999      105.627588    110.127931  2.3496854        .018792277
D: 70,000 - 89,999      106.630299     110.47287 2.26839281        .023307831
E: 90,000 - 109,999     103.396741    101.610416 -1.2603509        .207545662
F: 110,000 - 129,999     106.76476    105.981312 -.60580011        .544648553
G: 130,000 - 149,999    108.877532     107.31377 -.85219781        .394107755
H: 150,000 - 169,999    110.987258    107.152191 -1.9451486        .051762624
I: 170,000 - 189,999    102.808238     107.43556 2.14966921        .031587875
J: 190,000 - 249,999    108.040564    115.343356 2.54749867        .010854966
K: 250,000 - 299,999    112.377993    108.196097 -1.4115514        .158091676
L: 300,000 and above    120.970235    112.216342 -2.0726194        .038225611
                        107.121845     113.80441 .689462437        .490595765
                        106.663769    107.276386 1.07853782        .280794207
14 rows selected.
```

STATS_WSR_TEST

Syntax



Purpose

`STATS_WSR_TEST` is a Wilcoxon Signed Ranks test of paired samples to determine whether the median of the differences between the samples is significantly different from zero. The absolute values of the differences are ordered and assigned ranks. Then the null hypothesis states that the sum of the ranks of the positive differences is equal to the sum of the ranks of the negative differences.

This function takes two required arguments: *expr1* and *expr2* are the two samples being analyzed. The optional third argument lets you specify the meaning of the `NUMBER` value returned by this function, as shown in Table 7-10. For this argument, you can specify a text

literal, or a bind variable or expression that evaluates to a constant character value. If you omit the third argument, then the default is `'TWO_SIDED_SIG'`.

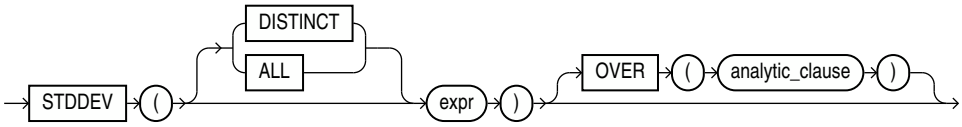
Table 7-10 STATS_WSR_TEST_* Return Values

Argument	Return Value Meaning
`'STATISTIC'`	The observed value of *Z*
`'ONE_SIDED_SIG'`	One-tailed significance of *Z*
`'TWO_SIDED_SIG'`	Two-tailed significance of *Z*

One-sided significance is always with respect to the upper tail. The high value (the value whose rejection region is the upper tail) is *expr1*.

STDDEV

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`STDDEV` returns the sample standard deviation of *expr*, a set of numbers. You can use it as both an aggregate and analytic function. It differs from `STDDEV_SAMP` in that `STDDEV` returns zero when it has only 1 row of input data, whereas `STDDEV_SAMP` returns null.

Oracle Database calculates the standard deviation as the square root of the variance defined for the `VARIANCE` aggregate function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

If you specify `DISTINCT`, then you can specify only the *query_partition_clause* of the *analytic_clause*. The *order_by_clause* and *windowing_clause* are not allowed.

> **See Also:**
>
> - "Aggregate Functions ", VARIANCE , and STDDEV_SAMP
> - "About SQL Expressions " for information on valid forms of *expr*

Aggregate Examples

The following example returns the standard deviation of the salaries in the sample `hr.employees` table:

```
SELECT STDDEV(salary) "Deviation"
   FROM employees;

 Deviation
----------
3909.36575
```

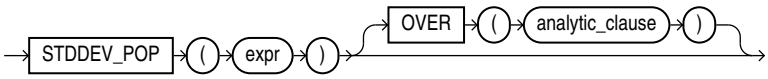
Analytic Examples

The query in the following example returns the cumulative standard deviation of the salaries in Department 80 in the sample table `hr.employees`, ordered by `hire_date`:

```
SELECT last_name, salary,
   STDDEV(salary) OVER (ORDER BY hire_date) "StdDev"
   FROM employees
   WHERE department_id = 30
   ORDER BY last_name, salary, "StdDev";

LAST_NAME                     SALARY     StdDev
------------------------- ---------- ----------
Baida                           2900 4035.26125
Colmenares                      2500 3362.58829
Himuro                          2600  3649.2465
Khoo                            3100 5586.14357
Raphaely                       11000          0
Tobias                          2800  4650.0896
```

STDDEV_POP

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`STDDEV_POP` computes the population standard deviation and returns the square root of the population variance. You can use it as both an aggregate and analytic function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

This function is the same as the square root of the `VAR_POP` function. When `VAR_POP` returns null, this function returns null.

> **See Also:**
>
> - "Aggregate Functions " and VAR_POP
> - "About SQL Expressions " for information on valid forms of *expr*

Aggregate Example

The following example returns the population and sample standard deviations of the amount of sales in the sample table `sh.sales`:

```
SELECT STDDEV_POP(amount_sold) "Pop",
   STDDEV_SAMP(amount_sold) "Samp"
   FROM sales;

       Pop       Samp
---------- ----------
896.355151 896.355592
```

Analytic Example

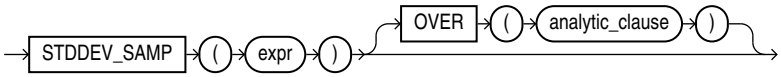
The following example returns the population standard deviations of salaries in the sample `hr.employees` table by department:

```
SELECT department_id, last_name, salary,
   STDDEV_POP(salary) OVER (PARTITION BY department_id) AS pop_std
   FROM employees
   ORDER BY department_id, last_name, salary, pop_std;

DEPARTMENT_ID LAST_NAME                 SALARY    POP_STD
------------- ------------------------- ---------- ----------
           10 Whalen                      4400          0
           20 Fay                         6000       3500
           20 Hartstein                  13000       3500
           30 Baida                       2900  3069.6091
. . .
          100 Urman                      7800 1644.18166
          110 Gietz                      8300       1850
```

```
                110 Higgins                       12000       1850
                    Grant                          7000          0
```

STDDEV_SAMP

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`STDDEV_SAMP` computes the cumulative sample standard deviation and returns the square root of the sample variance. You can use it as both an aggregate and analytic function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

This function is same as the square root of the `VAR_SAMP` function. When `VAR_SAMP` returns null, this function returns null.

> **See Also:**
>
> - "Aggregate Functions " and VAR_SAMP
> - "About SQL Expressions " for information on valid forms of *expr*

Aggregate Example

Refer to the aggregate example for STDDEV_POP .

Analytic Example

The following example returns the sample standard deviation of salaries in the `employees` table by department:

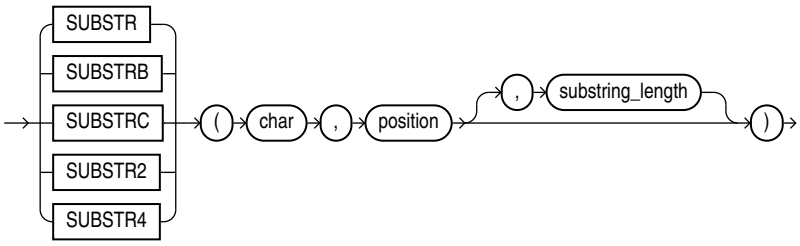
```
SELECT department_id, last_name, hire_date, salary,
   STDDEV_SAMP(salary) OVER (PARTITION BY department_id
      ORDER BY hire_date
```

```
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cum_sdev
   FROM employees
   ORDER BY department_id, last_name, hire_date, salary, cum_sdev;

DEPARTMENT_ID LAST_NAME       HIRE_DATE     SALARY   CUM_SDEV
------------- --------------- --------- ---------- ----------
           10 Whalen          17-SEP-03       4400
           20 Fay             17-AUG-05       6000 4949.74747
           20 Hartstein       17-FEB-04      13000
           30 Baida           24-DEC-05       2900 4035.26125
           30 Colmenares      10-AUG-07       2500 3362.58829
           30 Himuro          15-NOV-06       2600  3649.2465
           30 Khoo            18-MAY-03       3100 5586.14357
           30 Raphaely        07-DEC-02      11000
. . .
          100 Greenberg       17-AUG-02      12008  2126.9772
          100 Popp            07-DEC-07       6900 1804.13155
          100 Sciarra         30-SEP-05       7700 1929.76233
          100 Urman           07-MAR-06       7800 1788.92504
          110 Gietz           07-JUN-02       8300 2621.95194
          110 Higgins         07-JUN-02      12008
              Grant           24-MAY-07       7000
```

SUBSTR

Syntax

***substr*::=**



Purpose

The `SUBSTR` functions return a portion of *`char`*, beginning at character *`position`*, *`substring_length`* characters long. `SUBSTR` calculates lengths using characters as defined by the input character set. `SUBSTRB` uses bytes instead of characters. `SUBSTRC` uses Unicode complete characters. `SUBSTR2` uses UCS2 code points. `SUBSTR4` uses UCS4 code points.

- If *`position`* is 0, then it is treated as 1.
- If *`position`* is positive, then Oracle Database counts from the beginning of *`char`* to find the first character.
- If *`position`* is negative, then Oracle counts backward from the end of *`char`*.
- If *`substring_length`* is omitted, then Oracle returns all characters to the end of *`char`*. If *`substring_length`* is less than 1, then Oracle returns null.

`char` can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The exceptions are `SUBSTRC`, `SUBSTR2`, and `SUBSTR4`, which do not allow *`char`* to be a `CLOB` or `NCLOB`. Both *`position`* and *`substring_length`* must be of data type `NUMBER`, or any data type that can

be implicitly converted to NUMBER, and must resolve to an integer. The return value is the same data type as *char*, except that for a CHAR argument a VARCHAR2 value is returned, and for an NCHAR argument an NVARCHAR2 value is returned. Floating-point numbers passed as arguments to SUBSTR are automatically converted to integers.

> **See Also:**
>
> - For a complete description of character length see *Oracle Database Globalization Support Guide* and *Oracle Database SecureFiles and Large Objects Developer's Guide*
> - *Oracle Database Globalization Support Guide* for more information about SUBSTR functions and length semantics in different locales
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of SUBSTR

Examples

The following example returns several specified substrings of "ABCDEFG":

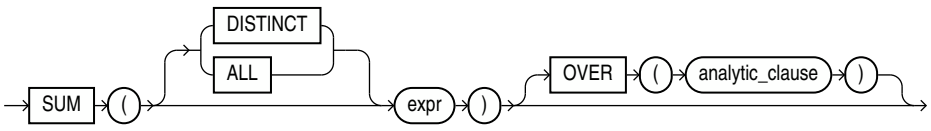
```
SELECT SUBSTR('ABCDEFG',3,4) "Substring"
     FROM DUAL;

Substring
---------
CDEF

SELECT SUBSTR('ABCDEFG',-5,4) "Substring"
     FROM DUAL;

Substring
---------
CDEF
```

Assume a double-byte database character set:

```
SELECT SUBSTRB('ABCDEFG',5,4.2) "Substring with bytes"
     FROM DUAL;

Substring with bytes
--------------------
CD
```

SUM

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`SUM` returns the sum of values of *`expr`*. You can use it as an aggregate or analytic function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

Release 23 adds support for `INTERVAL` interval data types. However interval data types cannot be implicitly converted to a numeric data type. If the input is an `INTERVAL`, the function returns an `INTERVAL` with the same units as the input.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

If you specify `DISTINCT`, then you can specify only the *`query_partition_clause`* of the *`analytic_clause`*. The *`order_by_clause`* and *`windowing_clause`* are not allowed.

> **See Also:**
>
> "About SQL Expressions " for information on valid forms of *`expr`* and "Aggregate Functions "

Vector Aggregate Operations

You can use `SUM` to perform vector addition operations on non-null inputs.

`expr` must evaluate to `VECTOR` and must not be `BINARY` vectors. The returned vector has the same number of dimensions as the input, and the format is always `FLOAT64`. For flexible number of dimensions, all inputs must have the same number of dimensions within each aggregation group.

NULL vectors are ignored. They are not counted when calculating the average vector. If all inputs within an aggregation group are NULL, the result is NULL for that group. If a certain dimension overflows when applying arithmetic operations, an error is raised.

Rules

- `DISTINCT` syntax is not allowed.
- Only `GROUP BY` and `GROUP BY ROLLUP` are supported.
- Analytic functions are not supported for input arguments of type `VECTOR`.

See *Arithmetic Operators*of the *AI Vector Search User's Guide* for examples.

Aggregate Example

The following example calculates the sum of all salaries in the sample `hr.employees` table:

```
SELECT SUM(salary) "Total"
   FROM employees;

   Total
----------
    691400
```

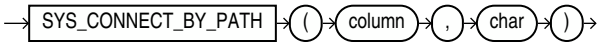
Analytic Example

The following example calculates, for each manager in the sample table `hr.employees`, a cumulative total of salaries of employees who answer to that manager that are equal to or less than the current salary. You can see that Raphaely and Cambrault have the same cumulative total. This is because Raphaely and Cambrault have the identical salaries, so Oracle Database adds together their salary values and applies the same cumulative total to both rows.

```
SELECT manager_id, last_name, salary,
   SUM(salary) OVER (PARTITION BY manager_id ORDER BY salary
   RANGE UNBOUNDED PRECEDING) l_csum
   FROM employees
   ORDER BY manager_id, last_name, salary, l_csum;

MANAGER_ID LAST_NAME                 SALARY     L_CSUM
---------- ------------------------- ---------- ----------
       100 Cambrault                      11000      68900
       100 De Haan                        17000     155400
       100 Errazuriz                      12000      80900
       100 Fripp                           8200      36400
       100 Hartstein                      13000      93900
       100 Kaufling                        7900      20200
       100 Kochhar                        17000     155400
       100 Mourgos                         5800       5800
       100 Partners                       13500     107400
       100 Raphaely                       11000      68900
       100 Russell                        14000     121400
. . .
       149 Hutton                          8800      39000
       149 Johnson                         6200       6200
       149 Livingston                      8400      21600
       149 Taylor                          8600      30200
       201 Fay                             6000       6000
       205 Gietz                           8300       8300
           King                           24000      24000
```

SYS_CONNECT_BY_PATH

Syntax



Purpose

`SYS_CONNECT_BY_PATH` is valid only in hierarchical queries. It returns the path of a column value from root to node, with column values separated by *`char`* for each row returned by `CONNECT BY` condition.

Both *`column`* and *`char`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The string returned is of `VARCHAR2` data type and is in the same character set as *`column`*.

> **See Also:**
>
> - "Hierarchical Queries " for more information about hierarchical queries and `CONNECT BY` conditions
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `SYS_CONNECT_BY_PATH`

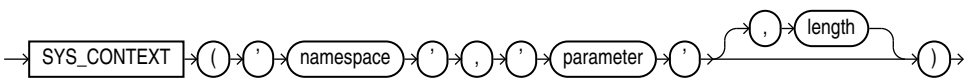
Examples

The following example returns the path of employee names from employee `Kochhar` to all employees of `Kochhar` (and their employees):

```
SELECT LPAD(' ', 2*level-1)||SYS_CONNECT_BY_PATH(last_name, '/') "Path"
   FROM employees
   START WITH last_name = 'Kochhar'
   CONNECT BY PRIOR employee_id = manager_id;

Path
------------------------------
     /Kochhar/Greenberg/Chen
     /Kochhar/Greenberg/Faviet
     /Kochhar/Greenberg/Popp
     /Kochhar/Greenberg/Sciarra
     /Kochhar/Greenberg/Urman
     /Kochhar/Higgins/Gietz
   /Kochhar/Baer
   /Kochhar/Greenberg
   /Kochhar/Higgins
   /Kochhar/Mavris
   /Kochhar/Whalen
 /Kochhar
```

SYS_CONTEXT

Syntax



Purpose

SYS_CONTEXT returns the value of *parameter* associated with the context *namespace* at the current instant. You can use this function in both SQL and PL/SQL statements. SYS_CONTEXT must be executed locally.

For *namespace* and *parameter*, you can specify either a string or an expression that resolves to a string designating a namespace or an attribute. If you specify literal arguments for *namespace* and *parameter*, and you are using SYS_CONTEXT explicitly in a SQL statement—rather than in a PL/SQL function that in turn is in mentioned in a SQL statement—then Oracle Database evaluates SYS_CONTEXT only once per SQL statement execution for each call site that invokes the SYS_CONTEXT function.

The context *namespace* must already have been created, and the associated *parameter* and its value must also have been set using the DBMS_SESSION.set_context procedure. The *namespace* must be a valid identifier. The *parameter* name can be any string. It is not case sensitive, but it cannot exceed 30 bytes in length.

The data type of the return value is VARCHAR2. The default maximum size of the return value is 256 bytes. You can override this default by specifying the optional *length* parameter, which must be a NUMBER or a value that can be implicitly converted to NUMBER. The valid range of values is 1 to 4000 bytes. If you specify an invalid value, then Oracle Database ignores it and uses the default.

Oracle provides the following built-in namespaces:

- USERENV - Describes the current session. The predefined parameters of namespace USERENV are listed in Table 7-11.
- SYS_SESSION_ROLES - Indicates whether a specified role is currently enabled for the session. Oracle Database evaluates the SYS_SESSION_ROLES context for the current user, and assumes the defining user's role when it evaluates SYS_SESSION_ROLES within a definer's rights procedure or function. An alternative to using SYS_SESSION_ROLES to find the login user's enabled roles in a definer's rights procedure is to use the DBMS_SESSION:SESSION_IS_ROLE_ENABLED function. Invoker's rights, procedures or functions, and/or code based access control (CBAC) are also alternatives.

See Also:

- Using Code Based Access Control for Definer's Rights and Invoker's Rights
- *Oracle Database Security Guide* for information on using the application context feature in your application development
- CREATE CONTEXT for information on creating user-defined context namespaces
- *Oracle Database PL/SQL Packages and Types Reference* for information on the DBMS_SESSION.set_context procedure
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of SYS_CONTEXT

Examples

The following statement returns the name of the user who logged onto the database:

```
CONNECT OE
Enter password: password

SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER')
   FROM DUAL;

SYS_CONTEXT ('USERENV', 'SESSION_USER')
----------------------------------------
OE
```

The following example queries the `SESSION_ROLES` data dictionary view to show that `RESOURCE` is the only role currently enabled for the session. It then uses the `SYS_CONTEXT` function to show that the `RESOURCE` role is currently enabled for the session and the `DBA` role is not.

```
CONNECT OE
Enter password: password

SELECT role FROM session_roles;

ROLE
--------
RESOURCE

SELECT SYS_CONTEXT('SYS_SESSION_ROLES', 'RESOURCE')
  FROM DUAL

SYS_CONTEXT('SYS_SESSION_ROLES','RESOURCE')
-------------------------------------
TRUE

SELECT SYS_CONTEXT('SYS_SESSION_ROLES', 'DBA')
  FROM DUAL;

SYS_CONTEXT('SYS_SESSION_ROLES','DBA')
-------------------------------------
FALSE
```

> **Note:**
>
> For simplicity in demonstrating this feature, these examples do not perform the password management techniques that a deployed system normally uses. In a production environment, follow the Oracle Database password management guidelines, and disable any sample accounts. See *Oracle Database Security Guide* for password management guidelines and other security recommendations.

The following hypothetical example returns the group number that was set as the value for the attribute `group_no` in the PL/SQL package that was associated with the context `hr_apps` when `hr_apps` was created:

```
SELECT SYS_CONTEXT ('hr_apps', 'group_no') "User Group"
   FROM DUAL;
```

Starting with Oracle Database 23ai, users authenticating to the database using the legacy RADIUS API are not granted administrative privileges such as SYSDBA or SYSBACKUP.

In Oracle Database 23ai Oracle introduces a new RADIUS API that uses the latest standards to grant administrative privileges to users.

You must ensure that the database connection to the database uses the new RADIUS API and that you are using the Oracle Database 23ai client to connect to the Oracle Database 23ai server.

Table 7-11 Predefined Parameters of Namespace USERENV

Parameter	Return Value
ACTION	Identifies the position in the module (application name) and is set through the DBMS_APPLICATION_INFO package or OCI.
AUDITED_CURSORID	Returns the cursor ID of the SQL that triggered the audit. This parameter is not valid in a fine-grained auditing environment. If you specify it in such an environment, then Oracle Database always returns null.
AUTHENTICATED_IDENTITY	Returns the identity used in authentication. In the list that follows, the type of user is followed by the value returned: • Kerberos-authenticated enterprise user: kerberos principal name • Kerberos-authenticated external user : kerberos principal name; same as the schema name • SSL-authenticated enterprise user: the DN in the user's PKI certificate • SSL-authenticated external user: the DN in the user's PKI certificate • Password-authenticated enterprise user: nickname; same as the login name • Password-authenticated database user: the database username; same as the schema name • OS-authenticated external user: the external operating system user name • Radius-authenticated external user: the schema name • Proxy with DN : Oracle Internet Directory DN of the client • Proxy with certificate: certificate DN of the client • For single session proxy or dual session proxy without client authentication: database user name if proxy is a local database user; nickname if proxy is an enterprise user. For dual session proxy with client authentication: database user name if client is a local database user; nickname if client is an enterprise user. • SYSDBA/SYSOPER using Password File: login name • SYSDBA/SYSOPER using OS authentication: operating system user name
AUTHENTICATION_DATA	Data being used to authenticate the login user. For X.503 certificate authenticated sessions, this field returns the context of the certificate in HEX2 format. **Note:** You can change the return value of the AUTHENTICATION_DATA attribute using the *length* parameter of the syntax. Values of up to 4000 are accepted. This is the only attribute of USERENV for which Oracle Database implements such a change.

Table 7-11 (Cont.) Predefined Parameters of Namespace USERENV

Parameter	Return Value
AUTHENTICATION_METHOD	Returns the method of authentication. In the list that follows, the type of user is followed by the method returned: • Password-authenticated enterprise user, local database user, or user with the SYSDBA or SYSOPER administrative privilege using a password file; proxy with username using password: PASSWORD • Password-authenticated enterprise user, local database user, or user with the SYSDBA or SYSOPER administrative privilege using a password file; proxy with username using password: PASSWORD_GLOBAL • Kerberos-authenticated enterprise user or external user (with no administrative privileges): KERBEROS • Kerberos-authenticated enterprise user (with administrative privileges): KERBEROS_GLOBAL • Kerberos-authenticated external user (with administrative privileges): KERBEROS_EXTERNAL • SSL-authenticated enterprise or external user (with no administrative privileges): SSL • SSL-authenticated enterprise user (with administrative privileges): SSL_GLOBAL • SSL-authenticated external user (with administrative privileges): SSL_EXTERNAL • Radius-authenticated external user: RADIUS • OS-authenticated external user or use with the SYSDBA or SYSOPER administrative privilege: OS • Proxy authentication: AUTHENTICATION_METHOD used during authentication of PROXY USER with "_PROXY" added at end. For example, if a proxy user uses PASSWORD to connect to the database, then the AUTHENTICATION_METHOD will be PASSWORD_PROXY. In the case of dual session proxy without client authentication: PROXYUSER_AUTHENTICATED_PROXY • Background process (job queue slave process): JOB • Parallel Query Slave process: PQ_SLAVE For non-administrative connections, you can use IDENTIFICATION_TYPE to distinguish between external and enterprise users when the authentication method is PASSWORD, KERBEROS, or SSL. For administrative connections, AUTHENTICATION_METHOD is sufficient for the PASSWORD, SSL_EXTERNAL, and SSL_GLOBAL authentication methods.
BG_JOB_ID	Job ID of the current session if it was established by an Oracle Database background process. Null if the session was not established by a background process.
CDB_DOMAIN	CDB_DOMAIN is the DB_DOMAIN of the CDB and is the same for all the PDBs associated with it.
CDB_NAME	If queried while connected to a multitenant container database (CDB), returns the name of the CDB. Otherwise, returns null.
CLIENT_IDENTIFIER	Returns an identifier that is set by the application through the DBMS_SESSION.SET_IDENTIFIER procedure, the OCI attribute OCI_ATTR_CLIENT_IDENTIFIER, or Oracle Dynamic Monitoring Service (DMS). This attribute is used by various database components to identify lightweight application users who authenticate as the same database user.
CLIENT_INFO	Returns up to 64 bytes of user session information that can be stored by an application using the DBMS_APPLICATION_INFO package.
CLIENT_PROGRAM_NAME	The name of the program used for the database session.
CLOUD_MIGRATION_MODE	The session parameter to specify ON when CLOUD_MIGRATION_MODE is TRUE, else OFF.

Table 7-11 (Cont.) Predefined Parameters of Namespace USERENV

Parameter	Return Value
CLOUD_SERVICE	Only valid for cloud implementations. Returns DWCS on autonomous database management systems (ADW), OLTP on autonomous transaction processing systems (ATP), and JDCS on autonomous JSON database systems.
CON_ID	Returns the current container ID that the session is connected to.
CON_NAME	Returns the current container name that the session is connected to.
CURRENT_BIND	The bind variables for fine-grained auditing. You can specify this attribute only inside the event handler for the fine-grained auditing feature.
CURRENT_EDITION_ID	The identifier of the current edition.
CURRENT_EDITION_NAME	The name of the current edition.
CURRENT_SCHEMA	The name of the currently active default schema. This value may change during the duration of a session through use of an ALTER SESSION SET CURRENT_SCHEMA statement. This may also change during the duration of a session to reflect the owner of any active definer's rights object. When used directly in the body of a view definition, this returns the default schema used when executing the cursor that is using the view; it does not respect views used in the cursor as being definer's rights. **Note**: Oracle recommends against issuing the SQL statement ALTER SESSION SET CURRENT_SCHEMA from within all types of stored PL/SQL units except logon triggers.
CURRENT_SCHEMAID	Identifier of the currently active default schema.
CURRENT_SQL CURRENT_SQL*n*	CURRENT_SQL returns the first 4K bytes of the current SQL that triggered the fine-grained auditing event. The CURRENT_SQL*n* attributes return subsequent 4K-byte increments, where *n* can be an integer from 1 to 7, inclusive. CURRENT_SQL1 returns bytes 4K to 8K; CURRENT_SQL2 returns bytes 8K to 12K, and so forth. You can specify these attributes only inside the event handler for the fine-grained auditing feature.
CURRENT_SQL_LENGTH	The length of the current SQL statement that triggers fine-grained audit or row-level security (RLS) policy functions or event handlers. You can specify this attribute only inside the event handler for the fine-grained auditing feature.
CURRENT_USER	The name of the database user whose privileges are currently active. This may change during the duration of a database session as Real Application Security sessions are attached or detached, or to reflect the owner of any active definer's rights object. When no definer's rights object is active, CURRENT_USER returns the same value as SESSION_USER. When used directly in the body of a view definition, this returns the user that is executing the cursor that is using the view; it does not respect views used in the cursor as being definer's rights. For enterprise users, returns schema. If a Real Application Security user is currently active, returns user XS$NULL.
CURRENT_USERID	The identifier of the database user whose privileges are currently active.
DATABASE_ROLE	The database role using the SYS_CONTEXT function with the USERENV namespace. The role is one of the following: PRIMARY, PHYSICAL STANDBY, LOGICAL STANDBY, SNAPSHOT STANDBY, TRUE CACHE .
DB_DOMAIN	Domain of the database as specified in the DB_DOMAIN initialization parameter.
DB_NAME	Name of the database as specified in the DB_NAME initialization parameter.
DB_SUPPLEMENTAL_LOG_LEVEL	If supplemental logging is enabled, returns a string containing the list of enabled supplemental logging levels. Possible values are: ALL_COLUMN, FOREIGN_KEY, MINIMAL, PRIMARY_KEY, PROCEDURAL, and UNIQUE_INDEX. If supplemental logging is not enabled, returns null.

Table 7-11 (Cont.) Predefined Parameters of Namespace USERENV

Parameter	Return Value
DB_UNIQUE_NAME	Name of the database as specified in the DB_UNIQUE_NAME initialization parameter.
DBLINK_INFO	Returns the source of a database link session. Specifically, it returns a string of the form: SOURCE_GLOBAL_NAME=*dblink_src_global_name*, DBLINK_NAME=*dblink_name*, SOURCE_AUDIT_SESSIONID=*dblink_src_audit_sessionid* For a multitenant database, it returns the string above with an additional field SOURCE_DB_NAME : SOURCE_GLOBAL_NAME=*dblink_src_global_name*, SOURCE_DB_NAME=*source_database_name*, DBLINK_NAME=*dblink_name*, SOURCE_AUDIT_SESSIONID=*dblink_src_audit_sessionid* where: • *dblink_src_global_name* is the unique global name of the source database • *dblink_name* is the name of the database link on the source database • *dblink_src_audit_sessionid* is the audit session ID of the session on the source database that initiated the connection to the remote database using *dblink_name* • SOURCE_DB_NAME is the database identifier of the source database
DRAIN_STATUS	Displays the draining status for the current session. Returns DRAINING if the session is a candidate for drain else returns NONE.
ENTRYID	The current audit entry number. The audit entryid sequence is shared between fine-grained audit records and regular audit records. You cannot use this attribute in distributed SQL statements. The correct auditing entry identifier can be seen only through an audit handler for standard or fine-grained audit.
ENTERPRISE_IDENTITY	Returns the user's enterprise-wide identity: • For enterprise users: the Oracle Internet Directory DN. • For external users: the external identity (Kerberos principal name, Radius schema names, OS user name, Certificate DN). • For local users and SYSDBA/SYSOPER logins: NULL. The value of the attribute differs by proxy method: • For a proxy with DN: the Oracle Internet Directory DN of the client • For a proxy with certificate: the certificate DN of the client for external users; the Oracle Internet Directory DN for global users • For a proxy with username: the Oracle Internet Directory DN if the client is an enterprise users; Null if the client is a local database user.
FG_JOB_ID	If queried from within a job that was created using the DBMS_JOB package: Returns the job ID of the current session if it was established by a client foreground process. Null if the session was not established by a foreground process. Otherwise: Returns 0.
GLOBAL_CONTEXT_MEMORY	Returns the number being used in the System Global Area by the globally accessed context.
GLOBAL_UID	Returns the global user ID (GUID) from Active Directory for Centrally Managed Users (CMU) logins, or from Oracle Internet Directory for Enterprise User Security (EUS) logins. Returns null for all other logins.
HOST	Name of the host machine from which the client has connected.

Table 7-11 (Cont.) Predefined Parameters of Namespace USERENV

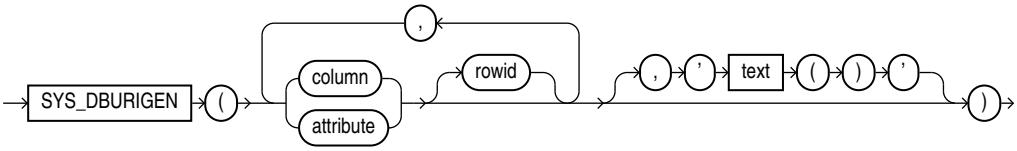
Parameter	Return Value
IDENTIFICATION_TYPE	Returns the way the user's schema was created in the database. Specifically, it reflects the IDENTIFIED clause in the CREATE/ALTER USER syntax. In the list that follows, the syntax used during schema creation is followed by the identification type returned: • IDENTIFIED BY *password*: LOCAL • IDENTIFIED EXTERNALLY: EXTERNAL • IDENTIFIED GLOBALLY: GLOBAL SHARED • IDENTIFIED GLOBALLY AS *DN*: GLOBAL PRIVATE • GLOBAL EXCLUSIVE for exclusive global user mapping. • GLOBAL SHARED for shared user mapping. • NONE when the schema is created with no authentication.
INSTANCE	The instance identification number of the current instance.
INSTANCE_NAME	The name of the instance.
IP_ADDRESS	IP address of the machine from which the client is connected. If the client and server are on the same machine and the connection uses IPv6 addressing, then ::1 is returned.
IS_APPLY_SERVER	Returns TRUE if queried from within a SQL Apply server in a logical standby database. Otherwise, returns FALSE.
IS_DG_ROLLING_UPGRADE	Returns TRUE if a rolling upgrade of the database software in a Data Guard configuration, initiated by way of the DBMS_ROLLING package, is active. Otherwise, returns FALSE.
ISDBA	Returns TRUE if the user has been authenticated as having DBA privileges either through the operating system or through a password file.
LANG	The abbreviated name for the language, a shorter form than the existing 'LANGUAGE' parameter.
LANGUAGE	The language and territory currently used by your session, along with the database character set, in this form: language_territory.characterset
LDAP_SERVER_TYPE	Returns the configured LDAP server type, one of OID, AD(Active Directory), OID_G, OPENLDAP.
MODULE	The application name (module) set through the DBMS_APPLICATION_INFO package or OCI.
NETWORK_PROTOCOL	Network protocol being used for communication, as specified in the 'PROTOCOL=*protocol*' portion of the connect string.
NLS_CALENDAR	The current calendar of the current session.
NLS_CURRENCY	The currency of the current session.
NLS_DATE_FORMAT	The date format for the session.
NLS_DATE_LANGUAGE	The language used for expressing dates.
NLS_SORT	BINARY or the linguistic sort basis.
NLS_TERRITORY	The territory of the current session.
ORACLE_HOME	The full path name for the Oracle home directory.
OS_USER	Operating system user name of the client process that initiated the database session.
PID	Oracle process ID.
PLATFORM_SLASH	The slash character that is used as the file path delimiter for your platform.

Table 7-11 (Cont.) Predefined Parameters of Namespace USERENV

Parameter	Return Value
POLICY_INVOKER	The invoker of row-level security (RLS) policy functions.
PROXY_ENTERPRISE_IDENTITY	Returns the Oracle Internet Directory DN when the proxy user is an enterprise user.
PROXY_USER	Name of the database user who opened the current session on behalf of SESSION_USER.
PROXY_USERID	Identifier of the database user who opened the current session on behalf of SESSION_USER.
RESET_STATE	RESET_STATE can be set using DBMS_APP_CONT_ADMIN.ENABLE_RESET_STATE() procedure call and is related to Application Continuity.
SCHEDULER_JOB	Returns Y if the current session belongs to a foreground job or background job. Otherwise, returns N.
SERVER_HOST	The host name of the machine on which the instance is running.
SERVICE_NAME	The name of the service to which a given session is connected.
SESSION_DEFAULT_COLLATION	The default collation for the session, which is set by the ALTER SESSION SET DEFAULT_COLLATION ... statement.
SESSION_EDITION_ID	The identifier of the session edition.
SESSION_EDITION_NAME	The name of the session edition.
SESSION_USER	The name of the session user (the user who logged on). This may change during the duration of a database session as Real Application Security sessions are attached or detached. If a Real Application Security session is currently attached to the database session, returns user XS$NULL.
SESSION_USERID	The identifier of the session user (the user who logged on).
SESSIONID	The auditing session identifier. You cannot use this attribute in distributed SQL statements.
SID	The session ID.
STANDBY_MAX_DATA_DELAY	The session parameter to specify allowed time limit to elapse between when changes are committed on primary database and when those changes can be queried on the standby database. If not set, returns null.
STATEMENTID	The auditing statement identifier. STATEMENTID represents the number of SQL statements audited in a given session. You cannot use this attribute in distributed SQL statements. The correct auditing statement identifier can be seen only through an audit handler for standard or fine-grained audit.
TERMINAL	The operating system identifier for the client of the current session. In distributed SQL statements, this attribute returns the identifier for your local session. In a distributed environment, this is supported only for remote SELECT statements, not for remote INSERT, UPDATE, or DELETE operations. (The return length of this parameter may vary by operating system.)
TLS_CIPHERSUITE	Used to retrieve the ciphersuite negotiated during the TLS. Valid ciphersuite values can be found in the TLS chapter of the Database Security Guide.
TLS_VERSION	Used to retrieve the TLS version negotiated during the TLS session.
UNIFIED_AUDIT_SESSIONID	If queried while connected to a database that uses unified auditing or mixed mode auditing, returns the unified audit session ID. If queried while connected to a database that uses traditional auditing, returns null.

SYS_DBURIGEN

Syntax



Purpose

`SYS_DBURIGen` takes as its argument one or more columns or attributes, and optionally a rowid, and generates a URL of data type `DBURIType` to a particular column or row object. You can then use the URL to retrieve an XML document from the database.

All columns or attributes referenced must reside in the same table. They must perform the function of a primary key. They need not actually match the primary key of the table, but they must reference a unique value. If you specify multiple columns, then all but the final column identify the row in the database, and the last column specified identifies the column within the row.

By default the URL points to a formatted XML document. If you want the URL to point only to the text of the document, then specify the optional '`text()`'.

> **Note:**
>
> In this XML context, the lowercase `text` is a keyword, not a syntactic placeholder.

If the table or view containing the columns or attributes does not have a schema specified in the context of the query, then Oracle Database interprets the table or view name as a public synonym.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for information on the `DBURIType` data type and XML documents in the database

Examples

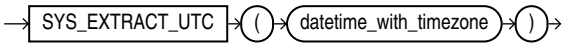
The following example uses the `SYS_DBURIGen` function to generate a URL of data type `DBURIType` to the `email` column of the row in the sample table `hr.employees` where the `employee_id` = 206:

```
SELECT SYS_DBURIGEN(employee_id, email)
   FROM employees
   WHERE employee_id = 206;

SYS_DBURIGEN(EMPLOYEE_ID,EMAIL)(URL, SPARE)
```

```
--------------------------------------------------------------------
DBURITYPE('/PUBLIC/EMPLOYEES/ROW[EMPLOYEE_ID=''206'']/EMAIL', NULL)
```

SYS_EXTRACT_UTC

Syntax



Purpose

`SYS_EXTRACT_UTC` extracts the UTC (Coordinated Universal Time—formerly Greenwich Mean Time) from a datetime value with time zone offset or time zone region name. If a time zone is not specified, then the datetime is associated with the session time zone.

Examples

The following example extracts the UTC from a specified datetime:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2000-03-28 11:30:00.00 -08:00')
   FROM DUAL;

SYS_EXTRACT_UTC(TIMESTAMP'2000-03-2811:30:00.00-08:00')
--------------------------------------------------------------------
28-MAR-00 07.30.00 PM
```

SYS_GUID

Syntax

Purpose

`SYS_GUID` generates and returns a globally unique identifier (`RAW` value) made up of 16 bytes. On most platforms, the generated identifier consists of a host identifier, a process or thread identifier of the process or thread invoking the function, and a nonrepeating value (sequence of bytes) for that process or thread.

Examples

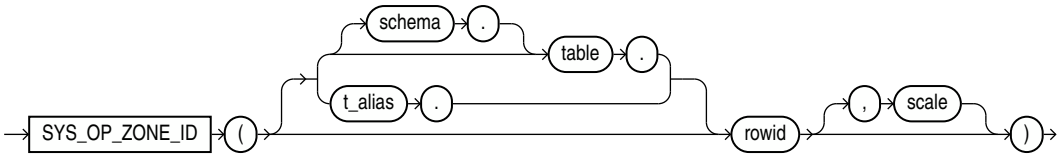
The following example adds a column to the sample table `hr.locations`, inserts unique identifiers into each row, and returns the 32-character hexadecimal representation of the 16-byte `RAW` value of the global unique identifier:

```
ALTER TABLE locations ADD (uid_col RAW(16));

UPDATE locations SET uid_col = SYS_GUID();

SELECT location_id, uid_col FROM locations
   ORDER BY location_id, uid_col;

LOCATION_ID UID_COL
```

```
----------- ------------------------------------------------------------
       1000 09F686761827CF8AE040578CB20B7491
       1100 09F686761828CF8AE040578CB20B7491
       1200 09F686761829CF8AE040578CB20B7491
       1300 09F68676182ACF8AE040578CB20B7491
       1400 09F68676182BCF8AE040578CB20B7491
       1500 09F68676182CCF8AE040578CB20B7491
. . .
```

SYS_OP_ZONE_ID

Syntax



Purpose

`SYS_OP_ZONE_ID` takes as its argument a rowid and returns a zone ID. The rowid identifies a row in a table. The zone ID identifies the set of contiguous disk blocks, called the zone, that contains the row. The function returns a `NUMBER` value.

The `SYS_OP_ZONE_ID` function is used when creating a zone map with the `CREATE MATERIALIZED ZONEMAP` statement. You must specify `SYS_OP_ZONE_ID` in the `SELECT` and `GROUP BY` clauses of the defining subquery of the zone map.

For *`rowid`*, specify the `ROWID` pseudocolumn of the fact table of the zone map.

Use *`schema`* and *`table`* to specify the schema and name of the fact table, or *`t_alias`* to specify the table alias for the fact table. The specification of these parameters depends on the `FROM` clause in the defining subquery of the zone map:

- If the `FROM` clause specifies a table alias for the fact table, then you must also specify the table alias (*`t_alias`*) in `SYS_OP_ZONE_ID`.
- If the `FROM` clause does not specify a table alias for the fact table, then use *`table`* to specify the name of the fact table. You can use the *`schema`* qualifier if the fact table is in a schema other than your own. If you omit *`schema`*, then the database assumes the fact table is in your own schema. If the `FROM` clause specifies only one table (the fact table) then you need not specify *`schema`* or *`table`*.

The optional *`scale`* parameter represents the scale of the zone map. It is not necessary to specify this parameter because, by default, `SYS_OP_ZONE_ID` uses the scale of the zone map being created. If you do specify *`scale`*, then it must match the scale of the zone map being created. Refer to the SCALE clause of `CREATE MATERIALIZED ZONEMAP` for information on specifying the scale of a zone map.

> **See Also:**
>
> CREATE MATERIALIZED ZONEMAP for more information on creating zone maps

Examples

The following example uses the `SYS_OP_ZONE_ID` function when creating a basic zone map that tracks the column `time_id` of the fact table `sales`. The scale of the zone map is the default value of 10. Therefore, the `SYS_OP_ZONE_ID` function will default to a scale value of 10.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
AS
  SELECT SYS_OP_ZONE_ID(rowid), MIN(time_id), MAX(time_id)
  FROM sales
  GROUP BY SYS_OP_ZONE_ID(rowid);
```

The following example is similar to the previous example, except that the scale of the zone map being created is specified as 8. Therefore, the `SYS_OP_ZONE_ID` function will default to a scale value of 8.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
SCALE 8
AS
  SELECT SYS_OP_ZONE_ID(rowid), MIN(time_id), MAX(time_id)
  FROM sales
  GROUP BY SYS_OP_ZONE_ID(rowid);
```

The following example returns an error because the scale of the zone map being created is specified as 8, which does not match the *`scale`* argument of 12 specified in the `SYS_OP_ZONE_ID` function.

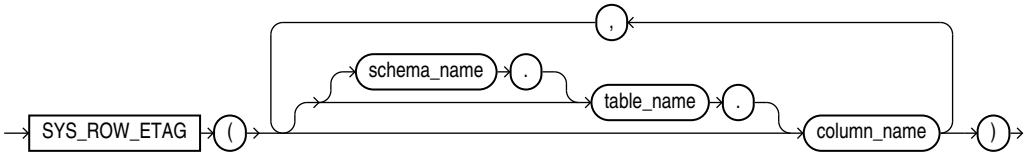
```
CREATE MATERIALIZED ZONEMAP sales_zmap
SCALE 8
AS
  SELECT SYS_OP_ZONE_ID(rowid,12), MIN(time_id), MAX(time_id)
  FROM sales
  GROUP BY SYS_OP_ZONE_ID(rowid,12);
```

The following example creates a join zone map. The fact table is `sales` and the dimension tables are `products` and `customers`. Because the table alias `s` is specified for the fact table in the `FROM` clause, the table alias *`s`* is also specified in the `SYS_OP_ZONE_ID` function.

```
CREATE MATERIALIZED ZONEMAP sales_zmap
AS
  SELECT SYS_OP_ZONE_ID(s.rowid),
         MIN(prod_category), MAX(prod_category),
         MIN(country_id), MAX(country_id)
  FROM sales s, products p, customers c
  WHERE s.prod_id = p.prod_id(+) AND
        s.cust_id = c.cust_id(+)
  GROUP BY SYS_OP_ZONE_ID(s.rowid);
```

SYS_ROW_ETAG

Syntax



Purpose

You can use ETAGs with table data, for lock-free row updates using SQL. To do that, use function SYS_ROW_ETAG, to obtain the current state of a given set of columns in a table row as an ETAG hash value. Function SYS_ROW_ETAG calculates an etag (128 bits hash value) for a row using the values of a set of columns in the row that you want the etag to be computed on. You can pass the function the names of the columns in any order.

Function SYS_ROW_ETAG calculates the ETAG value for a row using only the values of those columns in the row: you pass it the names of all columns that you want to be sure no other session tries to update concurrently. This includes the columns that the current session intends to update, but also any other columns on whose value that updating operation logically depends for your application. (The order in which you pass the columns to SYS_ROW_ETAG as arguments is irrelevant.)

Example

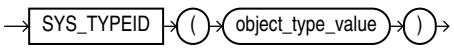
The example below creates table foo with columns c1, c2, and c3 of type NUMBER, and inserts values into the table. It then passes columns c2 and c1 to SYS_ROW_ETAG to get the etag for c2 and c1:

```
CREATE TABLE foo (c1 NUMBER, c2 NUMBER, c3 NUMBER);

Table created.

INSERT INTO foo VALUES (1, 2, 3);

1 row created.

SELECT SYS_ROW_ETAG(c2, c1) FROM foo;

SYS_ROW_ETAG(C2,C1)
--------------------------------
3B978191AD0C828DA0E6A53EDF0B278A

---------------
```

> **See Also:**
>
> Example 4.18 in the JSON-Relational Duality Developer's Guide *Using Function SYS_ROW_ETAG To Optimistically Control Concurrent Table Updates*

SYS_TYPEID

Syntax



Purpose

SYS_TYPEID returns the typeid of the most specific type of the operand. This value is used primarily to identify the type-discriminant column underlying a substitutable column. For

example, you can use the value returned by `SYS_TYPEID` to build an index on the type-discriminant column.

You can use this function only on object type operands. All final root object types—final types not belonging to a type hierarchy—have a null typeid. Oracle Database assigns to all types belonging to a type hierarchy a unique non-null typeid.

> **See Also:**
>
> *Oracle Database Object-Relational Developer's Guide* for more information on typeids

Examples

The following examples use the tables `persons` and `books`, which are created in "Substitutable Table and Column Examples". The first query returns the most specific types of the object instances stored in the `persons` table.

```
SELECT name, SYS_TYPEID(VALUE(p)) "Type_id" FROM persons p;

NAME                      Type_id
------------------------- --------------------------------
Bob                       01
Joe                       02
Tim                       03
```

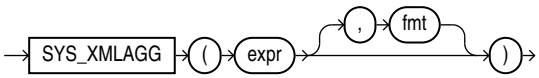
The next query returns the most specific types of authors stored in the table `books`:

```
SELECT b.title, b.author.name, SYS_TYPEID(author)
   "Type_ID" FROM books b;

TITLE                     AUTHOR.NAME          Type_ID
------------------------- -------------------- --------------------
An Autobiography          Bob                  01
Business Rules            Joe                  02
Mixing School and Work    Tim                  03
```

You can use the `SYS_TYPEID` function to create an index on the type-discriminant column of a table. For an example, see "Indexing on Substitutable Columns: Examples".

SYS_XMLAGG

Syntax



Purpose

`SYS_XMLAgg` aggregates all of the XML documents or fragments represented by *`expr`* and produces a single XML document. It adds a new enclosing element with a default name `ROWSET`. If you want to format the XML document differently, then specify *`fmt`*, which is an instance of the `XMLFormat` object.

> **See Also:**
>
> SYS_XMLGEN and "XML Format Model " for using the attributes of the `XMLFormat` type to format `SYS_XMLAgg` results

Examples

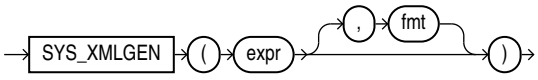
The following example uses the `SYS_XMLGen` function to generate an XML document for each row of the sample table `employees` where the employee's last name begins with the letter R, and then aggregates all of the rows into a single XML document in the default enclosing element `ROWSET`:

```
SELECT SYS_XMLAGG(SYS_XMLGEN(last_name)) XMLAGG
   FROM employees
   WHERE last_name LIKE 'R%'
   ORDER BY xmlagg;

XMLAGG
--------------------------------------------------------------------------------
<?xml version="1.0"?>
<ROWSET>
<LAST_NAME>Rajs</LAST_NAME>
<LAST_NAME>Raphaely</LAST_NAME>
<LAST_NAME>Rogers</LAST_NAME>
<LAST_NAME>Russell</LAST_NAME>
</ROWSET>
```

SYS_XMLGEN

> **Note:**
>
> The `SYS_XMLGen` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use the SQL/XML generation functions instead. See *Oracle XML DB Developer's Guide* for more information.

Syntax



Purpose

`SYS_XMLGen` takes an expression that evaluates to a particular row and column of the database, and returns an instance of type `XMLType` containing an XML document. The *`expr`* can be a scalar value, a user-defined type, or an `XMLType` instance.

- If *`expr`* is a scalar value, then the function returns an XML element containing the scalar value.
- If *`expr`* is a type, then the function maps the user-defined type attributes to XML elements.

- If *expr* is an XMLType instance, then the function encloses the document in an XML element whose default tag name is ROW.

By default the elements of the XML document match the elements of *expr*. For example, if *expr* resolves to a column name, then the enclosing XML element will be the same column name. If you want to format the XML document differently, then specify *fmt*, which is an instance of the XMLFormat object.

> **See Also:**
>
> "XML Format Model " for a description of the XMLFormat type and how to use its attributes to format SYS_XMLGen results

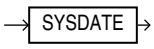
Examples

The following example retrieves the employee email ID from the sample table oe.employees where the employee_id value is 205, and generates an instance of an XMLType containing an XML document with an EMAIL element.

```
SELECT SYS_XMLGEN(email)
   FROM employees
   WHERE employee_id = 205;

SYS_XMLGEN(EMAIL)
--------------------------------------------------------------------
<?xml version="1.0"?>
<EMAIL>SHIGGINS</EMAIL>
```

SYSDATE

Syntax



Purpose

SYSDATE returns the current date and time set for the operating system on which the database server resides. The data type of the returned value is DATE, and the format returned depends on the value of the NLS_DATE_FORMAT initialization parameter. The function requires no arguments. In distributed SQL statements, this function returns the date and time set for the operating system of your local database. You cannot use this function in the condition of a CHECK constraint.

In a multitenant setup existing PDBs and PDBs created later inherit the timezone of the system.

If you want SYSDATE to return the timezone of the PDB, then you must set the initialization parameter TIME_AT_DBTIMEZONE to TRUE before starting the PDB.

You can change the timezone using ALTER SYSTEM SET TIME_ZONE or ALTER DATABASE db_name SET TIME_ZONE.

You can set SYSTIMESTAMP to return system time by setting the initialization parameter TIME_AT_DBTIMEZONE to FALSE and restarting the database.

> **Note:**
>
> - For more see TIME_AT_DBTIMEZONE of the Oracle Database Reference.
> - The FIXED_DATE initialization parameter enables you to set a constant date and time that SYSDATE will always return instead of the current date and time. This parameter is useful primarily for testing. Refer to *Oracle Database Reference* for more information on the FIXED_DATE initialization parameter.

Examples

The following example returns the current operating system date and time:

```
SELECT TO_CHAR
    (SYSDATE, 'MM-DD-YYYY HH24:MI:SS') "NOW"
     FROM DUAL;

NOW
-------------------
04-13-2001 09:45:51
```

SYSTIMESTAMP

Syntax

Purpose

SYSTIMESTAMP returns the system date, including fractional seconds and time zone, of the system on which the database resides. The return type is TIMESTAMP WITH TIME ZONE.

In a multitenant setup existing PDBs and PDBs created later inherit the timezone of the system.

If you want SYSTIMESTAMP to return the timezone of the PDB, then you must set the initialization parameter TIME_AT_DBTIMEZONE to TRUE before starting the PDB.

You can change the timezone using ALTER SYSTEM SET TIME_ZONE or ALTER DATABASE db_name SET TIME_ZONE.

You can set SYSTIMESTAMP to return system time by setting the initialization parameter TIME_AT_DBTIMEZONE to FALSE and restarting the database.

> **Note:**
>
> For more see TIME_AT_DBTIMEZONE of the Oracle Database Reference.

Examples

The following example returns the system timestamp:

```
SELECT SYSTIMESTAMP FROM DUAL;

SYSTIMESTAMP
---------------------------------------------------------------------
28-MAR-00 12.38.55.538741 PM -08:00
```

The following example shows how to explicitly specify fractional seconds:

```
SELECT TO_CHAR(SYSTIMESTAMP, 'SSSSS.FF') FROM DUAL;

TO_CHAR(SYSTIME
---------------
55615.449255
```

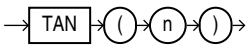
The following example returns the current timestamp in a specified time zone:

```
SELECT SYSTIMESTAMP AT TIME ZONE 'UTC' FROM DUAL;

SYSTIMESTAMPATTIMEZONE'UTC'
---------------------------------------------------------------------------
08-07-21 20:39:52,743557 UTC
```

The output format in this example depends on the `NLS_TIMESTAMP_TZ_FORMAT` for the session.

TAN

Syntax



Purpose

`TAN` returns the tangent of *n* (an angle expressed in radians).

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

Examples

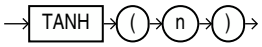
The following example returns the tangent of 135 degrees:

```
SELECT TAN(135 * 3.14159265359/180)
   "Tangent of 135 degrees"  FROM DUAL;
```

```
Tangent of 135 degrees
----------------------
                    - 1
```

TANH

Syntax



Purpose

`TANH` returns the hyperbolic tangent of *n*.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If the argument is `BINARY_FLOAT`, then the function returns `BINARY_DOUBLE`. Otherwise the function returns the same numeric data type as the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

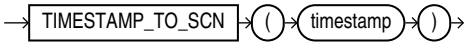
Examples

The following example returns the hyperbolic tangent of .5:

```
SELECT TANH(.5) "Hyperbolic tangent of .5"
   FROM DUAL;

Hyperbolic tangent of .5
------------------------
               .462117157
```

TIMESTAMP_TO_SCN

Syntax



Purpose

`TIMESTAMP_TO_SCN` takes as an argument a timestamp value and returns the approximate system change number (SCN) associated with that timestamp. The returned value is of data type `NUMBER`. This function is useful any time you want to know the SCN associated with a particular timestamp.

> **Note:**
>
> The association between an SCN and a timestamp when the SCN is generated is remembered by the database for a limited period of time. This period is the maximum of the auto-tuned undo retention period, if the database runs in the Automatic Undo Management mode, and the retention times of all flashback archives in the database, but no less than 120 hours. The time for the association to become obsolete elapses only when the database is open. An error is returned if the timestamp specified for the argument to `TIMESTAMP_TO_SCN` is too old.

> **See Also:**
>
> SCN_TO_TIMESTAMP for information on converting SCNs to timestamp

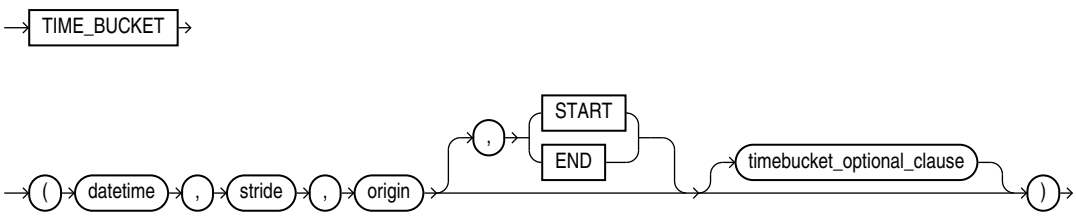
Examples

The following example inserts a row into the `oe.orders` table and then uses `TIMESTAMP_TO_SCN` to determine the system change number of the insert operation. (The actual SCN returned will differ on each system.)

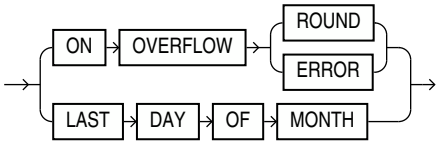
```
INSERT INTO orders (order_id, order_date, customer_id, order_total)
   VALUES (5000, SYSTIMESTAMP, 188, 2345);
1 row created.

COMMIT;
Commit complete.

SELECT TIMESTAMP_TO_SCN(order_date) FROM orders
   WHERE order_id = 5000;

TIMESTAMP_TO_SCN(ORDER_DATE)
----------------------------
                      574100
```

TIME_BUCKET (datetime)

Syntax



***timebucket_optional_clause*::=**



Purpose

Use `TIME_BUCKET`(datetime) to obtain the datetime over an interval that you specify.

`TIME_BUCKET` has three required arguments, and two optional arguments .

- The first argument *`datetime`* is the input to the bucket.

  The third argument *`origin`* is an anchor to which all buckets are aligned.

  `datetime` and *`origin`* can be `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIMEZONE`, `TIMESTAMP WITH LOCAL TIMEZONE`, `EPOCH TIME`, `BINARY_FLOAT`, `BINARY_DOUBLE`, `CHAR`, expression, or a bind variable.

  `EPOCH TIME` is represented by Oracle type `NUMBER`, which is the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970. The supported `EPOCH TIME` range is from `SB8MINVAL`(`- 9223372036854775808`, inclusive) to `SB8MAXVAL`(`9223372036854775807`, inclusive).

  There are implicit conversions for *`datetime`* and *`origin`*:

  – If it is `BINARY_FLOAT` or `BINARY_DOUBLE`, it will be converted to `NUMBER` implicitly. Note that you must account for the loss in precision from implicit conversions.

  – If it is `CHAR`, it will be converted to `TIMESTAMP` implicitly. Note that `CHAR` should match the session `NLS_TIMESTAMP_FORMAT`. Otherwise an error is raised.

  Fractional second is supported only if *`datetime`* and *`origin`* are `EPOCH TIME`, `BINARY_FLOAT` or `BINARY_DOUBLE`.

  The valid range for *`datetime`* and *`origin`* is from `-4712-01-01 00:00:00` inclusive to `9999-12-31 23:59:59:00` inclusive.

- The second argument *`stride`* is a positive Oracle `INTERVAL`, ISO 8601 time interval string, expression or bind variable. Fractional second is supported only if *`datetime`* and *`origin`* are `EPOCH TIME`, `BINARY_FLOAT` or `BINARY_DOUBLE`.

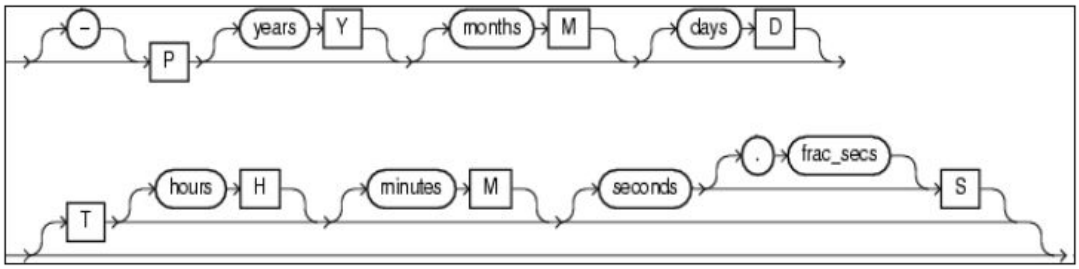
  Oracle `INTERVAL` has two types of valid intervals: `INTERVAL YEAR TO MONTH` and `INTERVAL DAY TO SECOND`. If year or month is specified, all other units are ignored, if specified.

  For the ISO 8601 time interval string, years, months, days, hours, minutes and seconds are integers between `0` and `999999999`. `frac_secs` is the fractional part of seconds between `.0` and `.999999999`.

  The ISO 8601 time interval string that you specify should match the definition of Oracle `INTERVAL`. `P` is required, and no blanks are allowed in the value. If you specify `T`, then you must specify at least one of `hours`, `minutes`, or `seconds`. `hours` are based on 24-hour time.

  For example, `P100DT05H` indicates 100 days and 5 hours. `P1Y2M`'indicates 1 year and 2 months. `P1M1DT5H30M30S` is equivalent to `P1M` which indicates 1 month.

  The syntax of the ISO 8601 time interval string:



Use only postive values for *stride*. (Although the Oracle INTERVAL and ISO 8601 time interval string can be positive or negative.)

If *datetime* or *origin* is EPOCH TIME, BINARY_FLOAT or BINARY_DOUBLE, then *stride* cannot contain YEAR or MONTH. This is because month is variable and could be one of 28, 29, 30 or 31.

- The fourth argument is optional and specifies whether the start or the end of the time bucket is returned. Specify START to return the start value of the time bucket or END to return the end value . The values are case-insensitive. The default value is START.
- The fifth argument is optional and controls how the buckets (strides) are determined.

  ON OVERFLOW ROUND (default): The buckets will be cut on the same day as *origin* in the corresponding month. For a month that does not have that day, the bucket is rounded to the last day of the month.

  ON OVERFLOW ERROR: The buckets will be cut on the same day as *origin* in the corresponding month. For a month that does not have that day will error out.

  LAST DAY OF MONTH: If *origin* is the last day of the month and *stride* only contains MONTH and/or YEAR , the buckets will be cut on the corresponding last day of the month.

  For example, if *origin* is '1991-11-30' and *stride* is 'P1M', then:

  – For ON OVERFLOW ROUND, the start of each bucket will be:

    ```
    ..., 1991-11-30, 1991-12-30, 1992-01-30, 1992-02-29, 1992-03-30,
    1992-04-30,...
    ```

  – For ON OVERFLOW ERROR, the start of each bucket will be:

    ```
    ..., 1991-11-30, 1991-12-30, 1992-01-30, error (or 1992-02-30),
    1992-03-30, 1992-04-30,...
    ```

  – For LAST DAY OF MONTH, the start of each bucket will be:

    ```
    ..., 1991-11-30, 1991-12-31, 1992-01-31, 1992-02-29, 1992-03-31,
    1992-04-30, ...
    ```

Rules

- The end of each bucket is the same as the beginning of the following bucket. For example, if the bucket is 2 years and the start of the slice is 2000-01-01, then the end of the bucket will be 2002-01- 01, not 2001-12-31. In other words, the bucket contains *datetime* greater than or equal to the start and less than (but not equal to) the end.
- In general, START of a bucket is always less than END of the bucket. But for the bucket on the two sides of the valid time range, START can be equal to END.

- *origin* and *datetime* can be positive or negative as long as it is in the valid range. Errors are raised if *origin* or *datetime* is outside of the valid range, or if the return value is outside of the valid range.
- If the input value is of type TIMESTAMP WITH TIME ZONE or TIMESTAMP WITH LOCAL TIME ZONE, then a time bucket might cross the daylight saving time boundaries. In this case, the duration of the time bucket is still the same as any other time bucket.
- If *origin* and *datetime* are TIMESTAMP WITH TIME ZONE or TIMESTAMP WITH LOCAL TIME ZONE, all arithmetic calculations are based in UTC time.

Examples

The following examples use the NLS_DATE_FORMAT YYYY-MM-DD. Set the date format with ALTER SESSION:

```
ALTER SESSION SET NLS_DATE_FORMAT='YYYY-MM-DD';
```

Example 1

```
SELECT TIME_BUCKET (DATE '2022-06-29', INTERVAL '5' YEAR, DATE '2000-01-01', START);
```

The result is:

```
2020-01-01
```

The 5-year time bucket that contains 2022-06-29 is from 2020-01-01(start) to 2025-01-01(end). The fourth argument START is used, so the start of the time bucket 2020-01-01 is returned.

Example 2

The following two queries are equivalent:

```
SELECT TIME_BUCKET ( DATE '-2022-06-29', 'P5M', DATE '-2022-01-01', END );
```

Or:

```
SELECT TIME_BUCKET ( DATE '-2022-06-29', INTERVAL '5' MONTH, DATE '-2022-01-01', END);
```

The result is:

```
-2022-11-01
```

The 5-month time bucket that contains -2022-06-29 is from 2022-06-01(start) to -2022-11-01(end). The fourth argument END is used, so the end of the time bucket 2022-11-01 is returned.

Example 3

```
SELECT TIME_BUCKET ( DATE '2005-03-10', 'P1Y', DATE '2004-02-29' ON OVERFLOW ERROR );
```

The result is:

```
 ORA-01839: date not valid for month specified
```

The one-year time bucket that contains `'2005-03-10'` is from error (or `'2005-02-29'`) (start) to error (or `'2006-02-29'`) (end). Default fourth argument `START` is used, so the start of the time bucket should be returned which is an error.

Example 4

```
SELECT TIME_BUCKET ( DATE '2005-03-10', 'P1Y', DATE '2004-02-29' ON OVERFLOW ROUND );
```

The result is:

```
2005-02-28
```

The one-year time bucket that contains `'2005-03-10'` is from `'2005-02-28'` (start) to `'2006-02-28'` (end) since February 29 is rounded to February 28. Default fourth argument `START` is used, so the start of the time bucket is returned: `'2005-02-28'`.

Example 5

```
SELECT TIME_BUCKET ( DATE '2004-04-02', 'P1Y', DATE '2003-02-28' LAST DAY OF MONTH );
```

The result is:

```
2004-02-29
```

The one-year time bucket that contains `'2003-02-28'` is from `'2004-02-29'` (start) to `'2005-02-28'` (end) since `'2004-02-28'` is rounded to the last day of that month which is `'2004-02-29'`. Default fourth argument `START` is used, so the start of the time bucket is returned: `'2004-02- 29'`.

TO_APPROX_COUNT_DISTINCT

Syntax

Purpose

`TO_APPROX_COUNT_DISTINCT` takes as its input a detail containing information about an approximate distinct value count, and converts it to a `NUMBER` value.

For *`detail`*, specify a detail of type `BLOB`, which was created by the `APPROX_COUNT_DISTINCT_DETAIL` function or the `APPROX_COUNT_DISTINCT_AGG` function.

See Also:

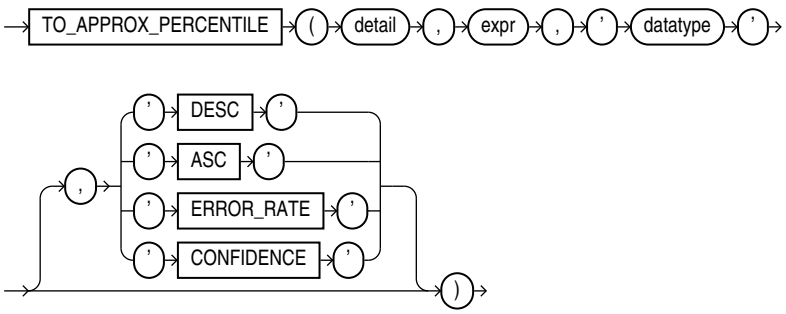
- APPROX_COUNT_DISTINCT_DETAIL
- TO_APPROX_COUNT_DISTINCT

Examples

Refer to TO_APPROX_COUNT_DISTINCT: Examples for examples of using the `TO_APPROX_COUNT_DISTINCT` function in conjunction with the `APPROX_COUNT_DISTINCT_DETAIL` and `APPROX_COUNT_DISTINCT_AGG` functions.

TO_APPROX_PERCENTILE

Syntax



(*datatype*::=)

Purpose

`TO_APPROX_PERCENTILE` takes as its input a detail containing approximate percentile information, a percentile value, and a sort specification, and returns an approximate interpolated value that would fall into that percentile value with respect to the sort specification.

For *`detail`*, specify a detail of type `BLOB`, which was created by the `APPROX_PERCENTILE_DETAIL` function or the `APPROX_PERCENTLE_AGG` function.

For *`expr`*, specify a percentile value, which must evaluate to a numeric value between 0 and 1. If you specify the `ERROR_RATE` or `CONFIDENCE` clause, then the percentile value does not apply. In this case, for *`expr`* you must specify null or a numeric value between 0 and 1. However, the value will be ignored.

For *`datatype`*, specify the data type of the approximate percentile information in the detail. This is the data type of the expression supplied to the `APPROX_PERCENTILE_DETAIL` function that originated the detail. Valid data types are `NUMBER`, `BINARY_FLOAT`, `BINARY_DOUBLE`, `DATE`, `TIMESTAMP`, `INTERVAL YEAR TO MONTH`, and `INTERVAL DAY TO SECOND`.

DESC | ASC

Specify the sort specification for the interpolation. Specify `DESC` for a descending sort order, or `ASC` for an ascending sort order. `ASC` is the default.

ERROR_RATE | CONFIDENCE

These clauses let you determine the accuracy of the percentile evaluation of the detail. If you specify one of these clauses, then instead of returning the approximate interpolated value, the function returns a decimal value from 0 to 1, inclusive, which represents one of the following values:

- If you specify ERROR_RATE, then the return value represents the error rate of the percentile evaluation for the detail.
- If you specify CONFIDENCE, then the return value represents the confidence level for the error rate returned when you specify ERROR_RATE.

If you specify ERROR_RATE or CONFIDENCE, then the percentile value *expr* is ignored.

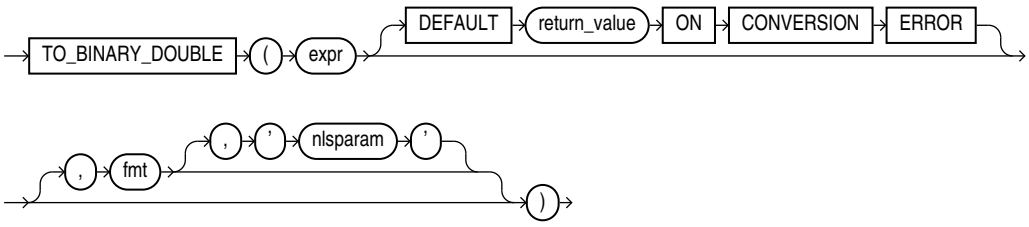
> **See Also:**
>
> - APPROX_PERCENTILE_DETAIL
> - APPROX_PERCENTILE_AGG

Examples

Refer to APPROX_PERCENTILE_AGG: Examples for examples of using the TO_APPROX_PERCENTILE function in conjunction with the APPROX_PERCENTILE_DETAIL and APPROX_PERCENTILE_AGG functions.

TO_BINARY_DOUBLE

Syntax



Purpose

TO_BINARY_DOUBLE converts *expr* to a double-precision floating-point number.

- *expr* can be any expression that evaluates to a character string of type CHAR, VARCHAR2, NCHAR, or NVARCHAR2, a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE,BOOLEAN, or null. If *expr* is BINARY_DOUBLE, then the function returns *expr*. If *expr* evaluates to null, then the function returns null. Otherwise, the function converts *expr* to a BINARY_DOUBLE value.
- The optional DEFAULT *return_value* ON CONVERSION ERROR clause allows you to specify the value returned by this function if an error occurs while converting *expr* to BINARY_DOUBLE. This clause has no effect if an error occurs while evaluating *expr*. The *return_value* can be an expression or a bind variable, and must evaluate to a character string of type CHAR, VARCHAR2, NCHAR, or NVARCHAR2, a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE, or null. The function converts *return_value* to BINARY_DOUBLE in the same way it converts *expr* to BINARY_DOUBLE. If *return_value* cannot be converted to BINARY_DOUBLE, then the function returns an error.
- The optional '*fmt*' and '*nlsparam*' arguments serve the same purpose as for the TO_NUMBER function. If you specify these arguments, then *expr* and *return_value*, if specified, must

each be a character string or null. If either is a character string, then the function uses the *fmt* and *nlsparam* arguments to convert the character string to a BINARY_DOUBLE value.

If *expr* or *return_value* evaluate to the following character strings, then the function converts them as follows:

- The case-insensitive string 'INF' is converted to positive infinity.
- The case-insensitive string '-INF' is converted to negative identity.
- The case-insensitive string 'NaN' is converted to NaN (not a number).

You cannot use a floating-point number format element (F, f, D, or d) in a character string *expr*.

Conversions from character strings or NUMBER to BINARY_DOUBLE can be inexact, because the NUMBER and character types use decimal precision to represent the numeric value, and BINARY_DOUBLE uses binary precision.

Conversions from BINARY_FLOAT to BINARY_DOUBLE are exact.

If you specify an *expr* of type BOOLEAN, then TRUE will be converted to 1 and FALSE will be converted to 0.

> **See Also:**
>
> TO_CHAR (number) and "Floating-Point Numbers "

Examples

The examples that follow are based on a table with three columns, each with a different numeric data type:

```
CREATE TABLE float_point_demo
  (dec_num NUMBER(10,2), bin_double BINARY_DOUBLE, bin_float BINARY_FLOAT);

INSERT INTO float_point_demo
  VALUES (1234.56,1234.56,1234.56);

SELECT * FROM float_point_demo;

   DEC_NUM BIN_DOUBLE  BIN_FLOAT
---------- ---------- ----------
   1234.56 1.235E+003 1.235E+003
```

The following example converts a value of data type NUMBER to a value of data type BINARY_DOUBLE:

```
SELECT dec_num, TO_BINARY_DOUBLE(dec_num)
  FROM float_point_demo;

   DEC_NUM TO_BINARY_DOUBLE(DEC_NUM)
---------- -------------------------
   1234.56                1.235E+003
```

The following example compares extracted dump information from the dec_num and bin_double columns:

```
SELECT DUMP(dec_num) "Decimal",
   DUMP(bin_double) "Double"
```

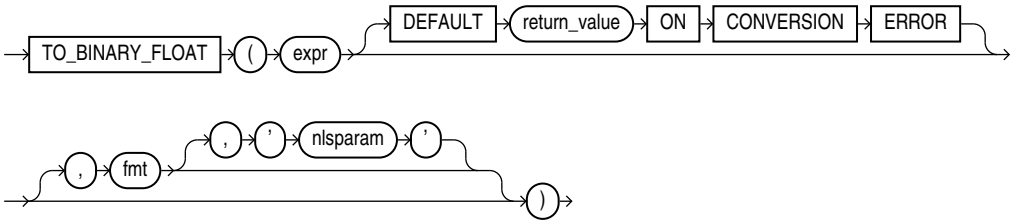
```
   FROM float_point_demo;

Decimal                     Double
--------------------------- ----------------------------------------------
Typ=2 Len=4: 194,13,35,57   Typ=101 Len=8: 192,147,74,61,112,163,215,10
```

The following example returns the default value of `0` because the specified expression cannot be converted to a `BINARY_DOUBLE` value:

```
SELECT TO_BINARY_DOUBLE('2oo' DEFAULT 0 ON CONVERSION ERROR) "Value"
  FROM DUAL;

     Value
----------
         0
```

TO_BINARY_FLOAT

Syntax



Purpose

`TO_BINARY_FLOAT` converts *`expr`* to a single-precision floating-point number.

- *`expr`* can be any expression that evaluates to a character string of type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`, a numeric value of type `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`,`BOOLEAN`, or null. If *`expr`* is `BINARY_FLOAT`, then the function returns *`expr`*. If *`expr`* evaluates to null, then the function returns null. Otherwise, the function converts *`expr`* to a `BINARY_FLOAT` value.
- The optional `DEFAULT` *`return_value`* `ON CONVERSION ERROR` clause allows you to specify the value returned by this function if an error occurs while converting *`expr`* to `BINARY_FLOAT`. This clause has no effect if an error occurs while evaluating *`expr`*. The *`return_value`* can be an expression or a bind variable, and must evaluate to a character string of type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`, a numeric value of type `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`, or null. The function converts *`return_value`* to `BINARY_FLOAT` in the same way it converts *`expr`* to `BINARY_FLOAT`. If *`return_value`* cannot be converted to `BINARY_FLOAT`, then the function returns an error.
- The optional '*`fmt`*' and '*`nlsparam`*' arguments serve the same purpose as for the `TO_NUMBER` function. If you specify these arguments, then *`expr`* and *`return_value`*, if specified, must each be a character string or null. If either is a character string, then the function uses the *`fmt`* and *`nlsparam`* arguments to convert the character string to a `BINARY_FLOAT` value.

If *`expr`* or *`return_value`* evaluate to the following character strings, then the function converts them as follows:

- The case-insensitive string '`INF`' is converted to positive infinity.

- The case-insensitive string '-INF' is converted to negative identity.
- The case-insensitive string 'NaN' is converted to NaN (not a number).

You cannot use a floating-point number format element (F, f, D, or d) in a character string *expr*.

Conversions from character strings or NUMBER to BINARY_FLOAT can be inexact, because the NUMBER and character types use decimal precision to represent the numeric value and BINARY_FLOAT uses binary precision.

Conversions from BINARY_DOUBLE to BINARY_FLOAT are inexact if the BINARY_DOUBLE value uses more bits of precision than supported by the BINARY_FLOAT.

If you specify an *expr* of type BOOLEAN, then TRUE will be converted to 1 and FALSE will be converted to 0.

> **See Also:**
>
> TO_CHAR (number) and "Floating-Point Numbers "

Examples

Using table float_point_demo created for TO_BINARY_DOUBLE , the following example converts a value of data type NUMBER to a value of data type BINARY_FLOAT:

```
SELECT dec_num, TO_BINARY_FLOAT(dec_num)
  FROM float_point_demo;

   DEC_NUM TO_BINARY_FLOAT(DEC_NUM)
---------- ------------------------
   1234.56               1.235E+003
```

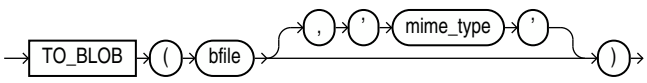
The following example returns the default value of 0 because the specified expression cannot be converted to a BINARY_FLOAT value:

```
SELECT TO_BINARY_FLOAT('2oo' DEFAULT 0 ON CONVERSION ERROR) "Value"
  FROM DUAL;

     Value
----------
         0
```

TO_BLOB (bfile)

Syntax

***to_blob_bfile*::=**



Purpose

TO_BLOB (bfile) converts a BFILE value to a BLOB value.

For *mime_type*, specify the MIME type to be set on the BLOB value returned by this function. If you omit *mime_type*, then a MIME type will not be set on the BLOB value.

Example

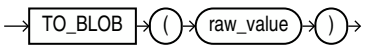
The following hypothetical example returns the BLOB of a BFILE column value media_col in table media_tab. It sets the MIME type to JPEG on the resulting BLOB.

```
SELECT TO_BLOB(media_col, 'JPEG') FROM media_tab;
```

TO_BLOB (raw)

Syntax

***to_blob*::=**



Purpose

> **Note:**
>
> All forms of LONG data types (LONG, LONG RAW, LONG VARCHAR, LONG VARRAW) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the LONG data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use CLOB and NCLOB data types for large amounts of character data.

TO_BLOB (raw) converts LONG RAW and RAW values to BLOB values.

From within a PL/SQL package, you can use TO_BLOB (raw) to convert RAW and BLOB values to BLOB.

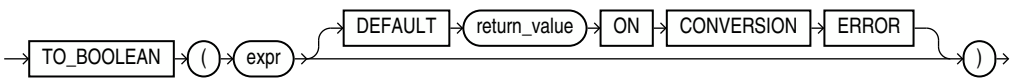
Examples

The following hypothetical example returns the BLOB of a RAW column value:

```
SELECT TO_BLOB(raw_column) blob FROM raw_table;

BLOB
----------------------
00AADD343CDBBD
```

TO_BOOLEAN

Syntax



Purpose

Use `TO_BOOLEAN` to explicitly convert character value expressions or numeric value expressions to boolean values.

If *expr* is a string, it must evaluate to the allowed string inputs. See Table 2-6.

expr can take one of the following types, or null:

- A character string of type `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`
- A numeric value of type `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`
- A boolean value of type `BOOLEAN`.

Examples

```
SELECT TO_BOOLEAN(0), TO_BOOLEAN('true'), TO_BOOLEAN('no');
```

The output is:

```
TO_BOOLEAN( TO_BOOLEAN( TO_BOOLEAN(
----------- ----------- -----------
FALSE       TRUE        FALSE

SELECT TO_BOOLEAN(1) FROM DUAL;
```

The output is:

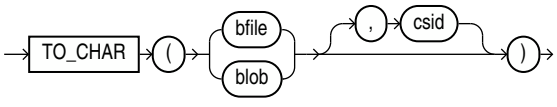
```
TO_BOOLEAN(
-----------
TRUE
```

> **See Also:**
>
> - CAST for conversion rules.
> - Boolean Data Type for more details on the built-in boolean data type.

TO_CHAR (bfile|blob)

Syntax

***to_char_bfile_blob*::=**



Purpose

`TO_CHAR` (bfile|blob) converts `BFILE` or `BLOB` data to the database character set. The value returned is always `VARCHAR2`. If the value returned is too large to fit into the `VARCHAR2` data type, then the data is truncated.

For *`csid`*, specify the character set ID of the `BFILE` or `BLOB` data. If the character set of the `BFILE` or `BLOB` data is the database character set, then you can specify a value of 0 for *`csid`*, or omit *`csid`* altogether.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

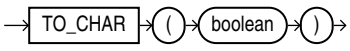
Example

The following hypothetical example takes as its input a `BFILE` column `media_col` in table `media_tab`, which uses the character set with ID 873. The example returns a `VARCHAR2` value that uses the database character set.

```
SELECT TO_CHAR(media_col, 873) FROM media_tab;
```

TO_CHAR (boolean)

Syntax



Purpose

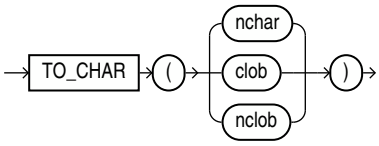
Use `TO_CHAR(boolean)` to explicitly convert a boolean value to a character value of `'TRUE'` or `'FALSE'`.

> **See Also:**
>
> - CAST for conversion rules.
> - Boolean Data Type for more details on the built-in boolean data type.

TO_CHAR (character)

Syntax

***to_char_char*::=**



Purpose

`TO_CHAR` (character) converts `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB` data to the database character set. The value returned is always `VARCHAR2`.

When you use this function to convert a character LOB into the database character set, if the LOB value to be converted is larger than the target type, then the database returns an error.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

The following example interprets a simple string as character data:

```
SELECT TO_CHAR('01110') FROM DUAL;

TO_CH
-----
01110
```

Compare this example with the first example for TO_CHAR (number) .

The following example converts some `CLOB` data from the `pm.print_media` table to the database character set:

```
SELECT TO_CHAR(ad_sourcetext) FROM print_media
      WHERE product_id = 2268;

TO_CHAR(AD_SOURCETEXT)
--------------------------------------------------------------------------------
******************************
TIGER2 2268...Standard Hayes Compatible Modem
Product ID: 2268
The #1 selling modem in the universe! Tiger2's modem includes call management
and Internet voicing. Make real-time full duplex phone calls at the same time
you're online.
*********************************
```

TO_CHAR (character) Function: Example

The following statements create a table named `empl_temp` and populate it with employee details:

```
CREATE TABLE empl_temp
  (
     employee_id NUMBER(6),
     first_name  VARCHAR2(20),
     last_name   VARCHAR2(25),
     email       VARCHAR2(25),
     hire_date   DATE DEFAULT SYSDATE,
     job_id      VARCHAR2(10),
     clob_column CLOB
  );

INSERT INTO empl_temp
VALUES(111,'John','Doe','example.com','10-JAN-2015','1001','Experienced
Employee');

INSERT INTO empl_temp
VALUES(112,'John','Smith','example.com','12-JAN-2015','1002','Junior
Employee');

INSERT INTO empl_temp
VALUES(113,'Johnnie','Smith','example.com','12-JAN-2014','1002','Mid-Career
Employee');

INSERT INTO empl_temp
VALUES(115,'Jane','Doe','example.com','15-JAN-2015','1005','Executive
Employee');
```

The following statement converts CLOB data to the database character set:

```
SELECT To_char(clob_column) "CLOB_TO_CHAR"
FROM   empl_temp
WHERE  employee_id IN ( 111, 112, 115 );

CLOB_TO_CHAR
--------------------
Experienced Employee
Junior Employee
Executive Employee
```

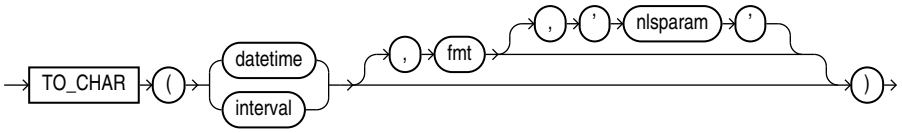
Live SQL:

View and run a related example on Oracle Live SQL at *Using the TO_CHAR Function*

TO_CHAR (datetime)

Syntax

***to_char_date*::=**



Purpose

`TO_CHAR` (datetime) converts a datetime or interval value of `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL DAY TO SECOND`, or `INTERVAL YEAR TO MONTH` data type to a value of `VARCHAR2` data type in the format specified by the date format *`fmt`*. If you omit *`fmt`*, then *`date`* is converted to a `VARCHAR2` value as follows:

- `DATE` values are converted to values in the default date format.
- `TIMESTAMP` and `TIMESTAMP WITH LOCAL TIME ZONE` values are converted to values in the default timestamp format.
- `TIMESTAMP WITH TIME ZONE` values are converted to values in the default timestamp with time zone format.
- Interval values are converted to the numeric representation of the interval literal.

Refer to "Format Models " for information on datetime formats.

The *`'nlsparam'`* argument specifies the language in which month and day names and abbreviations are returned. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

If you omit *`'nlsparam'`*, then this function uses the default date language for your session.

> **See Also:**
>
> "Security Considerations for Data Conversion"

You can use this function in conjunction with any of the XML functions to generate a date in the database format rather than the XML Schema standard format.

> **See Also:**
>
> - *Oracle XML DB Developer's Guide* for information about formatting of XML dates and timestamps, including examples
> - "XML Functions " for a listing of the XML functions
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

The following example uses this table:

```
CREATE TABLE date_tab (
   ts_col      TIMESTAMP,
   tsltz_col   TIMESTAMP WITH LOCAL TIME ZONE,
   tstz_col    TIMESTAMP WITH TIME ZONE);
```

The example shows the results of applying `TO_CHAR` to different `TIMESTAMP` data types. The result for a `TIMESTAMP WITH LOCAL TIME ZONE` column is sensitive to session time zone, whereas the results for the `TIMESTAMP` and `TIMESTAMP WITH TIME ZONE` columns are not sensitive to session time zone:

```
ALTER SESSION SET TIME_ZONE = '-8:00';
INSERT INTO date_tab VALUES (
   TIMESTAMP'1999-12-01 10:00:00',
   TIMESTAMP'1999-12-01 10:00:00',
   TIMESTAMP'1999-12-01 10:00:00');
INSERT INTO date_tab VALUES (
   TIMESTAMP'1999-12-02 10:00:00 -8:00',
   TIMESTAMP'1999-12-02 10:00:00 -8:00',
   TIMESTAMP'1999-12-02 10:00:00 -8:00');

SELECT TO_CHAR(ts_col, 'DD-MON-YYYY HH24:MI:SSxFF') AS ts_date,
   TO_CHAR(tstz_col, 'DD-MON-YYYY HH24:MI:SSxFF TZH:TZM') AS tstz_date
   FROM date_tab
   ORDER BY ts_date, tstz_date;

TS_DATE                       TSTZ_DATE
----------------------------- -----------------------------------
01-DEC-1999 10:00:00.000000   01-DEC-1999 10:00:00.000000 -08:00
02-DEC-1999 10:00:00.000000   02-DEC-1999 10:00:00.000000 -08:00

SELECT SESSIONTIMEZONE,
   TO_CHAR(tsltz_col, 'DD-MON-YYYY HH24:MI:SSxFF') AS tsltz
   FROM date_tab
   ORDER BY sessiontimezone, tsltz;

SESSIONTIM TSLTZ
---------- ------------------------------
-08:00     01-DEC-1999 10:00:00.000000
-08:00     02-DEC-1999 10:00:00.000000

ALTER SESSION SET TIME_ZONE = '-5:00';
SELECT TO_CHAR(ts_col, 'DD-MON-YYYY HH24:MI:SSxFF') AS ts_col,
   TO_CHAR(tstz_col, 'DD-MON-YYYY HH24:MI:SSxFF TZH:TZM') AS tstz_col
   FROM date_tab
```

```
   ORDER BY ts_col, tstz_col;

TS_COL                         TSTZ_COL
------------------------------ ------------------------------------
01-DEC-1999 10:00:00.000000    01-DEC-1999 10:00:00.000000 -08:00
02-DEC-1999 10:00:00.000000    02-DEC-1999 10:00:00.000000 -08:00

SELECT SESSIONTIMEZONE,
TO_CHAR(tsltz_col, 'DD-MON-YYYY HH24:MI:SSxFF') AS tsltz_col
   FROM date_tab
   ORDER BY sessiontimezone, tsltz_col;
  2    3    4
SESSIONTIM TSLTZ_COL
---------- ------------------------------
-05:00     01-DEC-1999 13:00:00.000000
-05:00     02-DEC-1999 13:00:00.000000
```

The following example converts an interval literal into a text literal:

```
SELECT TO_CHAR(INTERVAL '123-2' YEAR(3) TO MONTH) FROM DUAL;

TO_CHAR
-------
+123-02
```

Using TO_CHAR to Format Dates and Numbers: Example

The following statement converts date values to the format specified in the `TO_CHAR` function:

```
WITH dates AS (
  SELECT date'2015-01-01' d FROM dual union
  SELECT date'2015-01-10' d FROM dual union
  SELECT date'2015-02-01' d FROM dual
)
SELECT d "Original Date",
       to_char(d, 'dd-mm-yyyy') "Day-Month-Year",
       to_char(d, 'hh24:mi') "Time in 24-hr format",
       to_char(d, 'iw-iyyy') "ISO Year and Week of Year"
FROM dates;
```

The following statement converts date and timestamp values to the format specified in the `TO_CHAR` function:

```
WITH dates AS (
  SELECT date'2015-01-01' d FROM dual union
  SELECT date'2015-01-10' d FROM dual union
  SELECT date'2015-02-01' d FROM dual union
  SELECT timestamp'2015-03-03 23:44:32' d FROM dual union
  SELECT timestamp'2015-04-11 12:34:56' d FROM dual
)
SELECT d "Original Date",
       to_char(d, 'dd-mm-yyyy') "Day-Month-Year",
       to_char(d, 'hh24:mi') "Time in 24-hr format",
       to_char(d, 'iw-iyyy') "ISO Year and Week of Year",
       to_char(d, 'Month') "Month Name",
       to_char(d, 'Year') "Year"
FROM dates;
```

The following statement extracts the datetime fields specified in the `EXTRACT` function from the input datetime expressions:

```
WITH dates AS (
  SELECT date'2015-01-01' d FROM dual union
```

```
  SELECT date'2015-01-10' d FROM dual union
  SELECT date'2015-02-01' d FROM dual union
  SELECT timestamp'2015-03-03 23:44:32' d FROM dual union
  SELECT timestamp'2015-04-11 12:34:56' d FROM dual
)
SELECT extract(minute from d) minutes,
       extract(hour from d) hours,
       extract(day from d) days,
       extract(month from d) months,
       extract(year from d) years
FROM dates;
```

The following statement displays the input numbers as per the format specified in the `TO_CHAR` function:

```
WITH nums AS (
  SELECT 10 n FROM dual union
  SELECT 9.99 n FROM dual union
  SELECT 1000000 n FROM dual  --one million
)
SELECT n "Input Number N",
       to_char(n),
       to_char(n, '9,999,999.99') "Number with Commas",
       to_char(n, '0,000,000.000') "Zero-padded Number",
       to_char(n, '9.9EEEE') "Scientific Notation"
FROM nums;
```

The following statement converts the input numbers as per the format specified in the `TO_CHAR` function:

```
WITH nums AS (
  SELECT 10 n FROM dual union
  SELECT 9.99 n FROM dual union
  SELECT .99 n FROM dual union
  SELECT 1000000 n FROM dual  --one million
)
SELECT n "Input Number N",
       to_char(n),
       to_char(n, '9,999,999.99') "Number with Commas",
       to_char(n, '0,000,000.000') "Zero_padded Number",
       to_char(n, '9.9EEEE') "Scientific Notation",
       to_char(n, '$9,999,990.00') Monetary,
       to_char(n, 'X') "Hexadecimal Value"
FROM nums;
```

The following statement converts the input numbers as per the format specified in the `TO_CHAR` function:

```
WITH nums AS (
  SELECT 10 n FROM dual union
  SELECT 9.99 n FROM dual union
  SELECT .99 n FROM dual union
  SELECT 1000000 n FROM dual  --one million
)
SELECT n "Input Number N",
       to_char(n),
       to_char(n, '9,999,999.99') "Number with Commas",
       to_char(n, '0,000,000.000') "Zero_padded Number",
       to_char(n, '9.9EEEE') "Scientific Notation",
       to_char(n, '$9,999,990.00') Monetary,
       to_char(n, 'XXXXXX') "Hexadecimal Value"
FROM nums;
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Using TO_CHAR to Format Dates and Numbers*

TO_CHAR (datetime) Function: Example

The following statements create a table named `empl_temp` and populate it with employee details:

```
CREATE TABLE empl_temp
  (
     employee_id NUMBER(6),
     first_name  VARCHAR2(20),
     last_name   VARCHAR2(25),
     email       VARCHAR2(25),
     hire_date   DATE DEFAULT SYSDATE,
     job_id      VARCHAR2(10),
     clob_column CLOB
  );

INSERT INTO empl_temp
VALUES(111,'John','Doe','example.com','10-JAN-2015','1001','Experienced
Employee');

INSERT INTO empl_temp
VALUES(112,'John','Smith','example.com','12-JAN-2015','1002','Junior
Employee');

INSERT INTO empl_temp
VALUES(113,'Johnnie','Smith','example.com','12-JAN-2014','1002','Mid-Career
Employee');

INSERT INTO empl_temp
VALUES(115,'Jane','Doe','example.com','15-JAN-2015','1005','Executive
Employee');
```

The following statement displays dates by using the short and long formats:

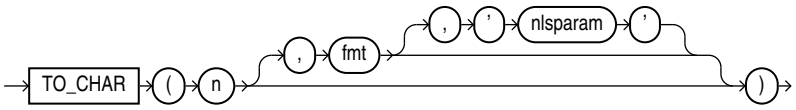
```
SELECT hire_date "Default",
       TO_CHAR(hire_date,'DS') "Short",
       TO_CHAR(hire_date,'DL') "Long"FROM empl_temp
WHERE employee_id IN (111, 112, 115);

Default    Short      Long
---------- ---------- -------------------------
10-JAN-15  1/10/2015  Saturday, January 10, 2015
12-JAN-15  1/12/2015  Monday, January 12, 2015
15-JAN-15  1/15/2015  Thursday, January 15, 2015
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Using the TO_CHAR Function*

TO_CHAR (number)

Syntax

***to_char_number*::=**



Purpose

`TO_CHAR` (number) converts *`n`* to a value of `VARCHAR2` data type, using the optional number format *`fmt`*. The value *`n`* can be of type `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`. If you omit *`fmt`*, then *`n`* is converted to a `VARCHAR2` value exactly long enough to hold its significant digits.

If *`n`* is negative, then the sign is applied after the format is applied. Thus `TO_CHAR(-1, '$9')` returns -$1, rather than $-1.

Refer to "Format Models " for information on number formats.

The *`'nlsparam'`* argument specifies these characters that are returned by number format elements:

- Decimal character
- Group separator
- Local currency symbol
- International currency symbol

This argument can have this form:

```
'NLS_NUMERIC_CHARACTERS = ''dg''
   NLS_CURRENCY = ''text''
   NLS_ISO_CURRENCY = territory '
```

The characters *`d`* and *`g`* represent the decimal character and group separator, respectively. They must be different single-byte characters. Within the quoted string, you must use two single quotation marks around the parameter values. Ten characters are available for the currency symbol.

If you omit *`'nlsparam'`* or any one of the parameters, then this function uses the default parameter values for your session.

> **See Also:**
>
> - "Security Considerations for Data Conversion"
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

The following statement uses implicit conversion to combine a string and a number into a number:

```
SELECT TO_CHAR('01110' + 1) FROM DUAL;

TO_C
----
1111
```

Compare this example with the first example for TO_CHAR (character) .

In the next example, the output is blank padded to the left of the currency symbol. In the optional number format fmt, `L` designates local currency symbol and `MI` designates a trailing minus sign. See Table 2-16 for a complete listing of number format elements. The example shows the output in a session in which the session parameter `NLS_TERRITORY` is set to `AMERICA`.

```
SELECT TO_CHAR(-10000,'L99G999D99MI') "Amount"
     FROM DUAL;

Amount
--------------
  $10,000.00-
```

In the next example, `NLS_CURRENCY` specifies the string to use as the local currency symbol for the `L` number format element. `NLS_NUMERIC_CHARACTERS` specifies comma as the character to use as the decimal separator for the `D` number format element and period as the character to use as the group separator for the `G` number format element. These characters are expected in many countries, for example in Germany.

```
SELECT TO_CHAR(-10000,'L99G999D99MI',
   'NLS_NUMERIC_CHARACTERS = '',.''
   NLS_CURRENCY = ''AusDollars'' ') "Amount"
     FROM DUAL;

Amount
-------------------
AusDollars10.000,00-
```

In the next example, `NLS_ISO_CURRENCY` instructs the database to use the international currency symbol for the territory of `POLAND` for the `C` number format element:

```
SELECT TO_CHAR(-10000,'99G999D99C',
   'NLS_NUMERIC_CHARACTERS = '',.''
   NLS_ISO_CURRENCY=POLAND') "Amount"
     FROM DUAL;

Amount
```

```
-----------------
    -10.000,00PLN
```

TO_CHAR (number) Function: Example

The following statements create a table named `empl_temp` and populate it with employee details:

```
CREATE TABLE empl_temp
  (
     employee_id NUMBER(6),
     first_name  VARCHAR2(20),
     last_name   VARCHAR2(25),
     email       VARCHAR2(25),
     hire_date   DATE DEFAULT SYSDATE,
     job_id      VARCHAR2(10),
     clob_column CLOB
  );

INSERT INTO empl_temp
VALUES(111,'John','Doe','example.com','10-JAN-2015','1001','Experienced
Employee');

INSERT INTO empl_temp
VALUES(112,'John','Smith','example.com','12-JAN-2015','1002','Junior
Employee');

INSERT INTO empl_temp
VALUES(113,'Johnnie','Smith','example.com','12-JAN-2014','1002','Mid-Career
Employee');

INSERT INTO empl_temp
VALUES(115,'Jane','Doe','example.com','15-JAN-2015','1005','Executive
Employee');
```

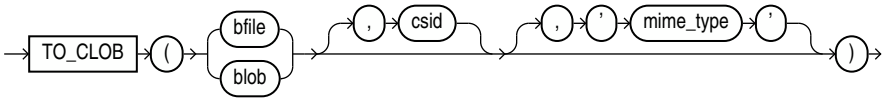
The following statement converts numeric data to the database character set:

```
SELECT To_char(employee_id) "NUM_TO_CHAR"
FROM   empl_temp
WHERE  employee_id IN ( 111, 112, 113, 115 );

NUM_TO_CHAR
--------------------
111
112
113
115
```

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Using the TO_CHAR Function*

TO_CLOB (bfile|blob)

Syntax



Purpose

`TO_CLOB` (bfile|blob) converts `BFILE` or `BLOB` data to the database character set and returns the data as a `CLOB` value.

For *`csid`*, specify the character set ID of the `BFILE` or `BLOB` data. If the character set of the `BFILE` or `BLOB` data is the database character set, then you can specify a value of 0 for *`csid`*, or omit *`csid`* altogether.

For *`mime_type`*, specify the MIME type to be set on the `CLOB` value returned by this function. If you omit *`mime_type`*, then a MIME type will not be set on the `CLOB` value.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

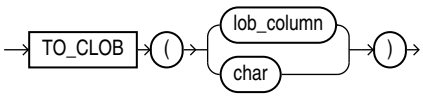
Example

The following hypothetical example returns the `CLOB` of a `BFILE` column value `docu` in table `media_tab`, which uses the character set with ID 873. It sets the MIME type to `text/xml` for the resulting `CLOB`.

```
SELECT TO_CLOB(docu, 873, 'text/xml') FROM media_tab;
```

TO_CLOB (character)

Syntax



Purpose

`TO_CLOB` (character) converts `NCLOB` values in a LOB column or other character strings to `CLOB` values. *`char`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`.

Oracle Database executes this function by converting the underlying LOB data from the national character set to the database character set.

From within a PL/SQL package, you can use the `TO_CLOB` (character) function to convert `RAW`, `CHAR`, `VARCHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB` values to `CLOB` or `NCLOB` values.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

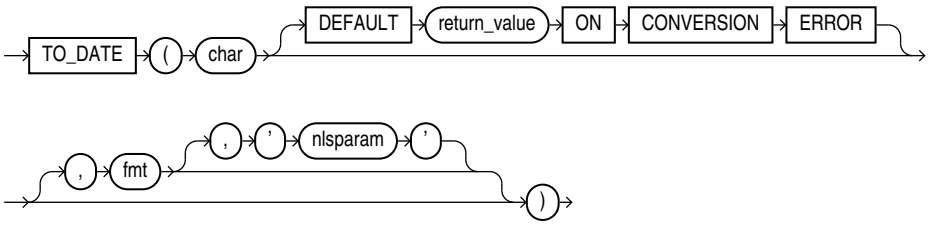
Examples

The following statement converts `NCLOB` data from the sample `pm.print_media` table to `CLOB` and inserts it into a `CLOB` column, replacing existing data in that column.

```
UPDATE PRINT_MEDIA
   SET AD_FINALTEXT = TO_CLOB (AD_FLTEXTN);
```

TO_DATE

Syntax



Purpose

`TO_DATE` converts *`char`* to a value of `DATE` data type.

For *`char`*, you can specify any expression that evaluates to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type.

> **Note:**
>
> This function does not convert data to any of the other datetime data types. For information on other datetime conversions, refer to TO_TIMESTAMP , TO_TIMESTAMP_TZ , TO_DSINTERVAL , and TO_YMINTERVAL .

The optional `DEFAULT` *`return_value`* `ON CONVERSION ERROR` clause allows you to specify the value this function returns if an error occurs while converting *`char`* to `DATE`. This clause has no effect if an error occurs while evaluating *`char`*. The *`return_value`* can be an expression or a bind variable, and it must evaluate to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type, or null. The function converts *`return_value`* to `DATE` using the same method it uses

to convert *char* to DATE. If *return_value* cannot be converted to DATE, then the function returns an error.

The *fmt* is a datetime model format specifying the format of *char*. If you omit *fmt*, then *char* must be in the default date format. The default date format is determined implicitly by the NLS_TERRITORY initialization parameter or can be set explicitly by the NLS_DATE_FORMAT parameter. If *fmt* is J, for Julian, then *char* must be an integer.

> **Caution:**
>
> It is good practice always to specify a format mask (*fmt*) with TO_DATE, as shown in the examples in the section that follow, if *char* is a literal or an expression that evaluates to a known, fixed format, independent of the locale (NLS) configuration of the session. When TO_DATE is used without a format mask, the function is valid only if *char* uses the same format as is determined by the NLS_TERRITORY and NLS_DATE_FORMAT parameters.
>
> However, if *char* corresponds to user input provided by an application, for example, in a bind variable, and the user input is expected to follow the locale (NLS) conventions set for the session provided in the NLS_DATE_FORMAT parameter, then the format mask should not be specified.

The *'nlsparam'* argument specifies the language of the text string that is being converted to a date. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

Do not use the TO_DATE function with a DATE value for the *char* argument. The first two digits of the returned DATE value can differ from the original *char*, depending on *fmt* or the default date format.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> "Datetime Format Models " and "Data Type Comparison Rules " for more information

Examples

The following example converts a character string into a date:

```
SELECT TO_DATE(
  'January 15, 1989, 11:00 A.M.',
  'Month dd, YYYY, HH:MI A.M.',
  'NLS_DATE_LANGUAGE = American')
    FROM DUAL;

TO_DATE('
---------
15-JAN-89
```

The value returned reflects the default date format if the NLS_TERRITORY parameter is set to 'AMERICA'. Different NLS_TERRITORY values result in different default date formats:

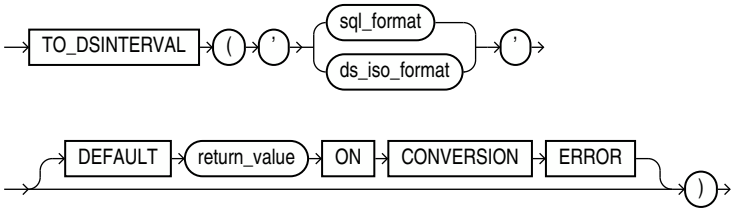
```
ALTER SESSION SET NLS_TERRITORY = 'KOREAN';

SELECT TO_DATE(
    'January 15, 1989, 11:00 A.M.',
    'Month dd, YYYY, HH:MI A.M.',
     'NLS_DATE_LANGUAGE = American')
     FROM DUAL;

TO_DATE(
--------
89/01/15
```

The following example returns the default value because the specified expression cannot be converted to a DATE value, due to a misspelling of the month:

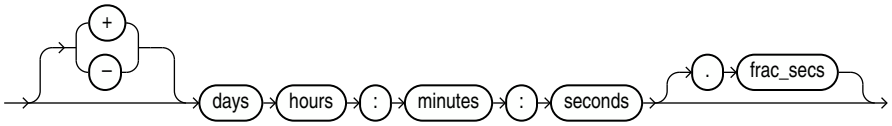
```
SELECT TO_DATE('Febuary 15, 2016, 11:00 A.M.'
       DEFAULT 'January 01, 2016 12:00 A.M.' ON CONVERSION ERROR,
       'Month dd, YYYY, HH:MI A.M.') "Value"
  FROM DUAL;

Value
---------
01-JAN-16
```

TO_DSINTERVAL

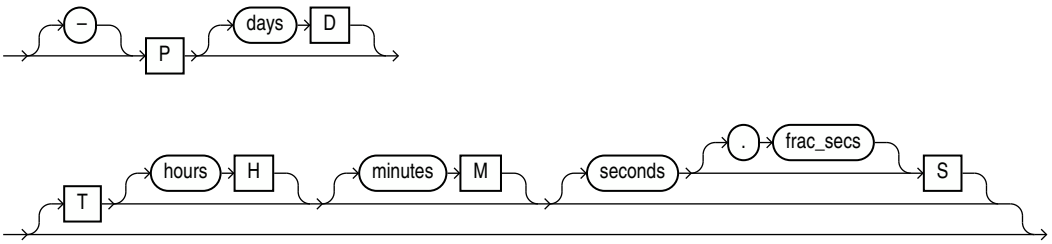
Syntax



***sql_format*::=**



***ds_iso_format*::=**



> **Note:**
>
> In earlier releases, the `TO_DSINTERVAL` function accepted an optional *`nlsparam`* clause. This clause is still accepted for backward compatibility, but has no effect.

Purpose

`TO_DSINTERVAL` converts its argument to a value of `INTERVAL DAY TO SECOND` data type.

For the argument, you can specify any expression that evaluates to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type.

`TO_DSINTERVAL` accepts argument in one of the two formats:

- SQL interval format compatible with the SQL standard (ISO/IEC 9075)
- ISO duration format compatible with the ISO 8601:2004 standard

In the SQL format, *`days`* is an integer between 0 and 999999999, *`hours`* is an integer between 0 and 23, and *`minutes`* and *`seconds`* are integers between 0 and 59. *`frac_secs`* is the fractional part of seconds between .0 and .999999999. One or more blanks separate days from hours. Additional blanks are allowed between format elements.

In the ISO format, *`days`*, *`hours`*, *`minutes`* and *`seconds`* are integers between 0 and 999999999. *`frac_secs`* is the fractional part of seconds between .0 and .999999999. No blanks are allowed in the value. If you specify `T`, then you must specify at least one of the *`hours`*, *`minutes`*, or *`seconds`* values.

The optional `DEFAULT` *`return_value`* `ON CONVERSION ERROR` clause allows you to specify the value this function returns if an error occurs while converting the argument to an `INTERVAL DAY TO SECOND` type. This clause has no effect if an error occurs while evaluating the argument. The *`return_value`* can be an expression or a bind variable, and it must evaluate to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type. It can be in either the SQL format or ISO format, and need not be in the same format as the function argument. If *`return_value`* cannot be converted to an `INTERVAL DAY TO SECOND` type, then the function returns an error.

Examples

The following example uses the SQL format to select from the `hr.employees` table the employees who had worked for the company for at least 100 days on November 1, 2002:

```
SELECT employee_id, last_name FROM employees
   WHERE hire_date + TO_DSINTERVAL('100 00:00:00')
```

```
   <= DATE '2002-11-01'
   ORDER BY employee_id;

EMPLOYEE_ID LAST_NAME
----------- ---------------
        102 De Haan
        203 Mavris
        204 Baer
        205 Higgins
        206 Giet
```

The following example uses the ISO format to display the timestamp 100 days and 5 hours after the beginning of the year 2009:

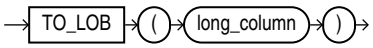
```
SELECT TO_CHAR(TIMESTAMP '2009-01-01 00:00:00' + TO_DSINTERVAL('P100DT05H'),
   'YYYY-MM-DD HH24:MI:SS') "Time Stamp"
     FROM DUAL;

Time Stamp
-------------------
2009-04-11 05:00:00
```

The following example returns the default value because the specified expression cannot be converted to an `INTERVAL DAY TO SECOND` value:

```
SELECT TO_DSINTERVAL('1o 1:02:10'
       DEFAULT '10 8:00:00' ON CONVERSION ERROR) "Value"
  FROM DUAL;

Value
------------------------------
+000000010 08:00:00.000000000
```

TO_LOB

Syntax



Purpose

> **Note:**
>
> All forms of `LONG` data types (`LONG`, `LONG RAW`, `LONG VARCHAR`, `LONG VARRAW`) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the `LONG` data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use `CLOB` and `NCLOB` data types for large amounts of character data.

`TO_LOB` converts `LONG` or `LONG RAW` values in the column *`long_column`* to LOB values. You can apply this function only to a `LONG` or `LONG RAW` column, and only in the select list of a subquery in an `INSERT` statement.

Before using this function, you must create a LOB column to receive the converted `LONG` values. To convert `LONG` values, create a `CLOB` column. To convert `LONG` RAW values, create a `BLOB` column.

You cannot use the `TO_LOB` function to convert a `LONG` column to a LOB column in the subquery of a `CREATE TABLE ... AS SELECT` statement if you are creating an index-organized table. Instead, create the index-organized table without the `LONG` column, and then use the `TO_LOB` function in an `INSERT ... AS SELECT` statement.

You cannot use this function within a PL/SQL package. Instead use the `TO_CLOB` (character) or `TO_BLOB` (raw) functions.

> **See Also:**
>
> - the *modify_col_properties* clause of ALTER TABLE for an alternative method of converting `LONG` columns to LOB
> - INSERT for information on the subquery of an `INSERT` statement
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

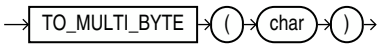
Examples

The following syntax shows how to use the `TO_LOB` function on your `LONG` data in a hypothetical table *old_table*:

```
CREATE TABLE new_table (col1, col2, ... lob_col CLOB);
INSERT INTO new_table (select o.col1, o.col2, ... TO_LOB(o.old_long_col)
   FROM old_table o;
```

TO_MULTI_BYTE

Syntax



Purpose

`TO_MULTI_BYTE` returns *char* with all of its single-byte characters converted to their corresponding multibyte characters. *char* can be of data type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The value returned is in the same data type as *char*.

Any single-byte characters in *char* that have no multibyte equivalents appear in the output string as single-byte characters. This function is useful only if your database character set contains both single-byte and multibyte characters.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `TO_MULTI_BYTE`

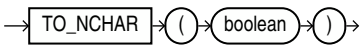
Examples

The following example illustrates converting from a single byte `A` to a multibyte `A` in UTF8:

```
SELECT dump(TO_MULTI_BYTE( 'A')) FROM DUAL;

DUMP(TO_MULTI_BYTE('A'))
------------------------
Typ=1 Len=3: 239,188,161
```

TO_NCHAR (boolean)

Syntax



Purpose

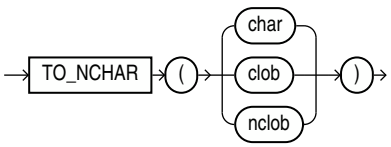
Use `TO_NCHAR(boolean)` to explicitly convert a boolean value to a character value of `'TRUE'` or `'FALSE'`.

> **See Also:**
>
> - CAST for conversion rules.
> - Boolean Data Type for more details on the built-in boolean data type.

TO_NCHAR (character)

Syntax

***to_nchar_char*::=**



Purpose

`TO_NCHAR` (character) converts a character string, `CHAR`, `VARCHAR2`, `CLOB`, or `NCLOB` value to the national character set. The value returned is always `NVARCHAR2`. This function is equivalent to the `TRANSLATE ... USING` function with a `USING` clause in the national character set.

> **See Also:**
>
> - "Data Conversion " and TRANSLATE ... USING
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

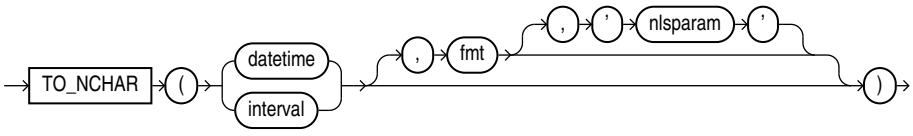
The following example converts `VARCHAR2` data from the `oe.customers` table to the national character set:

```
SELECT TO_NCHAR(cust_last_name) FROM customers
   WHERE customer_id=103;

TO_NCHAR(CUST_LAST_NAME)
--------------------------------------------------
Taylor
```

TO_NCHAR (datetime)

Syntax

***to_nchar_date*::=**



Purpose

`TO_NCHAR` (datetime) converts a datetime or interval value of `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL MONTH TO YEAR`, or `INTERVAL DAY TO SECOND` data type from the database character set to the national character set.

> **See Also:**
>
> - "Security Considerations for Data Conversion"
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

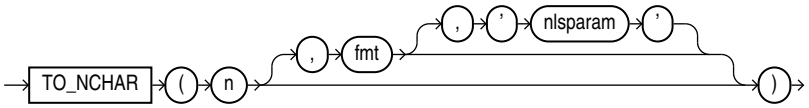
The following example converts the `order_date` of all orders whose status is `9` to the national character set:

```
SELECT TO_NCHAR(ORDER_DATE) AS order_date
   FROM ORDERS
   WHERE ORDER_STATUS > 9
   ORDER BY order_date;

ORDER_DATE
-------------------------------------------------------------------------------
06-DEC-99 02.22.34.225609 PM
13-SEP-99 10.19.00.654279 AM
14-SEP-99 09.53.40.223345 AM
26-JUN-00 10.19.43.190089 PM
27-JUN-00 09.53.32.335522 PM
```

TO_NCHAR (number)

Syntax

***to_nchar_number*::=**



Purpose

`TO_NCHAR` (number) converts *`n`* to a string in the national character set. The value *`n`* can be of type `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`. The function returns a value of the same type as the argument. The optional *`fmt`* and *`'nlsparam'`* corresponding to *`n`* can be of `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL MONTH TO YEAR`, or `INTERVAL DAY TO SECOND` data type.

> **See Also:**
>
> - "Security Considerations for Data Conversion"
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

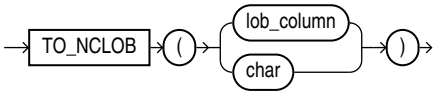
The following example converts the `customer_id` values from the sample table `oe.orders` to the national character set:

```
SELECT TO_NCHAR(customer_id) "NCHAR_Customer_ID"  FROM orders
   WHERE order_status > 9
   ORDER BY "NCHAR_Customer_ID";
```

```
NCHAR_Customer_ID
----------------------------------------
102
103
148
148
149
```

TO_NCLOB

Syntax



Purpose

`TO_NCLOB` converts `CLOB` values in a LOB column or other character strings to `NCLOB` values. *`char`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. Oracle Database implements this function by converting the character set of *`char`* from the database character set to the national character set.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of this function

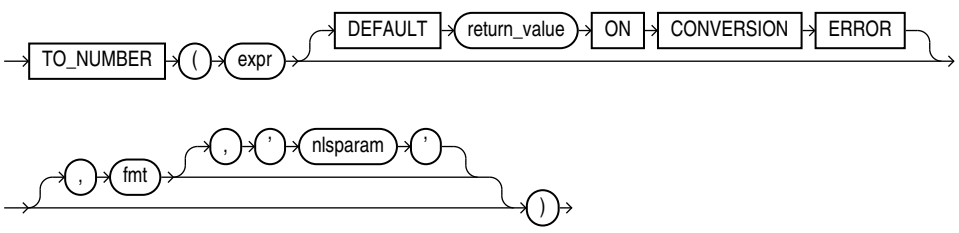
Examples

The following example inserts some character data into an `NCLOB` column of the `pm.print_media` table by first converting the data with the `TO_NCLOB` function:

```
INSERT INTO print_media (product_id, ad_id, ad_fltextn)
   VALUES (3502, 31001,
      TO_NCLOB('Placeholder for new product description'));
```

TO_NUMBER

Syntax



Purpose

TO_NUMBER converts *expr* to a value of NUMBER data type.

expr can be any expression that evaluates to a character string of type CHAR, VARCHAR2, NCHAR, or NVARCHAR2, a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE, BOOLEAN, or null. If *expr* is NUMBER, then the function returns *expr*. If *expr* evaluates to null, then the function returns null. Otherwise, the function converts *expr* to a NUMBER value.

- If you specify an *expr* of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, then you can optionally specify the format model *fmt*.
- If you specify an *expr* of BINARY_FLOAT or BINARY_DOUBLE data type, then you cannot specify a format model because a float can be interpreted only by its internal representation.
- If you specify an *expr* of type BOOLEAN, then TRUE will be converted to 1 and FALSE will be converted to 0. You cannot specify a format model with inputs of type BOOLEAN.

Refer to "Format Models " for information on number formats.

The *'nlsparam'* argument in this function has the same purpose as it does in the TO_CHAR function for number conversions. Refer to TO_CHAR (number) for more information.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> "Data Type Comparison Rules " for more information.

Examples

The following examples convert character string data into a number:

```
UPDATE employees SET salary = salary +
   TO_NUMBER('100.00', '9G999D99')
   WHERE last_name = 'Perkins';

SELECT TO_NUMBER('-AusDollars100','L9G999D99',
   ' NLS_NUMERIC_CHARACTERS = '',.''
     NLS_CURRENCY            = ''AusDollars''
   ') "Amount"
     FROM DUAL;

    Amount
----------
      -100
```

The following example returns the default value of 0 because the specified expression cannot be converted to a NUMBER value:

```
SELECT TO_NUMBER('2,00' DEFAULT 0 ON CONVERSION ERROR) "Value"
  FROM DUAL;

   Value
--------
       0
```

TO_SINGLE_BYTE

Syntax

Purpose

`TO_SINGLE_BYTE` returns *`char`* with all of its multibyte characters converted to their corresponding single-byte characters. *`char`* can be of data type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The value returned is in the same data type as *`char`*.

Any multibyte characters in *`char`* that have no single-byte equivalents appear in the output as multibyte characters. This function is useful only if your database character set contains both single-byte and multibyte characters.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information.
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `TO_SINGLE_BYTE`

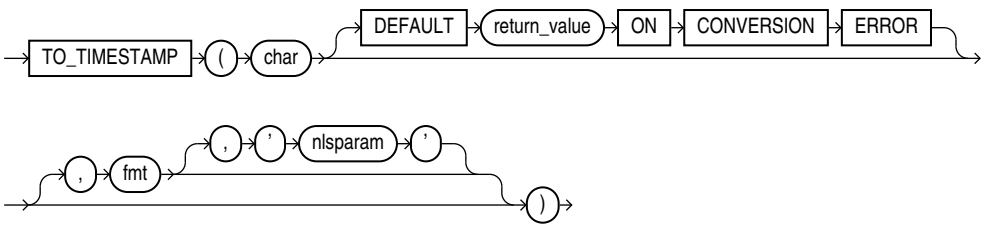
Examples

The following example illustrates going from a multibyte `A` in UTF8 to a single byte ASCII `A`:

```
SELECT TO_SINGLE_BYTE( CHR(15711393)) FROM DUAL;

T
-
A
```

TO_TIMESTAMP

Syntax



Purpose

TO_TIMESTAMP converts *char* to a value of TIMESTAMP data type.

For *char*, you can specify any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type.

The optional DEFAULT *return_value* ON CONVERSION ERROR clause allows you to specify the value this function returns if an error occurs while converting *char* to TIMESTAMP. This clause has no effect if an error occurs while evaluating *char*. The *return_value* can be an expression or a bind variable, and it must evaluate to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, or null. The function converts *return_value* to TIMESTAMP using the same method it uses to convert *char* to TIMESTAMP. If *return_value* cannot be converted to TIMESTAMP, then the function returns an error.

The optional *fmt* specifies the format of *char*. If you omit *fmt*, then *char* must be in the default format of the TIMESTAMP data type, which is determined by the NLS_TIMESTAMP_FORMAT initialization parameter. The optional *'nlsparam'* argument has the same purpose in this function as in the TO_CHAR function for date conversion.

> **Caution:**
>
> It is good practice always to specify a format mask (*fmt*) with TO_TIMESTAMP, as shown in the examples in the section that follow, if *char* is a literal or an expression that evaluates to a known, fixed format, independent of the locale (NLS) configuration of the session. When TO_TIMESTAMP is used without a format mask, the function is valid only if *char* uses the same format as is determined by the NLS_TERRITORY and NLS_TIMESTAMP_FORMAT parameters.
>
> However, if *char* corresponds to user input provided by an application, for example, in a bind variable, and the user input is expected to follow the locale (NLS) conventions set for the session provided in the NLS_TIMESTAMP_FORMAT parameter, then the format mask should not be specified.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> "Data Type Comparison Rules " for more information.

Examples

The following example converts a character string to a timestamp. The character string is not in the default TIMESTAMP format, so the format mask must be specified:

```
SELECT TO_TIMESTAMP ('10-Sep-02 14:10:10.123000', 'DD-Mon-RR HH24:MI:SS.FF')
  FROM DUAL;

TO_TIMESTAMP('10-SEP-0214:10:10.123000','DD-MON-RRHH24:MI:SS.FF')
```

```
--------------------------------------------------------------------------------
10-SEP-02 02.10.10.123000000 PM
```

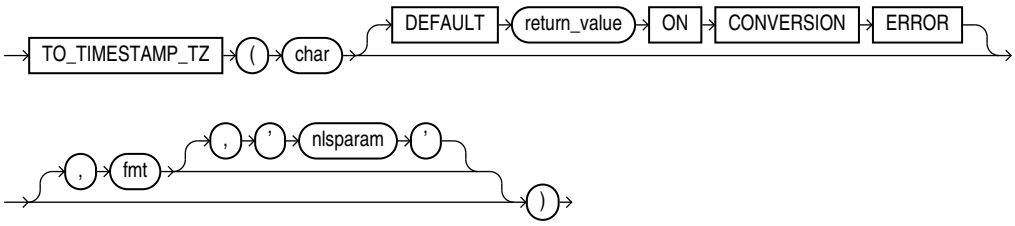
The following example returns the default value of NULL because the specified expression cannot be converted to a `TIMESTAMP` value, due to an invalid month specification:

```
SELECT TO_TIMESTAMP ('10-Sept-02 14:10:10.123000'
  DEFAULT NULL ON CONVERSION ERROR,
  'DD-Mon-RR HH24:MI:SS.FF',
  'NLS_DATE_LANGUAGE = American') "Value"
  FROM DUAL;
```

> **See Also:**
>
> `NLS_TIMESTAMP_FORMAT` initialization parameter for information on the default `TIMESTAMP` format and "Datetime Format Models " for information on specifying the format mask

TO_TIMESTAMP_TZ

Syntax



Purpose

`TO_TIMESTAMP_TZ` converts *`char`* to a value of `TIMESTAMP WITH TIME ZONE` data type.

For *`char`*, you can specify any expression that evaluates to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type.

> **Note:**
>
> This function does not convert character strings to `TIMESTAMP WITH LOCAL TIME ZONE`. To do this, use a `CAST` function, as shown in CAST .

The optional `DEFAULT` *`return_value`* `ON CONVERSION ERROR` clause allows you to specify the value this function returns if an error occurs while converting *`char`* to `TIMESTAMP WITH TIME ZONE`. This clause has no effect if an error occurs while evaluating *`char`*. The *`return_value`* can be an expression or a bind variable, and it must evaluate to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type, or null. The function converts *`return_value`* to `TIMESTAMP WITH TIME ZONE` using the same method it uses to convert *`char`* to `TIMESTAMP WITH`

TIME ZONE. If *return_value* cannot be converted to TIMESTAMP WITH TIME ZONE, then the function returns an error.

The optional *fmt* specifies the format of *char*. If you omit *fmt*, then *char* must be in the default format of the TIMESTAMP WITH TIME ZONE data type. The optional *'nlsparam'* has the same purpose in this function as in the TO_CHAR function for date conversion.

> **Caution:**
>
> It is good practice always to specify a format mask (*fmt*) with TO_TIMESTAMP_TZ, as shown in the examples in the section that follow, if *char* is a literal or an expression that evaluates to a known, fixed format, independent of the locale (NLS) configuration of the session. When TO_TIMESTAMP_TZ is used without a format mask, the function is valid only if *char* uses the same format as is determined by the NLS_TERRITORY and NLS_TIMESTAMP_TZ_FORMAT parameters.
>
> However, if *char* corresponds to user input provided by an application, for example, in a bind variable, and the user input is expected to follow the locale (NLS) conventions set for the session provided in the NLS_TIMESTAMP_TZ_FORMAT parameter, then the format mask should not be specified.

Examples

The following example converts a character string to a value of TIMESTAMP WITH TIME ZONE:

```
SELECT TO_TIMESTAMP_TZ('1999-12-01 11:00:00 -8:00',
  'YYYY-MM-DD HH:MI:SS TZH:TZM') FROM DUAL;

TO_TIMESTAMP_TZ('1999-12-0111:00:00-08:00','YYYY-MM-DDHH:MI:SSTZH:TZM')
---------------------------------------------------------------------
01-DEC-99 11.00.00.000000000 AM -08:00
```

The following example casts a null column in a UNION operation as TIMESTAMP WITH LOCAL TIME ZONE using the sample tables oe.order_items and oe.orders:

```
SELECT order_id, line_item_id,
  CAST(NULL AS TIMESTAMP WITH LOCAL TIME ZONE) order_date
  FROM order_items
UNION
SELECT order_id, to_number(null), order_date
  FROM orders;

  ORDER_ID LINE_ITEM_ID ORDER_DATE
---------- ------------ ---------------------------------
      2354            1
      2354            2
      2354            3
      2354            4
      2354            5
      2354            6
      2354            7
      2354            8
      2354            9
      2354           10
      2354           11
```

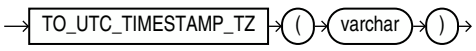
```
      2354           12
      2354           13
      2354              14-JUL-00 05.18.23.234567 PM
      2355            1
      2355            2
. . .
```

The following example returns the default value of NULL because the specified expression cannot be converted to a `TIMESTAMP WITH TIME ZONE` value, due to an invalid month specification:

```
SELECT TO_TIMESTAMP_TZ('1999-13-01 11:00:00 -8:00'
  DEFAULT NULL ON CONVERSION ERROR,
  'YYYY-MM-DD HH:MI:SS TZH:TZM') "Value"
  FROM DUAL;
```

TO_UTC_TIMESTAMP_TZ

Syntax



Purpose

The SQL function `TO_UTC_TIMESTAMP_TZ` takes an ISO 8601 date format string as the `varchar` input and returns an instance of SQL data type `TIMESTAMP WITH TIMEZONE`. It normalizes the input to UTC time (Coordinated Universal Time, formerly Greenwich Mean Time). Unlike SQL function `TO_TIMESTAMP_TZ` , the new function assumes that the input string uses the ISO 8601 date format, defaulting the time zone to UTC 0.

A typical use of this function would be to provide its output to SQL function `SYS_EXTRACT_UTC`, obtaining a UTC time that is then passed as a SQL bind variable to SQL/JSON condition `JSON_EXISTS`, to perform a time-stamp range comparison.

This is the allowed syntax for dates and times:

- Date (only): *YYYY-MM-DD*
- Date with time: *YYYY*-*MM*-*DD*T*hh*:*mm*:*ss*[.*s*[*s*[*s*[*s*[*s*[*s*]]]]]][Z|(+|-)*hh*:*mm*]

where:

- ***YYYY*** specifies the *year*, as four decimal digits.
- ***MM*** specifies the *month*, as two decimal digits, `00` to `12`.
- ***DD*** specifies the *day*, as two decimal digits, `00` to `31`.
- ***hh*** specifies the *hour*, as two decimal digits, `00` to `23`.
- ***mm*** specifies the *minutes*, as two decimal digits, `00` to `59`.
- ***ss***[.***s***[***s***[***s***[***s***[***s***]]]]]] specifies the *seconds*, as two decimal digits, `00` to `59`, optionally followed by a decimal point and 1 to 6 decimal digits (representing the fractional part of a second).
- **Z** specifies *UTC* time (time zone 0). (It can also be specified by `+00:00`, but not by `-00:00`.)

- `(+|-)hh:mm` specifies the time-zone as *difference from UTC*. (One of `+` or `-` is required.)

For a time value, the time-zone part is optional. If it is absent then UTC time is assumed.

No other ISO 8601 date-time syntax is supported. In particular:

- Negative dates (dates prior to year 1 BCE), which begin with a hyphen (e.g. `-2018-10-26T21:32:52`), are not supported.
- Hyphen and colon separators are required: so-called “basic” format, `YYYYMMDDThhmmss`, is not supported.
- Ordinal dates (year plus day of year, calendar week plus day number) are not supported.
- Using more than four digits for the year is not supported.

Supported dates and times include the following:

- `2018-10-26T21:32:52`
- `2018-10-26T21:32:52+02:00`
- `2018-10-26T19:32:52Z`
- `2018-10-26T19:32:52+00:00`
- `2018-10-26T21:32:52.12679`

Unsupported dates and times include the following:

- `2018-10-26T21:32` (if a time is specified then all of its parts must be present)
- `2018-10-26T25:32:52+02:00` (the hours part, 25, is out of range)
- `18-10-26T21:32` (the year is not specified fully)

Examples

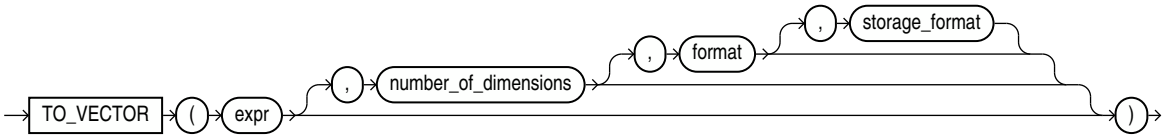
```
SELECT TO_UTC_TIMESTAMP_TZ('1998-01-01') FROM DUAL;

TO_UTC_TIMESTAMP_TZ('1998-01-01')
---------------------------------------------------------------------------
01-JAN-98 12.00.00.000000000 AM +00:00

SELECT TO_UTC_TIMESTAMP_TZ('2000-01-02T12:34:56.789') FROM DUAL;

TO_UTC_TIMESTAMP_TZ('2000-01-02T12:34:56.789')
---------------------------------------------------------------------------
02-JAN-00 12.34.56.789000000 PM +00:00

SELECT TO_UTC_TIMESTAMP_TZ('2016-05-05T00:00:00.000Z') FROM DUAL;

TO_UTC_TIMESTAMP_TZ('2016-05-05T00:00:00.000Z')
---------------------------------------------------------------------------
05-MAY-16 12.00.00.000000000 AM +00:00

SELECT TO_UTC_TIMESTAMP_TZ('2016-05-05T02:04:35.4678Z') FROM DUAL;

TO_UTC_TIMESTAMP_TZ('2016-05-05T02:04:35.4678Z')
---------------------------------------------------------------------------
05-MAY-16 02.04.35.467800000 AM +00:00
```

> **See Also:**
>
> - ISO 8601 standard
> - ISO 8601 at Wikipedia

TO_VECTOR

`TO_VECTOR` is a constructor that takes a string of type `VARCHAR2`, `CLOB`, `BLOB`, or `JSON` as input, converts it to a vector, and returns a vector as output. `TO_VECTOR` also takes another vector as input, adjusts its format, and returns the adjusted vector as output. `TO_VECTOR` is synonymous with `VECTOR`.

Syntax



Parameters

- *expr* must evaluate to one of:
  - A string (of character types or `CLOB`) that represents a vector.
  - A `VECTOR`.
  - A `BLOB`. The `BLOB` must represent the vector's binary bytes.
  - A JSON array. All elements in the array must be numeric.

  If *expr* is NULL, the result is NULL.

  The string representation of the vector must be in the form of an array of non-null numbers enclosed with a bracket and separated by commas, such as `[1, 3.4, -05.60, 3e+4]`. `TO_VECTOR` converts a valid string representation of a vector to a vector in the format specified. If no format is specified the default format is used.
- *number_of_dimensions* must be a numeric value that describes the number of dimensions of the vector to construct. The number of dimensions may also be specified as an asterisk (*), in which case the dimension is determined by *expr*.
- *format* must be one of the following tokens: `INT8`, `FLOAT32`, `FLOAT64`, `BINARY`, or `*`. This is the target internal storage format of the vector. If `*` is used, the format will be `FLOAT32`.

  Note that this behavior is different from declaring a vector column. When you declare a column of type `VECTOR(3, *)`, then all inserted vectors will be stored as is without a change in format.
- *storage_format* must be one of the following tokens: `DENSE`, `SPARSE`, or `*`. If no storage format is specified or if `*` is used, the following will be observed depending on the input type:
  - Textual input: the storage format will default to `DENSE`.

- JSON input: the storage format will default to `DENSE`.
- `VECTOR` input: there is no default and the storage format is not changed.
- `BLOB` input: there is no default and the storage format is not changed.

Examples

```
SELECT TO_VECTOR('[34.6, 77.8]');

TO_VECTOR('[34.6,77.8]')
------------------------------------------------------------
[3.45999985E+001,7.78000031E+001]


SELECT TO_VECTOR('[34.6, 77.8]', 2, FLOAT32);

TO_VECTOR('[34.6,77.8]',2,FLOAT32)
------------------------------------------------------------
[3.45999985E+001,7.78000031E+001]


SELECT TO_VECTOR('[34.6, 77.8, -89.34]', 3, FLOAT32);

TO_VECTOR('[34.6,77.8,-89.34]',3,FLOAT32)
------------------------------------------------------------
[3.45999985E+001,7.78000031E+001,-8.93399963E+001]


SELECT TO_VECTOR('[34.6, 77.8, -89.34]', 3, FLOAT32, DENSE);

TO_VECTOR('[34.6,77.8,-89.34]',3,FLOAT32,DENSE)
------------------------------------------------------------
[3.45999985E+001,7.78000031E+001,-8.93399963E+001]
```

Note:

- For applications using Oracle Client libraries prior to 23ai connected to Oracle Database 23ai, use the `TO_VECTOR` function to insert vector data. For example:

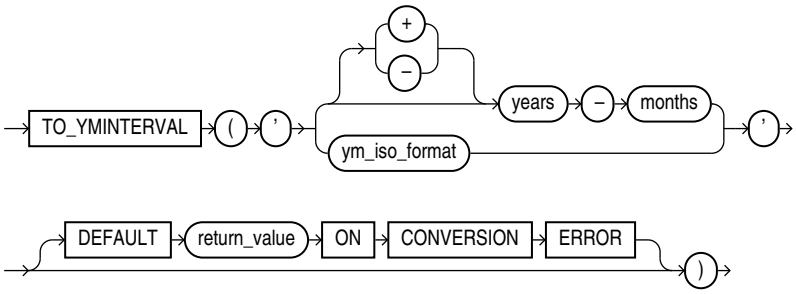
  ```
  INSERT INTO vecTab VALUES(TO_VECTOR('[1.1, 2.9, 3.14]'));
  ```

- Applications using Oracle Client 23ai libraries or Thin mode drivers can insert vector data directly as a string or a `CLOB`. For example:

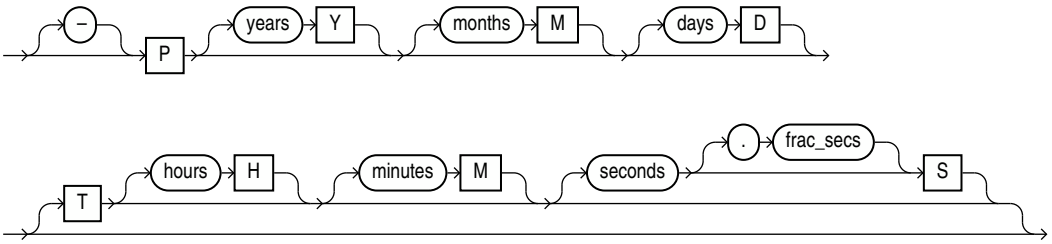
  ```
  INSERT INTO vecTab VALUES ('[1.1, 2.9, 3.14]');
  ```

TO_YMINTERVAL

Syntax



***ym_iso_format*::=**



Purpose

`TO_YMINTERVAL` converts its argument to a value of `INTERVAL MONTH TO YEAR` data type.

For the argument, you can specify any expression that evaluates to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type.

`TO_YMINTERVAL` accepts argument in one of the two formats:

- SQL interval format compatible with the SQL standard (ISO/IEC 9075)
- ISO duration format compatible with the ISO 8601:2004 standard

In the SQL format, *`years`* is an integer between 0 and 999999999, and *`months`* is an integer between 0 and 11. Additional blanks are allowed between format elements.

In the ISO format, years and months are integers between 0 and 999999999. Days, *`hours`*, *`minutes`*, *`seconds`*, and frac_secs are non-negative integers, and are ignored, if specified. No blanks are allowed in the value. If you specify `T`, then you must specify at least one of the *`hours`*, *`minutes`*, or *`seconds`* values.

The optional `DEFAULT` *`return_value`* `ON CONVERSION ERROR` clause allows you to specify the value this function returns if an error occurs while converting the argument to an `INTERVAL MONTH TO YEAR` type. This clause has no effect if an error occurs while evaluating the argument. The *`return_value`* can be an expression or a bind variable, and it must evaluate to a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type. It can be in either the SQL format or ISO format, and need not be in the same format as the function argument. If *`return_value`* cannot be converted to an `INTERVAL MONTH TO YEAR` type, then the function returns an error.

Examples

The following example calculates for each employee in the sample `hr.employees` table a date one year two months after the hire date:

```
SELECT hire_date, hire_date + TO_YMINTERVAL('01-02') "14 months"
   FROM employees;

HIRE_DATE 14 months
--------- ---------
17-JUN-03 17-AUG-04
21-SEP-05 21-NOV-06
13-JAN-01 13-MAR-02
20-MAY-08 20-JUL-09
21-MAY-07 21-JUL-08

. . .
```

The following example makes the same calculation using the ISO format:

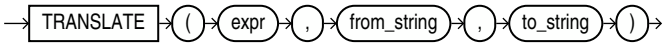
```
SELECT hire_date, hire_date + TO_YMINTERVAL('P1Y2M') FROM employees;
```

The following example returns the default value because the specified expression cannot be converted to an `INTERVAL MONTH TO YEAR` value:

```
SELECT TO_YMINTERVAL('1x-02'
       DEFAULT '00-00' ON CONVERSION ERROR) "Value"
  FROM DUAL;

Value
-------------
+000000000-00
```

TRANSLATE

Syntax



Purpose

`TRANSLATE` returns *`expr`* with all occurrences of each character in *`from_string`* replaced by its corresponding character in *`to_string`*. Characters in *`expr`* that are not in *`from_string`* are not replaced. The argument *`from_string`* can contain more characters than *`to_string`*. In this case, the extra characters at the end of *`from_string`* have no corresponding characters in *`to_string`*. If these extra characters appear in *`expr`*, then they are removed from the return value.

If a character appears multiple times in *`from_string`*, then the *`to_string`* mapping corresponding to the first occurrence is used.

You cannot use an empty string for *`to_string`* to remove all characters in *`from_string`* from the return value. Oracle Database interprets the empty string as null, and if this function has a null argument, then it returns null. To remove all characters in *`from_string`*, concatenate another character to the beginning of *`from_string`* and specify this character as the *`to_string`*. For example, `TRANSLATE`(*`expr`*, '`x0123456789`', '`x`') removes all digits from *`expr`*.

`TRANSLATE` provides functionality related to that provided by the `REPLACE` function. `REPLACE` lets you substitute a single string for another single string, as well as remove character strings. `TRANSLATE` lets you make several single-character, one-to-one substitutions in one operation.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> - "Data Type Comparison Rules " for more information and REPLACE
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `TRANSLATE` uses to compare characters from *`expr`* with characters from *`from_string`*, and for the collation derivation rules, which define the collation assigned to the character return value of `TRANSLATE`

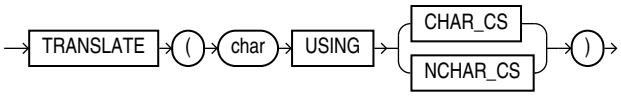
Examples

The following statement translates a book title into a string that could be used (for example) as a filename. The *`from_string`* contains four characters: a space, asterisk, slash, and apostrophe (with an extra apostrophe as the escape character). The *`to_string`* contains only three underscores. This leaves the fourth character in the *`from_string`* without a corresponding replacement, so apostrophes are dropped from the returned value.

```
SELECT TRANSLATE('SQL*Plus User''s Guide', ' */''', '___') FROM DUAL;

TRANSLATE('SQL*PLUSU
--------------------
SQL_Plus_Users_Guide
```

TRANSLATE ... USING

Syntax



Purpose

`TRANSLATE` ... `USING` converts *`char`* into the character set specified for conversions between the database character set and the national character set.

> **Note:**
>
> The TRANSLATE ... USING function is supported primarily for ANSI compatibility. Oracle recommends that you use the TO_CHAR and TO_NCHAR functions, as appropriate, for converting data to the database or national character set. TO_CHAR and TO_NCHAR can take as arguments a greater variety of data types than TRANSLATE ... USING, which accepts only character data.

The *char* argument is the expression to be converted.

- Specifying the USING CHAR_CS argument converts *char* into the database character set. The output data type is VARCHAR2.
- Specifying the USING NCHAR_CS argument converts *char* into the national character set. The output data type is NVARCHAR2.

This function is similar to the Oracle CONVERT function, but must be used instead of CONVERT if either the input or the output data type is being used as NCHAR or NVARCHAR2. If the input contains UCS2 code points or backslash characters (\), then use the UNISTR function.

> **See Also:**
>
> - CONVERT and UNISTR
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of TRANSLATE ... USING

Examples

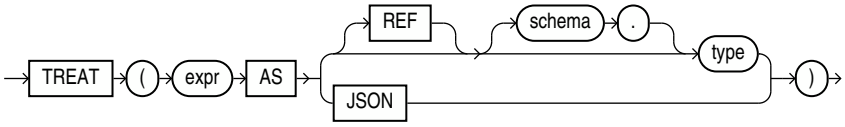
The following statements use data from the sample table oe.product_descriptions to show the use of the TRANSLATE ... USING function:

```
CREATE TABLE translate_tab (char_col  VARCHAR2(100),
                            nchar_col NVARCHAR2(50));
INSERT INTO translate_tab
   SELECT NULL, translated_name
      FROM product_descriptions
      WHERE product_id = 3501;

SELECT * FROM translate_tab;

CHAR_COL             NCHAR_COL
-------------------- --------------------------------------------------
. . .
                     C pre SPNIX4.0 - Sys
                     C pro SPNIX4.0 - Sys
                     C til SPNIX4.0 - Sys
                     C voor SPNIX4.0 - Sys
. . .

UPDATE translate_tab
   SET char_col = TRANSLATE (nchar_col USING CHAR_CS);
```

```
SELECT * FROM translate_tab;

CHAR_COL                 NCHAR_COL
------------------------ ------------------------
. . .
C per a SPNIX4.0 - Sys   C per a SPNIX4.0 - Sys
C pro SPNIX4.0 - Sys     C pro SPNIX4.0 - Sys
C for SPNIX4.0 - Sys     C for SPNIX4.0 - Sys
C til SPNIX4.0 - Sys     C til SPNIX4.0 - Sys
. . .
```

TREAT

Syntax



Purpose

You can use the `TREAT` function to change the declared type of an expression.

Use the keywords `AS JSON` when you want the expression to return JSON data. This is useful when you want to force some text to be interpreted as JSON data. For example, you can use it to interpret a `VARCHAR2` value of {} as an empty JSON object instead of a string.

You must have the `EXECUTE` object privilege on *`type`* to use this function.

- In *`expr AS JSON`*, *`expr`* is a SQL data type containing JSON, for example `CLOB`.
- In *`expr AS type`*, *`expr`* and *`type`* must be a user-defined object types, excluding top-level collections.
- *`type`* must be some supertype or subtype of the declared type of *`expr`*. If the most specific type of *`expr`* is *`type`* (or some subtype of *`type`*), then `TREAT` returns *`expr`*. If the most specific type of *`expr`* is not *`type`* (or some subtype of *`type`*), then `TREAT` returns `NULL`.
- You can specify `REF` only if the declared type of *`expr`* is a `REF` type.
- If the declared type of *`expr`* is a `REF` to a source type of *`expr`*, then *`type`* must be some subtype or supertype of the source type of *`expr`*. If the most specific type of `DEREF`(*`expr`*) is *`type`* (or a subtype of *`type`*), then `TREAT` returns *`expr`*. If the most specific type of `DEREF`(*`expr`*) is not *`type`* (or a subtype of *`type`*), then `TREAT` returns `NULL`.

> **See Also:**
>
> "Data Type Comparison Rules " for more information

Examples

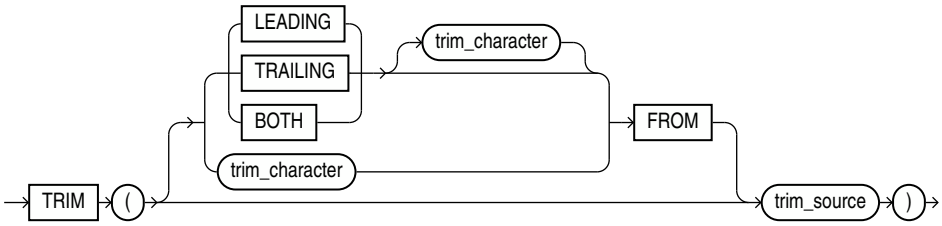
The following statement uses the table `oe.persons`, which is created in "Substitutable Table and Column Examples". The example retrieves the salary attribute of all people in the `persons` table, the value being null for instances of people that are not employees.

```
SELECT name, TREAT(VALUE(p) AS employee_t).salary salary
   FROM persons p;

NAME                          SALARY
----------------------- ----------
Bob
Joe                           100000
Tim                             1000
```

You can use the `TREAT` function to create an index on the subtype attributes of a substitutable column. For an example, see "Indexing on Substitutable Columns: Examples".

TRIM

Syntax



Purpose

`TRIM` enables you to trim leading or trailing characters (or both) from a character string. If *trim_character* or *trim_source* is a character literal, then you must enclose it in single quotation marks.

- If you specify `LEADING`, then Oracle Database removes any leading characters equal to *trim_character*.
- If you specify `TRAILING`, then Oracle removes any trailing characters equal to *trim_character*.
- If you specify `BOTH` or none of the three, then Oracle removes leading and trailing characters equal to *trim_character*.
- If you do not specify *trim_character*, then the default value is a blank space.
- If you specify only *trim_source*, then Oracle removes leading and trailing blank spaces.
- The function returns a value with data type `VARCHAR2`. The maximum length of the value is the length of *trim_source*.
- If either *trim_source* or *trim_character* is null, then the `TRIM` function returns null.

Both *trim_character* and *trim_source* can be `VARCHAR2` or any data type that can be implicitly converted to `VARCHAR2`. The string returned is a `VARCHAR2` (`NVARCHAR2`) data type if *trim_source* is a `CHAR` or `VARCHAR2` (`NCHAR` or `NVARCHAR2`) data type, and a `CLOB` if *trim_source* is a `CLOB` data type. The return string is in the same character set as *trim_source*.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation determination rules, which define the collation `TRIM` uses to compare characters from *`trim_character`* with characters from *`trim_source`*, and for the collation derivation rules, which define the collation assigned to the character return value of this function

Examples

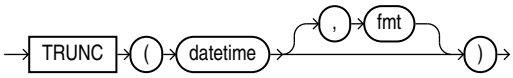
This example trims leading zeros from the hire date of the employees in the `hr` schema:

```
SELECT employee_id,
      TO_CHAR(TRIM(LEADING 0 FROM hire_date))
      FROM employees
      WHERE department_id = 60
      ORDER BY employee_id;

EMPLOYEE_ID TO_CHAR(T
----------- ---------
        103 20-MAY-08
        104 21-MAY-07
        105 25-JUN-05
        106 5-FEB-06
        107 7-FEB-07
```

TRUNC (datetime)

Syntax

***trunc_datetime*::=**



Purpose

The `TRUNC` (datetime) function returns *`date`* with the time portion of the day truncated to the unit specified by the format model *`fmt`*.

This function is not sensitive to the `NLS_CALENDAR` session parameter. It operates according to the rules of the Gregorian calendar. The value returned is always of data type `DATE`, even if you specify a different datetime data type for date. If you do not specify the second argument *`fmt`*, then the default format model '`DD`' is used and the value returned is *`date`* truncated to the day with a time of midnight.

The `TRUNC` and `FLOOR` functions are synonymous for dates and timestamps.

Refer to "CEIL, FLOOR, ROUND, and TRUNC Date Functions" for the permitted format models to use in *`fmt`*.

Examples

The following example truncates a date:

```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR')
  "New Year" FROM DUAL;

New Year
---------
01-JAN-92
```

Formatting Dates using TRUNC: Examples

In the following example, the `TRUNC` function returns the input date with the time portion of the day truncated as specified in the format model:

```
WITH dates AS (
  SELECT date'2015-01-01' d FROM dual union
  SELECT date'2015-01-10' d FROM dual union
  SELECT date'2015-02-01' d FROM dual union
  SELECT timestamp'2015-03-03 23:45:00' d FROM dual union
  SELECT timestamp'2015-04-11 12:34:56' d FROM dual
)
SELECT d "Original Date",
       trunc(d) "Nearest Day, Time Removed",
       trunc(d, 'ww') "Nearest Week",
       trunc(d, 'iw') "Start of Week",
       trunc(d, 'mm') "Start of Month",
       trunc(d, 'year') "Start of Year"
FROM dates;
```

In the following example, the input date values are truncated and the `TO_CHAR` function is used to obtain the minute component of the truncated date values:

```
WITH dates AS (
  SELECT date'2015-01-01' d FROM dual union
  SELECT date'2015-01-10' d FROM dual union
  SELECT date'2015-02-01' d FROM dual union
  SELECT timestamp'2015-03-03 23:45:00' d FROM dual union
  SELECT timestamp'2015-04-11 12:34:56' d FROM dual
)
SELECT d "Original Date",
       trunc(d) "Date with Time Removed",
       to_char(trunc(d, 'mi'), 'dd-mon-yyyy hh24:mi') "Nearest Minute",
       trunc(d, 'iw') "Start of Week",
       trunc(d, 'mm') "Start of Month",
       trunc(d, 'year') "Start of Year"
FROM dates;
```

The following statement alters the date format for the current session:

```
ALTER SESSION SET nls_date_format = 'dd-mon-yyyy hh24:mi';
```

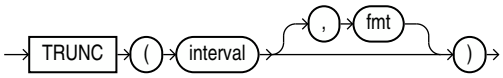
In the following example, the data is displayed in the new date format:

```
WITH dates AS (
  SELECT date'2015-01-01' d FROM dual union
  SELECT date'2015-01-10' d FROM dual union
  SELECT date'2015-02-01' d FROM dual union
  SELECT timestamp'2015-03-03 23:44:32' d FROM dual union
  SELECT timestamp'2015-04-11 12:34:56' d FROM dual
)
SELECT d "Original Date",
       trunc(d) "Date, time removed",
       to_char(trunc(d, 'mi'), 'dd-mon-yyyy hh24:mi') "Nearest Minute",
       trunc(d, 'iw') "Start of Week",
       trunc(d, 'mm') "Start of Month",
```

```
      trunc(d, 'year') "Start of Year"
FROM dates;
```

TRUNC (interval)

Syntax



Purpose

`TRUNC(interval)` returns the interval rounded down to the unit specified by the second argument *`fmt`*, the format model .

The absolute value of `TRUNC(interval)` is never greater than the absolute value of *`interval`*. The result precision is the same as the input precision, since there is no overflow issue for `TRUNC(interval)`.

For `INTERVAL YEAR TO MONTH`, *`fmt`* can only be year. The default *`fmt`* is year.

For `INTERVAL DAY TO SECOND`, *`fmt`* can be day, hour and minute. The default *`fmt`* is day. Note that *`fmt`* does not support second.

> **See Also:**
>
> Refer to CEIL, FLOOR, ROUND, and TRUNC Date Functions for the permitted format models to use in *`fmt`*.

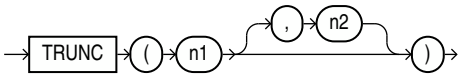
Examples

```
SELECT TRUNC(INTERVAL '+123-06' YEAR(3) TO MONTH) AS year_trunc;

YEAR_TRUNC
----------
+123-00


SELECT TRUNC(INTERVAL '+99-11' YEAR(2) TO MONTH, 'YEAR') AS year_trunc;

YEAR_TRUNC
----------
+99-00


SELECT TRUNC(INTERVAL '+4 12:42:10.222' DAY(2) TO SECOND(3), 'DD') AS day_trunc;

DAY_TRUNC
-------------------
+04 00:00:00.000000
```

TRUNC (number)

Syntax

***trunc_number*::=**



Purpose

The `TRUNC` (number) function returns *n1* truncated to *n2* decimal places. If *n2* is omitted, then *n1* is truncated to 0 places. *n2* can be negative to truncate (make zero) *n2* digits left of the decimal point.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. If you omit *n2*, then the function returns the same data type as the numeric data type of the argument. If you include *n2*, then the function returns `NUMBER`.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

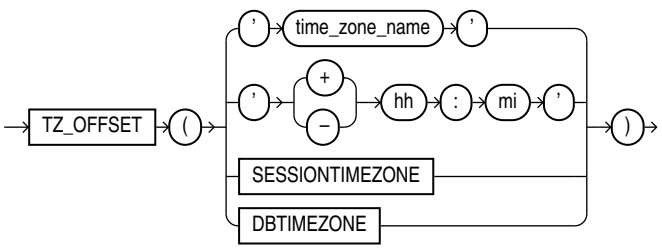
Examples

The following examples truncate numbers:

```
SELECT TRUNC(15.79,1) "Truncate" FROM DUAL;

  Truncate
----------
      15.7

SELECT TRUNC(15.79,-1) "Truncate" FROM DUAL;

  Truncate
----------
        10
```

TZ_OFFSET

Syntax



Purpose

`TZ_OFFSET` returns the time zone offset corresponding to the argument based on the date the statement is executed. You can enter a valid time zone region name, a time zone offset from UTC (which simply returns itself), or the keyword `SESSIONTIMEZONE` or `DBTIMEZONE`. For a listing of valid values for *`time_zone_name`*, query the `TZNAME` column of the `V$TIMEZONE_NAMES` dynamic performance view.

> **Note:**
>
> Time zone region names are needed by the daylight saving feature. These names are stored in two types of time zone files: one large and one small. One of these files is the default file, depending on your environment and the release of Oracle Database you are using. For more information regarding time zone files and names, see *Oracle Database Globalization Support Guide*.

> **See Also:**
>
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `TZ_OFFSET`

Examples

The following example returns the time zone offset of the US/Eastern time zone from UTC:

```
SELECT TZ_OFFSET('US/Eastern') FROM DUAL;

TZ_OFFS
-------
-04:00
```

UID

Syntax

Purpose

`UID` returns an integer that uniquely identifies the session user (the user who logged on).

> **See Also:**
>
> USER to learn how Oracle Database determines the session user

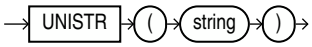
Examples

The following example returns the UID of the session user:

```
SELECT UID FROM DUAL;
```

UNISTR

Syntax



Purpose

`UNISTR` takes as its argument a text literal or an expression that resolves to character data and returns it in the national character set. The national character set of the database can be either AL16UTF16 or UTF8. `UNISTR` provides support for Unicode string literals by letting you specify the Unicode encoding value of characters in the string. This is useful, for example, for inserting data into `NCHAR` columns.

The Unicode encoding value has the form '\xxxx' where 'xxxx' is the hexadecimal value of a character in UCS-2 encoding format. Supplementary characters are encoded as two code units, the first from the high-surrogates range (U+D800 to U+DBFF), and the second from the low-surrogates range (U+DC00 to U+DFFF). To include the backslash in the string itself, precede it with another backslash (\\).

For portability and data preservation, Oracle recommends that in the `UNISTR` string argument you specify only ASCII characters and the Unicode encoding values.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for information on Unicode and national character sets
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `UNISTR`

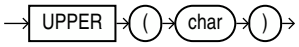
Examples

The following example passes both ASCII characters and Unicode encoding values to the `UNISTR` function, which returns the string in the national character set:

```
SELECT UNISTR('abc\00e5\00f1\00f6') FROM DUAL;

UNISTR
------
abcåñö
```

UPPER

Syntax



Purpose

`UPPER` returns *`char`*, with all letters uppercase. *`char`* can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The return value is the same data type as *`char`*. The database sets the case of the characters based on the binary mapping defined for the underlying character set. For linguistic-sensitive uppercase, refer to NLS_UPPER .

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `UPPER`

Examples

The following example returns each employee's last name in uppercase:

```
SELECT UPPER(last_name) "Uppercase"
   FROM employees;
```

USER

Syntax

Purpose

`USER` returns the name of the session user (the user who logged on). This may change during the duration of a database session as Real Application Security sessions are attached or detached. If a Real Application Security session is currently attached to the database session, then it returns user `XS$NULL`.

This function returns a `VARCHAR2` value.

Oracle Database compares values of this function with blank-padded comparison semantics.

In a distributed SQL statement, the `UID` and `USER` functions together identify the user on your local database. You cannot use these functions in the condition of a `CHECK` constraint.

> **See Also:**
>
> Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `USER`

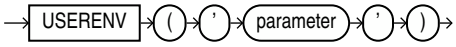
Examples

The following example returns the session user and the user's UID:

```
SELECT USER, UID FROM DUAL;
```

USERENV

Syntax



Purpose

> **Note:**
>
> `USERENV` is a legacy function that is retained for backward compatibility. Oracle recommends that you use the `SYS_CONTEXT` function with the built-in `USERENV` namespace for current functionality. See SYS_CONTEXT for more information.

`USERENV` returns information about the current session. This information can be useful for writing an application-specific audit trail table or for determining the language-specific characters currently used by your session. You cannot use `USERENV` in the condition of a `CHECK` constraint. Table 7-12 describes the values for the *`parameter`* argument.

All calls to `USERENV` return `VARCHAR2` data except for calls with the `SESSIONID`, `SID`, and `ENTRYID` parameters, which return `NUMBER`.

Table 7-12 Parameters of the USERENV Function

Parameter	Return Value
`CLIENT_INFO`	`CLIENT_INFO` returns up to 64 bytes of user session information that can be stored by an application using the `DBMS_APPLICATION_INFO` package. **Caution:** Some commercial applications may be using this context value. Refer to the applicable documentation for those applications to determine what restrictions they may impose on use of this context area. **See Also:** *Oracle Database Security Guide* for more information on application context, CREATE CONTEXT, and SYS_CONTEXT

Table 7-12 (Cont.) Parameters of the USERENV Function

Parameter	Return Value
ENTRYID	The current audit entry number. The audit entryid sequence is shared between fine-grained audit records and regular audit records. You cannot use this attribute in distributed SQL statements.
ISDBA	ISDBA returns 'TRUE' if the user has been authenticated as having DBA privileges either through the operating system or through a password file.
LANG	LANG returns the ISO abbreviation for the language name, a shorter form than the existing 'LANGUAGE' parameter.
LANGUAGE	LANGUAGE returns the language and territory used by the current session along with the database character set in this form: language_territory.characterset
SESSIONID	SESSIONID returns the auditing session identifier. You cannot specify this parameter in distributed SQL statements.
SID	SID returns the session ID.
TERMINAL	TERMINAL returns the operating system identifier for the terminal of the current session. In distributed SQL statements, this parameter returns the identifier for your local session. In a distributed environment, this parameter is supported only for remote SELECT statements, not for remote INSERT, UPDATE, or DELETE operations.

See Also:

Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of USERENV

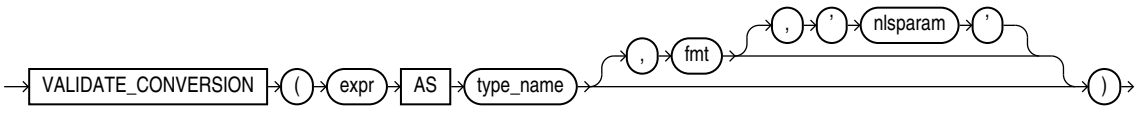
Examples

The following example returns the LANGUAGE parameter of the current session:

```
SELECT USERENV('LANGUAGE') "Language" FROM DUAL;

Language
----------------------------------
AMERICAN_AMERICA.WE8ISO8859P1
```

VALIDATE_CONVERSION

Syntax



Purpose

VALIDATE_CONVERSION determines whether *expr* can be converted to the specified data type. If *expr* can be successfully converted, then this function returns 1; otherwise, this function returns 0. If *expr* evaluates to null, then this function returns 1. If an error occurs while evaluating *expr*, then this function returns the error.

For *expr*, specify a SQL expression. The acceptable data types for *expr*, and the purpose of the optional *fmt* and *nlsparam* arguments, depend on the data type you specify for *type_name*.

For *type_name*, specify the data type to which you want to convert *expr*. You can specify the following data types:

- BINARY_DOUBLE

  If you specify BINARY_DOUBLE, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, or a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_BINARY_DOUBLE function. Refer to TO_BINARY_DOUBLE for more information.

- BINARY_FLOAT

  If you specify BINARY_FLOAT, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, or a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_BINARY_FLOAT function. Refer to TO_BINARY_FLOAT for more information.

- DATE

  If you specify DATE, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_DATE function. Refer to TO_DATE for more information.

- INTERVAL DAY TO SECOND

  If you specify INTERVAL DAY TO SECOND, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, and must contain a value in either the SQL interval format or the ISO duration format. The optional *fmt* and *nlsparam* arguments do not apply for this data type. Refer to TO_DSINTERVAL for more information on the SQL interval format and the ISO duration format.

- INTERVAL YEAR TO MONTH

  If you specify INTERVAL YEAR TO MONTH, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, and must contain a value in either the SQL interval format or the ISO duration format. The optional *fmt* and *nlsparam* arguments do not apply for this data type. Refer to TO_YMINTERVAL for more information on the SQL interval format and the ISO duration format.

- NUMBER

  If you specify NUMBER, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type, a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE or value of type BOOLEAN. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_NUMBER function. Refer to TO_NUMBER for more information.

If *expr* is a value of type NUMBER, then the VALIDATE_CONVERSION function verifies that *expr* is a legal numeric value. If *expr* is not a legal numeric value, then the function returns 0. This enables you to identify corrupt numeric values in your database.

- TIMESTAMP

  If you specify TIMESTAMP, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_TIMESTAMP function. If you omit *fmt*, then *expr* must be in the default format of the TIMESTAMP data type, which is determined by the NLS_TIMESTAMP_FORMAT initialization parameter. Refer to TO_TIMESTAMP for more information.

- TIMESTAMP WITH TIME ZONE

  If you specify TIMESTAMP WITH TIME ZONE, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_TIMESTAMP_TZ function. If you omit *fmt*, then *expr* must be in the default format of the TIMESTAMP WITH TIME ZONE data type, which is determined by the NLS_TIMESTAMP_TZ_FORMAT initialization parameter. Refer to TO_TIMESTAMP_TZ for more information.

- TIMESTAMP WITH LOCAL TIME ZONE

  If you specify TIMESTAMP, then *expr* can be any expression that evaluates to a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The optional *fmt* and *nlsparam* arguments serve the same purpose as for the TO_TIMESTAMP function. If you omit *fmt*, then *expr* must be in the default format of the TIMESTAMP data type, which is determined by the NLS_TIMESTAMP_FORMAT initialization parameter. Refer to TO_TIMESTAMP for more information.

- BOOLEAN

  BOOLEAN is supported as a target type. It supports NUMBER type family, VARCHAR type family, and BOOLEAN itself as input.

Examples

In each of the following statements, the specified value can be successfully converted to the specified data type. Therefore, each of these statements returns a value of 1.

```
SELECT VALIDATE_CONVERSION(1000 AS BINARY_DOUBLE)
  FROM DUAL;

SELECT VALIDATE_CONVERSION('1234.56' AS BINARY_FLOAT)
  FROM DUAL;

SELECT VALIDATE_CONVERSION('July 20, 1969, 20:18' AS DATE,
    'Month dd, YYYY, HH24:MI', 'NLS_DATE_LANGUAGE = American')
  FROM DUAL;

SELECT VALIDATE_CONVERSION('200 00:00:00' AS INTERVAL DAY TO SECOND)
  FROM DUAL;

SELECT VALIDATE_CONVERSION('P1Y2M' AS INTERVAL YEAR TO MONTH)
  FROM DUAL;

SELECT VALIDATE_CONVERSION('$100,00' AS NUMBER,
    '$999D99', 'NLS_NUMERIC_CHARACTERS = '',.''')
  FROM DUAL;
```

```
SELECT VALIDATE_CONVERSION('29-Jan-02 17:24:00' AS TIMESTAMP,
    'DD-MON-YY HH24:MI:SS')
  FROM DUAL;

SELECT VALIDATE_CONVERSION('1999-12-01 11:00:00 -8:00'
    AS TIMESTAMP WITH TIME ZONE, 'YYYY-MM-DD HH:MI:SS TZH:TZM')
  FROM DUAL;

SELECT VALIDATE_CONVERSION('11-May-16 17:30:00'
    AS TIMESTAMP WITH LOCAL TIME ZONE, 'DD-MON-YY HH24:MI:SS')
  FROM DUAL;
```

The following statement returns 0, because the specified value cannot be converted to `BINARY_FLOAT`:

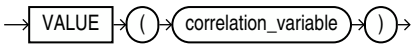
```
SELECT VALIDATE_CONVERSION('$29.99' AS BINARY_FLOAT)
  FROM DUAL;
```

The following statement returns 1, because the specified number format model enables the value to be converted to `BINARY_FLOAT`:

```
SELECT VALIDATE_CONVERSION('$29.99' AS BINARY_FLOAT, '$99D99')
  FROM DUAL;
```

VALUE

Syntax



Purpose

`VALUE` takes as its argument a correlation variable (table alias) associated with a row of an object table and returns object instances stored in the object table. The type of the object instances is the same type as the object table.

Examples

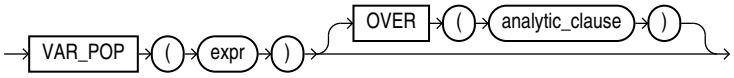
The following example uses the sample table `oe.persons`, which is created in "Substitutable Table and Column Examples":

```
SELECT VALUE(p) FROM persons p;

VALUE(P)(NAME, SSN)
-------------------------------------------------------------
PERSON_T('Bob', 1234)
EMPLOYEE_T('Joe', 32456, 12, 100000)
PART_TIME_EMP_T('Tim', 5678, 13, 1000, 20)
```

> **See Also:**
>
> "IS OF *type* Condition " for information on using `IS OF` type conditions with the `VALUE` function

VAR_POP

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`VAR_POP` returns the population variance of a set of numbers after discarding the nulls in this set. You can use it as both an aggregate and analytic function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

If the function is applied to an empty set, then it returns null. The function makes the following calculation:

```
SUM((expr - (SUM(expr) / COUNT(expr)))2) / COUNT(expr)
```

> **See Also:**
>
> "About SQL Expressions " for information on valid forms of *expr* and "Aggregate Functions "

Aggregate Example

The following example returns the population variance of the salaries in the `employees` table:

```
SELECT VAR_POP(salary) FROM employees;

VAR_POP(SALARY)
---------------
     15141964.9
```

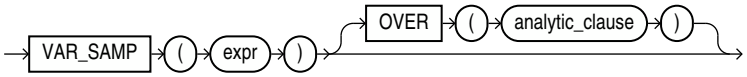
Analytic Example

The following example calculates the cumulative population and sample variances in the `sh.sales` table of the monthly sales in 1998:

```
SELECT t.calendar_month_desc,
   VAR_POP(SUM(s.amount_sold))
      OVER (ORDER BY t.calendar_month_desc) "Var_Pop",
   VAR_SAMP(SUM(s.amount_sold))
      OVER (ORDER BY t.calendar_month_desc) "Var_Samp"
  FROM sales s, times t
  WHERE s.time_id = t.time_id AND t.calendar_year = 1998
  GROUP BY t.calendar_month_desc
  ORDER BY t.calendar_month_desc, "Var_Pop", "Var_Samp";

CALENDAR    Var_Pop   Var_Samp
-------- ---------- ----------
1998-01           0
1998-02  2269111326 4538222653
1998-03  5.5849E+10 8.3774E+10
1998-04  4.8252E+10 6.4336E+10
1998-05  6.0020E+10 7.5025E+10
1998-06  5.4091E+10 6.4909E+10
1998-07  4.7150E+10 5.5009E+10
1998-08  4.1345E+10 4.7252E+10
1998-09  3.9591E+10 4.4540E+10
1998-10  3.9995E+10 4.4439E+10
1998-11  3.6870E+10 4.0558E+10
1998-12  4.0216E+10 4.3872E+10
```

VAR_SAMP

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`VAR_SAMP` returns the sample variance of a set of numbers after discarding the nulls in this set. You can use it as both an aggregate and analytic function.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

If the function is applied to an empty set, then it returns null. The function makes the following calculation:

```
(SUM(expr - (SUM(expr) / COUNT(expr)))2) / (COUNT(expr) - 1)
```

This function is similar to `VARIANCE`, except that given an input set of one element, `VARIANCE` returns 0 and `VAR_SAMP` returns null.

> **See Also:**
>
> "About SQL Expressions " for information on valid forms of *expr* and "Aggregate Functions "

Aggregate Example

The following example returns the sample variance of the salaries in the sample `employees` table.

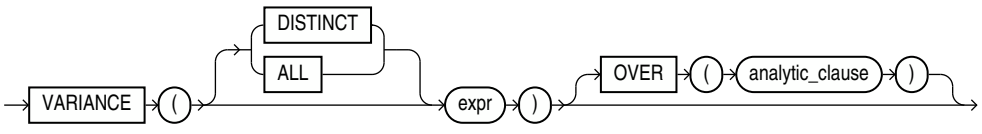
```
SELECT VAR_SAMP(salary) FROM employees;

VAR_SAMP(SALARY)
----------------
      15284813.7
```

Analytic Example

Refer to the analytic example for VAR_POP .

VARIANCE

Syntax



> **See Also:**
>
> "Analytic Functions " for information on syntax, semantics, and restrictions

Purpose

`VARIANCE` returns the variance of *expr*. You can use it as an aggregate or analytic function.

Oracle Database calculates the variance of *expr* as follows:

- 0 if the number of rows in *expr* = 1
- VAR_SAMP if the number of rows in *expr* > 1

If you specify DISTINCT, then you can specify only the *query_partition_clause* of the *analytic_clause*. The *order_by_clause* and *windowing_clause* are not allowed.

This function takes as an argument any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. The function returns the same data type as the numeric data type of the argument.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion, "About SQL Expressions " for information on valid forms of *expr* and "Aggregate Functions "

Aggregate Example

The following example calculates the variance of all salaries in the sample employees table:

```
SELECT VARIANCE(salary) "Variance"
   FROM employees;

  Variance
----------
15283140.5
```

Analytic Example

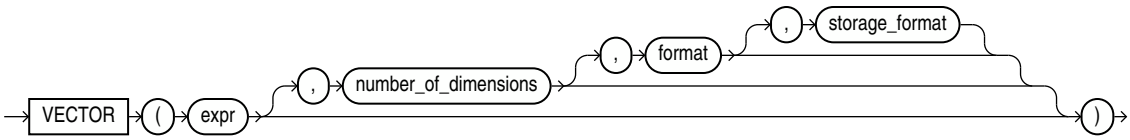
The following example returns the cumulative variance of salary values in Department 30 ordered by hire date.

```
SELECT last_name, salary, VARIANCE(salary)
      OVER (ORDER BY hire_date) "Variance"
   FROM employees
   WHERE department_id = 30
   ORDER BY last_name, salary, "Variance";

LAST_NAME                     SALARY   Variance
------------------------- ---------- ----------
Baida                           2900 16283333.3
Colmenares                      2500   11307000
Himuro                          2600   13317000
Khoo                            3100   31205000
Raphaely                       11000          0
Tobias                          2800 21623333.3
```

VECTOR

`VECTOR` is synonymous with `TO_VECTOR`.

Syntax



Purpose

See TO_VECTOR for semantics and examples.

> **Note:**
>
> Applications using Oracle Client 23ai libraries or Thin mode drivers can insert vector data directly as a string or a `CLOB`. For example:
>
> ```
> INSERT INTO vecTab VALUES ('[1.1, 2.9, 3.14]');
> ```

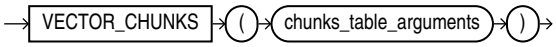
Examples

```
SELECT VECTOR('[34.6, 77.8]');

VECTOR('[34.6,77.8]')
------------------------------------------------------------
[3.45999985E+001,7.78000031E+001]


SELECT VECTOR('[34.6, 77.8]', 2, FLOAT32);

VECTOR('[34.6,77.8]',2,FLOAT32)
------------------------------------------------------------
[3.45999985E+001,7.78000031E+001]


SELECT VECTOR('[34.6, 77.8, -89.34]', 3, FLOAT32);

VECTOR('[34.6,77.8,-89.34]',3,FLOAT32)
------------------------------------------------------------------
[3.45999985E+001,7.78000031E+001,-8.93399963E+001]
```

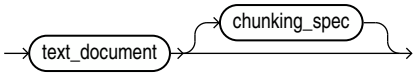
VECTOR_CHUNKS

Use `VECTOR_CHUNKS` to split plain text into smaller chunks to generate vector embeddings that can be used with vector indexes or hybrid vector indexes.

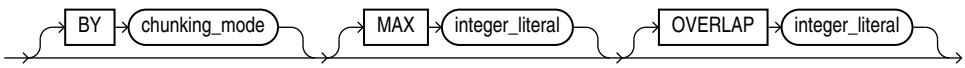
Syntax

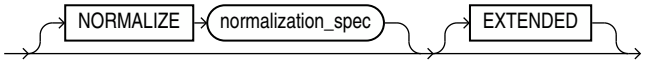


chunks_table_arguments::=

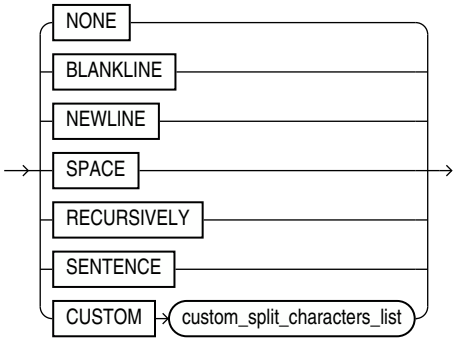


chunking_spec::=

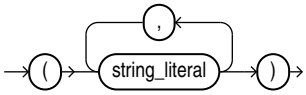




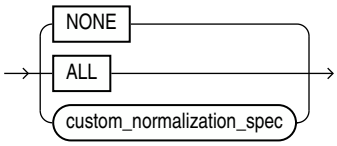
split_characters_list::=



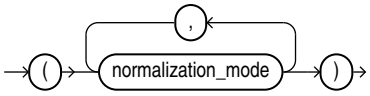
custom_split_characters_list



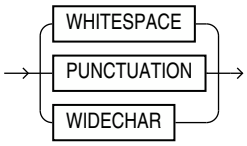
normalization_spec



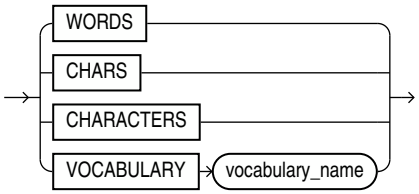
custom_normalization_spec



normalization_mode



***chunking_mode*::=**



Purpose

`VECTOR_CHUNKS` takes a character value as the *`text_document`* argument and splits it into chunks using a process controlled by the chunking parameters given in the optional *`chunking_spec`*. The chunks are returned as rows of a virtual relational table. Therefore, `VECTOR_CHUNKS` can only appear in the `FROM` clause of a subquery.

The returned virtual table has the following columns:

- `CHUNK_OFFSET` of data type `NUMBER` is the position of each chunk in the source document, relative to the start of the document, which has a position of 1.
- `CHUNK_LENGTH` of data type `NUMBER` is the length of each chunk.
- `CHUNK_TEXT` is a segment of text that has been split off from *`text_document`*.

The data type of the `CHUNK_TEXT` column and the length unit used by the values of `CHUNK_OFFSET` and `CHUNK_LENGTH` depend on the data type of *`text_document`* as listed in the following table:

Table 7-13 Input and Output Data Type Details

Input Data Type	Output Data Type	Offset and Length Unit
VARCHAR2	VARCHAR2	byte
CHAR	VARCHAR2	byte
CLOB	VARCHAR2	character
NVARCHAR2	NVARCHAR2	byte
NCHAR	NVARCHAR2	byte
NCLOB	NVARCHAR2	character

Note:

- For more information about data types, see *Data Types* in the SQL Reference Manual.
- The `VARCHAR2` input data type is limited to `4000` bytes unless the `MAX_STRING_SIZE` parameter is set to `EXTENDED`, which increases the limit to `32767`.

Parameters

All chunking parameters are optional, and the default chunking specifications are automatically applied to your chunk data.

When specifying chunking parameters for this API, ensure that you provide these parameters only in the listed order.

Table 7-14 Chunking Parameters Table

Parameter	Description and Acceptable Values
BY	Specifies the mode for splitting your data, that is, to split by counting the number of characters, words, or vocabulary tokens. **Valid values**: • CHARACTERS (or CHARS): Splits by counting the number of characters. • WORDS: Splits by counting the number of words. Words are defined as sequences of alphabetic characters, sequences of digits, individual punctuation marks, or symbols. For segmented languages without whitespace word boundaries (such as Chinese, Japanese, or Thai), each native character is considered a word (that is, unigram). • VOCABULARY: Splits by counting the number of vocabulary tokens. Vocabulary tokens are words or word pieces, recognized by the vocabulary of the tokenizer that your embedding model uses. You can load your vocabulary file using the VECTOR_CHUNKS helper API DBMS_VECTOR_CHAIN.CREATE_VOCABULARY. **Note**: For accurate results, ensure that the chosen model matches the vocabulary file used for chunking. If you are not using a vocabulary file, then ensure that the input length is defined within the token limits of your model. **Default value**: WORDS
MAX	Specifies a limit on the maximum size of each chunk. This setting splits the input text at a fixed point where the maximum limit occurs in the larger text. The units of MAX correspond to the BY mode, that is, to split data when it reaches the maximum size limit of a certain number of characters, words, numbers, punctuation marks, or vocabulary tokens. **Valid values**: • BY CHARACTERS: 50 to 4000 characters • BY WORDS: 10 to 1000 words • BY VOCABULARY: 10 to 1000 tokens **Default value**: 100

Table 7-14 (Cont.) Chunking Parameters Table

Parameter	Description and Acceptable Values
SPLIT [BY]	Specifies where to split the input text when it reaches the maximum size limit. This helps to keep related data together by defining appropriate boundaries for chunks. **Valid values**: • NONE: Splits at the MAX limit of characters, words, or vocabulary tokens. • NEWLINE, BLANKLINE, and SPACE: These are single-split character conditions that split at the last split character before the MAX value. Use NEWLINE to split at the end of a line of text. Use BLANKLINE to split at the end of a blank line (sequence of characters, such as two newlines). Use SPACE to split at the end of a blank space. • RECURSIVELY: This is a multiple-split character condition that breaks the input text using an ordered list of characters (or sequences). RECURSIVELY is predefined as BLANKLINE, NEWLINE, SPACE, NONE in this order: 1. If the input text is more than the MAX value, then split by the first split character. 2. If that fails, then split by the second split character. 3. And so on. 4. If no split characters exist, then split by MAX wherever it appears in the text. • SENTENCE: This is an end-of-sentence split condition that breaks the input text at a sentence boundary. This condition automatically determines sentence boundaries by using knowledge of the input language's sentence punctuation and contextual rules. This language-specific condition relies mostly on end-of-sentence (EOS) punctuations and common abbreviations. Contextual rules are based on word information, so this condition is only valid when splitting the text by words or vocabulary (not by characters). **Note**: This condition obeys the BY WORD and MAX settings, and thus may not determine accurate sentence boundaries in some cases. For example, when a sentence is larger than the MAX value, it splits the sentence at MAX. Similarly, it includes multiple sentences in the text only when they fit within the MAX limit. • CUSTOM: Splits based on a custom list of characters strings, for example, markup tags. You can provide custom sequences up to a limit of 16 split character strings, with a maximum length of 10 bytes each. Provide valid text literals as follows: `VECTOR_CHUNKS(c. doc, BY character SPLIT CUSTOM ('<html>' , '</html>')) vc` **Default value**: RECURSIVELY
OVERLAP	Specifies the amount (as a positive integer literal or zero) of the preceding text that the chunk should contain, if any. This helps in logically splitting up related text (such as a sentence) by including some amount of the preceding chunk text. The amount of overlap depends on how the maximum size of the chunk is measured (in characters, words, or vocabulary tokens). The overlap begins at the specified SPLIT condition (for example, at NEWLINE). **Valid value**: 5% to 20% of MAX **Default value**: 0

Table 7-14 (Cont.) Chunking Parameters Table

Parameter	Description and Acceptable Values

LANGUAGE	Specifies the language of your input data.
	This clause is important, especially when your text contains certain characters (for example, punctuations or abbreviations) that may be interpreted differently in another language.
	Valid values:
	• NLS-supported language name or its abbreviation, as listed in *Oracle Database Globalization Support Guide*.
	• Custom language name or its abbreviation, as listed in Supported Languages and Data File Locations. You use the DBMS_VECTOR_CHAIN.CREATE_LANG_DATA chunker helper API to load language-specific data (abbreviation tokens) into the database, for your specified language.
	You must use double quotation marks (") for any language name with spaces. For example:
	`LANGUAGE "simplified chinese"`
	For one-word language names, quotation marks are not needed. For example:
	`LANGUAGE american`
	Default value: NLS_LANGUAGE from session

NORMALIZE	Automatically pre-processes or post-processes issues (such as multiple consecutive spaces and smart quotes) that may arise when documents are converted into text. Oracle recommends you to use a normalization mode to extract high-quality chunks. **Valid values**: • NONE: Applies no normalization. • ALL: Normalizes multi-byte (Unicode) punctuation to standard single-byte. • Applies all supported normalization modes: PUNCTUATION, WHITESPACE, and WIDECHAR. – PUNCTUATION: Converts quotes, dashes, and other punctuation characters supported in the character set of the text to their common ASCII form. For example: * U+2013 (En Dash) maps to U+002D (Hyphen-Minus) * U+2018 (Left Single Quotation Mark) maps to U+0027 (Apostrophe) * U+2019 (Right Single Quotation Mark) maps to U+0027 (Apostrophe) * U+201B (Single High-Reversed-9 Quotation Mark) maps to U+0027 (Apostrophe) – WHITESPACE: Minimizes whitespace by eliminating unnecessary characters. For example, retain blanklines, but remove any extra newlines and interspersed spaces or tabs: `" \n \n " => "\n\n"` – WIDECHAR: Normalizes wide, multi-byte digits and (a-z) letters to single-byte. These are multi-byte equivalents for 0-9 and a-z A-Z, which can show up in Chinese, Japanese, or Korean text. **Default value**: NONE
EXTENDED	Increases the output limit of a VARCHAR2 string to 32767 bytes, without requiring you to set the MAX_STRING_SIZE parameter to EXTENDED. If EXTENDED is present in *chunking_spec*, the maximum length of a CHUNK_TEXT column value is 32767 bytes. If it is absent, the maximum length is 4000 bytes if MAX_STRING_SIZE is set to STANDARD and 32767 bytes if MAX_STRING_SIZE is set to EXTENDED.

Examples

VECTOR_CHUNKS can be called for a single character value provided in a character literal or a bind variable as shown in the following example:

```
COLUMN chunk_offset HEADING Offset FORMAT 999
COLUMN chunk_length HEADING Len    FORMAT 999
COLUMN chunk_text   HEADING Text   FORMAT a60

VARIABLE txt VARCHAR2(4000)
EXECUTE :txt := 'An example text value to split with VECTOR_CHUNKS, having over 10 words
because the minimum MAX value is 10';

SELECT * FROM VECTOR_CHUNKS(:txt BY WORDS MAX 10);

SELECT * FROM VECTOR_CHUNKS('Another example text value to split with VECTOR_CHUNKS,
having over 10 words because the minimum MAX value is 10' BY WORDS MAX 10);
```

To chunk values of a table column, the table needs to be joined with the VECTOR_CHUNKS call using left correlation as shown in the following example:

```
CREATE TABLE documentation_tab (
  id   NUMBER,
  text VARCHAR2(2000));

INSERT INTO documentation_tab
   VALUES(1, 'sample');

COMMIT;

SET LINESIZE 100;
SET PAGESIZE 20;
COLUMN pos FORMAT 999;
COLUMN siz FORMAT 999;
COLUMN txt FORMAT a60;

PROMPT SQL VECTOR_CHUNKS
SELECT D.id id, C.chunk_offset pos, C.chunk_length siz, C.chunk_text txt
FROM documentation_tab D, VECTOR_CHUNKS(D.text
                                BY words
                                MAX 200
                                OVERLAP 10
                                SPLIT BY recursively
                                LANGUAGE american
                                NORMALIZE all) C;
```

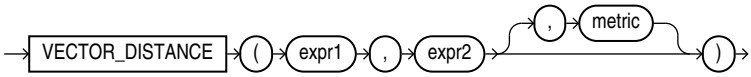
See Also:

- For a complete set of examples on each of the chunking parameters listed in the preceding table, see *Explore Chunking Techniques and Examples of the AI Vector Search User's Guide.*
- To run an end-to-end example scenario using this function, see *Convert Text to Chunks With Custom Chunking Specifications of the AI Vector Search User's Guide.*

VECTOR_DISTANCE

VECTOR_DISTANCE is the main function that you can use to calculate the distance between two vectors.

Syntax



Purpose

VECTOR_DISTANCE takes two vectors as parameters. You can optionally specify a distance metric to calculate the distance. If you do not specify a distance metric, then the default distance metric is cosine. If the input vectors are BINARY vectors, the default metric is hamming.

You can optionally use the following shorthand vector distance functions:

- L1_DISTANCE
- L2_DISTANCE
- COSINE_DISTANCE
- INNER_PRODUCT
- HAMMING_DISTANCE
- JACCARD_DISTANCE

All the vector distance functions take two vectors as input and return the distance between them as a BINARY_DOUBLE.

Note the following caveats:

- If you specify a metric as the third argument, then that metric is used.
- If you do not specify a metric, then the following rules apply:
  - If there is a single column referenced in *expr1* and *expr2* as in: VECTOR_DISTANCE(vec1, :bind), and if there is a vector index defined on *vec1*, then the metric used when defining the vector index is used.

    If no vector index is defined on *vec1*, then the COSINE metric is used.
  - If there are multiple columns referenced in *expr1* and *expr2* as in: VECTOR_DISTANCE(vec1, vec2), or VECTOR_DISTANCE(vec1+vec2, :bind), then for all indexed columns, if their metrics used in the definitions of the indexes are the same, then that metric is used.

    On the other hand, if the indexed columns do not have a common metric, or none of the columns have an index defined, then the COSINE metric is used.
- In a similarity search query, if *expr1* or *expr2* reference an indexed column and you specify a distance metric that conflicts with the metric specified in the vector index, then the vector index is not used and the metric you specified is used to perform an exact search.
- Approximate (index-based) searches can be done if only one column is referenced by either *expr1* or *expr2*, and this column has a vector index defined, and the metric that is

specified in the vector_distance matches the metric used in the definition of the vector index.

Parameters

- *expr1* and *expr2* must evaluate to vectors and have the same format and number of dimensions.

  If you use JACCARD_DISTANCE or the JACCARD metric, then *expr1* and *expr2* must evaluate to BINARY vectors.
- This function returns NULL if either *expr1* or *expr2* is NULL.
- *metric* must be one of the following tokens :
  - COSINE metric is the default metric. It calculates the cosine distance between two vectors.
  - DOT metric calculates the negated dot product of two vectors. The INNER_PRODUCT function calculates the dot product, as in the negation of this metric.
  - EUCLIDEAN metric, also known as L2 distance, calculates the Euclidean distance between two vectors.
  - EUCLIDEAN_SQUARED metric, also called L2_SQUARED, is the Euclidean distance without taking the square root.
  - HAMMING metric calculates the hamming distance between two vectors by counting the number dimensions that differ between the two vectors.
  - MANHATTAN metric, also known as L1 distance or taxicab distance, calculates the Manhattan distance between two vectors.
  - JACCARD metric calculates the Jaccard distance. The two vectors used in the query must be BINARY vectors.

Shorthand Operators for Distances

Syntax

- `expr1 <-> expr2`

  <-> is the **Euclidean distance operator**: `expr1 <-> expr2` is equivalent to `L2_DISTANCE(expr1, expr2)` or `VECTOR_DISTANCE(expr1, expr2, EUCLIDEAN)`
- `expr1 <=> expr2`

  <=> is the **cosine distance operator**: `expr1 <=> expr2` is equivalent to `COSINE_DISTANCE(expr1, expr2)` or `VECTOR_DISTANCE(expr1, expr2, COSINE)`
- `expr1 <#> expr2`

  <#> is the **negative dot product operator**: `expr1 <#> expr2` is equivalent to `-1*INNER_PRODUCT(expr1, expr2)` or `VECTOR_DISTANCE(expr1, expr2, DOT)`

Examples Using Shorthand Operators for Distances

```
 '[1, 2]' <-> '[0,1]'

v1 <-> '[' || '1,2,3' || ']' is equivalent to v1 <-> '[1, 2, 3]'

v1 <-> '[1,2]' is equivalent to L2_DISTANCE(v1, '[1,2]')

v1 <=> v2 is equivalent to COSINE_DISTANCE(v1, v2)
```

```
v1 <#> v2 is equivalent to -1*INNER_PRODUCT(v1, v2)
```

Examples

`VECTOR_DISTANCE` with metric `EUCLIDEAN` is equivalent to `L2_DISTANCE`:

```
VECTOR_DISTANCE(expr1, expr2, EUCLIDEAN);
```

```
L2_DISTANCE(expr1, expr2);
```

`VECTOR_DISTANCE` with metric `COSINE` is equivalent to `COSINE_DISTANCE`:

```
VECTOR_DISTANCE(expr1, expr2, COSINE);
```

```
COSINE_DISTANCE(expr1, expr2);
```

`VECTOR_DISTANCE` with metric `DOT` is equivalent to -1 * `INNER_PRODUCT`:

```
VECTOR_DISTANCE(expr1, expr2, DOT);
```

```
-1*INNER_PRODUCT(expr1, expr2);
```

`VECTOR_DISTANCE` with metric `MANHATTAN` is equivalent to `L1_DISTANCE`:

```
VECTOR_DISTANCE(expr1, expr2, MANHATTAN);
```

```
L1_DISTANCE(expr1, expr2);
```

`VECTOR_DISTANCE` with metric `HAMMING` is equivalent to `HAMMING_DISTANCE`:

```
VECTOR_DISTANCE(expr1, expr2, HAMMING);
```

```
HAMMING_DISTANCE(expr1, expr2);
```

`VECTOR_DISTANCE` with metric `JACCARD` is equivalent to `JACCARD_DISTANCE`:

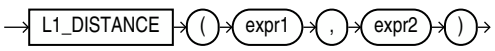
```
VECTOR_DISTANCE(expr1, expr2, JACCARD);
```

```
JACCARD_DISTANCE(expr1, expr2);
```

L1_DISTANCE

`L1_DISTANCE` is a shorthand version of the `VECTOR_DISTANCE` function that calculates the distance between two vectors. It takes two vectors as input and returns the distance between them as a `BINARY_DOUBLE`.

Syntax



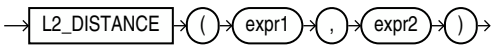
Parameters

- *expr1* and *expr2* must evaluate to vectors and have the same format and number of dimensions.
- L1_DISTANCE returns NULL, if either *expr1* or *expr2* is NULL.

L2_DISTANCE

L2_DISTANCE is a shorthand version of the VECTOR_DISTANCE function that calculates the distance between two vectors. It takes two vectors as input and returns the distance between them as a BINARY_DOUBLE.

Syntax



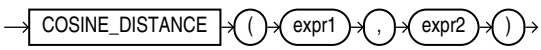
Parameters

- *expr1* and *expr2* must evaluate to vectors that have the same format and number of dimensions.
- L2_DISTANCE returns NULL, if either *expr1* or *expr2* is NULL.

COSINE_DISTANCE

COSINE_DISTANCE is a shorthand version of the VECTOR_DISTANCE function that calculates the distance between two vectors. It takes two vectors as input and returns the distance between them as a BINARY_DOUBLE.

Syntax



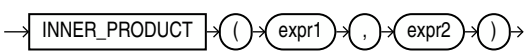
Parameters

- *expr1* and *expr2* must evaluate to vectors that have the same format and number of dimensions.
- COSINE_DISTANCE returns NULL, if either *expr1* or *expr2* is NULL.

INNER_PRODUCT

INNER_PRODUCT calculates the inner product of two vectors. It takes two vectors as input and returns the inner product as a BINARY_DOUBLE. INNER_PRODUCT(<*expr1*>, <*expr2*>) is equivalent to -1 * VECTOR_DISTANCE(<*expr1*>, <*expr2*>, DOT).

Syntax



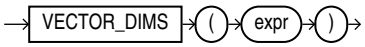
Parameters

- *expr1* and *expr2* must evaluate to vectors that have the same format and number of dimensions.
- INNER_PRODUCT returns NULL, if either *expr1* or *expr2* is NULL.

VECTOR_DIMS

VECTOR_DIMS returns the number of dimensions of a vector as a NUMBER. VECTOR_DIMS is synonymous with VECTOR_DIMENSION_COUNT.

Syntax



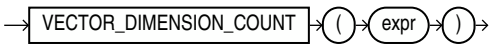
Purpose

Refer to VECTOR_DIMENSION_COUNT for full semantics.

VECTOR_DIMENSION_COUNT

VECTOR_DIMENSION_COUNT returns the number of dimensions of a vector as a NUMBER.

Syntax



Purpose

VECTOR_DIMENSION_COUNT is synonymous with VECTOR_DIMS.

Parameters

expr must evaluate to a vector.

If *expr* is NULL, NULL is returned.

Example

```
SELECT VECTOR_DIMENSION_COUNT( TO_VECTOR('[34.6, 77.8]', 2, FLOAT64) );

VECTOR_DIMENSION_COUNT(TO_VECTOR('[34.6,77.8]',2,FLOAT64))
----------------------------------------------------------
2
```

VECTOR_DIMENSION_FORMAT

`VECTOR_DIMENSION_FORMAT` returns the storage format of the vector. It returns a `VARCHAR2`, which can be one of the following values: `INT8`, `FLOAT32`, `FLOAT64`, or `BINARY`.

Syntax

Parameters

expr must evaluate to a vector.

If *expr* is NULL, NULL is returned.

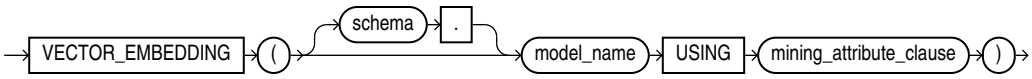
Examples

```
SELECT VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6, 77.8]', 2, FLOAT64));

VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6,77.8]',2,
--------------------------------------------------
FLOAT64


SELECT VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6, 77.8, 9]', 3, FLOAT32));

VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6,77.8,9]',
--------------------------------------------------
FLOAT32


SELECT VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6, 77.8, 9.10]', 3, INT8));

VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6,77.8,9.10
--------------------------------------------------
INT8


SELECT VECTOR_DIMENSION_FORMAT(TO_VECTOR('[206, 32]', 16, BINARY));

VECTOR_DIMENSION_FORMAT(TO_VECTOR('[206,32]',16,BI
--------------------------------------------------
BINARY


SELECT VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6, 77.8, 9, 10]', 3, INT8));

SELECT VECTOR_DIMENSION_FORMAT(TO_VECTOR('[34.6, 77.8, 9, 10]', 3, INT8))
                               *
ERROR at line 1:
ORA-51803: Vector dimension count must match the dimension count specified in
the column definition (expected 3 dimensions, specified 4 dimensions).
```

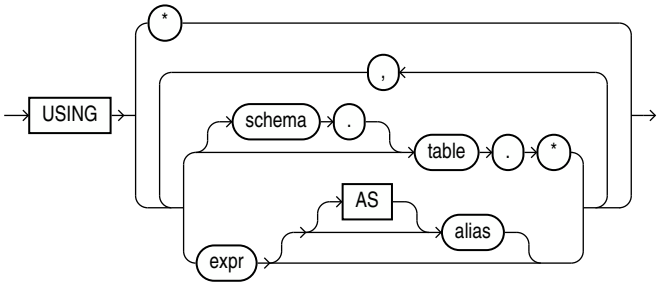
VECTOR_EMBEDDING

Use `VECTOR_EMBEDDING` to generate a single vector embedding for different data types using embedding or feature extraction machine learning models.

Syntax



***mining_attribute_clause*::=**



Purpose

The function accepts the following types as input:

`VARCHAR2` for text embedding models. Oracle automatically converts any other type to `VARCHAR2` except for `NCLOB`, which is automatically converted to `NVARCHAR2`. Oracle does not expect values whose textual representation exceeds the maximum size of a `VARCHAR2`, since embedding models support only text that translates to a couple of thousand tokens. An attribute with a type that has no conversion to `VARCHAR2` results in a SQL compilation error.

For feature extraction models Oracle Machine Learning for SQL supports standard Oracle data types except `DATE`, `TIMESTAMP`, `RAW`, and `LONG`. Oracle Machine Learning supports date type (datetime, date, timestamp) for case_id, `CLOB`/`BLOB`/`FILE` that are interpreted as text columns, and the following collection types as well:

- `DM_NESTED_CATEGORICALS`
- `DM_NESTED_NUMERICALS`
- `DM_NESTED_BINARY_DOUBLES`
- `DM_NESTED_BINARY_FLOATS`

The function always returns a `VECTOR` type, whose dimension is dictated by the model itself. The model stores the dimension information in metadata within the data dictionary.

You can use `VECTOR_EMBEDDING` in `SELECT` clauses, in predicates, and as an operand for SQL operations accepting a `VECTOR` type.

Parameters

model_name refers to the name of the imported embedding model that implements the embedding machine learning function.

mining_attribute_clause

- The *mining_attribute_clause* argument identifies the column attributes to use as predictors for scoring. This is used as a convenience, as the embedding operator only accepts single input value.
- USING * : all the relevant attributes present in the input (supplied in JSON metadata) are used. This is used as a convenience. For an embedding model, the operator only takes one input value as embedding models have only one column.
- USING *expr* [AS *alias*] [, *expr* [AS *alias*]] : all the relevant attributes present in the comma-separated list of column expressions are used. This syntax is consistent with the syntax of other machine learning operators. You may specify more than one attribute, however, the embedding model only takes one relevant input. Therefore, you must specify a single mining attribute.

Example

The following example generates vector embeddings with "hello" as the input, utilizing the pretrained ONNX format model my_embedding_model.onnx imported into the Database. For complete example, see Import ONNX Models and Generate Embeddings

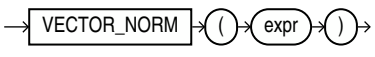
```
SELECT TO_VECTOR(VECTOR_EMBEDDING(model USING 'hello' as data)) AS embedding;
--------------------------------------------------------------------------------
[-9.76553112E-002,-9.89954844E-002,7.69771636E-003,-4.16760892E-003,-9.69305634E-002,
-3.01141385E-002,-2.63396613E-002,-2.98553891E-002,5.96499592E-002,4.13885899E-002,
5.32859489E-002,6.57707453E-002,-1.47056757E-002,-4.18472625E-002,4.1588001E-002,
-2.86354572E-002,-7.56499246E-002,-4.16395674E-003,-1.52879998E-001,6.60010576E-002,
-3.9013084E-002,3.15719917E-002,1.2428958E-002,-2.47651711E-002,-1.16851285E-001,
-7.82847106E-002,3.34323719E-002,8.03267583E-002,1.70483496E-002,-5.42407483E-002,
6.54291287E-002,-4.81935125E-003,6.11041225E-002,6.64106477E-003,-5.47
```

> **See Also:**
>
> - *Data Requirements for Machine Learning*
> - *Vector Distance Metrics*

VECTOR_NORM

VECTOR_NORM returns the Euclidean norm of a vector (SQRT(SUM((xi-yi)2))) as a BINARY_DOUBLE. This value is also called magnitude or size and represents the Euclidean distance between the vector and the origin.

Syntax



Parameters

expr must evaluate to a vector.

If *expr* is NULL, NULL is returned.

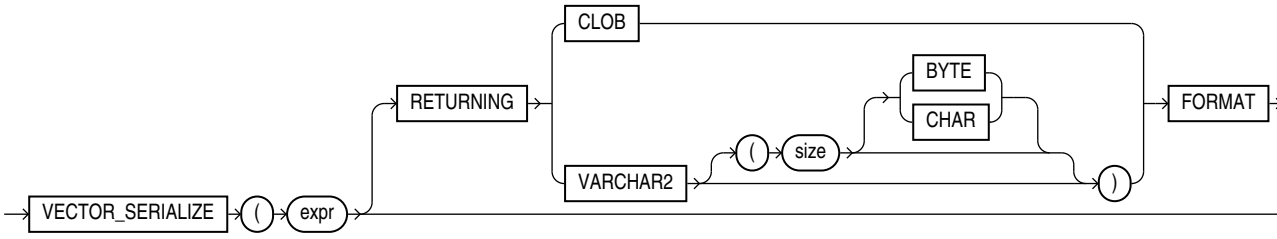
Example

```
SELECT VECTOR_NORM( TO_VECTOR('[4, 3]', 2, FLOAT32) );

VECTOR_NORM(TO_VECTOR('[4,3]',2,FLOAT32))
-----------------------------------------
5.0E+000
```

VECTOR_SERIALIZE

VECTOR_SERIALIZE is synonymous with FROM_VECTOR.

Syntax



Purpose

See FROM_VECTOR for semantics and examples.

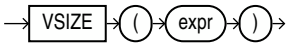
Examples

```
SELECT VECTOR_SERIALIZE(VECTOR('[1.1,2.2,3.3]',3,FLOAT32));


VECTOR_SERIALIZE(VECTOR('[1.1,2.2,3.3]',3,FLOAT32))
------------------------------------------------------------
[1.10000002E+000,2.20000005E+000,3.29999995E+000]

1 row selected.


SELECT VECTOR_SERIALIZE(VECTOR('[1.1, 2.2, 3.3]',3,FLOAT32) RETURNING VARCHAR2(1000));


VECTOR_SERIALIZE(VECTOR('[...]',3,FLOAT32)RETURNINGVARCHAR2(1000))
------------------------------------------------------------------
[1.10000002E+000,2.20000005E+000,3.29999995E+000]

1 row selected.


SELECT VECTOR_SERIALIZE(VECTOR('[1.1, 2.2, 3.3]',3,FLOAT32) RETURNING CLOB);
```

```
VECTOR_SERIALIZE(VECTOR('[1.1, 2.2, 3.3]',3,FLOAT32)RETURNINGCLOB)
------------------------------------------------------
[1.10000002E+000,2.20000005E+000,3.29999995E+000]

1 row selected.


SELECT VECTOR_SERIALIZE(TO_VECTOR('[5,[2,4],[1.0,2.0]]', 5, FLOAT64, SPARSE) RETURNING
CLOB FORMAT SPARSE);

VECTOR_SERIALIZE(TO_VECTOR('[5,[2,4],[1.0,2.0]]',5,FLOAT64,SPARSE)RETURNINGCLOBF
--------------------------------------------------------------------------------
[5,[2,4],[1.0E+000,2.0E+000]]

1 row selected.


SELECT VECTOR_SERIALIZE(TO_VECTOR('[5,[2,4],[1.0,2.0]]', 5, FLOAT64, SPARSE) RETURNING
CLOB FORMAT DENSE);

VECTOR_SERIALIZE(TO_VECTOR('[5,[2,4],[1.0,2.0]]',5,FLOAT64,SPARSE)RETURNINGCLOBF
--------------------------------------------------------------------------------
[0,1.0E+000,0,2.0E+000,0]

1 row selected.
```

VSIZE

Syntax



Purpose

`VSIZE` returns the number of bytes in the internal representation of *`expr`*. If *`expr`* is null, then this function returns null.

This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion.

> **See Also:**
>
> "Data Type Comparison Rules " for more information

Examples

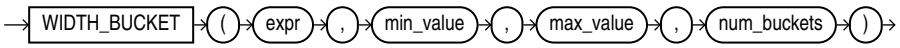
The following example returns the number of bytes in the `last_name` column of the employees in department 10:

```
SELECT last_name, VSIZE (last_name) "BYTES"
  FROM employees
  WHERE department_id = 10
  ORDER BY employee_id;
```

```
LAST_NAME            BYTES
-------------- ----------
Whalen                    6
```

WIDTH_BUCKET

Syntax



Purpose

`WIDTH_BUCKET` lets you construct equiwidth histograms, in which the histogram range is divided into intervals that have identical size. (Compare this function with `NTILE`, which creates equiheight histograms.) Ideally each bucket is a closed-open interval of the real number line. For example, a bucket can be assigned to scores between 10.00 and 19.999 ... to indicate that 10 is included in the interval and 20 is excluded. This is sometimes denoted [10, 20).

For a given expression, `WIDTH_BUCKET` returns the bucket number into which the value of this expression would fall after being evaluated.

- *`expr`* is the expression for which the histogram is being created. This expression must evaluate to a numeric or datetime value or to a value that can be implicitly converted to a numeric or datetime value. If *`expr`* evaluates to null, then the expression returns null.
- *`min_value`* and *`max_value`* are expressions that resolve to the end points of the acceptable range for *`expr`*. Both of these expressions must also evaluate to numeric or datetime values, and neither can evaluate to null.
- *`num_buckets`* is an expression that resolves to a constant indicating the number of buckets. This expression must evaluate to a positive integer.

> **See Also:**
>
> Table 2-9 for more information on implicit conversion

When needed, Oracle Database creates an underflow bucket numbered 0 and an overflow bucket numbered *`num_buckets`*+1. These buckets handle values less than *`min_value`* and more than *`max_value`* and are helpful in checking the reasonableness of endpoints.

Examples

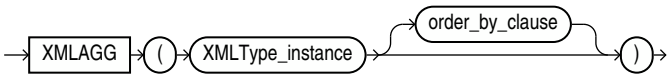
The following example creates a ten-bucket histogram on the `credit_limit` column for customers in Switzerland in the sample table `oe.customers` and returns the bucket number ("Credit Group") for each customer. Customers with credit limits greater than or equal to the maximum value are assigned to the overflow bucket, 11:

```
SELECT customer_id, cust_last_name, credit_limit,
   WIDTH_BUCKET(credit_limit, 100, 5000, 10) "Credit Group"
   FROM customers WHERE nls_territory = 'SWITZERLAND'
   ORDER BY "Credit Group", customer_id, cust_last_name, credit_limit;

CUSTOMER_ID CUST_LAST_NAME       CREDIT_LIMIT Credit Group
```

```
---------- -------------------- ------------ ------------
       825 Dreyfuss                      500            1
       826 Barkin                        500            1
       827 Siegel                        500            1
       853 Palin                         400            1
       843 Oates                         700            2
       844 Julius                        700            2
       835 Eastwood                     1200            3
       836 Berenger                     1200            3
       837 Stanton                      1200            3
       840 Elliott                      1400            3
       841 Boyer                        1400            3
       842 Stern                        1400            3
       848 Olmos                        1800            4
       849 Kaurusmdki                   1800            4
       828 Minnelli                     2300            5
       829 Hunter                       2300            5
       850 Finney                       2300            5
       851 Brown                        2300            5
       852 Tanner                       2300            5
       830 Dutt                         3500            7
       831 Bel Geddes                   3500            7
       832 Spacek                       3500            7
       833 Moranis                      3500            7
       834 Idle                         3500            7
       838 Nicholson                    3500            7
       839 Johnson                      3500            7
       845 Fawcett                      5000           11
       846 Brando                       5000           11
       847 Streep                       5000           11
```

XMLAGG

Syntax



Purpose

`XMLAgg` is an aggregate function. It takes a collection of XML fragments and returns an aggregated XML document. Any arguments that return null are dropped from the result.

`XMLAgg` is similar to `SYS_XMLAgg` except that `XMLAgg` returns a collection of nodes but it does not accept formatting using the `XMLFormat` object. Also, `XMLAgg` does not enclose the output in an element tag as does `SYS_XMLAgg`.

Within the *`order_by_clause`*, Oracle Database does not interpret number literals as column positions, as it does in other uses of this clause, but simply as number literals.

> **See Also:**
>
> XMLELEMENT and SYS_XMLAGG

Examples

The following example produces a `Department` element containing `Employee` elements with employee job ID and last name as the contents of the elements:

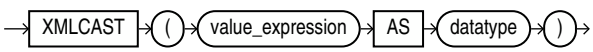
```
SELECT XMLELEMENT("Department",
   XMLAGG(XMLELEMENT("Employee",
   e.job_id||' '||e.last_name)
   ORDER BY last_name))
   as "Dept_list"
   FROM employees e
   WHERE e.department_id = 30;

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

Dept_list
-----------------------------------------------------------
<Department>
  <Employee>AD_ASST Whalen</Employee>
</Department>

<Department>
  <Employee>MK_MAN Hartstein</Employee>
  <Employee>MK_REP Fay</Employee>
</Department>

<Department>
  <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>
. . .
```

XMLCAST

Syntax



(*datatype*::=)

Purpose

`XMLCast` casts *value_expression* to the scalar SQL data type specified by *datatype*. The *value_expression* argument is a SQL expression that is evaluated.

datatype

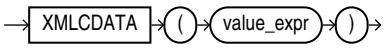
The *datatype* argument can be of data type `NUMBER`, `VARCHAR2`, `VARCHAR`, `CHAR`, `CLOB`, `BLOB`, `REF XMLTYPE`, and any of the datetime data types.

`BLOB`, or `CLOB` with options *reference* or *value*. The default is *reference*.

> **See Also:**
>
> - *Oracle XML DB Developer's Guide* for more information on uses for this function and examples
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the return value of `XMLCAST` when it is a character value

XMLCDATA

Syntax



Purpose

`XMLCData` generates a CDATA section by evaluating *value_expr*. The *value_expr* must resolve to a string. The value returned by the function takes the following form:

```
<![CDATA[string]]>
```

If the resulting value is not a valid XML CDATA section, then the function returns an error.The following conditions apply to `XMLCData`:

- The *value_expr* cannot contain the substring `]]>`.
- If *value_expr* evaluates to null, then the function returns null.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

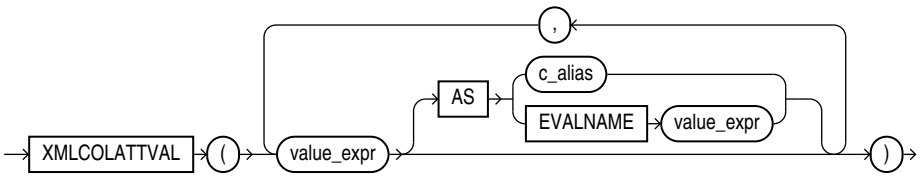
Examples

The following statement uses the `DUAL` table to illustrate the syntax of `XMLCData`:

```
SELECT XMLELEMENT("PurchaseOrder",
   XMLAttributes(dummy as "pono"),
   XMLCdata('<!DOCTYPE po_dom_group [
   <!ELEMENT po_dom_group(student_name)*>
   <!ELEMENT po_purch_name (#PCDATA)>
   <!ATTLIST po_name po_no ID #REQUIRED>
   <!ATTLIST po_name trust_1 IDREF #IMPLIED>
   <!ATTLIST po_name trust_2 IDREF #IMPLIED>
   ]>')) "XMLCData" FROM DUAL;

XMLCData
------------------------------------------------------------
<PurchaseOrder pono="X"><![CDATA[
<!DOCTYPE po_dom_group [
   <!ELEMENT po_dom_group(student_name)*>
   <!ELEMENT po_purch_name (#PCDATA)>
   <!ATTLIST po_name po_no ID #REQUIRED>
   <!ATTLIST po_name trust_1 IDREF #IMPLIED>
   <!ATTLIST po_name trust_2 IDREF #IMPLIED>
   ]>
  ]]>
</PurchaseOrder>
```

XMLCOLATTVAL

Syntax



Purpose

`XMLColAttVal` creates an XML fragment and then expands the resulting XML so that each XML fragment has the name `column` with the attribute `name`.

You can use the `AS` clause to change the value of the `name` attribute to something other than the column name. You can do this by specifying *`c_alias`*, which is a string literal, or by specifying `EVALNAME` *`value_expr`*. In the latter case, the value expression is evaluated and the result, which must be a string literal, is used as the alias. The alias can be up to 4000 characters if the initialization parameter `MAX_STRING_SIZE = STANDARD`, and 32767 characters if `MAX_STRING_SIZE = EXTENDED`. See "Extended Data Types" for more information.

You must specify a value for *`value_expr`*. If *`value_expr`* is null, then no element is returned.

Restriction on XMLColAttVal

You cannot specify an object type column for *`value_expr`*.

Examples

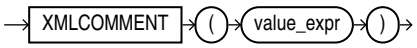
The following example creates an `Emp` element for a subset of employees, with nested `employee_id`, `last_name`, and `salary` elements as the contents of `Emp`. Each nested element is named `column` and has a `name` attribute with the column name as the attribute value:

```
SELECT XMLELEMENT("Emp",
   XMLCOLATTVAL(e.employee_id, e.last_name, e.salary)) "Emp Element"
   FROM employees e
   WHERE employee_id = 204;

Emp Element
--------------------------------------------------------------------
<Emp>
  <column name="EMPLOYEE_ID">204</column>
  <column name="LAST_NAME">Baer</column>
  <column name="SALARY">10000</column>
</Emp>
```

Refer to the example for XMLFOREST to compare the output of these two functions.

XMLCOMMENT

Syntax



Purpose

`XMLComment` generates an XML comment using an evaluated result of *`value_expr`*. The *`value_expr`* must resolve to a string. It cannot contain two consecutive dashes (hyphens). The value returned by the function takes the following form:

```
<!--string-->
```

If *`value_expr`* resolves to null, then the function returns null.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

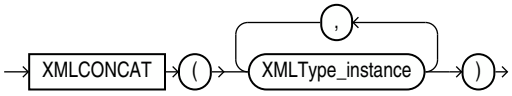
Examples

The following example uses the `DUAL` table to illustrate the `XMLComment` syntax:

```
SELECT XMLCOMMENT('OrderAnalysisComp imported, reconfigured, disassembled')
   AS "XMLCOMMENT" FROM DUAL;

XMLCOMMENT
--------------------------------------------------------------------------------
<!--OrderAnalysisComp imported, reconfigured, disassembled-->
```

XMLCONCAT

Syntax



Purpose

`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`.

Null expressions are dropped from the result. If all the value expressions are null, then the function returns null.

> **See Also:**
>
> XMLSEQUENCE

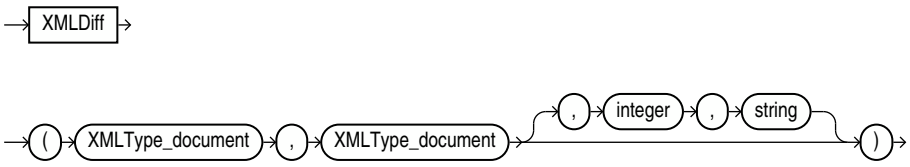
Examples

The following example creates XML elements for the first and last names of a subset of employees, and then concatenates and returns those elements:

```
SELECT XMLCONCAT(XMLELEMENT("First", e.first_name),
   XMLELEMENT("Last", e.last_name)) AS "Result"
   FROM employees e
   WHERE e.employee_id > 202;

Result
--------------------------------------------------------------------
<First>Susan</First>
<Last>Mavris</Last>

<First>Hermann</First>
<Last>Baer</Last>

<First>Shelley</First>
<Last>Higgins</Last>

<First>William</First>
<Last>Gietz</Last>

4 rows selected.
```

XMLDIFF

Syntax



Purpose

The `XMLDiff` function is the SQL interface for the XmlDiff C API. This function compares two XML documents and captures the differences in XML conforming to an Xdiff schema. The diff document is returned as an XMLType document.

- For the first two arguments, specify the names of two XMLType documents.
- For the *integer*, specify a number representing the hashLevel for a C function XmlDiff. If you do not want hashing, set this argument to 0 or omit it entirely. If you do not want hashing, but you want to specify flags, then you must set this argument to 0.
- For *string*, specify the flags that control the behavior of the function. These flags are specified by one or more names separated by semicolon. The names are the same as the names of constants for XmlDiff function.

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide* for more information on using this function, including examples, and *Oracle Database XML C API Reference* for information on the XML APIs for C

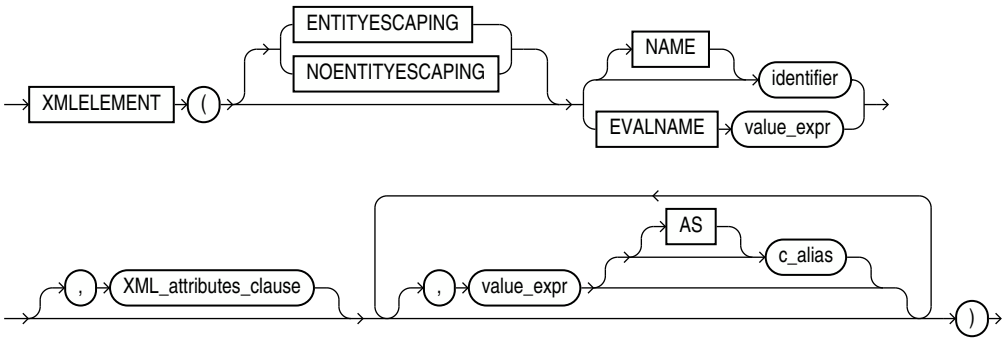
Examples

The following example compares two XML documents and returns the difference as an XMLType document:

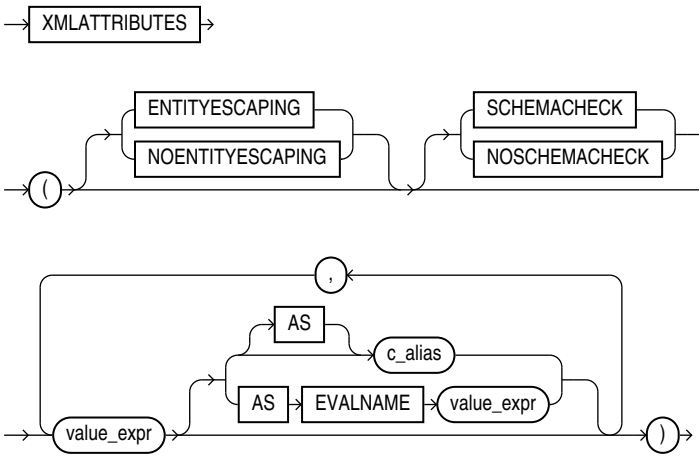
```
SELECT XMLDIFF(
XMLTYPE('<?xml version="1.0"?>
<bk:book xmlns:bk="http://example.com">
   <bk:tr>
        <bk:td>
                <bk:chapter>
                        Chapter 1.
                </bk:chapter>
        </bk:td>
        <bk:td>
                 <bk:chapter>
                        Chapter 2.
                </bk:chapter>
        </bk:td>
   </bk:tr>
</bk:book>'),
XMLTYPE('<?xml version="1.0"?>
<bk:book xmlns:bk="http://example.com">
   <bk:tr>
        <bk:td>
                <bk:chapter>
                        Chapter 1.
                </bk:chapter>
        </bk:td>
        <bk:td/>
   </bk:tr>
</bk:book>')
)
FROM DUAL;
```

XMLELEMENT

Syntax



***XML_attributes_clause*::=**



Purpose

`XMLElement` takes an element name for *`identifier`* or evaluates an element name for `EVALNAME` *`value_expr`*, an optional collection of attributes for the element, and arguments that make up the content of the element. It returns an instance of type `XMLType`. `XMLElement` is similar to `SYS_XMLGen` except that `XMLElement` can include attributes in the XML returned, but it does not accept formatting using the `XMLFormat` object.

The `XMLElement` function is typically nested to produce an XML document with a nested structure, as in the example in the following section.

For an explanation of the `ENTITYESCAPING` and `NONENTITYESCAPING` keywords, refer to *Oracle XML DB Developer's Guide*.

You must specify a value for Oracle Database to use an the enclosing tag. You can do this by specifying *`identifier`*, which is a string literal, or by specifying `EVALNAME` *`value_expr`*. In the latter case, the value expression is evaluated and the result, which must be a string literal, is used as the identifier. The identifier does not have to be a column name or column reference. It

cannot be an expression or null. It can be up to 4000 characters if the initialization parameter MAX_STRING_SIZE = STANDARD, and 32767 characters if MAX_STRING_SIZE = EXTENDED.

The objects that make up the element content follow the XMLATTRIBUTES keyword. In the *XML_attributes_clause*, if the *value_expr* is null, then no attribute is created for that value expression. The type of *value_expr* cannot be an object type or collection. If you specify an alias for *value_expr* using the AS clause, then the *c_alias* or the evaluated value expression (EVALNAME *value_expr*) can be up to 4000 characters if the initialization parameter MAX_STRING_SIZE = STANDARD, and 32767 characters if MAX_STRING_SIZE = EXTENDED.

> **See Also:**
>
> "Extended Data Types" for more information on MAX_STRING_SIZE

For the optional *value_expr* that follows the *XML_attributes_clause* in the diagram:

- If *value_expr* is a scalar expression, then you can omit the AS clause, and Oracle uses the column name as the element name.
- If *value_expr* is an object type or collection, then the AS clause is mandatory, and Oracle uses the specified *c_alias* as the enclosing tag.
- If *value_expr* is null, then no element is created for that value expression.

> **See Also:**
>
> SYS_XMLGEN

Examples

The following example produces an Emp element for a series of employees, with nested elements that provide the employee's name and hire date:

```
SELECT XMLELEMENT("Emp", XMLELEMENT("Name",
   e.job_id||' '||e.last_name),
   XMLELEMENT("Hiredate", e.hire_date)) as "Result"
   FROM employees e WHERE employee_id > 200;

Result
--------------------------------------------------------------------
<Emp>
  <Name>MK_MAN Hartstein</Name>
  <Hiredate>2004-02-17</Hiredate>
</Emp>

<Emp>
  <Name>MK_REP Fay</Name>
  <Hiredate>2005-08-17</Hiredate>
</Emp>

<Emp>
  <Name>HR_REP Mavris</Name>
  <Hiredate>2002-06-07</Hiredate>
</Emp>
```

```
<Emp>
  <Name>PR_REP Baer</Name>
  <Hiredate>2002-06-07</Hiredate>
</Emp>

<Emp>
  <Name>AC_MGR Higgins</Name>
  <Hiredate>2002-06-07</Hiredate>
</Emp>

<Emp>
  <Name>AC_ACCOUNT Gietz</Name>
  <Hiredate>2002-06-07</Hiredate>
</Emp>

6 rows selected.
```

The following similar example uses the `XMLElement` function with the *`XML_attributes_clause`* to create nested XML elements with attribute values for the top-level element:

```
SELECT XMLELEMENT("Emp",
      XMLATTRIBUTES(e.employee_id AS "ID", e.last_name),
      XMLELEMENT("Dept", e.department_id),
      XMLELEMENT("Salary", e.salary)) AS "Emp Element"
   FROM employees e
   WHERE e.employee_id = 206;

Emp Element
--------------------------------------------------------------
<Emp ID="206" LAST_NAME="Gietz">
  <Dept>110</Dept>
  <Salary>8300</Salary>
</Emp>
```

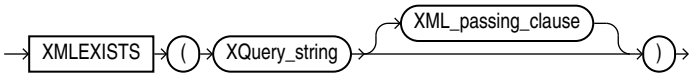
Notice that the `AS` *`identifier`* clause was not specified for the `last_name` column. As a result, the XML returned uses the column name `last_name` as the default.

Finally, the next example uses a subquery within the *`XML_attributes_clause`* to retrieve information from another table into the attributes of an element:

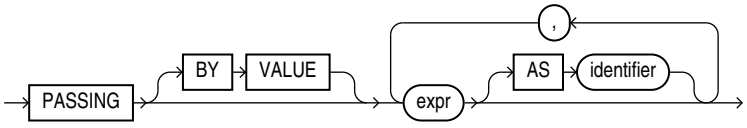
```
SELECT XMLELEMENT("Emp", XMLATTRIBUTES(e.employee_id, e.last_name),
   XMLELEMENT("Dept", XMLATTRIBUTES(e.department_id,
   (SELECT d.department_name FROM departments d
   WHERE d.department_id = e.department_id) as "Dept_name")),
   XMLELEMENT("salary", e.salary),
   XMLELEMENT("Hiredate", e.hire_date)) AS "Emp Element"
   FROM employees e
   WHERE employee_id = 205;

Emp Element
---------------------------------------------------------------------
<Emp EMPLOYEE_ID="205" LAST_NAME="Higgins">
  <Dept DEPARTMENT_ID="110" Dept_name="Accounting"/>
  <salary>12008</salary>
  <Hiredate>2002-06-07</Hiredate>
</Emp>
```

XMLEXISTS

Syntax



***XML_passing_clause*::=**



Purpose

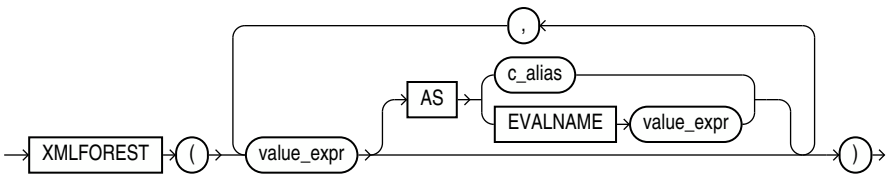
`XMLExists` checks whether a given XQuery expression returns a nonempty XQuery sequence. If so, the function returns `TRUE`; otherwise, it returns `FALSE`. The argument *`XQuery_string`* is a literal string, but it can contain XQuery variables that you bind using the *`XML_passing_clause`*.

The *`expr`* in the *`XML_passing_clause`* is an expression returning an `XMLType` or an instance of a SQL scalar data type that is used as the context for evaluating the XQuery expression. You can specify only one *`expr`* in the `PASSING` clause without an identifier. The result of evaluating each *`expr`* is bound to the corresponding identifier in the *`XQuery_string`*. If any *`expr`* that is not followed by an `AS` clause, then the result of evaluating that expression is used as the context item for evaluating the *`XQuery_string`*. If *`expr`* is a relational column, then its declared collation is ignored by Oracle XML DB.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on uses for this function and examples

XMLFOREST

Syntax



Purpose

`XMLForest` converts each of its argument parameters to XML, and then returns an XML fragment that is the concatenation of these converted arguments.

- If *value_expr* is a scalar expression, then you can omit the `AS` clause, and Oracle Database uses the column name as the element name.
- If *value_expr* is an object type or collection, then the `AS` clause is mandatory, and Oracle uses the specified expression as the enclosing tag.

  You can do this by specifying *c_alias*, which is a string literal, or by specifying `EVALNAME` *value_expr*. In the latter case, the value expression is evaluated and the result, which must be a string literal, is used as the identifier. The identifier does not have to be a column name or column reference. It cannot be an expression or null. It can be up to 4000 characters if the initialization parameter `MAX_STRING_SIZE = STANDARD`, and 32767 characters if `MAX_STRING_SIZE = EXTENDED`. See "Extended Data Types" for more information.
- If *value_expr* is null, then no element is created for that *value_expr*.

Examples

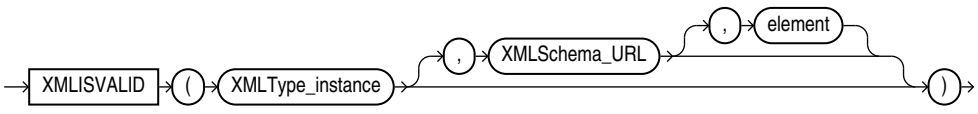
The following example creates an `Emp` element for a subset of employees, with nested `employee_id`, `last_name`, and `salary` elements as the contents of `Emp`:

```
SELECT XMLELEMENT("Emp",
   XMLFOREST(e.employee_id, e.last_name, e.salary))
   "Emp Element"
   FROM employees e WHERE employee_id = 204;

Emp Element
--------------------------------------------------------------------
<Emp>
  <EMPLOYEE_ID>204</EMPLOYEE_ID>
  <LAST_NAME>Baer</LAST_NAME>
  <SALARY>10000</SALARY>
</Emp>
```

Refer to the example for XMLCOLATTVAL to compare the output of these two functions.

XMLISVALID

Syntax



Purpose

`XMLISVALID` checks whether the input *XMLType_instance* conforms to the relevant XML schema. It does not change the validation status recorded for *XMLType_instance*.

If the input XML document is determined to be valid, then `XMLISVALID` returns 1; otherwise, it returns 0. If you provide *XMLSchema_URL* as an argument, then that is used to check

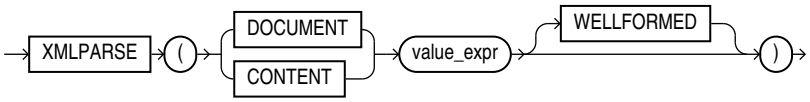
conformance. Otherwise, the XML schema specified by the XML document is used to check conformance.

- *XMLType_instance* is the XMLType instance to be validated.
- *XMLSchema_URL* is the URL of the XML schema against which to check conformance.
- *element* is the element of the specified schema against which to check conformance. Use this if you have an XML schema that defines more than one top level element, and you want to check conformance against a specific one of those elements.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for information on the use of this function, including examples

XMLPARSE

Syntax



Purpose

`XMLParse` parses and generates an XML instance from the evaluated result of *value_expr*. The *value_expr* must resolve to a string. If *value_expr* resolves to null, then the function returns null.

- If you specify `DOCUMENT`, then *value_expr* must resolve to a singly rooted XML document.
- If you specify `CONTENT`, then *value_expr* must resolve to a valid XML value.
- When you specify `WELLFORMED`, you are guaranteeing that *value_expr* resolves to a well-formed XML document, so the database does not perform validity checks to ensure that the input is well formed.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

Examples

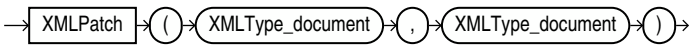
The following example uses the `DUAL` table to illustrate the syntax of `XMLParse`:

```
SELECT XMLPARSE(CONTENT '124 <purchaseOrder poNo="12435">
   <customerName> Acme Enterprises</customerName>
   <itemNo>32987457</itemNo>
   </purchaseOrder>'
WELLFORMED) AS PO FROM DUAL;
```

```
PO
--------------------------------------------------------------------
124 <purchaseOrder poNo="12435">
   <customerName> Acme Enterprises</customerName>
   <itemNo>32987457</itemNo>
   </purchaseOrder>
```

XMLPATCH

Syntax



Purpose

The `XMLPatch` function is the SQL interface for the XmlPatch C API. This function patches an XML document with the changes specified. A patched `XMLType` document is returned.

- For the first argument, specify the name of the input `XMLType` document.
- For the second argument, specify the XMLType document containing the changes to be applied to the first document. The changes should conform to the Xdiff XML schema. You can supply the XML output from the Oracle XML Developer's Kit Java method `diff()`.

> **See Also:**
>
> *Oracle XML Developer's Kit Programmer's Guide* for more information on using this function, including examples, and *Oracle Database XML C API Reference* for information on the XML APIs for C

Examples

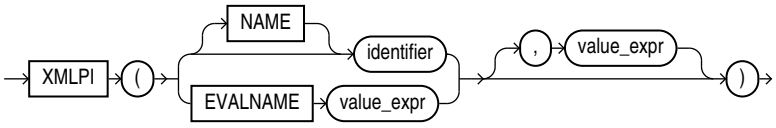
The following example patches an `XMLType` document with the changes specified in another `XMLType` and returns a patched `XMLType` document:

```
SELECT XMLPATCH(
XMLTYPE('<?xml version="1.0"?>
<bk:book xmlns:bk="http://example.com">
   <bk:tr>
        <bk:td>
                <bk:chapter>
                        Chapter 1.
                </bk:chapter>
        </bk:td>
        <bk:td>
                 <bk:chapter>
                        Chapter 2.
                </bk:chapter>
        </bk:td>
   </bk:tr>
</bk:book>'),
XMLTYPE('<?xml version="1.0"?>
<xd:xdiff xsi:schemaLocation="http://xmlns.oracle.com/xdb/xdiff.xsd
  http://xmlns.oracle.com/xdb/xdiff.xsd"
  xmlns:xd="http://xmlns.oracle.com/xdb/xdiff.xsd"
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:bk="http://example.com">
  <?oracle-xmldiff operations-in-docorder="true" output-model="snapshot"
    diff-algorithm="global"?>
  <xd:delete-node xd:node-type="element"
   xd:xpath="/bk:book[1]/bk:tr[1]/bk:td[2]/bk:chapter[1]"/>
</xd:xdiff>')
)
FROM DUAL;
```

XMLPI

Syntax



Purpose

`XMLPI` generates an XML processing instruction using *`identifier`* and optionally the evaluated result of *`value_expr`*. A processing instruction is commonly used to provide to an application information that is associated with all or part of an XML document. The application uses the processing instruction to determine how best to process the XML document.

You must specify a value for Oracle Database to use an the enclosing tag. You can do this by specifying *`identifier`*, which is a string literal, or by specifying `EVALNAME` *`value_expr`*. In the latter case, the value expression is evaluated and the result, which must be a string literal, is used as the identifier. The identifier does not have to be a column name or column reference. It cannot be an expression or null. It can be up to 4000 characters if the initialization parameter `MAX_STRING_SIZE = STANDARD`, and 32767 characters if `MAX_STRING_SIZE = EXTENDED`. See "Extended Data Types" for more information.

The optional *`value_expr`* must resolve to a string. If you omit the optional *`value_expr`*, then a zero-length string is the default. The value returned by the function takes this form:

```
<?identifier string?>
```

`XMLPI` is subject to the following restrictions:

- The *`identifier`* must be a valid target for a processing instruction.
- You cannot specify `xml` in any case combination for *`identifier`*.
- The *`identifier`* cannot contain the consecutive characters `?>`.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

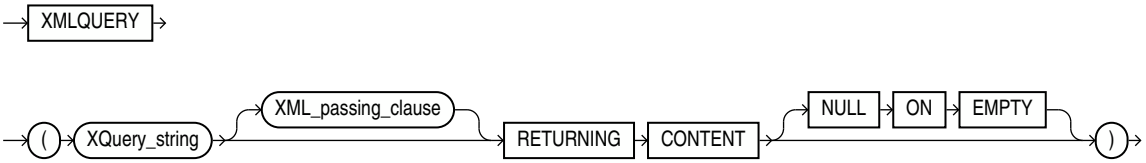
Examples

The following statement uses the `DUAL` table to illustrate the use of the `XMLPI` syntax:

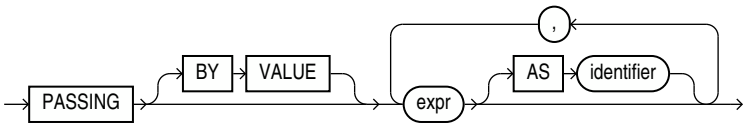
```
SELECT XMLPI(NAME "Order analysisComp", 'imported, reconfigured, disassembled')
   AS "XMLPI" FROM DUAL;

XMLPI
--------------------------------------------------------------------------------
<?Order analysisComp imported, reconfigured, disassembled?>
```

XMLQUERY

Syntax



***XML_passing_clause*::=**



Purpose

`XMLQUERY` lets you query XML data in SQL statements. It 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.

- *`XQuery_string`* is a complete XQuery expression, including prolog.
- The *`expr`* in the *`XML_passing_clause`* is an expression returning an `XMLType` or an instance of a SQL scalar data type that is used as the context for evaluating the XQuery expression. You can specify only one *`expr`* in the `PASSING` clause without an identifier. The result of evaluating each *`expr`* is bound to the corresponding identifier in the *`XQuery_string`*. If any *`expr`* that is not followed by an `AS` clause, then the result of evaluating that expression is used as the context item for evaluating the *`XQuery_string`*. If *`expr`* is a relational column, then its declared collation is ignored by Oracle XML DB.
- `RETURNING CONTENT` indicates that the result from the XQuery evaluation is either an XML 1.0 document or a document fragment conforming to the XML 1.0 semantics.
- If the result set is empty, then the function returns the SQL `NULL` value. The `NULL ON EMPTY` keywords are implemented by default and are shown for semantic clarity.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

Examples

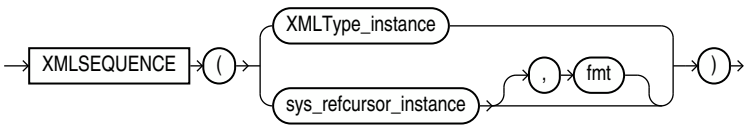
The following statement specifies the *warehouse_spec* column of the `oe.warehouses` table in the *XML_passing_clause* as a context item. The statement returns specific information about the warehouses with area greater than 50K.

```
SELECT warehouse_name,
EXTRACTVALUE(warehouse_spec, '/Warehouse/Area'),
XMLQuery(
   'for $i in /Warehouse
   where $i/Area > 50000
   return <Details>
             <Docks num="{$i/Docks}"/>
             <Rail>
               {
               if ($i/RailAccess = "Y") then "true" else "false"
               }
             </Rail>
          </Details>' PASSING warehouse_spec RETURNING CONTENT) "Big_warehouses"
   FROM warehouses;

WAREHOUSE_ID Area      Big_warehouses
------------ --------- ------------------------------------------------------
           1     25000
           2     50000
           3     85700 <Details><Docks></Docks><Rail>false</Rail></Details>
           4    103000 <Details><Docks num="3"></Docks><Rail>true</Rail></Details>
 . . .
```

XMLSEQUENCE

> **Note:**
>
> The `XMLSEQUENCE` function is deprecated. It is still supported for backward compatibility. However, Oracle recommends that you use the `XMLTABLE` function instead. See XMLTABLE for more information.

Syntax



Purpose

`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`. This form is effectively superseded by the SQL/XML standard function `XMLTable`, which provides for more readable SQL code. Prior to Oracle Database 10g Release 2, `XMLSequence` was used with SQL function `TABLE` to do some of what can now be done better with the `XMLTable` function.

- 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.

Because `XMLSequence` returns a collection of `XMLType`, you can use this function in a `TABLE` clause to unnest the collection values into multiple rows, which can in turn be further processed in the SQL query.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function, and XMLTABLE

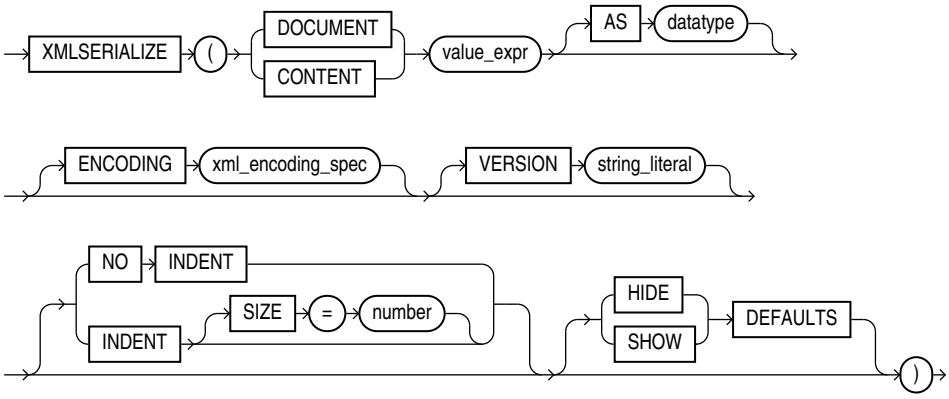
Examples

The following example shows how `XMLSequence` divides up an XML document with multiple elements into `VARRAY` single-element documents. In this example, the `TABLE` keyword instructs Oracle Database to consider the collection a table value that can be used in the `FROM` clause of the subquery:

```
SELECT EXTRACT(warehouse_spec, '/Warehouse') as "Warehouse"
   FROM warehouses WHERE warehouse_name = 'San Francisco';

Warehouse
------------------------------------------------------------
<Warehouse>
  <Building>Rented</Building>
  <Area>50000</Area>
  <Docks>1</Docks>
  <DockType>Side load</DockType>
  <WaterAccess>Y</WaterAccess>
  <RailAccess>N</RailAccess>
  <Parking>Lot</Parking>
  <VClearance>12 ft</VClearance>
</Warehouse>

1 row selected.

SELECT VALUE(p)
   FROM warehouses w,
   TABLE(XMLSEQUENCE(EXTRACT(warehouse_spec, '/Warehouse/*'))) p
   WHERE w.warehouse_name = 'San Francisco';

VALUE(P)
--------------------------------------------------------------------
<Building>Rented</Building>
<Area>50000</Area>
<Docks>1</Docks>
<DockType>Side load</DockType>
<WaterAccess>Y</WaterAccess>
<RailAccess>N</RailAccess>
<Parking>Lot</Parking>
<VClearance>12 ft</VClearance>

8 rows selected.
```

XMLSERIALIZE

Syntax



(*datatype*::=)

Purpose

`XMLSerialize` creates a string or LOB containing the contents of *`value_expr`*.

Any lob returned by `XMLSERIALIZE` will be read-only.

If you specify `DOCUMENT`, then the *`value_expr`* must be a valid XML document.

If you specify `CONTENT`, then the *`value_expr`* need not be a singly rooted XML document. However it must be valid XML content.

datatype

The *`datatype`* specified can be:

- `VARCHAR2` or `VARCHAR`, but not `NVARCHAR2`
- `BLOB`, or `CLOB` with options *`reference`* or *`value`*. The default is *`reference`*.
- With `BLOB`, you can specify the `ENCODING` clause to use the specified encoding in the prolog. The *`xml_encoding_spec`* is an XML encoding declaration (`encoding="..."`).

The default type is `CLOB`.

Specify the `VERSION` clause to use the version you provide as *`string_literal`* in the XML declaration (`<?xml version="..." ...?>`).

Specify `NO INDENT` to strip all insignificant whitespace from the output. Specify `INDENT SIZE =` *`N`*, where *`N`* is a whole number, for output that is pretty-printed using a relative indentation of *`N`* spaces. If *`N`* is `0`, then pretty-printing inserts a newline character after each element, placing each element on a line by itself, but omitting all other insignificant whitespace in the output. If `INDENT` is present without a `SIZE` specification, then 2-space indenting is used. If you omit this clause, then the behavior (pretty-printing or not) is indeterminate.

`HIDE DEFAULTS` and `SHOW DEFAULTS` apply only to XML schema-based data. If you specify `SHOW DEFAULTS` and the input data is missing any optional elements or attributes for which the XML schema defines default values, then those elements or attributes are included in the output

with their default values. If you specify `HIDE DEFAULTS`, then no such elements or attributes are included in the output. `HIDE DEFAULTS` is the default behavior.

> **See Also:**
>
> - *Oracle XML DB Developer's Guide* for more information on this function
> - Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to the character return value of `XMLSERIALIZE`

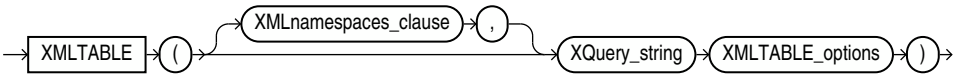
Examples

The following statement uses the `DUAL` table to illustrate the syntax of `XMLSerialize`:

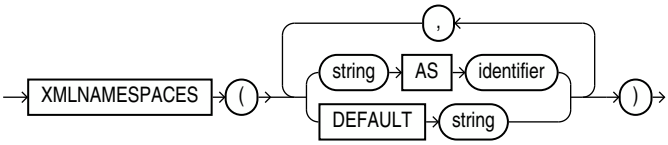
```
SELECT XMLSERIALIZE(CONTENT XMLTYPE('<Owner>Grandco</Owner>')) AS xmlserialize_doc
   FROM DUAL;

XMLSERIALIZE_DOC
----------------
<Owner>Grandco</Owner>
```

XMLTABLE

Syntax

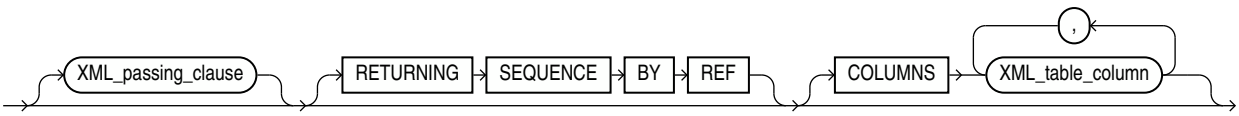


***XMLnamespaces_clause*::=**

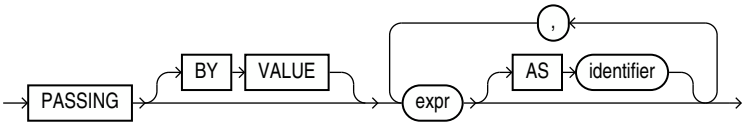


> **Note:**
>
> You can specify at most one `DEFAULT` *`string`* clause.

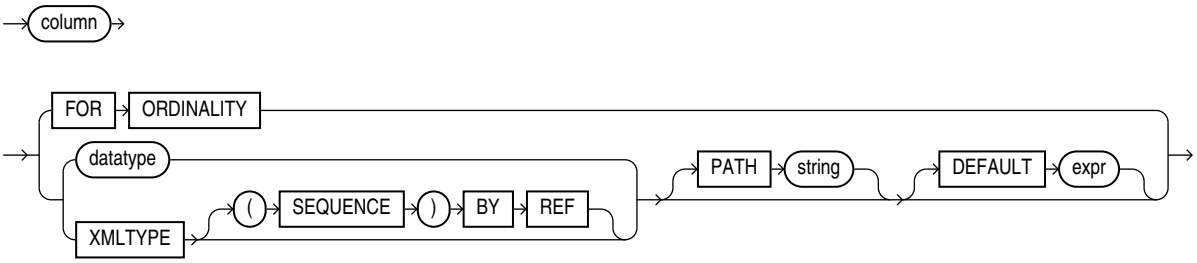
***XMLTABLE_options*::=**



***XML_passing_clause*::=**



***XML_table_column*::=**



(*datatype*::=)

Purpose

`XMLTable` maps the result of an XQuery evaluation into relational rows and columns. You can query the result returned by the function as a virtual relational table using SQL.

- The `XMLNAMESPACES` clause contains a set of XML namespace declarations. These declarations are referenced by the XQuery expression (the evaluated *`XQuery_string`*), which computes the row, and by the XPath expression in the `PATH` clause of *`XML_table_column`*, which computes the columns for the entire `XMLTable` function. If you want to use qualified names in the `PATH` expressions of the `COLUMNS` clause, then you need to specify the `XMLNAMESPACES` clause.
- *`XQuery_string`* is a literal string. It is a complete XQuery expression and can include prolog declarations. The value of XQuery_string serves as input to the XMLTable function; it is this XQuery result that is decomposed and stored as relational data.
- The *`expr`* in the *`XML_passing_clause`* is an expression returning an `XMLType` or an instance of a SQL scalar data type that is used as the context for evaluating the XQuery expression. You can specify only one *`expr`* in the `PASSING` clause without an identifier. The result of evaluating each *`expr`* is bound to the corresponding identifier in the *`XQuery_string`*. If any *`expr`* that is not followed by an `AS` clause, then the result of evaluating that expression is used as the context item for evaluating the *`XQuery_string`*. This clause supports only passing by value, not passing by reference. Therefore, the `BY VALUE` keywords are optional and are provided for semantic clarity.
- The optional `RETURNING SEQUENCE BY REF` clause causes the result of the XQuery evaluation to be returned by reference. This allows you to refer to any part of the source data in the *`XML_table_column`* clause.

  If you omit this clause, then the result of the XQuery evaluation is returned by value. That is, a copy of the targeted nodes is returned instead of a reference to the actual nodes. In this case, you cannot refer to any data that is not in the returned copy in the

XML_table_column clause. In particular, you cannot refer to data that precedes the targeted nodes in the source data.

- The optional COLUMNS clause defines the columns of the virtual table to be created by XMLTable.
  - If you omit the COLUMNS clause, then XMLTable returns a row with a single XMLType pseudocolumn named COLUMN_VALUE.
  - FOR ORDINALITY specifies that *column* is to be a column of generated row numbers. There must be at most one FOR ORDINALITY clause. It is created as a NUMBER column.
  - For each resulting column except the FOR ORDINALITY column, you must specify the column data type, which can be XMLType or any other data type.

    If the column data type is XMLType, then specify the XMLTYPE clause. If you specify the optional (SEQUENCE) BY REF clause, then a reference to the source data targeted by the PATH expression is returned as the column content. Otherwise, *column* contains a copy of that targeted data.

    Returning the XMLType data by reference lets you specify other columns whose paths target nodes in the source data that are outside those targeted by the PATH expression for column.

    If the column data type is any other data type, then specify *datatype_clause*.

    datatype

    The *datatype* specified can be:

    * BLOB, or CLOB with options *reference* or *value*. The default is *reference*.
    * Any other data type.
  - The optional PATH clause specifies that the portion of the XQuery result that is addressed by XQuery expression string is to be used as the column content.

    If you omit PATH, then the XQuery expression *column* is assumed. For example:

    ```
    XMLTable(... COLUMNS xyz)
    ```

    is equivalent to

    ```
    XMLTable(... COLUMNS xyz PATH 'XYZ')
    ```

    You can use different PATH clauses to split the XQuery result into different virtual-table columns.
  - The optional DEFAULT clause specifies the value to use when the PATH expression results in an empty sequence. Its *expr* is an XQuery expression that is evaluated to produce the default value.

See Also:

- *Oracle XML DB Developer's Guide* for more information on the XMLTable function, including additional examples, and on XQuery in general
- Appendix C in *Oracle Database Globalization Support Guide* for the collation derivation rules, which define the collation assigned to each character data type column in the table generated by XMLTABLE

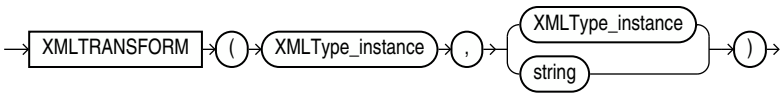
Examples

The following example converts the result of applying the XQuery `'/Warehouse'` to each value in the `warehouse_spec` column of the `warehouses` table into a virtual relational table with columns `Water` and `Rail`:

```
SELECT warehouse_name warehouse,
   warehouse2."Water", warehouse2."Rail"
   FROM warehouses,
   XMLTABLE('/Warehouse'
      PASSING warehouses.warehouse_spec
      COLUMNS
         "Water" varchar2(6) PATH 'WaterAccess',
         "Rail" varchar2(6) PATH 'RailAccess')
      warehouse2;

WAREHOUSE                          Water  Rail
---------------------------------- ------ ------
Southlake, Texas                   Y      N
San Francisco                      Y      N
New Jersey                         N      N
Seattle, Washington                N      Y
```

XMLTRANSFORM

Syntax



Purpose

`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`.

This function is useful for organizing data according to a style sheet as you are retrieving it from the database.

> **See Also:**
>
> *Oracle XML DB Developer's Guide* for more information on this function

Examples

The `XMLTransform` function requires the existence of an XSL style sheet. Here is an example of a very simple style sheet that alphabetizes elements within a node:

```
CREATE TABLE xsl_tab (col1 XMLTYPE);

INSERT INTO xsl_tab VALUES (
   XMLTYPE.createxml(
   '<?xml version="1.0"?>
    <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
```

```
      <xsl:output encoding="utf-8"/>
      <!-- alphabetizes an xml tree -->
      <xsl:template match="*">
        <xsl:copy>
          <xsl:apply-templates select="*|text()">
            <xsl:sort select="name(.)" data-type="text" order="ascending"/>
          </xsl:apply-templates>
        </xsl:copy>
      </xsl:template>
      <xsl:template match="text()">
        <xsl:value-of select="normalize-space(.)"/>
      </xsl:template>
    </xsl:stylesheet>  '));

1 row created.
```

The next example uses the `xsl_tab` XSL style sheet to alphabetize the elements in one `warehouse_spec` of the sample table `oe.warehouses`:

```
SELECT XMLTRANSFORM(w.warehouse_spec, x.col1).GetClobVal()
   FROM warehouses w, xsl_tab x
   WHERE w.warehouse_name = 'San Francisco';

XMLTRANSFORM(W.WAREHOUSE_SPEC,X.COL1).GETCLOBVAL()
--------------------------------------------------------------------------------
<Warehouse>
  <Area>50000</Area>
  <Building>Rented</Building>
  <DockType>Side load</DockType>
  <Docks>1</Docks>
  <Parking>Lot</Parking>
  <RailAccess>N</RailAccess>
  <VClearance>12 ft</VClearance>
  <WaterAccess>Y</WaterAccess>
</Warehouse>
```

CEIL, FLOOR, ROUND, and TRUNC Date Functions

Table 7-15 lists the format models you can use with the `CEIL`, `FLOOR`, `ROUND`, and `TRUNC` date functions and the units to which they round and truncate dates. The default model, 'DD', returns the date rounded or truncated to the day with a time of midnight.

Table 7-15 Date Format Models for the CEIL, FLOOR, ROUND, and TRUNC Date Functions

Format Model	Rounding or Truncating Unit
`CC` `SCC`	One greater than the first two digits of a four-digit year

`SYYYY`
`YYYY`
`YEAR`
`SYEAR`
`YYY`
`YY`
`Y`
	Year (rounds up on July 1)

Table 7-15 (Cont.) Date Format Models for the CEIL, FLOOR, ROUND, and TRUNC Date Functions

Format Model	Rounding or Truncating Unit

`IYYY`
`IY`
`IY`
`I`
	Year containing the calendar week, as defined by the ISO 8601 standard

`Q`	Quarter (rounds up on the sixteenth day of the second month of the quarter)

`MONTH`
`MON`
`MM`
`RM`
	Month (rounds up on the sixteenth day)

`WW`	Same day of the week as the first day of the year
`IW`	Same day of the week as the first day of the calendar week as defined by the ISO 8601 standard, which is Monday
`W`	Same day of the week as the first day of the month

`DDD`
`DD`
`J`
	Day

`DAY`
`DY`
`D`
	Starting day of the week

`HH`
`HH12`
`HH24`
	Hour

`MI`	Minute

The starting day of the week used by the format models DAY, DY, and D is specified implicitly by the initialization parameter `NLS_TERRITORY`.

> **See Also:**
>
> *Oracle Database Reference* and *Oracle Database Globalization Support Guide* for information on this parameter

About User-Defined Functions

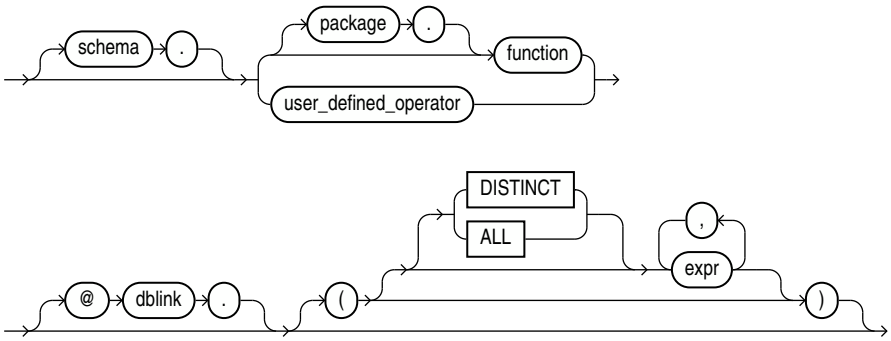
You can write user-defined functions in PL/SQL, Java, or C to provide functionality that is not available in SQL or SQL built-in functions. User-defined functions can appear in a SQL statement wherever an expression can occur.

For example, user-defined functions can be used in the following:

- The select list of a `SELECT` statement
- The condition of a `WHERE` clause
- `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses
- The `VALUES` clause of an `INSERT` statement
- The `SET` clause of an `UPDATE` statement

> **Note:**
>
> Oracle SQL does not support calling of functions with Boolean parameters or returns. Therefore, if your user-defined functions will be called from SQL statements, you must design them to return numbers (0 or 1) or character strings ('`TRUE`' or '`FALSE`').

***user_defined_function*::=**



The optional expression list must match attributes of the function, package, or operator.

Restriction on User-defined Functions

The `DISTINCT` and `ALL` keywords are valid only with a user-defined aggregate function.

> **See Also:**
>
> - CREATE FUNCTION for information on creating functions, including restrictions on user-defined functions
> - *Oracle Database Development Guide* for a complete discussion of the creation and use of user functions

Prerequisites

User-defined functions must be created as top-level functions or declared with a package specification before they can be named within a SQL statement.

To use a user function in a SQL expression, you must own or have `EXECUTE` privilege on the user function. To query a view defined with a user function, you must have the `READ` or `SELECT` privilege on the view. No separate `EXECUTE` privileges are needed to select from the view.

> **See Also:**
>
> CREATE FUNCTION for information on creating top-level functions and CREATE PACKAGE for information on specifying packaged functions

Name Precedence

Within a SQL statement, the names of database columns take precedence over the names of functions with no parameters. For example, if the Human Resources manager creates the following two objects in the `hr` schema:

```
CREATE TABLE new_emps (new_sal NUMBER, ...);
CREATE FUNCTION new_sal RETURN NUMBER IS BEGIN ... END;
```

then in the following two statements, the reference to `new_sal` refers to the column `new_emps.new_sal`:

```
SELECT new_sal FROM new_emps;
SELECT new_emps.new_sal FROM new_emps;
```

To access the function `new_sal`, you would enter:

```
SELECT hr.new_sal FROM new_emps;
```

Here are some sample calls to user functions that are allowed in SQL expressions:

```
circle_area (radius)
payroll.tax_rate (empno)
hr.employees.tax_rate (dependent, empno)@remote
```

Example

To call the `tax_rate` user function from schema `hr`, execute it against the `ss_no` and `sal` columns in `tax_table`, specify the following:

```
SELECT hr.tax_rate (ss_no, sal)
    INTO income_tax
    FROM tax_table WHERE ss_no = tax_id;
```

The `INTO` clause is PL/SQL that lets you place the results into the variable `income_tax`.

Naming Conventions

If only one of the optional schema or package names is given, then the first identifier can be either a schema name or a package name. For example, to determine whether `PAYROLL` in the

reference `PAYROLL.TAX_RATE` is a schema or package name, Oracle Database proceeds as follows:

1. Check for the `PAYROLL` package in the current schema.
2. If a `PAYROLL` package is not found, then look for a schema name `PAYROLL` that contains a top-level `TAX_RATE` function. If no such function is found, then return an error.
3. If the `PAYROLL` package is found in the current schema, then look for a `TAX_RATE` function in the `PAYROLL` package. If no such function is found, then return an error.

You can also refer to a stored top-level function using any synonym that you have defined for it.