2

Basic Elements of Oracle SQL

This chapter contains reference information on the basic elements of Oracle SQL. These elements are the simplest building blocks of SQL statements. Therefore, before using the SQL statements described in this book, you should familiarize yourself with the concepts covered in this chapter.

This chapter contains these sections:

Data Types

Each value manipulated by Oracle Database has a **data type**. The data type of a value associates a fixed set of properties with the value. These properties cause Oracle to treat values of one data type differently from values of another. For example, you can add values of `NUMBER` data type, but not values of `RAW` data type.

When you create a table or cluster, you must specify a data type for each of its columns. When you create a procedure or stored function, you must specify a data type for each of its arguments. These data types define the domain of values that each column can contain or each argument can have. For example, `DATE` columns cannot accept the value February 29 (except for a leap year) or the values 2 or 'SHOE'. Each value subsequently placed in a column assumes the data type of the column. For example, if you insert `'01-JAN-98'` into a `DATE` column, then Oracle treats the `'01-JAN-98'` character string as a `DATE` value after verifying that it translates to a valid date.

Oracle Database provides a number of built-in data types as well as several categories for user-defined types that can be used as data types. The syntax of Oracle data types appears in the diagrams that follow. The text of this section is divided into the following sections:

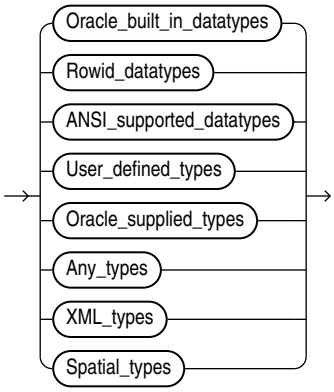
- XML Types
- Spatial Types

A data type is either scalar or nonscalar. A scalar type contains an atomic value, whereas a nonscalar (sometimes called a "collection") contains a set of values. A large object (LOB) is a special form of scalar data type representing a large scalar value of binary or character data. LOBs are subject to some restrictions that do not affect other scalar types because of their size. Those restrictions are documented in the context of the relevant SQL syntax.

> **See Also:**
>
> Restrictions on LOB Columns

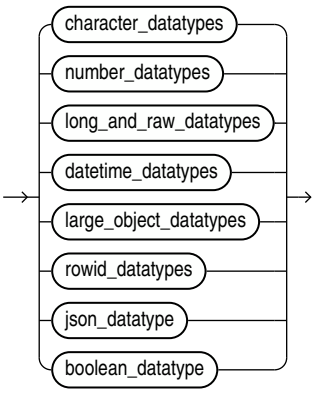
The Oracle precompilers recognize other data types in embedded SQL programs. These data types are called **external data types** and are associated with host variables. Do not confuse built-in data types and user-defined types with external data types. For information on external data types, including how Oracle converts between them and built-in data types or user-defined types, see *Pro*COBOL Developer's Guide*, and *Pro*C/C++ Developer's Guide*.

***datatype*::=**

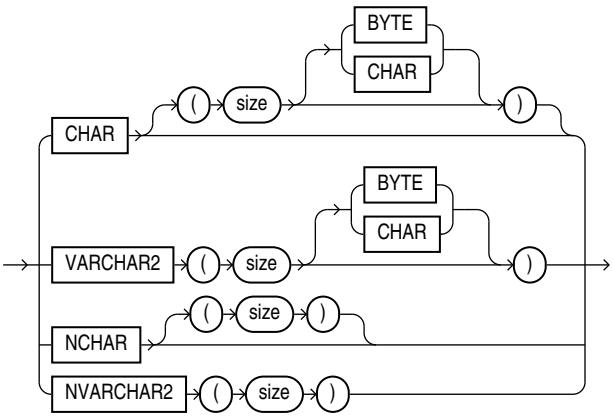


The Oracle built-in data types appear in the figures that follows. For descriptions, refer to Oracle Built-in Data Types.

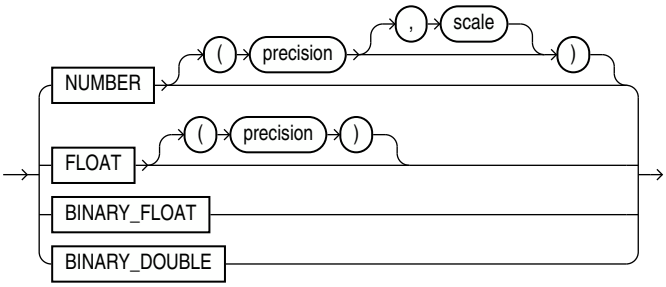
***Oracle_built_in_datatypes*::=**



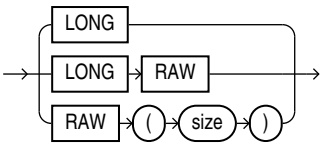
***character_datatypes*::=**



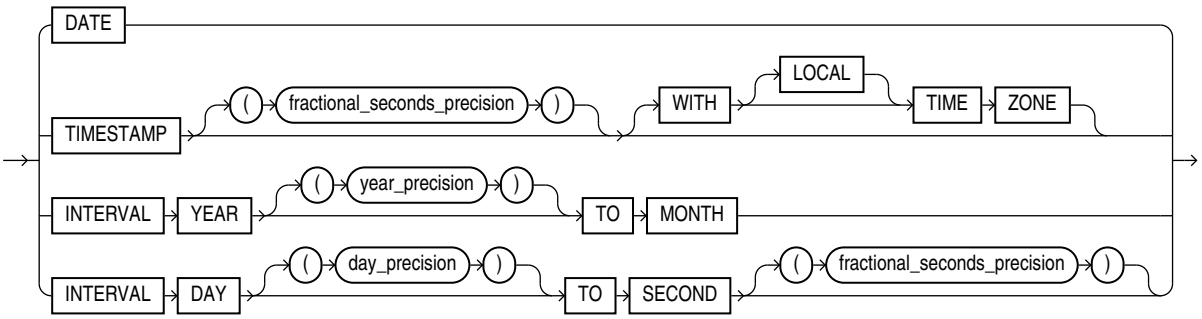
***number_datatypes*::=**



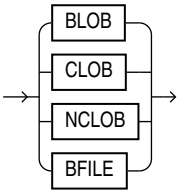
***long_and_raw_datatypes*::=**



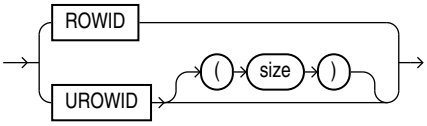
***datetime_datatypes*::=**



***large_object_datatypes*::=**

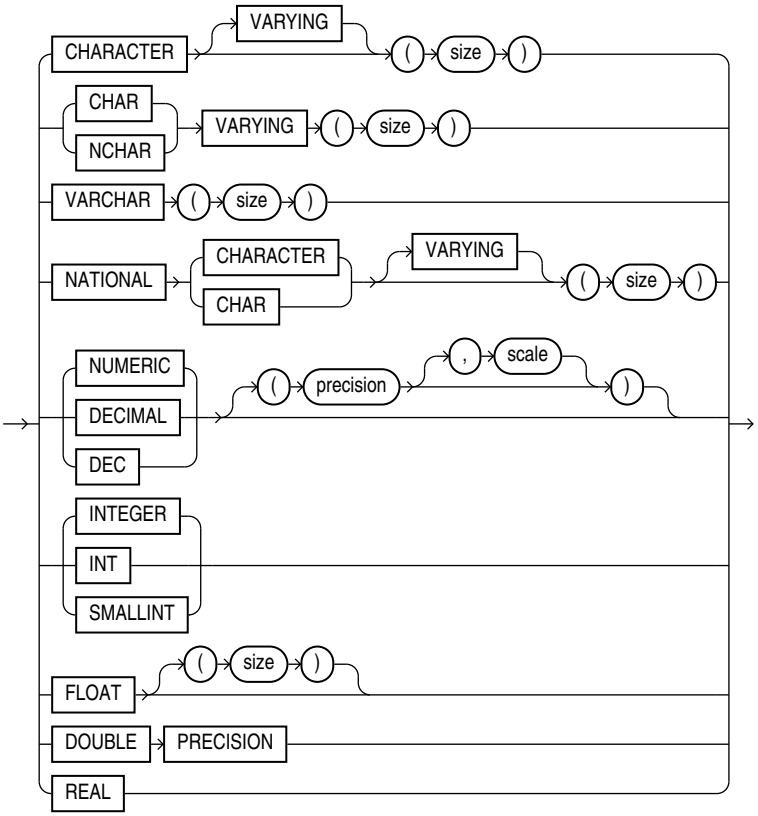


***rowid_datatypes*::=**



The ANSI-supported data types appear in the figure that follows. ANSI, DB2, and SQL/DS Data Types discusses the mapping of ANSI-supported data types to Oracle built-in data types.

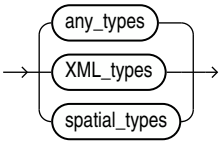
***ANSI_supported_datatypes*::=**



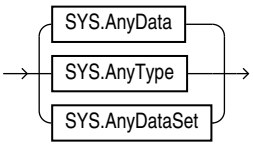
For descriptions of user-defined types, refer to User-Defined Types .

The Oracle-supplied data types appear in the figures that follows. For descriptions, refer to Oracle-Supplied Types .

***Oracle_supplied_types*::=**

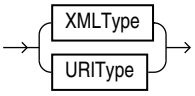


***any_types*::=**



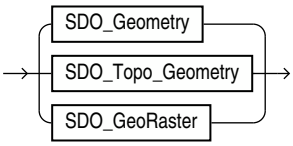
For descriptions of the `Any` types, refer to Any Types .

***XML_types*::=**



For descriptions of the XML types, refer to XML Types .

***spatial_types*::=**



For descriptions of the spatial types, refer to Spatial Types .

Oracle Built-in Data Types

The **Built-In Data Type Summary** table lists the built-in data types available. Oracle Database uses a code to identify the data type internally. This is the number in the **Code** column of the **Built-In Data Type Summary** table. You can verify the codes in the table using the `DUMP` function.

In addition to the built-in data types listed in the **Built-In Data Type Summary** table, Oracle Database uses many data types internally that are visible via the `DUMP` function.

Table 2-1 Built-In Data Type Summary

Code	Data Type	Description
1	VARCHAR2(*size* [BYTE | CHAR])	Variable-length character string having maximum length *size* bytes or characters. You must specify *size* for VARCHAR2. Minimum *size* is 1 byte or 1 character. Maximum size is: • 32767 bytes or characters if MAX_STRING_SIZE = EXTENDED • 4000 bytes or characters if MAX_STRING_SIZE = STANDARD Refer to Extended Data Types for more information on the MAX_STRING_SIZE initialization parameter. BYTE indicates that the column will have byte length semantics. CHAR indicates that the column will have character semantics.
1	NVARCHAR2(*size*)	Variable-length Unicode character string having maximum length *size* characters. You must specify *size* for NVARCHAR2. The number of bytes can be up to two times *size* for AL16UTF16 encoding and three times *size* for UTF8 encoding. Maximum *size* is determined by the national character set definition, with an upper limit of: • 32767 bytes if MAX_STRING_SIZE = EXTENDED • 4000 bytes if MAX_STRING_SIZE = STANDARD Refer to Extended Data Types for more information on the MAX_STRING_SIZE initialization parameter.
2	NUMBER [(*p* [, *s*])]	Number having precision *p* and scale *s*. The precision *p* can range from 1 to 38. The scale *s* can range from -84 to 127. Both precision and scale are in decimal digits. A NUMBER value requires from 1 to 22 bytes.
2	FLOAT [(*p*)]	A subtype of the NUMBER data type having precision *p*. A FLOAT value is represented internally as NUMBER. The precision *p* can range from 1 to 126 binary digits. A FLOAT value requires from 1 to 22 bytes.
8	LONG	Character data of variable length up to 2 gigabytes, or 2^{31} -1 bytes. Provided for backward compatibility.
12	DATE	Valid date range from January 1, 4712 BC, to December 31, 9999 AD. The default format is determined explicitly by the NLS_DATE_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. The size is fixed at 7 bytes. This data type contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. It does not have fractional seconds or a time zone.
100	BINARY_FLOAT	32-bit floating point number. This data type requires 4 bytes.
101	BINARY_DOUBLE	64-bit floating point number. This data type requires 8 bytes.
180	TIMESTAMP [(*fractional_seconds_precision*)]	Year, month, and day values of date, as well as hour, minute, and second values of time, where *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values of *fractional_seconds_precision* are 0 to 9. The default is 6. The default format is determined explicitly by the NLS_TIMESTAMP_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. The size is 7 or 11 bytes, depending on the precision. This data type contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. It contains fractional seconds but does not have a time zone.

Table 2-1 (Cont.) Built-In Data Type Summary

Code	Data Type	Description
181	TIMESTAMP [(*fractional_seconds_precision*)] WITH TIME ZONE	All values of TIMESTAMP as well as time zone displacement value, where *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6. The default date format for the TIMESTAMP WITH TIME ZONE data type is determined by the NLS_TIMESTAMP_TZ_FORMAT initialization parameter. The size is fixed at 13 bytes. This data type contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, and TIMEZONE_MINUTE. It has fractional seconds and an explicit time zone.
231	TIMESTAMP [(*fractional_seconds_precision*)] WITH LOCAL TIME ZONE	All values of TIMESTAMP WITH TIME ZONE, with the following exceptions: • Data is normalized to the database time zone when it is stored in the database. • When the data is retrieved, users see the data in the session time zone. The default format is determined explicitly by the NLS_TIMESTAMP_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. The size is 7 or 11 bytes, depending on the precision.
182	INTERVAL YEAR [(*year_precision*)] TO MONTH	Stores a period of time in years and months, where *year_precision* is the number of digits in the YEAR datetime field. Accepted values are 0 to 9. The default is 2. The size is fixed at 5 bytes.
183	INTERVAL DAY [(*day_precision*)] TO SECOND [(*fractional_seconds_precision*)]	Stores a period of time in days, hours, minutes, and seconds, where • *day_precision* is the maximum number of digits in the DAY datetime field. Accepted values are 0 to 9. The default is 2. • *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND field. Accepted values are 0 to 9. The default is 6. The size is fixed at 11 bytes.
23	RAW(*size*)	Raw binary data of length *size* bytes. You must specify *size* for a RAW value. Maximum *size* is: • 32767 bytes if MAX_STRING_SIZE = EXTENDED • 2000 bytes if MAX_STRING_SIZE = STANDARD Refer to Extended Data Types for more information on the MAX_STRING_SIZE initialization parameter.
24	LONG RAW	Raw binary data of variable length up to 2 gigabytes.
69	ROWID	Base 64 string representing the unique address of a row in its table. This data type is primarily for values returned by the ROWID pseudocolumn.
208	UROWID [(*size*)]	Base 64 string representing the logical address of a row of an index-organized table. The optional *size* is the size of a column of type UROWID. The maximum size and default is 4000 bytes.
96	CHAR [(*size* [BYTE | CHAR])]	Fixed-length character data of length *size* bytes or characters. Maximum *size* is 2000 bytes or characters. Default and minimum *size* is 1 byte. BYTE and CHAR have the same semantics as for VARCHAR2.

Table 2-1 (Cont.) Built-In Data Type Summary

Code	Data Type	Description
96	NCHAR[(*size*)]	Fixed-length character data of length *size* characters. The number of bytes can be up to two times *size* for AL16UTF16 encoding and three times *size* for UTF8 encoding. Maximum *size* is determined by the national character set definition, with an upper limit of 2000 bytes. Default and minimum *size* is 1 character.
112	CLOB	A character large object containing single-byte or multibyte characters. Both fixed-width and variable-width character sets are supported, both using the database character set. Maximum size is (4 gigabytes - 1) * (database block size).
112	NCLOB	A character large object containing Unicode characters. Both fixed-width and variable-width character sets are supported, both using the database national character set. Maximum size is (4 gigabytes - 1) * (database block size). Stores national character set data.
113	BLOB	A binary large object. Maximum size is (4 gigabytes - 1) * (database block size).
114	BFILE	Contains a locator to a large binary file stored outside the database. Enables byte stream I/O access to external LOBs residing on the database server. Maximum size is 4 gigabytes.
119	JSON	Maximum size is 32 megabytes.
252	BOOLEAN	The BOOLEAN data type comprises the distinct truth values *True* and *False*. Unless prohibited by a NOT NULL constraint, the boolean data type also supports the truth value *UNKOWN* as the null value.
127	VECTOR	The VECTOR data type represents a vector as a series of numbers stored in one of the following formats: • INT8 (8-bit integers) • FLOAT32 (32-bit floating-point numbers) • FLOAT64 (64-bit floating-point numbers) • BINARY FLOAT32 and FLOAT64 are IEEE standards. Oracle Database automatically casts the values as needed.

The sections that follow describe the Oracle data types as they are stored in Oracle Database. For information on specifying these data types as literals, refer to Literals .

Character Data Types

Character data types store character (alphanumeric) data, which are words and free-form text, in the database character set or national character set. They are less restrictive than other data types and consequently have fewer properties. For example, character columns can store all alphanumeric values, but NUMBER columns can store only numeric values.

Character data is stored in strings with byte values corresponding to one of the character sets, such as 7-bit ASCII or EBCDIC, specified when the database was created. Oracle Database supports both single-byte and multibyte character sets.

These data types are used for character data:

- CHAR Data Type
- NCHAR Data Type

- VARCHAR2 Data Type
- NVARCHAR2 Data Type

For information on specifying character data types as literals, refer to Text Literals .

CHAR Data Type

The `CHAR` data type specifies a fixed-length character string in the database character set. You specify the database character set when you create your database.

When you create a table with a `CHAR` column, you specify the column length as *`size`* optionally followed by a length qualifier. The qualifier `BYTE` denotes byte length semantics while the qualifier `CHAR` denotes character length semantics. In the byte length semantics, *`size`* is the number of bytes to store in the column. In the character length semantics, *`size`* is the number of code points in the database character set to store in the column. A code point may have from 1 to 4 bytes depending on the database character set and the particular character encoded by the code point. Oracle recommends that you specify one of the length qualifiers to explicitly document the desired length semantics of the column. If you do not specify a qualifier, the value of the `NLS_LENGTH_SEMANTICS` parameter of the session creating the column defines the length semantics, unless the table belongs to the schema `SYS`, in which case the default semantics is `BYTE`.

Oracle ensures that all values stored in a `CHAR` column have the length specified by *`size`* in the selected length semantics. If you insert a value that is shorter than the column length, then Oracle blank-pads the value to column length. If you try to insert a value that is too long for the column, then Oracle returns an error. Note that if the column length is expressed in characters (code points), blank-padding does not guarantee that all column values have the same byte length.

You can omit *`size`* from the column definition. The default value is 1.

The maximum value of *`size`* is 2000, which means 2000 bytes or characters (code points), depending on the selected length semantics. However, independently, the absolute maximum length of any character value that can be stored into a `CHAR` column is 2000 bytes. For example, even if you define the column length to be 2000 characters, Oracle returns an error if you try to insert a 2000-character value in which one or more code points are wider than 1 byte. The value of *`size`* in characters is a length constraint, not guaranteed capacity. If you want a `CHAR` column to be always able to store *`size`* characters in any database character set, use a value of *`size`* that is less than or equal to 500.

To ensure proper data conversion between databases and clients with different character sets, you must ensure that `CHAR` data consists of well-formed strings.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on character set support and Data Type Comparison Rules for information on comparison semantics

NCHAR Data Type

The `NCHAR` data type specifies a fixed-length character string in the national character set. You specify the national character set as either AL16UTF16 or UTF8 when you create your database. AL16UTF16 and UTF8 are two encoding forms of the Unicode character set (UTF-16 and CESU-8, correspondingly) and hence `NCHAR` is a Unicode-only data type.

When you create a table with an NCHAR column, you specify the column length as *size* characters, or more precisely, code points in the national character set. One code point has always 2 bytes in AL16UTF16 and from 1 to 3 bytes in UTF8, depending on the particular character encoded by the code point.

Oracle ensures that all values stored in an NCHAR column have the length of *size* characters. If you insert a value that is shorter than the column length, then Oracle blank-pads the value to the column length. If you try to insert a value that is too long for the column, then Oracle returns an error. Note that if the national character set is UTF8, blank-padding does not guarantee that all column values have the same byte length.

You can omit *size* from the column definition. The default value is 1.

The maximum value of *size* is 1000 characters when the national character set is AL16UTF16, and 2000 characters when the national character set is UTF8. However, independently, the absolute maximum length of any character value that can be stored into an NCHAR column is 2000 bytes. For example, even if you define the column length to be 1000 characters, Oracle returns an error if you try to insert a 1000-character value but the national character set is UTF8 and all code points are 3 bytes wide. The value of *size* is a length constraint, not guaranteed capacity. If you want an NCHAR column to be always able to store *size* characters in both national character sets, use a value of *size* that is less than or equal to 666.

To ensure proper data conversion between databases and clients with different character sets, you must ensure that NCHAR data consists of well-formed strings.

If you assign a CHAR value to an NCHAR column, the value is implicitly converted from the database character set to the national character set. If you assign an NCHAR value to a CHAR column, the value is implicitly converted from the national character set to the database character set. If some of the characters from the NCHAR value cannot be represented in the database character set, then if the value of the session parameter NLS_NCHAR_CONV_EXCP is TRUE, then Oracle reports an error. If the value of the parameter is FALSE, non-representable characters are replaced with the default replacement character of the database character set, which is usually the question mark '?' or the inverted question mark '¿'.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information on Unicode data type support

VARCHAR2 Data Type

The VARCHAR2 data type specifies a variable-length character string in the database character set. You specify the database character set when you create your database.

When you create a table with a VARCHAR2 column, you must specify the column length as *size* optionally followed by a length qualifier. The qualifier BYTE denotes byte length semantics while the qualifier CHAR denotes character length semantics. In the byte length semantics, *size* is the maximum number of bytes that can be stored in the column. In the character length semantics, *size* is the maximum number of code points in the database character set that can be stored in the column. A code point may have from 1 to 4 bytes depending on the database character set and the particular character encoded by the code point. Oracle recommends that you specify one of the length qualifiers to explicitly document the desired length semantics of the column. If you do not specify a qualifier, the value of the NLS_LENGTH_SEMANTICS parameter of the session

creating the column defines the length semantics, unless the table belongs to the schema SYS, in which case the default semantics is BYTE.

Oracle stores a character value in a VARCHAR2 column exactly as you specify it, without any blank-padding, provided the value does not exceed the length of the column. If you try to insert a value that exceeds the specified length, then Oracle returns an error.

The minimum value of *size* is 1. The maximum value is:

- 32767 bytes if MAX_STRING_SIZE = EXTENDED
- 4000 bytes if MAX_STRING_SIZE = STANDARD

Refer to Extended Data Types for more information on the MAX_STRING_SIZE initialization parameter and the internal storage mechanisms for extended data types.

While *size* may be expressed in bytes or characters (code points) the independent absolute maximum length of any character value that can be stored into a VARCHAR2 column is 32767 or 4000 bytes, depending on MAX_STRING_SIZE. For example, even if you define the column length to be 32767 characters, Oracle returns an error if you try to insert a 32767-character value in which one or more code points are wider than 1 byte. The value of *size* in characters is a length constraint, not guaranteed capacity. If you want a VARCHAR2 column to be always able to store *size* characters in any database character set, use a value of *size* that is less than or equal to 8191, if MAX_STRING_SIZE = EXTENDED, or 1000, if MAX_STRING_SIZE = STANDARD.

Oracle compares VARCHAR2 values using non-padded comparison semantics.

To ensure proper data conversion between databases with different character sets, you must ensure that VARCHAR2 data consists of well-formed strings. See *Oracle Database Globalization Support Guide* for more information on character set support.

> **See Also:**
>
> Data Type Comparison Rules for information on comparison semantics

VARCHAR Data Type

Do not use the VARCHAR data type. Use the VARCHAR2 data type instead. Although the VARCHAR data type is currently synonymous with VARCHAR2, the VARCHAR data type is scheduled to be redefined as a separate data type used for variable-length character strings compared with different comparison semantics.

NVARCHAR2 Data Type

The NVARCHAR2 data type specifies a variable-length character string in the national character set. You specify the national character set as either AL16UTF16 or UTF8 when you create your database. AL16UTF16 and UTF8 are two encoding forms of the Unicode character set (UTF-16 and CESU-8, correspondingly) and hence NVARCHAR2 is a Unicode-only data type.

When you create a table with an NVARCHAR2 column, you must specify the column length as *size* characters, or more precisely, code points in the national character set. One code point has always 2 bytes in AL16UTF16 and from 1 to 3 bytes in UTF8, depending on the particular character encoded by the code point.

Oracle stores a character value in an NVARCHAR2 column exactly as you specify it, without any blank-padding, provided the value does not exceed the length of the column. If you try to insert a value that exceeds the specified length, then Oracle returns an error.

The minimum value of *size* is 1. The maximum value is:

- 16383 if MAX_STRING_SIZE = EXTENDED and the national character set is AL16UTF16
- 32767 if MAX_STRING_SIZE = EXTENDED and the national character set is UTF8
- 2000 if MAX_STRING_SIZE = STANDARD and the national character set is AL16UTF16
- 4000 if MAX_STRING_SIZE = STANDARD and the national character set is UTF8

Refer to Extended Data Types for more information on the MAX_STRING_SIZE initialization parameter and the internal storage mechanisms for extended data types.

Independently of the maximum column length in characters, the absolute maximum length of any value that can be stored into an NVARCHAR2 column is 32767 or 4000 bytes, depending on MAX_STRING_SIZE. For example, even if you define the column length to be 16383 characters, Oracle returns an error if you try to insert a 16383-character value but the national character set is UTF8 and all code points are 3 bytes wide. The value of *size* is a length constraint, not guaranteed capacity. If you want an NVARCHAR2 column to be always able to store *size* characters in both national character sets, use a value of *size* that is less than or equal to 10922, if MAX_STRING_SIZE = EXTENDED, or 1333, if MAX_STRING_SIZE = STANDARD.

Oracle compares NVARCHAR2 values using non-padded comparison semantics.

To ensure proper data conversion between databases and clients with different character sets, you must ensure that NVARCHAR2 data consists of well-formed strings.

If you assign a VARCHAR2 value to an NVARCHAR2 column, the value is implicitly converted from the database character set to the national character set. If you assign an NVARCHAR2 value to a VARCHAR2 column, the value is implicitly converted from the national character set to the database character set. If some of the characters from the NVARCHAR2 value cannot be represented in the database character set, then if the value of the session parameter NLS_NCHAR_CONV_EXCP is TRUE, then Oracle reports an error. If the value of the parameter is FALSE, non-representable characters are replaced with the default replacement character of the database character set, which is usually the question mark '?' or the inverted question mark '¿'.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information on Unicode data type support.

Numeric Data Types

The Oracle Database numeric data types store positive and negative fixed and floating-point numbers, zero, infinity, and values that are the undefined result of an operation—"not a number" or NAN. For information on specifying numeric data types as literals, refer to Numeric Literals .

NUMBER Data Type

The NUMBER data type stores zero as well as positive and negative fixed numbers with absolute values from 1.0×10^{-130} to but not including 1.0×10^{126}. If you specify an arithmetic expression

whose value has an absolute value greater than or equal to 1.0 x 10^{126}, then Oracle returns an error. Each `NUMBER` value requires from 1 to 22 bytes.

Specify a fixed-point number using the following form:

```
NUMBER(p,s)
```

where:

- *p* is the **precision**, or the maximum number of significant decimal digits, where the most significant digit is the left-most nonzero digit, and the least significant digit is the right-most known digit. Oracle guarantees the portability of numbers with precision of up to 20 base-100 digits, which is equivalent to 39 or 40 decimal digits depending on the position of the decimal point.
- *s* is the **scale**, or the number of digits from the decimal point to the least significant digit. The scale can range from -84 to 127.
  - Positive scale is the number of significant digits to the right of the decimal point to and including the least significant digit.
  - Negative scale is the number of significant digits to the left of the decimal point, to but not including the least significant digit. For negative scale the least significant digit is on the left side of the decimal point, because the actual data is rounded to the specified number of places to the left of the decimal point. For example, a specification of (10,-2) means to round to hundreds.

Scale can be greater than precision, most commonly when `e` notation is used. When scale is greater than precision, the precision specifies the maximum number of significant digits to the right of the decimal point. For example, a column defined as `NUMBER(4,5)` requires a zero for the first digit after the decimal point and rounds all values past the fifth digit after the decimal point.

It is good practice to specify the scale and precision of a fixed-point number column for extra integrity checking on input. Specifying scale and precision does not force all values to a fixed length. If a value exceeds the precision, then Oracle returns an error. If a value exceeds the scale, then Oracle rounds it.

Specify an integer using the following form:

```
NUMBER(p)
```

This represents a fixed-point number with precision *p* and scale 0 and is equivalent to `NUMBER(p,0)`.

Specify a floating-point number using the following form:

```
NUMBER
```

The absence of precision and scale designators specifies the maximum range and precision for an Oracle number.

> **See Also:**
>
> Floating-Point Numbers

Table 2-2 show how Oracle stores data using different precisions and scales.

Table 2-2 Storage of Scale and Precision

Actual Data	Specified As	Stored As
123.89	NUMBER	123.89
123.89	NUMBER(3)	124
123.89	NUMBER(3,2)	exceeds precision
123.89	NUMBER(4,2)	exceeds precision
123.89	NUMBER(5,2)	123.89
123.89	NUMBER(6,1)	123.9
123.89	NUMBER(6,-2)	100
.01234	NUMBER(4,5)	.01234
.00012	NUMBER(4,5)	.00012
.000127	NUMBER(4,5)	.00013
.0000012	NUMBER(2,7)	.0000012
.00000123	NUMBER(2,7)	.0000012
1.2e-4	NUMBER(2,5)	0.00012
1.2e-5	NUMBER(2,5)	0.00001

FLOAT Data Type

The FLOAT data type is a subtype of NUMBER. It can be specified with or without precision, which has the same definition it has for NUMBER and can range from 1 to 126. Scale cannot be specified, but is interpreted from the data. Each FLOAT value requires from 1 to 22 bytes.

To convert from binary to decimal precision, multiply *n* by 0.30103. To convert from decimal to binary precision, multiply the decimal precision by 3.32193. The maximum of 126 digits of binary precision is roughly equivalent to 38 digits of decimal precision.

The difference between NUMBER and FLOAT is best illustrated by example. In the following example the same values are inserted into NUMBER and FLOAT columns:

```
CREATE TABLE test (col1 NUMBER(5,2), col2 FLOAT(5));

INSERT INTO test VALUES (1.23, 1.23);
INSERT INTO test VALUES (7.89, 7.89);
INSERT INTO test VALUES (12.79, 12.79);
INSERT INTO test VALUES (123.45, 123.45);

SELECT * FROM test;

      COL1       COL2
---------- ----------
      1.23        1.2
      7.89        7.9
     12.79         13
    123.45        120
```

In this example, the FLOAT value returned cannot exceed 5 binary digits. The largest decimal number that can be represented by 5 binary digits is 31. The last row contains decimal values that exceed 31. Therefore, the FLOAT value must be truncated so that its significant digits do

not require more than 5 binary digits. Thus 123.45 is rounded to 120, which has only two significant decimal digits, requiring only 4 binary digits.

Oracle Database uses the Oracle `FLOAT` data type internally when converting ANSI `FLOAT` data. Oracle `FLOAT` is available for you to use, but Oracle recommends that you use the `BINARY_FLOAT` and `BINARY_DOUBLE` data types instead, as they are more robust. Refer to Floating-Point Numbers for more information.

Floating-Point Numbers

Floating-point numbers can have a decimal point anywhere from the first to the last digit or can have no decimal point at all. An exponent may optionally be used following the number to increase the range, for example, 1.777 e^{-20}. A scale value is not applicable to floating-point numbers, because the number of digits that can appear after the decimal point is not restricted.

Binary floating-point numbers differ from `NUMBER` in the way the values are stored internally by Oracle Database. Values are stored using decimal precision for `NUMBER`. All literals that are within the range and precision supported by `NUMBER` are stored exactly as `NUMBER`. Literals are stored exactly because literals are expressed using decimal precision (the digits 0 through 9). Binary floating-point numbers are stored using binary precision (the digits 0 and 1). Such a storage scheme cannot represent all values using decimal precision exactly. Frequently, the error that occurs when converting a value from decimal to binary precision is undone when the value is converted back from binary to decimal precision. The literal 0.1 is such an example.

Oracle Database provides two numeric data types exclusively for floating-point numbers:

BINARY_FLOAT

`BINARY_FLOAT` is a 32-bit, single-precision floating-point number data type. Each `BINARY_FLOAT` value requires 4 bytes.

BINARY_DOUBLE

`BINARY_DOUBLE` is a 64-bit, double-precision floating-point number data type. Each `BINARY_DOUBLE` value requires 8 bytes.

In a `NUMBER` column, floating point numbers have decimal precision. In a `BINARY_FLOAT` or `BINARY_DOUBLE` column, floating-point numbers have binary precision. The binary floating-point numbers support the special values infinity and `NaN` (not a number).

You can specify floating-point numbers within the limits listed in Table 2-3. The format for specifying floating-point numbers is defined in Numeric Literals .

Table 2-3 Floating Point Number Limits

Value	BINARY_FLOAT	BINARY_DOUBLE
Maximum positive finite value	3.40282E+38F	1.79769313486231E+308
Minimum positive finite value	1.17549E-38F	2.22507485850720E-308

IEEE754 Conformance

The Oracle implementation of floating-point data types conforms substantially with the Institute of Electrical and Electronics Engineers (IEEE) Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-1985 (IEEE754). The floating-point data types conform to IEEE754 in the following areas:

- The SQL function SQRT implements square root. See SQRT .
- The SQL function REMAINDER implements remainder. See REMAINDER .
- Arithmetic operators conform. See Arithmetic Operators .
- Comparison operators conform, except for comparisons with NaN. Oracle orders NaN greatest with respect to all other values, and evaluates NaN equal to NaN. See Floating-Point Conditions .
- Conversion operators conform. See Conversion Functions .
- The default rounding mode is supported.
- The default exception handling mode is supported.
- The special values INF, -INF, and NaN are supported. See Floating-Point Conditions .
- Rounding of BINARY_FLOAT and BINARY_DOUBLE values to integer-valued BINARY_FLOAT and BINARY_DOUBLE values is provided by the SQL functions ROUND, TRUNC, CEIL, and FLOOR.
- Rounding of BINARY_FLOAT/BINARY_DOUBLE to decimal and decimal to BINARY_FLOAT/BINARY_DOUBLE is provided by the SQL functions TO_CHAR, TO_NUMBER, TO_NCHAR, TO_BINARY_FLOAT, TO_BINARY_DOUBLE, and CAST.

The floating-point data types do not conform to IEEE754 in the following areas:

- -0 is coerced to +0.
- Comparison with NaN is not supported.
- All NaN values are coerced to either BINARY_FLOAT_NAN or BINARY_DOUBLE_NAN.
- Non-default rounding modes are not supported.
- Non-default exception handling mode are not supported.

Numeric Precedence

Numeric precedence determines, for operations that support numeric data types, the data type Oracle uses if the arguments to the operation have different data types. BINARY_DOUBLE has the highest numeric precedence, followed by BINARY_FLOAT, and finally by NUMBER. Therefore, in any operation on multiple numeric values:

- If any of the operands is BINARY_DOUBLE, then Oracle attempts to convert all the operands implicitly to BINARY_DOUBLE before performing the operation.
- If none of the operands is BINARY_DOUBLE but any of the operands is BINARY_FLOAT, then Oracle attempts to convert all the operands implicitly to BINARY_FLOAT before performing the operation.
- Otherwise, Oracle attempts to convert all the operands to NUMBER before performing the operation.

If any implicit conversion is needed and fails, then the operation fails. Refer to Table 2-9 for more information on implicit conversion.

In the context of other data types, numeric data types have lower precedence than the datetime/interval data types and higher precedence than character and all other data types.

LONG Data Type

> **Note:**
>
> All forms of LONG data types (LONG, LONG RAW, LONG VARCHAR, LONG VARRAW) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the LONG data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use CLOB and NCLOB data types for large amounts of character data.

Do not create tables with LONG columns. Use LOB columns (CLOB, NCLOB, BLOB) instead. LONG columns are supported only for backward compatibility.

LONG columns store variable-length character strings containing up to 2 gigabytes -1, or 2^{31}-1 bytes. LONG columns have many of the characteristics of VARCHAR2 columns. You can use LONG columns to store long text strings. The length of LONG values may be limited by the memory available on your computer. LONG literals are formed as described for Text Literals .

Oracle also recommends that you convert existing LONG columns to LOB columns. LOB columns are subject to far fewer restrictions than LONG columns. Further, LOB functionality is enhanced in every release, whereas LONG functionality has been static for several releases. See the *modify_col_properties* clause of ALTER TABLE and TO_LOB for more information on converting LONG columns to LOB.

You can reference LONG columns in SQL statements in these places:

- SELECT lists
- SET clauses of UPDATE statements
- VALUES clauses of INSERT statements

The use of LONG values is subject to these restrictions:

- A table can contain only one LONG column.
- You cannot create an object type with a LONG attribute.
- LONG columns cannot appear in WHERE clauses or in integrity constraints (except that they can appear in NULL and NOT NULL constraints).
- LONG columns cannot be indexed.
- LONG data cannot be specified in regular expressions.
- A stored function cannot return a LONG value.
- You can declare a variable or argument of a PL/SQL program unit using the LONG data type. However, you cannot then call the program unit from SQL.
- Within a single SQL statement, all LONG columns, updated tables, and locked tables must be located on the same database.
- LONG and LONG RAW columns cannot be used in distributed SQL statements and cannot be replicated.

- If a table has both LONG and LOB columns, then you cannot bind more than 4000 bytes of data to both the LONG and LOB columns in the same SQL statement. However, you can bind more than 4000 bytes of data to either the LONG or the LOB column.

In addition, LONG columns cannot appear in these parts of SQL statements:

- GROUP BY clauses, ORDER BY clauses, or CONNECT BY clauses or with the DISTINCT operator in SELECT statements
- The UNIQUE operator of a SELECT statement
- The column list of a CREATE CLUSTER statement
- The CLUSTER clause of a CREATE MATERIALIZED VIEW statement
- SQL built-in functions, expressions, or conditions
- SELECT lists of queries containing GROUP BY clauses
- SELECT lists of subqueries or queries combined by the UNION, INTERSECT, or MINUS set operators
- SELECT lists of CREATE TABLE ... AS SELECT statements
- ALTER TABLE ... MOVE statements
- SELECT lists in subqueries in INSERT statements

Triggers can use the LONG data type in the following manner:

- A SQL statement within a trigger can insert data into a LONG column.
- If data from a LONG column can be converted to a constrained data type (such as CHAR and VARCHAR2), then a LONG column can be referenced in a SQL statement within a trigger.
- Variables in triggers cannot be declared using the LONG data type.
- :NEW and :OLD cannot be used with LONG columns.

You can use Oracle Call Interface functions to retrieve a portion of a LONG value from the database.

> **See Also:**
>
> *Oracle Call Interface Developer's Guide*

Datetime and Interval Data Types

The datetime data types are DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE. Values of datetime data types are sometimes called **datetimes**. The interval data types are INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND. Values of interval data types are sometimes called **intervals**. For information on expressing datetime and interval values as literals, refer to Datetime Literals and Interval Literals.

Both datetimes and intervals are made up of fields. The values of these fields determine the value of the data type. Table 2-4 lists the datetime fields and their possible values for datetimes and intervals.

To avoid unexpected results in your DML operations on datetime data, you can verify the database and session time zones by querying the built-in SQL functions DBTIMEZONE and

SESSIONTIMEZONE. If the time zones have not been set manually, then Oracle Database uses the operating system time zone by default. If the operating system time zone is not a valid Oracle time zone, then Oracle uses UTC as the default value.

Table 2-4 Datetime Fields and Values

Datetime Field	Valid Values for Datetime	Valid Values for INTERVAL
YEAR	-4712 to 9999 (excluding year 0)	Any positive or negative integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the current NLS calendar parameter)	Any positive or negative integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds. The 9(n) portion is not applicable for DATE.	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (This range accommodates daylight saving time changes.) Not applicable for DATE or TIMESTAMP.	Not applicable
TIMEZONE_MINUTE (See note at end of table)	00 to 59. Not applicable for DATE or TIMESTAMP.	Not applicable
TIMEZONE_REGION	Query the TZNAME column of the V$TIMEZONE_NAMES data dictionary view. Not applicable for DATE or TIMESTAMP. For a complete listing of all time zone region names, refer to *Oracle Database Globalization Support Guide*.	Not applicable
TIMEZONE_ABBR	Query the TZABBREV column of the V$TIMEZONE_NAMES data dictionary view. Not applicable for DATE or TIMESTAMP.	Not applicable

> **Note:**
>
> TIMEZONE_HOUR and TIMEZONE_MINUTE are specified together and interpreted as an entity in the format +|- *hh*:*mi*, with values ranging from -12:59 to +14:00. Refer to *Oracle Data Provider for .NET Developer's Guide* for information on specifying time zone values for that API.

DATE Data Type

The DATE data type stores date and time information. Although date and time information can be represented in both character and number data types, the DATE data type has special associated properties. For each DATE value, Oracle stores the following information: year, month, day, hour, minute, and second.

You can specify a DATE value as a literal, or you can convert a character or numeric value to a date value with the TO_DATE function. For examples of expressing DATE values in both these ways, refer to Datetime Literals .

Using Julian Days

A Julian day number is the number of days since January 1, 4712 BC. Julian days allow continuous dating from a common reference. You can use the date format model "J" with date functions `TO_DATE` and `TO_CHAR` to convert between Oracle `DATE` values and their Julian equivalents.

> **Note:**
>
> Oracle Database uses the astronomical system of calculating Julian days, in which the year 4713 BC is specified as -4712. The historical system of calculating Julian days, in contrast, specifies 4713 BC as -4713. If you are comparing Oracle Julian days with values calculated using the historical system, then take care to allow for the 365-day difference in BC dates.

The default date values are determined as follows:

- The year is the current year, as returned by `SYSDATE`.
- The month is the current month, as returned by `SYSDATE`.
- The day is 01 (the first day of the month).
- The hour, minute, and second are all 0.

These default values are used in a query that requests date values where the date itself is not specified, as in the following example, which is issued in the month of May:

```
SELECT TO_DATE('2009', 'YYYY')
  FROM DUAL;

TO_DATE('
---------
01-MAY-09
```

Example

This statement returns the Julian equivalent of January 1, 2009:

```
SELECT TO_CHAR(TO_DATE('01-01-2009', 'MM-DD-YYYY'),'J')
    FROM DUAL;

TO_CHAR
-------
2454833
```

> **See Also:**
>
> Selecting from the DUAL Table for a description of the `DUAL` table

TIMESTAMP Data Type

The `TIMESTAMP` data type is an extension of the `DATE` data type. It stores the year, month, and day of the `DATE` data type, plus hour, minute, and second values. This data type is useful for

storing precise time values and for collecting and evaluating date information across geographic regions. Specify the `TIMESTAMP` data type as follows:

```
TIMESTAMP [(fractional_seconds_precision)]
```

where *fractional_seconds_precision* optionally specifies the number of digits Oracle stores in the fractional part of the `SECOND` datetime field. When you create a column of this data type, the value can be a number in the range 0 to 9. The default is 6.

> **See Also:**
>
> TO_TIMESTAMP for information on converting character data to `TIMESTAMP` data

TIMESTAMP WITH TIME ZONE Data Type

`TIMESTAMP WITH TIME ZONE` is a variant of `TIMESTAMP` that includes a **time zone region name** or a **time zone offset** in its value. The time zone offset is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). This data type is useful for preserving local time zone information.

Specify the `TIMESTAMP WITH TIME ZONE` data type as follows:

```
TIMESTAMP [(fractional_seconds_precision)] WITH TIME ZONE
```

where *fractional_seconds_precision* optionally specifies the number of digits Oracle stores in the fractional part of the `SECOND` datetime field. When you create a column of this data type, the value can be a number in the range 0 to 9. The default is 6.

Oracle time zone data is derived from the public domain information available at `http://www.iana.org/time-zones/`. Oracle time zone data may not reflect the most recent data available at this site.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for more information on Oracle time zone data
> - Support for Daylight Saving Times and Table 2-20 for information on daylight saving support
> - TO_TIMESTAMP_TZ for information on converting character data to `TIMESTAMP WITH TIME ZONE` data
> - ALTER SESSION for information on the `ERROR_ON_OVERLAP_TIME` session parameter

TIMESTAMP WITH LOCAL TIME ZONE Data Type

`TIMESTAMP WITH LOCAL TIME ZONE` is another variant of `TIMESTAMP` that is sensitive to time zone information. It differs from `TIMESTAMP WITH TIME ZONE` in that data stored in the database is normalized to the database time zone, and the time zone information is not stored as part of the column data. When a user retrieves the data, Oracle returns it in the user's local session

time zone. This data type is useful for date information that is always to be displayed in the time zone of the client system in a two-tier application.

Specify the `TIMESTAMP WITH LOCAL TIME ZONE` data type as follows:

```
TIMESTAMP [(fractional_seconds_precision)] WITH LOCAL TIME ZONE
```

where *fractional_seconds_precision* optionally specifies the number of digits Oracle stores in the fractional part of the `SECOND` datetime field. When you create a column of this data type, the value can be a number in the range 0 to 9. The default is 6.

Oracle time zone data is derived from the public domain information available at http://www.iana.org/time-zones/. Oracle time zone data may not reflect the most recent data available at this site.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for more information on Oracle time zone data
> - *Oracle Database Development Guide* for examples of using this data type and CAST for information on converting character data to `TIMESTAMP WITH LOCAL TIME ZONE`

INTERVAL YEAR TO MONTH Data Type

`INTERVAL YEAR TO MONTH` stores a period of time using the `YEAR` and `MONTH` datetime fields. This data type is useful for representing the difference between two datetime values when only the year and month values are significant.

Specify `INTERVAL YEAR TO MONTH` as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

where *year_precision* is the number of digits in the `YEAR` datetime field. The default value of *year_precision* is 2.

You have a great deal of flexibility when specifying interval values as literals. Refer to Interval Literals for detailed information on specifying interval values as literals. Also see Datetime and Interval Examples for an example using intervals.

INTERVAL DAY TO SECOND Data Type

`INTERVAL DAY TO SECOND` stores a period of time in terms of days, hours, minutes, and seconds. This data type is useful for representing the precise difference between two datetime values.

Specify this data type as follows:

```
INTERVAL DAY [(day_precision)]
   TO SECOND [(fractional_seconds_precision)]
```

where

- *day_precision* is the number of digits in the `DAY` datetime field. Accepted values are 0 to 9. The default is 2.

- *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.

You have a great deal of flexibility when specifying interval values as literals. Refer to Interval Literals for detailed information on specify interval values as literals. Also see Datetime and Interval Examples for an example using intervals.

Datetime/Interval Arithmetic

You can perform a number of arithmetic operations on date (DATE), timestamp (TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE) and interval (INTERVAL DAY TO SECOND and INTERVAL YEAR TO MONTH) data. Oracle calculates the results based on the following rules:

- You can use NUMBER constants in arithmetic operations on date and timestamp values, but not interval values. Oracle internally converts timestamp values to date values and interprets NUMBER constants in arithmetic datetime and interval expressions as numbers of days. For example, SYSDATE + 1 is tomorrow. SYSDATE - 7 is one week ago. SYSDATE + (10/1440) is ten minutes from now. Subtracting the hire_date column of the sample table employees from SYSDATE returns the number of days since each employee was hired. You cannot multiply or divide date or timestamp values.
- Oracle implicitly converts BINARY_FLOAT and BINARY_DOUBLE operands to NUMBER.
- Each DATE value contains a time component, and the result of many date operations include a fraction. This fraction means a portion of one day. For example, 1.5 days is 36 hours. These fractions are also returned by Oracle built-in functions for common operations on DATE data. For example, the MONTHS_BETWEEN function returns the number of months between two dates. The fractional portion of the result represents that portion of a 31-day month.
- If one operand is a DATE value or a numeric value, neither of which contains time zone or fractional seconds components, then:
  - Oracle implicitly converts the other operand to DATE data. The exception is multiplication of a numeric value times an interval, which returns an interval.
  - If the other operand has a time zone value, then Oracle uses the session time zone in the returned value.
  - If the other operand has a fractional seconds value, then the fractional seconds value is lost.
- When you pass a timestamp, interval, or numeric value to a built-in function that was designed only for the DATE data type, Oracle implicitly converts the non-DATE value to a DATE value. Refer to Datetime Functions for information on which functions cause implicit conversion to DATE.
- When interval calculations return a datetime value, the result must be an actual datetime value or the database returns an error. For example, the next two statements return errors:

```
SELECT TO_DATE('31-AUG-2004','DD-MON-YYYY') + TO_YMINTERVAL('0-1')
  FROM DUAL;

SELECT TO_DATE('29-FEB-2004','DD-MON-YYYY') + TO_YMINTERVAL('1-0')
  FROM DUAL;
```

The first fails because adding one month to a 31-day month would result in September 31, which is not a valid date. The second fails because adding one year to a date that exists only every four years is not valid. However, the next statement succeeds, because adding four years to a February 29 date is valid:

```
SELECT TO_DATE('29-FEB-2004', 'DD-MON-YYYY') + TO_YMINTERVAL('4-0')
  FROM DUAL;

TO_DATE('
---------
29-FEB-08
```

- Oracle performs all timestamp arithmetic in UTC time. For `TIMESTAMP WITH LOCAL TIME ZONE`, Oracle converts the datetime value from the database time zone to UTC and converts back to the database time zone after performing the arithmetic. For `TIMESTAMP WITH TIME ZONE`, the datetime value is always in UTC, so no conversion is necessary.

Table 2-5 is a matrix of datetime arithmetic operations. Dashes represent operations that are not supported.

Table 2-5 Matrix of Datetime Arithmetic

Operand & Operator	DATE	TIMESTAMP	INTERVAL	Numeric
DATE				
+	–	–	DATE	DATE
-	NUMBER	INTERVAL	DATE	DATE
*****	–	–	–	–
/	–	–	–	–
TIMESTAMP				
+	–	–	TIMESTAMP	DATE
-	INTERVAL	INTERVAL	TIMESTAMP	DATE
*****	–	–	–	–
/	–	–	–	–
INTERVAL				
+	DATE	TIMESTAMP	INTERVAL	–
-	–	–	INTERVAL	–
*****	–	–	–	INTERVAL
/	–	–	–	INTERVAL
Numeric				
+	DATE	DATE	–	NA
-	–	–	–	NA
*****	–	–	INTERVAL	NA
/	–	–	–	NA

Examples

You can add an interval value expression to a start time. Consider the sample table `oe.orders` with a column `order_date`. The following statement adds 30 days to the value of the `order_date` column:

```
SELECT order_id, order_date + INTERVAL '30' DAY AS "Due Date"
  FROM orders
  ORDER BY order_id, "Due Date";
```

Support for Daylight Saving Times

Oracle Database automatically determines, for any given time zone region, whether daylight saving is in effect and returns local time values accordingly. The datetime value is sufficient for Oracle to determine whether daylight saving time is in effect for a given region in all cases except **boundary cases**. A boundary case occurs during the period when daylight saving goes into or comes out of effect. For example, in the US-Pacific region, when daylight saving goes into effect, the time changes from 2:00 a.m. to 3:00 a.m. The one hour interval between 2 and 3 a.m. does not exist. When daylight saving goes out of effect, the time changes from 2:00 a.m. back to 1:00 a.m., and the one-hour interval between 1 and 2 a.m. is repeated.

To resolve these boundary cases, Oracle uses the `TZR` and `TZD` format elements, as described in Table 2-20. `TZR` represents the time zone region name in datetime input strings. Examples are '`Australia/North`', '`UTC`', and '`Singapore`'. `TZD` represents an abbreviated form of the time zone region name with daylight saving information. Examples are '`PST`' for US/Pacific standard time and '`PDT`' for US/Pacific daylight time. To see a listing of valid values for the `TZR` and `TZD` format elements, query the `TZNAME` and `TZABBREV` columns of the `V$TIMEZONE_NAMES` dynamic performance view.

> **Note:**
>
> Time zone region names are needed by the daylight saving feature. These names are stored in two types of time zone files: one large and one small. One of these files is the default file, depending on your environment and the release of Oracle Database you are using. For more information regarding time zone files and names, see *Oracle Database Globalization Support Guide*.

For a complete listing of the time zone region names in both files, refer to *Oracle Database Globalization Support Guide*.

Oracle time zone data is derived from the public domain information available at `http://www.iana.org/time-zones/`. Oracle time zone data may not reflect the most recent data available at this site.

> **See Also:**
>
> - Datetime Format Models for information on the format elements and the session parameter ERROR_ON_OVERLAP_TIME .
> - *Oracle Database Globalization Support Guide* for more information on Oracle time zone data
> - *Oracle Database Reference* for information on the dynamic performance views

Datetime and Interval Examples

The following example shows an `INTERVAL` aggregation query:

```
SELECT job_name,
       SUM( cpu_used )
  FROM DBA_SCHEDULER_JOB_RUN_DETAILS
```

```
  GROUP BY job_name
  HAVING SUM ( cpu_used ) > interval '5' minute;
```

The view `DBA_SCHEDULER_JOB_RUN_DETAILS` contains the log run details for all scheduler jobs in the database. The column `CPU_USED` of type `INTERVAL DAY(3) TO SECOND(2)` displays the amount of CPU used for the job run. This query returns the names of all the scheduler jobs that have lasted more than 5 minutes.

The following example shows how to specify some datetime and interval data types.

```
CREATE TABLE time_table
  (start_time    TIMESTAMP,
   duration_1    INTERVAL DAY (6) TO SECOND (5),
   duration_2    INTERVAL YEAR TO MONTH);
```

The `start_time` column is of type `TIMESTAMP`. The implicit fractional seconds precision of `TIMESTAMP` is 6.

The `duration_1` column is of type `INTERVAL DAY TO SECOND`. The maximum number of digits in field `DAY` is 6 and the maximum number of digits in the fractional second is 5. The maximum number of digits in all other datetime fields is 2.

The `duration_2` column is of type `INTERVAL YEAR TO MONTH`. The maximum number of digits of the value in each field (`YEAR` and `MONTH`) is 2.

Interval data types do not have format models. Therefore, to adjust their presentation, you must combine character functions such as `EXTRACT` and concatenate the components. For example, the following examples query the `hr.employees` and `oe.orders` tables, respectively, and change interval output from the form "*yy-mm*" to "*yy* years *mm* months" and from "*dd-hh*" to "*dddd* days *hh* hours":

```
SELECT last_name, EXTRACT(YEAR FROM (SYSDATE - hire_date) YEAR TO MONTH)
       || ' years '
       || EXTRACT(MONTH FROM (SYSDATE - hire_date) YEAR TO MONTH)
       || ' months'  "Interval"
  FROM employees;

LAST_NAME                 Interval
------------------------- --------------------
OConnell                  2 years 3 months
Grant                     1 years 9 months
Whalen                    6 years 1 months
Hartstein                 5 years 8 months
Fay                       4 years 2 months
Mavris                    7 years 4 months
Baer                      7 years 4 months
Higgins                   7 years 4 months
Gietz                     7 years 4 months
. . .

SELECT order_id, EXTRACT(DAY FROM (SYSDATE - order_date) DAY TO SECOND)
       || ' days '
       || EXTRACT(HOUR FROM (SYSDATE - order_date) DAY TO SECOND)
       || ' hours' "Interval"
  FROM orders;

  ORDER_ID Interval
---------- --------------------
      2458 780 days 23 hours
      2397 685 days 22 hours
      2454 733 days 21 hours
```

```
      2354 447 days 20 hours
      2358 635 days 20 hours
      2381 508 days 18 hours
      2440 765 days 17 hours
      2357 1365 days 16 hours
      2394 602 days 15 hours
      2435 763 days 15 hours
. . .
```

RAW and LONG RAW Data Types

> **Note:**
>
> All forms of `LONG` data types (`LONG`, `LONG RAW`, `LONG VARCHAR`, `LONG VARRAW`) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the `LONG` data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use `CLOB` and `NCLOB` data types for large amounts of character data.

The `RAW` and `LONG RAW` data types store data that is not to be explicitly converted by Oracle Database when moving data between different systems. These data types are intended for binary data or byte strings. For example, you can use `LONG RAW` to store graphics, sound, documents, or arrays of binary data, for which the interpretation is dependent on the use.

Oracle strongly recommends that you convert `LONG RAW` columns to binary LOB (`BLOB`) columns. LOB columns are subject to far fewer restrictions than `LONG` columns. See TO_LOB for more information.

`RAW` is a variable-length data type like `VARCHAR2`, except that Oracle Net (which connects client software to a database or one database to another) and the Oracle import and export utilities do not perform character conversion when transmitting `RAW` or `LONG RAW` data. In contrast, Oracle Net and the Oracle import and export utilities automatically convert `CHAR`, `VARCHAR2`, and `LONG` data between different database character sets, if data is transported between databases, or between the database character set and the client character set, if data is transported between a database and a client. The client character set is determined by the type of the client interface, such as OCI or JDBC, and the client configuration (for example, the `NLS_LANG` environment variable).

When Oracle implicitly converts `RAW` or `LONG RAW` data to character data, the resulting character value contains a hexadecimal representation of the binary input, where each character is a hexadecimal digit (`0-9`, `A-F`) representing four consecutive bits of `RAW` data. For example, one byte of `RAW` data with bits 11001011 becomes the value `CB`.

When Oracle implicitly converts character data to `RAW` or `LONG RAW`, it interprets each consecutive input character as a hexadecimal representation of four consecutive bits of binary data and builds the resulting `RAW` or `LONG RAW` value by concatenating those bits. If any of the input characters is not a hexadecimal digit (`0-9`, `A-F`, `a-f`), then an error is reported. If the number of characters is odd, then the result is undefined.

The SQL functions `RAWTOHEX` and `HEXTORAW` perform explicit conversions that are equivalent to the above implicit conversions. Other types of conversions between `RAW` and character data are possible with functions in the Oracle-supplied PL/SQL packages `UTL_RAW` and `UTL_I18N`.

Large Object (LOB) Data Types

The built-in LOB data types `BLOB`, `CLOB`, and `NCLOB` (stored internally) and `BFILE` (stored externally) can store large and unstructured data such as text, image, video, and spatial data. The size of `BLOB`, `CLOB`, and `NCLOB` data can be up to (2^{32}-1 bytes) * (the value of the `CHUNK` parameter of LOB storage). If the tablespaces in your database are of standard block size, and if you have used the default value of the `CHUNK` parameter of LOB storage when creating a LOB column, then this is equivalent to (2^{32}-1 bytes) * (database block size). `BFILE` data can be up to 2^{64}-1 bytes, although your operating system may impose restrictions on this maximum.

When creating a table, you can optionally specify different tablespace and storage characteristics for LOB columns or LOB object attributes from those specified for the table.

`CLOB`, `NCLOB`, and `BLOB` values up to approximately 4000 bytes are stored inline if you enable storage in row at the time the LOB column is created. LOBs greater than 4000 bytes are always stored externally. Refer to ENABLE STORAGE IN ROW for more information.

LOB columns contain LOB locators that can refer to internal (in the database) or external (outside the database) LOB values. Selecting a LOB from a table actually returns the LOB locator and not the entire LOB value. The `DBMS_LOB` package and Oracle Call Interface (OCI) operations on LOBs are performed through these locators.

LOBs are similar to `LONG` and `LONG RAW` types, but differ in the following ways:

- LOBs can be attributes of an object type (user-defined data type).
- The LOB locator is stored in the table column, either with or without the actual LOB value. `BLOB`, `NCLOB`, and `CLOB` values can be stored in separate tablespaces. `BFILE` data is stored in an external file on the server.
- When you access a LOB column, the locator is returned.
- A LOB can be up to (2^{32}-1 bytes)*(database block size) in size. `BFILE` data can be up to 2^{64}-1 bytes, although your operating system may impose restrictions on this maximum.
- LOBs permit efficient, random, piece-wise access to and manipulation of data.
- You can define more than one LOB column in a table.
- With the exception of `NCLOB`, you can define one or more LOB attributes in an object.
- You can declare LOB bind variables.
- You can select LOB columns and LOB attributes.
- You can insert a new row or update an existing row that contains one or more LOB columns or an object with one or more LOB attributes. In update operations, you can set the internal LOB value to `NULL`, empty, or replace the entire LOB with data. You can set the `BFILE` to `NULL` or make it point to a different file.
- You can update a LOB row-column intersection or a LOB attribute with another LOB row-column intersection or LOB attribute.
- You can delete a row containing a LOB column or LOB attribute and thereby also delete the LOB value. For BFILEs, the actual operating system file is not deleted.

You can access and populate rows of an inline LOB column (a LOB column stored in the database) or a LOB attribute (an attribute of an object type column stored in the database) simply by issuing an `INSERT` or `UPDATE` statement.

Restrictions on LOB Columns

LOB columns are subject to a number of rules and restrictions. See *Oracle Database SecureFiles and Large Objects Developer's Guide* for a complete listing.

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* and *Oracle Call Interface Developer's Guide* for more information about these interfaces and LOBs
> - the *modify_col_properties* clause of ALTER TABLE and TO_LOB for more information on converting LONG columns to LOB columns

BFILE Data Type

The BFILE data type enables access to binary file LOBs that are stored in file systems outside Oracle Database. A BFILE column or attribute stores a BFILE locator, which serves as a pointer to a binary file on the server file system. The locator maintains the directory name and the filename.

You can change the filename and path of a BFILE without affecting the base table by using the BFILENAME function. Refer to BFILENAME for more information on this built-in SQL function.

Binary file LOBs do not participate in transactions and are not recoverable. Rather, the underlying operating system provides file integrity and durability. BFILE data can be up to 2^{64}-1 bytes, although your operating system may impose restrictions on this maximum.

The database administrator must ensure that the external file exists and that Oracle processes have operating system read permissions on the file.

The BFILE data type enables read-only support of large binary files. You cannot modify or replicate such a file. Oracle provides APIs to access file data. The primary interfaces that you use to access file data are the DBMS_LOB package and Oracle Call Interface (OCI).

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* and *Oracle Call Interface Programmer's Guide* for more information about LOBs and CREATE DIRECTORY

BLOB Data Type

The BLOB data type stores unstructured binary large objects. BLOB objects can be thought of as bitstreams with no character set semantics. BLOB objects can store binary data up to (4 gigabytes -1) * (the value of the CHUNK parameter of LOB storage). If the tablespaces in your database are of standard block size, and if you have used the default value of the CHUNK parameter of LOB storage when creating a LOB column, then this is equivalent to (4 gigabytes - 1) * (database block size).

`BLOB` objects have full transactional support. Changes made through SQL, the `DBMS_LOB` package, or Oracle Call Interface (OCI) participate fully in the transaction. `BLOB` value manipulations can be committed and rolled back. However, you cannot save a `BLOB` locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

CLOB Data Type

The `CLOB` data type stores single-byte and multibyte character data. Both fixed-width and variable-width character sets are supported, and both use the database character set. `CLOB` objects can store up to (4 gigabytes -1) * (the value of the `CHUNK` parameter of LOB storage) of character data. If the tablespaces in your database are of standard block size, and if you have used the default value of the `CHUNK` parameter of LOB storage when creating a LOB column, then this is equivalent to (4 gigabytes - 1) * (database block size).

`CLOB` objects have full transactional support. Changes made through SQL, the `DBMS_LOB` package, or Oracle Call Interface (OCI) participate fully in the transaction. `CLOB` value manipulations can be committed and rolled back. However, you cannot save a `CLOB` locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

NCLOB Data Type

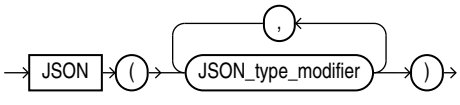
The `NCLOB` data type stores Unicode data. Both fixed-width and variable-width character sets are supported, and both use the national character set. `NCLOB` objects can store up to (4 gigabytes -1) * (the value of the `CHUNK` parameter of LOB storage) of character text data. If the tablespaces in your database are of standard block size, and if you have used the default value of the `CHUNK` parameter of LOB storage when creating a LOB column, then this is equivalent to (4 gigabytes - 1) * (database block size).

`NCLOB` objects have full transactional support. Changes made through SQL, the `DBMS_LOB` package, or OCI participate fully in the transaction. `NCLOB` value manipulations can be committed and rolled back. However, you cannot save an `NCLOB` locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information on Unicode data type support

JSON Data Type

***json_type_column*::=**



For the syntax of *`JSON_column_modifier`* see *JSON_type_modifier*::= in `IS_JSON` condition.

> **Note:**
>
> You can create tables with JSON data type only in ASSM tablespaces.

You can use the `JSON` data type to store JSON data natively in binary format. This improves query performance because textual JSON data no longer needs to be parsed. You can create `JSON` type instances from other SQL data, and conversely.

You must set the database initialization parameter `compatible` to `20` in order to use the new `JSON` data type.

The other SQL data types that support JSON data, besides `JSON` type, are `VARCHAR2`, `CLOB`, and `BLOB`. Non-JSON type data is called textual, or serialized, JSON data. It is unparsed character data.

You can use the `JSON` constructor function to convert textual JSON data to `JSON` type data.

To convert `JSON` type data to textual data, you can use the `JSON_SERIALIZE` function.

You can create complex `JSON` type data from non-`JSON` type data using the JSON generation functions: `JSON_OBJECT`, `JSON_ARRAY`, `JSON_OBJECTAGG`, and `JSON_ARRAYAGG`.

You can create a `JSON` type instance with a scalar JSON value using the function `JSON_SCALAR` .

In the other direction, you can use the function `JSON_VALUE` to query `JSON` type data and return an instance of a SQL object type or collection type.

When defining a JSON-type column you can follow the type keyword JSON with a JSON-type modifier, in parentheses: (`OBJECT`), (`ARRAY`), or (`SCALAR`). This requires the column content to be a JSON object, array, or scalar value, respectively. (This is similar to using `VARCHAR(42)` instead of just `VARCHAR2`.)

Modifier keyword `SCALAR` can be followed by a keyword that specifies the required type of scalar: `BOOLEAN`, `BINARY`, `BINARY_DOUBLE`, `BINARY_FLOAT`, `DATE`, `INTERVAL DAY TO SECOND` , `INTERVAL YEAR TO MONTH`, `NULL`, `NUMBER`, `STRING`, `TIMESTAMP`, or `TIMESTAMP WITH TIME ZONE`.

You can provide more than one modifier between the parentheses, separating them with commas. For example, (`OBJECT`, `ARRAY`) requires nonscalar values, and (`OBJECT`, `SCALAR DATE`) allows only objects or dates.

Create a Table with a JSON Type Column of JSON OBJECT: Example

The following table definition requires the JSON data type column `po_document` to be a JSON object by using a JSON modifier:

```
CREATE TABLE j_purchaseorder
  (id VARCHAR2 (32) NOT NULL PRIMARY KEY,
   date_loaded TIMESTAMP (6) WITH TIME ZONE,
   po_document JSON (OBJECT));
```

> **See Also:**
>
> - *JSON Data Type of the JSON Developer's Guide.*
> - For more information on creating a JSON column see *Creating a Table with a JSON Column of the JSON developer's Guide.*
> - For the syntax of JSON modifiers see IS JSON Condition

Extended Data Types

Beginning with Oracle Database 12*c*, you can specify a maximum size of 32767 bytes for the `VARCHAR2`, `NVARCHAR2`, and `RAW` data types. You can control whether your database supports this new maximum size by setting the initialization parameter `MAX_STRING_SIZE` as follows:

- If `MAX_STRING_SIZE = STANDARD`, then the size limits for releases prior to Oracle Database 12*c* apply: 4000 bytes for the `VARCHAR2` and `NVARCHAR2` data types, and 2000 bytes for the `RAW` data type. This is the default.
- If `MAX_STRING_SIZE = EXTENDED`, then the size limit is 32767 bytes for the `VARCHAR2`, `NVARCHAR2`, and `RAW` data types.

> **See Also:**
>
> Setting `MAX_STRING_SIZE = EXTENDED` may update database objects and possibly invalidate them. Refer to *Oracle Database Reference* for complete information on the implications of this parameter and how to set and enable this new functionality.

A `VARCHAR2` or `NVARCHAR2` data type with a declared size of greater than 4000 bytes, or a `RAW` data type with a declared size of greater than 2000 bytes, is an **extended data type**. Extended data type columns are stored out-of-line, leveraging Oracle's LOB technology. The LOB storage is always aligned with the table. In tablespaces managed with Automatic Segment Space Management (ASSM), extended data type columns are stored as SecureFiles LOBs. Otherwise, they are stored as BasicFiles LOBs. The use of LOBs as a storage mechanism is internal only. Therefore, you cannot manipulate these LOBs using the `DBMS_LOB` package.

> **Note:**
>
> - Oracle strongly recommends the use of SecureFiles LOBs as a storage mechanism. Note that BasicFiles LOBs impose restrictions on the capabilities of extended data type columns.
> - Extended data types are subject to the same rules and restrictions as LOBs. Refer to *Oracle Database SecureFiles and Large Objects Developer's Guide* for more information.

Note that, although you must set `MAX_STRING_SIZE = EXTENDED` in order to set the size of a `RAW` data type to greater than 2000 bytes, a `RAW` data type is stored as an out-of-line LOB only if it

has a size of greater than 4000 bytes. For example, you must set `MAX_STRING_SIZE = EXTENDED` in order to declare a `RAW(3000)` data type. However, the column is stored inline.

You can use extended data types just as you would standard data types, with the following considerations:

- For special considerations when creating an index on an extended data type column, or when requiring an index to enforce a primary key or unique constraint, see Creating an Index on an Extended Data Type Column.
- If the partitioning key column for a list partition is an extended data type column, then the list of values that you want to specify for a partition may exceed the 4K byte limit for the partition bounds. See the *list_partitions* clause of `CREATE TABLE` for information on how to work around this issue.
- The value of the initialization parameter `MAX_STRING_SIZE` affects the following:
  - The maximum length of a text literal. See Text Literals for more information.
  - The size limit for concatenating two character strings. See Concatenation Operator for more information.
  - The length of the collation key returned by the `NLSSORT` function. See NLSSORT .
  - The size of some of the attributes of the `XMLFormat` object. See XML Format Model for more information.
  - The size of some expressions in the following XML functions: XMLCOLATTVAL , XMLELEMENT , XMLFOREST , XMLPI , and XMLTABLE .

Boolean Data Type

Release 23 introduces the SQL boolean data type. The data type boolean has the truth values `TRUE` and `FALSE`. If there is no `NOT NULL` constraint, the boolean data type also supports the truth value `UNKNOWN` as the null value.

You can use the boolean data type wherever data type appears in Oracle SQL syntax. For example, you can specify a boolean column with the keywords `BOOLEAN` or `BOOL` in `CREATE TABLE`:

```
CREATE TABLE example (id NUMBER, c1 BOOLEAN, c2 BOOL);
```

You can use SQL keywords `TRUE`, `FALSE` and `NULL` to represent states “TRUE”, “FALSE”, and “NULL” respectively. For example, using the table `example` created above, you can insert the following:

```
INSERT INTO example VALUES (1, TRUE, NULL);

INSERT INTO example VALUES (2, FALSE, true);
```

You can use literals to represent "TRUE" and "FALSE" states. Case is not enforced in "TRUE" and "FALSE", you can have all lower case, all upper case, or a combination of upper and lower case. Leading and trailing white spaces are ignored.

Table 2-6 String Literals To Represent "TRUE" and "FALSE"

STATE	TRUE	FALSE
-	'true'	'false'

Table 2-6 (Cont.) String Literals To Represent "TRUE" and "FALSE"

STATE	TRUE	FALSE
-	'yes'	'no'
-	'on'	'off'
-	'1'	'0'
-	't'	'f'
-	'y'	'n'

Note that numbers are translated into boolean as follows:

- 0 translates to `FALSE`.
- Non 0 values like 42 or -3.14 translate to `TRUE`.

Given the table `example` created below with two boolean columns `c1` and `c2`:

```
CREATE TABLE example (id NUMBER, c1 BOOLEAN, c2 BOOL);
```

Insert into `example` the following rows:

```
INSERT INTO example VALUES (1, TRUE, NULL);
INSERT INTO example VALUES (2, FALSE, true);
INSERT INTO example VALUES (3, 0, 'off');
INSERT INTO example VALUES (4, 'no', 'yes');
INSERT INTO example VALUES (5, 'f', 't' );
INSERT INTO example VALUES (6, false, true);
INSERT INTO example VALUES (7, 'on', 'off');
INSERT INTO example VALUES (8, -3.14, 1);
```

`SELECT` of a boolean type column always returns `TRUE` , `FALSE`. A value of `NULL` returns nothing.

```
SELECT * FROM example;
ID          C1      C2
---------- -----    -----
1          TRUE
2          FALSE    TRUE
3          FALSE    FALSE
4          FALSE    TRUE
5          FALSE    TRUE
6          FALSE    TRUE
7          TRUE     FALSE
8          TRUE     TRUE
8 rows selected.
```

Constraints on Boolean Columns

The following constraints are supported on boolean columns:

- `NOT NULL`
- `UNIQUE`
- `PRIMARY KEY`

- FOREIGN KEY
- CHECK

Comparison and Assignment of Booleans

The following comparison operators are supported to compare boolean values: `=`, `!=`, `< >`, `<`, `<=`, `>`, `>=`, `GREATEST`, `LEAST`, `[NOT] IN`

```
SELECT * FROM example WHERE c1 = c2;

    ID   C1     C2
----------  -------   --------
     3   FALSE     FALSE
     8   TRUE      TRUE


SELECT * FROM example e1
WHERE c1 >= ALL (SELECT c2 FROM example e2 WHERE e2.id > e1.id);

    ID    C1    C2
----------   -----    -------
     1    TRUE
     7    TRUE      FALSE
     8    TRUE      TRUE
```

Operations on Booleans that Return Booleans

You can use the `NOT`, `AND`, and `OR` operators on SQL conditions, boolean columns, and boolean constants. For example:

```
 SELECT * FROM example WHERE NOT c2;

    ID    C1     C2
----------   ------    ------
     3    FALSE     FALSE
     7    TRUE      FALSE


SELECT * FROM example WHERE c1 AND c2;

    ID   C1      C2
----------  -----  -------
     8   TRUE    TRUE



SELECT * FROM example WHERE c1 AND TRUE;

    ID    C1     C2
----------   -------  -------
     7    TRUE      FALSE
     8    TRUE      TRUE
     1    TRUE


SELECT * FROM example WHERE c1 OR c2;

    ID    C1         C2
----------   -----------  -----
        1    TRUE
```

```
     2    FALSE       TRUE
     4    FALSE       TRUE
     5    FALSE       TRUE
     6    FALSE       TRUE
     7    TRUE        FALSE
     8    TRUE        TRUE

7 rows selected.
```

Boolean Operator `NOT`

The `NOT` (TRUE) is FALSE. `NOT` (FALSE) is true. `NOT` (NULL) is NULL.

Boolean Operator `AND`

Truth Table for the AND Boolean Operator

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	FALSE	FALSE	NULL

Boolean Operator `OR`

Truth Table for the OR Boolean Operator

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Boolean Operator `IS`

Truth Table for the IS Boolean Operator

IS	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE
NULL	FALSE	FALSE	TRUE

Boolean Operator `IS NOT`

Truth Table for the IS NOT Boolean Operator

IS NOT	TRUE	FALSE	NULL
TRUE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE

NULL	TRUE	TRUE	FALSE

In addition to supporting SQL conditions, the `NOT`, `AND`, and `OR` operators support operations on boolean columns and boolean constants. For example, these are all valid statements:

```
SELECT * FROM example WHERE NOT c2;
SELECT * FROM example WHERE c1 AND c2;
SELECT * FROM example WHERE c1 AND TRUE;
SELECT * FROM example WHERE c1 OR c2;
```

You can use `IS [NOT] NULL` on a boolean value expression to determine its state. For example:

```
SELECT * FROM example WHERE c2 IS NULL;

    ID C1           C2
---------- ----------- -----------
     1 TRUE
```

Booleans in SQL Expressions

Boolean expressions are supported in SQL syntax wherever *`expr`* is used.

SQL expressions and conditions have been enhanced to support the new boolean data type. Links to relevant SQL syntax:

- BOOLEAN Expressions

CAST Between Boolean Data Type and Other Oracle Built-In Data Types

The rules to cast between `BOOLEAN` and other Oracle built-in data types are as follows:

When casting `BOOLEAN` to numeric :

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

When casting numeric to `BOOLEAN` :

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

When casting `BOOLEAN` to `CHAR(n)` and `NCHAR(n)`:

- If the boolean value is true and `n` is not less than 4, then the resulting value is 'TRUE' extended on the right by `n` - 4 spaces.
- If the boolean value is false and `n` is not less than 5, then the resulting value is 'FALSE' extended on the right by `n` – 5 spaces.
- Otherwise, a data exception error is raised.

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

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

- If the boolean value is true and `n` is not less than 4, then resulting value is true.
- If the boolean value is false and `n` is not less than 5, then resulting value is false.

- Otherwise, a data exception error is raised.

You can use the function `TO_BOOLEAN` to explicitly convert character value expressions or numeric value expressions to boolean values.

Functions `TO_CHAR`, `TO_NCHAR`, `TO_CLOB`, `TO_NCLOB`, `TO_NUMBER`, `TO_BINARY_DOUBLE`, and `TO_BINARY_FLOAT` have boolean overloads to convert boolean values to number or character types.

> **Note:**
>
> TO_BOOLEAN

Vector Data Type

Vector is a new Oracle built-in data type. This data type represents a vector as an array of numbers, called dimensions stored in one of the following formats:

- `INT8` (8-bit integers)
- `FLOAT32` (32-bit, single precision floating-point numbers)
- `FLOAT64` (64-bit, double precision floating-point numbers)
- `BINARY` (packed `UINT8` bytes where each dimension is a single bit)

`FLOAT32` and `FLOAT64` are IEEE standards.

You can declare a column as vector data type, and optionally specify the dimension count and dimension format.

Syntax Examples:

```
CREATE TABLE t (v VECTOR);
CREATE TABLE t (v VECTOR(*, *));
CREATE TABLE t (v VECTOR(100));
CREATE TABLE t (v VECTOR(100, *));
CREATE TABLE t (v VECTOR(*, FLOAT32));
CREATE TABLE t (v VECTOR(100, FLOAT32));
```

Rules

- If you specify the number of dimensions at declaration, then you must input the same number of dimensions.

  If you do not specify the number of dimensions, then you can input any number of dimensions.
- If you specify the storage format at declaration and the input's format is different from the declared format, it is converted, either up or down, to the declared format.

  If the storage format is not specified, every vector will have its dimensions stored without format modification.
- The number of dimensions must be an integer greater than 0. Note that the number of dimensions must not be 0.
- Vectors are nullable, but dimensions are not (e.g., you cannot have `[1.1, NULL, 2.2]`).

- In an `UNION ALL` statement, if the number of dimensions and the storage format are different between any two branches, then the result vector's number of dimensions and format are flexible.

Declaration Formats for the VECTOR Data Type

The following table lists the possible declaration format for a `VECTOR` data type:

Possible Declaration Format	Explanation
`VECTOR`	Vectors can have an arbitrary number of dimensions and formats.
`VECTOR(*, *)`	Vectors can have an arbitrary number of dimensions and formats. `VECTOR` and `VECTOR(*,*)` are equivalent.
`VECTOR`(number_of_dimensions, *) equivalent to `VECTOR`(number_of_dimensions)	Vectors must all have the specified number of dimensions or an error is thrown. Every vector will have its dimensions stored without format modification.
`VECTOR`(*, dimension_element_format)	Vectors can have an arbitrary number of dimensions, but their format will be up-converted or down-converted to the specified dimension element format (`INT8`, `FLOAT32`, `FLOAT64`,).

A vector can be `NULL` but its dimensions cannot (for example, you cannot have a `VECTOR` with a `NULL` dimension such as `[1.1, NULL, 2.2]`).

The following example shows how the system interprets various vector definitions:

```
CREATE TABLE my_vect_tab (
     v1 VECTOR(3, FLOAT32),
     v2 VECTOR(2, FLOAT64),
     v3 VECTOR(1, INT8),
     v4 VECTOR(1, *),
     v5 VECTOR(*, FLOAT32),
     v6 VECTOR(*, *),
     v7 VECTOR
   );

Table created.

DESC my_vect_tab;
 Name                          Null?    Type
 ----------------------------- -------- ----------------------------
 V1                                     VECTOR(3 , FLOAT32)
 V2                                     VECTOR(2 , FLOAT64)
 V3                                     VECTOR(1 , INT8)
 V4                                     VECTOR(1 , *)
 V5                                     VECTOR(* , FLOAT32)
 V6                                     VECTOR(* , *)
 V7                                     VECTOR(* , *)
```

Restrictions

You cannot define `VECTOR` columns in:

- External Tables
- IOTs (neither as Primary Key nor as non-Key column)
- Clusters or Cluster Tables
- Global Temporary Tables
- MSSM tablespaces (only SYS user can create VECTORs as Basicfiles in MSSM tablespace)
- CQN queries
- Non-vector indexes such as B-tree, Bitmap, Reverse Key, Text, Spatial indexes

You cannot define a `VECTOR` column as a:

- Partitioning or Subpartitioning Key
- Primary Key
- Foreign Key
- Unique Constraint
- Check Constraint
- Default Value
- Modify Column

Oracle Database does not support the following SQL constructs with `VECTOR` columns:

- Distinct, Count Distinct
- Order By, Group By
- Join condition
- Comparison operators (>, <, =)

Create Tables with Column as a VECTOR Data Type

Example 1: Create a table with a column of type vector

The following command creates a table `my_vectors` with two columns: `id` of type `NUMBER` and `embedding` of type `VECTOR`:

```
CREATE TABLE my_vectors (id NUMBER, embedding VECTOR);
```

Example 2: Create a table with a column of type vector and specify dimensions and format

```
CREATE TABLE my_vectors (id NUMBER, embedding VECTOR(768, INT8)) ;
```

In the `my_vectors` table above, each vector that is stored:

- Must have 768 dimensions.
- Each dimension must be formatted as `INT8`.
- The number of dimensions must be strictly greater than zero with no practical upper limit.

There are a new set of SQL functions that use the `VECTOR` data type. See Vector Functions

Rowid Data Types

Each row in the database has an address. The sections that follow describe the two forms of row address in an Oracle Database.

ROWID Data Type

The rows in heap-organized tables that are native to Oracle Database have row addresses called **rowids**. You can examine a rowid row address by querying the pseudocolumn `ROWID`. Values of this pseudocolumn are strings representing the address of each row. These strings have the data type `ROWID`. Refer to Pseudocolumns for more information on the `ROWID` pseudocolumn.

Rowids contain the following information:

- The **data block** of the data file containing the row. The length of this string depends on your operating system.
- The **row** in the data block.
- The **database file** containing the row. The first data file has the number 1. The length of this string depends on your operating system.
- The **data object number**, which is an identification number assigned to every database segment. You can retrieve the data object number from the data dictionary views `USER_OBJECTS`, `DBA_OBJECTS`, and `ALL_OBJECTS`. Objects that share the same segment (clustered tables in the same cluster, for example) have the same object number.

Rowids are stored as base 64 values that can contain the characters A-Z, a-z, 0-9, and the plus sign (+) and forward slash (/). Rowids are not available directly. You can use the supplied package `DBMS_ROWID` to interpret rowid contents. The package functions extract and provide information on the four rowid elements listed above.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information on the functions available with the `DBMS_ROWID` package and how to use them

UROWID Data Type

The rows of some tables have addresses that are not physical or permanent or were not generated by Oracle Database. For example, the row addresses of index-organized tables are stored in index leaves, which can move. Rowids of foreign tables (such as DB2 tables accessed through a gateway) are not standard Oracle rowids.

Oracle uses universal rowids (**urowids**) to store the addresses of index-organized and foreign tables. Index-organized tables have logical urowids and foreign tables have foreign urowids. Both types of urowid are stored in the `ROWID` pseudocolumn (as are the physical rowids of heap-organized tables).

Oracle creates logical rowids based on the primary key of the table. The logical rowids do not change as long as the primary key does not change. The `ROWID` pseudocolumn of an index-organized table has a data type of `UROWID`. You can access this pseudocolumn as you would the `ROWID` pseudocolumn of a heap-organized table (using a `SELECT ... ROWID` statement). If you

want to store the rowids of an index-organized table, then you can define a column of type UROWID for the table and retrieve the value of the ROWID pseudocolumn into that column.

ANSI, DB2, and SQL/DS Data Types

SQL statements that create tables and clusters can also use ANSI data types and data types from the IBM products SQL/DS and DB2. Oracle recognizes the ANSI or IBM data type name that differs from the Oracle Database data type name. It converts the data type to the equivalent Oracle data type, records the Oracle data type as the name of the column data type, and stores the column data in the Oracle data type based on the conversions shown in the tables that follow.

Table 2-7 ANSI Data Types Converted to Oracle Data Types

ANSI SQL Data Type	Oracle Data Type
CHARACTER(n) CHAR(n)	CHAR(n)
CHARACTER VARYING(n) CHAR VARYING(n)	VARCHAR2(n)
NATIONAL CHARACTER(n) NATIONAL CHAR(n) NCHAR(n)	NCHAR(n)
NATIONAL CHARACTER VARYING(n) NATIONAL CHAR VARYING(n) NCHAR VARYING(n)	NVARCHAR2(n)
NUMERIC[(p,s)] DECIMAL[(p,s)] **(Note 1)**	NUMBER(p,s)
INTEGER INT SMALLINT	NUMBER(38)
FLOAT **(Note 2)**	FLOAT(126)
DOUBLE PRECISION **(Note 3)**	FLOAT(126)
REAL **(Note 4)**	FLOAT(63)

Notes:

1. The NUMERIC and DECIMAL data types can specify only fixed-point numbers. For those data types, the scale (s) defaults to 0.
2. The FLOAT data type is a floating-point number with a binary precision b. The default precision for this data type is 126 binary, or 38 decimal.
3. The DOUBLE PRECISION data type is a floating-point number with binary precision 126.
4. The REAL data type is a floating-point number with a binary precision of 63, or 18 decimal.

Do not define columns with the following SQL/DS and DB2 data types, because they have no corresponding Oracle data type:

- GRAPHIC
- LONG VARGRAPHIC

- VARGRAPHIC
- TIME

Note that data of type TIME can also be expressed as Oracle datetime data.

> **See Also:**
>
> Datetime and Interval Data Types

Table 2-8 SQL/DS and DB2 Data Types Converted to Oracle Data Types

SQL/DS or DB2 Data Type	Oracle Data Type
CHARACTER(n)	CHAR(n)
VARCHAR(n)	VARCHAR(n)
LONG VARCHAR	LONG
DECIMAL(p,s) (**Note 1**)	NUMBER(p,s)
INTEGER SMALLINT	NUMBER(p,0)
FLOAT (**Note 2**)	NUMBER

Notes:

1. The DECIMAL data type can specify only fixed-point numbers. For this data type, *s* defaults to 0.
2. The FLOAT data type is a floating-point number with a binary precision *b*. The default precision for this data type is 126 binary or 38 decimal.

User-Defined Types

User-defined data types use Oracle built-in data types and other user-defined data types as the building blocks of object types that model the structure and behavior of data in applications. The sections that follow describe the various categories of user-defined types.

> **See Also:**
>
> - *Oracle Database Concepts* for information about Oracle built-in data types
> - CREATE TYPE and the CREATE TYPE BODY for information about creating user-defined types
> - *Oracle Database Object-Relational Developer's Guide* for information about using user-defined types

Object Types

Object types are abstractions of the real-world entities, such as purchase orders, that application programs deal with. An object type is a schema object with three kinds of components:

- A **name**, which identifies the object type uniquely within that schema.
- **Attributes**, which are built-in types or other user-defined types. Attributes model the structure of the real-world entity.
- **Methods**, which are functions or procedures written in PL/SQL and stored in the database, or written in a language like C or Java and stored externally. Methods implement operations the application can perform on the real-world entity.

REF Data Types

An **object identifier** (represented by the keyword `OID`) uniquely identifies an object and enables you to reference the object from other objects or from relational tables. A data type category called `REF` represents such references. A `REF` data type is a container for an object identifier. `REF` values are pointers to objects.

When a `REF` value points to a nonexistent object, the `REF` is said to be "dangling". A dangling `REF` is different from a null `REF`. To determine whether a `REF` is dangling or not, use the condition `IS [NOT] DANGLING`. For example, given object view `oc_orders` in the sample schema `oe`, the column `customer_ref` is of type `REF` to type `customer_typ`, which has an attribute `cust_email`:

```
SELECT o.customer_ref.cust_email
  FROM oc_orders o
  WHERE o.customer_ref IS NOT DANGLING;
```

Varrays

An array is an ordered set of data elements. All elements of a given array are of the same data type. Each element has an **index**, which is a number corresponding to the position of the element in the array.

The number of elements in an array is the size of the array. Oracle arrays are of variable size, which is why they are called **varrays**. You must specify a maximum size when you declare the varray.

When you declare a varray, it does not allocate space. It defines a type, which you can use as:

- The data type of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type

Oracle normally stores an array object either in line (as part of the row data) or out of line (in a LOB), depending on its size. However, if you specify separate storage characteristics for a varray, then Oracle stores it out of line, regardless of its size. Refer to the *varray_col_properties* of CREATE TABLE for more information about varray storage.

Nested Tables

A nested table type models an unordered set of elements. The elements may be built-in types or user-defined types. You can view a nested table as a single-column table or, if the nested

table is an object type, as a multicolumn table, with a column for each attribute of the object type.

A nested table definition does not allocate space. It defines a type, which you can use to declare:

- The data type of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type

When a nested table appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table.

Oracle-Supplied Types

Oracle provides SQL-based interfaces for defining new types when the built-in or ANSI-supported types are not sufficient. The behavior for these types can be implemented in C/C++, Java, or PL/ SQL. Oracle Database automatically provides the low-level infrastructure services needed for input-output, heterogeneous client-side access for new data types, and optimizations for data transfers between the application and the database.

These interfaces can be used to build user-defined (or object) types and are also used by Oracle to create some commonly useful data types. Several such data types are supplied with the server, and they serve both broad horizontal application areas (for example, the `Any` types) and specific vertical ones (for example, the spatial types).

The Oracle-supplied types, along with cross-references to the documentation of their implementation and use, are described in the following sections:

- Any Types
- XML Types
- Spatial Types

Any Types

The `Any` types provide highly flexible modeling of procedure parameters and table columns where the actual type is not known. These data types let you dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type. These types have OCI and PL/SQL interfaces for construction and access.

ANYTYPE

This type can contain a type description of any named SQL type or unnamed transient type.

ANYDATA

This type contains an instance of a given type, with data, plus a description of the type. `ANYDATA` can be used as a table column data type and lets you store heterogeneous values in a single column. The values can be of SQL built-in types as well as user-defined types.

ANYDATASET

This type contains a description of a given type plus a set of data instances of that type. `ANYDATASET` can be used as a procedure parameter data type where such flexibility is needed. The values of the data instances can be of SQL built-in types as well as user-defined types.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information on the `ANYTYPE`, `ANYDATA`, and `ANYDATASET` types

XML Types

Extensible Markup Language (XML) is a standard format developed by the World Wide Web Consortium (W3C) for representing structured and unstructured data on the World Wide Web. Universal resource identifiers (URIs) identify resources such as Web pages anywhere on the Web. Oracle provides types to handle XML and URI data, as well as a class of URIs called `DBURIRef` types to access data stored within the database itself. It also provides a set of types to store and access both external and internal URIs from within the database.

XMLType

This Oracle-supplied type can be used to store and query XML data in the database. `XMLType` has member functions you can use to access, extract, and query the XML data using XPath expressions. XPath is another standard developed by the W3C committee to traverse XML documents. Oracle `XMLType` functions support many W3C XPath expressions. Oracle also provides a set of SQL functions and PL/SQL packages to create `XMLType` values from existing relational or object-relational data.

`XMLType` is a system-defined type, so you can use it as an argument of a function or as the data type of a table or view column. You can also create tables and views of `XMLType`. When you create an `XMLType` column in a table, you can choose to store the XML data in a `CLOB` column, as binary XML (stored internally as a `BLOB`), or object relationally.

You can also register the schema (using the `DBMS_XMLSCHEMA` package) and create a table or column conforming to the registered schema. In this case Oracle stores the XML data in underlying object-relational columns by default, but you can specify storage in a `CLOB` or binary XML column even for schema-based data.

Queries and DML on `XMLType` columns operate the same regardless of the storage mechanism.

> **See Also:**
>
> *Oracle XML DB Developer's Guide*for information about using XMLType columns

URI Data Types

Oracle supplies a family of URI types—URIType, DBURIType, XDBURIType, and HTTPURIType—which are related by an inheritance hierarchy. URIType is an object type and the others are subtypes of URIType. Since URIType is the supertype, you can create columns of this type and store DBURIType or HTTPURIType type instances in this column.

HTTPURIType

You can use HTTPURIType to store URLs to external Web pages or to files. Oracle accesses these files using HTTP (Hypertext Transfer Protocol).

XDBURIType

You can use XDBURIType to expose documents in the XML database hierarchy as URIs that can be embedded in any URIType column in a table. The XDBURIType consists of a URL, which comprises the hierarchical name of the XML document to which it refers and an optional fragment representing the XPath syntax. The fragment is separated from the URL part by a pound sign (#). The following lines are examples of XDBURIType:

```
/home/oe/doc1.xml
/home/oe/doc1.xml#/orders/order_item
```

DBURIType

DBURIType can be used to store DBURIRef values, which reference data inside the database. Storing DBURIRef values lets you reference data stored inside or outside the database and access the data consistently.

DBURIRef values use an XPath-like representation to reference data inside the database. If you imagine the database as an XML tree, then you would see the tables, rows, and columns as elements in the XML document. For example, the sample human resources user hr would see the following XML tree:

```
<HR>
  <EMPLOYEES>
    <ROW>
      <EMPLOYEE_ID>205</EMPLOYEE_ID>
      <LAST_NAME>Higgins</LAST_NAME>
      <SALARY>12008</SALARY>
      .. <!-- other columns -->
    </ROW>
    ... <!-- other rows -->
  </EMPLOYEES>
  <!-- other tables..-->
</HR>
<!-- other user schemas on which you have some privilege on..-->
```

The DBURIRef is an XPath expression over this virtual XML document. So to reference the SALARY value in the EMPLOYEES table for the employee with employee number 205, you can write a DBURIRef as,

```
/HR/EMPLOYEES/ROW[EMPLOYEE_ID=205]/SALARY
```

Using this model, you can reference data stored in CLOB columns or other columns and expose them as URLs to the external world.

URIFactory Package

Oracle also provides the `URIFactory` package, which can create and return instances of the various subtypes of the `URITypes`. The package analyzes the URL string, identifies the type of URL (HTTP, `DBURI`, and so on), and creates an instance of the subtype. To create a `DBURI` instance, the URL must begin with the prefix `/oradb`. For example, `URIFactory.getURI('/oradb/HR/EMPLOYEES')` would create a `DBURIType` instance and `URIFactory.getUri('/sys/schema')` would create an `XDBURIType` instance.

> **See Also:**
>
> - *Oracle Database Object-Relational Developer's Guide* for general information on object types and type inheritance
> - *Oracle XML DB Developer's Guide* for more information about these supplied types and their implementation
> - *Oracle Database Advanced Queuing User's Guide* for information about using `XMLType` with Oracle Advanced Queuing

Spatial Types

Oracle Spatial and Graph is designed to make spatial data management easier and more natural to users of location-enabled applications, geographic information system (GIS) applications, and geoimaging applications. After the spatial data is stored in an Oracle Database, you can easily manipulate, retrieve, and relate it to all the other data stored in the database. The following data types are available only if you have installed Oracle Spatial and Graph.

SDO_GEOMETRY

The geometric description of a spatial object is stored in a single row, in a single column of object type `SDO_GEOMETRY` in a user-defined table. Any table that has a column of type `SDO_GEOMETRY` must have another column, or set of columns, that defines a unique primary key for that table. Tables of this sort are sometimes called geometry tables.

The `SDO_GEOMETRY` object type has the following definition:

```
CREATE TYPE SDO_GEOMETRY AS OBJECT
  (sgo_gtype        NUMBER,
   sdo_srid         NUMBER,
   sdo_point        SDO_POINT_TYPE,
   sdo_elem_info    SDO_ELEM_INFO_ARRAY,
   sdo_ordinates    SDO_ORDINATE_ARRAY);
/
```

SDO_TOPO_GEOMETRY

This type describes a topology geometry, which is stored in a single row, in a single column of object type `SDO_TOPO_GEOMETRY` in a user-defined table.

The `SDO_TOPO_GEOMETRY` object type has the following definition:

```
CREATE TYPE SDO_TOPO_GEOMETRY AS OBJECT
  (tg_type        NUMBER,
   tg_id          NUMBER,
   tg_layer_id    NUMBER,
   topology_id    NUMBER);
/
```

SDO_GEORASTER

In the GeoRaster object-relational model, a raster grid or image object is stored in a single row, in a single column of object type `SDO_GEORASTER` in a user-defined table. Tables of this sort are called GeoRaster tables.

The `SDO_GEORASTER` object type has the following definition:

```
CREATE TYPE SDO_GEORASTER AS OBJECT
  (rasterType         NUMBER,
   spatialExtent      SDO_GEOMETRY,
   rasterDataTable    VARCHAR2(32),
   rasterID           NUMBER,
   metadata           XMLType);
/
```

> **See Also:**
>
> *Oracle Spatial Developer's Guide*, *Oracle Spatial Topology and Network Data Model Developer's Guide*, and *Oracle Spatial GeoRaster Developer's Guide* for information on the full implementation of the spatial data types and guidelines for using them

Data Type Comparison Rules

This section describes how Oracle Database compares values of each data type.

Numeric Values

A larger value is considered greater than a smaller one. All negative numbers are less than zero and all positive numbers. Thus, -1 is less than 100; -100 is less than -1.

The floating-point value `NaN` (not a number) is greater than any other numeric value and is equal to itself.

> **See Also:**
>
> Numeric Precedence and Floating-Point Numbers for more information on comparison semantics

Datetime Values

A later date or timestamp is considered greater than an earlier one. For example, the date equivalent of '29-MAR-2005' is less than that of '05-JAN-2006' and the timestamp equivalent of '05-JAN-2006 1:35pm' is greater than that of '05-JAN-2005 10:09am'.

When two timestamps with time zone are compared, they are first normalized to UTC, that is, to the timezone offset '+00:00'. For example, the timestamp with time zone equivalent of '16-OCT-2016 05:59am Europe/Warsaw' is equal to that of '15-OCT-2016 08:59pm US/Pacific'. Both represent the same absolute point in time, which represented in UTC is October 16th, 2016, 03:59am.

Binary Values

A binary value of the data type `RAW` or `BLOB` is a sequence of bytes. When two binary values are compared, the corresponding, consecutive bytes of the two byte sequences are compared in turn. If the first bytes of both compared values are different, the binary value that contains the byte with the lower numeric value is considered smaller. If the first bytes are equal, second bytes are compared analogously, and so on, until either the compared bytes differ or the comparison process reaches the end of one of the values. In the latter case, the value that is shorter is considered smaller.

Binary values of the data type `BLOB` cannot be compared directly in comparison conditions. However, they can be compared with the PL/SQL function `DBMS_LOB.COMPARE`.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information on the `DBMS_LOB.COMPARE` function

Character Values

Character values are compared on the basis of two measures:

- Binary or linguistic collation
- Blank-padded or nonpadded comparison semantics

The following subsections describe the two measures.

Binary and Linguistic Collation

In binary collation, which is the default, Oracle compares character values like binary values. Two sequences of bytes that form the encodings of two character values in their storage character set are treated as binary values and compared as described in Binary Values . The result of this comparison is returned as the result of the binary comparison of the source character values.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information on character sets

For many languages, the binary collation can yield a linguistically incorrect ordering of character values. For example, in most common character sets, all the uppercase Latin letters

have character codes with lower values than all the lowercase Latin letters. Hence, the binary collation yields the following order:

```
MacDonald
MacIntosh
Macdonald
Macintosh
```

However, most users expect these four values to be presented in the order:

```
MacDonald
Macdonald
MacIntosh
Macintosh
```

This shows that binary collation may not be suitable even for English character values.

Oracle Database supports linguistic collations that order strings according to rules of various spoken languages. It also supports collation variants that match character values case- and accent-insensitively. Linguistic collations are more expensive but they provide superior user experience.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information about linguistic sorting

Restrictions for Linguistic Collations

Comparison conditions, `ORDER BY`, `GROUP BY` and `MATCH_RECOGNIZE` query clauses, `COUNT(DISTINCT)` and statistical aggregate functions, `LIKE` conditions, and `ORDER BY` and `PARTITION BY` analytic clauses generate collation keys when using linguistic collations. The collation keys are the same values that are returned by the function `NLSSORT` and are subject to the same restrictions that are described in NLSSORT .

Blank-Padded and Nonpadded Comparison Semantics

With blank-padded semantics, if the two values have different lengths, then Oracle first adds blanks to the end of the shorter one so their lengths are equal. Oracle then compares the values character by character up to the first character that differs. The value with the greater character in the first differing position is considered greater. If two values have no differing characters, then they are considered equal. This rule means that two values are equal if they differ only in the number of trailing blanks. Oracle uses blank-padded comparison semantics only when both values in the comparison are either expressions of data type `CHAR`, `NCHAR`, text literals, or values returned by the `USER` function.

With nonpadded semantics, Oracle compares two values character by character up to the first character that differs. The value with the greater character in that position is considered greater. If two values of different length are identical up to the end of the shorter one, then the longer value is considered greater. If two values of equal length have no differing characters, then the values are considered equal. Oracle uses nonpadded comparison semantics whenever one or both values in the comparison have the data type `VARCHAR2` or `NVARCHAR2`.

The results of comparing two character values using different comparison semantics may vary. The table that follows shows the results of comparing five pairs of character values using each comparison semantic. Usually, the results of blank-padded and nonpadded comparisons are the same. The last comparison in the table illustrates the differences between the blank-padded and nonpadded comparison semantics.

Blank-Padded	Nonpadded
`'ac' > 'ab'`	`'ac' > 'ab'`
`'ab' > 'a '`	`'ab' > 'a '`
`'ab' > 'a'`	`'ab' > 'a'`
`'ab' = 'ab'`	`'ab' = 'ab'`
`'a ' = 'a'`	**`'a ' > 'a'`**

Data-Bound Collation

Starting with Oracle Database 12*c* Release 2 (12.2), the collation to use when comparing or matching a given character value is associated with the value itself. It is called the **data-bound collation**. The data-bound collation can be viewed as an attribute of the data type of the value.

In previous Oracle Database releases, the session parameters `NLS_COMP` and `NLS_SORT` coarsely determined the collation for all collation-sensitive SQL operations in a database session. The data-bound collation architecture enables applications to consistently apply language-specific comparison rules to exactly the data that needs these rules.

Oracle Database 12*c* Release 2 (12.2) allows you to declare a collation for a table column. When a column is passed as an argument to a collation-sensitive SQL operation, the SQL operation uses the column's declared collation to process the column's values. If the SQL operation has multiple character arguments that are compared to each other, the **collation determination rules** determine the collation to use.

There are two types of data-bound collations:

- Named Collation: This collation is a particular set of collating rules specified by a collation name. Named collations are the same collations that are specified as values for the `NLS_SORT` parameter. A named collation can be either a binary collation or a linguistic collation.
- Pseudo-collation: This collation does not directly specify the collating rules for a SQL operation. Instead, it instructs the operation to check the values of the session parameters `NLS_SORT` and `NLS_COMP` for the actual named collation to use. Pseudo-collations are the bridge between the new declarative method of specifying collations and the old method that uses session parameters. In particular, the pseudo-collation `USING_NLS_COMP` directs a SQL operation to behave exactly as it used to behave before Oracle Database 12*c* Release 2.

When you declare a named collation for a column, you statically determine how the column values are compared. When you declare a pseudo-collation, you can dynamically control comparison behavior with the session parameter `NLS_COMP` and `NLS_SORT`. However, static objects, such as indexes and constraints, defined on a column declared with a pseudo-collation, fall back to using a binary collation. Dynamically settable collating rules cannot be used to compare values for a static object.

The collation for a character literal or bind variable that is used in an expression is derived from the default collation of the database object containing the expression, such as a view or materialized view query, a PL/SQL stored unit code, a user-defined type method code, or a standalone DML or query statement. In Oracle Database 12*c* Release 2, the default collation of

PL/SQL stored units, user-defined type methods, and standalone SQL statements is always the pseudo-collation `USING_NLS_COMP`. The default collation of views and materialized views can be specified in the `DEFAULT COLLATION` clause of the `CREATE VIEW` and `CREATE MATERIALIZED VIEW` statements.

If a SQL operation returns character values, the **collation derivation rules** determine the **derived collation** for the result, so that its collation is known, when the result is passed as an argument to another collation-sensitive SQL operation in the expression tree or to a top-level consumer, such as an SQL statement clause in a `SELECT` statement. If a SQL operation operates on character argument values, then the derived collation of its character result is based on the collations of the arguments. Otherwise, the derivation rules are the same as for a character literal.

You can override the derived collation of an expression node, such as a simple expression or an operator result, by using the `COLLATE` operator.

Oracle Database allows you to declare a case-insensitive collation for a column, table or schema, so that the column or all character columns in a table or a schema can be always compared in a case-insensitive way.

See Also:

- *Oracle Database Globalization Support Guide* for more information on data-bound collation architecture, including the detailed collation derivation and determination rules
- COLLATE Operator

Object Values

Object values are compared using one of two comparison functions: `MAP` and `ORDER`. Both functions compare object type instances, but they are quite different from one another. These functions must be specified as part of any object type that will be compared with other object types.

See Also:

CREATE TYPE for a description of `MAP` and `ORDER` methods and the values they return

Varrays and Nested Tables

Comparison of nested tables is described in Comparison Conditions .

Data Type Precedence

Oracle uses data type precedence to determine implicit data type conversion, which is discussed in the section that follows. Oracle data types take the following precedence:

- Datetime and interval data types

- BINARY_DOUBLE
- BINARY_FLOAT
- NUMBER
- Character data types
- All other built-in data types

Data Conversion

Generally an expression cannot contain values of different data types. For example, an expression cannot multiply 5 by 10 and then add 'JAMES'. However, Oracle supports both implicit and explicit conversion of values from one data type to another.

Implicit and Explicit Data Conversion

Oracle recommends that you specify explicit conversions, rather than rely on implicit or automatic conversions, for these reasons:

- SQL statements are easier to understand when you use explicit data type conversion functions.
- Implicit data type conversion can have a negative impact on performance, especially if the data type of a column value is converted to that of a constant rather than the other way around.
- Implicit conversion depends on the context in which it occurs and may not work the same way in every case. For example, implicit conversion from a datetime value to a VARCHAR2 value may return an unexpected year depending on the value of the NLS_DATE_FORMAT parameter.
- Algorithms for implicit conversion are subject to change across software releases and among Oracle products. Behavior of explicit conversions is more predictable.
- If implicit data type conversion occurs in an index expression, then Oracle Database might not use the index because it is defined for the pre-conversion data type. This can have a negative impact on performance.

Implicit Data Conversion

Oracle Database automatically converts a value from one data type to another when such a conversion makes sense.

Table 2-9 is a matrix of Oracle implicit conversions. The table shows all possible conversions, without regard to the direction of the conversion or the context in which it is made.

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

Table 2-9 Implicit Type Conversion Matrix

Data Type	CHAR	VARCHAR2	NCHAR	NVARCHAR2	DATE	DATETIME/INTERVAL	NUMBER	BINARY_FLOAT	BINARY_DOUBLE	LONG	RAW	ROWID	CLOB	BLOB	NCLOB	BOOLEAN
CHAR	--	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
VARCHAR2	X	--	X	X	X	X	X	X	X	X	X	X	X	--	X	--
NCHAR	X	X	--	X	X	X	X	X	X	X	X	X	X	--	X	X
NVARCHAR2	X	X	X	--	X	X	X	X	X	X	X	X	X	--	X	--
DATE	X	X	X	X	--	--	--	--	--	--	--	--	--	--	--	--
DATETIME/INTERVAL	X	X	X	X	--	--	--	--	--	X	--	--	--	--	--	--
NUMBER	X	X	X	X	--	--	--	X	X	--	--	--	--	--	--	X
BINARY_FLOAT	X	X	X	X	--	--	X	--	X	--	--	--	--	--	--	X
BINARY_DOUBLE	X	X	X	X	--	--	X	X	--	--	--	--	--	--	--	X
LONG	X	X	X	X	--	X[1]	--	--	--	--	X	--	X	--	X	--
RAW	X	X	X	X	--	--	--	--	--	X	--	--	--	X	--	--
ROWID	X	X	X	X	--	--	--	--	--	--	--	--	--	--	--	--
CLOB	X	X	X	X	--	--	--	--	-	X	--	--	--	--	X	--
BLOB	--	--	--	--	--	--	--	--	-	--	X	--	--	--	--	--
NCLOB	X	X	X	X	--	--	--	--	-	X	--	--	X	--	--	--
JSON	--	X	--	--	--	--	--	--	--	--	--	--	X	X	--	--
BOOLEAN	X	X	X	X	--	--	X	X	X	--	--	--	--	--	--	--

[1] You cannot convert `LONG` to `INTERVAL` directly, but you can convert `LONG` to `VARCHAR2` using `TO_CHAR(`*`interval`*`)`, and then convert the resulting `VARCHAR2` value to `INTERVAL`.

Implicit Data Type Conversion Rules

- During `INSERT` and `UPDATE` operations, Oracle converts the value to the data type of the affected column.
- During `SELECT FROM` operations, Oracle converts the data from the column to the type of the target variable.
- When manipulating numeric values, Oracle usually adjusts precision and scale to allow for maximum capacity. In such cases, the numeric data type resulting from such operations can differ from the numeric data type found in the underlying tables.

- When comparing a character value with a numeric value, Oracle converts the character data to a numeric value.
- Conversions between character values or NUMBER values and floating-point number values can be inexact, because the character types and NUMBER use decimal precision to represent the numeric value, and the floating-point numbers use binary precision.
- When converting a CLOB value into a character data type such as VARCHAR2, or converting BLOB to RAW data, if the data to be converted is larger than the target data type, then the database returns an error.
- During conversion from a timestamp value to a DATE value, the fractional seconds portion of the timestamp value is truncated. This behavior differs from earlier releases of Oracle Database, when the fractional seconds portion of the timestamp value was rounded.
- Conversions from BINARY_FLOAT to BINARY_DOUBLE are exact.
- Conversions from BINARY_DOUBLE to BINARY_FLOAT are inexact if the BINARY_DOUBLE value uses more bits of precision that supported by the BINARY_FLOAT.
- When comparing a character value with a DATE value, Oracle converts the character data to DATE.
- When you use a SQL function or operator with an argument of a data type other than the one it accepts, Oracle converts the argument to the accepted data type.
- When making assignments, Oracle converts the value on the right side of the equal sign (=) to the data type of the target of the assignment on the left side.
- During concatenation operations, Oracle converts from noncharacter data types to CHAR or NCHAR.
- During arithmetic operations on and comparisons between character and noncharacter data types, Oracle converts from any character data type to a numeric, date, or rowid, as appropriate. In arithmetic operations between CHAR/VARCHAR2 and NCHAR/NVARCHAR2, Oracle converts to a NUMBER.
- Most SQL character functions are enabled to accept CLOBs as parameters, and Oracle performs implicit conversions between CLOB and character types. Therefore, functions that are not yet enabled for CLOBs can accept CLOBs through implicit conversion. In such cases, Oracle converts the CLOBs to CHAR or VARCHAR2 before the function is invoked. If the CLOB is larger than 4000 bytes, then Oracle converts only the first 4000 bytes to CHAR.
- When converting RAW or LONG RAW data to or from character data, the binary data is represented in hexadecimal form, with one hexadecimal character representing every four bits of RAW data. Refer to "RAW and LONG RAW Data Types " for more information.
- Comparisons between CHAR and VARCHAR2 and between NCHAR and NVARCHAR2 types may entail different character sets. The default direction of conversion in such cases is from the database character set to the national character set. Table 2-10 shows the direction of implicit conversions between different character types.

Table 2-10 Conversion Direction of Different Character Types

Source Data Type	to CHAR	to VARCHAR2	to NCHAR	to NVARCHAR2
from CHAR	-	VARCHAR2	NCHAR	NVARCHAR2
from VARCHAR2	VARCHAR2	-	NVARCHAR2	NVARCHAR2
from NCHAR	NCHAR	NCHAR	-	NVARCHAR2

Table 2-10 (Cont.) Conversion Direction of Different Character Types

Source Data Type	to CHAR	to VARCHAR2	to NCHAR	to NVARCHAR2
from NVARCHAR2	NVARCHAR2	NVARCHAR2	NVARCHAR2	-

User-defined types such as collections cannot be implicitly converted, but must be explicitly converted using `CAST ... MULTISET`.

Implicit Data Conversion Examples

Text Literal Example

The text literal '10' has data type `CHAR`. Oracle implicitly converts it to the `NUMBER` data type if it appears in a numeric expression as in the following statement:

```
SELECT salary + '10'
  FROM employees;
```

Character and Number Values Example

When a condition compares a character value and a `NUMBER` value, Oracle implicitly converts the character value to a `NUMBER` value, rather than converting the `NUMBER` value to a character value. In the following statement, Oracle implicitly converts '200' to 200:

```
SELECT last_name
  FROM employees
  WHERE employee_id = '200';
```

Date Example

In the following statement, Oracle implicitly converts '`24-JUN-06`' to a `DATE` value using the default date format '`DD-MON-YY`':

```
SELECT last_name
  FROM employees
  WHERE hire_date = '24-JUN-06';
```

Explicit Data Conversion

You can explicitly specify data type conversions using SQL conversion functions. Table 2-11 shows SQL functions that explicitly convert a value from one data type to another.

You cannot specify `LONG` and `LONG RAW` values in cases in which Oracle can perform implicit data type conversion. For example, `LONG` and `LONG RAW` values cannot appear in expressions with functions or operators. Refer to LONG Data Type for information on the limitations on `LONG` and `LONG RAW` data types.

Table 2-11 Explicit Type Conversions

Source Data Type	to CHAR, VARCHAR2, NCHAR, NVARCHAR2	to NUMBER	to Datetime/ Interval	to RAW	to ROWID	to LONG, LONG RAW	to CLOB, NCLOB, BLOB	to BINARY_FLOAT	to BINARY_DOUBLE	to BOOLEAN
from CHAR, VARCHAR2, NCHAR, NVARCHAR2	TO_CHAR (char.) TO_NCHAR (char.)	TO_NUMBER	TO_DATE TO_TIMESTAMP TO_TIMESTAMP_TZ TO_YMINTERVAL TO_DSINTERVAL	HEXTORAW	CHARTO=ROWID	--	TO_CLOB TO_NCLOB	TO_BINARY_FLOAT	TO_BINARY_DOUBLE	TO_BOOLEAN
from NUMBER	TO_CHAR (number) TO_NCHAR (number)	--	TO_DATE NUMTOYM-INTERVAL NUMTODS-INTERVAL	--	--	--	--	TO_BINARY_FLOAT	TO_BINARY_DOUBLE	TO_BOOLEAN
from Datetime/ Interval	TO_CHAR (date) TO_NCHAR (datetime)	--	--	--	--	--	--	--	--	--
from RAW	RAWTOHEX RAWTONHEX	--	--	--	--	--	TO_BLOB	--	--	--
from ROWID	ROWIDTOCHAR	--	--	--	--	--	--	--	--	--
from LONG / LONG RAW	--	--	--	--	--	--	TO_LOB	--	--	--
from CLOB, NCLOB, BLOB	TO_CHAR TO_NCHAR	--	--	--	--	--	TO_CLOB TO_NCLOB	--	--	--
from CLOB, NCLOB, BLOB	TO_CHAR TO_NCHAR	--	--	--	--	--	TO_CLOB TO_NCLOB	--	--	--
from BINARY_FLOAT	TO_CHAR (char.) TO_NCHAR (char.)	TO_NUMBER	--	--	--	--	--	TO_BINARY_FLOAT	TO_BINARY_DOUBLE	TO_BOOLEAN

Table 2-11 (Cont.) Explicit Type Conversions

Source Data Type	to CHAR, VARCHAR2, NCHAR, NVARCHAR2	to NUMBER	to Datetime/ Interval	to RAW	to ROWID	to LONG, LONG RAW	to CLOB, NCLOB, BLOB	to BINARY_FLOAT	to BINARY_DOUBLE	to BOOLEAN
from BINARY_DOUBLE	TO_CHAR (char.) TO_NCHAR (char.)	TO_NUMBER	--	--	--	--	--	TO_BINARY_FLOAT	TO_BINARY_DOUBLE	TO_BOOLEAN
from BOOLEAN	TO_CHAR (boolean) TO_NCHAR (boolean)	TO_NUMBER	--	--	--	--	--	TO_BINARY_FLOAT	TO_BINARY_DOUBLE	TO_BOOLEAN

> **See Also:**
>
> Conversion Functions for details on all of the explicit conversion functions

Security Considerations for Data Conversion

When a datetime value is converted to text, either by implicit conversion or by explicit conversion that does not specify a format model, the format model is defined by one of the globalization session parameters. Depending on the source data type, the parameter name is `NLS_DATE_FORMAT`, `NLS_TIMESTAMP_FORMAT`, or `NLS_TIMESTAMP_TZ_FORMAT`. The values of these parameters can be specified in the client environment or in an `ALTER SESSION` statement.

The dependency of format models on session parameters can have a negative impact on database security when conversion without an explicit format model is applied to a datetime value that is being concatenated to text of a dynamic SQL statement. Dynamic SQL statements are those statements whose text is concatenated from fragments before being passed to a database for execution. Dynamic SQL is frequently associated with the built-in PL/SQL package `DBMS_SQL` or with the PL/SQL statement `EXECUTE IMMEDIATE`, but these are not the only places where dynamically constructed SQL text may be passed as argument. For example:

```
EXECUTE IMMEDIATE
'SELECT last_name FROM employees WHERE hire_date > ''' || start_date || '''';
```

where *start_date* has the data type `DATE`.

In the above example, the value of *start_date* is converted to text using a format model specified in the session parameter `NLS_DATE_FORMAT`. The result is concatenated into SQL text. A datetime format model can consist simply of literal text enclosed in double quotation marks. Therefore, any user who can explicitly set globalization parameters for a session can decide what text is produced by the above conversion. If the SQL statement is executed by a PL/SQL procedure, the procedure becomes vulnerable to SQL injection through the session parameter.

If the procedure runs with definer's rights, with higher privileges than the session itself, the user can gain unauthorized access to sensitive data.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for further examples and for recommendations on avoiding this security risk

> **Note:**
>
> This security risk also applies to middle-tier applications that construct SQL text from datetime values converted to text by the database or by OCI datetime functions. Those applications are vulnerable if session globalization parameters are obtained from a user preference.

Implicit and explicit conversion for numeric values may also suffer from the analogous problem, as the conversion result may depend on the session parameter `NLS_NUMERIC_CHARACTERS`. This parameter defines the decimal and group separator characters. If the decimal separator is defined to be the quotation mark or the double quotation mark, some potential for SQL injection emerges.

> **See Also:**
>
> - *Oracle Database Globalization Support Guide* for detailed descriptions of the session globalization parameters
> - Format Models for information on the format models

Literals

The terms **literal** and **constant value** are synonymous and refer to a fixed data value. For example, 'JACK', 'BLUE ISLAND', and '101' are all character literals; 5001 is a numeric literal. Character literals are enclosed in single quotation marks so that Oracle can distinguish them from schema object names.

This section contains these topics:

- Text Literals
- Numeric Literals
- Datetime Literals
- Interval Literals

Many SQL statements and functions require you to specify character and numeric literal values. You can also specify literals as part of expressions and conditions. You can specify character literals with the '*text*' notation, national character literals with the *N'text'* notation, and numeric literals with the *integer*, or *number* notation, depending on the context of the literal. The syntactic forms of these notations appear in the sections that follow.

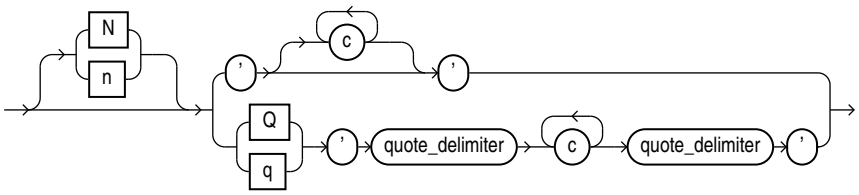
To specify a datetime or interval data type as a literal, you must take into account any optional precisions included in the data types. Examples of specifying datetime and interval data types as literals are provided in the relevant sections of Data Types .

Text Literals

Use the text literal notation to specify values whenever *string* appears in the syntax of expressions, conditions, SQL functions, and SQL statements in other parts of this reference. This reference uses the terms **text literal**, **character literal**, and **string** interchangeably. Text, character, and string literals are always surrounded by single quotation marks. If the syntax uses the term *char*, then you can specify either a text literal or another expression that resolves to character data — for example, the `last_name` column of the `hr.employees` table. When *char* appears in the syntax, the single quotation marks are not used.

The syntax of text literals or strings follows:

***string*::=**



where `N` or `n` specifies the literal using the national character set (`NCHAR` or `NVARCHAR2` data). By default, text entered using this notation is translated into the national character set by way of the database character set when used by the server. To avoid potential loss of data during the text literal conversion to the database character set, set the environment variable `ORA_NCHAR_LITERAL_REPLACE` to `TRUE`. Doing so transparently replaces the `n'` internally and preserves the text literal for SQL processing.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more information about N-quoted literals

In the top branch of the syntax:

- *c* is any member of the user's character set. A single quotation mark (') within the literal must be preceded by an escape character. To represent one single quotation mark within a literal, enter two single quotation marks.
- `''` are two single quotation marks that begin and end text literals.

In the bottom branch of the syntax:

- `Q` or `q` indicates that the alternative quoting mechanism will be used. This mechanism allows a wide range of delimiters for the text string.
- The outermost `' '` are two single quotation marks that precede and follow, respectively, the opening and closing *quote_delimiter*.

- *c* is any member of the user's character set. You can include quotation marks (") in the text literal made up of *c* characters. You can also include the *quote_delimiter*, as long as it is not immediately followed by a single quotation mark.
- *quote_delimiter* is any single- or multibyte character except space, tab, and return. The *quote_delimiter* can be a single quotation mark. However, if the *quote_delimiter* appears in the text literal itself, ensure that it is not immediately followed by a single quotation mark.

  If the opening *quote_delimiter* is one of [, {, <, or (, then the closing *quote_delimiter* must be the corresponding], }, >, or). In all other cases, the opening and closing *quote_delimiter* must be the same character.

Text literals have properties of both the CHAR and VARCHAR2 data types:

- Within expressions and conditions, Oracle treats text literals as though they have the data type CHAR by comparing them using blank-padded comparison semantics.
- A text literal can have a maximum length of 4000 bytes if the initialization parameter MAX_STRING_SIZE = STANDARD, and 32767 bytes if MAX_STRING_SIZE = EXTENDED. See Extended Data Types for more information.

Here are some valid text literals:

```
'Hello'
'ORACLE.dbs'
'Jackie''s raincoat'
'09-MAR-98'
N'nchar literal'
```

Here are some valid text literals using the alternative quoting mechanism:

```
q'!name LIKE '%DBMS_%%'!'
q'<'So,' she said, 'It's finished.'>'
q'{SELECT * FROM employees WHERE last_name = 'Smith';}'
nq'ï Ÿ1234 ï'
q'"name like '['"'
```

> **See Also:**
>
> Blank-Padded and Nonpadded Comparison Semantics

Numeric Literals

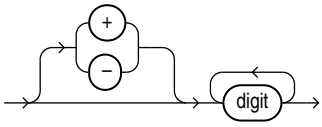
Use numeric literal notation to specify fixed and floating-point numbers.

Integer Literals

You must use the integer notation to specify an integer whenever *integer* appears in expressions, conditions, SQL functions, and SQL statements described in other parts of this reference.

The syntax of *integer* follows:

***integer*::=**



where *digit* is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

An integer can store a maximum of 38 digits of precision.

Here are some valid integers:

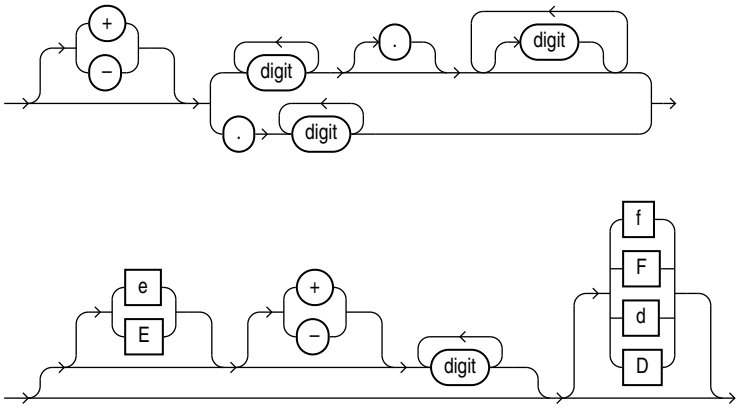
```
7
+255
```

NUMBER and Floating-Point Literals

You must use the number or floating-point notation to specify values whenever *number* or *n* appears in expressions, conditions, SQL functions, and SQL statements in other parts of this reference.

The syntax of *number* follows:

***number*::=**



where

- + or - indicates a positive or negative value. If you omit the sign, then a positive value is the default.
- *digit* is one of 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9.
- e or E indicates that the number is specified in scientific notation. The digits after the E specify the exponent. The exponent can range from -130 to 125.
- f or F indicates that the number is a 32-bit binary floating point number of type `BINARY_FLOAT`.
- d or D indicates that the number is a 64-bit binary floating point number of type `BINARY_DOUBLE`.

If you omit f or F and d or D, then the number is of type `NUMBER`.

The suffixes f (F) and d (D) are supported only in floating-point number literals, not in character strings that are to be converted to `NUMBER`. For example, if Oracle is expecting a `NUMBER` and it encounters the string `'9'`, then it converts the string to the number 9. However, if Oracle encounters the string `'9f'`, then conversion fails and an error is returned.

A number of type `NUMBER` can store a maximum of 38 digits of precision. If the literal requires more precision than provided by `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`, then Oracle truncates the value. If the range of the literal exceeds the range supported by `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE`, then Oracle raises an error.

Numeric literals are SQL syntax elements, which are not sensitive to NLS settings. The decimal separator character in numeric literals is always the period (.). However, if a text literal is specified where a numeric value is expected, then the text literal is implicitly converted to a number in an NLS-sensitive way. The decimal separator contained in the text literal must be the one established with the initialization parameter `NLS_NUMERIC_CHARACTERS`. Oracle recommends that you use numeric literals in SQL scripts to make them work independently of the NLS environment.

The following examples illustrate the behavior of decimal separators in numeric literals and text literals. These examples assume that you have established the comma (,) as the NLS decimal separator for the current session with the following statement:

```
ALTER SESSION SET NLS_NUMERIC_CHARACTERS=',.';
```

The previous statement also establishes the period (.) as the NLS group separator, but that is irrelevant for these examples.

This example uses the required decimal separator (.) in the numeric literal `1.23` and the established NLS decimal separator (,) in the text literal `'2,34'`. The text literal is converted to the numeric value 2.34, and the output is displayed using commas for the decimal separators.

```
SELECT 2 * 1.23, 3 * '2,34' FROM DUAL;

    2*1.23   3*'2,34'
---------- ----------
      2,46       7,02
```

The next example shows that a comma is not treated as part of a numeric literal. Rather, the comma is treated as the delimiter in a list of two numeric expressions: `2*1` and `23`.

```
SELECT 2 * 1,23 FROM DUAL;

       2*1         23
---------- ----------
         2         23
```

The next example shows that the decimal separator in a text literal must match the NLS decimal separator in order for implicit text-to-number conversion to succeed. The following statement fails because the decimal separator (.) does not match the established NLS decimal separator (,):

```
SELECT 3 * '2.34' FROM DUAL;
           *
ERROR at line 1:
ORA-01722: invalid number
```

> **See Also:**
>
> ALTER SESSION and *Oracle Database Reference*

Here are some valid `NUMBER` literals:

```
25
+6.34
0.5
25e-03
-1
```

Here are some valid floating-point number literals:

```
25f
+6.34F
0.5d
-1D
```

You can also use the following supplied floating-point literals in situations where a value cannot be expressed as a numeric literal:

Table 2-12 Floating-Point Literals

Literal	Meaning	Example
`binary_float_nan`	A value of type `BINARY_FLOAT` for which the condition `IS NAN` is true	`SELECT COUNT(*) FROM employees WHERE TO_BINARY_FLOAT(commission_pct) != BINARY_FLOAT_NAN;`
`binary_float_infinity`	Single-precision positive infinity	`SELECT COUNT(*) FROM employees WHERE salary < BINARY_FLOAT_INFINITY;`
`binary_double_nan`	A value of type `BINARY_DOUBLE` for which the condition `IS NAN` is true	`SELECT COUNT(*) FROM employees WHERE TO_BINARY_FLOAT(commission_pct) != BINARY_FLOAT_NAN;`
`binary_double_infinity`	Double-precision positive infinity	`SELECT COUNT(*) FROM employees WHERE salary < BINARY_DOUBLE_INFINITY;`

Datetime Literals

Oracle Database supports four datetime data types: `DATE`, `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, and `TIMESTAMP WITH LOCAL TIME ZONE`.

Date Literals

You can specify a DATE value as a string literal, or you can convert a character or numeric value to a date value with the TO_DATE function. DATE literals are the only case in which Oracle Database accepts a TO_DATE expression in place of a string literal.

To specify a DATE value as a literal, you must use the Gregorian calendar. You can specify an ANSI literal, as shown in this example:

```
DATE '1998-12-25'
```

The ANSI date literal contains no time portion, and must be specified in the format 'YYYY-MM-DD'. Alternatively you can specify an Oracle date value, as in the following example:

```
TO_DATE('98-DEC-25 17:30','YY-MON-DD HH24:MI')
```

The default date format for an Oracle DATE value is specified by the initialization parameter NLS_DATE_FORMAT. This example date format includes a two-digit number for the day of the month, an abbreviation of the month name, the last two digits of the year, and a 24-hour time designation.

Oracle automatically converts character values that are in the default date format into date values when they are used in date expressions.

If you specify a date value without a time component, then the default time is midnight (00:00:00 or 12:00:00 for 24-hour and 12-hour clock time, respectively). If you specify a date value without a date, then the default date is the first day of the current month.

Oracle DATE columns always contain both the date and time fields. Therefore, if you query a DATE column, then you must either specify the time field in your query or ensure that the time fields in the DATE column are set to midnight. Otherwise, Oracle may not return the query results you expect. You can use the TRUNC date function to set the time field to midnight, or you can include a greater-than or less-than condition in the query instead of an equality or inequality condition.

Here are some examples that assume a table my_table with a number column row_num and a DATE column datecol:

```
INSERT INTO my_table VALUES (1, SYSDATE);
INSERT INTO my_table VALUES (2, TRUNC(SYSDATE));

SELECT *
  FROM my_table;

   ROW_NUM DATECOL
---------- ---------
         1 03-OCT-02
         2 03-OCT-02

SELECT *
  FROM my_table
  WHERE datecol > TO_DATE('02-OCT-02', 'DD-MON-YY');

   ROW_NUM DATECOL
---------- ---------
         1 03-OCT-02
         2 03-OCT-02

SELECT *
  FROM my_table
```

```
  WHERE datecol = TO_DATE('03-OCT-02','DD-MON-YY');

   ROW_NUM DATECOL
---------- ---------
         2 03-OCT-02
```

If you know that the time fields of your `DATE` column are set to midnight, then you can query your `DATE` column as shown in the immediately preceding example, or by using the `DATE` literal:

```
SELECT *
  FROM my_table
  WHERE datecol = DATE '2002-10-03';


   ROW_NUM DATECOL
---------- ---------
         2 03-OCT-02
```

However, if the `DATE` column contains values other than midnight, then you must filter out the time fields in the query to get the correct result. For example:

```
SELECT *
  FROM my_table
  WHERE TRUNC(datecol) = DATE '2002-10-03';


   ROW_NUM DATECOL
---------- ---------
         1 03-OCT-02
         2 03-OCT-02
```

Oracle applies the `TRUNC` function to each row in the query, so performance is better if you ensure the midnight value of the time fields in your data. To ensure that the time fields are set to midnight, use one of the following methods during inserts and updates:

- Use the `TO_DATE` function to mask out the time fields:

  ```
  INSERT INTO my_table
    VALUES (3, TO_DATE('3-OCT-2002','DD-MON-YYYY'));
  ```

- Use the `DATE` literal:

  ```
  INSERT INTO my_table
    VALUES (4, '03-OCT-02');
  ```

- Use the `TRUNC` function:

  ```
  INSERT INTO my_table
    VALUES (5, TRUNC(SYSDATE));
  ```

The date function `SYSDATE` returns the current system date and time. The function `CURRENT_DATE` returns the current session date. For information on `SYSDATE`, the `TO_*` datetime functions, and the default date format, see Datetime Functions .

TIMESTAMP Literals

The `TIMESTAMP` data type stores year, month, day, hour, minute, and second, and fractional second values. When you specify `TIMESTAMP` as a literal, the *`fractional_seconds_precision`* value can be any number of digits up to 9, as follows:

```
TIMESTAMP '1997-01-31 09:26:50.124'
```

TIMESTAMP WITH TIME ZONE Literals

The `TIMESTAMP WITH TIME ZONE` data type is a variant of `TIMESTAMP` that includes a time zone region name or time zone offset. When you specify `TIMESTAMP WITH TIME ZONE` as a literal, the *`fractional_seconds_precision`* value can be any number of digits up to 9. For example:

```
TIMESTAMP '1997-01-31 09:26:56.66 +02:00'
```

Two `TIMESTAMP WITH TIME ZONE` values are considered identical if they represent the same instant in UTC, regardless of the `TIME ZONE` offsets stored in the data. For example,

```
TIMESTAMP '1999-04-15 8:00:00 -8:00'
```

is the same as

```
TIMESTAMP '1999-04-15 11:00:00 -5:00'
```

8:00 a.m. Pacific Standard Time is the same as 11:00 a.m. Eastern Standard Time.

You can replace the UTC offset with the `TZR` (time zone region name) format element. For example, the following example has the same value as the preceding example:

```
TIMESTAMP '1999-04-15 8:00:00 US/Pacific'
```

To eliminate the ambiguity of boundary cases when the daylight saving time switches, use both the `TZR` and a corresponding `TZD` format element. The following example ensures that the preceding example will return a daylight saving time value:

```
TIMESTAMP '1999-10-29 01:30:00 US/Pacific PDT'
```

You can also express the time zone offset using a datetime expression:

```
SELECT TIMESTAMP '2009-10-29 01:30:00' AT TIME ZONE 'US/Pacific'
  FROM DUAL;
```

> **See Also:**
>
> Datetime Expressions for more information

If you do not add the `TZD` format element, and the datetime value is ambiguous, then Oracle returns an error if you have the `ERROR_ON_OVERLAP_TIME` session parameter set to `TRUE`. If that parameter is set to `FALSE`, then Oracle interprets the ambiguous datetime as standard time in the specified region.

TIMESTAMP WITH LOCAL TIME ZONE Literals

The `TIMESTAMP WITH LOCAL TIME ZONE` data type differs from `TIMESTAMP WITH TIME ZONE` in that data stored in the database is normalized to the database time zone. The time zone offset is not stored as part of the column data. There is no literal for `TIMESTAMP WITH LOCAL TIME ZONE`. Rather, you represent values of this data type using any of the other valid datetime literals. The table that follows shows some of the formats you can use to insert a value into a `TIMESTAMP WITH LOCAL TIME ZONE` column, along with the corresponding value returned by a query.

Table 2-13 TIMESTAMP WITH LOCAL TIME ZONE Literals

Value Specified in INSERT Statement	Value Returned by Query
`'19-FEB-2004'`	19-FEB-2004.00.00.000000 AM
`SYSTIMESTAMP`	19-FEB-04 02.54.36.497659 PM
`TO_TIMESTAMP('19-FEB-2004', 'DD-MON-YYYY')`	19-FEB-04 12.00.00.000000 AM
`SYSDATE`	19-FEB-04 02.55.29.000000 PM
`TO_DATE('19-FEB-2004', 'DD-MON-YYYY')`	19-FEB-04 12.00.00.000000 AM
`TIMESTAMP'2004-02-19 8:00:00 US/Pacific'`	19-FEB-04 08.00.00.000000 AM

Notice that if the value specified does not include a time component (either explicitly or implicitly), then the value returned defaults to midnight.

Interval Literals

An interval literal specifies a period of time. You can specify these differences in terms of years and months, or in terms of days, hours, minutes, and seconds. Oracle Database supports two types of interval literals, `YEAR TO MONTH` and `DAY TO SECOND`. Each type contains a leading field and may contain a trailing field. The leading field defines the basic unit of date or time being measured. The trailing field defines the smallest increment of the basic unit being considered. For example, a `YEAR TO MONTH` interval considers an interval of years to the nearest month. A `DAY TO MINUTE` interval considers an interval of days to the nearest minute.

If you have date data in numeric form, then you can use the `NUMTOYMINTERVAL` or `NUMTODSINTERVAL` conversion function to convert the numeric data into interval values.

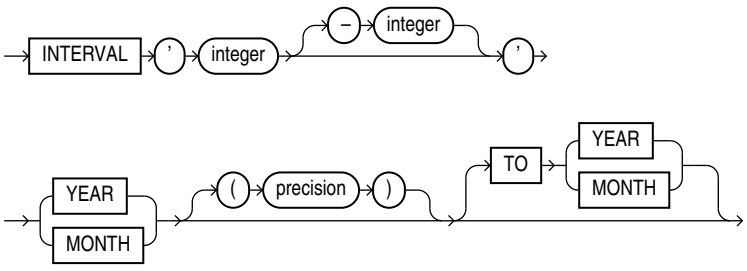
Interval literals are used primarily with analytic functions.

> **See Also:**
>
> Analytic Functions , NUMTODSINTERVAL , and NUMTOYMINTERVAL

INTERVAL YEAR TO MONTH

Specify `YEAR TO MONTH` interval literals using the following syntax:

***interval_year_to_month*::=**



where

- *'integer [-integer]'* specifies integer values for the leading and optional trailing field of the literal. If the leading field is YEAR and the trailing field is MONTH, then the range of integer values for the month field is 0 to 11.
- *precision* is the maximum number of digits in the leading field. The valid range of the leading field precision is 0 to 9 and its default value is 2.

Restriction on the Leading Field

If you specify a trailing field, then it must be less significant than the leading field. For example, INTERVAL '0-1' MONTH TO YEAR is not valid.

The following INTERVAL YEAR TO MONTH literal indicates an interval of 123 years, 2 months:

```
INTERVAL '123-2' YEAR(3) TO MONTH
```

Examples of the other forms of the literal follow, including some abbreviated versions:

Table 2-14 Forms of INTERVAL YEAR TO MONTH Literals

Form of Interval Literal	Interpretation
INTERVAL '123-2' YEAR(3) TO MONTH	An interval of 123 years, 2 months. You must specify the leading field precision if it is greater than the default of 2 digits.
INTERVAL '123' YEAR(3)	An interval of 123 years 0 months.
INTERVAL '300' MONTH(3)	An interval of 300 months.
INTERVAL '4' YEAR	Maps to INTERVAL '4-0' YEAR TO MONTH and indicates 4 years.
INTERVAL '50' MONTH	Maps to INTERVAL '4-2' YEAR TO MONTH and indicates 50 months or 4 years 2 months.
INTERVAL '123' YEAR	Returns an error, because the default precision is 2, and '123' has 3 digits.

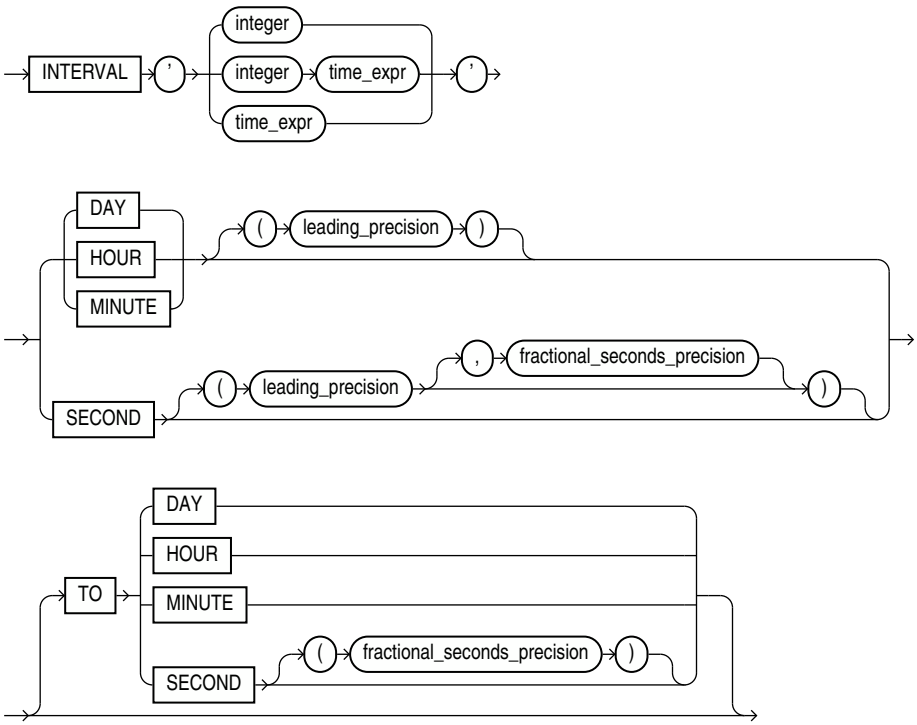
You can add or subtract one INTERVAL YEAR TO MONTH literal to or from another to yield another INTERVAL YEAR TO MONTH literal. For example:

```
INTERVAL '5-3' YEAR TO MONTH + INTERVAL'20' MONTH =
INTERVAL '6-11' YEAR TO MONTH
```

INTERVAL DAY TO SECOND

Specify DAY TO SECOND interval literals using the following syntax:

***interval_day_to_second*::=**



where

- *integer* specifies the number of days. If this value contains more digits than the number specified by the leading precision, then Oracle returns an error.
- *time_expr* specifies a time in the format *HH*[:*MI*[:*SS*[.*n*]]] or *MI*[:*SS*[.*n*]] or *SS*[.*n*], where *n* specifies the fractional part of a second. If *n* contains more digits than the number specified by *fractional_seconds_precision*, then *n* is rounded to the number of digits specified by the *fractional_seconds_precision* value. You can specify *time_expr* following an integer and a space only if the leading field is DAY.
- *leading_precision* is the number of digits in the leading field. Accepted values are 0 to 9. The default is 2.
- *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 1 to 9. The default is 6.

Restriction on the Leading Field:

If you specify a trailing field, then it must be less significant than the leading field. For example, INTERVAL MINUTE TO DAY is not valid. As a result of this restriction, if SECOND is the leading field, the interval literal cannot have any trailing field.

The valid range of values for the trailing field are as follows:

- HOUR: 0 to 23
- MINUTE: 0 to 59
- SECOND: 0 to 59.999999999

Examples of the various forms of INTERVAL DAY TO SECOND literals follow, including some abbreviated versions:

Table 2-15 Forms of INTERVAL DAY TO SECOND Literals

Form of Interval Literal	Interpretation
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)	4 days, 5 hours, 12 minutes, 10 seconds, and 222 thousandths of a second.
INTERVAL '4 5:12' DAY TO MINUTE	4 days, 5 hours and 12 minutes.
INTERVAL '400 5' DAY(3) TO HOUR	400 days 5 hours.
INTERVAL '400' DAY(3)	400 days.
INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)	11 hours, 12 minutes, and 10.2222222 seconds.
INTERVAL '11:20' HOUR TO MINUTE	11 hours and 20 minutes.
INTERVAL '10' HOUR	10 hours.
INTERVAL '10:22' MINUTE TO SECOND	10 minutes 22 seconds.
INTERVAL '10' MINUTE	10 minutes.
INTERVAL '4' DAY	4 days.
INTERVAL '25' HOUR	25 hours.
INTERVAL '40' MINUTE	40 minutes.
INTERVAL '120' HOUR(3)	120 hours.
INTERVAL '30.12345' SECOND(2,4)	30.1235 seconds. The fractional second '12345' is rounded to '1235' because the precision is 4.

You can add or subtract one DAY TO SECOND interval literal from another DAY TO SECOND literal. For example.

```
INTERVAL'20' DAY - INTERVAL'240' HOUR = INTERVAL'10-0' DAY TO SECOND
```

Format Models

A **format model** is a character literal that describes the format of datetime or numeric data stored in a character string. A format model does not change the internal representation of the value in the database. When you convert a character string into a date or number, a format model determines how Oracle Database interprets the string. In SQL statements, you can use a format model as an argument of the TO_CHAR and TO_DATE functions to specify:

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

For example:

- The datetime format model for the string '17:45:29' is 'HH24:MI:SS'.
- The datetime format model for the string '11-Nov-1999' is 'DD-Mon-YYYY'.
- The number format model for the string '$2,304.25' is '$9,999.99'.

For lists of number and datetime format model elements, see Table 2-16 and Table 2-18.

The values of some formats are determined by the value of initialization parameters. For such formats, you can specify the characters returned by these format elements implicitly using the

initialization parameter NLS_TERRITORY. You can change the default date format for your session with the ALTER SESSION statement.

> **See Also:**
>
> - ALTER SESSION for information on changing the values of these parameters and Format Model Examples for examples of using format models
> - TO_CHAR (datetime) , TO_CHAR (number) , and TO_DATE
> - *Oracle Database Reference* and *Oracle Database Globalization Support Guide* for information on these parameters

This remainder of this section describes how to use the following format models:

- Number Format Models
- Datetime Format Models
- Format Model Modifiers

Number Format Models

You can use number format models in the following functions:

- In the TO_CHAR function to translate a value of NUMBER, BINARY_FLOAT, or BINARY_DOUBLE data type to VARCHAR2 data type
- In the TO_NUMBER function to translate a value of CHAR or VARCHAR2 data type to NUMBER data type
- In the TO_BINARY_FLOAT and TO_BINARY_DOUBLE functions to translate CHAR and VARCHAR2 expressions to BINARY_FLOAT or BINARY_DOUBLE values

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, then pound signs (#) replace the value. This event typically occurs when you are using TO_CHAR with a restrictive number format string, causing a rounding operation.

- If a positive NUMBER value is extremely large and cannot be represented in the specified format, then the infinity sign (~) replaces the value. Likewise, if a negative NUMBER value is extremely small and cannot be represented by the specified format, then the negative infinity sign replaces the value (-~).
- If a BINARY_FLOAT or BINARY_DOUBLE value is converted to CHAR or NCHAR, and the input is either infinity or NaN (not a number), then Oracle always returns the pound signs to replace the value. However, if you omit the format model, then Oracle returns either Inf or Nan as a string.

Number Format Elements

A number format model is composed of one or more number format elements. The tables that follow list the elements of a number format model and provide some examples.

Negative return values automatically contain a leading negative sign and positive values automatically contain a leading space unless the format model contains the `MI`, `S`, or `PR` format element.

Table 2-16 Number Format Elements

Element	Example	Description
, (comma)	`9,999`	Returns a comma in the specified position. You can specify multiple commas in a number format model. **Restrictions:** • A comma element cannot begin a number format model. • A comma cannot appear to the right of a decimal character or period in a number format model.
. (period)	`99.99`	Returns a decimal point, which is a period (.) in the specified position. **Restriction:** You can specify only one period in a number format model.
$	`$9999`	Returns value with a leading dollar sign.
0	`0999` `9990`	Returns leading zeros. Returns trailing zeros.
9	`9999`	Returns value with the specified number of digits with a leading space if positive or 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.
B	`B9999`	Returns blanks for the integer part of a fixed-point number when the integer part is zero (regardless of zeros in the format model).
C	`C999`	Returns in the specified position the ISO currency symbol (the current value of the `NLS_ISO_CURRENCY` parameter).
D	`99D99`	Returns in the specified position the decimal character, which is the current value of the `NLS_NUMERIC_CHARACTER` parameter. The default is a period (.). **Restriction:** You can specify only one decimal character in a number format model.
EEEE	`9.9EEEE`	Returns a value using in scientific notation.
G	`9G999`	Returns in the specified position the group separator (the current value of the `NLS_NUMERIC_CHARACTER` parameter). You can specify multiple group separators in a number format model. **Restriction:** A group separator cannot appear to the right of a decimal character or period in a number format model.
L	`L999`	Returns in the specified position the local currency symbol (the current value of the `NLS_CURRENCY` parameter).
MI	`9999MI`	Returns negative value with a trailing minus sign (-). Returns positive value with a trailing blank. **Restriction:** The MI format element can appear only in the last position of a number format model.
PR	`9999PR`	Returns negative value in <angle brackets>. Returns positive value with a leading and trailing blank. **Restriction:** The PR format element can appear only in the last position of a number format model.
RN rn	`RN` `rn`	Returns a value as Roman numerals in uppercase. Returns a value as Roman numerals in lowercase. Value can be an integer between 1 and 3999.

Table 2-16 (Cont.) Number Format Elements

Element	Example	Description
S	`S9999` `9999S`	Returns negative value with a leading minus sign (-). Returns positive value with a leading plus sign (+). Returns negative value with a trailing minus sign (-). Returns positive value with a trailing plus sign (+). **Restriction:** The S format element can appear only in the first or last position of a number format model.
TM	`TM`	The text minimum number format model returns (in decimal output) the smallest number of characters possible. This element is case insensitive. The default is TM9, which returns the number in fixed notation unless the output exceeds 64 characters. If the output exceeds 64 characters, then Oracle Database automatically returns the number in scientific notation. **Restrictions:** • You cannot precede this element with any other element. • You can follow this element only with one 9 or one E (or e), but not with any combination of these. The following statement returns an error: `SELECT TO_CHAR(1234, 'TM9e') FROM DUAL;`
U	`U9999`	Returns in the specified position the Euro (or other) dual currency symbol, determined by the current value of the `NLS_DUAL_CURRENCY` parameter.
V	`999V99`	Returns a value multiplied by 10^n (and if necessary, round it up), where *n* is the number of 9's after the `V`.
X	`XXXX` `xxxx`	Returns the hexadecimal value of the specified number of digits. If the specified number is not an integer, then Oracle Database rounds it to an integer. **Restrictions:** • This element accepts only positive values or 0. Negative values return an error. • You can precede this element only with 0 (which returns leading zeroes) or FM. Any other elements return an error. If you specify neither 0 nor FM with X, then the return always has one leading blank. Refer to the format model modifier FM for more information.

Table 2-17 shows the results of the following query for different values of *number* and *'fmt'*:

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

Table 2-17 Results of Number Conversions

number	'fmt'	Result
-1234567890	9999999999S	'1234567890-'
0	99.99	' .00'
+0.1	99.99	' .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'

Table 2-17 (Cont.) Results of Number Conversions

number	'fmt'	Result
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	9.9EEEE	' 1.2E+02'
+1E+123	9.9EEEE	' 1.0E+123'
+123.456	FM9.9EEEE	'1.2E+02'
+123.45	FM999.009	'123.45'
+123.0	FM999.009	'123.00'
+123.45	L999.99	' $123.45'
+123.45	FML999.99	'$123.45'
+1234567890	9999999999S	'1234567890+'

Datetime Format Models

You can use datetime format models in the following functions:

- In the `TO_*` datetime functions to translate a character value that is in a format other than the default format into a datetime value. (The `TO_*` datetime functions are `TO_DATE`, `TO_TIMESTAMP`, and `TO_TIMESTAMP_TZ`.)
- In the `TO_CHAR` function to translate a datetime value into a character value that is in a format other than the default format (for example, to print the date from an application)

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

The default datetime formats are specified either explicitly with the NLS session parameters `NLS_DATE_FORMAT`, `NLS_TIMESTAMP_FORMAT`, and `NLS_TIMESTAMP_TZ_FORMAT`, or implicitly with the NLS session parameter `NLS_TERRITORY`. You can change the default datetime formats for your session with the `ALTER SESSION` statement.

> **See Also:**
>
> ALTER SESSION and *Oracle Database Globalization Support Guide* for information on the NLS parameters

Datetime Format Elements

A datetime format model is composed of one or more datetime format elements as listed in Table 2-18.

- 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.
- The second column indicates whether the format element can be used in the `TO_*` datetime functions. All format elements can be used in the `TO_CHAR` function.
- The following datetime format elements can be used in timestamp and interval format models, but not in the original `DATE` format model: `FF`, `TZD`, `TZH`, `TZM`, and `TZR`.
- Many datetime format elements are padded with blanks or leading zeroes to a specific length. Refer to the format model modifier FM for more information.

> **Note:**
>
> Oracle recommends that you use the 4-digit year element (`YYYY`) instead of the shorter year elements for these reasons:
>
> - The 4-digit year element eliminates ambiguity.
> - The shorter year elements may affect query optimization because the year is not known at query compile time and can only be determined at run time.

Uppercase Letters in 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 Datetime Format Models

You can 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.

Table 2-18 Datetime Format Elements

Element	TO_* datetime functions?	Description

`-`
`/`
`,`
`.`
`;`
`:`
`"text"`
	Yes	Punctuation and quoted text is reproduced in the result

`AD` `A.D.`	Yes	Gregorian calendar era indicator with or without periods [1]

Table 2-18 (Cont.) Datetime Format Elements

Element	TO_* datetime functions?	Description
AM A.M.	Yes	Meridian indicator with or without periods 1
BC B.C.	Yes	Gregorian calendar era indicator with or without periods 1
CC SCC		Century • If the last 2 digits of a 4-digit year are between 01 and 99 (inclusive), then the century is one greater than the first 2 digits of that year. • If the last 2 digits of a 4-digit year are 00, then the century is the same as the first 2 digits of that year. For example: 2002 returns 21, 2000 returns 20.
D	Yes	Day of week (1-7). This element depends on the NLS territory of the session.
DAY	Yes	Name of day
DD	Yes	Day of month (1-31)
DDD	Yes	Day of year (1-366)
DL	Yes	Returns a value in the long date format, which is an extension of the Oracle Database DATE format, determined by the current value of the NLS_DATE_FORMAT parameter. Makes the appearance of the date components (day name, month number, and so forth) depend on the NLS_TERRITORY and NLS_LANGUAGE parameters. For example, in the AMERICAN_AMERICA locale, this is equivalent to specifying the format 'fmDay, Month dd, yyyy'. In the GERMAN_GERMANY locale, it is equivalent to specifying the format 'fmDay, dd. Month yyyy'. **Restriction:** You can specify this format only with the TS element, separated by white space.
DS	Yes	Returns a value in the short date format. Makes the appearance of the date components (day name, month number, and so forth) depend on the NLS_TERRITORY and NLS_LANGUAGE parameters. For example, in the AMERICAN_AMERICA locale, this is equivalent to specifying the format 'MM/DD/RRRR'. In the ENGLISH_UNITED_KINGDOM locale, it is equivalent to specifying the format 'DD/MM/RRRR'. **Restriction:** You can specify this format only with the TS element, separated by white space.
DY	Yes	Abbreviated name of day
E	Yes	Abbreviated era name (Japanese Imperial, ROC Official, and Thai Buddha calendars)
EE	Yes	Full era name (Japanese Imperial, ROC Official, and Thai Buddha calendars)

Table 2-18 (Cont.) Datetime Format Elements

Element	TO_* datetime functions?	Description
`FF [1..9]`	Yes	Fractional seconds; no decimal character is printed. Use the X format element to add the decimal character. Use the numbers 1 to 9 after FF to specify the number of digits in the fractional second portion of the datetime value returned. If you do not specify a digit, then Oracle Database uses the precision specified for the datetime data type or the data type's default precision. Valid in timestamp formats, but not in `DATE` formats. **Examples:** `'HH:MI:SS.FF'` `SELECT TO_CHAR(SYSTIMESTAMP, 'SS.FF3') from DUAL;`
`FM`	Yes	Returns a value with no leading or trailing blanks. **See Also**: FM
`FX`	Yes	Requires exact matching between the character data and the format model. **See Also**: FX
`HH` `HH12`	Yes	Hour of day (1-12)
`HH24`	Yes	Hour of day (0-23)
`IW`		Calendar week of year (1-52 or 1-53), as defined by the ISO 8601 standard: • A calendar week starts on Monday • The first calendar week of the year includes January 4 • The first calendar week of the year may include December 29, 30 and 31 • The last calendar week of the year may include January 1, 2, and 3
`IYYY`		4-digit year of the year containing the calendar week, as defined by the ISO 8601 standard

`IYY`
`IY`
`I`
		Last 3, 2, or 1 digit(s) of the year containing the calendar week, as defined by the ISO 8601 standard

`J`	Yes	Julian day: the number of days since January 1, 4712 BC. The number specified with J must be an integer.
`MI`	Yes	Minute (0-59)
`MM`	Yes	Month (01-12; January = 01)
`MON`	Yes	Abbreviated name of month
`MONTH`	Yes	Name of month
`PM` `P.M.`	Yes	Meridian indicator with or without periods [1]
`Q`		Quarter of year (1, 2, 3, 4; January - March = 1)

Table 2-18 (Cont.) Datetime Format Elements

Element	TO_* datetime functions?	Description
RM	Yes	Roman numeral month (I-XII; January = I)
RR	Yes	Lets you store 20th century dates in the 21st century using only two digits. **See Also:** The RR Datetime Format Element
RRRR	Yes	Round year. Accepts either 4-digit or 2-digit input. If 2-digit, provides the same return as RR. If you do not want this functionality, enter 4-digit year.
SS	Yes	Second (0-59)
SSSSS	Yes	Seconds past midnight (0-86399)
TS	Yes	Returns a value in the short time format. Makes the appearance of the time components (hour, minutes, and so forth) depend on the NLS_TERRITORY and NLS_LANGUAGE initialization parameters. **Restriction:** You can specify this format only with the DL or DS element, separated by white space.
TZD	Yes	Daylight saving information. The TZD value is an abbreviated time zone string with daylight saving information. It must correspond with the region specified in TZR. Valid in timestamp with time zone format models only. **Example:** PST (for US/Pacific standard time); PDT (for US/Pacific daylight time).
TZH	Yes	Time zone hour. (See TZM format element.) Valid in timestamp with time zone format models only. **Example:** 'HH:MI:SS.FFTZH:TZM'.
TZM	Yes	Time zone minute. (See TZH format element.) Valid in timestamp with time zone format models only. **Example:** 'HH:MI:SS.FFTZH:TZM'.
TZR	Yes	Time zone region information. On input, the value must be one of the time zone region names supported in the database. If the format exact mode is not active, the value may also be a time zone offset in the form [+-]hours:minutes. Valid in timestamp with time zone format models only. **Example:** US/Pacific
WW		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		Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh day of the month
X	Yes	Local decimal character **Example:** 'HH:MI:SSXFF'
Y,YYY	Yes	Year with comma in this position
YEAR SYEAR		Year, spelled out S prefixes BC dates with a minus sign (-)

Table 2-18 (Cont.) Datetime Format Elements

Element	TO_* datetime functions?	Description
YYYY SYYYY	Yes	4-digit year S prefixes BC dates with a minus sign
YYY YY Y	Yes	Last 3, 2, or 1 digit(s) of year

[1] If the NLS_DATE_LANGUAGE parameter is AMERICAN, the format model elements AD, BC, AM, and PM output or expect the corresponding indicators without periods. The format model elements A.D., B.C., A.M., and P.M. output or expect the corresponding indicators with periods. If the NLS_DATE_FORMAT parameter is not AMERICAN, the format model elements AD, BC, AM, and PM with and without periods are equivalent, and output or expect indicators that are defined for the given language in Oracle locale data. You can view this language-specific indicator text in the Oracle Locale Builder utility.

Oracle Database converts strings to dates with some flexibility. For example, when the TO_DATE function is used, a format model containing punctuation characters matches an input string lacking some or all of these characters, provided each numerical element in the input string contains the maximum allowed number of digits—for example, two digits '05' for 'MM' or four digits '2007' for 'YYYY'. The following statement does not return an error:

```
SELECT TO_CHAR(TO_DATE('0207','MM/YY'), 'MM/YY') FROM DUAL;

TO_CH
-----
02/07
```

However, the following format string does return an error, because the FX (format exact) format modifier requires an exact match of the expression and the format string:

```
SELECT TO_CHAR(TO_DATE('0207', 'fxmm/yy'), 'mm/yy') FROM DUAL;
SELECT TO_CHAR(TO_DATE('0207', 'fxmm/yy'), 'mm/yy') FROM DUAL;
                       *
ERROR at line 1:
ORA-01861: literal does not match format string
```

Any non-alphanumeric character is allowed to match the punctuation characters in the format model. For example, the following statement does not return an error:

```
SELECT TO_CHAR (TO_DATE('02#07','MM/YY'), 'MM/YY') FROM DUAL;

TO_CH
-----
02/07
```

See Also:

Format Model Modifiers and String-to-Date Conversion Rules for more information

Datetime Format Elements and Globalization Support

The functionality of some datetime format elements depends on the country and language in which you are using Oracle Database. For example, these datetime 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` datetime format elements are always in English.

The datetime 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`.

> **See Also:**
>
> *Oracle Database Reference* and *Oracle Database Globalization Support Guide* for information on globalization support initialization parameters

ISO Standard Date Format Elements

Oracle calculates the values returned by the datetime 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 datetime format elements YYYY, YYY, YY, Y, and WW, see the discussion of globalization support in *Oracle Database Globalization Support Guide*.

The RR Datetime Format Element

The `RR` datetime format element is similar to the `YY` datetime format element, but it provides additional flexibility for storing date values in other centuries. The `RR` datetime format element lets you store 20th century dates in the 21st century by specifying only the last two digits of the year.

If you use the `TO_DATE` function with the `YY` datetime format element, then the year returned always has the same first 2 digits as the current year. If you use the `RR` datetime format element instead, then the century of the return value varies according to the specified two-digit year and the last two digits of the current year.

That is:

- If the specified two-digit year is 00 to 49, then
  - If the last two digits of the current year are 00 to 49, then the returned year has the same first two digits as the current year.

  - If the last two digits of the current year are 50 to 99, then the first 2 digits of the returned year are 1 greater than the first 2 digits of the current year.
- If the specified two-digit year is 50 to 99, then
  - If the last two digits of the current year are 00 to 49, then the first 2 digits of the returned year are 1 less than the first 2 digits of the current year.
  - If the last two digits of the current year are 50 to 99, then the returned year has the same first two digits as the current year.

The following examples demonstrate the behavior of the RR datetime format element.

RR Datetime Format Examples

Assume these queries are issued between 1950 and 1999:

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR'), 'YYYY') "Year" FROM DUAL;

Year
----
1998

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

Year
----
2017
```

Now assume these queries are issued between 2000 and 2049:

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR'), 'YYYY') "Year" FROM DUAL;

Year
----
1998

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 datetime format element lets you write SQL statements that will return the same values from years whose first two digits are different.

Datetime Format Element Suffixes

Table 2-19 lists suffixes that can be added to datetime format elements:

Table 2-19 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

Notes on date format element suffixes:

- When you add one of these suffixes to a datetime format element, the return value is always in English.
- Datetime suffixes are valid only to format output. You cannot use them to insert a date into the database.

Format Model Modifiers

The `FM` and `FX` modifiers, used in format models in the `TO_CHAR` function, 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. Oracle uses trailing blank characters and leading zeroes to fill format elements to a constant width. The width is equal to the display width of the largest element for the relevant format model:

- Numeric elements are padded with leading zeros to the width of the maximum value allowed for the element. For example, the `YYYY` element is padded to four digits (the length of '9999'), `HH24` to two digits (the length of '23'), and `DDD` to three digits (the length of '366').
- The character elements `MONTH`, `MON`, `DAY`, and `DY` are padded with trailing blanks to the width of the longest full month name, the longest abbreviated month name, the longest full date name, or the longest abbreviated day name, respectively, among valid names determined by the values of `NLS_DATE_LANGUAGE` and `NLS_CALENDAR` parameters. For example, when `NLS_DATE_LANGUAGE` is `AMERICAN` and `NLS_CALENDAR` is `GREGORIAN` (the default), the largest element for `MONTH` is `SEPTEMBER`, so all values of the `MONTH` format element are padded to nine display characters. The values of the `NLS_DATE_LANGUAGE` and `NLS_CALENDAR` parameters are specified in the third argument to `TO_CHAR` and `TO_*` datetime functions or they are retrieved from the NLS environment of the current session.
- The character element RM is padded with trailing blanks to the length of 4, which is the length of 'viii'.
- Other character elements and spelled-out numbers (`SP`, `SPTH`, and `THSP` suffixes) are not padded.

The `FM` modifier suppresses the above padding in the return value of the `TO_CHAR` function.

FX

Format exact. This modifier specifies exact matching for the character argument and datetime 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 zeros.

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

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

Format Model Examples

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

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

Ides
------------------
3RD of April, 2008
```

The preceding statement also uses the `FM` modifier. If `FM` is omitted, then the month is blank-padded to nine characters:

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

Ides
-----------------------
03RD of April    , 2008
```

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.

Table 2-20 shows whether the following statement meets the matching conditions for different values of *char* and '*fmt*' using `FX` (the table named `table` has a column `date_column` of data type `DATE`):

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

Table 2-20 Matching Character Data and Format Models with the FX Format Model Modifier

char	'fmt'	Match or Error?
'15/ JAN /1998'	'DD-MON-YYYY'	Match
' 15! JAN % /1998'	'DD-MON-YYYY'	Error
'15/JAN/1998'	'FXDD-MON-YYYY'	Error
'15-JAN-1998'	'FXDD-MON-YYYY'	Match
'1-JAN-1998'	'FXDD-MON-YYYY'	Error
'01-JAN-1998'	'FXDD-MON-YYYY'	Match
'1-JAN-1998'	'FXFMDD-MON-YYYY'	Match

Format of Return Values: Examples

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

The following statement selects the salaries of the employees in Department 80 and uses the `TO_CHAR` function to convert these salaries into character values with the format specified by the number format model '`$99,990.99`':

```
SELECT last_name employee, TO_CHAR(salary, '$99,990.99')
  FROM employees
  WHERE department_id = 80;
```

Because of this format model, Oracle returns salaries with leading dollar signs, commas every three digits, and two decimal places.

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 last_name employee, TO_CHAR(hire_date,'fmMonth DD, YYYY') hiredate
  FROM employees
  WHERE department_id = 20;
```

With this format model, Oracle returns the hire dates without blank padding (as specified by `fm`), two digits for the day, and the century included in the year.

> **See Also:**
>
> Format Model Modifiers for a description of the `fm` format element

Supplying the Correct Format Model: Examples

When you insert or update a column value, the data type of the value that you specify must correspond to the column data type of the column. You can use format models to specify the format of a value that you are converting from one data type to another data type required for a column.

For example, a value that you insert into a DATE column must be a value of the DATE data type or a character string in the default date format (Oracle implicitly converts character strings in the default date format to the DATE data type). If the value is in another format, then you must use the TO_DATE function to convert the value to the DATE data type. You must also use a format model to specify the format of the character string.

The following statement updates Hunold's hire date using the TO_DATE function with the format mask 'YYYY MM DD' to convert the character string '2008 05 20' to a DATE value:

```
UPDATE employees
  SET hire_date = TO_DATE('2008 05 20','YYYY MM DD')
  WHERE last_name = 'Hunold';
```

String-to-Date Conversion Rules

The following additional formatting rules apply when converting string values to date values (unless you have used the FX or FXFM modifiers in the format model to control exact format checking):

- 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. For example, 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.
- You can use any non-alphanumeric character in the date string to match the punctuation symbol in the format string.
- If a match fails between a datetime format element and the corresponding characters in the date string, then Oracle attempts alternative format elements, as shown in Table 2-21.

Table 2-21 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'

XML Format Model

The SYS_XMLAgg and SYS_XMLGen (deprecated) functions return an instance of type XMLType containing an XML document. Oracle provides the XMLFormat object, which lets you format the output of these functions.

Table 2-22 lists and describes the attributes of the XMLFormat object. The function that implements this type follows the table.

> **See Also:**
>
> - SYS_XMLAGG for information on the `SYS_XMLAgg` function
> - SYS_XMLGEN for information on the `SYS_XMLGen` function
> - *Oracle XML DB Developer's Guide* for more information on the implementation of the `XMLFormat` object and its use

Table 2-22 Attributes of the XMLFormat Object

Attribute	Data Type	Purpose
`enclTag`	`VARCHAR2(4000)` or `VARCHAR2(32767)`[1]	The name of the enclosing tag for the result of the `SYS_XMLAgg` or `SYS_XMLGen` (deprecated) function. `SYS_XMLAgg`: The default is `ROWSET`. `SYS_XMLGen`: If the input to the function is a column name, then the default is the column name. Otherwise the default is `ROW`. When `schemaType` is set to `USE_GIVEN_SCHEMA`, this attribute also gives the name of the XMLSchema element.
`schemaType`	`VARCHAR2(100)`	The type of schema generation for the output document. Valid values are 'NO_SCHEMA' and 'USE_GIVEN_SCHEMA'. The default is 'NO_SCHEMA'.
`schemaName`	`VARCHAR2(4000)` or `VARCHAR2(32767)`[1]	The name of the target schema Oracle uses if the value of the `schemaType` is 'USE_GIVEN_SCHEMA'. If you specify `schemaName`, then Oracle uses the enclosing tag as the element name.
`targetNameSpace`	`VARCHAR2(4000)` or `VARCHAR2(32767)`[1]	The target namespace if the schema is specified (that is, `schemaType` is `GEN_SCHEMA_*`, or `USE_GIVEN_SCHEMA`)
`dburlPrefix`	`VARCHAR2(4000)` or `VARCHAR2(32767)`[1]	The URL to the database to use if `WITH_SCHEMA` is specified. If this attribute is not specified, then Oracle declares the URL to the types as a relative URL reference.
`processingIns`	`VARCHAR2(4000)` or `VARCHAR2(32767)`[1]	User-provided processing instructions, which are appended to the top of the function output before the element.

[1] The data type for this attribute is `VARCHAR2(4000)` if the initialization parameter `MAX_STRING_SIZE` = `STANDARD`, and `VARCHAR2(32767)` if `MAX_STRING_SIZE` = `EXTENDED`. See Extended Data Types for more information.

The function that implements the `XMLFormat` object follows:

```
STATIC FUNCTION createFormat(
     enclTag IN varchar2 := 'ROWSET',
     schemaType IN varchar2 := 'NO_SCHEMA',
     schemaName IN varchar2 := null,
     targetNameSpace IN varchar2 := null,
     dburlPrefix IN varchar2 := null,
     processingIns IN varchar2 := null) RETURN XMLGenFormatType
       deterministic parallel_enable,
  MEMBER PROCEDURE genSchema (spec IN varchar2),
  MEMBER PROCEDURE setSchemaName(schemaName IN varchar2),
  MEMBER PROCEDURE setTargetNameSpace(targetNameSpace IN varchar2),
  MEMBER PROCEDURE setEnclosingElementName(enclTag IN varchar2),
  MEMBER PROCEDURE setDbUrlPrefix(prefix IN varchar2),
  MEMBER PROCEDURE setProcessingIns(pi IN varchar2),
  CONSTRUCTOR FUNCTION XMLGenFormatType (
     enclTag IN varchar2 := 'ROWSET',
```

```
    schemaType IN varchar2 := 'NO_SCHEMA',
    schemaName IN varchar2 := null,
    targetNameSpace IN varchar2 := null,
    dbUrlPrefix IN varchar2 := null,
    processingIns IN varchar2 := null) RETURN SELF AS RESULT
     deterministic parallel_enable,
  STATIC function createFormat2(
     enclTag in varchar2 := 'ROWSET',
     flags in raw) return sys.xmlgenformattype
     deterministic parallel_enable
);
```

Nulls

If a column in a row has no value, then the column is said to be **null**, or to contain null. Nulls can appear in columns of any data type that are not restricted by `NOT NULL` or `PRIMARY KEY` integrity constraints. Use a null when the actual value is not known or when a value would not be meaningful.

Oracle Database treats a character value with a length of zero as null. However, do not use null to represent a numeric value of zero, because they are not equivalent.

> **Note:**
>
> Oracle Database currently treats a character value with a length of zero as null. However, this may not continue to be true in future releases, and Oracle recommends that you do not treat empty strings the same as nulls.

Any arithmetic expression containing a null always evaluates to null. For example, null added to 10 is null. In fact, all operators (except concatenation) return null when given a null operand.

Nulls in SQL Functions

For information on null handling in SQL functions, see Nulls in SQL Functions .

Nulls with Comparison Conditions

To test for nulls, use only the comparison conditions `IS NULL` and `IS NOT NULL`. If you use any other condition with nulls and the result depends on the value of the null, then the result is `UNKNOWN`. Because null represents a lack of data, a null cannot be equal or unequal to any value or to another null. However, Oracle considers two nulls to be equal when evaluating a `DECODE` function. Refer to DECODE for syntax and additional information.

Oracle also considers two nulls to be equal if they appear in compound keys. That is, Oracle considers identical two compound keys containing nulls if all the non-null components of the keys are equal.

Nulls in Conditions

A condition that evaluates to `UNKNOWN` acts almost like `FALSE`. For example, a `SELECT` statement with a condition in the `WHERE` clause that evaluates to `UNKNOWN` returns no rows. However, a condition evaluating to `UNKNOWN` differs from `FALSE` in that further operations on an `UNKNOWN`

condition evaluation will evaluate to UNKNOWN. Thus, NOT FALSE evaluates to TRUE, but NOT UNKNOWN evaluates to UNKNOWN.

Table 2-23 shows examples of various evaluations involving nulls in conditions. If the conditions evaluating to UNKNOWN were used in a WHERE clause of a SELECT statement, then no rows would be returned for that query.

Table 2-23 Conditions Containing Nulls

Condition	Value of A	Evaluation
a IS NULL	10	FALSE
a IS NOT NULL	10	TRUE
a IS NULL	NULL	TRUE
a IS NOT NULL	NULL	FALSE
a = NULL	10	UNKNOWN
a != NULL	10	UNKNOWN
a = NULL	NULL	UNKNOWN
a != NULL	NULL	UNKNOWN
a = 10	NULL	UNKNOWN
a != 10	NULL	UNKNOWN

For the truth tables showing the results of logical conditions containing nulls, see Table 6-5, Table 6-6, and Table 6-7.

Comments

You can create two types of comments:

- Comments within SQL statements are stored as part of the application code that executes the SQL statements.
- Comments associated with individual schema or nonschema objects are stored in the data dictionary along with metadata on the objects themselves.

Comments Within SQL Statements

Comments can make your application easier for you to read and maintain. For example, you can include a comment in a statement that describes the purpose of the statement within your application. With the exception of hints, comments within SQL statements do not affect the statement execution. Refer to Hints on using this particular form of comment.

A comment can appear between any keywords, parameters, or punctuation marks in a statement. You can include a comment in a statement in two ways:

- Begin the comment with a slash and an asterisk (/*). Proceed with the text of the comment. This text can span multiple lines. End the comment with an asterisk and a slash (*/). The opening and terminating characters need not be separated from the text by a space or a line break.
- Begin the comment with -- (two hyphens). Proceed with the text of the comment. This text cannot extend to a new line. End the comment with a line break.

Some of the tools used to enter SQL have additional restrictions. For example, if you are using SQL*Plus, by default you cannot have a blank line inside a multiline comment. For more information, refer to the documentation for the tool you use as an interface to the database.

A SQL statement can contain multiple comments of both styles. The text of a comment can contain any printable characters in your database character set.

Example

These statements contain many comments:

```
SELECT last_name, employee_id, salary + NVL(commission_pct, 0),
       job_id, e.department_id
  /* Select all employees whose compensation is
  greater than that of Pataballa.*/
  FROM employees e, departments d
  /*The DEPARTMENTS table is used to get the department name.*/
  WHERE e.department_id = d.department_id
    AND salary + NVL(commission_pct,0) >   /* Subquery:       */
      (SELECT salary + NVL(commission_pct,0)
        /* total compensation is salary + commission_pct */
        FROM employees
        WHERE last_name = 'Pataballa')
  ORDER BY last_name, employee_id;

SELECT last_name,                                   -- select the name
       employee_id                                  -- employee id
       salary + NVL(commission_pct, 0),             -- total compensation
       job_id,                                      -- job
       e.department_id                              -- and department
  FROM employees e,                                 -- of all employees
       departments d
  WHERE e.department_id = d.department_id
    AND salary + NVL(commission_pct, 0) >           -- whose compensation
                                                    -- is greater than
        (SELECT salary + NVL(commission_pct,0)      -- the compensation
          FROM employees
          WHERE last_name = 'Pataballa')            -- of Pataballa
  ORDER BY last_name                                -- and order by last name
           employee_id                              -- and employee id.
;
```

Comments on Schema and Nonschema Objects

You can use the `COMMENT` command to associate a comment with a schema object (table, view, materialized view, operator, indextype, mining model) or a nonschema object (edition) using the `COMMENT` command. You can also create a comment on a column, which is part of a table schema object. Comments associated with schema and nonschema objects are stored in the data dictionary. Refer to COMMENT for a description of this form of comment.

Hints

Hints are comments in a SQL statement that pass instructions to the Oracle Database optimizer. The optimizer uses these hints to choose an execution plan for the statement, unless some condition exists that prevents the optimizer from doing so.

Hints were introduced in Oracle7, when users had little recourse if the optimizer generated suboptimal plans. Now Oracle provides a number of tools, including the SQL Tuning Advisor, SQL plan management, and SQL Performance Analyzer, to help you address performance

problems that are not solved by the optimizer. Oracle strongly recommends that you use those tools rather than hints. The tools are far superior to hints, because when used on an ongoing basis, they provide fresh solutions as your data and database environment change.

Hints should be used sparingly, and only after you have collected statistics on the relevant tables and evaluated the optimizer plan without hints using the `EXPLAIN PLAN` statement. Changing database conditions as well as query performance enhancements in subsequent releases can have significant impact on how hints in your code affect performance.

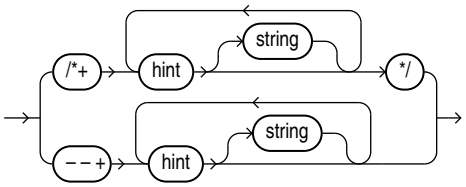
The remainder of this section provides information on some commonly used hints. If you decide to use hints rather than the more advanced tuning tools, be aware that any short-term benefit resulting from the use of hints may not continue to result in improved performance over the long term.

Using Hints

A statement block can have only one comment containing hints, and that comment must follow the `SELECT`, `UPDATE`, `INSERT`, `MERGE`, or `DELETE` keyword.

The following syntax diagram shows hints contained in both styles of comments that Oracle supports within a statement block. The hint syntax must follow immediately after an `INSERT`, `UPDATE`, `DELETE`, `SELECT`, or `MERGE` keyword that begins the statement block.

***hint*::=**



where:

- The plus sign (+) causes Oracle to interpret the comment as a list of hints. The plus sign must follow immediately after the comment delimiter. No space is permitted.
- *`hint`* is one of the hints discussed in this section. The space between the plus sign and the hint is optional. If the comment contains multiple hints, then separate the hints by at least one space.
- *`string`* is other commenting text that can be interspersed with the hints.

The `--+` syntax requires that the entire comment be on a single line.

Oracle Database ignores hints and does not return an error under the following circumstances:

- The hint contains misspellings or syntax errors. However, the database does consider other correctly specified hints in the same comment.
- The comment containing the hint does not follow a `DELETE`, `INSERT`, `MERGE`, `SELECT`, or `UPDATE` keyword.
- A combination of hints conflict with each other. However, the database does consider other hints in the same comment.
- The database environment uses PL/SQL version 1, such as Forms version 3 triggers, Oracle Forms 4.5, and Oracle Reports 2.5.

- A global hint refers to multiple query blocks. Refer to Specifying Multiple Query Blocks in a Global Hint for more information.

With 19c you can use `DBMS_XPLAN` to find out whether a hint is used or not used. For more information, see the *Database SQL Tuning Guide*.

Specifying a Query Block in a Hint

You can specify an optional query block name in many hints to specify the query block to which the hint applies. This syntax lets you specify in the outer query a hint that applies to an inline view.

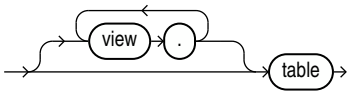
The syntax of the query block argument is of the form `@queryblock`, where `queryblock` is an identifier that specifies a query block in the query. The `queryblock` identifier can either be system-generated or user-specified. When you specify a hint in the query block itself to which the hint applies, you omit the `@queryblock` syntax.

- The system-generated identifier can be obtained by using `EXPLAIN PLAN` for the query. Pretransformation query block names can be determined by running `EXPLAIN PLAN` for the query using the `NO_QUERY_TRANSFORMATION` hint. See NO_QUERY_TRANSFORMATION Hint .
- The user-specified name can be set with the `QB_NAME` hint. See QB_NAME Hint .

Specifying Global Hints

Many hints can apply both to specific tables or indexes and more globally to tables within a view or to columns that are part of indexes. The syntactic elements `tablespec` and `indexspec` define these **global hints**.

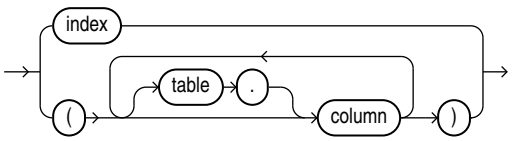
***tablespec*::=**



You must specify the table to be accessed exactly as it appears in the statement. If the statement uses an alias for the table, then use the alias rather than the table name in the hint. However, do not include the schema name with the table name within the hint, even if the schema name appears in the statement.

> **Note:**
>
> Specifying a global hint using the `tablespec` clause does not work for queries that use ANSI joins, because the optimizer generates additional views during parsing. Instead, specify `@queryblock` to indicate the query block to which the hint applies.

***indexspec*::=**



When *tablespec* is followed by *indexspec* in the specification of a hint, a comma separating the table name and index name is permitted but not required. Commas are also permitted, but not required, to separate multiple occurrences of *indexspec*.

Specifying Multiple Query Blocks in a Global Hint

Oracle Database ignores global hints that refer to multiple query blocks. To avoid this issue, Oracle recommends that you specify the object alias in the hint instead of using *tablespec* and *indexspec*.

For example, consider the following view `v` and table `t`:

```
CREATE VIEW v AS
  SELECT e.last_name, e.department_id, d.location_id
  FROM employees e, departments d
  WHERE e.department_id = d.department_id;

CREATE TABLE t AS
  SELECT * from employees
  WHERE employee_id < 200;
```

> **Note:**
>
> The following examples use the `EXPLAIN PLAN` statement, which enables you to display the execution plan and determine if a hint is honored or ignored. Refer to EXPLAIN PLAN for more information.

The `LEADING` hint is ignored in the following query because it refers to multiple query blocks, that is, the main query block containing table `t` and the view query block `v`:

```
EXPLAIN PLAN
  SET STATEMENT_ID = 'Test 1'
  INTO plan_table FOR
    (SELECT /*+ LEADING(v.e v.d t) */ *
     FROM t, v
     WHERE t.department_id = v.department_id);
```

The following `SELECT` statement returns the execution plan, which shows that the `LEADING` hint was ignored:

```
SELECT id, LPAD(' ',2*(LEVEL-1))||operation operation, options, object_name,
object_alias
  FROM plan_table
  START WITH id = 0 AND statement_id = 'Test 1'
  CONNECT BY PRIOR id = parent_id AND statement_id = 'Test 1'
  ORDER BY id;

 ID OPERATION            OPTIONS    OBJECT_NAME   OBJECT_ALIAS
--- -------------------- ---------- ------------- --------------------
  0 SELECT STATEMENT
  1   HASH JOIN
  2     HASH JOIN
  3       TABLE ACCESS   FULL       DEPARTMENTS   D@SEL$2
  4       TABLE ACCESS   FULL       EMPLOYEES     E@SEL$2
  5     TABLE ACCESS     FULL       T             T@SEL$1
```

The `LEADING` hint is honored in the following query because it refers to object aliases, which can be found in the execution plan that was returned by the previous query:

```
EXPLAIN PLAN
  SET STATEMENT_ID = 'Test 2'
  INTO plan_table FOR
    (SELECT /*+ LEADING(E@SEL$2 D@SEL$2 T@SEL$1) */ *
     FROM t, v
     WHERE t.department_id = v.department_id);
```

The following `SELECT` statement returns the execution plan, which shows that the `LEADING` hint was honored:

```
SELECT id, LPAD(' ',2*(LEVEL-1))||operation operation, options,
  object_name, object_alias
  FROM plan_table
  START WITH id = 0 AND statement_id = 'Test 2'
  CONNECT BY PRIOR id = parent_id AND statement_id = 'Test 2'
  ORDER BY id;

 ID OPERATION            OPTIONS    OBJECT_NAME   OBJECT_ALIAS
--- -------------------- ---------- ------------- --------------------
  0 SELECT STATEMENT
  1   HASH JOIN
  2     HASH JOIN
  3       TABLE ACCESS   FULL       EMPLOYEES     E@SEL$2
  4       TABLE ACCESS   FULL       DEPARTMENTS   D@SEL$2
  5     TABLE ACCESS     FULL       T             T@SEL$1
```

> **See Also:**
>
> The *Oracle Database SQL Tuning Guide* describes hints and the `EXPLAIN PLAN` .

Hints by Functional Category

Table 2-24 lists the hints by functional category and contains cross-references to the syntax and semantics for each hint. An alphabetical reference of the hints follows the table.

Table 2-24 Hints by Functional Category

Hint	Link to Syntax and Semantics
Optimization Goals and Approaches	ALL_ROWS Hint FIRST_ROWS Hint
Access Path Hints	CLUSTER Hint
--	CLUSTERING Hint NO_CLUSTERING Hint
--	FULL Hint
--	HASH Hint
--	INDEX Hint NO_INDEX Hint
--	INDEX_ASC Hint INDEX_DESC Hint
--	INDEX_COMBINE Hint
--	INDEX_JOIN Hint

Table 2-24 (Cont.) Hints by Functional Category

Hint	Link to Syntax and Semantics
--	INDEX_FFS Hint
--	INDEX_SS Hint
--	INDEX_SS_ASC Hint
--	INDEX_SS_DESC Hint
--	NATIVE_FULL_OUTER_JOIN Hint NO_NATIVE_FULL_OUTER_JOIN Hint
--	NO_INDEX_FFS Hint
--	NO_INDEX_SS Hint
--	NO_ZONEMAP Hint
In-Memory Column Store Hints	INMEMORY Hint NO_INMEMORY Hint
--	INMEMORY_PRUNING Hint NO_INMEMORY_PRUNING Hint
Join Order Hints	ORDERED Hint
--	LEADING Hint
Join Operation Hints	USE_BAND Hint NO_USE_BAND Hint
--	USE_CUBE Hint NO_USE_CUBE Hint
--	USE_HASH Hint NO_USE_HASH Hint
--	USE_MERGE Hint NO_USE_MERGE Hint
--	USE_NL Hint USE_NL_WITH_INDEX Hint NO_USE_NL Hint
Parallel Execution Hints	ENABLE_PARALLEL_DML Hint DISABLE_PARALLEL_DML Hint
--	PARALLEL Hint NO_PARALLEL Hint
--	PARALLEL_INDEX Hint NO_PARALLEL_INDEX Hint
--	PQ_CONCURRENT_UNION Hint NO_PQ_CONCURRENT_UNION Hint
--	PQ_DISTRIBUTE Hint
--	PQ_FILTER Hint
--	PQ_SKEW Hint NO_PQ_SKEW Hint
Online Application Upgrade Hints	CHANGE_DUPKEY_ERROR_INDEX Hint

Table 2-24 (Cont.) Hints by Functional Category

Hint	Link to Syntax and Semantics
--	IGNORE_ROW_ON_DUPKEY_INDEX Hint
--	RETRY_ON_ROW_CHANGE Hint
Query Transformation Hints	FACT Hint NO_FACT Hint
--	MERGE Hint NO_MERGE Hint
--	NO_EXPAND Hint USE_CONCAT Hint
--	REWRITE Hint NO_REWRITE Hint
--	UNNEST Hint NO_UNNEST Hint
--	STAR_TRANSFORMATION Hint NO_STAR_TRANSFORMATION Hint
--	NO_QUERY_TRANSFORMATION Hint
XML Hints	NO_XMLINDEX_REWRITE Hint
--	NO_XML_QUERY_REWRITE Hint
Other Hints	APPEND Hint APPEND_VALUES Hint NOAPPEND Hint
--	CACHE Hint NOCACHE Hint
--	CONTAINERS Hint
--	CURSOR_SHARING_EXACT Hint
--	DRIVING_SITE Hint
--	DYNAMIC_SAMPLING Hint
	FRESH_MV Hint
--	GATHER_OPTIMIZER_STATISTICS Hint NO_GATHER_OPTIMIZER_STATISTICS Hint
	GROUPING Hint
--	MODEL_MIN_ANALYSIS Hint
--	MONITOR Hint
--	NO_MONITOR Hint
--	OPT_PARAM Hint
--	PUSH_PRED Hint NO_PUSH_PRED Hint
--	PUSH_SUBQ Hint NO_PUSH_SUBQ Hint

Table 2-24 (Cont.) Hints by Functional Category

Hint	Link to Syntax and Semantics
--	PX_JOIN_FILTER Hint NO_PX_JOIN_FILTER Hint
--	QB_NAME Hint

Alphabetical Listing of Hints

This section provides syntax and semantics for all hints in alphabetical order.

ALL_ROWS Hint

The `ALL_ROWS` hint instructs the optimizer to optimize a statement block with a goal of best throughput, which is minimum total resource consumption. For example, the optimizer uses the query optimization approach to optimize this statement for best throughput:

```
SELECT /*+ ALL_ROWS */ employee_id, last_name, salary, job_id
  FROM employees
  WHERE employee_id = 107;
```

If you specify either the `ALL_ROWS` or the `FIRST_ROWS` hint in a SQL statement, and if the data dictionary does not have statistics about tables accessed by the statement, then the optimizer uses default statistical values, such as allocated storage for such tables, to estimate the missing statistics and to subsequently choose an execution plan. These estimates might not be as accurate as those gathered by the `DBMS_STATS` package, so you should use the `DBMS_STATS` package to gather statistics.

If you specify hints for access paths or join operations along with either the `ALL_ROWS` or `FIRST_ROWS` hint, then the optimizer gives precedence to the access paths and join operations specified by the hints.

APPEND Hint

The `APPEND` hint instructs the optimizer to use direct-path `INSERT` with the subquery syntax of the `INSERT` statement.

- Conventional `INSERT` is the default in serial mode. In serial mode, direct path can be used only if you include the `APPEND` hint.
- Direct-path `INSERT` is the default in parallel mode. In parallel mode, conventional insert can be used only if you specify the `NOAPPEND` hint.

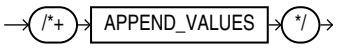
The decision whether the `INSERT` will go parallel or not is independent of the `APPEND` hint.

In direct-path INSERT, data is appended to the end of the table, rather than using existing space currently allocated to the table. As a result, direct-path INSERT can be considerably faster than conventional INSERT.

The APPEND hint is only supported with the subquery syntax of the INSERT statement, not the VALUES clause. If you specify the APPEND hint with the VALUES clause, it is ignored and conventional insert will be used. To use direct-path INSERT with the VALUES clause, refer to "APPEND_VALUES Hint ".

> **See Also:**
>
> NOAPPEND Hint for information on that hint and *Oracle Database Administrator's Guide* for information on direct-path inserts

APPEND_VALUES Hint



The APPEND_VALUES hint instructs the optimizer to use direct-path INSERT with the VALUES clause. If you do not specify this hint, then conventional INSERT is used.

In direct-path INSERT, data is appended to the end of the table, rather than using existing space currently allocated to the table. As a result, direct-path INSERT can be considerably faster than conventional INSERT.

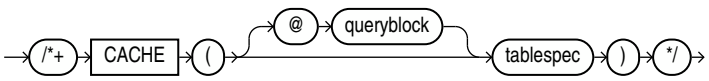
The APPEND_VALUES hint can be used to greatly enhance performance. Some examples of its uses are:

- In an Oracle Call Interface (OCI) program, when using large array binds or array binds with row callbacks
- In PL/SQL, when loading a large number of rows with a FORALL loop that has an INSERT statement with a VALUES clause

The APPEND_VALUES hint is only supported with the VALUES clause of the INSERT statement. If you specify the APPEND_VALUES hint with the subquery syntax of the INSERT statement, it is ignored and conventional insert will be used. To use direct-path INSERT with a subquery, refer to "APPEND Hint ".

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information on direct-path inserts

CACHE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

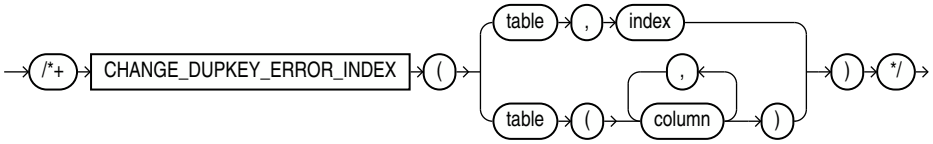
The `CACHE` hint instructs the optimizer to place the blocks retrieved for the table at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This hint is useful for small lookup tables.

In the following example, the `CACHE` hint overrides the default caching specification of the table:

```
SELECT /*+ FULL (hr_emp) CACHE(hr_emp) */ last_name
  FROM employees hr_emp;
```

The `CACHE` and `NOCACHE` hints affect system statistics `table scans (long tables)` and `table scans (short tables)`, as shown in the `V$SYSSTAT` data dictionary view.

CHANGE_DUPKEY_ERROR_INDEX Hint



> **Note:**
>
> The `CHANGE_DUPKEY_ERROR_INDEX`, `IGNORE_ROW_ON_DUPKEY_INDEX`, and `RETRY_ON_ROW_CHANGE` hints are unlike other hints in that they have a semantic effect. The general philosophy explained in Hints does not apply for these three hints.

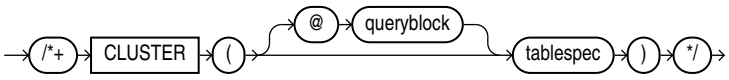
The `CHANGE_DUPKEY_ERROR_INDEX` hint provides a mechanism to unambiguously identify a unique key violation for a specified set of columns or for a specified index. When a unique key violation occurs for the specified index, an ORA-38911 error is reported instead of an ORA-001.

This hint applies to `INSERT`, `UPDATE` operations. If you specify an index, then the index must exist and be unique. If you specify a column list instead of an index, then a unique index whose columns match the specified columns in number and order must exist.

This use of this hint results in error messages if specific rules are violated. Refer to IGNORE_ROW_ON_DUPKEY_INDEX Hint for details.

> **Note:**
>
> This hint disables both `APPEND` mode and parallel DML.

CLUSTER Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `CLUSTER` hint instructs the optimizer to use a cluster scan to access the specified table. This hint applies only to tables in an indexed cluster.

CLUSTERING Hint

This hint is valid only for `INSERT` and `MERGE` operations on tables that are enabled for attribute clustering. The `CLUSTERING` hint enables attribute clustering for direct-path inserts (serial or parallel). This results in partially-clustered data, that is, data that is clustered per each insert or merge operation. This hint overrides a `NO ON LOAD` setting in the DDL that created or altered the table. This hint has no effect on tables that are not enabled for attribute clustering.

See Also:

- clustering_when clause of `CREATE TABLE` for more information on the `NO ON LOAD` setting
- NO_CLUSTERING Hint

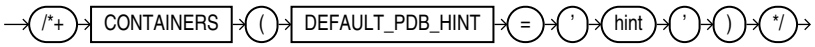
COMPRESS_IMMEDIATE Hint

Syntax

`COMPRESS_IMMEDIATE` forces compression to happen immediately during direct load.

When Automatic Storage Compression is enabled via `DBMS_ILM_ADMIN.ENABLE_AUTO_OPTIMIZE`, compression is delayed for new direct loads. Use this hint to overide the delay and compress the direct load immediately.

CONTAINERS Hint



The `CONTAINERS` hint is useful in a multitenant container database (CDB). You can specify this hint in a `SELECT` statement that contains the `CONTAINERS()` clause. Such a statement lets you query data in the specified table or view across all containers in a CDB or application container.

- To query data in a CDB, you must be a common user connected to the CDB root, and the table or view must exist in the root and all PDBs. The query returns all rows from the table or view in the CDB root and in all open PDBs.
- To query data in an application container, you must be a common user connected to the application root, and the table or view must exist in the application root and all PDBs in the application container. The query returns all rows from the table or view in the application root and in all open PDBs in the application container.

Statements that contain the `CONTAINERS()` clause generate and execute recursive SQL statements in each queried PDB. You can use the `CONTAINERS` hint to pass a default PDB hint to each recursive SQL statement. For *`hint`*, you can specify any SQL hint that is appropriate for the `SELECT` statement.

In the following example, the `NO_PARALLEL` hint is passed to each recursive SQL statement that is executed as part of the evaluation of the `CONTAINERS()` clause:

```
SELECT /*+ CONTAINERS(DEFAULT_PDB_HINT='NO_PARALLEL') */
  (CASE WHEN COUNT(*) < 10000
        THEN 'Less than 10,000'
        ELSE '10,000 or more' END) "Number of Tables"
  FROM CONTAINERS(DBA_TABLES);
```

> **See Also:**
>
> *containers_clause* for more information on the `CONTAINERS()` clause

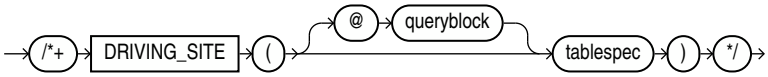
CURSOR_SHARING_EXACT Hint

Oracle can replace literals in SQL statements with bind variables, when it is safe to do so. This replacement is controlled with the `CURSOR_SHARING` initialization parameter. The `CURSOR_SHARING_EXACT` hint instructs the optimizer to switch this behavior off. When you specify this hint, Oracle executes the SQL statement without any attempt to replace literals with bind variables.

DISABLE_PARALLEL_DML Hint

The `DISABLE_PARALLEL_DML` hint disables parallel DML for `DELETE`, `INSERT`, `MERGE`, and `UPDATE` statements. You can use this hint to disable parallel DML for an individual statement when parallel DML is enabled for the session with the `ALTER SESSION ENABLE PARALLEL DML` statement.

DRIVING_SITE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

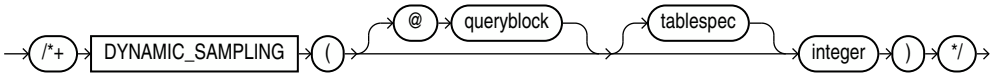
The `DRIVING_SITE` hint instructs the optimizer to execute the query at a different site than that selected by the database. This hint is useful if you are using distributed query optimization.

For example:

```
SELECT /*+ DRIVING_SITE(departments) */ *
  FROM employees, departments@rsite
  WHERE employees.department_id = departments.department_id;
```

If this query is executed without the hint, then rows from `departments` are sent to the local site, and the join is executed there. With the hint, the rows from `employees` are sent to the remote site, and the query is executed there and the result set is returned to the local site.

DYNAMIC_SAMPLING Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `DYNAMIC_SAMPLING` hint instructs the optimizer how to control dynamic sampling to improve server performance by determining more accurate predicate selectivity and statistics for tables and indexes.

You can set the value of `DYNAMIC_SAMPLING` to a value from 0 to 10. The higher the level, the more effort the compiler puts into dynamic sampling and the more broadly it is applied. Sampling defaults to cursor level unless you specify *`tablespec`*.

The *`integer`* value is `0` to `10`, indicating the degree of sampling.

If a cardinality statistic already exists for the table, then the optimizer uses it. Otherwise, the optimizer enables dynamic sampling to estimate the cardinality statistic.

If you specify *`tablespec`* and the cardinality statistic already exists, then:

- If there is no single-table predicate (a `WHERE` clause that evaluates only one table), then the optimizer trusts the existing statistics and ignores this hint. For example, the following query will not result in any dynamic sampling if `employees` is analyzed:

  ```
  SELECT /*+ DYNAMIC_SAMPLING(e 1) */ count(*)
    FROM employees e;
  ```

- If there is a single-table predicate, then the optimizer uses the existing cardinality statistic and estimates the selectivity of the predicate using the existing statistics.

To apply dynamic sampling to a specific table, use the following form of the hint:

```
SELECT /*+ DYNAMIC_SAMPLING(employees 1) */ *
  FROM employees
  WHERE ...
```

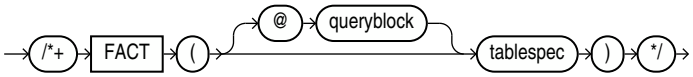
> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information about dynamic sampling and the sampling levels that you can set

ENABLE_PARALLEL_DML Hint

The `ENABLE_PARALLEL_DML` hint enables parallel DML for `DELETE`, `INSERT`, `MERGE`, and `UPDATE` statements. You can use this hint to enable parallel DML for an individual statement, rather than enabling parallel DML for the session with the `ALTER SESSION ENABLE PARALLEL DML` statement.

> **See Also:**
>
> *Oracle Database VLDB and Partitioning Guide* for information about enabling parallel DML

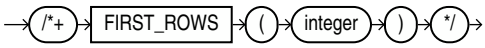
FACT Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `FACT` hint is used in the context of the star transformation. It instructs the optimizer that the table specified in *`tablespec`* should be considered as a fact table.

FIRST_ROWS Hint



The `FIRST_ROWS` hint instructs Oracle to optimize an individual SQL statement for fast response, choosing the plan that returns the first *`n`* rows most efficiently. For *`integer`*, specify the number of rows to return.

For example, the optimizer uses the query optimization approach to optimize the following statement for best response time:

```
SELECT /*+ FIRST_ROWS(10) */ employee_id, last_name, salary, job_id
  FROM employees
  WHERE department_id = 20;
```

In this example each department contains many employees. The user wants the first 10 employees of department 20 to be displayed as quickly as possible.

The optimizer ignores this hint in `DELETE` and `UPDATE` statement blocks and in `SELECT` statement blocks that include any blocking operations, such as sorts or groupings. Such statements cannot be optimized for best response time, because Oracle Database must retrieve all rows accessed by the statement before returning the first row. If you specify this hint in any such statement, then the database optimizes for best throughput.

> **See Also:**
>
> ALL_ROWS Hint for additional information on the `FIRST_ROWS` hint and statistics

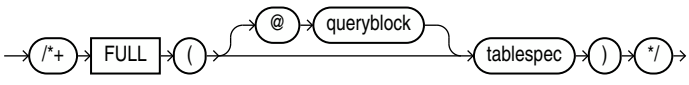
FRESH_MV Hint

The `FRESH_MV` hint applies when querying a real-time materialized view. This hint instructs the optimizer to use on-query computation to fetch up-to-date data from the materialized view, even if the materialized view is stale.

The optimizer ignores this hint in `SELECT` statement blocks that query an object that is not a real-time materialized view, and in all `UPDATE`, `INSERT`, `MERGE`, and `DELETE` statement blocks.

> **See Also:**
>
> The { ENABLE | DISABLE } ON QUERY COMPUTATION clause of `CREATE MATERIALIZED VIEW` for more information on real-time materialized views

FULL Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `FULL` hint instructs the optimizer to perform a full table scan for the specified table. For example:

```
SELECT /*+ FULL(e) */ employee_id, last_name
  FROM hr.employees e
  WHERE last_name LIKE :b1;
```

Oracle Database performs a full table scan on the `employees` table to execute this statement, even if there is an index on the `last_name` column that is made available by the condition in the `WHERE` clause.

The `employees` table has alias `e` in the `FROM` clause, so the hint must refer to the table by its alias rather than by its name. Do not specify schema names in the hint even if they are specified in the `FROM` clause.

GATHER_OPTIMIZER_STATISTICS Hint

The `GATHER_OPTIMIZER_STATISTICS` hint instructs the optimizer to enable statistics gathering during the following types of bulk loads:

- `CREATE TABLE ... AS SELECT`
- `INSERT INTO ... SELECT` into an empty table using a direct-path insert

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information on statistics gathering for bulk loads

GROUPING Hint

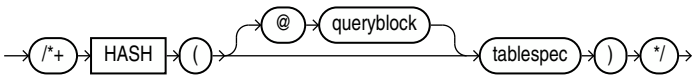
The `GROUPING` hint applies to data mining scoring functions when scoring partitioned models. This hint results in partitioning the input data set into distinct data slices so that each partition is scored in its entirety before advancing to the next partition; however, parallelism by partition is still available. Data slices are determined by the partitioning key columns that were used when the model was built. This method can be used with any data mining function against a partitioned model. The hint may yield a query performance gain when scoring large data that is associated with many partitions, but may negatively impact performance when scoring large data with few partitions on large systems. Typically, there is no performance gain if you use this hint for single row queries.

In the following example, the `GROUPING` hint is used in the `PREDICTION` function.

```
SELECT PREDICTION(/*+ GROUPING */my_model USING *) pred FROM <input table>;
```

> **See Also:**
>
> Oracle Machine Learning for SQL Functions

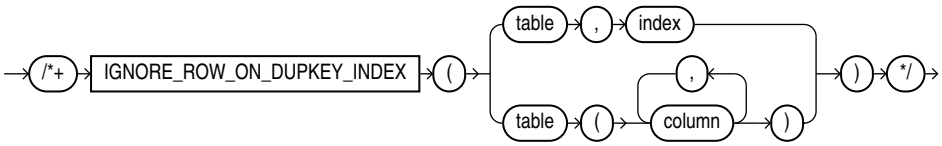
HASH Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `HASH` hint instructs the optimizer to use a hash scan to access the specified table. This hint applies only to tables in a hash cluster.

IGNORE_ROW_ON_DUPKEY_INDEX Hint



> **Note:**
>
> The `CHANGE_DUPKEY_ERROR_INDEX`, `IGNORE_ROW_ON_DUPKEY_INDEX`, and `RETRY_ON_ROW_CHANGE` hints are unlike other hints in that they have a semantic effect. The general philosophy explained in Hints does not apply for these three hints.

The `IGNORE_ROW_ON_DUPKEY_INDEX` hint applies only to single-table `INSERT` operations. It is not supported for `UPDATE`, `DELETE`, `MERGE`, or multitable insert operations. `IGNORE_ROW_ON_DUPKEY_INDEX` causes the statement to ignore a unique key violation for a specified set of columns or for a specified index. When a unique key violation is encountered, a row-level rollback occurs and execution resumes with the next input row. If you specify this hint when inserting data with DML error logging enabled, then the unique key violation is not logged and does not cause statement termination.

The semantic effect of this hint results in error messages if specific rules are violated:

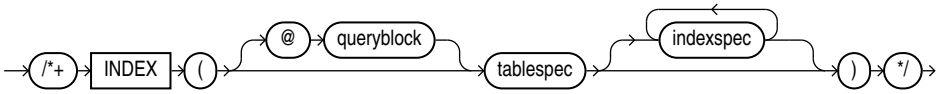
- If you specify *`index`*, then the index must exist and be unique. Otherwise, the statement causes ORA-38913.
- You must specify exactly one index. If you specify no index, then the statement causes ORA-38912. If you specify more than one index, then the statement causes ORA-38915.
- You can specify either a `CHANGE_DUPKEY_ERROR_INDEX` or `IGNORE_ROW_ON_DUPKEY_INDEX` hint in an `INSERT` statement, but not both. If you specify both, then the statement causes ORA-38915.

As with all hints, a syntax error in the hint causes it to be silently ignored. The result will be that ORA-00001 will be caused, just as if no hint were used.

> **Note:**
>
> This hint disables both `APPEND` mode and parallel `DML`.

> **See Also:**
>
> CHANGE_DUPKEY_ERROR_INDEX Hint

INDEX Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX` hint instructs the optimizer to use an index scan for the specified table. You can use the `INDEX` hint for function-based, domain, B-tree, bitmap, and bitmap join indexes.

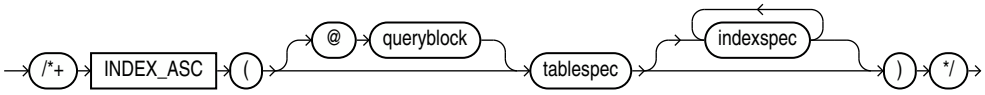
The behavior of the hint depends on the *`indexspec`* specification:

- If the `INDEX` hint specifies a single available index, then the database performs a scan on this index. The optimizer does not consider a full table scan or a scan of another index on the table.
- For a hint on a combination of multiple indexes, Oracle recommends using `INDEX_COMBINE` rather than `INDEX`, because it is a more versatile hint. If the `INDEX` hint specifies a list of available indexes, then the optimizer considers the cost of a scan on each index in the list and then performs the index scan with the lowest cost. The database can also choose to scan multiple indexes from this list and merge the results, if such an access path has the lowest cost. The database does not consider a full table scan or a scan on an index not listed in the hint.
- If the `INDEX` hint specifies no indexes, then the optimizer considers the cost of a scan on each available index on the table and then performs the index scan with the lowest cost. The database can also choose to scan multiple indexes and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan.

For example:

```
SELECT /*+ INDEX (employees emp_department_ix)*/ employee_id, department_id
  FROM employees
  WHERE department_id > 50;
```

INDEX_ASC Hint

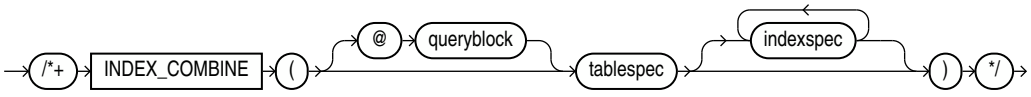


(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_ASC` hint instructs the optimizer to use an index scan for the specified table. If the statement uses an index range scan, then Oracle Database scans the index entries in ascending order of their indexed values. Each parameter serves the same purpose as in INDEX Hint .

The default behavior for a range scan is to scan index entries in ascending order of their indexed values, or in descending order for a descending index. This hint does not change the default order of the index, and therefore does not specify anything more than the `INDEX` hint. However, you can use the `INDEX_ASC` hint to specify ascending range scans explicitly should the default behavior change.

INDEX_COMBINE Hint

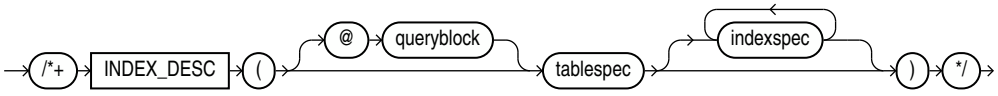


(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_COMBINE` hint can use any type of index: bitmap, b-tree, or domain. If you do not specify *`indexspec`* in the `INDEX_COMBINE` hint, the optimizer implicitly applies the`INDEX` hint to all indexes, using as many indexes as possible. If you specify *`indexspec`*, then the optimizer uses all the hinted indexes that are legal and valid to use, regardless of cost. Each parameter serves the same purpose as in INDEX Hint . For example:

```
SELECT /*+ INDEX_COMBINE(e emp_manager_ix emp_department_ix) */ *
  FROM employees e
  WHERE manager_id = 108
     OR department_id = 110;
```

INDEX_DESC Hint



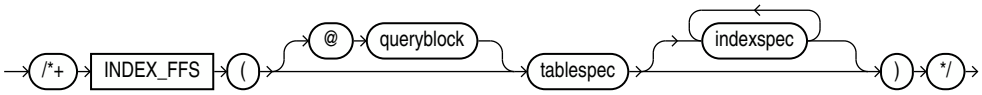
(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_DESC` hint instructs the optimizer to use a descending index scan for the specified table. If the statement uses an index range scan and the index is ascending, then Oracle scans the index entries in descending order of their indexed values. In a partitioned index, the results are in descending order within each partition. For a descending index, this hint effectively cancels out the descending order, resulting in a scan of the index entries in ascending order. Each parameter serves the same purpose as in INDEX Hint . For example:

```
SELECT /*+ INDEX_DESC(e emp_name_ix) */ *
  FROM employees e;
```

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information on full scans

INDEX_FFS Hint



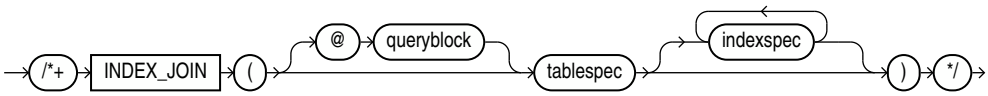
(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_FFS` hint instructs the optimizer to perform a fast full index scan rather than a full table scan.

Each parameter serves the same purpose as in INDEX Hint . For example:

```
SELECT /*+ INDEX_FFS(e emp_name_ix) */ first_name
  FROM employees e;
```

INDEX_JOIN Hint



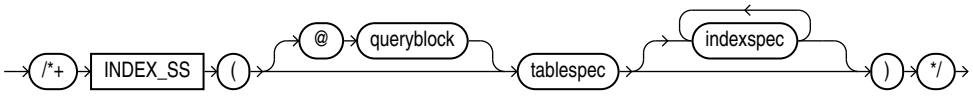
(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_JOIN` hint instructs the optimizer to use an index join as an access path. For the hint to have a positive effect, a sufficiently small number of indexes must exist that contain all the columns required to resolve the query.

Each parameter serves the same purpose as in INDEX Hint . For example, the following query uses an index join to access the `manager_id` and `department_id` columns, both of which are indexed in the `employees` table.

```
SELECT /*+ INDEX_JOIN(e emp_manager_ix emp_department_ix) */ department_id
  FROM employees e
  WHERE manager_id < 110
    AND department_id < 50;
```

INDEX_SS Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

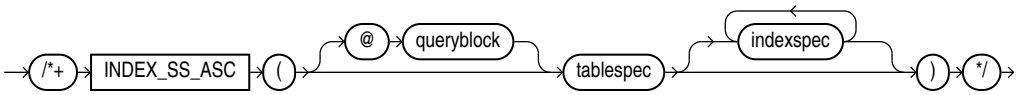
The `INDEX_SS` hint instructs the optimizer to perform an index skip scan for the specified table. If the statement uses an index range scan, then Oracle scans the index entries in ascending order of their indexed values. In a partitioned index, the results are in ascending order within each partition.

Each parameter serves the same purpose as in INDEX Hint . For example:

```
SELECT /*+ INDEX_SS(e emp_name_ix) */ last_name
  FROM employees e
  WHERE first_name = 'Steven';
```

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information on index skip scans

INDEX_SS_ASC Hint



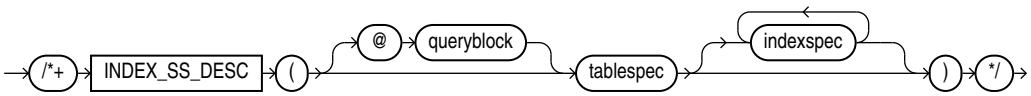
(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_SS_ASC` hint instructs the optimizer to perform an index skip scan for the specified table. If the statement uses an index range scan, then Oracle Database scans the index entries in ascending order of their indexed values. In a partitioned index, the results are in ascending order within each partition. Each parameter serves the same purpose as in INDEX Hint .

The default behavior for a range scan is to scan index entries in ascending order of their indexed values, or in descending order for a descending index. This hint does not change the default order of the index, and therefore does not specify anything more than the `INDEX_SS` hint. However, you can use the `INDEX_SS_ASC` hint to specify ascending range scans explicitly should the default behavior change.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information on index skip scans

INDEX_SS_DESC Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `INDEX_SS_DESC` hint instructs the optimizer to perform an index skip scan for the specified table. If the statement uses an index range scan and the index is ascending, then Oracle scans

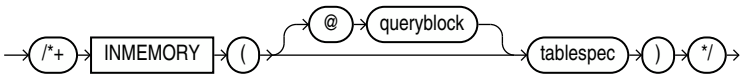
the index entries in descending order of their indexed values. In a partitioned index, the results are in descending order within each partition. For a descending index, this hint effectively cancels out the descending order, resulting in a scan of the index entries in ascending order.

Each parameter serves the same purpose as in the INDEX Hint . For example:

```
SELECT /*+ INDEX_SS_DESC(e emp_name_ix) */ last_name
  FROM employees e
  WHERE first_name = 'Steven';
```

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information on index skip scans

INMEMORY Hint

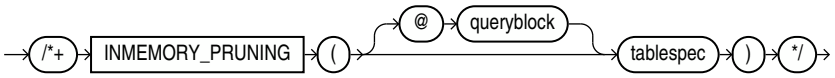


(See Specifying a Query Block in a Hint , *tablespec*::=)

The `INMEMORY` hint enables In-Memory queries.

This hint does not instruct the optimizer to perform a full table scan. If a full table scan is desired, then also specify the FULL Hint .

INMEMORY_PRUNING Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

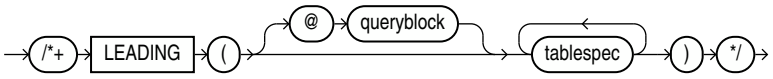
The `INMEMORY_PRUNING` hint enables pruning of In-Memory queries.

IVF_ITERATION Hint

Use the `IVF_ITERATION` hint to specify a terminable iteration IVF index.

For more on terminable iteration for an IVF index see Terminable Iteration for IVF Index of the *AI Vector Search User's Guide*.

LEADING Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

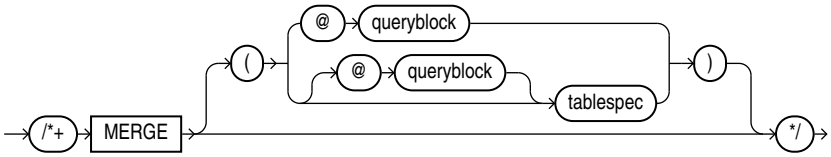
The `LEADING` hint is a multitable hint that can specify more than one table or view. `LEADING` instructs the optimizer to use the specified set of tables as the prefix in the execution plan. The first table specified is used to start the join.

This hint is more versatile than the `ORDERED` hint. For example:

```
SELECT /*+ LEADING(e j) */ *
    FROM employees e, departments d, job_history j
    WHERE e.department_id = d.department_id
      AND e.hire_date = j.start_date;
```

The `LEADING` hint is ignored if the tables specified cannot be joined first in the order specified because of dependencies in the join graph. If you specify two or more conflicting `LEADING` hints, then all of them are ignored. If you specify the `ORDERED` hint, it overrides all `LEADING` hints.

MERGE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `MERGE` hint lets you merge views in a query.

If a view's query block contains a `GROUP BY` clause or `DISTINCT` operator in the `SELECT` list, then the optimizer can merge the view into the accessing statement only if complex view merging is enabled. Complex merging can also be used to merge an `IN` subquery into the accessing statement if the subquery is uncorrelated.

For example:

```
SELECT /*+ MERGE(v) */ e1.last_name, e1.salary, v.avg_salary
   FROM employees e1,
        (SELECT department_id, avg(salary) avg_salary
           FROM employees e2
           GROUP BY department_id) v
   WHERE e1.department_id = v.department_id
     AND e1.salary > v.avg_salary
   ORDER BY e1.last_name;
```

When the `MERGE` hint is used without an argument, it should be placed in the view query block. When `MERGE` is used with the view name as an argument, it should be placed in the surrounding query.

MODEL_MIN_ANALYSIS Hint

The `MODEL_MIN_ANALYSIS` hint instructs the optimizer to omit some compile-time optimizations of spreadsheet rules—primarily detailed dependency graph analysis. Other spreadsheet optimizations, such as creating filters to selectively populate spreadsheet access structures and limited rule pruning, are still used by the optimizer.

This hint reduces compilation time because spreadsheet analysis can be lengthy if the number of spreadsheet rules is more than several hundreds.

MONITOR Hint

The `MONITOR` hint forces real-time SQL monitoring for the query, even if the statement is not long running. This hint is valid only when the parameter `CONTROL_MANAGEMENT_PACK_ACCESS` is set to `DIAGNOSTIC+TUNING`.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information about real-time SQL monitoring

NATIVE_FULL_OUTER_JOIN Hint

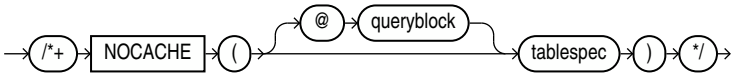
The `NATIVE_FULL_OUTER_JOIN` hint instructs the optimizer to use native full outer join, which is a native execution method based on a hash join.

> **See Also:**
>
> - NO_NATIVE_FULL_OUTER_JOIN Hint
> - *Oracle Database SQL Tuning Guide* for more information about native full outer joins

NOAPPEND Hint

The `NOAPPEND` hint instructs the optimizer to use conventional `INSERT` even when `INSERT` is performed in parallel mode.

NOCACHE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NOCACHE` hint instructs the optimizer to place the blocks retrieved for the table at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the normal behavior of blocks in the buffer cache. For example:

```
SELECT /*+ FULL(hr_emp) NOCACHE(hr_emp) */ last_name
  FROM employees hr_emp;
```

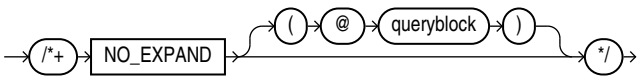
The `CACHE` and `NOCACHE` hints affect system statistics `table scans(long tables)` and `table scans(short tables)`, as shown in the `V$SYSSTAT` view.

NO_CLUSTERING Hint

This hint is valid only for `INSERT` and `MERGE` operations on tables that are enabled for attribute clustering. The `NO_CLUSTERING` hint disables attribute clustering for direct-path inserts (serial or parallel). This hint overrides a `YES ON LOAD` setting in the DDL that created or altered the table. This hint has no effect on tables that are not enabled for attribute clustering.

> **See Also:**
>
> - clustering_when clause of `CREATE TABLE` for more information on the `YES ON LOAD` setting
> - CLUSTERING Hint

NO_EXPAND Hint



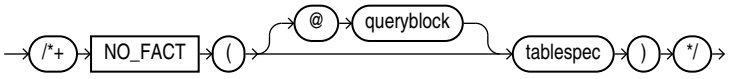
(See Specifying a Query Block in a Hint)

The `NO_EXPAND` hint instructs the optimizer not to consider `OR`-expansion for queries having `OR` conditions or `IN`-lists in the `WHERE` clause. Usually, the optimizer considers using `OR` expansion and uses this method if it decides that the cost is lower than not using it. For example:

```
SELECT /*+ NO_EXPAND */ *
  FROM employees e, departments d
  WHERE e.manager_id = 108
     OR d.department_id = 110;
```

> **See Also:**
>
> The USE_CONCAT Hint , which is the opposite of this hint

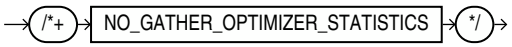
NO_FACT Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_FACT` hint is used in the context of the star transformation. It instruct the optimizer that the queried table should not be considered as a fact table.

NO_GATHER_OPTIMIZER_STATISTICS Hint



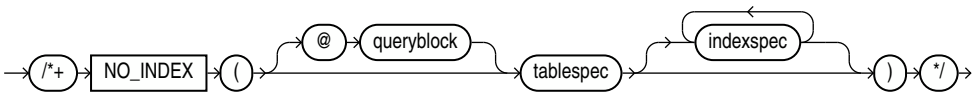
The `NO_GATHER_OPTIMIZER_STATISTICS` hint instructs the optimizer to disable statistics gathering during the following types of bulk loads:

- `CREATE TABLE AS SELECT`
- `INSERT INTO ... SELECT` into an empty table using a direct path insert

The `NO_GATHER_OPTIMIZER_STATISTICS` hint is applicable to a conventional load. If this hint is specified in the conventional insert statement, Oracle will obey the hint and not collect real-time statistics.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information on online statistics gathering for conventional loads.

NO_INDEX Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `NO_INDEX` hint instructs the optimizer not to use one or more indexes for the specified table. For example:

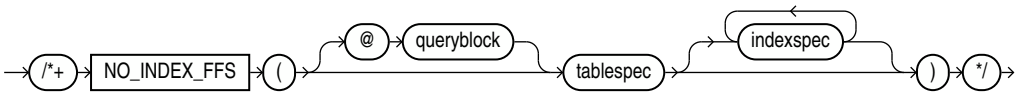
```
SELECT /*+ NO_INDEX(employees emp_empid) */ employee_id
  FROM employees
  WHERE employee_id > 200;
```

Each parameter serves the same purpose as in INDEX Hint with the following modifications:

- If this hint specifies a single available index, then the optimizer does not consider a scan on this index. Other indexes not specified are still considered.
- If this hint specifies a list of available indexes, then the optimizer does not consider a scan on any of the specified indexes. Other indexes not specified in the list are still considered.
- If this hint specifies no indexes, then the optimizer does not consider a scan on any index on the table. This behavior is the same as a `NO_INDEX` hint that specifies a list of all available indexes for the table.

The `NO_INDEX` hint applies to function-based, B-tree, bitmap, cluster, or domain indexes. If a `NO_INDEX` hint and an index hint (`INDEX`, `INDEX_ASC`, `INDEX_DESC`, `INDEX_COMBINE`, or `INDEX_FFS`) both specify the same indexes, then the database ignores both the `NO_INDEX` hint and the index hint for the specified indexes and considers those indexes for use during execution of the statement.

NO_INDEX_FFS Hint

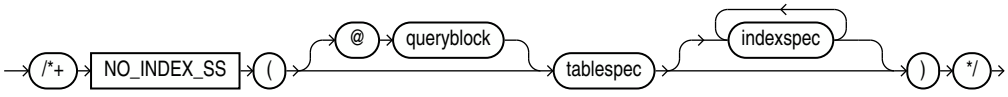


(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `NO_INDEX_FFS` hint instructs the optimizer to exclude a fast full index scan of the specified indexes on the specified table. Each parameter serves the same purpose as in the NO_INDEX Hint . For example:

```
SELECT /*+ NO_INDEX_FFS(items item_order_ix) */ order_id
  FROM order_items items;
```

NO_INDEX_SS Hint

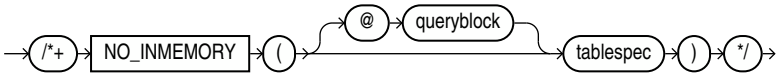


(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `NO_INDEX_SS` hint instructs the optimizer to exclude a skip scan of the specified indexes on the specified table. Each parameter serves the same purpose as in the NO_INDEX Hint .

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information on index skip scans

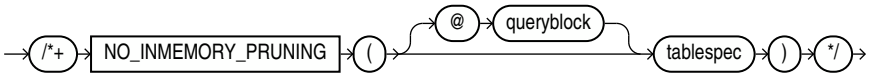
NO_INMEMORY Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_INMEMORY` hint disables In-Memory queries.

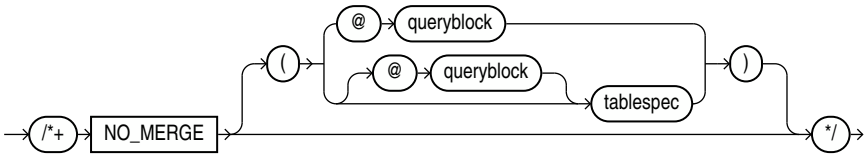
NO_INMEMORY_PRUNING Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_INMEMORY_PRUNING` hint disables pruning of In-Memory queries.

NO_MERGE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_MERGE` hint instructs the optimizer not to combine the outer query and any inline view queries into a single query.

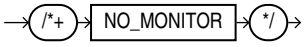
This hint lets you have more influence over the way in which the view is accessed. For example, the following statement causes view `seattle_dept` not to be merged:

```
SELECT /*+ NO_MERGE(seattle_dept) */ e1.last_name, seattle_dept.department_name
  FROM employees e1,
       (SELECT location_id, department_id, department_name
          FROM departments
```

```
        WHERE location_id = 1700) seattle_dept
WHERE e1.department_id = seattle_dept.department_id;
```

When you use the `NO_MERGE` hint in the view query block, specify it without an argument. When you specify `NO_MERGE` in the surrounding query, specify it with the view name as an argument.

NO_MONITOR Hint



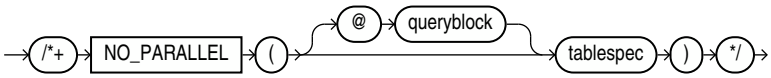
The `NO_MONITOR` hint disables real-time SQL monitoring for the query, even if the query is long running.

NO_NATIVE_FULL_OUTER_JOIN Hint

The `NO_NATIVE_FULL_OUTER_JOIN` hint instructs the optimizer to exclude the native execution method when joining each specified table. Instead, the full outer join is executed as a union of left outer join and anti-join.

> **See Also:**
>
> NATIVE_FULL_OUTER_JOIN Hint

NO_PARALLEL Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_PARALLEL` hint instructs the optimizer to run the statement serially. This hint overrides the value of the `PARALLEL_DEGREE_POLICY` initialization parameter. It also overrides a `PARALLEL` parameter in the DDL that created or altered the table. For example, the following `SELECT` statement will run serially:

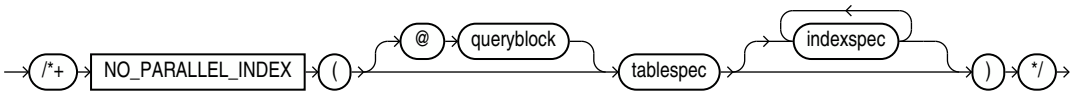
```
ALTER TABLE employees PARALLEL 8;
SELECT /*+ NO_PARALLEL(hr_emp) */ last_name
  FROM employees hr_emp;
```

> **See Also:**
>
> - Note on Parallel Hints for more information on the parallel hints
> - *Oracle Database Reference* for more information on the `PARALLEL_DEGREE_POLICY` initialization parameter

NOPARALLEL Hint

The `NOPARALLEL` hint has been deprecated. Use the `NO_PARALLEL` hint instead.

NO_PARALLEL_INDEX Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

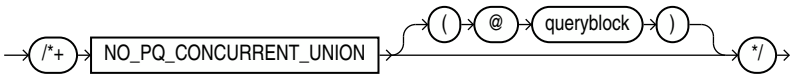
The `NO_PARALLEL_INDEX` hint overrides a `PARALLEL` parameter in the DDL that created or altered the index, thus avoiding a parallel index scan operation.

> **See Also:**
>
> Note on Parallel Hints for more information on the parallel hints

NOPARALLEL_INDEX Hint

The `NOPARALLEL_INDEX` hint has been deprecated. Use the `NO_PARALLEL_INDEX` hint instead.

NO_PQ_CONCURRENT_UNION Hint

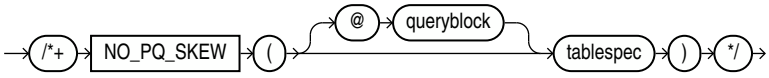


(See Specifying a Query Block in a Hint)

The `NO_PQ_CONCURRENT_UNION` hint instructs the optimizer to disable concurrent processing of `UNION` and `UNION ALL` operations.

> **See Also:**
>
> - PQ_CONCURRENT_UNION Hint
> - *Oracle Database VLDB and Partitioning Guide* for information about using this hint

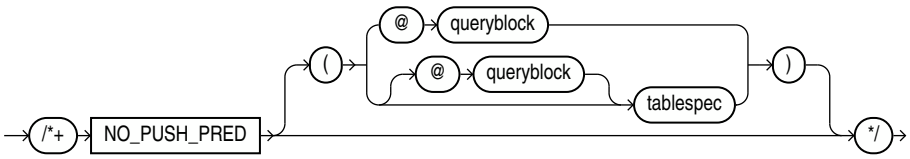
NO_PQ_SKEW Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_PQ_SKEW` hint advises the optimizer that the distribution of the values of the join keys for a parallel join is not skewed—that is, a high percentage of rows do not have the same join key values. The table specified in *`tablespec`* is the probe table of the hash join.

NO_PUSH_PRED Hint

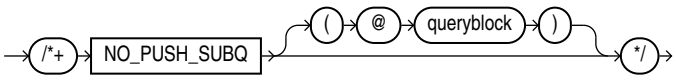


(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_PUSH_PRED` hint instructs the optimizer not to push a join predicate into the view. For example:

```
SELECT /*+ NO_MERGE(v) NO_PUSH_PRED(v) */ *
  FROM employees e,
       (SELECT manager_id
          FROM employees) v
  WHERE e.manager_id = v.manager_id(+)
    AND e.employee_id = 100;
```

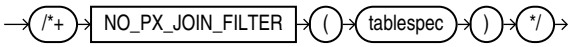
NO_PUSH_SUBQ Hint



(See Specifying a Query Block in a Hint)

The NO_PUSH_SUBQ hint instructs the optimizer to evaluate nonmerged subqueries as the last step in the execution plan. Doing so can improve performance if the subquery is relatively expensive or does not reduce the number of rows significantly.

NO_PX_JOIN_FILTER Hint



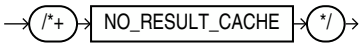
This hint prevents the optimizer from using parallel join bitmap filtering.

NO_QUERY_TRANSFORMATION Hint

The NO_QUERY_TRANSFORMATION hint instructs the optimizer to skip all query transformations, including but not limited to OR-expansion, view merging, subquery unnesting, star transformation, and materialized view rewrite. For example:

```
SELECT /*+ NO_QUERY_TRANSFORMATION */ employee_id, last_name
  FROM (SELECT * FROM employees e) v
  WHERE v.last_name = 'Smith';
```

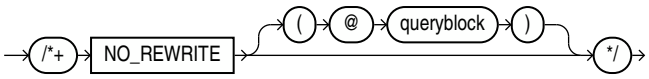
NO_RESULT_CACHE Hint



The optimizer caches query results in the result cache if the RESULT_CACHE_MODE initialization parameter is set to FORCE. In this case, the NO_RESULT_CACHE hint disables such caching for the current query.

If the query is executed from OCI client and OCI client result cache is enabled, then the NO_RESULT_CACHE hint disables caching for the current query.

NO_REWRITE Hint



(See Specifying a Query Block in a Hint)

The NO_REWRITE hint instructs the optimizer to disable query rewrite for the query block, overriding the setting of the parameter QUERY_REWRITE_ENABLED. For example:

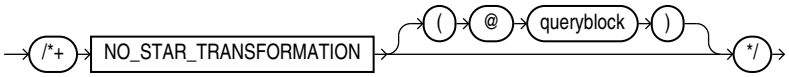
```
SELECT /*+ NO_REWRITE */ sum(s.amount_sold) AS dollars
  FROM sales s, times t
```

```
WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

NOREWRITE Hint

The `NOREWRITE` hint has been deprecated. Use the `NO_REWRITE` hint instead.

NO_STAR_TRANSFORMATION Hint



(See Specifying a Query Block in a Hint)

The `NO_STAR_TRANSFORMATION` hint instructs the optimizer not to perform star query transformation.

NO_STATEMENT_QUEUING Hint

The `NO_STATEMENT_QUEUING` hint influences whether or not a statement is queued with parallel statement queuing.

When `PARALLEL_DEGREE_POLICY` is set to `AUTO`, this hint enables a statement to bypass the parallel statement queue. However, a statement that bypasses the statement queue can potentially cause the system to exceed the maximum number of parallel execution servers defined by the value of the `PARALLEL_SERVERS_TARGET` initialization parameter, which determines the limit at which parallel statement queuing is initiated.

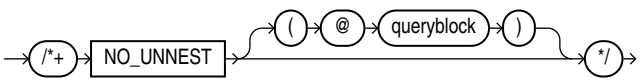
There is no guarantee that the statement that bypasses the parallel statement queue receives the number of parallel execution servers requested because only the number of parallel execution servers available on the system, up to the value of the `PARALLEL_MAX_SERVERS` initialization parameter, can be allocated.

For example:

```
SELECT /*+ NO_STATEMENT_QUEUING */ emp.last_name, dpt.department_name
  FROM employees emp, departments dpt
  WHERE emp.department_id = dpt.department_id;
```

> **See Also:**
>
> STATEMENT_QUEUING Hint

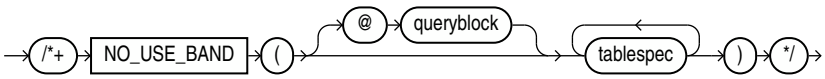
NO_UNNEST Hint



(See Specifying a Query Block in a Hint)

Use of the `NO_UNNEST` hint turns off unnesting .

NO_USE_BAND Hint

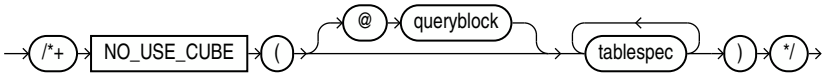


(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_USE_BAND` hint instructs the optimizer to exclude band joins when joining each specified table to another row source. For example:

```
SELECT /*+ NO_USE_BAND(e1 e2) */
  e1.last_name
  || ' has salary between 100 less and 100 more than '
  || e2.last_name AS "SALARY COMPARISON"
FROM employees e1, employees e2
WHERE e1.salary BETWEEN e2.salary - 100 AND e2.salary + 100;
```

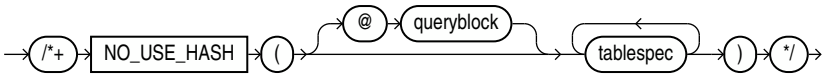
NO_USE_CUBE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_USE_CUBE` hint instructs the optimizer to exclude cube joins when joining each specified table to another row source using the specified table as the inner table.

NO_USE_HASH Hint

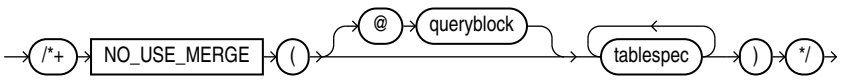


(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_USE_HASH` hint instructs the optimizer to exclude hash joins when joining each specified table to another row source using the specified table as the inner table. For example:

```
SELECT /*+ NO_USE_HASH(e d) */ *
  FROM employees e, departments d
  WHERE e.department_id = d.department_id;
```

NO_USE_MERGE Hint

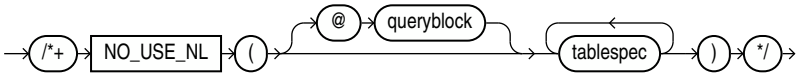


(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_USE_MERGE` hint instructs the optimizer to exclude sort-merge joins when joining each specified table to another row source using the specified table as the inner table. For example:

```
SELECT /*+ NO_USE_MERGE(e d) */ *
   FROM employees e, departments d
   WHERE e.department_id = d.department_id
   ORDER BY d.department_id;
```

NO_USE_NL Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_USE_NL` hint instructs the optimizer to exclude nested loops joins when joining each specified table to another row source using the specified table as the inner table. For example:

```
SELECT /*+ NO_USE_NL(l h) */ *
  FROM orders h, order_items l
  WHERE l.order_id = h.order_id
    AND l.order_id > 2400;
```

When this hint is specified, only hash join and sort-merge joins are considered for the specified tables. However, in some cases tables can be joined only by using nested loops. In such cases, the optimizer ignores the hint for those tables.

NO_XML_QUERY_REWRITE Hint

The `NO_XML_QUERY_REWRITE` hint instructs the optimizer to prohibit the rewriting of XPath expressions in SQL statements. By prohibiting the rewriting of XPath expressions, this hint also prohibits the use of any XMLIndexes for the current query. For example:

```
SELECT /*+NO_XML_QUERY_REWRITE*/ XMLQUERY('<A/>' RETURNING CONTENT)
  FROM DUAL;
```

> **See Also:**
>
> NO_XMLINDEX_REWRITE Hint

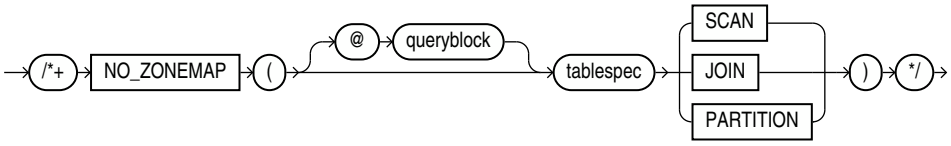
NO_XMLINDEX_REWRITE Hint

The `NO_XMLINDEX_REWRITE` hint instructs the optimizer not to use any XMLIndex indexes for the current query. For example:

```
SELECT /*+NO_XMLINDEX_REWRITE*/ count(*)
  FROM warehouses
  WHERE existsNode(warehouse_spec, '/Warehouse/Building') = 1;
```

> **See Also:**
>
> NO_XML_QUERY_REWRITE Hint for another way to disable the use of XMLIndexes

NO_ZONEMAP Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `NO_ZONEMAP` hint disables the use of a zone map for different types of pruning. This hint overrides an `ENABLE PRUNING` setting in the DDL that created or altered the zone map.

Specify one of the following options:

- `SCAN` - Disables the use of a zone map for scan pruning.
- `JOIN` - Disables the use of a zone map for join pruning.
- `PARTITION` - Disables the use of a zone map for partition pruning.

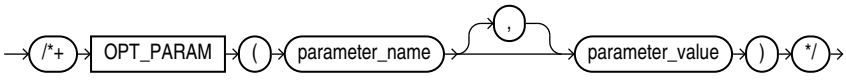
> **See Also:**
>
> - ENABLE | DISABLE PRUNING clause of `CREATE MATERIALIZED ZONEMAP`
> - *Oracle Database Data Warehousing Guide* for more information on pruning with zone maps

OPTIMIZER_FEATURES_ENABLE Hint

This hint is fully documented in the Database Reference book.

Please see Database Reference for details.

OPT_PARAM Hint



The OPT_PARAM hint lets you set an initialization parameter for the duration of the current query only. This hint is valid only for the following parameters: APPROX_FOR_AGGREGATION, APPROX_FOR_COUNT_DISTINCT, APPROX_FOR_PERCENTILE, OPTIMIZER_DYNAMIC_SAMPLING, OPTIMIZER_INDEX_CACHING, OPTIMIZER_INDEX_COST_ADJ, and STAR_TRANSFORMATION_ENABLED.

For example, the following hint sets the parameter STAR_TRANSFORMATION_ENABLED to TRUE for the statement to which it is added:

```
SELECT /*+ OPT_PARAM('star_transformation_enabled' 'true') */ *
  FROM ... ;
```

Parameter values that are strings are enclosed in single quotation marks. Numeric parameter values are specified without quotation marks.

ORDERED Hint

The ORDERED hint instructs Oracle to join tables in the order in which they appear in the FROM clause. Oracle recommends that you use the LEADING hint, which is more versatile than the ORDERED hint.

When you omit the ORDERED hint from a SQL statement requiring a join, the optimizer chooses the order in which to join the tables. You might want to use the ORDERED hint to specify a join order if you know something that the optimizer does not know about the number of rows selected from each table. Such information lets you choose an inner and outer table better than the optimizer could.

The following query is an example of the use of the ORDERED hint:

```
SELECT  /*+ ORDERED */ o.order_id, c.customer_id, l.unit_price * l.quantity
  FROM customers c, order_items l, orders o
  WHERE c.cust_last_name = 'Taylor'
    AND o.customer_id = c.customer_id
    AND o.order_id = l.order_id;
```

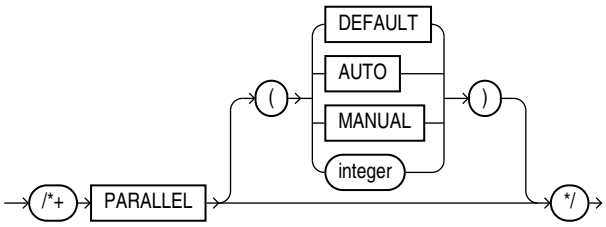
PARALLEL Hint

Note on Parallel Hints

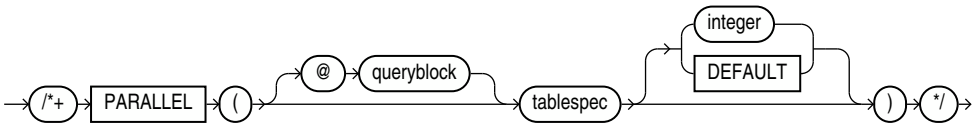
Beginning with Oracle Database 11*g* Release 2, the PARALLEL and NO_PARALLEL hints are statement-level hints and supersede the earlier object-level hints: PARALLEL_INDEX, NO_PARALLEL_INDEX, and previously specified PARALLEL and NO_PARALLEL hints. For PARALLEL, if you specify *integer*, then that degree of parallelism will be used for the statement. If you omit *integer*, then the database computes the degree of parallelism. All the access paths that can use parallelism will use the specified or computed degree of parallelism.

In the syntax diagrams below, *parallel_hint_statement* shows the syntax for statement-level hints, and *parallel_hint_object* shows the syntax for object-level hints. Object-level hints are supported for backward compatibility, and are superseded by statement-level hints.

***parallel_hint_statement*::=**



***parallel_hint_object*::=**



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `PARALLEL` hint instructs the optimizer to use the specified number of concurrent servers for a parallel operation. This hint overrides the value of the `PARALLEL_DEGREE_POLICY` initialization parameter. It applies to the `SELECT`, `INSERT`, `MERGE`, `UPDATE`, and `DELETE` portions of a statement, as well as to the table scan portion. If any parallel restrictions are violated, then the hint is ignored.

> **Note:**
>
> The number of servers that can be used is twice the value in the `PARALLEL` hint, if sorting or grouping operations also take place.

For a **statement-level PARALLEL hint**:

- `PARALLEL`: The statement results in a degree of parallelism equal to or greater than the computed degree of parallelism, except when parallelism is not feasible for the lowest cost plan. When parallelism is is not feasible, the statement runs serially.
- `PARALLEL` (`DEFAULT`): The optimizer calculates a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the `PARALLEL_THREADS_PER_CPU` initialization parameter.
- `PARALLEL` (`AUTO`): The statement results in a degree of parallelism that is equal to or greater than the computed degree of parallelism, except when parallelism is not feasible for the lowest cost plan. When parallelism is is not feasible, the statement runs serially.
- `PARALLEL` (`MANUAL`): The optimizer is forced to use the parallel settings of the objects in the statement.
- `PARALLEL` (*`integer`*): The optimizer uses the degree of parallelism specified by *`integer`*.

In the following example, the optimizer calculates the degree of parallelism. The statement always runs in parallel.

```
SELECT /*+ PARALLEL */ last_name
  FROM employees;
```

In the following example, the optimizer calculates the degree of parallelism, but that degree may be 1, in which case the statement will run serially.

```
SELECT /*+ PARALLEL (AUTO) */ last_name
  FROM employees;
```

In the following example, the `PARALLEL` hint advises the optimizer to use the degree of parallelism currently in effect for the table itself, which is 5:

```
CREATE TABLE parallel_table (col1 number, col2 VARCHAR2(10)) PARALLEL 5;

SELECT /*+ PARALLEL (MANUAL) */ col2
  FROM parallel_table;
```

For an **object-level PARALLEL hint**:

- `PARALLEL`: The query coordinator should examine the settings of the initialization parameters to determine the default degree of parallelism.
- `PARALLEL` (*`integer`*): The optimizer uses the degree of parallelism specified by *`integer`*.
- `PARALLEL` (`DEFAULT`): The optimizer calculates a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the `PARALLEL_THREADS_PER_CPU` initialization parameter.

In the following example, the `PARALLEL` hint overrides the degree of parallelism specified in the `employees` table definition:

```
SELECT /*+ FULL(hr_emp) PARALLEL(hr_emp, 5) */ last_name
  FROM employees hr_emp;
```

In the next example, the `PARALLEL` hint overrides the degree of parallelism specified in the `employees` table definition and instructs the optimizer to calculate a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the `PARALLEL_THREADS_PER_CPU` initialization parameter.

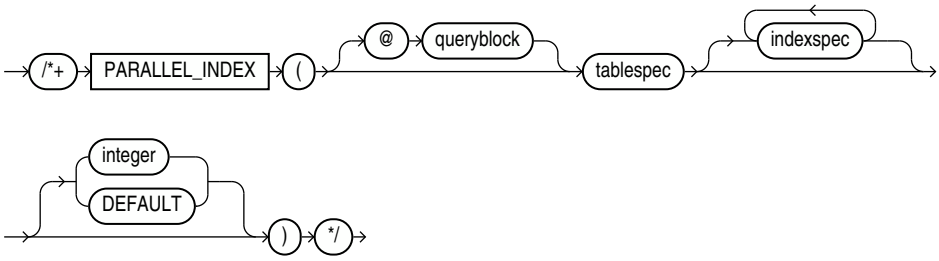
```
SELECT /*+ FULL(hr_emp) PARALLEL(hr_emp, DEFAULT) */ last_name
  FROM employees hr_emp;
```

Refer to CREATE TABLE and *Oracle Database Concepts* for more information on parallel execution.

See Also:

- CREATE TABLE and *Oracle Database Concepts* for more information on parallel execution.
- *Oracle Database PL/SQL Packages and Types Reference* for information on the `DBMS_PARALLEL_EXECUTE` package, which provides methods to apply table changes in chunks of rows. Changes to each chunk are independently committed when there are no errors.
- *Oracle Database Reference* for more information on the `PARALLEL_DEGREE_POLICY` initialization parameter
- NO_PARALLEL Hint

PARALLEL_INDEX Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

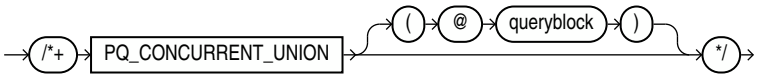
The `PARALLEL_INDEX` hint instructs the optimizer to use the specified number of concurrent servers to parallelize index range scans, full scans, and fast full scans for partitioned indexes.

The *`integer`* value indicates the degree of parallelism for the specified index. Specifying `DEFAULT` or no value signifies that the query coordinator should examine the settings of the initialization parameters to determine the default degree of parallelism. For example, the following hint indicates three parallel execution processes are to be used:

```
SELECT /*+ PARALLEL_INDEX(table1, index1, 3) */
```

> **See Also:**
>
> Note on Parallel Hints for more information on the parallel hints

PQ_CONCURRENT_UNION Hint

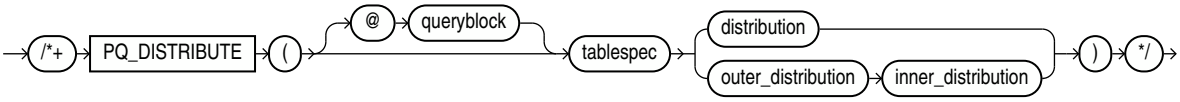


(See Specifying a Query Block in a Hint)

The `PQ_CONCURRENT_UNION` hint instructs the optimizer to enable concurrent processing of `UNION` and `UNION ALL` operations.

> **See Also:**
>
> - NO_PQ_CONCURRENT_UNION Hint
> - *Oracle Database VLDB and Partitioning Guide* for information about using this hint

PQ_DISTRIBUTE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `PQ_DISTRIBUTE` hint instructs the optimizer how to distribute rows among producer and consumer query servers. You can control the distribution of rows for either joins or for load.

Control of Distribution for Load

You can control the distribution of rows for parallel `INSERT ... SELECT` and parallel `CREATE TABLE ... AS SELECT` statements to direct how rows should be distributed between the producer (query) and the consumer (load) servers. Use the upper branch of the syntax by specifying a single distribution method. The values of the distribution methods and their semantics are described in Table 2-25.

Table 2-25 Distribution Values for Load

Distribution	Description
`NONE`	No distribution. That is the query and load operation are combined into each query server. All servers will load all partitions. This lack of distribution is useful to avoid the overhead of distributing rows where there is no skew. Skew can occur due to empty segments or to a predicate in the statement that filters out all rows evaluated by the query. If skew occurs due to using this method, then use either `RANDOM` or `RANDOM_LOCAL` distribution instead. **Note**: Use this distribution with care. Each partition loaded requires a minimum of 512 KB per process of PGA memory. If you also use compression, then approximately 1.5 MB of PGA memory is consumer per server.
`PARTITION`	This method uses the partitioning information of *`tablespec`* to distribute the rows from the query servers to the load servers. Use this distribution method when it is not possible or desirable to combine the query and load operations, when the number of partitions being loaded is greater than or equal to the number of load servers, and the input data will be evenly distributed across the partitions being loaded—that is, there is no skew.
`RANDOM`	This method distributes the rows from the producers in a round-robin fashion to the consumers. Use this distribution method when the input data is highly skewed.
`RANDOM_LOCAL`	This method distributes the rows from the producers to a set of servers that are responsible for maintaining a given set of partitions. Two or more servers can be loading the same partition, but no servers are loading all partitions. Use this distribution method when the input data is skewed and combining query and load operations is not possible due to memory constraints.

For example, in the following direct-path insert operation, the query and load portions of the operation are combined into each query server:

```
INSERT /*+ APPEND PARALLEL(target_table, 16) PQ_DISTRIBUTE(target_table, NONE) */
  INTO target_table
  SELECT * FROM source_table;
```

In the following table creation example, the optimizer uses the partitioning of target_table to distribute the rows:

```
CREATE /*+ PQ_DISTRIBUTE(target_table, PARTITION) */ TABLE target_table
  NOLOGGING PARALLEL 16
  PARTITION BY HASH (l_orderkey) PARTITIONS 512
  AS SELECT * FROM source_table;
```

Control of Distribution for Joins

You control the distribution method for joins by specifying two distribution methods, as shown in the lower branch of the syntax diagram, one distribution for the outer table and one distribution for the inner table.

- *outer_distribution* is the distribution for the outer table.
- *inner_distribution* is the distribution for the inner table.

The values of the distributions are HASH, BROADCAST, PARTITION, and NONE. Only six combinations table distributions are valid, as described in Table 2-26:

Table 2-26 Distribution Values for Joins

Distribution	Description
HASH, HASH	The rows of each table are mapped to consumer query servers, using a hash function on the join keys. When mapping is complete, each query server performs the join between a pair of resulting partitions. This distribution is recommended when the tables are comparable in size and the join operation is implemented by hash-join or sort merge join.
BROADCAST, NONE	All rows of the outer table are broadcast to each query server. The inner table rows are randomly partitioned. This distribution is recommended when the outer table is very small compared with the inner table. As a general rule, use this distribution when the inner table size multiplied by the number of query servers is greater than the outer table size.
NONE, BROADCAST	All rows of the inner table are broadcast to each consumer query server. The outer table rows are randomly partitioned. This distribution is recommended when the inner table is very small compared with the outer table. As a general rule, use this distribution when the inner table size multiplied by the number of query servers is less than the outer table size.
PARTITION, NONE	The rows of the outer table are mapped using the partitioning of the inner table. The inner table must be partitioned on the join keys. This distribution is recommended when the number of partitions of the outer table is equal to or nearly equal to a multiple of the number of query servers; for example, 14 partitions and 15 query servers. **Note**: The optimizer ignores this hint if the inner table is not partitioned or not equijoined on the partitioning key.

Table 2-26 (Cont.) Distribution Values for Joins

Distribution	Description
NONE, PARTITION	The rows of the inner table are mapped using the partitioning of the outer table. The outer table must be partitioned on the join keys. This distribution is recommended when the number of partitions of the outer table is equal to or nearly equal to a multiple of the number of query servers; for example, 14 partitions and 15 query servers. **Note**: The optimizer ignores this hint if the outer table is not partitioned or not equijoined on the partitioning key.
NONE, NONE	Each query server performs the join operation between a pair of matching partitions, one from each table. Both tables must be equipartitioned on the join keys.

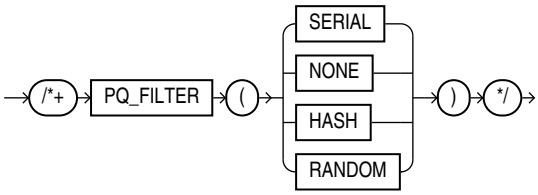
For example, given two tables `r` and `s` that are joined using a hash join, the following query contains a hint to use hash distribution:

```
SELECT /*+ORDERED PQ_DISTRIBUTE(s HASH, HASH) USE_HASH (s)*/ column_list
  FROM r,s
  WHERE r.c=s.c;
```

To broadcast the outer table `r`, the query is:

```
SELECT /*+ORDERED PQ_DISTRIBUTE(s BROADCAST, NONE) USE_HASH (s) */ column_list
  FROM r,s
  WHERE r.c=s.c;
```

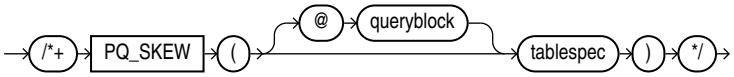
PQ_FILTER Hint



The `PQ_FILTER` hint instructs the optimizer on how to process rows when filtering correlated subqueries.

- `SERIAL`: Process rows serially on the left and right sides of the filter. Use this option when the overhead of parallelization is too high for the query, for example, when the left side has very few rows.
- `NONE`: Process rows in parallel on the left and right sides of the filter. Use this option when there is no skew in the distribution of the data on the left side of the filter and you would like to avoid distribution of the left side, for example, due to the large size of the left side.
- `HASH`: Process rows in parallel on the left side of the filter using a hash distribution. Process rows serially on the right side of the filter. Use this option when there is no skew in the distribution of data on the left side of the filter.
- `RANDOM`: Process rows in parallel on the left side of the filter using a random distribution. Process rows serially on the right side of the filter. Use this option when there is skew in the distribution of data on the left side of the filter.

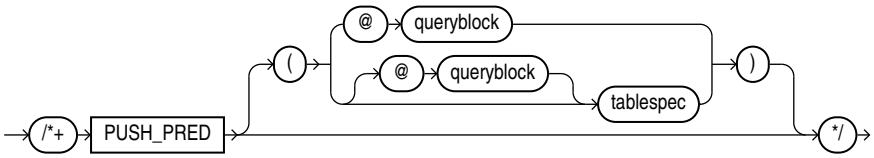
PQ_SKEW Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

The `PQ_SKEW` hint advises the optimizer that the distribution of the values of the join keys for a parallel join is highly skewed—that is, a high percentage of rows have the same join key values. The table specified in *`tablespec`* is the probe table of the hash join.

PUSH_PRED Hint

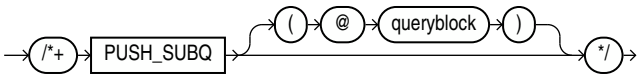


(See Specifying a Query Block in a Hint , *tablespec*::=)

The `PUSH_PRED` hint instructs the optimizer to push a join predicate into the view. For example:

```
SELECT /*+ NO_MERGE(v) PUSH_PRED(v) */ *
  FROM employees e,
    (SELECT manager_id
      FROM employees) v
  WHERE e.manager_id = v.manager_id(+)
    AND e.employee_id = 100;
```

PUSH_SUBQ Hint

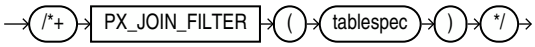


(See Specifying a Query Block in a Hint)

The `PUSH_SUBQ` hint instructs the optimizer to evaluate nonmerged subqueries at the earliest possible step in the execution plan. Generally, subqueries that are not merged are executed as the last step in the execution plan. If the subquery is relatively inexpensive and reduces the number of rows significantly, then evaluating the subquery earlier can improve performance.

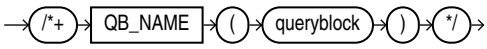
This hint has no effect if the subquery is applied to a remote table or one that is joined using a merge join.

PX_JOIN_FILTER Hint



This hint forces the optimizer to use parallel join bitmap filtering.

QB_NAME Hint



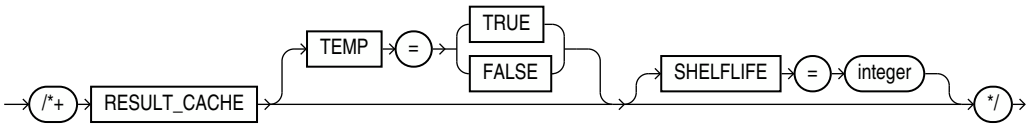
(See Specifying a Query Block in a Hint)

Use the `QB_NAME` hint to define a name for a query block. This name can then be used in a hint in the outer query or even in a hint in an inline view to affect query execution on the tables appearing in the named query block.

If two or more query blocks have the same name, or if the same query block is hinted twice with different names, then the optimizer ignores all the names and the hints referencing that query block. Query blocks that are not named using this hint have unique system-generated names. These names can be displayed in the plan table and can also be used in hints within the query block, or in query block hints. For example:

```
SELECT /*+ QB_NAME(qb) FULL(@qb e) */ employee_id, last_name
  FROM employees e
  WHERE last_name = 'Smith';
```

RESULT_CACHE Hint



The `RESULT_CACHE` hint instructs the database to cache the results of the current query or query fragment in memory and then to use the cached results in future executions of the query or query fragment. The hint is recognized in the top-level query, the *`subquery_factoring_clause`*, or `FROM` clause inline view. The cached results reside in the result cache memory portion of the shared pool.

A cached result is automatically invalidated whenever a database object used in its creation is successfully modified.

TEMP = TRUE | FALSE

If `TEMP` has a value of `TRUE` , then the query will be allowed to spill to disk and allocate space in the temporary tablespace, if needed.

If `TEMP` has a value of `FALSE` , then the query will not be allowed to spill to disk and use the temporary tablespace for caching the result.

Both values TRUE and FALSE override the value of the RESULT_CACHE_MODE initialization parameter.

If you do not specify TEMP, then the value of RESULT_CACHE_MODE holds.

SHELFLIFE

Use SHELFLIFE to specify how long (in seconds) the result of a query or a query fragment should be cached in memory.

SHELFLIFE has two purposes:

- It specifies how long results will be cached for objects where the database has no knowledge about when to invalidate. These are results based on objects like fixed objects, objects accessed via DB or Cloud Links, or Data Link objects.
- It specifies how long results will be cached for local objects. Without SHELFLIFE, results on local objects are cached until they are aged out of the result cache. With this object you can define when a result will be automatically invalidated even if no DML happened on the objects.

The SHELFLIFE value must be a positive integer. The maximum value is 4294967295 seconds.

Example: RESULT_CACHE with SHELFLIFE

The following example shows a RESULT_CACHE hint with a value of 120 forSHELFLIFE. This means that the result of the query or query fragment in which this hint appears will be cached for 120 seconds.

```
/*+ RESULT_CACHE (SHELFLIFE=120) */
```

After 120 seconds, the cached result is marked as invalid.

If the query result is large and does not fit in memory, use both the SHELFLIFE and the TEMP options to indicate that the result should be written to disk in the temporary tablespace.

Example: RESULT_CACHE with TEMP and SHELFLIFE

```
/*+ RESULT_CACHE ( TEMP= true SHELFLIFE=120) */
```

RESULT_CACHE_INTEGRITY Parameter

The initialization parameter RESULT_CACHE_INTEGRITY specifies whether the result cache will consider queries using non-deterministic constructs - such as PL/SQL functions that are not declared as deterministic, as queries that can be cached.

- If you set RESULT_CACHE_INTEGRITY to ENFORCED, then only deterministic constructs will be eligible for result caching. The ENFORCED setting overrides the setting of RESULT_CACHE_MODE or specified hints. For example, queries using PL/SQL functions that are not declared as deterministic will never be cached and must be declared as deterministic.
- If you set RESULT_CACHE_INTEGRITY to TRUSTED, then the database honors the setting of RESULT_CACHE_MODE and specified hints and considers queries using possibly non-deterministic constructs as candidates for result caching. For example, queries using PL/SQL functions that are not declared as deterministic can be cached. Note, however, that results that are known to be nondeterministic will not be cached, e.g. SYSDATE or constructs involving SYSDATE.

If the query is executed from an OCI client and the OCI client result cache is enabled, then the `RESULT_CACHE` hint enables client caching for the current query.

> **See Also:**
>
> *Oracle Database Performance Tuning Guide* for information about using this hint, *Oracle Database Reference* for information about the `RESULT_CACHE_MODE` initialization parameter, and *Oracle Call Interface Developer's Guide* for more information about the OCI result cache and usage guidelines

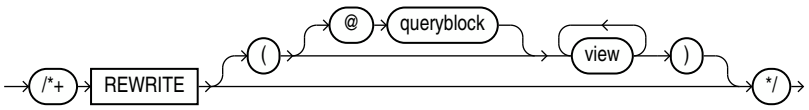
RETRY_ON_ROW_CHANGE Hint

> **Note:**
>
> The `CHANGE_DUPKEY_ERROR_INDEX`, `IGNORE_ROW_ON_DUPKEY_INDEX`, and `RETRY_ON_ROW_CHANGE` hints are unlike other hints in that they have a semantic effect. The general philosophy explained in Hints does not apply for these three hints.

This hint is valid only for `UPDATE` and `DELETE` operations. It is not supported for `INSERT` or `MERGE` operations. When you specify this hint, the operation is retried when the `ORA_ROWSCN` for one or more rows in the set has changed from the time the set of rows to be modified is determined to the time the block is actually modified.

> **See Also:**
>
> IGNORE_ROW_ON_DUPKEY_INDEX Hint and CHANGE_DUPKEY_ERROR_INDEX Hint

REWRITE Hint



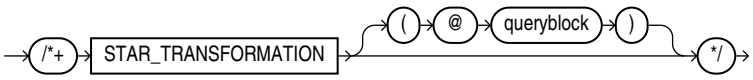
(See Specifying a Query Block in a Hint)

The `REWRITE` hint instructs the optimizer to rewrite a query in terms of materialized views, when possible, without cost consideration. Use the `REWRITE` hint with or without a view list. If you use `REWRITE` with a view list and the list contains an eligible materialized view, then Oracle uses that view regardless of its cost.

Oracle does not consider views outside of the list. If you do not specify a view list, then Oracle searches for an eligible materialized view and always uses it regardless of the cost of the final plan.

> **See Also:**
>
> - *Oracle Database Concepts* for more information on materialized views
> - *Oracle Database Data Warehousing Guide* for more information on using `REWRITE` with materialized views

STAR_TRANSFORMATION Hint



(See Specifying a Query Block in a Hint)

The `STAR_TRANSFORMATION` hint instructs the optimizer to use the best plan in which the transformation has been used. Without the hint, the optimizer could make a query optimization decision to use the best plan generated without the transformation, instead of the best plan for the transformed query. For example:

```
SELECT /*+ STAR_TRANSFORMATION */ s.time_id, s.prod_id, s.channel_id
  FROM sales s, times t, products p, channels c
  WHERE s.time_id = t.time_id
    AND s.prod_id = p.prod_id
    AND s.channel_id = c.channel_id
    AND c.channel_desc = 'Tele Sales';
```

Even if the hint is specified, there is no guarantee that the transformation will take place. The optimizer generates the subqueries only if it seems reasonable to do so. If no subqueries are generated, then there is no transformed query, and the best plan for the untransformed query is used, regardless of the hint.

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for a full discussion of star transformation.
> - *Oracle Database Reference* for more information on the `STAR_TRANSFORMATION_ENABLED` initialization parameter.

STATEMENT_QUEUING Hint

The NO_STATEMENT_QUEUING hint influences whether or not a statement is queued with parallel statement queuing.

When PARALLEL_DEGREE_POLICY is not set to AUTO, this hint enables a statement to be considered for parallel statement queuing, but to run only when enough parallel processes are available to run at the requested DOP. The number of available parallel execution servers, before queuing is enabled, is equal to the difference between the number of parallel execution servers in use and the maximum number allowed in the system, which is defined by the PARALLEL_SERVERS_TARGET initialization parameter.

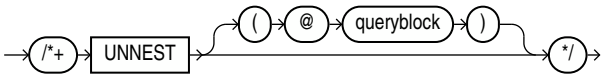
For example:

```
SELECT /*+ STATEMENT_QUEUING */ emp.last_name, dpt.department_name
  FROM employees emp, departments dpt
  WHERE emp.department_id = dpt.department_id;
```

See Also:

NO_STATEMENT_QUEUING Hint

UNNEST Hint



(See Specifying a Query Block in a Hint)

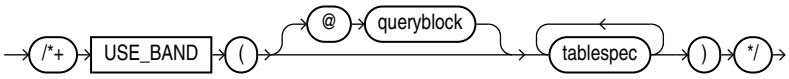
The UNNEST hint instructs the optimizer to unnest and merge the body of the subquery into the body of the query block that contains it, allowing the optimizer to consider them together when evaluating access paths and joins.

Before a subquery is unnested, the optimizer first verifies whether the statement is valid. The statement must then pass heuristic and query optimization tests. The UNNEST hint instructs the optimizer to check the subquery block for validity only. If the subquery block is valid, then subquery unnesting is enabled without checking the heuristics or costs.

See Also:

- Collection Unnesting: Examples for more information on unnesting nested subqueries and the conditions that make a subquery block valid
- *Oracle Database SQL Tuning Guide* for additional information on subquery unnesting

USE_BAND Hint



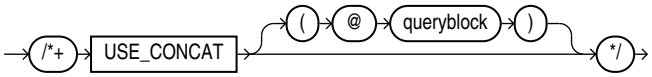
(See Specifying a Query Block in a Hint , *tablespec*::=)

The `USE_BAND` hint instructs the optimizer to join each specified table with another row source using a band join. For example:

```
SELECT /*+ USE_BAND(e1 e2) */
  e1.last_name
  || ' has salary between 100 less and 100 more than '
  || e2.last_name AS "SALARY COMPARISON"
FROM employees e1, employees e2
WHERE e1.salary BETWEEN e2.salary - 100 AND e2.salary + 100;
```

The order the tables are listed in the `USE_BAND` hint does not specify a join order. To hint a specific join order, the `LEADING` hint is required.

USE_CONCAT Hint



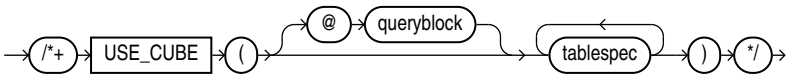
(See Specifying a Query Block in a Hint)

The `USE_CONCAT` hint instructs the optimizer to transform combined `OR`-conditions in the `WHERE` clause of a query into a compound query using the `UNION ALL` set operator. Without this hint, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them. The `USE_CONCAT` hint overrides the cost consideration. For example:

```
SELECT /*+ USE_CONCAT */ *
  FROM employees e
  WHERE manager_id = 108
     OR department_id = 110;
```

> **See Also:**
>
> The NO_EXPAND Hint , which is the opposite of this hint

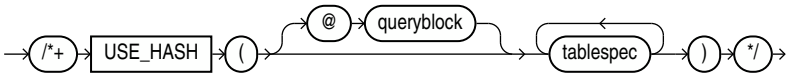
USE_CUBE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

When the right-hand side of the join is a cube, the `USE_CUBE` hint instructs the optimizer to join each specified table with another row source using a cube join. If the optimizer decides not to use the cube join based on statistical analysis, then you can use `USE_CUBE` to override that decision.

USE_HASH Hint



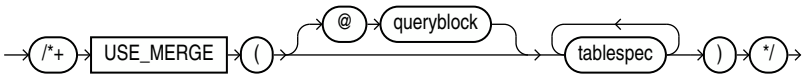
(See Specifying a Query Block in a Hint , *tablespec*::=)

The `USE_HASH` hint instructs the optimizer to join each specified table with another row source using a hash join. For example:

```
SELECT /*+ USE_HASH(l h) */ *
  FROM orders h, order_items l
  WHERE l.order_id = h.order_id
    AND l.order_id > 2400;
```

The order the tables are listed in the `USE_HASH` hint does not specify a join order. To hint a specific join order, the `LEADING` hint is required.

USE_MERGE Hint



(See Specifying a Query Block in a Hint , *tablespec*::=)

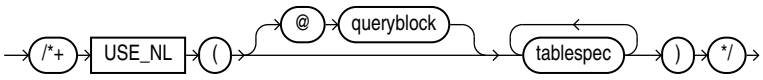
The `USE_MERGE` hint instructs the optimizer to join each specified table with another row source using a sort-merge join. For example:

```
SELECT /*+ USE_MERGE(employees departments) */ *
  FROM employees, departments
  WHERE employees.department_id = departments.department_id;
```

Use of the `USE_NL` and `USE_MERGE` hints is recommended with the `LEADING` and `ORDERED` hints. The optimizer uses those hints when the referenced table is forced to be the inner table of a join. The hints are ignored if the referenced table is the outer table.

USE_NL Hint

The `USE_NL` hint instructs the optimizer to join each specified table to another row source with a nested loops join, using the specified table as the inner table.



(See Specifying a Query Block in a Hint , *tablespec*::=)

Use of the `USE_NL` and `USE_MERGE` hints is recommended with the `LEADING` and `ORDERED` hints. The optimizer uses those hints when the referenced table is forced to be the inner table of a join. The hints are ignored if the referenced table is the outer table.

In the following example, where a nested loop is forced through a hint, `orders` is accessed through a full table scan and the filter condition `l.order_id = h.order_id` is applied to every row. For every row that meets the filter condition, `order_items` is accessed through the index `order_id`.

```
SELECT /*+ USE_NL(l h) */ h.customer_id, l.unit_price * l.quantity
  FROM orders h, order_items l
  WHERE l.order_id = h.order_id;
```

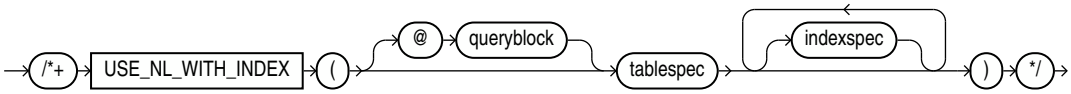
The order the tables are listed in the `USE_NL` hint does not specify a join order. To hint a specific join order, the `LEADING` hint is required.

Example

```
select /*+ LEADING(t2) USE_NL(t1) */ sum(t1.a),sum(t2.a)
from   t1 , t2
where   t1.b = t2.b;
select * from table(dbms_xplan.display_cursor()) ;
```

Adding an `INDEX` hint to the query could avoid the full table scan on `orders`, resulting in an execution plan similar to one used on larger systems, even though it might not be particularly efficient here.

USE_NL_WITH_INDEX Hint



(See Specifying a Query Block in a Hint , *tablespec*::=, *indexspec*::=)

The `USE_NL_WITH_INDEX` hint instructs the optimizer to join the specified table to another row source with a nested loops join using the specified table as the inner table. For example:

```
SELECT /*+ USE_NL_WITH_INDEX(l item_product_ix) */ *
  FROM orders h, order_items l
  WHERE l.order_id = h.order_id
    AND l.order_id > 2400;
```

The following conditions apply:

- If no index is specified, then the optimizer must be able to use some index with at least one join predicate as the index key.
- If an index is specified, then the optimizer must be able to use that index with at least one join predicate as the index key.

Database Objects

Oracle Database recognizes objects that are associated with a particular schema and objects that are not associated with any particular schema, as described in the sections that follow.

Schema Objects

A **schema** is a collection of logical structures of data, or schema objects. A schema is owned by a database user and has the same name as that user. Each user owns a single schema.

Schema objects can be created and manipulated with SQL and include the following types of objects:

- Analytic views
- Attribute dimensions
- Clusters
- Constraints
- Database links
- Database triggers
- Dimensions
- External procedure libraries
- Hierarchies
- Index-organized tables
- Indexes
- Indextypes
- Java classes
- Java resources
- Java sources
- Join groups
- Materialized views
- Materialized view logs
- Mining models
- Object tables
- Object types
- Object views
- Operators
- Packages
- Property Graphs
- Sequences
- Stored functions
- Stored procedures
- Synonyms
- Tables
- Views
- Zone maps

Nonschema Objects

Other types of objects are also stored in the database and can be created and manipulated with SQL but are not contained in a schema:

- Contexts
- Directories
- Editions
- Flashback archives
- Lockdown profiles
- Profiles
- Restore points
- Roles
- Rollback segments
- Tablespaces

Tablespace sets
Unified audit policies
Users

In this reference, each type of object is described in the section devoted to the statement that creates the database object. These statements begin with the keyword `CREATE`. For example, for the definition of a cluster, see CREATE CLUSTER .

> **See Also:**
>
> *Oracle Database Concepts* for an overview of database objects

You must provide names for most types of database objects when you create them. These names must follow the rules listed in the sections that follow.

Database Object Names and Qualifiers

Some database objects are made up of parts that you can or must name, such as the columns in a table or view, index and table partitions and subpartitions, integrity constraints on a table, and objects that are stored within a package, including procedures and stored functions. This section provides:

- Rules for naming database objects and database object location qualifiers
- Guidelines for naming database objects and qualifiers

> **Note:**
>
> Oracle uses system-generated names beginning with "`SYS_`" for implicitly generated database objects and subobjects, and names beginning with "`ORA_`" for some Oracle-supplied objects. Oracle discourages you from using these prefixes in the names you explicitly provide to your database objects and subobjects to avoid possible conflict in name resolution.

Database Object Naming Rules

Every database object has a name. In a SQL statement, you represent the name of an object with a **quoted identifier** or a **nonquoted identifier**.

- A quoted identifier begins and ends with double quotation marks ("). If you name a schema object using a quoted identifier, then you must use the double quotation marks whenever you refer to that object.
- A nonquoted identifier is not surrounded by any punctuation.

You must use double quotation marks (") for schema names that begin with numbers or special characters.

You can use either quoted or nonquoted identifiers to name any database object. However, database names, global database names, database link names, disk group names, and pluggable database (PDB) names are always case insensitive and are stored as uppercase. If you specify such names as quoted identifiers, then the quotation marks are silently ignored.

> **See Also:**
>
> CREATE USER for additional rules for naming users and passwords

> **Note:**
>
> Oracle does not recommend using quoted identifiers for database object names. These quoted identifiers are accepted by SQL*Plus, but they may not be valid when using other tools that manage database objects.

The following list of rules applies to both quoted and nonquoted identifiers unless otherwise indicated:

1. The maximum length of identifier names depends on the value of the `COMPATIBLE` initialization parameter.
   - **If COMPATIBLE is set to a value of 12.2 or higher,** then names must be from 1 to 128 bytes long with these exceptions:
     - Names of databases are limited to 8 bytes.
     - Names of disk groups, pluggable databases (PDBs), rollback segments, tablespaces, and tablespace sets are limited to 30 bytes.
     - From Release 21c onwards names of pluggable databases are limited to 64 bytes.

     If an identifier includes multiple parts separated by periods, then each attribute can be up to 128 bytes long. Each period separator, as well as any surrounding double quotation marks, counts as one byte. For example, suppose you identify a column like this:

     ```
     "schema"."table"."column"
     ```

     The schema name can be 128 bytes, the table name can be 128 bytes, and the column name can be 128 bytes. Each of the quotation marks and periods is a single-byte character, so the total length of the identifier in this example can be up to 392 bytes.
   - **If COMPATIBLE is set to a value lower than 12.2,** then names must be from 1 to 30 bytes long with these exceptions:
     - Names of databases are limited to 8 bytes.
     - Names of database links can be as long as 128 bytes.

     If an identifier includes multiple parts separated by periods, then each attribute can be up to 30 bytes long. Each period separator, as well as any surrounding double quotation marks, counts as one byte. For example, suppose you identify a column like this:

     ```
     "schema"."table"."column"
     ```

     The schema name can be 30 bytes, the table name can be 30 bytes, and the column name can be 30 bytes. Each of the quotation marks and periods is a single-byte character, so the total length of the identifier in this example can be up to 98 bytes.
2. Nonquoted identifiers cannot be Oracle SQL reserved words. Quoted identifiers can be reserved words, although this is not recommended.

Depending on the Oracle product you plan to use to access a database object, names might be further restricted by other product-specific reserved words.

> **Note:**
>
> The reserved word `ROWID` is an exception to this rule. You cannot use the uppercase word `ROWID`, either quoted or nonquoted, as a column name. However, you can use the uppercase word as a quoted identifier that is not a column name, and you can use the word with one or more lowercase letters (for example, "`Rowid`" or "`rowid`") as any quoted identifier, including a column name.

> **See Also:**
>
> - Oracle SQL Reserved Words for a listing of all Oracle SQL reserved words
> - The manual for a specific product, such as *Oracle Database PL/SQL Language Reference*, for a list of the reserved words of that product

3. The Oracle SQL language contains other words that have special meanings. These words include data types, schema names, function names, the dummy system table `DUAL`, and keywords (the uppercase words in SQL statements, such as `DIMENSION`, `SEGMENT`, `ALLOCATE`, `DISABLE`, and so forth). These words are not reserved. However, Oracle uses them internally in specific ways. Therefore, if you use these words as names for objects and object parts, then your SQL statements may be more difficult to read and may lead to unpredictable results.

   In particular, do not use words beginning with `SYS_` or `ORA_` as schema object names, and do not use the names of SQL built-in functions for the names of schema objects or user-defined functions.

   > **See Also:**
   >
   > - Oracle SQL Keywords for information how to obtain a list of keywords
   > - Data Types , About SQL Functions , and Selecting from the DUAL Table

4. You should use characters from the ASCII repertoire in database names, global database names, and database link names, because these characters provide optimal compatibility across different platforms and operating systems. You must use only characters from the ASCII repertoire in the names of common users, common roles, and common profiles in a multitenant container database (CDB).
5. You can include multibyte characters in passwords.
6. Nonquoted identifiers must begin with an alphabetic character from your database character set. Quoted identifiers can begin with any character.
7. Nonquoted identifiers can only contain alphanumeric characters from your database character set and the underscore (_), dollar sign ($), and pound sign (#). Database links can also contain periods (.) and "at" signs (@). Oracle strongly discourages you from using $ and # in nonquoted identifiers.

Quoted identifiers can contain any characters and punctuations marks as well as spaces. However, neither quoted nor nonquoted identifiers can contain double quotation marks or the null character (`\0`).

8. Within a namespace, no two objects can have the same name.

   The following schema objects share one namespace:

   - Packages
   - Private synonyms
   - Sequences
   - Stand-alone procedures
   - Stand-alone stored functions
   - Tables
   - User-defined operators
   - User-defined types
   - Views

   Each of the following schema objects has its own namespace:

   - Clusters
   - Constraints
   - Database triggers
   - Dimensions
   - Indexes
   - Materialized views (When you create a materialized view, the database creates an internal table of the same name. This table has the same namespace as the other tables in the schema. Therefore, a schema cannot contain a table and a materialized view of the same name.)
   - Private database links

   Because tables and sequences are in the same namespace, a table and a sequence in the same schema cannot have the same name. However, tables and indexes are in different namespaces. Therefore, a table and an index in the same schema can have the same name.

   Each schema in the database has its own namespaces for the objects it contains. This means, for example, that two tables in different schemas are in different namespaces and can have the same name.

   Each of the following nonschema objects also has its own namespace:

   - Editions
   - Parameter files (`PFILE`s) and server parameter files (`SPFILE`s)
   - Profiles
   - Public database links
   - Public synonyms
   - Tablespaces
   - User roles

Because the objects in these namespaces are not contained in schemas, these namespaces span the entire database.

9. Nonquoted identifiers are not case sensitive. Oracle interprets them as uppercase. Quoted identifiers are case sensitive.

   By enclosing names in double quotation marks, you can give the following names to different objects in the same namespace:

   ```
   "employees"
   "Employees"
   "EMPLOYEES"
   ```

   Note that Oracle interprets the following names the same, so they cannot be used for different objects in the same namespace:

   ```
   employees
   EMPLOYEES
   "EMPLOYEES"
   ```

10. When Oracle stores or compares identifiers in uppercase, the uppercase form of each character in the identifiers is determined by applying the uppercasing rules of the database character set. Language-specific rules determined by the session setting `NLS_SORT` are not considered. This behavior corresponds to applying the SQL function `UPPER` to the identifier rather than the function `NLS_UPPER`.

    The database character set uppercasing rules can yield results that are incorrect when viewed as being in a certain natural language. For example, small letter sharp s ("ß"), used in German, does not have an uppercase form according to the database character set uppercasing rules. It is not modified when an identifier is converted into uppercase, while the expected uppercase form in German is the sequence of two characters capital letter S ("SS"). Similarly, the uppercase form of small letter i, according to the database character set uppercasing rules, is capital letter I. However, the expected uppercase form in Turkish and Azerbaijani is capital letter I with dot above.

    The database character set uppercasing rules ensure that identifiers are interpreted the same in any linguistic configuration of a session. If you want an identifier to look correctly in a certain natural language, then you can quote it to preserve the lowercase form or you can use the linguistically correct uppercase form whenever you use that identifier.

11. Columns in the same table or view cannot have the same name. However, columns in different tables or views can have the same name.

12. Procedures or functions contained in the same package can have the same name, if their arguments are not of the same number and data types. Creating multiple procedures or functions with the same name in the same package with different arguments is called **overloading** the procedure or function.

13. Tablespace names are case sensitive, unlike other identifiers that are limited to 30 bytes.

Schema Object Naming Examples

The following examples are valid schema object names:

```
last_name
horse
hr.hire_date
"EVEN THIS & THAT!"
a_very_long_and_valid_name
```

All of these examples adhere to the rules listed in Database Object Naming Rules .

Schema Object Naming Guidelines

Here are several helpful guidelines for naming objects and their parts:

- Use full, descriptive, pronounceable names (or well-known abbreviations).
- Use consistent naming rules.
- Use the same name to describe the same entity or attribute across tables.

When naming objects, balance the objective of keeping names short and easy to use with the objective of making names as descriptive as possible. When in doubt, choose the more descriptive name, because the objects in the database may be used by many people over a period of time. Your counterpart ten years from now may have difficulty understanding a table column with a name like `pmdd` instead of `payment_due_date`.

Using consistent naming rules helps users understand the part that each table plays in your application. One such rule might be to begin the names of all tables belonging to the `FINANCE` application with `fin_`.

Use the same names to describe the same things across tables. For example, the department number columns of the sample `employees` and `departments` tables are both named `department_id`.

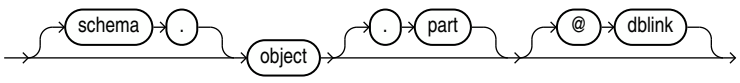
Syntax for Schema Objects and Parts in SQL Statements

This section tells you how to refer to schema objects and their parts in the context of a SQL statement. This section shows you:

- The general syntax for referring to an object
- How Oracle resolves a reference to an object
- How to refer to objects in schemas other than your own
- How to refer to objects in remote databases
- How to refer to table and index partitions and subpartitions

The following diagram shows the general syntax for referring to an object or a part:

***database_object_or_part*::=**



(*dblink*::=)

where:

- *object* is the name of the object.
- *schema* is the schema containing the object. The schema qualifier lets you refer to an object in a schema other than your own. You must be granted privileges to refer to objects in other schemas. If you omit *schema*, then Oracle assumes that you are referring to an object in your own schema.

  Only schema objects can be qualified with *schema*. Schema objects are shown with list item 8. Nonschema objects, also shown with list item 8, cannot be qualified with *schema*

because they are not schema objects. An exception is public synonyms, which can optionally be qualified with "PUBLIC". The quotation marks are required.

- *part* is a part of the object. This identifier lets you refer to a part of a schema object, such as a column or a partition of a table. Not all types of objects have parts.
- *dblink* applies only when you are using the Oracle Database distributed functionality. This is the name of the database containing the object. The *dblink* qualifier lets you refer to an object in a database other than your local database. If you omit *dblink*, then Oracle assumes that you are referring to an object in your local database. Not all SQL statements allow you to access objects on remote databases.

You can include spaces around the periods separating the components of the reference to the object, but it is conventional to omit them.

How Oracle Database Resolves Schema Object References

When you refer to an object in a SQL statement, Oracle considers the context of the SQL statement and locates the object in the appropriate namespace. After locating the object, Oracle performs the operation specified by the statement on the object. If the named object cannot be found in the appropriate namespace, then Oracle returns an error.

The following example illustrates how Oracle resolves references to objects within SQL statements. Consider this statement that adds a row of data to a table identified by the name `departments`:

```
INSERT INTO departments
  VALUES (280, 'ENTERTAINMENT_CLERK', 206, 1700);
```

Based on the context of the statement, Oracle determines that `departments` can be:

- A table in your own schema
- A view in your own schema
- A private synonym for a table or view
- A public synonym

Oracle always attempts to resolve an object reference within the namespaces in your own schema before considering namespaces outside your schema. In this example, Oracle attempts to resolve the name `departments` as follows:

1. First, Oracle attempts to locate the object in the namespace in your own schema containing tables, views, and private synonyms. If the object is a private synonym, then Oracle locates the object for which the synonym stands. This object could be in your own schema, another schema, or on another database. The object could also be another synonym, in which case Oracle locates the object for which this synonym stands.
2. If the object is in the namespace, then Oracle attempts to perform the statement on the object. In this example, Oracle attempts to add the row of data to `departments`. If the object is not of the correct type for the statement, then Oracle returns an error. In this example, `departments` must be a table, view, or a private synonym resolving to a table or view. If `departments` is a sequence, then Oracle returns an error.
3. If the object is not in any namespace searched in thus far, then Oracle searches the namespace containing public synonyms. If the object is in that namespace, then Oracle attempts to perform the statement on it. If the object is not of the correct type for the statement, then Oracle returns an error. In this example, if `departments` is a public synonym for a sequence, then Oracle returns an error.

If a public synonym has any dependent tables or user-defined types, then you cannot create an object with the same name as the synonym in the same schema as the dependent objects.

If a synonym does not have any dependent tables or user-defined types, then you can create an object with the same name in the same schema as the dependent objects. Oracle invalidates any dependent objects and attempts to revalidate them when they are next accessed.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for information about how PL/SQL resolves identifier names

References to Objects in Other Schemas

To refer to objects in schemas other than your own, prefix the object name with the schema name:

```
schema.object
```

For example, this statement drops the `employees` table in the sample schema `hr`:

```
DROP TABLE hr.employees;
```

References to Objects in Remote Databases

To refer to objects in databases other than your local database, follow the object name with the name of the database link to that database. A database link is a schema object that causes Oracle to connect to a remote database to access an object there. This section tells you:

- How to create database links
- How to use database links in your SQL statements

Creating Database Links

You create a database link with the statement CREATE DATABASE LINK . The statement lets you specify this information about the database link:

- The name of the database link
- The database connect string to access the remote database
- The username and password to connect to the remote database

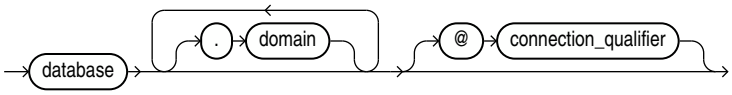
Oracle stores this information in the data dictionary.

Database Link Names

When you create a database link, you must specify its name. Database link names are different from names of other types of objects. They can be as long as 128 bytes and can contain periods (.) and the "at" sign (@).

The name that you give to a database link must correspond to the name of the database to which the database link refers and the location of that database in the hierarchy of database names. The following syntax diagram shows the form of the name of a database link:

***dblink*::=**



where:

- *database* should specify the *name* portion of the global name of the remote database to which the database link connects. This global name is stored in the data dictionary of the remote database. You can see this name in the GLOBAL_NAME data dictionary view.
- *domain* should specify the *domain* portion of the global name of the remote database to which the database link connects. If you omit *domain* from the name of a database link, then Oracle qualifies the database link name with the domain of your local database as it currently exists in the data dictionary.
- *connection_qualifier* lets you further qualify a database link. Using connection qualifiers, you can create multiple database links to the same database. For example, you can use connection qualifiers to create multiple database links to different instances of the Oracle Real Application Clusters that access the same database.

> **See Also:**
>
> *Oracle Database Administrator's Guide*for more information on connection qualifiers

The combination *database.domain* is sometimes called the **service name**.

> **See Also:**
>
> *Oracle Database Net Services Administrator's Guide*

Username and Password

Oracle uses the username and password to connect to the remote database. The username and password for a database link are optional.

Database Connect String

The database connect string is the specification used by Oracle Net to access the remote database. For information on writing database connect strings, see the Oracle Net documentation for your specific network protocol. The database connect string for a database link is optional.

References to Database Links

Database links are available only if you are using Oracle distributed functionality. When you issue a SQL statement that contains a database link, you can specify the database link name in one of these forms:

- The complete database link name as stored in the data dictionary, including the *database*, *domain*, and optional *connection_qualifier* components.
- The *partial* database link name is the *database* and optional *connection_qualifier* components, but not the *domain* component.

Oracle performs these tasks before connecting to the remote database:

1. If the database link name specified in the statement is partial, then Oracle expands the name to contain the domain of the local database as found in the global database name stored in the data dictionary. (You can see the current global database name in the GLOBAL_NAME data dictionary view.)
2. Oracle first searches for a private database link in your own schema with the same name as the database link in the statement. Then, if necessary, it searches for a public database link with the same name.
   - Oracle always determines the username and password from the first matching database link (either private or public). If the first matching database link has an associated username and password, then Oracle uses it. If it does not have an associated username and password, then Oracle uses your current username and password.
   - If the first matching database link has an associated database string, then Oracle uses it. Otherwise Oracle searches for the next matching (public) database link. If no matching database link is found, or if no matching link has an associated database string, then Oracle returns an error.
3. Oracle uses the database string to access the remote database. After accessing the remote database, if the value of the GLOBAL_NAMES parameter is true, then Oracle verifies that the *database.domain* portion of the database link name matches the complete global name of the remote database. If this condition is true, then Oracle proceeds with the connection, using the username and password chosen in Step 2. If not, Oracle returns an error.
4. If the connection using the database string, username, and password is successful, then Oracle attempts to access the specified object on the remote database using the rules for resolving object references and referring to objects in other schemas discussed earlier in this section.

You can disable the requirement that the *database.domain* portion of the database link name must match the complete global name of the remote database by setting to FALSE the initialization parameter GLOBAL_NAMES or the GLOBAL_NAMES parameter of the ALTER SYSTEM or ALTER SESSION statement.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information on remote name resolution

References to Partitioned Tables and Indexes

Tables and indexes can be partitioned. When partitioned, these schema objects consist of a number of parts called **partitions**, all of which have the same logical attributes. For example, all partitions in a table share the same column and constraint definitions, and all partitions in an index share the same index columns.

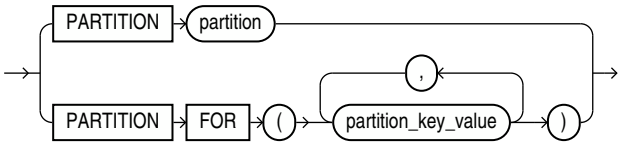
Partition-extended and subpartition-extended names let you perform some partition-level and subpartition-level operations, such as deleting all rows from a partition or subpartition, on only one partition or subpartition. Without extended names, such operations would require that you specify a predicate (`WHERE` clause). For range- and list-partitioned tables, trying to phrase a partition-level operation with a predicate can be cumbersome, especially when the range partitioning key uses more than one column. For hash partitions and subpartitions, using a predicate is more difficult still, because these partitions and subpartitions are based on a system-defined hash function.

Partition-extended names let you use partitions as if they were tables. An advantage of this method, which is most useful for range-partitioned tables, is that you can build partition-level access control mechanisms by granting (or revoking) privileges on these views to (or from) other users or roles. To use a partition as a table, create a view by selecting data from a single partition, and then use the view as a table.

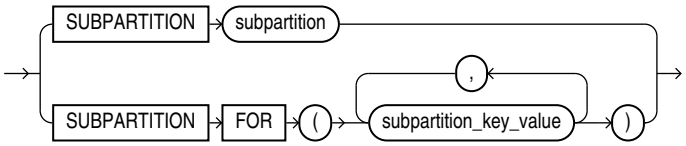
Syntax

You can specify partition-extended or subpartition-extended table names in any SQL statement in which the *`partition_extended_name`* or *`subpartition_extended_name`* element appears in the syntax.

***partition_extended_name*::=**

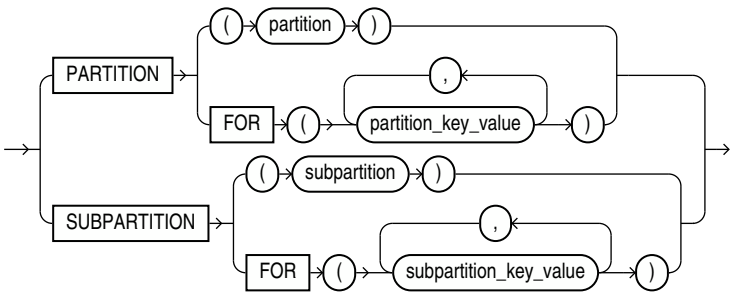


***subpartition_extended_name*::=**



The DML statements `INSERT`, `UPDATE`, and `DELETE` and the `ANALYZE` statement require parentheses around the partition or subpartition name. This small distinction is reflected in the *`partition_extension_clause`*:

***partition_extension_clause*::=**



In *partition_extended_name*, *subpartition_extended_name*, and *partition_extension_clause*, the PARTITION FOR and SUBPARTITION FOR clauses let you refer to a partition without using its name. They are valid with any type of partitioning and are especially useful for interval partitions. Interval partitions are created automatically as needed when data is inserted into a table.

For the respective *partition_key_value* or *subpartition_key_value*, specify one value for each partitioning key column. For multicolumn partitioning keys, specify one value for each partitioning key. For composite partitions, specify one value for each partitioning key, followed by one value for each subpartitioning key. All partitioning key values are comma separated. For interval partitions, you can specify only one *partition_key_value*, and it must be a valid NUMBER or datetime value. Your SQL statement will operate on the partition or subpartitions that contain the values you specify.

> **See Also:**
>
> The CREATE TABLE INTERVAL Clause for more information on interval partitions

Restrictions on Extended Names

Currently, the use of partition-extended and subpartition-extended table names has the following restrictions:

- No remote tables: A partition-extended or subpartition-extended table name cannot contain a database link (dblink) or a synonym that translates to a table with a dblink. To use remote partitions and subpartitions, create a view at the remote site that uses the extended table name syntax and then refer to the remote view.
- No synonyms: A partition or subpartition extension must be specified with a base table. You cannot use synonyms, views, or any other objects.
- The PARTITION FOR and SUBPARTITION FOR clauses are not valid for DDL operations on views.
- In the PARTITION FOR and SUBPARTITION FOR clauses, you cannot specify the keywords DEFAULT or MAXVALUE or a bind variable for the *partition_key_value* or *subpartition_key_value*.
- In the PARTITION and SUBPARTITION clauses, you cannot specify a bind variable for the partition or subpartition name.

Example

In the following statement, sales is a partitioned table with partition sales_q1_2000. You can create a view of the single partition sales_q1_2000, and then use it as if it were a table. This example deletes rows from the partition.

```
CREATE VIEW Q1_2000_sales AS
  SELECT *
    FROM sales PARTITION (SALES_Q1_2000);

DELETE FROM Q1_2000_sales
  WHERE amount_sold < 0;
```

References to Object Type Attributes and Methods

To refer to object type attributes or methods in a SQL statement, you must fully qualify the reference with a table alias. Consider the following example from the sample schema `oe`, which contains a type `cust_address_typ` and a table `customers` with a `cust_address` column based on the `cust_address_typ`:

```
CREATE TYPE cust_address_typ
  OID '82A4AF6A4CD1656DE034080020E0EE3D'
  AS OBJECT
    (street_address    VARCHAR2(40),
     postal_code       VARCHAR2(10),
     city              VARCHAR2(30),
     state_province    VARCHAR2(10),
     country_id        CHAR(2));
/
CREATE TABLE customers
  (customer_id        NUMBER(6),
   cust_first_name    VARCHAR2(20) CONSTRAINT cust_fname_nn NOT NULL,
   cust_last_name     VARCHAR2(20) CONSTRAINT cust_lname_nn NOT NULL,
   cust_address       cust_address_typ,
. . .
```

In a SQL statement, reference to the `postal_code` attribute must be fully qualified using a table alias, as illustrated in the following example:

```
SELECT c.cust_address.postal_code
  FROM customers c;

UPDATE customers c
  SET c.cust_address.postal_code = '14621-2604'
  WHERE c.cust_address.city = 'Rochester'
    AND c.cust_address.state_province = 'NY';
```

To reference a member method that does not accept arguments, you must provide empty parentheses. For example, the sample schema `oe` contains an object table `categories_tab`, based on `catalog_typ`, which contains the member function `getCatalogName`. In order to call this method in a SQL statement, you must provide empty parentheses as shown in this example:

```
SELECT TREAT(VALUE(c) AS catalog_typ).getCatalogName() "Catalog Type"
  FROM categories_tab c
  WHERE category_id = 90;

Catalog Type
-----------------------------------
online catalog
```