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:

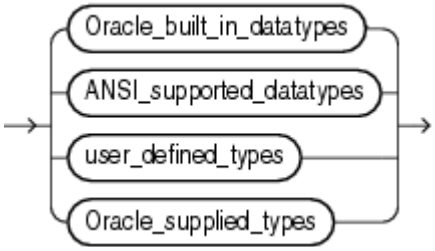
- Oracle Built-in Data Types
- ANSI, DB2, and SQL/DS Data Types
- User-Defined Types
- Oracle-Supplied Types
- Data Type Comparison Rules
- Data Conversion

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 Programmer's Guide*, and *Pro*C/C++ Programmer's Guide*.

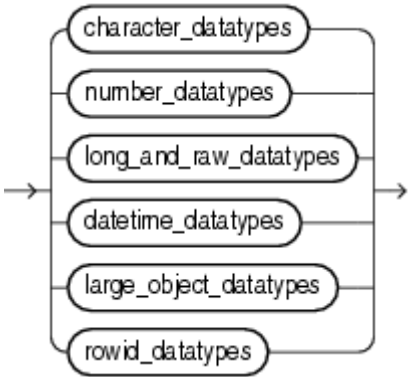
***datatypes*::=**



Description of the illustration "datatypes.gif"

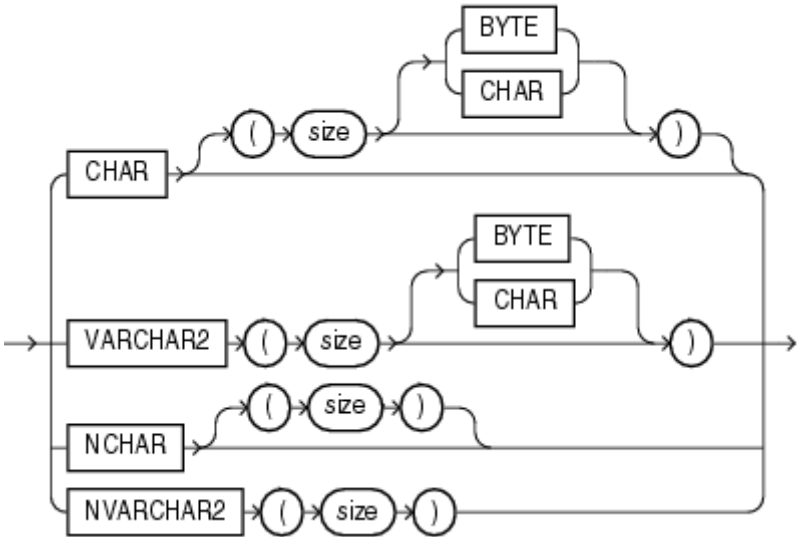
The Oracle built-in data types appear in the figures that follows. For descriptions, refer to "Oracle Built-in Data Types".

Oracle_built_in_datatypes::=



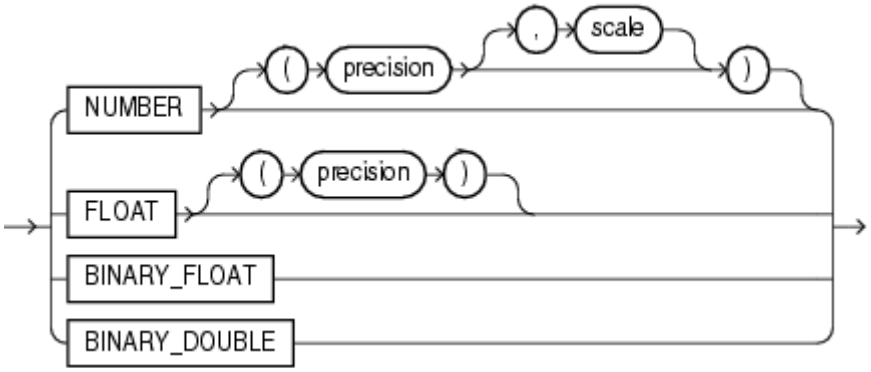
Description of the illustration "oracle_built_in_datatypes.gif"

character_datatypes::=



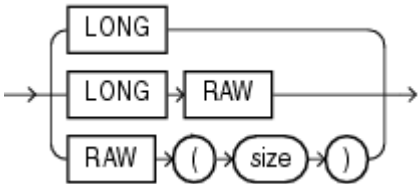
Description of the illustration "character_datatypes.gif"

number_datatypes::=



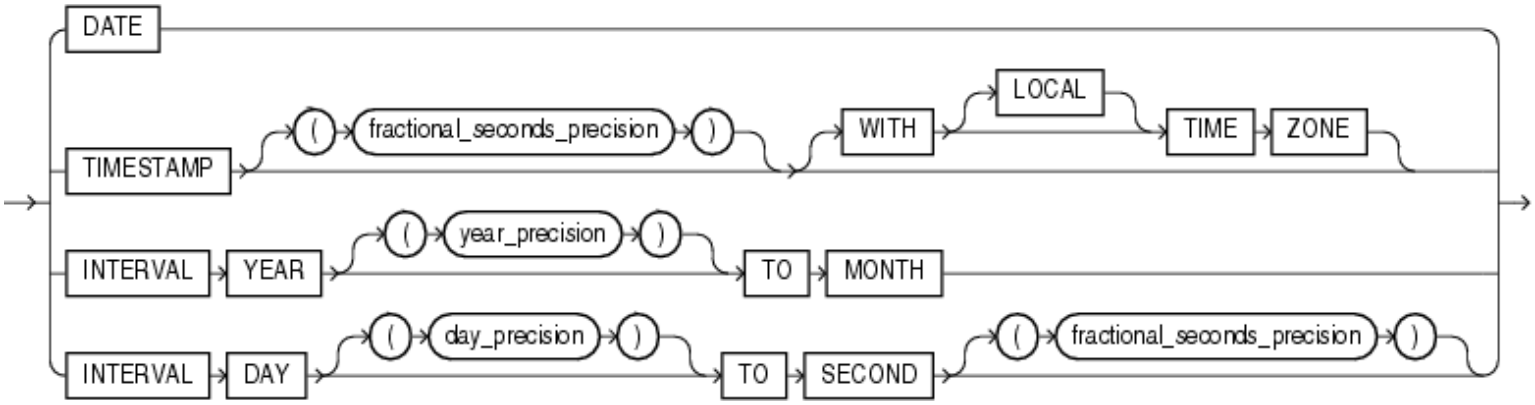
Description of the illustration "number_datatypes.gif"

long_and_raw_datatypes::=



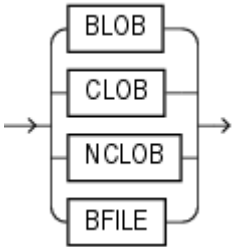
Description of the illustration "long_and_raw_datatypes.gif"

datetime_datatypes::=



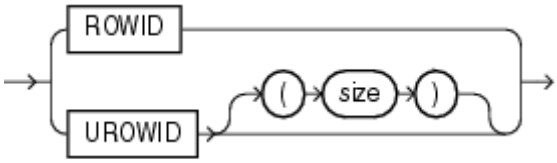
Description of the illustration "datetime_datatypes.gif"

large_object_datatypes::=



Description of the illustration "large_object_datatypes.gif"

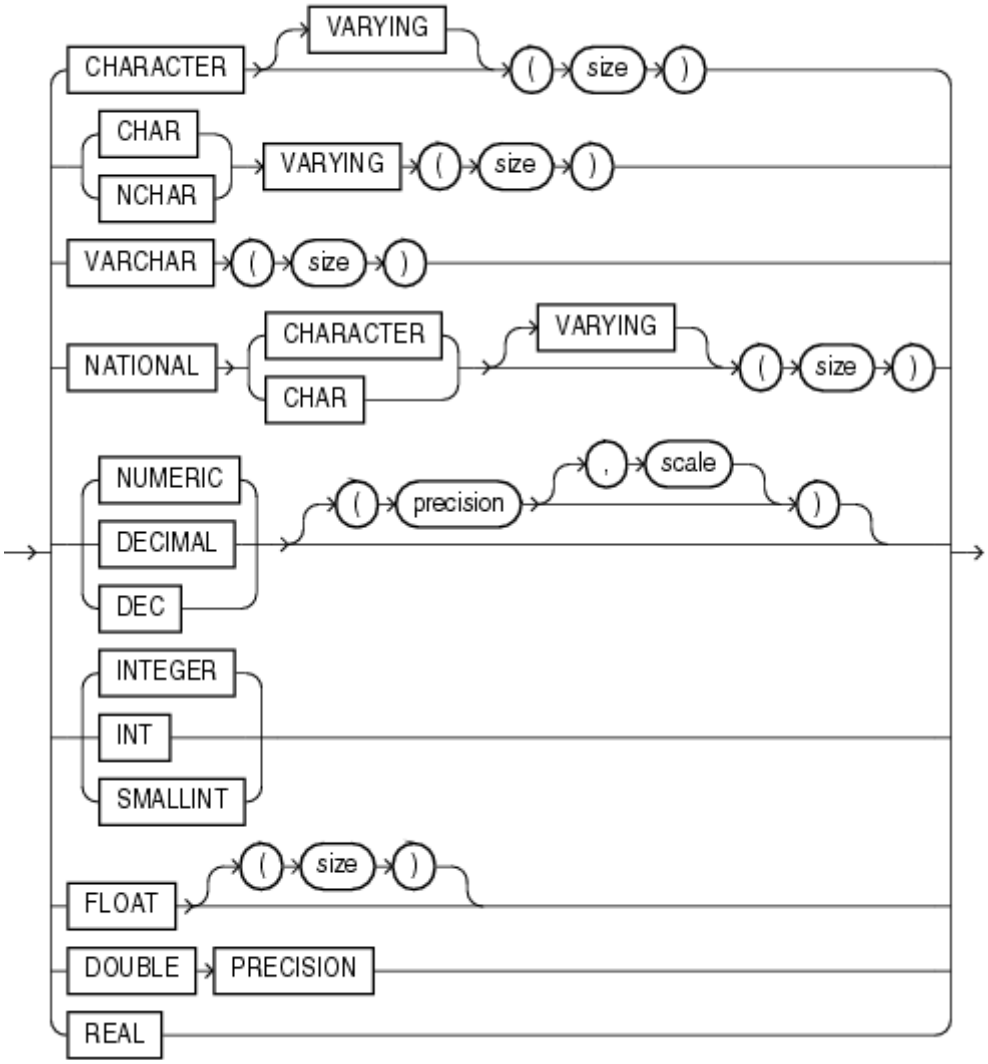
rowid_datatypes::=



Description of the illustration "rowid_datatypes.gif"

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::=

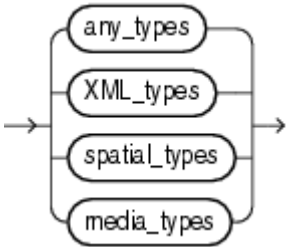


Description of the illustration "ansi_supported_datatypes.gif"

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

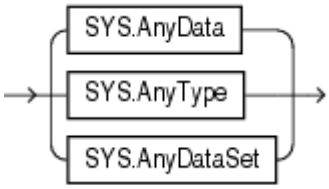
The Oracle-supplied data types appear in the figures that follows. For descriptions, refer to "Oracle-Supplied Types".

Oracle_supplied_types::=



Description of the illustration "oracle_supplied_types.gif"

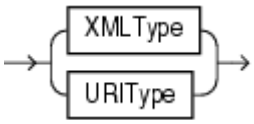
any_types::=



Description of the illustration "any_types.gif"

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

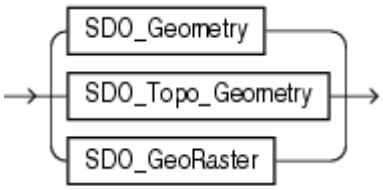
XML_types::=



Description of the illustration "xml_types.gif"

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

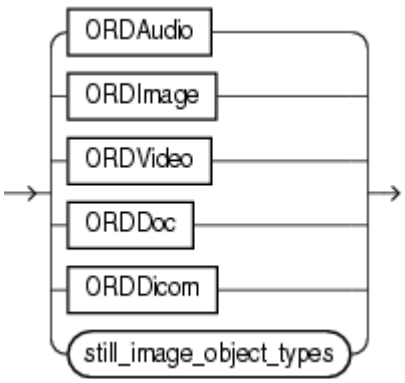
spatial_types::=



Description of the illustration "spatial_types.gif"

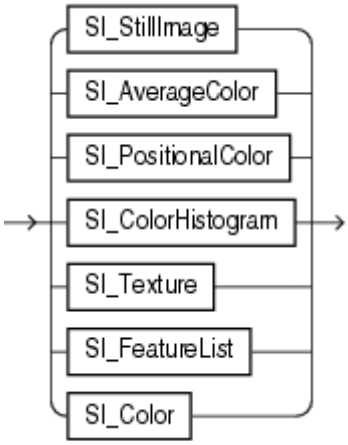
For descriptions of the spatial types, refer to "Spatial Types".

media_types::=



Description of the illustration "media_types.gif"

***still_image_object_types*::=**



Description of the illustration "still_image_object_types.gif"

For descriptions of the media types, refer to "Media Types".

Oracle Built-in Data Types

The table that follows summarizes Oracle built-in data types. Refer to the syntax in the preceding sections for the syntactic elements. The codes listed for the data types are used internally by Oracle Database. The data type code of a column or object attribute is returned by 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.

181	`TIMESTAMP`[(*`fractional_seconds_precision`*)] `WITHTIME 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 format is determined explicitly by the `NLS_TIMESTAMP_FORMAT` parameter or implicitly by the `NLS_TERRITORY`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`*)] `WITHLOCAL TIMEZONE`	All values of `TIMESTAMP WITH TIMEZONE`, 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)] TOSECOND[(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`.

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.

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
- NVARCHAR2 Data Type
- VARCHAR2 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. Oracle ensures that all values stored in a `CHAR` column have the length specified by *`size`*. 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.

The default length for a `CHAR` column is 1 byte and the maximum allowed is 2000 bytes. A 1-byte string can be inserted into a `CHAR(10)` column, but the string is blank-padded to 10 bytes before it is stored.

When you create a table with a `CHAR` column, by default you supply the column length in bytes. The `BYTE` qualifier is the same as the default. If you use the `CHAR` qualifier, for example `CHAR(10 CHAR)`, then you supply the column length in characters. A character is technically a code point of the database character set. Its size can range from 1 byte to 4 bytes, depending on the database character set. The `BYTE` and `CHAR` qualifiers override the semantics specified by the `NLS_LENGTH_SEMANTICS` parameter, which has a default of byte semantics. For performance reasons, Oracle recommends that you use the `NLS_LENGTH_SEMANTICS` parameter to set length semantics and that you use the `BYTE` and `CHAR` qualifiers only when necessary to override the parameter.

To ensure proper data conversion between databases 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 is a Unicode-only data type. When you create a table with an `NCHAR` column, you define the column length in characters. You define the national character set when you create your database.

The maximum length of a column is determined by the national character set definition. Width specifications of character data type `NCHAR` refer to the number of characters. The maximum column size allowed is 2000 bytes.

If you insert a value that is shorter than the column length, then Oracle blank-pads the value to column length. You cannot insert a `CHAR` value into an `NCHAR`column, nor can you insert an `NCHAR` value into a `CHAR` column.

The following example compares the `translated_description` column of the `pm.product_descriptions` table with a national character set string:

```
SELECT translated_description
  FROM product_descriptions
  WHERE translated_name = N'LCD Monitor 11/PM';
```

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

NVARCHAR2 Data Type

The `NVARCHAR2` data type is a Unicode-only data type. When you create a table with an `NVARCHAR2` column, you supply the maximum number of characters it can hold. Oracle subsequently stores each value in the column exactly as you specify it, provided the value does not exceed the maximum length of the column.

The maximum length of the column is determined by the national character set definition. Width specifications of character data type `NVARCHAR2` refer to the number of characters. The maximum column size allowed 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_SIZEinitialization parameter and the internal storage mechanisms for extended data types.

> **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. When you create a VARCHAR2 column, you supply the maximum number of bytes or characters of data that it can hold. Oracle subsequently stores each value in the column exactly as you specify it, provided the value does not exceed the maximum length of the column. If you try to insert a value that exceeds the specified length, then Oracle returns an error.

You must specify a maximum length for a VARCHAR2 column. This maximum must be at least 1 byte, although the actual string stored is permitted to be a zero-length string (''). You can use the CHAR qualifier, for example VARCHAR2(10 CHAR), to give the maximum length in characters instead of bytes. A character is technically a code point of the database character set. You can use the BYTE qualifier, for example VARCHAR2(10 BYTE), to explicitly give the maximum length in bytes. If no explicit qualifier is included in a column or attribute definition when a database object with this column or attribute is created, then the length semantics are determined by the value of the NLS_LENGTH_SEMANTICS parameter of the session creating the object. Independently of the maximum length in characters, the length of VARCHAR2data cannot exceed:

- 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_SIZEinitialization parameter and the internal storage mechanisms for extended data types.

Oracle compares VARCHAR2 values using nonpadded 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.

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 x 10^{-130} to but not including 1.0 x 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".

h

- 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 2-10, "Implicit Type Conversion Matrix" 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

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 LONGfunctionality 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 Programmer'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 TOMONTH 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

Datetime Field	Valid Values for Datetime	Valid Values for INTERVAL
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 Microsoft Windows* 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. For more information, see http://aa.usno.navy.mil/faq/docs/millennium.php.

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 2-17, "Matching Character Data and Format Models with the FX Format Model Modifier" 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 TIMEZONE`) 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;
```

h

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

- Oracle performs all timestamp arithmetic in UTC time. For `TIMESTAMP WITHLOCAL 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	–

Operand & Operator	DATE	TIMESTAMP	INTERVAL	Numeric
-	–	–	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 2-17. `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 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

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 Programmer'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_LOBpackage 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. NCLOBobjects 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 CHUNKparameter 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

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.

> **Note:**
>
> 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.

> **Notes:**
>
> - Oracle strongly discourages the use of BasicFiles LOBs as a storage mechanism. BasicFiles LOBs not only impose restrictions on the capabilities of extended data type columns, but the BasicFiles data type is planned to be deprecated in a future release.
> - 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.

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`. You can also create tables and clusters that contain actual columns having the `ROWID` data type. Oracle Database does not guarantee that the values of such columns are valid rowids. Refer to Chapter 3, "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 2-6 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)`

ANSI SQL Data Type	Oracle Data Type
`NUMERIC[(p,s)]` `DECIMAL[(p,s)]` (**Note 1**)	`NUMBER(p,s)`

`INTEGER`
`INT`
`SMALLINT`
	`NUMBER(p,0)`

`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 2-7 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
- Media 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 `CLOB`), 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 and Graph Developer's Guide*, *Oracle Spatial and Graph Topology Data Model and Network Data Model Graph Developer's Guide*, and *Oracle Spatial and Graph GeoRaster Developer's Guide* for information on the full implementation of the spatial data types and guidelines for using them

Media Types

Oracle Multimedia uses object types, similar to Java or C++ classes, to describe multimedia data. An instance of these object types consists of attributes, including metadata and the media data, and methods. The Multimedia data types are created in the `ORDSYS` schema. Public synonyms exist for all the data types, so you can access them without specifying the schema name.

Oracle Multimedia provides the following object types:

- `ORDAudio`

  Supports the storage and management of audio data.

- `ORDDicom`

  Supports the storage and management of Digital Imaging and Communications in Medicine (DICOM), the format universally recognized as the standard for medical imaging.

- `ORDDoc`

  Supports storage and management of any type of media data, including audio, image and video data. Use this type when you want all media to be stored in a single column.

- `ORDImage`

Supports the storage and management of image data.

- `ORDVideo`

  Supports the storage and management of video data.

The following data types provide compliance with the ISO-IEC 13249-5 Still Image standard, commonly referred to as SQL/MM StillImage:

- `SI_AverageColor`

  Represents a feature that characterizes an image by its average color.

- `SI_Color`

  Encapsulates color values.

- `SI_ColorHistogram`

  Represents a feature that characterizes an image by the relative frequencies of the colors exhibited by samples of the raw image.

- `SI_FeatureList`

  A list containing up to four of the image features represented by the preceding object types (`SI_AverageColor`, `SI_ColorHistogram`, `SI_PositionalColor`, and `SI_Texture`), where each feature is associated with a feature weight.

- `SI_PositionalColor`

  Given an image divided into `n` by `m` rectangles, the `SI_PositionalColor`object type represents the feature that characterizes an image by the `n` by `m`most significant colors of the rectangles.

- `SI_StillImage`

  Represents digital images with inherent image characteristics such as height, width, and format.

- `SI_Texture`

  Represents a feature that characterizes an image by the size of repeating items (coarseness), brightness variations (contrast), and predominant direction (directionality).

See Also:

- *Oracle Multimedia DICOM Developer's Guide* for information on the `ORDDicom` object type
- *Oracle Multimedia Reference* for information on all other object types listed in this section