9

Using SQL Data Types in Database Applications

This chapter explains how to choose the correct SQL data types for database columns that you create for your database applications.

Topics:

- Using the Correct and Most Specific Data Type
- Representing Character Data
- Representing Numeric Data
- Representing Date and Time Data
- Representing Specialized Data
- Identifying Rows by Address
- Displaying Metadata for SQL Operators and Functions

Note:

Oracle precompilers recognize, in embedded SQL programs, data types other than SQL and PL/SQL data types. These **external data types** are associated with host variables.

See Also:

- *Oracle Database SQL Language Reference* for information about data type conversion
- PL/SQL Data Types
- Data Types
- Overview of Precompilers

9.1 Using the Correct and Most Specific Data Type

Using the correct and most specific data type for each database column that you create for your database application increases data integrity, decreases storage requirements, and improves performance.

Topics:

- How the Correct Data Type Increases Data Integrity

- How the Most Specific Data Type Decreases Storage Requirements
- How the Correct Data Type Improves Performance

9.1.1 How the Correct Data Type Increases Data Integrity

The correct data type increases data integrity by acting as a constraint. For example, if you use a datetime data type for a column of dates, then only dates can be stored in that column. However, if you use a character or numeric data type for the column, then eventually someone will store a character or numeric value that does not represent a date. You could write code to prevent this problem, but it is more efficient to use the correct data type. Therefore, store characters in character data types, numbers in numeric data types, and dates and times in datetime data types.

See Also:

Maintaining Data Integrity in Database Applications, for information about data integrity and constraints

9.1.2 How the Most Specific Data Type Decreases Storage Requirements

In addition to using the correct data type, use the most specific length or precision; for example:

- When creating a `VARCHAR2` column intended for strings of at most *n* characters, specify `VARCHAR2(`*`n`*`)`.
- When creating a column intended for integers, use the data type `NUMBER(38)` rather than `NUMBER`.

Besides acting as constraints and thereby increasing data integrity, length and precision affect storage requirements.

If you give every column the maximum length or precision for its data type, then your application needlessly allocates many megabytes of RAM. For example, suppose that a query selects 10 `VARCHAR2(4000)` columns and a bulk fetch operation returns 100 rows. The RAM that your application must allocate is 10 x 4,000 x 100—almost 4 MB. In contrast, if the column length is 80, the RAM that your application must allocate is 10 x 80 x 100—about 78 KB. This difference is significant for a single query, and your application will process many queries concurrently. Therefore, your application must allocate the 4 MB or 78 KB of RAM *for each connection*.

Therefore, do not give a column the maximum length or precision for its data type only because you might need to increase that property later. If you must change a column after creating it, then use the `ALTER TABLE` statement. For example, to increase the length of a column, use:

```
ALTER TABLE table_name MODIFY column_name VARCHAR2(larger_number)
```

Note:

The maximum length of the `VARCHAR2`, `NVARCHAR2`, and `RAW` data types is 32,767 bytes if the `MAX_STRING_SIZE` initialization parameter is `EXTENDED`.

See Also:

- *Oracle Database SQL Language Reference* for more information about `ALTERTABLE`
- *Oracle Database SQL Language Reference* for more information about extended data types

9.1.3 How the Correct Data Type Improves Performance

The correct data type improves performance because the incorrect data type can result in the incorrect execution plan.

Example 9-1 performs the same conceptual operation—selecting rows whose dates are between December 31, 2000 and January 1, 2001—for three columns with different data types and shows the execution plan for each query. In the three execution plans, compare Rows (cardinality), Cost, and Operation.

Example 9-1 Performance Comparison of Three Data Types

Create a table that stores the same dates in three columns: `str_date`, with data type `VARCHAR2`; `date_date`, with data type `DATE`, and `number_date`, with data type `NUMBER`:

```
CREATE TABLE t (str_date, date_date, number_date, data)
AS
SELECT TO_CHAR(dt+rownum,'yyyymmdd')              str_date,    -- VARCHAR2
       dt+rownum                                  date_date,   -- DATE
       TO_NUMBER(TO_CHAR(dt+rownum,'yyyymmdd'))   number_date, -- NUMBER
       RPAD('*',45,'*')                           data
FROM (SELECT TO_DATE('01-jan-1995', 'dd-mm-yyyy') dt
      FROM all_objects)
ORDER BY DBMS_RANDOM.VALUE
/
```

Create an index on each column:

```
CREATE INDEX t_str_date_idx ON t(str_date);
CREATE INDEX t_date_date_idx ON t(date_date);
CREATE INDEX t_number_date_idx ON t(number_date);
```

Gather statistics for the table:

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS (
    'HR',
    'T',
    method_opt => 'for all indexed columns size 254',
    cascade => TRUE
  );
```

```
END;
/
```

Show the execution plans of subsequent SQL statements (SQL*Plus command):

```
SET AUTOTRACE ON EXPLAIN
```

Select the rows for which the dates in `str_date` are between December 31, 2000 and January 1, 2001:

```
SELECT * FROM t WHERE str_date BETWEEN '20001231' AND '20010101'
ORDER BY str_date;
```

Result and execution plan:

```
STR_DATE DATE_DATE NUMBER_DATE DATA
-------- --------- ----------- --------------------------------------------
20001231 31-DEC-00    20001231 ********************************************
20010101 01-JAN-01    20010101 ********************************************

2 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 948745535

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |   236 | 11092 |   216   (8)| 00:00:01 |
|   1 |  SORT ORDER BY     |      |   236 | 11092 |   216   (8)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| T    |   236 | 11092 |   215   (8)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("STR_DATE"<='20010101' AND "STR_DATE">='20001231')
```

Select the rows for which the dates in `number_date` are between December 31, 2000 and January 1, 2001:

```
SELECT * FROM t WHERE number_date BETWEEN 20001231 AND 20010101;
ORDER BY str_date;
```

Result and execution plan:

```
STR_DATE DATE_DATE NUMBER_DATE DATA
-------- --------- ----------- --------------------------------------------
20001231 31-DEC-00    20001231 ********************************************
20010101 01-JAN-01    20010101 ********************************************

2 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 948745535

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT   |      |   234 | 10998 |   219  (10)| 00:00:01 |
|   1 |  SORT ORDER BY     |      |   234 | 10998 |   219  (10)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| T    |   234 | 10998 |   218   (9)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("NUMBER_DATE"<=20010101 AND "NUMBER_DATE">=20001231)
```

Select the rows for which the dates in `date_date` are between December 31, 2000 and January 1, 2001:

```
SELECT * FROM t WHERE date_date
  BETWEEN TO_DATE('20001231','yyyymmdd')
  AND     TO_DATE('20010101','yyyymmdd');
  ORDER BY str_date;
```

Result and execution plan (reformatted to fit the page):

```
STR_DATE DATE_DATE NUMBER_DATE DATA
-------- --------- ----------- --------------------------------------------
20001231 31-DEC-00    20001231 *********************************************
20010101 01-JAN-01    20010101 *********************************************

2 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 2411593187

--------------------------------------------------------------------------------

| Id  | Operation                             | Name            | Rows  | Bytes |
|   1 |  SORT ORDER BY                        |                 |     1 |    47 |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| T               |     1 |    47 |
|*  3 |    INDEX RANGE SCAN                   | T_DATE_DATE_IDX |     1 |       |
|   0 | SELECT STATEMENT                      |                 |     1 |    47 |

------------------------
 Cost (%CPU)| Time     |

     4  (25)| 00:00:01 |
     4  (25)| 00:00:01 |
     3   (0)| 00:00:01 |
     2   (0)| 00:00:01 |

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("DATE_DATE">=TO_DATE(' 2000-12-31 00:00:00',
   'syyyy-mm-dd hh24:mi:ss') AND

  "DATE_DATE"<=TO_DATE(' 2001-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

Performance improved for the final query because, for the `DATE` data type, the optimizer could determine that there was only one day between December 31, 2000 and January 1, 2001. Therefore, it performed an index range scan, which is faster than a full table scan.

See Also:

- EXPLAIN PLAN Statement
- *Oracle Database SQL Tuning Guide* for more information about Full Table Scans
- *Oracle Database SQL Tuning Guide* for more information about Index Range Scans

9.2 Representing Character Data

Table 9-1 summarizes the SQL data types that store character data.

Table 9-1 SQL Character Data Types

Data Types	Values Stored
`CHAR`	Fixed-length character literals
`VARCHAR2`	Variable-length character literals
`NCHAR`	Fixed-length Unicode character literals
`NVARCHAR2`	Variable-length Unicode character literals
`CLOB`	Single-byte and multibyte character strings of up to (4 gigabytes - 1) * (the value obtained from `DBMS_LOB.GETCHUNKSIZE`)
`NCLOB`	Single-byte and multibyte Unicode character strings of up to (4 gigabytes - 1) * (the value obtained from `DBMS_LOB.GETCHUNKSIZE`)
`LONG`	Variable-length character data of up to 2 gigabytes - 1. Provided only for backward compatibility.

Note:

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.

When choosing between `CHAR` and `VARCHAR2`, consider:

- Space usage

  Oracle Database blank-pads values stored in `CHAR` columns but not values stored in `VARCHAR2` columns. Therefore, `VARCHAR2` columns use space more efficiently than `CHAR` columns.

- Performance

  Because of the blank-padding difference, a full table scan on a large table containing `VARCHAR2` columns might read fewer data blocks than a full table scan on a table containing the same data stored in `CHAR` columns. If your application often performs full table scans on large tables containing character data, then you might be able to improve performance by storing data in `VARCHAR2` columns rather than in `CHAR` columns.

- Comparison semantics

  When you need ANSI compatibility in comparison semantics, use the `CHAR` data type. When trailing blanks are important in string comparisons, use the `VARCHAR2` data type.

For a client/server application, if the character set on the client side differs from the character set on the server side, then Oracle Database converts `CHAR`, `VARCHAR2`, and `LONG` data from the database character set (determined by the `NLS_LANGUAGE` parameter) to the character set defined for the user session.

See Also:

- *Oracle Database SQL Language Reference* for more information about comparison semantics for these data types
- Large Objects (LOBs) for more information about `CLOB` and `NCLOB` data types
- LONG and LONG RAW Data Types for more information about `LONG` data type

9.3 Representing Numeric Data

The SQL data types that store numeric data are `NUMBER`, `BINARY_FLOAT`, and `BINARY_DOUBLE`.

The `NUMBER` data type stores real numbers in either a fixed-point or floating-point format. `NUMBER` offers up to 38 decimal digits of precision. In a `NUMBER` column, you can store positive and negative numbers of magnitude 1 x 10^{-130} through 9.99 x10^{125}, and 0. All Oracle Database platforms support `NUMBER` values.

The `BINARY_FLOAT` and `BINARY_DOUBLE` data types store floating-point numbers in the single-precision (32-bit) IEEE 754 format and the double-precision (64-bit) IEEE 754 format, respectively. High-precision values use less space when stored as `BINARY_FLOAT` and `BINARY_DOUBLE` than when stored as `NUMBER`. Arithmetic operations on floating-point numbers are usually faster for `BINARY_FLOAT` and `BINARY_DOUBLE` values than for `NUMBER` values.

In client interfaces that Oracle Database supports, arithmetic operations on `BINARY_FLOAT` and `BINARY_DOUBLE` values are performed by the native instruction set that the hardware vendor supplies. The term **native floating-point data type** includes `BINARY_FLOAT` and `BINARY_DOUBLE` data types and all implementations of these types in supported client interfaces.

Native floating-point data types conform substantially with the Institute of Electrical and Electronics Engineers (IEEE) Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-1985 (IEEE754).

Note:

Oracle recommends using `BINARY_FLOAT` and `BINARY_DOUBLE` instead of `FLOAT`, a subtype of `NUMBER` .

Topics:

- Floating-Point Number Components

- Floating-Point Number Formats
- Representing Special Values with Native Floating-Point Data Types
- Comparing Native Floating-Point Values
- Arithmetic Operations with Native Floating-Point Data Types
- Conversion Functions for Native Floating-Point Data Types
- Client Interfaces for Native Floating-Point Data Types

See Also:

Oracle Database SQL Language Reference for more information about data types

9.3.1 Floating-Point Number Components

The formula for a floating-point value is:

$(-1)^{sign}.significand.base^{exponent}$

For example, the floating-point value 4.31 is represented:

$(-1)^{0}.431.10^{-2}$

The components of the preceding representation are:

Component Name	Component Value
Sign	0
Significand	431
Base	10
Exponent	-2

9.3.2 Floating-Point Number Formats

A floating-point number format specifies how the components of a floating-point number are represented, thereby determining the range and precision of the values that the format can represent. The **range** is the interval bounded by the smallest and largest values and the **precision** is the number of significant digits. Both range and precision are finite. If a floating-point number is too precise for a given format, then the number is rounded.

How the number is rounded depends on the base of its format, which can be either decimal or binary. A number stored in decimal format is rounded to the nearest decimal place (for example, 1000, 10, or 0.01). A number stored in binary format is rounded to the nearest binary place (for example, 1024, 512, or 1/64).

`NUMBER` values are stored in decimal format. For calculations that need decimal rounding, use the `NUMBER` data type.

Native floating-point values are stored in binary format.

Table 9-2 shows the range and precision of the IEEE 754 single- and double-precision formats and Oracle Database `NUMBER`. Range limits are expressed as positive numbers, but they also apply to absolute values of negative numbers. (The notation "*number* e *exponent*" means *number* * $10^{exponent}$.)

Table 9-2 Range and Precision of Floating-Point Data Types

Range and Precision	Single-precision 32-bit[1]	Double-precision 64-bit[1]	Oracle Database NUMBER Data Type
Maximum positive normal number	3.40282347e+38	1.7976931348623157e+308	< 1.0e126
Minimum positive normal number	1.17549435e-38	2.2250738585072014e-308	1.0e-130
Maximum positive subnormal number	1.17549421e-38	2.2250738585072009e-308	not applicable
Minimum positive subnormal number	1.40129846e-45	4.9406564584124654e-324	not applicable
Precision (decimal digits)	6 - 9	15 - 17	38 - 40

1 These numbers are from the *IEEE Numerical Computation Guide*.

9.3.2.1 Binary Floating-Point Formats

This formula determines the value of a floating-point number that uses a binary format:

$(-1)^{sign}\ 2^{E}\ (bit_0\ bit_1\ bit_2\ \ldots\ bit_{p-1})$

Table 9-3 describes the components of the preceding formula.

Table 9-3 Binary Floating-Point Format Components

Component	Component Value
`sign`	0 or 1
`E` (exponent)	For single-precision (32-bit) data type, an integer from -126 through 127. For double-precision (64-bit) data type, an integer from -1022 through 1023.
bit_i	0 or 1. (The bit sequence represents a number in base 2.)
`p` (precision)	For single-precision data type, 24. For double-precision data type, 53.

The leading bit of the significand, b_0, must be set (1), except for subnormal numbers (explained later). Therefore, the leading bit is not stored, and a binary format provides *n* bits of precision while storing only *n*-1 bits. The IEEE 754 standard defines the in-memory formats for single-precision and double-precision data types, as Table 9-4 shows.

Table 9-4 Summary of Binary Format Storage Parameters

Data Type	Sign Bit	Exponent Bits	Significand Bits	Total Bits
Single-precision	1	8	24 (23 stored)	32
Double-precision	1	11	53 (52 stored)	64

Note:

Oracle Database does not support the extended single- and double-precision formats that the IEEE 754 standard defines.

A significand whose leading bit is set is called **normalized**. The IEEE 754 standard defines **subnormal numbers** (also called **denormal numbers**) that are too small to represent with normalized significands. If the significand of a subnormal number were normalized, then its exponent would be too large. Subnormal numbers preserve this property: If *x*-*y*==0.0 (using floating-point subtraction), then *x*==*y*.

9.3.3 Representing Special Values with Native Floating-Point Data Types

The IEEE 754 standard supports the special values shown in Table 9-5.

Table 9-5 Special Values for Native Floating-Point Formats

Value	Meaning
`+INF`	Positive infinity
`-INF`	Negative infinity
`+0`	Positive zero
`-0`	Negative zero
`NaN`	Not a number

Each value in Table 9-5 is represented by a specific bit pattern, except `NaN`. `NaN`, the result of any undefined operation, is represented by many bit patterns. Some of these bits patterns have the sign bit set and some do not, but the sign bit has no meaning.

The IEEE 754 standard distinguishes between quiet `NaN`s (which do not raise additional exceptions as they propagate through most operations) and signaling `NaN`s (which do). The IEEE 754 standard specifies action for when exceptions are enabled and action for when they are disabled.

In Oracle Database, exceptions cannot be enabled. Oracle Database acts as the IEEE 754 standard specifies for when exceptions are disabled. In particular, Oracle Database does not distinguish between quiet and signaling `NaN`s. You can use Oracle Call Interface (OCI) to retrieve `NaN` values from Oracle Database, but whether a retrieved `NaN` value is signaling or quiet depends on the client platform and is beyond the control of Oracle Database.

The IEEE 754 standard defines these classes of special values:

- Zero
- Subnormal
- Normal
- Infinity
- `NaN`

The values in each class in the preceding list are larger than the values in the classes that precede it in the list (ignoring signs), except `NaN`. `NaN` is unordered with other classes of special values and with itself.

In Oracle Database:

- All `NaN`s are quiet.
- Any non-`NaN` value < `NaN`
- Any `NaN` == any other `NaN`
- All `NaN`s are converted to the same bit pattern.
- -0 is converted to +0.
- IEEE 754 exceptions are not raised.

See Also:

Oracle Database SQL Language Reference for information about floating-point conditions, which let you determine whether an expression is infinite or is the undefined result of an operation (is not a number or `NaN`).

9.3.4 Comparing Native Floating-Point Values

When comparing numeric expressions, Oracle Database uses numeric precedence to determine whether the condition compares `NUMBER`, `BINARY_FLOAT`, or `BINARY_DOUBLE` values.

Comparisons ignore the sign of zero (`-0` equals `+0`).

See Also:

- *Oracle Database SQL Language Reference* for more information about Numeric Precedence
- *Oracle Database SQL Language Reference* for more information about Comparison Conditions

9.3.5 Arithmetic Operations with Native Floating-Point Data Types

IEEE 754 does not require floating-point arithmetic to be exactly reproducible. Therefore, results of operations can be delivered to a destination that uses a range greater than the range that the operands of the operation use.

You can compute the result of a double-precision multiplication at an extended double-precision destination, but the result must be rounded as if the destination were single-precision or double-precision. The range of the result (that is, the number of bits used for the exponent) can use the range supported by the wider (extended double-precision) destination; however, this might cause a double-rounding error in which the least significant bit of the result is incorrect.

This situation can occur only for double-precision multiplication and division on hardware that implements the IA-32 and IA-64 instruction set architecture. Therefore, except for this case, arithmetic for these data types is reproducible across platforms. When the result of a computation is `NaN`, all platforms produce a value for which `IS NAN` is true. However, all platforms do not have to use the same bit pattern.

See Also:

Oracle Database SQL Language Reference for general information about arithmetic operations

9.3.6 Conversion Functions for Native Floating-Point Data Types

Oracle Database defines functions that convert between floating-point and other data types, including string formats that use decimal precision (but precision might be lost during the conversion). For example:

- `TO_BINARY_DOUBLE`, described in *Oracle Database SQL Language Reference*
- `TO_BINARY_FLOAT`, described in *Oracle Database SQL Language Reference*
- `TO_CHAR`, described in *Oracle Database SQL Language Reference*
- `TO_NUMBER`, described in *Oracle Database SQL Language Reference*

Oracle Database can raise exceptions during conversion. The IEEE 754 standard defines these exceptions:

- Invalid
- Inexact
- Divide by zero
- Underflow
- Overflow

However, Oracle Database does not raise these exceptions for native floating-point data types. Generally, operations that raise exceptions produce the values described in Table 9-6.

Table 9-6 Values Resulting from Exceptions

Exception	Value
Underflow	`0`
Overflow	`-INF`, `+INF`
Invalid Operation	`NaN`
Divide by Zero	`-INF`, `+INF`, `NaN`
Inexact	Any value – rounding was performed

9.3.7 Client Interfaces for Native Floating-Point Data Types

Oracle Database supports native floating-point data types in these client interfaces:

- SQL and PL/SQL

  Support for `BINARY_FLOAT` and `BINARY_DOUBLE` includes their use as attributes of Abstract Data Types (ADTs), which you create with the SQL statement `CREATE TYPE` (fully described in *Oracle Database PL/SQL Language Reference*).
- Oracle Call Interface (OCI)

For information about using `BINARY_FLOAT` and `BINARY_DOUBLE` with OCI, see *Oracle Call Interface Programmer's Guide*.

- Oracle C++ Call Interface (OCCI)

  For information about using `BINARY_FLOAT` with OCCI, see *Oracle C++ Call Interface Programmer's Guide*.

  For information about using `BINARY_DOUBLE` with OCCI, see *Oracle C++ Call Interface Programmer's Guide*.

- Pro*C/C++ precompiler

  To use `BINARY_FLOAT` and `BINARY_DOUBLE`, set the Pro*C/C++ precompiler command line option `NATIVE_TYPES` to `YES` when you compile your application. For information about the `NATIVE_TYPES` option, see *Pro*C/C++ Programmer's Guide*.

- Oracle JDBC

  For information about using `BINARY_FLOAT` and `BINARY_DOUBLE` with Oracle JDBC, see *Oracle Database JDBC Developer's Guide*.

9.4 Representing Date and Time Data

Oracle Database stores `DATE` and `TIMESTAMP` (**datetime**) data in a binary format that represents the century, year, month, day, hour, minute, second, and optionally, fractional seconds and timezones.

Table 9-7 summarizes the SQL datetime data types.

Table 9-7 SQL Datetime Data Types

Date Type	Usage
`DATE`	For storing datetime values in a table—for example, dates of jobs.
`TIMESTAMP`	For storing datetime values that are precise to fractional seconds—for example, times of events that must be compared to determine the order in which they occurred.
`TIMESTAMP WITH TIME ZONE`	For storing datetime values that must be gathered or coordinated across geographic regions.
`TIMESTAMP WITH LOCAL TIME ZONE`	For storing datetime values when the time zone is insignificant—for example, in an application that schedules teleconferences, where participants see the start and end times for their own time zone. Appropriate for two-tier applications in which you want to display dates and times that use the time zone of the client system. Usually inappropriate for three-tier applications, because data displayed in a web browser is formatted according to the time zone of the web server, not the time zone of the browser. The web server is the database client, so its local time is used.
`INTERVAL YEAR TO MONTH`	For storing the difference between two datetime values, where only the year and month are significant—for example, to set a reminder for a date 18 months in the future, or check whether 6 months have elapsed since a particular date.

Table 9-7 (Cont.) SQL Datetime Data Types

Date Type	Usage
`INTERVAL DAY TO SECOND`	For storing the precise difference between two datetime values—for example, to set a reminder for a time 36 hours in the future or to record the time between the start and end of a race. To represent long spans of time with high precision, use a large number of days.

See Also:

- *Oracle Call Interface Programmer's Guide* for more information about Oracle Database internal date types
- *Oracle Database SQL Language Reference* for more information about date and time data types

Topics:

- Displaying Current Date and Time
- Inserting and Displaying Dates
- Inserting and Displaying Times
- Arithmetic Operations with Datetime Data Types
- Conversion Functions for Datetime Data Types
- Importing_ Exporting_ and Comparing Datetime Types

9.4.1 Displaying Current Date and Time

The simplest way to display the current date and time is:

```
SELECT TO_CHAR(SYSDATE, format_model) FROM DUAL
```

The default format model depends on the initialization parameter `NLS_DATE_FORMAT`.

The standard Oracle Database default date format is `DD-MON-RR`. The `RR` datetime format element lets you store 20th century dates in the 21st century by specifying only the last two digits of the year. For example, in the datetime format `DD-MON-YY`, `13-NOV-54` refers to the year 1954 in a query issued between 1950 and 2049, but to the year 2054 in a query issued between 2050 and 2149.

Note:

For program correctness and to avoid problems with SQL injection and dynamic SQL, Oracle recommends specifying a format model for every datetime value.

The simplest way to display the current date and time using a format model is:

```
SELECT TO_CHAR(SYSDATE, format_model) FROM DUAL
```

Example 9-2 uses `TO_CHAR` with a format model to display `SYSDATE` in a format with the qualifier BC or AD. (By default, `SYSDATE` is displayed without this qualifier.)

Example 9-2 Displaying Current Date and Time

```
SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY BC') NOW FROM DUAL;
```

Result:

```
NOW
-----------------------
18-MAR-2009 AD

1 row selected.
```

Tip:

When testing code that uses `SYSDATE`, it can be helpful to set `SYSDATE` to a constant. Do this with the initialization parameter `FIXED_DATE`.

See Also:

- *Oracle Database SQL Language Reference* for more information about `SYSDATE`
- *Oracle Database Globalization Support Guide* for information about `NLS_DATE_FORMAT`
- *Oracle Database SQL Language Reference* for more information about `TO_CHAR`
- *Oracle Database SQL Language Reference* for information about datetime format models
- *Oracle Database SQL Language Reference* for more information about the `RR` datetime format element
- *Oracle Database Reference*for more information about `FIXED_DATE`

9.4.2 Inserting and Displaying Dates

When you display and insert dates, Oracle recommends using the `TO_CHAR` and `TO_DATE` functions, respectively, with datetime format models.

Example 9-3 creates a table with a `DATE` column and inserts a date into it, specifying a format model. Then the example displays the date with and without specifying a format model.

Example 9-3 Inserting and Displaying Dates

Create table:

```
DROP TABLE dates;
CREATE TABLE dates (d DATE);
```

Insert date specified into table, specifying a format model:

```
INSERT INTO dates VALUES (TO_DATE('OCT 27, 1998', 'MON DD, YYYY'));
```

Display date without specifying a format model:

```
SELECT d FROM dates;
```

Result:

```
D
---------
27-OCT-98

1 row selected.
```

Display date, specifying a format model:

```
SELECT TO_CHAR(d, 'YYYY-MON-DD') D FROM dates;
```

Result:

```
D
--------------------
1998-OCT-27

1 row selected.
```

Caution:

Be careful when using the `YY` datetime format element, which indicates the year in the current century. For example, in the 21st century, the format `DD-MON-YY`, `31-DEC-92` is December 31, 2092 (not December 31, 1992, as you might expect). To store 20th century dates in the 21st century by specifying only the last two digits of the year, use the `RR` datetime format element (the default).

See Also:

- *Oracle Database Globalization Support Guide* for information about `NLS_DATE_FORMAT`
- *Oracle Database SQL Language Reference* for more information about `TO_CHAR`
- *Oracle Database SQL Language Reference* for more information about `TO_DATE`
- *Oracle Database SQL Language Reference* for information about datetime format models
- *Oracle Database SQL Language Reference* for more information about the `RR` datetime format element

9.4.3 Inserting and Displaying Times

When you display and insert times, Oracle recommends using the `TO_CHAR` and `TO_DATE` functions, respectively, with datetime format models.

In a DATE column:

- The default time is 12:00:00 A.M. (midnight).

  The default time applies to any value in the column that has no time portion, either because none was specified or because the value was truncated.
- The default day is the first day of the current month.

  The default date applies to any value in the column that has no date portion, because none was specified.

Example 9-4 creates a table with a DATE column and inserts three dates into it, specifying a different format model for each date. The first format model has both date and time portions, the second has no time portion, and the third has no date portion. Then the example displays the three dates, specifying a format model that includes both date and time portions.

Example 9-4   Inserting and Displaying Dates and Times

Create table:

```
DROP TABLE birthdays;
CREATE TABLE birthdays (name VARCHAR2(20), day DATE);
```

Insert three dates, specifying a different format model for each date:

```
INSERT INTO birthdays (name, day)
VALUES ('Annie',
        TO_DATE('13-NOV-92 10:56 A.M.','DD-MON-RR HH:MI A.M.')
       );

INSERT INTO birthdays (name, day)
VALUES ('Bobby',
        TO_DATE('5-APR-02','DD-MON-RR')
       );

INSERT INTO birthdays (name, day)
VALUES ('Cindy',
        TO_DATE('8:25 P.M.','HH:MI A.M.')
       );
```

Display both date and time portions of stored datetime values:

```
SELECT name,
       TO_CHAR(day, 'Mon DD, RRRR') DAY,
       TO_CHAR(day, 'HH:MI A.M.') TIME
FROM birthdays;
```

Result:

```
NAME                 DAY                  TIME
-------------------- -------------------- ----------
Annie                Nov 13, 1992         10:56 A.M.
Bobby                Apr 05, 2002         12:00 A.M.
Cindy                Nov 01, 2010         08:25 P.M.

3 rows selected.
```

9.4.4 Arithmetic Operations with Datetime Data Types

The results of arithmetic operations on datetime values are determined by the rules in Oracle Database SQL Language Reference.

SQL has many datetime functions that you can use in datetime expressions. For example, the function `ADD_MONTHS` returns the date that is a specified number of months from a specified date. .

See Also:

- *Oracle Database SQL Language Reference* for the complete list of datetime functions
- *Oracle Database SQL Language Reference*

9.4.5 Conversion Functions for Datetime Data Types

Table 9-8 summarizes the SQL functions that convert to or from datetime data types. For more information about these functions, see *Oracle Database SQL Language Reference*.

Table 9-8 SQL Conversion Functions for Datetime Data Types

Function	Converts ...	To ...
`NUMTODSINTERVAL`	`NUMBER`	`INTERVAL DAY TO SECOND`
`NUMTOYMINTERVAL`	`NUMBER`	`INTERVAL DAY TO MONTH`

`TO_CHAR`	`DATE`
	`TIMESTAMP`
	`TIMESTAMP WITH TIME ZONE`
	`TIMESTAMP WITH LOCAL TIME ZONE`
	`INTERVAL DAY TO SECOND`
	`INTERVAL YEAR TO MONTH`
		`VARCHAR2`

`TO_DATE`	`CHAR`
	`VARCHAR2`
	`NCHAR`
	`NVARCHAR2`
		`DATE`

`TO_DSINTERVAL`	`CHAR`
	`VARCHAR2`
	`NCHAR`
	`NVARCHAR2`
		`INTERVAL DAY TO SECOND`

`TO_TIMESTAMP`	`CHAR`
	`VARCHAR2`
	`NCHAR`
	`NVARCHAR2`
		`TIMESTAMP`

Table 9-8 (Cont.) SQL Conversion Functions for Datetime Data Types

Function	Converts ...	To ...

`TO_TIMESTAMP_TZ`	`CHAR`
	`VARCHAR2`
	`NCHAR`
	`NVARCHAR2`
		`TIMESTAMP WITH TIME ZONE`

`TO_YMINTERVAL`	`CHAR`
	`VARCHAR2`
	`NCHAR`
	`NVARCHAR2`
		`INTERVAL DAY TO MONTH`

9.4.6 Importing, Exporting, and Comparing Datetime Types

You can import, export, and compare `TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` values without worrying about time zone offsets, because the database stores these values in normalized format.

When importing, exporting, and comparing `DATE` and `TIMESTAMP` values, you must adjust them to account for any time zone differences between source and target databases, because the database does not store their time zones.

9.5 Representing Specialized Data

Topics:

- Representing Spatial Data
- Representing Large Amounts of Data
- Representing Searchable Text
- Representing XML Data
- Representing Dynamically Typed Data
- Representing ANSI_ DB2_ and SQL/DS Data

9.5.1 Representing Spatial Data

Spatial data is used by location-enabled applications, geographic information system (GIS) applications, and geoimaging applications.

See Also:

Oracle Database SQL Language Reference for information about representing spatial data in Oracle Database

9.5.2 Representing Large Amounts of Data

For representing large amounts of data, Oracle Database provides:

- Large Objects (LOBs)
- LONG and LONG RAW Data Types (for backward compatibility)

9.5.2.1 Large Objects (LOBs)

Large Objects (LOBs) are data types that are designed to store large amounts of data in a way that lets your application access and manipulate it efficiently.

Table 9-9 summarizes the LOBs.

Table 9-9 Large Objects (LOBs)

Data Type	Description
`BLOB`	**Binary large object** Stores any kind of data in binary format. Typically used for multimedia data such as images, audio, and video.
`CLOB`	**Character large object** Stores string data in the database character set format. Used for large strings or documents that use the database character set exclusively.
`NCLOB`	**National character large object** Stores string data in National Character Set format. Used for large strings or documents in the National Character Set.
`BFILE`	**External large object** Stores a binary file outside the database in the host operating system file system. Applications have read-only access to `BFILE`s. Used for static data that applications do not manipulate, such as image data. Any kind of data (that is, any operating system file) can be stored in a `BFILE`. For example, you can store character data in a `BFILE` and then load the `BFILE` data into a `CLOB`, specifying the character set when loading.

An instance of type `BLOB`, `CLOB`, or `NCLOB` can be either **temporary** (declared in the scope of your application) or **persistent** (created and stored in the database).

See Also:

- *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about using LOBs in application development
- *Oracle Database SQL Language Reference* for more information about LOB functions

9.5.2.2 LONG and LONG RAW Data Types

Note:

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

For more information, see:

Migrating Columns from LONGs to LOBs

`LONG` columns store variable-length character strings containing up to 2 gigabytes - 1 bytes. .

The `LONG RAW` (and `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.

See Also:

Oracle Database SQL Language Reference for more information about data types

9.5.3 Representing JSON Data

A new JSON data type to natively store JSON data in Oracle Database.

Oracle Database stores JSON data type to natively store JSON data.

If you have information stored in JSON format, then you can use the Oracle-supplied type `JSON` type to natively store in the database. Oracle provides indexing, a rich set of packages and operators that can operate on JSON data.

With `JSON` values, you can use:

- `JSON` type standard functions
- `IS_JSON/IS_NOT_JSON` constraints

See Also:

- Oracle Database JSON Developer's Guide for information about Oracle JSON data type and how you can use it to store, generate, manipulate, manage, and query JSON data in the database

9.5.4 Representing Searchable Text

Rather than writing low-level code to do full-text searches, you can use Oracle Text. Oracle Text provides indexing, word and theme searching, and viewing capabilities for text in query applications and document classification applications. You can also use Oracle Text to search XML data.

See Also:

Oracle Text Application Developer's Guide for more information about Oracle Text

9.5.5 Representing XML Data

If you have information stored as files in XML format, or want to store an ADT in XML format, then you can use the Oracle-supplied type `XMLType`.

With `XMLType` values, you can use:

- `XMLType` member functions (see *Oracle XML DB Developer's Guide*).
- SQL XML functions (see *Oracle Database SQL Language Reference*)
- PL/SQL `DBMS_XML` packages (see *Oracle Database PL/SQL Packages and Types Reference*)

See Also:

- *Oracle XML DB Developer’s Guide* for information about Oracle XML DB and how you can use it to store, generate, manipulate, manage, and query XML data in the database
- *Oracle XML Developer's Kit Programmer's Guide*
- *Oracle Database SQL Language Reference* for more information about `XMLType`

9.5.6 Representing Dynamically Typed Data

Some languages allow data types to change at runtime, and some let a program check the type of a variable. For example, C has the `union` keyword and the `void *` pointer, and Java has the `typeof` operator and wrapper types such as `Number`.

In Oracle Database, you can create variables and columns that can hold data of any type and test their values to determine their underlying representation. For example, a single table column can have a numeric value in one row, a string value in another row, and an object in another row.

You can use the Oracle-supplied ADT `SYS.ANYDATA` to represent values of any scalar type or ADT. `SYS.ANYDATA` has methods that accept scalar values of any type, and turn them back into scalars or objects. Similarly, you can use the Oracle-supplied ADT `SYS.ANYDATASET` to represent values of any collection type.

To check and manipulate type information, use the `DBMS_TYPES` package, as in Example 9-5.

With OCI, use the `OCIAnyData` and `OCIAnyDataSet` interfaces.

Example 9-5   Accessing Information in a SYS.ANYDATA Column

```
CREATE OR REPLACE TYPE employee_type AS
  OBJECT (empno NUMBER, ename VARCHAR2(10));
/

DROP TABLE mytab;
CREATE TABLE mytab (id NUMBER, data SYS.ANYDATA);

INSERT INTO mytab (id, data)
VALUES (1, SYS.ANYDATA.ConvertNumber(5));

INSERT INTO mytab (id, data)
VALUES (2, SYS.ANYDATA.ConvertObject(Employee_type(5555, 'john')));

CREATE OR REPLACE PROCEDURE p IS
  CURSOR cur IS SELECT id, data FROM mytab;
  v_id                         mytab.id%TYPE;
  v_data                       mytab.data%TYPE;
  v_type                       SYS.ANYTYPE;
  v_typecode                   PLS_INTEGER;
  v_typename                   VARCHAR2(60);
  v_dummy                      PLS_INTEGER;
  v_n                          NUMBER;
  v_employee                   employee_type;
  non_null_anytype_for_NUMBER exception;
  unknown_typename             exception;
BEGIN
  FOR x IN cur LOOP
    FETCH cur INTO v_id, v_data;
    EXIT WHEN cur%NOTFOUND;

    /* typecode signifies type represented by v_data.
       GetType also produces a value of type SYS.ANYTYPE with methods you
       can call to find precision and scale of a number, length of a
       string, and so on. */

       v_typecode := v_data.GetType (v_type /* OUT */);

    /* Compare typecode to DBMS_TYPES constants to determine type of data
       and decide how to display it. */

    CASE v_typecode
      WHEN DBMS_TYPES.TYPECODE_NUMBER THEN
        IF v_type IS NOT NULL THEN  -- This condition should never happen.
          RAISE non_null_anytype_for_NUMBER;
        END IF;

      -- For each type, there is a Get method.
      v_dummy := v_data.GetNUMBER (v_n /* OUT */);
      DBMS_OUTPUT.PUT_LINE
        (TO_CHAR(v_id) || ': NUMBER = ' || TO_CHAR(v_n) );

       WHEN DBMS_TYPES.TYPECODE_OBJECT THEN
         v_typename := v_data.GetTypeName();
         IF v_typename NOT IN ('HR.EMPLOYEE_TYPE') THEN
           RAISE unknown_typename;
         END IF;
         v_dummy := v_data.GetObject (v_employee /* OUT */);
         DBMS_OUTPUT.PUT_LINE
           (TO_CHAR(v_id) || ': user-defined type = ' || v_typename ||
            ' ( ' || v_employee.empno || ', ' || v_employee.ename || ' )' );
    END CASE;
  END LOOP;
EXCEPTION
  WHEN non_null_anytype_for_NUMBER THEN
    RAISE_Application_Error (-20000,
      'Paradox: the return AnyType instance FROM GetType ' ||
      'should be NULL for all but user-defined types');
  WHEN unknown_typename THEN
    RAISE_Application_Error( -20000, 'Unknown user-defined type ' ||
      v_typename || ' - program written to handle only HR.EMPLOYEE_TYPE');
END;
/

SELECT t.data.gettypename() AS "Type Name" FROM mytab t;
```

Result:

```
Type Name
--------------------------------------------------------------------------------
SYS.NUMBER
HR.EMPLOYEE_TYPE

2 rows selected.
```

See Also:

- *Oracle Database Object-Relational Developer's Guide* for more information about these ADTs
- *Oracle Call Interface Programmer's Guide* for more information about `OCIAnyData` and `OCIAnyDataSet` interfaces
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_TYPES` package

9.5.7 Representing ANSI, DB2, and SQL/DS Data

SQL statements that create tables and clusters can use ANSI data types and data types from the IBM products SQL/DS and DB2 (except those noted after this paragraph). Oracle Database converts the ANSI or IBM 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.

Note:

SQL statements cannot use the SQL/DS and DB2 data types `TIME`, `GRAPHIC`, `VARGRAPHIC`, and `LONG VARGRAPHIC`, because they have no equivalent Oracle data types.

See Also:

Oracle Database SQL Language Reference for conversion details

9.6 Identifying Rows by Address

The fastest way to access the row of a database table is by its address, or **rowid**. If a row is larger than one data block, then its rowid identifies its initial row piece.

To see rowids, query the `ROWID` pseudocolumn. Each value in the `ROWID` pseudocolumn is a string that represents the address of a row. The data type of the string is either `ROWID` or `UROWID`.

Note:

The rowid for a row may change for a number of reasons, which may be user initiated or internally by the database engine. You cannot depend on the rowid to be pointing to the same row or a valid row at all after any of these operations has occurred.

See Also:

- *Oracle Database Concepts* for an overview of the `ROWID` pseudocolumn
- *Oracle Database Concepts* for an overview of rowid data types
- *Oracle Database SQL Language Reference* for more information about the `ROWID` pseudocolumn
- *Oracle Database SQL Language Reference* for more information about the `ROWID` data type
- *Oracle Database SQL Language Reference* for more information about the `UROWID` data type
- *Oracle Call Interface Programmer's Guide* for information about using the `ROWID` data type in C
- *Pro*C/C++ Programmer's Guide* for information about using the `ROWID` data type with the Pro*C/C++ precompiler
- *Oracle Database JDBC Developer's Guide* for information about using the `ROWID` data type in Java
- *Oracle Database Concepts* for more information about HCC

9.7 Displaying Metadata for SQL Operators and Functions

The dynamic performance view `V$SQLFN_METADATA` displays metadata about SQL operators and functions. For every function that `V$SQLFN_METADATA` displays, the dynamic performance view `V$SQLFN_ARG_METADATA` has one row of metadata about each function argument. If a function argument can be repeated (as in the functions `LEAST` and `GREATEST`), then `V$SQLFN_ARG_METADATA` has only one row for each repeating argument. You can join the views `V$SQLFN_METADATA` and `V$SQLFN_ARG_METADATA` on the column `FUNC_ID`.

These views let third-party tools leverage SQL functions without maintaining their metadata in the application layer.

Topics:

- ARGn Data Type
- DISP_TYPE Data Type
- SQL Data Type Families

See Also:

- *Oracle Database Reference* for more information about `V$SQLFN_METADATA`
- *Oracle Database Reference* for more information about `V$SQLFN_ARG_METADATA`

9.7.1 ARGn Data Type

In the view `V$SQLFN_METADATA`, the column `DATATYPE` is the data type of the function (that is, the data type that the function returns). This data type can be an Oracle data type, data type family, or `ARG`*`n`*. `ARG`*`n`* is the data type of the *n*th argument of the function. For example:

- The `MAX` function returns a value that has the data type of its first argument, so the `MAX` function has return data type `ARG1`.
- The `DECODE` function returns a value that has the data type of its third argument, so the `DECODE` function has data type `ARG3`.

See Also:

- SQL Data Type Families
- *Oracle Database SQL Language Reference* for more information about `MAX` function
- *Oracle Database SQL Language Reference* for more information about `DECODE` function

9.7.2 DISP_TYPE Data Type

In the view `V$SQLFN_METADATA`, the column `DISP_TYPE` is the data type of an argument that can be any expression. An expression is either a single value or a combination of values and SQL functions that has a single value.

Table 9-10 Display Types of SQL Functions

Display Type	Description	Example
NORMAL	FUNC(A,B,...)	LEAST(A,B,C)
ARITHMETIC	A FUNC B)	A+B
PARENTHESIS	FUNC()	SYS_GUID()
RELOP	A FUNC B	A IN B
CASE_LIKE	CASE statement or DECODE decode	
NOPAREN	FUNC	SYSDATE

9.7.3 SQL Data Type Families

Often, a SQL function argument can have any data type in a data type family. Table 9-11 shows the SQL data type families and their member data types.

Table 9-11 SQL Data Type Families

Family	Data Types
STRING	• CHARACTER • VARCHAR2 • CLOB • NCHAR • NVARCHAR2 • NCLOB • LONG
NUMERIC	• NUMBER • BINARY_FLOAT • BINARY_DOUBLE
DATETYPE	• DATE • TIMESTAMP • TIMESTAMP WITH TIME ZONE • TIMESTAMP WITH LOCAL TIME ZONE • INTERVAL YEAR TO MONTH • INTERVAL DAY TO SECOND
BINARY	• BLOB • RAW • LONG RAW