



3 Operators, Functions, Expressions, Conditions

This chapter describes methods of manipulating individual data items. Standard arithmetic operators such as addition and subtraction are discussed, as well as less common functions such as absolute value and string length. Topics include:

- [Operators](#)
- [SQL Functions](#)
- [User Functions](#)
- [Format Models](#)
- [Expressions](#)
- [Conditions](#)

Note:

Functions, expressions, and descriptions preceded by  are available only if the Oracle objects option is installed on your database server.

Operators

An operator manipulates individual data items and returns a result. The data items are called *operands* or *arguments*. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*) and the operator that tests for nulls is represented by the keywords IS NULL. Tables in this section list SQL operators.

Unary and Binary Operators

There are two general classes of operators:

unary A unary operator operates on only one operand. A unary operator typically appears with its operand in this format:

operator operand

binary A binary operator operates on two operands. A binary operator appears with its operands in this format:

operand1 operator operand2

Other operators with special formats accept more than two operands. If an operator is given a null operand, the result

is always null. The only operator that does not follow this rule is concatenation (||).

Precedence

Precedence is the order in which Oracle evaluates different operators in the same expression. When evaluating an expression containing multiple operators, Oracle evaluates operators with higher precedence before evaluating those with lower precedence. Oracle evaluates operators with equal precedence from left to right within an expression.

[Table 3-1](#) lists the levels of precedence among SQL operators from high to low. Operators listed on the same line have the same precedence.

Table 3-1 SQL Operator Precedence

Operator	Operation
+, -	identity, negation
*, /	multiplication, division
+, -,	addition, subtraction, concatenation
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	comparison
NOT	exponentiation, logical negation
AND	conjunction
OR	disjunction

Example

In the following expression multiplication has a higher precedence than addition, so Oracle first multiplies 2 by 3 and then adds the result to 1.

1+2*3

You can use parentheses in an expression to override operator precedence. Oracle evaluates expressions inside parentheses before evaluating those outside.

SQL also supports set operators (UNION, UNION ALL, INTERSECT, and MINUS), which combine sets of rows returned by queries, rather than individual data items. All set operators have equal precedence.

Arithmetic Operators

You can use an arithmetic operator in an expression to negate, add, subtract, multiply, and divide numeric values. The result of the operation is also a numeric value. Some of these operators are also used in date arithmetic. [Table 3-2](#) lists arithmetic operators.

Table 3-2 Arithmetic Operators

Operator	Purpose	Example
+ -	Denotes a positive or negative expression. These are unary operators.	<pre>SELECT * FROM orders WHERE qtysold = -1; SELECT * FROM emp WHERE -sal < 0;</pre>
* /	Multiplies, divides. These are binary operators.	<pre>UPDATE emp SET sal = sal * 1.1;</pre>
+ -	Adds, subtracts. These are binary operators.	<pre>SELECT sal + comm FROM emp WHERE SYSDATE - hiredate > 365;</pre>

Do not use two consecutive minus signs with no separation (--) in arithmetic expressions to indicate double negation or the subtraction of a negative value. The characters -- are used to begin comments within SQL statements. You should separate consecutive minus signs with a space or a parenthesis. For more information on comments within SQL statements, see ["Comments"](#).

Concatenation Operator

The concatenation operator manipulates character strings. [Table 3-3](#) describes the concatenation operator.

Table 3-3 Concatenation Operator

Operator	Purpose	Example
	Concatenates character strings.	<pre>SELECT 'Name is ' ename FROM emp;</pre>

The result of concatenating two character strings is another character string. If both character strings are of datatype CHAR, the result has datatype CHAR and is limited to 2000 characters. If either string is of datatype VARCHAR2, the result has datatype VARCHAR2 and is limited to 4000 characters. Trailing blanks in character strings are preserved by concatenation, regardless of the strings' datatypes. For more information on the differences between the CHAR and VARCHAR2 datatypes, see ["Character Datatypes"](#).

On most platforms, the concatenation operator is two solid vertical bars, as shown in [Table 3-3](#). However, some IBM platforms use broken vertical bars for this operator. When moving SQL script files between systems having different character sets, such as between ASCII and EBCDIC, vertical bars might not be translated into the vertical bar required by the target Oracle environment. Oracle provides the CONCAT character function as an alternative to the vertical bar operator for cases when it is difficult or impossible to control translation performed by operating system or network utilities. Use this function in applications that will be moved between environments with differing character sets.

Although Oracle treats zero-length character strings as nulls, concatenating a zero-length character string with another operand always results in the other operand, so null can result only from the concatenation of two null strings. However, this may not continue to be true in future versions of Oracle. To concatenate an expression that might be null, use the NVL function to explicitly convert the expression to a zero-length string.

Example

This example creates a table with both CHAR and VARCHAR2 columns, inserts values both with and without trailing blanks, and then selects these values, concatenating them. Note that for both CHAR and VARCHAR2 columns, the trailing blanks are preserved.

```
CREATE TABLE tab1 (col1 VARCHAR2(6), col2 CHAR(6),
                   col3 VARCHAR2(6), col4 CHAR(6) );
```

Table created.

```
INSERT INTO tab1 (col1, col2, col3, col4)
VALUES ('abc', 'def ', 'ghi ', 'jkl');
```

1 row created.

```
SELECT col1||col2||col3||col4 "Concatenation"
FROM tab1;
```

Concatenation

```
-----
abcdef  ghi   jkl
```

Comparison Operators

Comparison operators compare one expression with another. The result of such a comparison can be TRUE, FALSE, or UNKNOWN. For information on conditions, see ["Conditions"](#). [Table 3-4](#) lists comparison operators.

Table 3-4 Comparison Operators

Operator	Purpose	Example
=	Equality test.	SELECT * FROM emp WHERE sal = 1500;
!= ^= <> ¬=	Inequality test. Some forms of the inequality operator may be unavailable on some platforms.	SELECT * FROM emp WHERE sal != 1500;
> <	"Greater than" and "less than" tests.	SELECT * FROM emp WHERE sal > 1500; SELECT * FROM emp WHERE sal < 1500;
>= <=	"Greater than or equal to" and "less than or equal to" tests.	SELECT * FROM emp WHERE sal >= 1500; SELECT * FROM emp WHERE sal <= 1500;
IN	"Equal to any member of" test. Equivalent to "= ANY".	SELECT * FROM emp WHERE job IN

		<pre>('CLERK', 'ANALYST'); SELECT * FROM emp WHERE sal IN (SELECT sal FROM emp WHERE deptno = 30);</pre>
NOT IN	Equivalent to "!=ALL". Evaluates to FALSE if any member of the set is NULL.	<pre>SELECT * FROM emp WHERE sal NOT IN (SELECT sal FROM emp WHERE deptno = 30); SELECT * FROM emp WHERE job NOT IN ('CLERK', 'ANALYST');</pre>
ANY SOME	Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=.	<pre>SELECT * FROM emp WHERE sal = ANY (SELECT sal FROM emp WHERE deptno = 30);</pre>
ALL	Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=.	<pre>SELECT * FROM emp WHERE sal >= ALL (1400, 3000);</pre>
[NOT] BETWEEN x AND y	[Not] greater than or equal to <i>x</i> and less than or equal to <i>y</i> .	<pre>SELECT * FROM emp WHERE sal BETWEEN 2000 AND 3000;</pre>
EXISTS	TRUE if a subquery returns at least one row.	<pre>SELECT ename, deptno FROM dept WHERE EXISTS (SELECT * FROM emp WHERE dept.deptno = emp.deptno);</pre>
x [NOT] LIKE y [ESCAPE 'z']	TRUE if <i>x</i> does [not] match the pattern <i>y</i> . Within <i>y</i> , the character "%" matches any string of zero or more characters except null. The character "_" matches any single character. Any character, excepting percent (%) and underbar (_) may follow ESCAPE; a wildcard character is treated as a literal if preceded by the character designated as the escape character.	See "LIKE Operator" . <pre>SELECT * FROM tabl WHERE coll LIKE 'A_C/%E%' ESCAPE '/';</pre>
IS [NOT] NULL	Tests for nulls. This is the only operator that you should use to test for nulls. See "Nulls" .	<pre>SELECT ename, deptno FROM emp WHERE comm IS NULL;</pre>

Additional information on the NOT IN and LIKE operators appears in the sections that follow.

NOT IN Operator

If any item in the list following a NOT IN operation is null, all rows evaluate to UNKNOWN (and no rows are returned). For example, the following statement returns the string 'TRUE' for each row:

```
SELECT 'TRUE'
FROM emp
WHERE deptno NOT IN (5,15);
```

However, the following statement returns no rows:

```
SELECT 'TRUE'
FROM emp
WHERE deptno NOT IN (5,15,null);
```

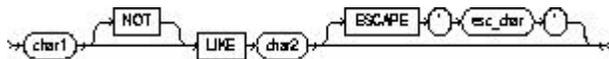
The above example returns no rows because the WHERE clause condition evaluates to:

```
deptno != 5 AND deptno != 15 AND deptno != null
```

Because all conditions that compare a null result in a null, the entire expression results in a null. This behavior can easily be overlooked, especially when the NOT IN operator references a subquery.

LIKE Operator

The LIKE operator is used in character string comparisons with pattern matching. The syntax for a condition using the LIKE operator is shown in this diagram:



where:

- char1* is a value to be compared with a pattern. This value can have datatype CHAR or VARCHAR2.
- NOT logically inverts the result of the condition, returning FALSE if the condition evaluates to TRUE and TRUE if it evaluates to FALSE.
- char2* is the pattern to which *char1* is compared. The pattern is a value of datatype CHAR or VARCHAR2 and can contain the special pattern matching characters % and _.
- ESCAPE identifies a single character as the escape character. The escape character can be used to cause Oracle to interpret % or _ literally, rather than as a special character, in the pattern.

If you wish to search for strings containing an escape character, you must specify this character twice. For example, if the escape character is '/', to search for the string 'client/server', you must specify, 'client//server'.

While the equal (=) operator exactly matches one character value to another, the LIKE operator matches a portion of one character value to another by searching the first value for the pattern specified by the second. Note that blank padding is *not* used for LIKE comparisons.

With the LIKE operator, you can compare a value to a pattern rather than to a constant. The pattern can only appear after the LIKE keyword. For example, you can issue the following query to find the salaries of all employees with names beginning with 'SM':

```
SELECT sal
FROM emp
```

```
WHERE ename LIKE 'SM%';
```

The following query uses the = operator, rather than the LIKE operator, to find the salaries of all employees with the name 'SM%':

```
SELECT sal
   FROM emp
  WHERE ename = 'SM%';
```

The following query finds the salaries of all employees with the name 'SM%'. Oracle interprets 'SM%' as a text literal, rather than as a pattern, because it *precedes* the LIKE operator:

```
SELECT sal
   FROM emp
  WHERE 'SM%' LIKE ename;
```

Patterns usually use special characters that Oracle matches with different characters in the value:

- An underscore (_) in the pattern matches exactly one character (as opposed to one byte in a multibyte character set) in the value.
- A percent sign (%) in the pattern can match zero or more characters (as opposed to bytes in a multibyte character set) in the value. Note that the pattern '%' cannot match a null.

Case Sensitivity and Pattern Matching

Case is significant in all conditions comparing character expressions including the LIKE and equality (=) operators. You can use the UPPER() function to perform a case-insensitive match, as in this condition:

```
UPPER(ename) LIKE 'SM%'
```

Pattern Matching on Indexed Columns

When LIKE is used to search an indexed column for a pattern, Oracle can use the index to improve the statement's performance if the leading character in the pattern is not "%" or "_". In this case, Oracle can scan the index by this leading character. If the first character in the pattern is "%" or "_", the index cannot improve the query's performance because Oracle cannot scan the index.

Example 1

This condition is true for all ENAME values beginning with "MA":

```
ename LIKE 'MA%'
```

All of these ENAME values make the condition TRUE:

```
MARTIN, MA, MARK, MARY
```

Case is significant, so ENAME values beginning with "Ma," "ma," and "mA" make the condition FALSE.

Example 2

Consider this condition:

```
ename LIKE 'SMITH_'
```

This condition is true for these ENAME values:

```
SMITHE, SMITHY, SMITHS
```

This condition is false for 'SMITH', since the special character "_" must match exactly one character of the ENAME value.

ESCAPE Option

You can include the actual characters "%" or "_" in the pattern by using the ESCAPE option. The ESCAPE option identifies the escape character. If the escape character appears in the pattern before the character "%" or "_" then Oracle interprets this character literally in the pattern, rather than as a special pattern matching character.

Example:

To search for any employees with the pattern 'A_B' in their name:

```
SELECT ename
FROM emp
WHERE ename LIKE '%A\_B%' ESCAPE '\';
```

The ESCAPE option identifies the backslash (\) as the escape character. In the pattern, the escape character precedes the underscore (_). This causes Oracle to interpret the underscore literally, rather than as a special pattern matching character.

Patterns Without %

If a pattern does not contain the "%" character, the condition can be TRUE only if both operands have the same length.

Example:

Consider the definition of this table and the values inserted into it:

```
CREATE TABLE fredS (f CHAR(6), v VARCHAR2(6));
INSERT INTO fredS VALUES ('FRED', 'FRED');
```

Because Oracle blank-pads CHAR values, the value of F is blank-padded to 6 bytes. V is not blank-padded and has length 4.

Logical Operators

A logical operator combines the results of two component conditions to produce a single result based on them or to invert the result of a single condition. [Table 3-5](#) lists logical operators.

Table 3-5 Logical Operators

Operator	Function	Example
NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	<pre>SELECT * FROM emp WHERE NOT (job IS NULL); SELECT * FROM emp WHERE NOT (sal BETWEEN 1000 AND 2000);</pre>
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE. Otherwise returns UNKNOWN.	<pre>SELECT * FROM emp WHERE job = 'CLERK' AND deptno = 10;</pre>
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise returns UNKNOWN.	<pre>SELECT * FROM emp WHERE job = 'CLERK' OR deptno = 10;</pre>

For example, in the WHERE clause of the following SELECT statement, the AND logical operator is used to ensure that only those hired before 1984 and earning more than \$1000 a month are returned:

```
SELECT *
  FROM emp
 WHERE hiredate < TO_DATE('01-JAN-1984', 'DD-MON-YYYY')
 AND sal > 1000;
```

NOT Operator

[Table 3-6](#) shows the result of applying the NOT operator to a condition.

Table 3-6 NOT Truth Table

NOT	TRUE	FALSE	UNKNOWN
	FALSE	TRUE	UNKNOWN

AND Operator

[Table 3-7](#) shows the results of combining two expressions with AND.

Table 3-7 AND Truth Table

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR Operator

[Table 3-8](#) shows the results of combining two expressions with OR.

Table 3-8 OR Truth Table

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Set Operators

Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries. [Table 3-9](#) lists SQL set operators.

Table 3-9 Set Operators

Operator	Returns
UNION	All rows selected by either query.
UNION ALL	All rows selected by either query, including all duplicates.
INTERSECT	All distinct rows selected by both queries.
MINUS	All distinct rows selected by the first query but not the second.

All set operators have equal precedence. If a SQL statement contains multiple set operators, Oracle evaluates them from the left to right if no parentheses explicitly specify another order. To comply with emerging SQL standards, a future release of Oracle will give the INTERSECT operator greater precedence than the other set operators. Therefore, you should use parentheses to specify order of evaluation in queries that use the INTERSECT operator with other set operators.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and datatype. If component queries select character data, the datatype of the return values are determined as follows:

- If both queries select values of datatype CHAR, the returned values have datatype CHAR.
- If either or both of the queries select values of datatype VARCHAR2, the returned values have datatype VARCHAR2.

Examples

Consider these two queries and their results:

```
SELECT part
   FROM orders_list1;
```

```
PART
-----
SPARKPLUG
FUEL PUMP
FUEL PUMP
TAILPIPE
```

```

SELECT part
   FROM orders_list2;

PART
-----
CRANKSHAFT
TAILPIPE
TAILPIPE

```

The following examples combine the two query results with each of the set operators.

UNION Example

The following statement combines the results with the UNION operator, which eliminates duplicate selected rows. This statement shows how datatype must match when columns do not exist in one or the other table:

```

SELECT part, partnum, to_date(null) date_in
   FROM orders_list1
UNION
SELECT part, to_null(null), date_in
   FROM orders_list2;

```

```

PART          PARTNUM DATE_IN
-----
SPARKPLUG    3323165
SPARKPLUG           10/24/98
FUEL PUMP    3323162
FUEL PUMP           12/24/99
TAILPIPE     1332999
TAILPIPE           01/01/01
CRANKSHAFT   9394991
CRANKSHAFT           09/12/02

```

```

SELECT part
   FROM orders_list1
UNION
SELECT part
   FROM orders_list2;

```

```

PART
-----
SPARKPLUG
FUEL PUMP
TAILPIPE
CRANKSHAFT

```

UNION ALL Example

The following statement combines the results with the UNION ALL operator, which does not eliminate duplicate selected rows:

```

SELECT part
   FROM orders_list1
UNION ALL
SELECT part
   FROM orders_list2;

```

```

PART
-----
SPARKPLUG

```

```
FUEL PUMP
FUEL PUMP
TAILPIPE
CRANKSHAFT
TAILPIPE
TAILPIPE
```

Note that the UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. A PART value that appears multiple times in either or both queries (such as 'FUEL PUMP') is returned only once by the UNION operator, but multiple times by the UNION ALL operator.

INTERSECT Example

The following statement combines the results with the INTERSECT operator which returns only those rows returned by both queries:

```
SELECT part
   FROM orders_list1
INTERSECT
SELECT part
   FROM orders_list2;
```

```
PART
-----
TAILPIPE
```

MINUS Example

The following statement combines results with the MINUS operator, which returns only rows returned by the first query but not by the second:

```
SELECT part
   FROM orders_list1
MINUS
SELECT part
   FROM orders_list2;
```

```
PART
-----
SPARKPLUG
FUEL PUMP
```

Other Operators

[Table 3-10](#) lists other SQL operators.

Table 3-10 Other SQL Operators

Operator	Purpose	Example
(+)	Indicates that the preceding column is the outer join column in a join. See "Outer Joins" .	<pre>SELECT ename, dname FROM emp, dept WHERE dept.deptno = emp.deptno(+);</pre>
PRIOR	Evaluates the following expression for the parent row of the current row in a hierarchical, or tree-structured, query. In such a query, you must use this operator in the CONNECT BY clause to define the relationship between parent and child rows. You can also use this operator in other parts of a SELECT statement that performs a hierarchical query. The PRIOR operator is a unary operator and has the same precedence as the unary + and - arithmetic operators. See "Hierarchical Queries" .	<pre>SELECT empno, ename, mgr FROM emp CONNECT BY PRIOR empno = mgr;</pre>

SQL Functions

A SQL function is similar to an operator in that it manipulates data items and returns a result. SQL functions differ from operators in the format in which they appear with their arguments. This format allows them to operate on zero, one, two, or more arguments:

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

If you call a SQL function with an argument of a datatype other than the datatype expected by the SQL function, Oracle implicitly converts the argument to the expected datatype before performing the SQL function. See ["Data Conversion"](#).

If you call a SQL function with a null argument, the SQL function automatically returns null. The only SQL functions that do not follow this rule are CONCAT, DECODE, DUMP, NVL, and REPLACE.

SQL functions should not be confused with user functions written in PL/SQL. User functions are described in ["User Functions"](#).

In the syntax diagrams for SQL functions, arguments are indicated with their datatypes following the conventions described in ["Syntax Diagrams and Notation"](#) in the Preface of this reference.

SQL functions are of these general types:

- single-row (or scalar) functions
- group (or aggregate) functions

The two types of SQL functions differ in the number of rows upon which they act. A single-row function returns a single result row for every row of a queried table or view; a group function returns a single result row for a group of queried rows.

Single-row functions can appear in select lists (if the SELECT statement does not contain a GROUP BY clause), WHERE clauses, START WITH clauses, and CONNECT BY clauses.

Group functions can appear in select lists and HAVING clauses. If you use the GROUP BY clause in a SELECT statement, Oracle divides the rows of a queried table or view into groups. In a query containing a GROUP BY clause, all elements of the select list must be expressions from the GROUP BY clause, expressions containing group

functions, or constants. Oracle applies the group functions in the select list to each group of rows and returns a single result row for each group.

If you omit the **GROUP BY** clause, Oracle applies group functions in the select list to all the rows in the queried table or view. You use group functions in the **HAVING** clause to eliminate groups from the output based on the results of the group functions, rather than on the values of the individual rows of the queried table or view. For more information on the **GROUP BY** and **HAVING** clauses, see the [GROUP BY Clause](#) and the [HAVING Clause](#).

In the sections that follow, functions are grouped by the datatypes of their arguments and return values.

Number Functions

Number functions accept numeric input and return numeric values. This section lists the SQL number functions. Most of these functions return 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.

ABS

Purpose Returns the absolute value of *n*.

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

```

      Absolute
-----
             15

```

ACOS

Purpose Returns the arc cosine of *n*. Inputs are in the range of -1 to 1, and outputs are in the range of 0 to π and are expressed in radians.

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

```

Arc_Cosine
-----
1.26610367

```

ASIN

Purpose Returns the arc sine of n . Inputs are in the range of -1 to 1, and outputs are in the range of $-\pi/2$ to $\pi/2$ and are expressed in radians.

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

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

ATAN

Purpose Returns the arc tangent of n . Inputs are in an unbounded range, and outputs are in the range of $-\pi/2$ to $\pi/2$ and are expressed in radians.

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

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

ATAN2

Purpose Returns the arc tangent of n and m . Inputs are in an unbounded range, and outputs are in the range of $-\pi$ to π , depending on the signs of n and m , and are expressed in radians. `Atan2(n,m)` is the same as `atan2(n/m)`

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

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

CEIL

Purpose Returns smallest integer greater than or equal to n .

Example `SELECT CEIL(15.7) "Ceiling" FROM DUAL;`

```

      Ceiling
-----
           16

```

COS

Purpose Returns the cosine of n (an angle expressed in radians).

Example `SELECT COS(180 * 3.14159265359/180) "Cosine of 180 degrees" FROM DUAL;`

```

Cosine of 180 degrees
-----
                    -1

```

COSH

Purpose Returns the hyperbolic cosine of n .

Example `SELECT COSH(0) "Hyperbolic cosine of 0" FROM DUAL;`

```

Hyperbolic cosine of 0
-----
                       1

```

EXP

Purpose Returns e raised to the n th power; $e = 2.71828183 \dots$

Example `SELECT EXP(4) "e to the 4th power" FROM DUAL;`

```

e to the 4th power
-----
          54.59815

```

FLOOR

Purpose Returns largest integer equal to or less than n .

Example `SELECT FLOOR(15.7) "Floor" FROM DUAL;`

```

      Floor
-----
         15

```

LN

Purpose Returns the natural logarithm of n , where n is greater than 0.

Example `SELECT LN(95) "Natural log of 95" FROM DUAL;`

```

Natural log of 95
-----
      4.55387689

```

LOG

Purpose Returns the logarithm, base m , of n . The base m can be any positive number other than 0 or 1 and n can be any positive number.

Example `SELECT LOG(10,100) "Log base 10 of 100" FROM DUAL;`

```

Log base 10 of 100
-----
                   2

```

MOD

Syntax `MOD(m,n)`

Purpose Returns remainder of *m* divided by *n*. Returns *m* if *n* is 0.

Example `SELECT MOD(11,4) "Modulus" FROM DUAL;`

```

Modulus
-----
      3

```

This function behaves differently from the classical mathematical modulus function when *m* is negative. The classical modulus can be expressed using the MOD function with this formula:

$$m - n * \text{FLOOR}(m/n)$$

The following statement illustrates the difference between the MOD function and the classical modulus:

```

SELECT m, n, MOD(m, n),
       m - n * FLOOR(m/n) "Classical Modulus"
FROM test_mod_table;

```

M	N	MOD(M,N)	Classical Modulus
11	4	3	3
11	-4	3	-1
-11	4	-3	1
-11	-4	-3	-3

POWER

Purpose Returns *m* raised to the *n*th power. The base *m* and the exponent *n* can be any numbers, but if *m* is negative, *n* must be an integer.

Example `SELECT POWER(3,2) "Raised" FROM DUAL;`

```

Raised
-----
      9

```

ROUND

Syntax `ROUND(n[,m])`

Purpose Returns *n* rounded to *m* places right of the decimal point; if *m* is omitted, to 0 places. *m* can be negative to round off digits left of the decimal point. *m* must be an integer.

Example 1 `SELECT ROUND(15.193,1) "Round" FROM DUAL;`

```

      Round
-----
     15.2

```

Example 2 `SELECT ROUND(15.193,-1) "Round" FROM DUAL;`

```

      Round
-----
      20

```

SIGN

Syntax `SIGN(n)`

Purpose If $n < 0$, the function returns -1; if $n = 0$, the function returns 0; if $n > 0$, the function returns 1.

Example `SELECT SIGN(-15) "Sign" FROM DUAL;`

```

      Sign
-----
     -1

```

SIN

Purpose Returns the sine of *n* (an angle expressed in radians).

Example `SELECT SIN(30 * 3.14159265359/180)`
 `"Sine of 30 degrees" FROM DUAL;`

```

Sine of 30 degrees
-----
                .5

```

SINH

Purpose Returns the hyperbolic sine of n .

Example `SELECT SINH(1) "Hyperbolic sine of 1" FROM DUAL;`

```
Hyperbolic sine of 1
-----
                1.17520119
```

SQRT

Purpose Returns square root of n . The value n cannot be negative. SQRT returns a "real" result.

Example `SELECT SQRT(26) "Square root" FROM DUAL;`

```
Square root
-----
5.09901951
```

TAN

Purpose Returns the tangent of n (an angle expressed in radians).

Example `SELECT TAN(135 * 3.14159265359/180)
"Tangent of 135 degrees" FROM DUAL;`

```
Tangent of 135 degrees
-----
                - 1
```

TANH

Purpose Returns the hyperbolic tangent of n .

Example `SELECT TANH(.5) "Hyperbolic tangent of .5"
FROM DUAL;`

```
Hyperbolic tangent of .5
-----
                .462117157
```

TRUNC

Purpose Returns n truncated to m decimal places; if m is omitted, to 0 places. m can be negative to truncate (make zero) m digits left of the decimal point.

Examples `SELECT TRUNC(15.79,1) "Truncate" FROM DUAL;`

```
Truncate
-----
      15.7
```

`SELECT TRUNC(15.79,-1) "Truncate" FROM DUAL;`

```
Truncate
-----
      10
```

Character Functions

Single-row character functions accept character input and can return either character or number values.

Character Functions Returning Character Values

This section lists character functions that return character values. Unless otherwise noted, these functions all return values with the datatype VARCHAR2 and are limited in length to 4000 bytes. Functions that return values of datatype CHAR are limited in length to 2000 bytes. If the length of the return value exceeds the limit, Oracle truncates it and returns the result without an error message.

CHR

Syntax CHR(*n* [USING NCHAR_CS])

Purpose Returns the character having the binary equivalent to *n* in either the database character set or the national character set.

If the USING NCHAR_CS clause is *not* specified, this function returns the character having the binary equivalent to *n* as a VARCHAR2 value in the database character set.

If the USING NCHAR_CS clause is specified, this function returns the character having the binary equivalent to *n* as a NVARCHAR2 value in the national character set.

Example 1

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

Example 2

```
SELECT CHR(16705 USING NCHAR_CS) FROM DUAL;
C
-
A
```

CONCAT

Syntax CONCAT(*char1*, *char2*)

Purpose Returns *char1* concatenated with *char2*. This function is equivalent to the concatenation operator (||). For information on this operator, see "[Concatenation Operator](#)".

Example This example uses nesting to concatenate three character strings:

```
SELECT CONCAT( CONCAT(ename, ' is a '), job) "Job"
FROM emp
WHERE empno = 7900;

Job
-----
JAMES is a CLERK
```

INITCAP

Purpose 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.

Example `SELECT INITCAP('the soap') "Capitals" FROM DUAL;`

```
Capitals
-----
The Soap
```

LOWER

Purpose Returns *char*, with all letters lowercase. The return value has the same datatype as the argument *char* (CHAR or VARCHAR2).

Example `SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase" FROM DUAL;`

```
Lowercase
-----
mr. scott mcmillan
```

LPAD

Purpose Returns *char1*, left-padded to length *n* with the sequence of characters in *char2*; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* 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.

Example `SELECT LPAD('Page 1',15,'*.*') "LPAD example" FROM DUAL;`

```
LPAD example
-----
*.*.*.*.*Page 1
```

LTRIM

Syntax LTRIM(*char* [*,set*])

Purpose Removes characters from the left of *char*, with all the leftmost characters that appear in *set* removed; *set* defaults to a single blank. Oracle 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.

Example SELECT LTRIM('xyxXxyLAST WORD', 'xy') "LTRIM example"
 FROM DUAL;

```
LTRIM exampl
-----
XxyLAST WORD
```

NLS_INITCAP

Purpose 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. The value of '*nlparams*' can have this form:

```
'NLS_SORT = sort'
```

where *sort* is either a linguistic sort sequence or BINARY. The linguistic sort sequence handles special linguistic requirements for case conversions. Note that these requirements can result in a return value of a different length than the *char*. If you omit '*nlparams*', this function uses the default sort sequence for your session. For information on sort sequences, see [Oracle8 Reference](#).

Example SELECT NLS_INITCAP
 ('ijsland', 'NLS_SORT = XDutch') "Capitalized"
 FROM DUAL;

```
Capital
-----
IJsland
```

NLS_LOWER

Syntax NLS_LOWER(char [, 'nlsparams'])

Purpose Returns *char*, with all letters lowercase. The '*nlsparams*' can have the same form and serve the same purpose as in the NLS_INITCAP function.

Example

```
SELECT NLS_LOWER
      ('CITTA''', 'NLS_SORT = XGerman') "Lowercase"
FROM DUAL;
```

```
Lower
-----
cittá
```

NLS_UPPER

Syntax NLS_UPPER(char [, 'nlsparams'])

Purpose Returns *char*, with all letters uppercase. The '*nlsparams*' can have the same form and serve the same purpose as in the NLS_INITCAP function.

Example

```
SELECT NLS_UPPER
      ('große', 'NLS_SORT = XGerman') "Uppercase"
FROM DUAL;
```

```
Upper
-----
GROSS
```

REPLACE

Syntax REPLACE(char, search_string[, replacement_string])

Purpose Returns *char* with every occurrence of *search_string* replaced with *replacement_string*. If *replacement_string* is omitted or null, all occurrences of *search_string* are removed. If *search_string* is null, *char* is returned. This function provides a superset of the functionality provided by the TRANSLATE function. TRANSLATE provides single-character, one-to-one substitution. REPLACE allows you to substitute one string for another as well as to remove character strings.

Example SELECT REPLACE('JACK and JUE', 'J', 'BL') "Changes"
FROM DUAL;

```
Changes
-----
BLACK and BLUE
```

RPAD

Syntax RPAD(char1, n [, char2])

Purpose Returns *char1*, right-padded to length *n* with *char2*, replicated as many times as necessary; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* 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.

Example SELECT RPAD('MORRISON', 12, 'ab') "RPAD example"
FROM DUAL;

```
RPAD example
-----
MORRISONabab
```

RTRIM

Syntax `RTRIM(char [,set])`

Purpose Returns *char*, with all the rightmost characters that appear in *set* removed; *set* defaults to a single blank. RTRIM works similarly to LTRIM.

Example

```
SELECT RTRIM('BROWNINGyxXxy','xy') "RTRIM e.g."
      FROM DUAL;

RTRIM e.g
-----
BROWNINGyxX
```

SOUNDEX

Syntax `SOUNDEX(char)`

Purpose Returns a character string containing the phonetic representation of *char*. This function allows you to 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:

- Retain the first letter of the string and remove all other occurrences of the following letters: a, e, h, i, o, u, w, y.
- 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
```
- If two or more letters with the same assigned number are adjacent, remove all but the first.
- Return the first four bytes padded with 0.

Example

```
SELECT ename
      FROM emp
      WHERE SOUNDEX(ename)
            = SOUNDEX('SMYTHE');
```

```

ENAME
-----
SMITH

```

SUBSTR

Syntax SUBSTR(char, m [,n])

Purpose Returns a portion of *char*, beginning at character *m*, *n* characters long. If *m* is 0, it is treated as 1. If *m* is positive, Oracle counts from the beginning of *char* to find the first character. If *m* is negative, Oracle counts backwards from the end of *char*. If *n* is omitted, Oracle returns all characters to the end of *char*. If *n* is less than 1, a null is returned.

Floating-point numbers passed as arguments to *substr* are automatically converted to integers.

Example 1 SELECT SUBSTR('ABCDEFG', 3.1, 4) "Subs"
 FROM DUAL;

```

Subs
----
CDEF

```

Example 2 SELECT SUBSTR('ABCDEFG', -5, 4) "Subs"
 FROM DUAL;

```

Subs
----
CDEF

```

SUBSTRB

Syntax SUBSTR(char, m [,n])

Purpose The same as SUBSTR, except that the arguments *m* and *n* are expressed in bytes, rather than in characters. For a single-byte database character set, SUBSTRB is equivalent to SUBSTR.

Floating-point numbers passed as arguments to *substrb* are automatically converted to integers.

Example Assume a double-byte database character set:

```
SELECT SUBSTRB('ABCDEFG',5,4.2)
       "Substring with bytes"
FROM DUAL;
```

```
Substring with bytes
-----
CD
```

TRANSLATE

Syntax TRANSLATE(char, from, to)

Purpose Returns *char* with all occurrences of each character in *from* replaced by its corresponding character in *to*. Characters in *char* that are not in *from* are not replaced. The argument *from* can contain more characters than *to*. In this case, the extra characters at the end of *from* have no corresponding characters in *to*. If these extra characters appear in *char*, they are removed from the return value. You cannot use an empty string for *to* to remove all characters in *from* from the return value. Oracle interprets the empty string as null, and if this function has a null argument, it returns null.

Example 1 The following statement translates a license number. All letters 'ABC...Z' are translated to 'X' and all digits '012...9' are translated to '9':

```
SELECT TRANSLATE('2KRW229',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'9999999999XXXXXXXXXXXXXXXXXXXXXXXXXXXXX') "License"
FROM DUAL;
```

```
License
-----
9XXX999
```

Example 2 The following statement returns a license number with the characters removed and the digits remaining:

```
SELECT TRANSLATE('2KRW229',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```

0123456789ABCDEFGHIJKLmnopqrstuvwxyz ,
'0123456789')
"Translate example"
  FROM DUAL;

Translate example
-----
2229

```

UPPER

Syntax UPPER (char)

Purpose Returns *char*, with all letters uppercase. The return value has the same datatype as the argument *char*.

```

Example  SELECT UPPER('Large') "Uppercase"
          FROM DUAL;

Upper
-----
LARGE

```

Character Functions Returning Number Values

This section lists character functions that return number values.

ASCII

Syntax ASCII (char)

Purpose Returns the decimal representation in the database character set of the first character of *char*. If your database character set is 7-bit ASCII, this function returns an ASCII value. If your database character set is EBCDIC Code Page 500, this function returns an EBCDIC value. Note that there is no similar EBCDIC character function.

```

Example  SELECT ASCII('Q')
          FROM DUAL;

ASCII('Q')
-----
      81

```

INSTR

Syntax INSTR (char1, char2 [,n[,m]])

Purpose Searches *char1* beginning with its *n*th character for the *m*th occurrence of *char2* and returns the position of the character in *char1* that is the first character of this occurrence. If *n* is negative, Oracle counts and searches backward from the end of *char1*. The value of *m* must be positive. The default values of both *n* and *m* are 1, meaning Oracle begins searching at the first character of *char1* for the first occurrence of *char2*. The return value is relative to the beginning of *char1*, regardless of the value of *n*, and is expressed in characters. If the search is unsuccessful (if *char2* does not appear *m* times after the *n*th character of *char1*) the return value is 0.

Example 1 SELECT INSTR('CORPORATE FLOOR','OR', 3, 2)
"Instring" FROM DUAL;

```
Instring
-----
          14
```

Example 2 SELECT INSTR('CORPORATE FLOOR','OR', -3, 2)
"Reversed Instring"
FROM DUAL;

```
Reversed Instring
-----
                   2
```

INSTRB

Syntax INSTRB(char1, char2[,n[,m]])

Purpose The same as INSTR, except that *n* and the return value are expressed in bytes, rather than in characters. For a single-byte database character set, INSTRB is equivalent to INSTR.

Example This example assumes a double-byte database character set.

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2)
" Instring in bytes"
FROM DUAL;
```

```
Instring in bytes
-----
                   27
```

LENGTH

Syntax `LENGTH(char)`

Purpose Returns the length of *char* in characters. If *char* has datatype CHAR, the length includes all trailing blanks. If *char* is null, this function returns null.

Example `SELECT LENGTH('CANDIDE') "Length in characters"`
`FROM DUAL;`

```
Length in characters
-----
                        7
```

LENGTHB

Syntax `LENGTHB(char)`

Purpose Returns the length of *char* in bytes. If *char* is null, this function returns null. For a single-byte database character set, LENGTHB is equivalent to LENGTH.

Example This example assumes a double-byte database character set.

```
SELECT LENGTHB ('CANDIDE') "Length in bytes"
          FROM DUAL;
```

```
Length in bytes
-----
                        14
```

NLSSORT

Syntax NLSSORT(char [, 'nlsparams'])

Purpose Returns the string of bytes used to sort *char*. The value of 'nlsparams' can have the form

```
'NLS_SORT = sort'
```

where *sort* is a linguistic sort sequence or BINARY. If you omit 'nlsparams', this function uses the default sort sequence for your session. If you specify BINARY, this function returns *char*. For information on sort sequences, see the discussions of national language support in [Oracle8 Reference](#).

Example This function can be used to specify comparisons based on a linguistic sort sequence rather on the binary value of a string:

```
SELECT ename FROM emp
WHERE NLSSORT (ename, 'NLS_SORT = German')
  > NLSSORT ('S', 'NLS_SORT = German') ORDER BY ename;

ENAME
-----
SCOTT
SMITH
TURNER
WARD
```

Date Functions

Date functions operate on values of the DATE datatype. All date functions return a value of DATE datatype, except the MONTHS_BETWEEN function, which returns a number.

ADD_MONTHS

Syntax ADD_MONTHS(d, n)

Purpose Returns the date *d* plus *n* months. The argument *n* can be any integer. If *d* is the last day of the month or if the resulting month has fewer days than the day component of *d*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *d*.

Example

```
SELECT TO_CHAR(
  ADD_MONTHS(hiredate, 1),
  'DD-MON-YYYY') "Next month"
FROM emp
WHERE ename = 'SMITH';
```

```
Next Month
-----
17-JAN-1981
```

LAST_DAY

Syntax `LAST_DAY(d)`

Purpose Returns the date of the last day of the month that contains *d*. You might use this function to determine how many days are left in the current month.

Example 1 `SELECT SYSDATE,
 LAST_DAY(SYSDATE) "Last",
 LAST_DAY(SYSDATE) - SYSDATE "Days Left"
 FROM DUAL;`

SYSDATE	Last	Days Left
-----	-----	-----
23-OCT-97	31-OCT-97	8

Example 2 `SELECT TO_CHAR(
 ADD_MONTHS(
 LAST_DAY(hiredate),5),
 'DD-MON-YYYY') "Five months"
 FROM emp
 WHERE ename = 'MARTIN';`

Five months

28-FEB-1982

MONTHS_BETWEEN

Syntax `MONTHS_BETWEEN(d1, d2)`

Purpose Returns number of months between dates *d1* and *d2*. If *d1* is later than *d2*, result is positive; if earlier, negative. If *d1* and *d2* are either the same days of the month or both last days of months, the result is always an integer; otherwise Oracle calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of *d1* and *d2*.

Example `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

NEW_TIME

Syntax `NEW_TIME(d, z1, z2)`

Purpose Returns the date and time in time zone *z2* when date and time in time zone *z1* are *d*. The arguments *z1* and *z2* can be any of these text strings:

AST	Atlantic Standard or Daylight Time
ADT	
BST	Bering Standard or Daylight Time
BDT	
CST	Central Standard or Daylight Time
CDT	
EST	Eastern Standard or Daylight Time
EDT	
GMT	Greenwich Mean Time
HST	Alaska-Hawaii Standard Time or Daylight Time.
HDT	
MST	Mountain Standard or Daylight Time
MDT	
NST	Newfoundland Standard Time
PST	Pacific Standard or Daylight Time
PDT	
YST	Yukon Standard or Daylight Time
YDT	

NEXT_DAY

Syntax `NEXT_DAY(d, char)`

Purpose Returns the date of the first weekday named by *char* that is later than the date *d*. The argument *char* must be a day of the week in your session's date language-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 *d*.

Example This example returns the date of the next Tuesday after March 15, 1992.

```
SELECT NEXT_DAY( '15-MAR-92' , 'TUESDAY' ) "NEXT DAY"
      FROM DUAL;
```

```
NEXT DAY
-----
17-MAR-92
```

ROUND

Syntax `ROUND(d [, fmt])`

Purpose Returns *d* rounded to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is rounded to the nearest day. See ["ROUND and TRUNC"](#) for the permitted format models to use in *fmt*.

Example `SELECT ROUND (TO_DATE ('27-OCT-92'), 'YEAR')`
`"New Year" FROM DUAL;`

```
New Year
-----
01-JAN-93
```

SYSDATE

Syntax SYSDATE

Purpose Returns the current date and time. Requires no arguments. In distributed SQL statements, this function returns the date and time on your local database. You cannot use this function in the condition of a CHECK constraint.

```
Example SELECT TO_CHAR
        (SYSDATE, 'MM-DD-YYYY HH24:MI:SS') "NOW"
        FROM DUAL;
NOW
-----
10-29-1993 20:27:11
```

TRUNC

Syntax 1TRUNC(*d*, [*fmt*])

Purpose Returns *d* with the time portion of the day truncated to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is truncated to the nearest day. See ["ROUND and TRUNC"](#) for the permitted format models to use in *fmt*.

```
Example SELECT TRUNC(TO_DATE('27-OCT-92', 'DD-MON-YY'), 'YEAR')
        "New Year" FROM DUAL;
New Year
-----
01-JAN-92
```

ROUND and TRUNC

[Table 3-11](#) lists the format models you can use with the 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 3-11 Date Format Models for the 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	Year (rounds up on July 1)

YY Y	
IYYY IY IY I	ISO Year
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 ISO year.
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. For information on this parameter, see [Oracle8 Reference](#).

Conversion Functions

Conversion functions convert a value from one datatype to another. Generally, the form of the function names follows the convention *datatype TO datatype*. The first datatype is the input datatype; the last datatype is the output datatype.

This section lists the SQL conversion functions.

CHARTOROWID

Syntax CHARTOROWID(char)

Purpose Converts a value from CHAR or VARCHAR2 datatype to ROWID datatype.

Example SELECT ename FROM emp
 WHERE ROWID = CHARTOROWID('AAAAfZAABAAACp8AAO');

```

ENAME
-----
LEWIS

```

CONVERT

Syntax CONVERT(char , dest_char_set [,source_char_set])

Purpose Converts a character string from one character set to another.

The *char* argument is the value to be converted.

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.

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.

Example SELECT CONVERT('Groß', 'US7ASCII', 'WE8HP')
 "Conversion"
 FROM DUAL;

```

Conversion
-----
Gross

```

Common character sets include:

US7ASCII	US 7-bit ASCII character set
WE8DEC	DEC West European 8-bit character set
WE8HP	HP West European Laserjet 8-bit character set
F7DEC	DEC French 7-bit character set
WE8EBCDIC500	IBM West European EBCDIC Code Page 500
WE8PC850	IBM PC Code Page 850
WE8ISO8859P1	ISO 8859-1 West European 8-bit character set

HEXTORAW

Syntax HEXTORAW(char)

Purpose Converts *char* containing hexadecimal digits to a raw value.

Example

```
INSERT INTO graphics (raw_column)
SELECT HEXTORAW('7D') FROM DUAL;
```

RAWTOHEX

Syntax RAWTOHEX(raw)

Purpose Converts *raw* to a character value containing its hexadecimal equivalent.

Example

```
SELECT RAWTOHEX(raw_column) "Graphics"
FROM graphics;
```

```
Graphics
-----
7D
```

ROWIDTOCHAR

Syntax ROWIDTOCHAR(*rowid*)

Purpose Converts a ROWID value to VARCHAR2 datatype. The result of this conversion is always 18 characters long.

Example

```
SELECT ROWID
       FROM offices
       WHERE
          ROWIDTOCHAR(ROWID) LIKE '%Br1AAB%';

ROWID
-----
AAAAZ6AABAAABr1AAB
```

TO_CHAR, date conversion

Syntax TO_CHAR(*d* [, *fmt* [, '*nlsparams*']])

Purpose Converts *d* of DATE datatype to a value of VARCHAR2 datatype in the format specified by the date format *fmt*. If you omit *fmt*, *d* is converted to a VARCHAR2 value in the default date format. For information on date formats, see "[Format Models](#)".

The '*nlsparams*' 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 *nlsparams*, this function uses the default date language for your session.

Example

```
SELECT TO_CHAR(HIREDATE, 'Month DD, YYYY')
       "New date format" FROM emp
       WHERE ename = 'BLAKE';

New date format
-----
May           01, 1981
```

TO_CHAR, number conversion

Syntax TO_CHAR(*n* [, *fmt* [, '*nlsparams*']])

Purpose Converts *n* of NUMBER datatype to a value of VARCHAR2 datatype, using the optional number format *fmt*. If you omit *fmt*, *n* is converted to a VARCHAR2 value exactly long enough to hold its

significant digits. For information on number formats, see "[Format Models](#)".

The *'nlsparams'* 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. Note that 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 *'nlsparams'* or any one of the parameters, this function uses the default parameter values for your session.

Example 1

In this example, the output is blank padded to the left of the currency symbol.

```
SELECT TO_CHAR(-10000, 'L99G999D99MI') "Amount "
       FROM DUAL;
```

```
Amount
-----
  $10,000.00-
```

Example 2

```
SELECT TO_CHAR(-10000, 'L99G999D99MI',
'NLS_NUMERIC_CHARACTERS = ',',.'
NLS_CURRENCY = 'AusDollars'') "Amount "
       FROM DUAL;
```

```
Amount
-----
AusDollars10.000,00-
```

Notes:

- In the optional number format *fmt*, L designates local currency symbol and MI designates a trailing minus sign. See [Table 3-13](#) for a complete listing of number format elements.
- During a conversion of Oracle numbers to string, if a rounding operation occurs that overflows or underflows the Oracle NUMBER range, then ~ or ~~ may be returned, representing infinity and negative infinity, respectively. This event typically occurs when you are using TO_CHAR() with a restrictive number format string, causing a rounding operation.

TO_DATE

Syntax TO_DATE(char [, fmt [, 'nlsparams']])

Purpose Converts *char* of CHAR or VARCHAR2 datatype to a value of DATE datatype. The *fmt* is a date format specifying the format of *char*. If you omit *fmt*, *char* must be in the default date format. If *fmt* is 'J', for Julian, then *char* must be an integer. For information on date formats, see "[Format Models](#)".

The '*nlsparams*' has the same purpose in this function as in the TO_CHAR function for date conversion.

Do not use the TO_DATE function with a DATE value for the *char* argument. The returned DATE value can have a different century value than the original *char*, depending on *fmt* or the default date format.

For information on date formats, see "[Date Format Models](#)".

Example

```
INSERT INTO bonus (bonus_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_MULTI_BYTE

Syntax `TO_MULTI_BYTE(char)`

Purpose Returns *char* with all of its single-byte characters converted to their corresponding multibyte characters. Any single-byte characters in *char* that have no multibyte equivalents appear in the output string as single-byte characters. This function is only useful if your database character set contains both single-byte and multibyte characters.

TO_NUMBER

Syntax `TO_NUMBER(char [,fmt [, 'nlsparams']])`

Purpose Converts *char*, a value of CHAR or VARCHAR2 datatype containing a number in the format specified by the optional format model *fmt*, to a value of NUMBER datatype.

Example 1

```
UPDATE emp SET sal = sal +
  TO_NUMBER('100.00', '9G999D99')
WHERE ename = 'BLAKE';
```

The '*nlsparams*' string in this function has the same purpose as it does in the TO_CHAR function for number conversions.

Example 2

```
SELECT TO_NUMBER('-AusDollars100','L9G999D99',
  ' NLS_NUMERIC_CHARACTERS = ','.'
  NLS_CURRENCY = 'AusDollars'
  ) "Amount"
FROM DUAL;
```

```
Amount
-----
-100
```

TO_SINGLE_BYTE

Syntax `TO_SINGLE_BYTE(char)`

Purpose Returns *char* with all of its multibyte character converted to their corresponding single-byte characters. Any multibyte characters in *char* that have no single-byte equivalents appear in the output as multibyte characters. This function is only useful if your database character set contains both single-byte and multibyte characters.

TRANSLATE USING

Syntax `TRANSLATE(text USING {CHAR_CS | NCHAR_CS })`

Purpose Converts *text* into the character set specified for conversions between the database character set and the national character set.

The *text* argument is the expression to be converted.

Specifying the USING CHAR_CS argument converts *text* into the database character set. The output datatype is VARCHAR2.

Specifying the USING NCHAR_CS argument converts *text* into the national character set. The output datatype 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 datatype is being used as NCHAR or NVARCHAR2.

Example `CREATE TABLE t1 (char_col CHAR(20),
1 nchar_col nchar(20));
INSERT INTO t1
VALUES ('Hi', N'Bye');
SELECT * FROM t1;`

CHAR_COL	NCHAR_COL
-----	-----
Hi	Bye

Example `UPDATE t1 SET
2 nchar_col = TRANSLATE(char_col USING NCHAR_CS);
UPDATE t1 SET
char_col = TRANSLATE(nchar_col USING CHAR_CS);
SELECT * FROM t1;`

CHAR_COL	NCHAR_COL
-----	-----
Hi	Hi

Example `UPDATE t1 SET
3 nchar_col = TRANSLATE('deo' USING NCHAR_CS);
UPDATE t1 SET
char_col = TRANSLATE(N'deo' USING CHAR_CS);`

CHAR_COL	NCHAR_COL
-----	-----
deo	deo

Other Single-Row Functions

DUMP

Syntax `DUMP(expr[,return_format[,start_position[,length]]])`

Purpose Returns a VARCHAR2 value containing the datatype code, length in bytes, and internal representation of *expr*. The returned result is always in the database character set. For the datatype corresponding to each code, see [Table 2-1](#).

The argument *return_format* specifies the format of the return value and can have any of the values listed below.

By default, the return value contains no character set information. To retrieve the character set name of *expr*, specify any of the format values below, plus 1000. For example, a *return_format* of 1008 returns the result in octal, plus provides the character set name of *expr*.

8 returns result in octal notation.

10 returns result in decimal notation.

16 returns result in hexadecimal notation.

17 returns result as single characters.

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, this function returns 'NULL'.

```
Example 1 SELECT DUMP('abc', 1016)
          FROM DUAL;

          DUMP('ABC',1016)
          -----
          Typ=96 Len=3 CharacterSet=WE8DEC: 61,62,63
```

```
Example 2 SELECT DUMP(ename, 8, 3, 2) "OCTAL"
          FROM emp
          WHERE ename = 'SCOTT';
```

```
OCTAL
-----
Type=1 Len=5: 117,124
```

Example 3 `SELECT DUMP(ename, 10, 3, 2) "ASCII "`
`FROM emp`
`WHERE ename = 'SCOTT';`

```
ASCII
-----
Type=1 Len=5: 79,84
```

EMPTY_[B | C]LOB

Syntax `EMPTY_[B|C]LOB()`

Purpose Returns 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.

You cannot use the locator returned from this function as a parameter to the DBMS_LOB package or the OCI.

Examples `INSERT INTO lob_tab1 VALUES (EMPTY_BLOB());`
`UPDATE lob_tab1`
`SET clob_col = EMPTY_BLOB();`

BFILENAME

Syntax `BFILENAME ('directory', 'filename')`

Purpose Returns a BFILE locator that is associated with a physical LOB binary file on the server's file system. A directory is an alias for a full pathname on the server's file system where the files are actually located; 'filename' is the name of the file in the server's file system.

Neither 'directory' nor 'filename' need to point to an existing object on the file system at the time you specify BFILENAME. However, you must associate a BFILE value with a physical file before performing subsequent SQL, PL/SQL, DBMS_LOB package, or OCI operations. For more information, see [CREATE DIRECTORY](#).

Note: This function does not verify that either the directory or file specified actually exists. Therefore, you can call the CREATE DIRECTORY command after BFILENAME. However, the object must exist by the time you actually use the BFILE locator (for example, as a parameter to one of the OCILob or DBMS_LOB operations such as OCILobFileOpen() or DBMS_LOB.FILEOPEN()).

For more information about LOBs, see [Oracle8 Application Developer's Guide](#) and [Oracle Call Interface Programmer's Guide](#).

Example

```
INSERT INTO file_tbl
VALUES (BFILENAME ('lob_dir1', 'image1.gif'));
```

GREATEST

Syntax `GREATEST(expr [,expr] ...)`

Purpose Returns the greatest of the list of *exprs*. All *exprs* after the first are implicitly converted to the datatype of the first *exprs* before the comparison. Oracle compares the *exprs* using nonpadded comparison semantics. Character comparison is based on the value of the character in the database character set. One character is greater than another if it has a higher value. If the value returned by this function is character data, its datatype is always VARCHAR2.

Example

```
SELECT GREATEST ('HARRY', 'HARRIOT', 'HAROLD')
"Great" FROM DUAL;
```

```
Great
-----
HARRY
```

LEAST

Syntax NLS_CHARSET_ID(text)

Purpose Returns the NLS character set ID number corresponding to NLS character set name, *text*. The *text* argument is a run-time VARCHAR2 value. The *text* value 'CHAR_CS' returns the server's database character set ID number. The *text* value 'NCHAR_CS' returns the server's national character set ID number.

Invalid character set names return null.

For a list of character set names, see [Oracle8 Reference](#).

Example 1 SELECT NLS_CHARSET_ID('ja16euc')
FROM DUAL;

```
NLS_CHARSET_ID('JA16EUC')
-----
                        830
```

Example 2 SELECT NLS_CHARSET_ID('char_cs')
FROM DUAL;

```
NLS_CHARSET_ID('CHAR_CS')
-----
                        2
```

Example 3 SELECT NLS_CHARSET_ID('nchar_cs')
FROM DUAL;

```
NLS_CHARSET_ID('NCHAR_CS')
-----
                        2
```

NLS_CHARSET_NAME

Syntax NLS_CHARSET_NAME (n)

Purpose Returns the name of the NLS character set corresponding to ID number *n*. The character set name is returned as a VARCHAR2 value in the database character set.

If *n* is not recognized as a valid character set ID, this function returns null.

For a list of character set IDs, see [Oracle8 Reference](#).

Example

```
SELECT NLS_CHARSET_NAME ( 2 )
FROM DUAL;
```

```
NLS_CH
-----
WE8DEC
```

NVL

Syntax NVL (expr1 , expr2)

Purpose If *expr1* is null, returns *expr2*; if *expr1* is not null, returns *expr1*. The arguments *expr1* and *expr2* can have any datatype. If their datatypes are different, Oracle converts *expr2* to the datatype of *expr1* before comparing them. The datatype of the return value is always the same as the datatype of *expr1*, unless *expr1* is character data, in which case the return value's datatype is VARCHAR2.

Example

```
SELECT ename, NVL ( TO_CHAR ( COMM ) , 'NOT
APPLICABLE' )
"COMMISSION" FROM emp
WHERE deptno = 30;
```

```
ENAME          COMMISSION
-----
ALLEN          300
WARD           500
MARTIN        1400
BLAKE         NOT APPLICABLE
TURNER         0
JAMES         NOT APPLICABLE
```

UID

Syntax UID

Purpose Returns an integer that uniquely identifies the current user.

USER

Syntax USER

Purpose Returns the current Oracle user with the datatype VARCHAR2. Oracle compares values of this function with blank-padded comparison semantics.

In a distributed SQL statement, the UID and USER functions identify the user on your local database. You cannot use these functions in the condition of a CHECK constraint.

Example `SELECT USER, UID FROM DUAL;`

USER	UID
SCOTT	19

USERENV

Syntax USERENV(*option*)

Purpose Returns information of VARCHAR2 datatype 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. The argument *option* can have any of these values:

' ISDBA '	returns 'TRUE' if you currently have the ISDBA role enabled and 'FALSE' if you do not.
' LANGUAGE '	returns the language and territory currently used by your session along with the database character set in this form: language_territory.characterset
' TERMINAL '	returns the operating system identifier for your current session's terminal. In distributed SQL statements, this option returns the identifier for your local session. In a distributed environment, this is supported only for remote SELECTs, not for remote INSERTs, UPDATEs, or DELETEs.
' SESSIONID '	returns your auditing session identifier. You cannot use this option in distributed SQL statements. To use this keyword in USERENV, the initialization parameter AUDIT_TRAIL must be set to TRUE.
' ENTRYID '	returns available auditing entry identifier. You cannot use this option in distributed SQL statements. To use this keyword in USERENV, the initialization parameter AUDIT_TRAIL must be set to TRUE.
' LANG '	Returns the ISO abbreviation for the language name, a shorter form than the existing 'LANGUAGE' parameter.
' INSTANCE '	Returns the instance identification number of the current instance.

Example SELECT USERENV('LANGUAGE') "Language" FROM DUAL;

```
Language
-----
AMERICAN_AMERICA.WE8DEC
```

VSIZE

Syntax `VSIZE (expr)`

Purpose Returns the number of bytes in the internal representation of *expr*. If *expr* is null, this function returns null.

Example

```
SELECT ename, VSIZE (ename) "BYTES"
FROM emp
WHERE deptno = 10;
```

ENAME	BYTES
-----	-----
CLARK	5
KING	4
MILLER	6

Object Reference Functions

Object reference functions manipulate REFs—references to objects of specified object types. For more information about REFs, see [Oracle8 Concepts](#) and [Oracle8 Application Developer's Guide](#).

DEREF

Syntax `DEREF (e)`

Purpose Returns the object reference of argument *e*. Argument *e* must be an expression that returns a REF to an object.

Example

```
CREATE TABLE tb1(c1 NUMBER, c2 REF t1);
SELECT Deref(c2) FROM tb1;
```

REFTOHEX

Syntax `REFTOHEX(r)`

Purpose Converts argument *r* to a character value containing its hexadecimal equivalent.

Example

```
CREATE TABLE tb1(c1 NUMBER, c2 REF t1);
SELECT REFTOHEX(c2) FROM tb1;
```

MAKE_REF

Syntax `MAKE_REF(table, key [,key...])`

Purpose Creates a REF to a row of an object view using *key* as the primary key. For more information about object views, see [Oracle8 Application Developer's Guide](#).

Example `CREATE TYPE t1 AS OBJECT(a NUMBER, b NUMBER);`
`CREATE TABLE tbl`
 `(c1 NUMBER, c2 NUMBER, PRIMARY KEY(c1, c2));`
`CREATE VIEW v1 OF t1 WITH OBJECT OID(a, b) AS`
 `SELECT * FROM tbl;`
`SELECT MAKE_REF(v1, 1, 3) FROM DUAL;`

Group Functions

Group functions return results based on groups of rows, rather than on single rows. In this way, group functions are different from single-row functions. For a discussion of the differences between group functions and single-row functions, see "[SQL Functions](#)".

Many group functions accept these options:

DISTINCT This option causes a group function to consider only distinct values of the argument expression.
ALL This option causes a group 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 neither option is specified, the default is **ALL**.

All group functions except **COUNT(*)** ignore nulls. You can use the **NVL** in the argument to a group function to substitute a value for a null.

If a query with a group function returns no rows or only rows with nulls for the argument to the group function, the group function returns null.

AVG

Syntax `AVG([DISTINCT|ALL] n)`

Purpose Returns average value of *n*.

Example `SELECT AVG(sal) "Average"`
 `FROM emp;`

```

                                  Average
                                  -----
                                  2077.21429

```

COUNT

Syntax `COUNT({* | [DISTINCT|ALL] expr})`

Purpose Returns the number of rows in the query.

 If you specify *expr*, this function returns rows where *expr* is not null. You can count either all rows, or only distinct values of *expr*.

 If you specify the asterisk (*), this function returns all rows, including duplicates and nulls.

Example `SELECT COUNT(*) "Total"`
1 `FROM emp;`

```

                                  Total
                                  -----
                                  18

```

Example `SELECT COUNT(job) "Count"`
2 `FROM emp;`

```

                                  Count
                                  -----
                                  14

```

Example `SELECT COUNT(DISTINCT job) "Jobs"`
3 `FROM emp;`

```

                                  Jobs
                                  -----
                                  5

```

MAX

Syntax `MAX([DISTINCT|ALL] expr)`

Purpose Returns maximum value of *expr*.

Example `SELECT MAX(sal) "Maximum" FROM emp;`

```

      Maximum
      -
      5000

```

MIN

Syntax `MIN([DISTINCT|ALL] expr)`

Purpose Returns minimum value of *expr*.

Example `SELECT MIN(hiredate) "Earliest" FROM emp;`

```

      Earliest
      -
      17-DEC-80

```

STDDEV

Syntax `STDDEV([DISTINCT|ALL] x)`

Purpose Returns standard deviation of *x*, a number. Oracle calculates the standard deviation as the square root of the variance defined for the VARIANCE group function.

Example `SELECT STDDEV(sal) "Deviation"
 FROM emp;`

```

      Deviation
      -
      1182.50322

```

SUM

Syntax `SUM([DISTINCT|ALL] n)`

Purpose Returns sum of values of n .

Example

```
SELECT SUM(sal) "Total"
FROM emp;

      Total
-----
      29081
```

VARIANCE

Syntax `VARIANCE([DISTINCT|ALL]x)`

Purpose Returns variance of x , a number. Oracle calculates the variance of x using this formula:

$$\frac{\sum_{j=1}^n x_j^2 - \frac{1}{n} \left[\sum_{j=1}^n x_j \right]^2}{n-1}$$

where:

x_i is one of the elements of x .

n is the number of elements in the set x . If n is 1, the variance is defined to be 0.

Example

```
SELECT VARIANCE(sal) "Variance"
FROM emp;

      Variance
-----
      1389313.87
```

User Functions

You can write your own user functions in PL/SQL to provide functionality that is not available in SQL or SQL functions. User functions are used in a SQL statement anywhere SQL functions can be used; that is, wherever expression can occur.

For example, user functions can be used in the following:

- the select list of a SELECT command
- the condition of a WHERE clause
- CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses
- the VALUES clause of an INSERT command
- the SET clause of an UPDATE command

For a complete description on the creation and use of user functions, see [Oracle8 Application Developer's Guide](#).

Prerequisites

User functions must be created as top-level PL/SQL functions or declared with a package specification before they can be named within a SQL statement. Create user functions as top-level PL/SQL functions by using the CREATE FUNCTION statement described in [CREATE FUNCTION](#). Specify packaged functions with a package with the CREATE PACKAGE statement described in [CREATE PACKAGE](#).

To call a packaged user function, you must declare the RESTRICT_REFERENCES pragma in the package specification.

Privileges Required

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 SELECT privileges on the view. No separate EXECUTE privileges are needed to select from the view.

Restrictions on User Functions

User functions cannot be used in situations that require an unchanging definition. Thus, a user function cannot:

- be used in a CHECK constraint clause of a CREATE TABLE or ALTER TABLE command
- be used in a DEFAULT clause of a CREATE TABLE or ALTER TABLE command
- contain OUT or IN OUT parameters
- update the database
- read or write package state if the function is a remote function
- use the *parallelism_clause* in SQL commands in the function if the function alters package state
- update variables defined in the function unless the function is a local function and is used in a SELECT list, VALUES clause of an INSERT command, or SET clause of an UPDATE command

Name Precedence

With PL/SQL, the names of database columns take precedence over the names of functions with no parameters. For example, if user SCOTT creates the following two objects in his own schema:

```
CREATE TABLE emp(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 EMP.NEW_SAL:

```
SELECT new_sal FROM emp;  
SELECT emp.new_sal FROM emp;
```

To access the function NEW_SAL, you would enter:

```
SELECT scott.new_sal FROM emp;
```

Here are some sample calls to user functions that are allowed in SQL expressions.

```
circle_area (radius)
payroll.tax_rate (empno)
scott.payroll.tax_rate (dependent, empno)@ny
```

Example

For example, to call the TAX_RATE user function from schema SCOTT, execute it against the SS_NO and SAL columns in TAX_TABLE, and place the results in the variable INCOME_TAX, specify the following:

```
SELECT scott.tax_rate (ss_no, sal)
       INTO income_tax
       FROM tax_table
       WHERE ss_no = tax_id;
```

Naming Conventions

If only one of the optional schema or package names is given, 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 proceeds as follows:

- Check for the PAYROLL package in the current schema.
- If a PAYROLL package is not found, look for a schema name PAYROLL that contains a top-level TAX_RATE function. If no such function is found, return an error message.
- If the PAYROLL package is found in the current schema, look for a TAX_RATE function in the PAYROLL package. If no such function is found, return an error message.

You can also refer to a stored top-level function using any synonym that you have defined for it.

Format Models

A *format model* is a character literal that describes the format of DATE or NUMBER data stored in a character string. You can use a format model as an argument of the TO_CHAR or TO_DATE function:

- to specify the format for Oracle to use to return a value from the database to you
- to specify the format for a value you have specified for Oracle to store in the database

Note that a format model does not change the internal representation of the value in the database.

This section describes how to use:

- number format models
- date format models
- format model modifiers

Changing the Return Format

You can use a format model to specify the format for Oracle to use to return values from the database to you.

Example 1

The following statement selects the commission values of the employees in Department 30 and uses the TO_CHAR function to convert these commissions into character values with the format specified by the number format model '\$9,990.99':

```
SELECT ename employee, TO_CHAR(comm, '$9,990.99') commission
FROM emp
WHERE deptno = 30;
```

EMPLOYEE	COMMISSION
ALLEN	\$300.00
WARD	\$500.00
MARTIN	\$1,400.00
BLAKE	
TURNER	\$0.00
JAMES	

Because of this format model, Oracle returns commissions with leading dollar signs, commas every three digits, and two decimal places. Note that TO_CHAR returns null for all employees with null in the COMM column.

Example 2

The following statement selects the date on which each employee from department 20 was hired and uses the TO_CHAR function to convert these dates to character strings with the format specified by the date format model 'fmMonth DD, YYYY':

```
SELECT ename, TO_CHAR(Hiredate, 'fmMonth DD, YYYY') hiredate
FROM emp
WHERE deptno = 20;
```

ENAME	HIREDATE
SMITH	December 17, 1980
JONES	April 2, 1981
SCOTT	April 19, 1987
ADAMS	May 23, 1987
FORD	December 3, 1981
LEWIS	October 23, 1997

With this format model, Oracle returns the hire dates with the month spelled out (as specified by "fm" and discussed in ["Format Model Modifiers"](#)), two digits for the day, and the century included in the year.

Supplying the Correct Format

You can use format models to specify the format of a value that you are converting from one datatype to another datatype required for a column. When you insert or update a column value, the datatype of the value that you specify must correspond to the column's datatype. For example, a value that you insert into a DATE column must be a value of the DATE datatype or a character string in the default date format (Oracle implicitly converts character strings in the default date format to the DATE datatype). If the value is in another format, you must use the TO_DATE function to convert the value to the DATE datatype. You must also use a format model to specify the format of the character string.

Example

The following statement updates BAKER's hire date using the TO_DATE function with the format mask 'YYYY MM DD' to convert the character string '1992 05 20' to a DATE value:

```
UPDATE emp
SET hiredate = TO_DATE('1992 05 20', 'YYYY MM DD')
WHERE ename = 'BLAKE';
```

Number Format Models

You can use number format models

- in the TO_CHAR function to translate a value of NUMBER datatype to VARCHAR2 datatype
- in the TO_NUMBER function to translate a value of CHAR or VARCHAR2 datatype to NUMBER datatype

All number format models cause the number to be rounded to the specified number of significant digits. If a value has more significant digits to the left of the decimal place than are specified in the format, pound signs (#) replace the value. If a positive value is extremely large and cannot be represented in the specified format, then the infinity sign (~) replaces the value. Likewise, if a negative value is extremely small and cannot be represented by the specified format, then the negative infinity sign replaces the value (-~).

Number Format Elements

A number format model is composed of one or more number format elements. [Table 3-12](#) lists the elements of a number format model. Examples are shown in [Table 3-13](#).

- If a number format model does not contain the MI, S, or PR format elements, negative return values automatically contain a leading negative sign and positive values automatically contain a leading space.
- A number format model can contain only a single decimal character (D) or period (.), but it can contain multiple group separators (G) or commas (,).
- A number format model must not begin with a comma (,).
- A group separator or comma cannot appear to the right of a decimal character or period in a number format model.

Table 3-12 Number Format Elements

Element	Example	Description
9	9999	Return value with the specified number of digits with a leading space if positive. Return value with the specified number of digits with a leading minus if negative. Leading zeros are blank, except for a zero value, which returns a zero for the integer part of the fixed-point number.
0	0999 9990	Return leading zeros. Return trailing zeros.
\$	\$9999	Return value with a leading dollar sign.
B	B9999	Return blanks for the integer part of a fixed-point number when the integer part is zero (regardless of "0's in the format model).
MI	9999MI	Return negative value with a trailing minus sign "-". Return positive value with a trailing blank.
S	S9999 9999S	Return negative value with a leading minus sign "-". Return positive value with a leading plus sign "+". Return negative value with a trailing minus sign "-". Return positive value with a trailing plus sign "+".
PR	9999PR	Return negative value in <angle brackets>. Return positive value with a leading and trailing blank.
D	99D99	Return a decimal character (that is, a period ".") in the specified position.
G	9G999	Return a group separator in the position specified.
C	C999	Return the ISO currency symbol in the specified position.
L	L999	Return the local currency symbol in the specified position.
, (comma)	9,999	Return a comma in the specified position.
. (period)	99.99	Return a decimal point (that is, a period ".") in the specified position.
V	999V99	Return a value multiplied by 10 ⁿ (and if necessary, round it up), where <i>n</i> is the number of 9's after the "V".
EEEE	9.9EEEE	Return a value using scientific notation.
RN	RN	Return a value as Roman numerals in uppercase.
rn		Return a value as Roman numerals in lowercase. Value can be an integer between 1 and 3999.
FM	FM90.9	Return a value with no leading or trailing blanks.

Example

[Table 3-13](#) shows the results of the following query for different values of *number* and *'fmt'*:

```
SELECT TO_CHAR(number, 'fmt')  
FROM DUAL
```

Table 3-13 Results of Example Number Conversions

number	'fmt'	Result
-1234567890	9999999999S	'1234567890-'
0	99.99	' .00 '
+0.1	99.99	' 0.10 '
-0.2	99.99	' -.20 '
0	90.99	' 0.00 '
+0.1	90.99	' 0.10 '
-0.2	90.99	' -0.20 '
0	9999	' 0 '
1	9999	' 1 '
0	B9999	' '
1	B9999	' 1 '
0	B90.99	' '
+123.456	999.999	' 123.456 '
-123.456	999.999	' -123.456 '
+123.456	FM999.009	'123.456 '
+123.456	9.9E999	' 1.2E+02 '
+1E+123	9.9E999	' 1.0E+123 '
+123.456	FM9.9E999	'1.23E+02 '
+123.45	FM999.009	'123.45 '
+123.0	FM999.009	'123.00 '
+123.45	L999.99	' \$123.45 '
+123.45	FML99.99	'\$123.45 '
+1234567890	9999999999S	'1234567890+ '

The MI and PR format elements can appear only in the last position of a number format model. The S format element can appear only in the first or last position of a number format model.

The characters returned by some of these format elements are specified by initialization parameters. [Table 3-14](#) lists these elements and parameters.

Table 3-14 Number Format Element Values Determined by Initialization Parameters

Element	Description	Initialization Parameter
D	Decimal character	NLS_NUMERIC_CHARACTER
G	Group separator	NLS_NUMERIC_CHARACTER
C	ISO currency symbol	NLS_ISO_CURRENCY
L	Local currency symbol	NLS_CURRENCY

You can specify the characters returned by these format elements implicitly using the initialization parameter NLS_TERRITORY. For information on these parameters, see [Oracle8 Reference](#).

You can change the characters returned by these format elements for your session with the ALTER SESSION command. You can also change the default date format for your session with the ALTER SESSION command. For information, see [ALTER SESSION](#).

Date Format Models

You can use date format models

- in the TO_CHAR function to translate a DATE value that is in a format other than the default date format
- in the TO_DATE function to translate a character value that is in a format other than the default date format

Default Date Format

The default date format is specified either explicitly with the initialization parameter NLS_DATE_FORMAT or implicitly with the initialization parameter NLS_TERRITORY. For information on these parameters, see [Oracle8 Reference](#).

You can change the default date format for your session with the ALTER SESSION command. For information, see [ALTER SESSION](#).

Maximum Length

The total length of a date format model cannot exceed 22 characters.

Date Format Elements

A date format model is composed of one or more date format elements as listed in [Table 3-15](#). For input format models, format items cannot appear twice, and format items that represent similar information cannot be combined. For example, you cannot use 'SYYYY' and 'BC' in the same format string. Only some of the date format elements can be used in the TO_DATE function as noted in [Table 3-15](#).

Table 3-15 Date Format Elements

Element	Specify in TO_DATE?	Meaning
- / ' . ; : 'text'	Yes	Punctuation and quoted text is reproduced in the result.

AD A . D .	Yes	AD indicator with or without periods.
AM A . M .	Yes	Meridian indicator with or without periods.
BC B . C .	Yes	BC indicator with or without periods.
CC SCC	No	One greater than the first two digits of a four-digit year; "S" prefixes BC dates with "-". For example, '20' from '1900'.
D	Yes	Day of week (1-7).
DAY	Yes	Name of day, padded with blanks to length of 9 characters.
DD	Yes	Day of month (1-31).
DDD	Yes	Day of year (1-366).
DY	Yes	Abbreviated name of day.
E	No	Abbreviated era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).
EE	No	Full era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).
HH	Yes	Hour of day (1-12).
HH12	No	Hour of day (1-12).
HH24	Yes	Hour of day (0-23).
IW	No	Week of year (1-52 or 1-53) based on the ISO standard.
IYY IY I	No	Last 3, 2, or 1 digit(s) of ISO year.

IYYY	No	4-digit year based on the ISO standard.
J	Yes	Julian day; the number of days since January 1, 4712 BC. Number specified with 'J' must be integers.
MI	Yes	Minute (0-59).
MM	Yes	Month (01-12; JAN = 01)
MON	Yes	Abbreviated name of month.
MONTH	Yes	Name of month, padded with blanks to length of 9 characters.
PM P.M.	No	Meridian indicator with or without periods.
Q	No	Quarter of year (1, 2, 3, 4; JAN-MAR = 1)
RM	Yes	Roman numeral month (I-XII; JAN = I).
RR	Yes	Given a year with 2 digits, returns a year in the next century if the year is <50 and the last 2 digits of the current year are >=50; returns a year in the preceding century if the year is >=50 and the last 2 digits of the current year are <50.
RRRR	Yes	Round year. Accepts either 4-digit or 2-digit input. If 2-digit, provides the same return as RR. If you don't want this functionality, simply enter the 4-digit year.
SS	Yes	Second (0-59).
SSSS	Yes	Seconds past midnight (0-86399).
WW	No	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	No	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
Y, YYY	Yes	Year with comma in this position.

YEAR SYEAR	No	Year, spelled out; "S" prefixes BC dates with "-".
YYYY SYYYY	Yes	4-digit year; "S" prefixes BC dates with "-".
YYY YY Y	Yes	Last 3, 2, or 1 digit(s) of year.

Oracle returns an error if an alphanumeric character is found in the date string where punctuation character is found in the format string. For example:

```
TO_CHAR (TO_DATE('0297', 'MM/YY'), 'MM/YY')
```

returns an error.

Date Format Elements and National Language Support

The functionality of some date format elements depends on the country and language in which you are using Oracle. For example, these date format elements return spelled values:

- MONTH
- MON
- DAY
- DY
- BC or AD or B.C. or A.D.
- AM or PM or A.M or P.M.

The language in which these values are returned is specified either explicitly with the initialization parameter `NLS_DATE_LANGUAGE` or implicitly with the initialization parameter `NLS_LANGUAGE`. The values returned by the YEAR and SYEAR date format elements are always in English.

The date format element D returns the number of the day of the week (1-7). The day of the week that is numbered 1 is specified implicitly by the initialization parameter `NLS_TERRITORY`.

For information on these initialization parameters, see *Oracle8 Reference*.

ISO Standard Date Format Elements

Oracle calculates the values returned by the date format elements IYYY, IYY, IY, I, and IW according to the ISO standard. For information on the differences between these values and those returned by the date format elements YYYY, YYY, YY, Y, and WW, see the discussion of national language support in [Oracle8 Reference](#).

The RR Date Format Element

The RR date format element is similar to the YY date format element, but it provides additional flexibility for storing date values in other centuries. The RR date format element allows you to store 21st century dates in the 20th century by specifying only the last two digits of the year. It will also allow you to store 20th century dates in the 21st century in the same way if necessary.

If you use the TO_DATE function with the YY date format element, the date value returned is always in the current century. If you use the RR date format element instead, the century of the return value varies according to the specified two-digit year and the last two digits of the current year. [Table 3-16](#) summarizes the behavior of the RR date format element.

Table 3-16 The RR Date Element Format

		If the specified two-digit year is	
		0 - 49	50 - 99
If the last two digits of the current year are:	0-49	The return date is in the current century.	The return date is in the preceding century.
	50-99	The return date is in the next century.	The return date is in the current century.

The following example demonstrates the behavior of the RR date format element.

Example 1

Assume these queries are issued between 1950 and 1999:

```
SELECT TO_CHAR(TO_DATE('27-OCT-95', 'DD-MON-RR'), 'YYYY') "Year"
      FROM DUAL;
```

```
Year
----
1995
```

```
SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR'), 'YYYY') "Year";
      FROM DUAL;
```

```
Year
----
2017
```

Example 2

Assume these queries are issued between 2000 and 2049:

```
SELECT TO_CHAR(TO_DATE('27-OCT-95', 'DD-MON-RR'), 'YYYY') "Year";
      FROM DUAL;
```

```
Year
----
1995
```

```
SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR'), 'YYYY') "Year";
      FROM DUAL;
```

```
Year
----
2017
```

Note that the queries return the same values regardless of whether they are issued before or after the year 2000. The RR date format element allows you to write SQL statements that will return the same values after the turn of the century.

Date Format Element Suffixes

[Table 3-17](#) lists suffixes that can be added to date format elements:

Table 3-17 Date Format Element Suffixes

Suffix	Meaning	Example Element	Example Value
TH	Ordinal Number	DDTH	4TH
SP	Spelled Number	DDSP	FOUR
SPTH or THSP	Spelled, ordinal number	DDSPTH	FOURTH

When you add one of these suffixes to a date format element, the return value is always in English.

Note:

Date suffixes are valid only on output and cannot be used to insert a date into the database.

Capitalization of Date Format Elements

Capitalization in a spelled-out word, abbreviation, or Roman numeral follows capitalization in the corresponding format element. For example, the date format model 'DAY' produces capitalized words like 'MONDAY'; 'Day' produces 'Monday'; and 'day' produces 'monday'.

Punctuation and Character Literals in Date Format Models

You can also include these characters in a date format model:

- punctuation such as hyphens, slashes, commas, periods, and colons
- character literals, enclosed in double quotation marks

These characters appear in the return value in the same location as they appear in the format model.

Format Model Modifiers

You can use the FM and FX modifiers in format models for the TO_CHAR function to control blank padding and exact format checking.

A modifier can appear in a format model more than once. In such a case, each subsequent occurrence toggles the effects of the modifier. Its effects are enabled for the portion of the model following its first occurrence, and then disabled for the portion following its second, and then reenabled for the portion following its third, and so on.

FM

"Fill mode". This modifier suppresses blank padding in the return value of the TO_CHAR function:

- In a date format element of a TO_CHAR function, this modifier suppresses blanks in subsequent character elements (such as MONTH) and suppresses leading and trailing zeroes for subsequent number elements (such

as MI) in a date format model. Without FM, the result of a character element is always right padded with blanks to a fixed length, and leading zeroes are always returned for a number element. With FM, because there is no blank padding, the length of the return value may vary

- In a number format element of a TO_CHAR function, this modifier suppresses blanks added to the left of the number, so that the result is left-justified in the output buffer. Without FM, the result is always right-justified in the buffer, resulting in blank-padding to the left of the number.

FX

"Format exact". This modifier specifies exact matching for the character argument and date format model of a TO_DATE function:

- Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.
- The character argument cannot have extra blanks. Without FX, Oracle ignores extra blanks.
- Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without FX, numbers in the character argument can omit leading zeroes.

When FX is enabled, you can disable this check for leading zeroes by using the FM modifier as well.

If any portion of the character argument violates any of these conditions, Oracle returns an error message.

Example 1

The following statement uses a date format model to return a character expression:

```
SELECT TO_CHAR(SYSDATE, 'fmDDTH') || ' of ' || TO_CHAR
       (SYSDATE, 'Month') || ', ' || TO_CHAR(SYSDATE, 'YYYY') "Ides"
FROM DUAL;
```

```
Ides
-----
3RD of April, 1995
```

Note that the statement above also uses the FM modifier. If FM is omitted, the month is blank-padded to nine characters:

```
SELECT TO_CHAR(SYSDATE, 'DDTH') || ' of ' ||
       TO_CHAR(Month, 'YYYY') "Ides"
FROM DUAL;
```

```
Ides
-----
03RD of April      , 1995
```

Example 2

The following statement places a single quotation mark in the return value by using a date format model that includes two consecutive single quotation marks:

```
SELECT TO_CHAR(SYSDATE, 'fmDay') || ''''s Special') "Menu"
FROM DUAL;
```

```
Menu
-----
Tuesday's Special
```

Two consecutive single quotation marks can be used for the same purpose within a character literal in a format model.

Example 3

[Table 3-18](#) shows whether the following statement meets the matching conditions for different values of *char* and *'fmt'* using FX:

```
UPDATE table
  SET date_column = TO_DATE(char, 'fmt');
```

Table 3-18 Matching Character Data and Format Models with the FX Format Model Modifier

char	'fmt'	Match or Error?
'15/ JAN /1993 '	'DD-MON-YYYY '	Match
' 15! JAN % /1993 '	'DD-MON-YYYY '	Error
'15/JAN/1993 '	'FXDD-MON-YYYY '	Error
'15-JAN-1993 '	'FXDD-MON-YYYY '	Match
'1-JAN-1993 '	'FXDD-MON-YYYY '	Error
'01-JAN-1993 '	'FXDD-MON-YYYY '	Match
'1-JAN-1993 '	'FXFMDD-MON-YYYY '	Match

String-to-Date Conversion Rules

The following additional formatting rules apply when converting string values to date values:

- You can omit punctuation included in the format string from the date string if all the digits of the numerical format elements, including leading zeros, are specified. In other words, specify 02 and not 2 for two-digit format elements such as MM, DD, and YY.
- You can omit time fields found at the end of a format string from the date string.
- If a match fails between a date format element and the corresponding characters in the date string, Oracle attempts alternative format elements, as shown in [Table 3-19](#).

Table 3-19 Oracle Format Matching

Original Format Element	Additional Format Elements to Try in Place of the Original
'MM'	'MON' and 'MONTH'
'MON'	'MONTH'
'MONTH'	'MON'
'YY'	'YYYY'
'RR'	'RRRR'

Expressions

An *expression* is a combination of one or more values, operators, and SQL functions that evaluate to a value. An expression generally assumes the datatype of its components.

This simple expression evaluates to 4 and has datatype NUMBER (the same datatype as its components):

```
2*2
```

The following expression is an example of a more complex expression that uses both functions and operators. The expression adds seven days to the current date, removes the time component from the sum, and converts the result to CHAR datatype:

```
TO_CHAR ( TRUNC ( SYSDATE+7 ) )
```

You can use expressions in

- the select list of the SELECT command
- a condition of the WHERE and HAVING clauses
- the CONNECT BY, START WITH, and ORDER BY clauses
- the VALUES clause of the INSERT command
- the SET clause of the UPDATE command

For example, you could use an expression in place of the quoted string 'smith' in this UPDATE statement SET clause:

```
SET ename = 'smith';
```

This SET clause has the expression LOWER(ename) instead of the quoted string 'smith':

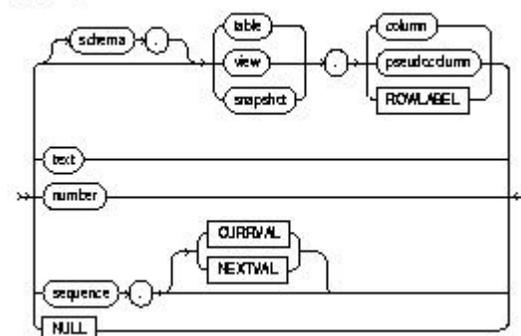
```
SET ename = LOWER(ename);
```

Expressions have several forms. Oracle does not accept all forms of expressions in all parts of all SQL commands. You must use appropriate expression notation whenever *expr* appears in conditions, SQL functions, or SQL commands in other parts of this reference. The description of each command in [Chapter 4, "Commands"](#), documents the restrictions on the expressions in the command. The sections that follow describe and provide examples of the various forms of expressions.

Form I

A Form I expression specifies column, pseudocolumn, constant, sequence number, or NULL.

`expr_form1 ::=`



In addition to the schema of a user, *schema* can also be "PUBLIC" (double quotation marks required), in which case it must qualify a public synonym for a table, view, or snapshot. Qualifying a public synonym with "PUBLIC" is supported only in data manipulation language (DML) commands, not data definition language (DDL) commands.

The *pseudocolumn* can be either LEVEL, ROWID, or ROWNUM. You can use a pseudocolumn only with a table, not with a view or snapshot. NCHAR and NVARCHAR2 are not valid pseudocolumn or ROWLABEL datatypes. For more information on pseudocolumns, see "[Pseudocolumns](#)".

If you are not using Trusted Oracle, the expression ROWLABEL always returns NULL. For information on using labels and ROWLABEL, see your Trusted Oracle documentation.

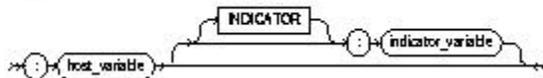
Some valid Form I expressions are:

```
emp.ename
'this is a text string'
10
N'this is an NCHAR string'
```

Form II

A Form II expression specifies a host variable with an optional indicator variable. Note that this form of expression can only appear in embedded SQL statements or SQL statements processed in an Oracle Call Interface (OCI) program.

`expr_form_II ::=`



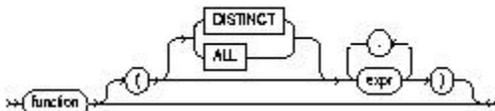
Some valid Form II expressions are:

```
:employee_name INDICATOR :employee_name_indicator_var
:department_location
```

Form III

A Form III expression specifies a call to a SQL function operating on a single row.

`expr_form_III ::=`



Some valid Form III expressions are:

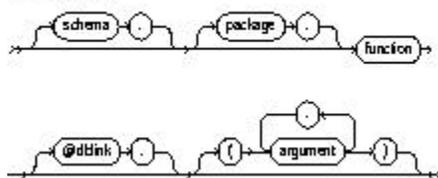
```
LENGTH('BLAKE')
ROUND(1234.567*43)
SYSDATE
```

For information on SQL functions, see "[SQL Functions](#)".

Form IV

A Form IV expression specifies a call to a user function

`expr_form_IV ::=`



Some valid Form IV expressions are:

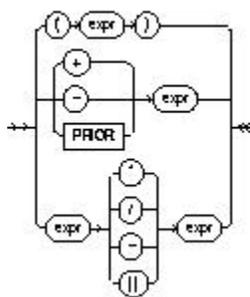
```
circle_area(radius)
payroll.tax_rate(empno)
scott.payrol.tax_rate(dependents, empno)@ny
```

For information on user functions, see ["User Functions"](#).

Form V

A Form V expression specifies a combination of other expressions.

`expr_form_V ::=`



Note that some combinations of functions are inappropriate and are rejected. For example, the LENGTH function is inappropriate within a group function.

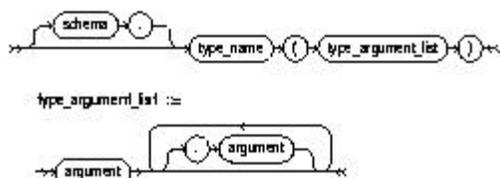
Some valid Form V expressions are:

```
('CLARK' || 'SMITH')
LENGTH('MOOSE') * 57
SQRT(144) + 72
my_fun(TO_CHAR(sysdate, 'DD-MMM-YY'))
```

Form VI

A Form VI expression specifies a call to a type constructor.

`expr_form_VI ::=`



If *type_name* is an object type, then the type argument list must be an ordered list of arguments, where the first argument is a value whose type matches the first attribute of the object type, the second argument is a value whose type matches the second attribute of the object type, and so on. The total number of arguments to the constructor must match the total number of attributes of the object type; the maximum number of arguments is 999.

If *type_name* is a VARRAY or nested table type, then the argument list can contain zero or more arguments. Zero arguments imply construction of an empty collection. Otherwise, each argument corresponds to an element value whose type is the element type of the collection type.

Whether *type_name* is an object type, a VARRAY, or a nested table type, the maximum number of arguments it can contain is 999.

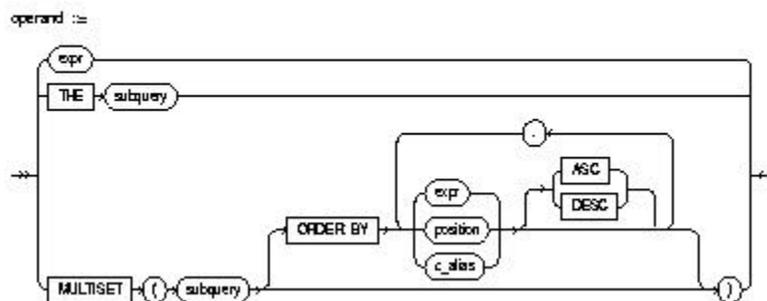
Example

```
CREATE TYPE address_t AS OBJECT
  (no NUMBER, street CHAR(31), city CHAR(21), state CHAR(3), zip NUMBER);
CREATE TYPE address_book_t AS TABLE OF address_t;
DECLARE
  /* Object Type variable initialized via Object Type Constructor */
  myaddr address_t = address_t(500, 'Oracle Parkway', 'Redwood Shores',
                              'CA', 94065);
  /* nested table variable initialized to an empty table via a
     constructor*/
  alladdr address_book_t = address_book_t();
BEGIN
  /* below is an example of a nested table constructor with two elements
     specified, where each element is specified as an object type
     constructor. */
  insert into employee values (666999, address_book_t(address_t(500,
    'Oracle Parkway', 'Redwood Shores', 'CA', 94065), address_t(400,
    'Mission Street', 'Fremont', 'CA', 94555)));
END;
```

Form VII

A Form VII expression converts one collection-typed value into another collection-typed value.

→ CAST ((operand) AS (type_name)) ←



CAST allows you to convert collection-typed values of one type into another collection type. You can cast an unnamed collection (such as the result set of a subquery) or a named collection (such as a VARRAY or a nested table) into a type-compatible named collection. The *type_name* must be the name of a collection type and the *operand* must evaluate to a collection value.

To cast a named collection type into another named collection type, the elements of both collections must be of the same type.

If the result set of *subquery* can evaluate to multiple rows, 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, which is not supported in the **CAST** expression. In other words, scalar subqueries as arguments of the **CAST** operator are not valid in Oracle8.

The **CAST** examples that follow use the following user-defined types and tables:

```
CREATE TYPE address_t AS OBJECT
  (no NUMBER, street CHAR(31), city CHAR(21), state CHAR(2));
CREATE TYPE address_book_t AS TABLE OF address_t;
CREATE TYPE address_array_t AS VARRAY(3) OF address_t;
CREATE TABLE emp_address (empno NUMBER, no NUMBER, street CHAR(31),
                           city CHAR(21), state CHAR(2));
CREATE TABLE employees (empno NUMBER, name CHAR(31));
CREATE TABLE dept (dno NUMBER, addresses address_array_t);
```

Example 1

CAST a subquery:

```
SELECT e.empno, e.name, CAST(MULTISET(SELECT ea.no, ea.street,
                                     ea.city, ea.state
                                     FROM emp_address ea
                                     WHERE ea.empno = e.empno)
                             AS address_book_t)
FROM employees e;
```

Example 2

CAST converts a **VARRAY** type column into a nested table. The table values are generated by a flattened subquery. See ["Using Flattened Subqueries"](#).

```
SELECT *
FROM THE(SELECT CAST(d.addresses AS address_book_t)
         FROM dept d
         WHERE d.dno = 111) a
WHERE a.city = 'Redwood Shores';
```

Example 3

The following example casts a **MULTISET** expression with an **ORDER BY** clause:

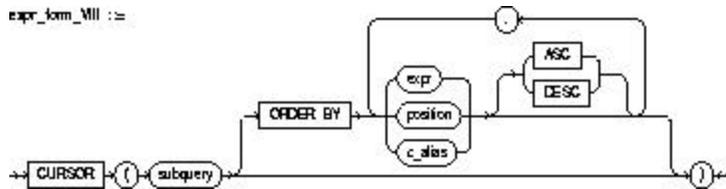
```
CREATE TABLE projects (empid NUMBER, projname VARCHAR2(10));
CREATE TABLE employees (empid NUMBER, ename VARCHAR2(10));
CREATE TYPE projname_table_type AS TABLE OF VARCHAR2(10);
```

An example of a **MULTISET** expression with the above schema is:

```
SELECT e.name, CAST(MULTISET(SELECT p.projname
                             FROM projects p
                             WHERE p.empid=e.empid
                             ORDER BY p.projname)
                    AS projname_table_type)
FROM employees e;
```

Form VIII

A Form VIII expression returns a nested **CURSOR**. This form of expression is similar to the PL/SQL REF cursor.



A nested cursor is implicitly opened when the containing row is fetched from the parent cursor. The nested cursor is closed only when

- explicitly closed by the user
- the parent cursor is reexecuted
- the parent cursor is closed
- the parent cursor is cancelled
- an error arises during fetch on one of its parent cursors (it is closed as part of the clean-up)

The following restrictions apply to the CURSOR expression:

- Nested cursors can appear only in a SELECT statement that is not nested in any other query expression, except when it is a subquery of the CURSOR expression itself.
- Nested cursors can appear only in the outermost SELECT list of the query specification.
- Nested cursors cannot appear in views.
- You cannot perform BIND and EXECUTE operations on nested cursors.

Example

```
SELECT d.deptno, CURSOR(SELECT e.empno, CURSOR(SELECT p.projnum,
                                             p.projname
                                             FROM   projects p
                                             WHERE  p.empno = e.empno)
                        FROM TABLE(d.employees) e)
FROM dept d
WHERE d.dno = 605;
```

Form IX

A Form IX expression constructs a reference to an object.



In a SQL statement, REF takes as its argument a 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. For more information about REFs, see [Oracle8 Concepts](#).

Example 1

```
SELECT REF(e)
FROM employee_t e
WHERE e.empno = 10000;
```

Example 2

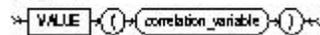
This example uses REF in a predicate:

```
SELECT e.name
FROM employee_t
  e INTO :x
WHERE REF(e) = empref1;
```

Form X

A Form X expression returns the row object.

`expr_form_X ::=`



In a SQL statement, VALUE takes as its argument a correlation variable (table alias) associated with a row of an object table.

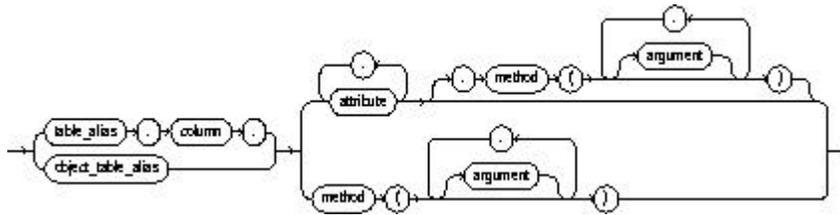
Example

```
SELECT VALUE(e)
FROM employee e
WHERE e.name = 'John Smith';
```

Form XI

A Form XI expression specifies attribute reference and method invocation.

`expr_form_XI ::=`



The `column` parameter can be an object or REF column. Examples in this section use the following user-defined types and tables:

```
CREATE OR REPLACE TYPE employee_t AS OBJECT
  (empid NUMBER,
   name CHAR(31),
   birthdate DATE,
   MEMBER FUNCTION age RETURN NUMBER,
   PRAGMA RESTRICT REFERENCES(age, RNPS, WNPS, WNDS)
  );
CREATE OR REPLACE TYPE BODY employee_t AS
  MEMBER FUNCTION age RETURN NUMBER IS
  var NUMBER;
  BEGIN
    var := months_between(ROUND(SYSDATE, 'YEAR'),
      ROUND(birthdate, 'YEAR'))/12;
    RETURN(var);
  END;
END; /
CREATE TABLE department (dno NUMBER, manager EMPLOYEE_T);
```

Examples

The following examples update and select from the object columns and method defined above.

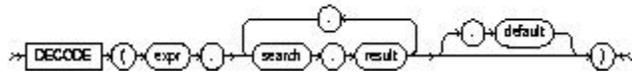
```
UPDATE department d
  SET d.manager.empid = 100;

SELECT d.manager.name, d.manager.age()
  FROM department d;
```

Decoded Expression

A decoded expression uses the special DECODE syntax:

`decode_expr ::=`



To evaluate this expression, Oracle compares *expr* to each *search* value one by one. If *expr* is equal to a *search*, Oracle returns the corresponding *result*. If no match is found, Oracle returns *default*, or, if *default* is omitted, returns null. If *expr* and *search* contain character data, Oracle compares them using nonpadded comparison semantics. For information on these semantics, see the section "[Datatype Comparison Rules](#)".

The *search*, *result*, and *default* values can be derived from expressions. Oracle 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 datatype of the first *search* value before comparing. Oracle automatically converts the return value to the same datatype as the first *result*. If the first *result* has the datatype CHAR or if the first *result* is null, then Oracle converts the return value to the datatype VARCHAR2. For information on datatype conversion, see "[Data Conversion](#)".

In a DECODE expression, Oracle considers two nulls to be equivalent. If *expr* is null, Oracle returns the *result* of the first *search* that is also null.

The maximum number of components in the DECODE expression, including *expr*, *searches*, *results*, and *default* is 255.

Example

This expression decodes the value DEPTNO. If DEPTNO is 10, the expression evaluates to 'ACCOUNTING'; if DEPTNO is 20, it evaluates to 'RESEARCH'; etc. If DEPTNO is not 10, 20, 30, or 40, the expression returns 'NONE'.

```
DECODE (deptno,10, 'ACCOUNTING',
        20, 'RESEARCH',
        30, 'SALES',
        40, 'OPERATION',
        'NONE')
```

List of Expressions

A list of expressions is a parenthesized series of expressions separated by a comma.

`expr_list ::=`



An expression list can contain up to 1000 expressions. Some valid expression lists are:

```
10, 20, 40)
('SCOTT', 'BLAKE', 'TAYLOR')
(LENGTH('MOOSE') * 57, -SQRT(144) + 72, 69)
```

Conditions

A condition specifies a combination of one or more expressions and logical operators that evaluates to either TRUE, FALSE, or unknown. You must use this syntax whenever *condition* appears in SQL commands in [Chapter 4, "Commands"](#).

You can use a condition in the WHERE clause of these statements:

- DELETE
- SELECT
- UPDATE

You can use a condition in any of these clauses of the SELECT command:

- WHERE
- START WITH
- CONNECT BY
- HAVING

A condition could be said to be of the "logical" datatype, although Oracle does not formally support such a datatype.

The following simple condition always evaluates to TRUE:

```
1 = 1
```

The following more complex condition adds the SAL value to the COMM value (substituting the value 0 for null) and determines whether the sum is greater than the number constant 2500:

```
NVL(sal, 0) + NVL(comm, 0) > 2500
```

Logical operators can combine multiple conditions into a single condition. For example, you can use the AND operator to combine two conditions:

```
(1 = 1) AND (5 < 7)
```

Here are some valid conditions:

```
name = 'SMITH'
emp.deptno = dept.deptno
hiredate > '01-JAN-88'
job IN ('PRESIDENT', 'CLERK', 'ANALYST')
sal BETWEEN 500 AND 1000
comm IS NULL AND sal = 2000
```

Conditions can have several forms. The description of each command in [Chapter 4, "Commands"](#), documents the restrictions on the conditions in the command. The sections that follow describe the various forms of conditions.

A Form IV condition tests for inclusion in a range.

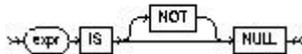
condition_form_IV ::=



Form V

A Form V condition tests for nulls.

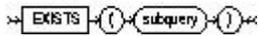
condition_form_V ::=



Form VI

A Form VI condition tests for existence of rows in a subquery.

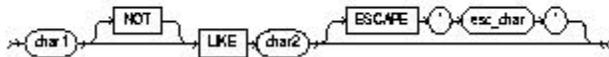
condition_form_VI ::=



Form VII

A Form VII condition specifies a test involving pattern matching.

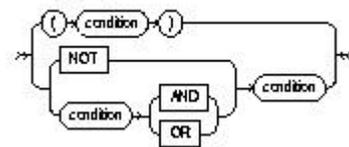
condition_form_VII ::=



Form VIII

A Form VIII condition specifies a combination of other conditions.

condition_form_VIII ::=



[Prev](#) [Next](#)

ORACLE

Copyright © 1997 Oracle Corporation.

All Rights Reserved.



[Library](#) [Product](#) [Contents](#) [Index](#)