# 3

# Selecting a Datatype

This chapter discusses how to use Oracle *built-in* datatypes in applications. Topics include:

**See Also:**

- *Oracle9i Application Developer's Guide - Object-Relational Features* , for information about more complex types, such as object types, varrays, and nested tables.
- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* , for information about LOB datatypes.
- *PL/SQL User's Guide and Reference* , for information the PL/SQL data types. Many SQL datatypes are the same or similar in PL/SQL.

## Summary of Oracle Built-In Datatypes

A datatype associates a fixed set of properties with the values that can be used in a column of a table or in an argument of a procedure or function. These properties cause Oracle to treat values of one datatype differently from values of another datatype. For example, Oracle can add values of `NUMBER` datatype, but not values of `RAW` datatype.

Oracle supplies the following built-in datatypes:

- Character datatypes
  - `CHAR`
  - `NCHAR`
  - `VARCHAR2` and `VARCHAR`
  - `NVARCHAR2`
  - `CLOB`
  - `NCLOB`
  - `LONG`
- `NUMBER` datatype
- Time and date datatypes:
  - `DATE`
  - `INTERVAL DAY TO SECOND`
  - `INTERVAL YEAR TO MONTH`
  - `TIMESTAMP`

  - TIMESTAMP WITH TIME ZONE
  - TIMESTAMP WITH LOCAL TIME ZONE
- Binary datatypes
  - BLOB
  - BFILE
  - RAW
  - LONG RAW

Another datatype, ROWID, is used for values in the ROWID pseudocolumn, which represents the unique address of each row in a table.

> **See Also:** See *Oracle9i SQL Reference* for general descriptions of these datatypes, and see *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for information about the LOB datatypes.

Table 3–1 summarizes the information about each Oracle built-in datatype.

***Table 3–1 Summary of Oracle Built-In Datatypes***

| Datatype | Description | Column Length and Default |
|---|---|---|
| CHAR (*size* [BYTE \| CHAR]) | Fixed-length character data of length *size* bytes or characters. | Fixed for every row in the table (with trailing blanks); maximum size is 2000 bytes per row, default size is 1 byte per row. Consider the character set (single-byte or multibyte) before setting *size.* |
| VARCHAR2 (*size* [BYTE \| CHAR]) | Variable-length character data, with maximum length *size* bytes or characters. | Variable for each row, up to 4000 bytes per row. Consider the character set (single-byte or multibyte) before setting *size.* A maximum *size* must be specified. |
| NCHAR (*size*) | Fixed-length Unicode character data of length *size* characters. | Fixed for every row in the table (with trailing blanks). Column *size* is the number of characters. (The number of bytes is 2 times this number for the AL16UTF16 encoding and 3 times this number for the UTF8 encoding.) The upper limit is 2000 bytes per row. Default is 1 character. |

***Table 3–1 Summary of Oracle Built-In Datatypes***

| Datatype | Description | Column Length and Default |
|---|---|---|
| NVARCHAR2 (*size*) | Variable-length Unicode character data of length *size* characters. A maximum *size* must be specified. | Variable for each row. Column *size* is the number of characters. (The number of bytes may be up to 2 times this number for a the AL16UTF16 encoding and 3 times this number for the UTF8 encoding.) The upper limit is 4000 bytes per row. Default is 1 character. |
| CLOB | Single-byte character data | Up to $2^{32}$ - 1 bytes, or 4 gigabytes. |
| NCLOB | Unicode national character set (NCHAR) data. | Up to $2^{32}$ - 1 bytes, or 4 gigabytes. |
| LONG | Variable-length character data. | Variable for each row in the table, up to $2^{32}$ - 1 bytes, or 2 gigabytes, per row. Provided for backward compatibility. |
| NUMBER (*p*, *s*) | Variable-length numeric data. Maximum precision *p* and / or scale *s* is 38. | Variable for each row. The maximum space required for a given column is 21 bytes per row. |
| DATE | Fixed-length date and time data, ranging from Jan. 1, 4712 B.C.E. to Dec. 31, 4712 C.E. | Fixed at 7 bytes for each row in the table. Default format is a string (such as DD-MON-RR) specified by the NLS_DATE_FORMAT parameter. |
| INTERVAL YEAR (*precision*) TO MONTH | A period of time, represented as years and months. The *precision* value specifies the number of digits in the YEAR field of the date. The precision can be from 0 to 9, and defaults to 2 for years. | Fixed at 5 bytes. |
| INTERVAL DAY (*precision*) TO SECOND (*precision*) | A period of time, represented as days, hours, minutes, and seconds. The *precision* values specify the number of digits in the DAY and the fractional SECOND fields of the date. The precision can be from 0 to 9, and defaults to 2 for days and 6 for seconds. | Fixed at 11 bytes. |

***Table 3–1 Summary of Oracle Built-In Datatypes***

| Datatype | Description | Column Length and Default |
|---|---|---|
| TIMESTAMP (*precision*) | A value representing a date and time, including fractional seconds. (The exact resolution depends on the operating system clock.)<br><br>The precision value specifies the number of digits in the fractional second part of the SECOND date field. The precision can be from 0 to 9, and defaults to 6 | Varies from 7 to 11 bytes, depending on the precision. The default is determined by the NLS_TIMESTAMP_FORMAT initialization parameter. |
| TIMESTAMP (*precision*) WITH TIME ZONE | A value representing a date and time, plus an associated time zone setting. The time zone can be an offset from UTC, such as '-5:0', or a region name, such as 'US/Pacific'. | Fixed at 13 bytes. The default is determined by the NLS_TIMESTAMP_TZ_FORMAT initialization parameter. |
| TIMESTAMP (*precision*) WITH LOCAL TIME ZONE | Similar to TIMESTAMP WITH TIME ZONE, except that the data is normalized to the database time zone when stored, and adjusted to match the client's time zone when retrieved. | Varies from 7 to 11 bytes, depending on the precision. The default is determined by the NLS_TIMESTAMP_FORMAT initialization parameter. |
| BLOB | Unstructured binary data | Up to $2^{32}$ - 1 bytes, or 4 gigabytes. |
| BFILE | Binary data stored in an external file | Up to $2^{32}$ - 1 bytes, or 4 gigabytes. |
| RAW (*size*) | Variable-length raw binary data | Variable for each row in the table, up to 2000 bytes per row. A maximum *size* must be specified. Provided for backward compatibility. |
| LONG RAW | Variable-length raw binary data | Variable for each row in the table, up to $2^{31}$ - 1 bytes, or 2 gigabytes, per row. Provided for backward compatibility. |
| ROWID | Binary data representing row addresses | Fixed at 10 bytes (extended ROWID) or 6 bytes (restricted ROWID) for each row in the table. |

***Table 3–1 Summary of Oracle Built-In Datatypes***

| Datatype | Description | Column Length and Default |
|---|---|---|
| CHAR (*size* [BYTE \| CHAR]) | Fixed-length character data of length *size* bytes or characters. | Fixed for every row in the table (with trailing blanks); maximum size is 2000 bytes per row, default size is 1 byte per row. Consider the character set (single-byte or multibyte) before setting *size.* |
| VARCHAR2 (*size* [BYTE \| CHAR]) | Variable-length character data, with maximum length *size* bytes or characters. | Variable for each row, up to 4000 bytes per row. Consider the character set (single-byte or multibyte) before setting *size.* A maximum *size* must be specified. |
| NCHAR (*size*) | Fixed-length Unicode character data of length *size* characters. | Fixed for every row in the table (with trailing blanks). Column *size* is the number of characters. (The number of bytes is 2 times this number for the AL16UTF16 encoding and 3 times this number for the UTF8 encoding.) The upper limit is 2000 bytes per row. Default is 1 character. |
| NVARCHAR2 (*size*) | Variable-length Unicode character data of length *size* characters. A maximum *size* must be specified. | Variable for each row. Column *size* is the number of characters. (The number of bytes may be up to 2 times this number for a the AL16UTF16 encoding and 3 times this number for the UTF8 encoding.) The upper limit is 4000 bytes per row. Default is 1 character. |
| CLOB | Single-byte character data | Up to $2^{32}$ - 1 bytes, or 4 gigabytes. |
| NCLOB | Unicode national character set (NCHAR) data. | Up to $2^{32}$ - 1 bytes, or 4 gigabytes. |
| LONG | Variable-length character data. | Variable for each row in the table, up to $2^{32}$ - 1 bytes, or 2 gigabytes, per row. Provided for backward compatibility. |
| NUMBER (*p*, *s*) | Variable-length numeric data. Maximum precision *p* and / or scale *s* is 38. | Variable for each row. The maximum space required for a given column is 21 bytes per row. |

***Table 3–1 Summary of Oracle Built-In Datatypes***

| Datatype | Description | Column Length and Default |
|---|---|---|
| DATE | Fixed-length date and time data, ranging from Jan. 1, 4712 B.C.E. to Dec. 31, 4712 C.E. | Fixed at 7 bytes for each row in the table. Default format is a string (such as DD-MON-RR) specified by the NLS_DATE_FORMAT parameter. |
| INTERVAL YEAR (*precision*) TO MONTH | A period of time, represented as years and months. The *precision* value specifies the number of digits in the YEAR field of the date. The precision can be from 0 to 9, and defaults to 2 for years. | Fixed at 5 bytes. |
| INTERVAL DAY (*precision*) TO SECOND (*precision*) | A period of time, represented as days, hours, minutes, and seconds. The *precision* values specify the number of digits in the DAY and the fractional SECOND fields of the date. The precision can be from 0 to 9, and defaults to 2 for days and 6 for seconds. | Fixed at 11 bytes. |
| TIMESTAMP (*precision*) | A value representing a date and time, including fractional seconds. (The exact resolution depends on the operating system clock.)<br>The precision value specifies the number of digits in the fractional second part of the SECOND date field. The precision can be from 0 to 9, and defaults to 6 | Varies from 7 to 11 bytes, depending on the precision. The default is determined by the NLS_TIMESTAMP_FORMAT initialization parameter. |
| TIMESTAMP (*precision*) WITH TIME ZONE | A value representing a date and time, plus an associated time zone setting. The time zone can be an offset from UTC, such as '-5:0', or a region name, such as 'US/Pacific'. | Fixed at 13 bytes. The default is determined by the NLS_TIMESTAMP_TZ_FORMAT initialization parameter. |

***Table 3–1 Summary of Oracle Built-In Datatypes***

| Datatype | Description | Column Length and Default |
|---|---|---|
| `TIMESTAMP (`*`precision`*`) WITH LOCAL TIME ZONE` | Similar to `TIMESTAMP WITH TIME ZONE`, except that the data is normalized to the database time zone when stored, and adjusted to match the client's time zone when retrieved. | Varies from 7 to 11 bytes, depending on the precision. The default is determined by the `NLS_TIMESTAMP_FORMAT` initialization parameter. |
| `BLOB` | Unstructured binary data | Up to $2^{32}$ - 1 bytes, or 4 gigabytes. |
| `BFILE` | Binary data stored in an external file | Up to $2^{32}$ - 1 bytes, or 4 gigabytes. |
| `RAW (`*`size`*`)` | Variable-length raw binary data | Variable for each row in the table, up to 2000 bytes per row. A maximum *size* must be specified. Provided for backward compatibility. |
| `LONG RAW` | Variable-length raw binary data | Variable for each row in the table, up to $2^{31}$ - 1 bytes, or 2 gigabytes, per row. Provided for backward compatibility. |
| `ROWID` | Binary data representing row addresses | Fixed at 10 bytes (extended `ROWID`) or 6 bytes (restricted `ROWID`) for each row in the table. |
| `CHAR (`*`size`* `[BYTE \| CHAR])` | Fixed-length character data of length *size* bytes or characters. | Fixed for every row in the table (with trailing blanks); maximum size is 2000 bytes per row, default size is 1 byte per row. Consider the character set (single-byte or multibyte) before setting *size.* |
| `VARCHAR2 (`*`size`* `[BYTE \| CHAR])` | Variable-length character data, with maximum length *size* bytes or characters. | Variable for each row, up to 4000 bytes per row. Consider the character set (single-byte or multibyte) before setting *size.* A maximum *size* must be specified. |

***Table 3–1 Summary of Oracle Built-In Datatypes***

| Datatype | Description | Column Length and Default |
|---|---|---|
| NCHAR (*size*) | Fixed-length Unicode character data of length *size* characters. | Fixed for every row in the table (with trailing blanks). Column *size* is the number of characters. (The number of bytes is 2 times this number for the AL16UTF16 encoding and 3 times this number for the UTF8 encoding.) The upper limit is 2000 bytes per row. Default is 1 character. |
| NVARCHAR2 (*size*) | Variable-length Unicode character data of length *size* characters. A maximum *size* must be specified. | Variable for each row. Column *size* is the number of characters. (The number of bytes may be up to 2 times this number for a the AL16UTF16 encoding and 3 times this number for the UTF8 encoding.) The upper limit is 4000 bytes per row. Default is 1 character. |
| CLOB | Single-byte character data | Up to $2^{32}$ - 1 bytes, or 4 gigabytes. |
| NCLOB | Unicode national character set (NCHAR) data. | Up to $2^{32}$ - 1 bytes, or 4 gigabytes. |
| LONG | Variable-length character data. | Variable for each row in the table, up to $2^{32}$ - 1 bytes, or 2 gigabytes, per row. Provided for backward compatibility. |
| NUMBER (*p*, *s*) | Variable-length numeric data. Maximum precision *p* and / or scale *s* is 38. | Variable for each row. The maximum space required for a given column is 21 bytes per row. |
| DATE | Fixed-length date and time data, ranging from Jan. 1, 4712 B.C.E. to Dec. 31, 4712 C.E. | Fixed at 7 bytes for each row in the table. Default format is a string (such as DD-MON-RR) specified by the NLS_DATE_FORMAT parameter. |
| INTERVAL YEAR (*precision*) TO MONTH | A period of time, represented as years and months. The *precision* value specifies the number of digits in the YEAR field of the date. The precision can be from 0 to 9, and defaults to 2 for years. | Fixed at 5 bytes. |

*Table 3–1 Summary of Oracle Built-In Datatypes*

| Datatype | Description | Column Length and Default |
|---|---|---|
| INTERVAL DAY (*precision*) TO SECOND (*precision*) | A period of time, represented as days, hours, minutes, and seconds. The *precision* values specify the number of digits in the DAY and the fractional SECOND fields of the date. The precision can be from 0 to 9, and defaults to 2 for days and 6 for seconds. | Fixed at 11 bytes. |
| TIMESTAMP (*precision*) | A value representing a date and time, including fractional seconds. (The exact resolution depends on the operating system clock.)<br>The precision value specifies the number of digits in the fractional second part of the SECOND date field. The precision can be from 0 to 9, and defaults to 6 | Varies from 7 to 11 bytes, depending on the precision. The default is determined by the NLS_TIMESTAMP_FORMAT initialization parameter. |

# Representing Character Data

Use the character datatypes to store alphanumeric data:

- CHAR and NCHAR datatypes store fixed-length character strings.
- VARCHAR2 and NVARCHAR2 datatypes store variable-length character strings. (The VARCHAR datatype is synonymous with the VARCHAR2 datatype.)
- NCHAR and NVARCHAR2 datatypes store Unicode character data only.
- CLOB and NCLOB datatypes store single-byte and multibyte character strings of up to four gigabytes.

  **See Also:** *Oracle9i Application Developer's Guide - Large Objects (LOBs)*

- The LONG datatype stores variable-length character strings containing up to two gigabytes, but with many restrictions.

**See Also:** "Restrictions on LONG and LONG RAW Datatypes"

This datatype is provided for backward compatibility with existing applications; in general, new applications should use CLOB and NCLOB datatypes to store large amounts of character data, and BLOB and BFILE to store large amounts of binary data.

When deciding which datatype to use for a column that will store alphanumeric data in a table, consider the following points of distinction:

### Space Usage

- To store data more efficiently, use the VARCHAR2 datatype. The CHAR datatype blank-pads and stores trailing blanks up to a fixed column length for all column values, while the VARCHAR2 datatype does not add any extra blanks.

### Comparison Semantics

- Use the CHAR datatype when you require ANSI compatibility in comparison semantics (when trailing blanks are not important in string comparisons). Use the VARCHAR2 when trailing blanks are important in string comparisons.

### Future Compatibility

- The CHAR and VARCHAR2 datatypes are and will always be fully supported. At this time, the VARCHAR datatype automatically corresponds to the VARCHAR2 datatype and is reserved for future use.

CHAR, VARCHAR2, and LONG data is automatically converted from the database character set to the character set defined for the user session by the NLS_LANGUAGE parameter, where these are different.

### Column Lengths for Single-Byte and Multibyte Character Sets

The lengths of CHAR and VARCHAR2 columns can be specified as either bytes or characters.

The lengths of NCHAR and NVARCHAR2 columns are always specified in characters, making them ideal for storing Unicode data, where a character might consist of multiple bytes.

```
-- ID contains only single-byte data, up to 32 bytes.
ID VARCHAR2(32 BYTE);
-- NAME contains data in the database character set. The 32 characters might
-- be physically stored as more than 32 bytes, if the database character set
```

```
allows
-- multibyte characters.
NAME VARCHAR2(32 CHAR);
-- BIOGRAPHY can represent 2000 characters in any Unicode-representable
language.
-- The exact encoding depends on the national character set, but the column
-- can contain multibyte values even if the database character set is
single-byte.
BIOGRAPHY NVARCHAR2(2000);
-- The representation of COMMENT, as 2000 bytes or 2000 characters, depends
-- on the initialization parameter NLS_LENGTH_SEMANTICS.
COMMENT VARCHAR2(2000);
```

When using a multibyte database character encoding scheme, consider carefully the space required for tables with character columns. If the database character encoding scheme is single-byte, then the number of bytes and the number of characters in a column is the same. If it is multibyte, then there generally is no such correspondence. A character might consist of one or more bytes depending upon the specific multibyte encoding scheme, and whether shift-in / shift-out control codes are present. To avoid overflowing buffers, specify data as NCHAR or NVARCHAR2 if it might use a Unicode encoding that is different from the database character set.

**See Also:**

- *Oracle9i Globalization and National Language Support Guide*
- *Oracle9i SQL Reference*
- *Oracle Time Series User's Guide*

for information about globalization support within Oracle and support for different character encoding schemes.

### Implicit Conversion Between CHAR/VARCHAR2 and NCHAR/NVARCHAR2

In previous releases (Oracle8*i* and earlier), the NCHAR and NVARCHAR2 types were difficult to use because they could not be interchanged with CHAR and VARCHAR2. For example, an NVARCHAR2 literal required special notation, such as `N'string_value'`. Now, you can specify NCHAR and NVARCHAR2 without the N qualifier, and can mix them with CHAR and VARCHAR2 values in SQL statements and functions.

### Comparison Semantics

Oracle compares CHAR and NCHAR values using **blank-padded comparison semantics**. If two values have different lengths, then Oracle adds blanks at the end of the shorter value, until the two values are the same length. Oracle then compares the values character-by-character up to the first character that differs. The value with the greater character in the first differing position is considered greater. Two values that differ only in the number of trailing blanks are considered equal.

Oracle compares VARCHAR2 and NVARCHAR2 values using **non-padded comparison semantics**. Two values are considered equal only if they have the same characters and are of equal length. Oracle compares the values character-by-character up to the first character that differs. The value with the greater character in that position is considered greater.

Because Oracle blank-pads values stored in CHAR columns but not in VARCHAR2 columns, a value stored in a VARCHAR2 column may take up less space than if it were stored in a CHAR column. For this reason, a full table scan on a large table containing VARCHAR2 columns may 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 this data in VARCHAR2 columns rather than in CHAR columns.

However, performance is not the only factor to consider when deciding which of these datatypes to use. Oracle uses different semantics to compare values of each datatype. You might choose one datatype over the other if your application is sensitive to the differences between these semantics. For example, if you want Oracle to ignore trailing blanks when comparing character values, then you must store these values in CHAR columns.

> **See Also:** For more information on comparison semantics for these datatypes, see the *Oracle9i SQL Reference*.

## Representing Numeric Data

Use the NUMBER datatype to store real numbers in a fixed-point or floating-point format. Numbers using this datatype are guaranteed to be portable among different Oracle platforms, and offer up to 38 decimal digits of precision. You can store positive and negative numbers of magnitude $1 \times 10^{-130}$ through $9.9\overline{9} \times 10^{125}$, as well as zero, in a NUMBER column.

You can specify that a column contains a floating-point number, for example:

```
distance NUMBER
```

Or, you can specify a precision (total number of digits) and scale (number of digits to right of decimal point):

```
price NUMBER (8, 2)
```

Although not required, specifying precision and scale helps to identify bad input values. If a precision is not specified, the column stores values as given. The following table shows examples of how data different scale factors affect storage.

***Table 3–2 How Scale Factors Affect Numeric Data Storage***

| Input Data | Specified As | Stored As |
|---|---|---|
| 7,456,123.89 | NUMBER | 7456123.89 |
| 7,456,123.89 | NUMBER (9) | 7456124 |
| 7,456,123.89 | NUMBER (9,2) | 7456123.89 |
| 7,456,123.89 | NUMBER (9,1) | 7456123.9 |
| 7,456,123.89 | NUMBER (6) | (not accepted, exceeds precision) |
| 7,456,123.89 | NUMBER (7, -2) | 7456100 |

**See Also:** For information about the internal format for the NUMBER datatype, see *Oracle9i Database Concepts*.

## Representing Date and Time Data

Use the DATE datatype to store point-in-time values (dates and times) in a table. The DATE datatype stores the century, year, month, day, hours, minutes, and seconds.

Use the TIMESTAMP datatype to store precise values, down to fractional seconds. For example, an application that must decide which of two events occurred first might use TIMESTAMP. An application that needs to specify the time for a job to execute might use DATE.

Because TIMESTAMP WITH TIME ZONE can also store time zone information, it is particularly suited for recording date information that must be gathered or coordinated across geographic regions.

Use TIMESTAMP WITH LOCAL TIME ZONE values when the time zone is not significant. For example, you might use it in an application that schedules teleconferences, where each participant sees the start and end times for their own time zone.

The TIMESTAMP WITH LOCAL TIME ZONE type is appropriate for two-tier applications where you want to display dates and times using the time zone of the client system. You should not use it in three-tier applications, such as those involving a web server, because in that case the client is the web server, so data displayed in a web browser is formatted according to the time zone of the web server rather than the time zone of the browser.

Use INTERVAL DAY TO SECOND to represent the precise difference between two datetime values. For example, you might use this value 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, including multiple years, with high precision, you can use a large value for the days portion.

Use INTERVAL YEAR TO MONTH to represent the difference between two datetime values, where the only significant portions are the year and month. For example, you might use this value to set a reminder for a date 18 months in the future, or check whether 6 months have elapsed since a particular date.

Oracle uses its own internal format to store dates. Date data is stored in fixed-length fields of seven bytes each, corresponding to century, year, month, day, hour, minute, and second.

> **See Also:** See the *Oracle Call Interface Programmer's Guide* for a complete description of the Oracle internal date format.

## Date Format

For input and output of dates, the standard Oracle default date format is DD-MON-RR. For example:

```
'13-NOV-1992'
```

To change this default date format on an instance-wide basis, use the NLS_DATE_FORMAT parameter. To change the format during a session, use the ALTER SESSION statement. To enter dates that are not in the current default date format, use the TO_DATE function with a format mask. For example:

```
TO_DATE ('November 13, 1992', 'MONTH DD, YYYY')
```

> **See Also:** Oracle Julian dates might not be compatible with Julian dates generated by other date algorithms. For information about Julian dates, see *Oracle9i Database Concepts*.

Be careful using a date format like DD-MON-YY. The YY indicates the year in the current century. For example, **31-DEC-92** is **December 31, 2092**, not 1992 as you might expect. If you want to indicate years in any century other than the current one, use a different format mask, such as the default RR.

Example: Printing a Date with BC/AD Notation

```
SQL> -- By default, the date is printed without any BC or AD qualifier.
SQL> select sysdate from dual;

SYSDATE
---------
24-JAN-02
SQL> -- Adding BC to the format string prints the date with BC or AD
SQL> -- as appropriate.
SQL> select to_char(sysdate, 'DD-MON-YYYY BC') from dual;

TO_CHAR(SYSDAT
--------------
24-JAN-2002 AD
```

### Checking If Two DATE Values Refer to the Same Day

To compare dates that have time data, use the SQL function TRUNC to ignore the time component.

### Displaying the Current Date and Time

Use the SQL function SYSDATE to return the system date and time.

### Tip: Setting SYSDATE to a Constant Value

The FIXED_DATE initialization parameter lets you set SYSDATE to a constant, which can be useful for testing.

## Time Format

Time is stored in 24-hour format, HH24:MI:SS. By default, the time in a DATE column is 12:00:00 A.M. (midnight) if no time portion is entered, or if the DATE is

truncated. In a time-only entry, the date portion defaults to the first day of the current month. To enter the time portion of a date, use the TO_DATE function with a format mask indicating the time portion, as in:

> **Note:** You may need to set up the following data structures for certain examples to work:
>
> ```
> CREATE TABLE Birthdays_tab (Bname VARCHAR2(20),Bday DATE)
> ```

```
INSERT INTO Birthdays_tab (bname, bday) VALUES
    ('ANNIE',TO_DATE('13-NOV-92 10:56 A.M.','DD-MON-YY HH:MI A.M.'));
```

### Performing Date Arithmetic

Oracle provides a number of features to help with date arithmetic, so that you do not need to perform your own calculations on the number of seconds in a day, the number of days in each month, and so on.

Some useful functions include:

- ADD_MONTHS
- SYSDATE
- SYSTIMESTAMP
- TRUNC. When applied to a DATE value, it trims off the time portion so that it represents the very beginning of the day (the stroke of midnight). By truncating two DATE values and comparing them, you can check whether they refer to the same day. You can also use TRUNC along with a GROUP BY clause to produce daily totals.
- Arithmetic operators such as + and -.
- INTERVAL datatype. To represent constants when performing date arithmetic, you can use the INTERVAL datatype rather than performing your own calculations. For example, you might add or subtract INTERVAL constants from DATE values, or subtract two DATE values and compare the result to an INTERVAL.
- Comparison operators such as >, <, =, and BETWEEN.

### Converting Between Datetime Types

Some useful functions include:

- EXTRACT
- NUMTODSINTERVAL
- NUMTOYMINTERVAL
- TO_DATE (and its opposite, TO_CHAR)
- TO_DSINTERVAL
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ
- TO_YMINTERVAL

> **See Also:** *Oracle9i SQL Reference* for full details about each function.

### Handling Time Zones

Oracle provides a number of functions to help with calculations involving time zones. For example, TO_DATE does not work with values of type TIMESTAMP WITH TIME ZONE; you must use TO_TIMESTAMP_TZ instead.

Some useful functions include:

- CURRENT_DATE
- CURRENT_TIMESTAMP
- DBTIMEZONE
- EXTRACT
- FROM_TZ
- LOCALTIMESTAMP
- SESSIONTIMEZONE
- SYS_EXTRACT_UTC
- SYSTIMESTAMP
- TO_TIMESTAMP_TZ

**See Also:** *Oracle9i SQL Reference*. for full details about each function.

### Importing and Exporting Datetime Types

`TIMESTAMP WITH TIME ZONE` and `TIMESTAMP WITH LOCAL TIME ZONE` values are always stored in normalized format, so that you can export, import, and compare them without worrying about time zone offsets. `DATE` and `TIMESTAMP` values do not store an associated time zone, and you must adjust them to account for any time zone differences between source and target databases.

## Establishing Year 2000 Compliance

An application must satisfy the following criteria to meet the requirements for Year 2000 (Y2K) compliance:

- Process date information before, during, and after 1st January 2000 without error. This entails accepting date input, providing date output, storing date information and performing calculation on dates or portions of dates.
- Provide services as published in its documentation before, during and after 1st January 2000 without changes in operation resulting from the advent of the new century.
- Respond to two digit date input in a way that resolves ambiguity as to the century in a clearly defined manner.
- Manage the leap year occurring in the year 2000 according to the quad-centennial rule.

These criteria are a superset of the Year 2000 conformance requirements set out by the British Standards Institute in DISC PD-2000-1 A Definition of Year 2000 Conformity Requirements.

You can warrant your application as Y2K compliant only if you have validated its conformance at all three of the following system levels:

- Hardware
- System software, including databases, transaction processors and operating systems
- Application software, from third parties or developed in-house

### Centuries and the Year 2000

Oracle stores year data with the century information. For example, the Oracle database stores 1996 or 2001, and not just 96 or 01. The DATE datatype always stores a four-digit year internally, and all other dates stored internally in the database have four digit years. Oracle utilities such as import, export, and recovery also deal properly with four-digit years.

Applications that use the Oracle RDBMS (Oracle7 or later) and exploit the DATE data type (for dates or dates with time values) need have no concerns about their stored data and the year 2000. Beginning with Oracle7, the DATE data type stores date and time data to a precision that includes a four digit year and a time component down to seconds (typically `'YYYY:MM:DD:HH24:MI:SS'`)

However, some applications might be written with an assumption about the year (such as assuming that everything is 19*xx*). The application might hand over a two-digit year to the database, and the procedures that Oracle uses for determining the century could be different from what the programmer expects (see "Troubleshooting Y2K Problems in Applications" on page 3-24). For this reason, you should review and test your code with regard to the Year 2000.

### Examples of The 'RR' Date Format

The RR date format element of the TO_DATE and TO_CHAR functions allows a database site to default the century to different values depending on the two-digit year, so that years 50 to 99 default to 19*xx* and years 00 to 49 default to 20*xx*. Therefore, regardless of the current century at the time the data is entered, the 'RR' format will ensure that the year stored in the database is as follows:

- If the current year is in the second half of the century (50 - 99), and a two-digit year between '00' and '49' is entered, this will be stored as a 'next century' year. For example, '02' entered in 1996 will be stored as '2002'.
- If the current year is in the second half of the century (50 - 99), and a two-digit year between '50' and '99' is entered, this will be stored as a 'current century' year. For example, '97' entered in 1996 will be stored as '1997'.
- If the current year is in the first half of the century (00 - 49), and a two-digit year between '00' and '49' is entered, this will be stored as a 'current century' year. For example, '02' entered in 2001 will be stored as '2002'.
- If the current year is in the first half of the century (00 - 49), and a two-digit year between '50' and '99' is entered, this will be stored as a 'previous century' year. For example, '97' entered in 2001 will be stored as '1997'.

The ‘RR’ date format is available for inserting and updating DATE data in the database. It is not required for retrieval or query of data already stored in the database as Oracle has always stored the YEAR component of a date in its four-digit form.

Here is an example of the RR usage:

```
INSERT INTO emp (empno, deptno,hiredate) VALUES
   (9999, 20, TO_DATE('01-jan-03', 'DD-MON-RR'));

 INSERT INTO emp (empno, deptno,hiredate) VALUES
    (8888, 20, TO_DATE('01-jan-67',  'DD-MON-RR'));

SELECT empno, deptno,
   TO_CHAR(hiredate, 'DD-MON-YYYY') hiredate
FROM emp;
```

This produces the following data:

```
EMPNO          DEPTNO        HIREDATE
----------     ----------    -----------------
8888           20            01-JAN-1967
9999           20            01-JAN-2003
```

### Examples of The ’CC’ Date Format

The CC date format element of the TO_CHAR function returns the century of a given date. For example:

```
SELECT TO_CHAR(TO_DATE('01-JAN-2000','DD-MON-YYYY'),'CC') CENTURY FROM DUAL;

CENTURY
-------
20

SELECT TO_CHAR(TO_DATE('01-JAN-2001','DD-MON-YYYY'),'CC') CENTURY FROM DUAL;

CENTURY
-------
21
```

The CC date format element of the TO_CHAR function sets the century value to one greater than the first two digits of a four-digit year (for example, '20' from '1900'). For years that are a multiple of 100, this is not the true century. Strictly speaking, the century of '1900' is not the twentieth century (which began in 1901) but rather the nineteenth century.

The following workaround computes the correct century for any Common Era (CE, formerly known as AD) date. If *userdate* is a CE date for which you want the true century, use the following expression:

```
SELECT DECODE (TO_CHAR (Hiredate, 'YY'),
    '00', TO_CHAR (Hiredate - 366, 'CC'),
    TO_CHAR (Hiredate, 'CC'))  FROM Emp_tab;
```

This expression works as follows: Get the last two digits of the year. If it is '00', then it is a year in which the Oracle century is one year too large, and compute a date in the preceding year (whose Oracle century is the desired true century). Otherwise, use the Oracle century.

> **See Also:** For more information about date format codes, see *Oracle9i SQL Reference*.

### Storing Dates in Character Data Types

Where applications store date values in CHAR or VARCHAR2 datatypes, and the century information is not maintained, you will need to modify the application to include routines which ensure that such dates are treated appropriately when affected by the change in century. You can do this by changing the strings to maintain century information or, with certain constraints, by using the 'RR' date format when interpreting the string as a date.

If you are creating a new application, or if you are modifying an application to ensure that dates stored as character strings are Year 2000 compliant, we advise that you convert dates to use the Oracle DATE data type. If this is not feasible, store the dates in a form which is language and format independent, and which handles full years. For example, use 'SYYYY/MM/DD' plus the time element as 'HH24:MI:SS' if necessary. Note that dates stored in this form must be converted to the correct external format whenever they are received or displayed.

The format 'SYYYY/MM/DD HH24:MI:SS' has the following advantages:

- It is language-independent in that the months are numeric.
- It contains the full four-digit year so centuries are unambiguous.
- The time is represented fully. Since the most significant elements occur first, character-based sort operations will process the dates correctly.

The “S” format element prefixes BC dates with “-“.

### Viewing Date Settings

The following views let you verify what your settings are:

- V$NLS_DATABASE_PARAMETERS shows instance-wide Globalization Support parameters, whether the default or a value explicitly declared in the initialization parameter file.
- NLS_SESSION_PARAMETERS shows current session values, which may have been changed by ALTER SESSION.

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

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

Please note that the format does not change the internal representation of the value in the database.

To see the available values for time zone region and time zone abbreviation, you can query the view V$TIMEZONE_NAMES.

### Altering Date Settings

You may set the date format in your environment or as the default for the entire database. If you set this in your environment, it will override the setting in the initialization parameter.

Change the NLS_DATE_FORMAT parameter settings in the following order:

1. Set the Client side, such as the Windows NT registry and Unix environment variables.
2. Set theSession using ALTER SESSION SET NLS_DATE_FORMAT. To change the date format for the session, issue the following SQL command:

   ```
   ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-RR'
   ```

3. Set the Server using the init.ora NLS_DATE_FORMAT parameter. To change the default date format for the entire database, change INIT.ORA to include the following

```
NLS_DATE_FORMAT = DD-MON-RR
```

The NLS_DATE_FORMAT setting relies on the above order. Therefore, for a client/server application, NLS_DATE_FORMAT must be set on the server and on the client.

---

**Caution:** Changing this parameter at the *database* level will change all existing date fields as described above. Oracle Corporation suggests that you make changes at the *session* level unless all users and all currently running applications process dates in the range 1950-2049.

---

## Troubleshooting Y2K Problems in Applications

In this section we describe some common programming problems around Y2K compliance. These problems may seem to derive from incorrect Year 2000 processing by the database engine, but on closer inspection are seen to arise from incorrect use of Oracle technology.

### Y2K Example: Date Columns Too Short

Your application may have defined the year of a date using a column of CHAR(2) or NUMBER(2) in order to save disk space. This can lead to unpredictable results when 20xx dates are mixed with 19xx dates. To resolve this, modify your application to use the full 4-digit year.

### Y2K Example: 4-Digit Years Mixed with 2-Digit Years

You application may be designed to store a 4-digit year, but the code may allow for the incorrect storage of 2-digit year rows with the 4-digit year rows. This will lead to unpredictable results for queries by date if the date columns contains dates earlier than 1900. To deal with this problem, have your application check for rows which contain dates earlier than 1900, and then adjust for this.

### Y2K Example: Wide Range of Years Stored as 2 Digits

Examine your applications to determine if it processes dates prior to 1950 or later than 2049, and store the year as 2-digits. If both conditions are met, your application should not use the 'RR' format but should instead expand the 2 digit year 'YY ' into a 4 digit year 'YYYY', and store the 4 digit number in the database.

### Y2K Example: Handling Feb. 29, 2000

The following unusual error helps illuminate the interaction between NLS_DATE_FORMAT and the Oracle 'RR' format mask. The following is a syntactically correct statement but contains a logical flaw:

```
SELECT TO_CHAR(TO_DATE(LAST_DAY('01-FEB-00'),'DD-MON-RR'),'MM/DD/RRRR')
FROM DUAL;
```

The above query returns 02/28/2000. This is consistent with the defined behavior of the 'RR' format element, but it is incorrect because the year 2000 is a leap year.

The problem is that the operation is using the default NLS_DATE_FORMAT, which is 'DD-MON-YY'. If the NLS_DATE_FORMAT is changed to 'DD-MON-RR', then the same select returns 02/29/2000, which is the correct value.

Let us evaluate the query as the Oracle Server engine does. The first function processed is the innermost function, LAST_DAY. Because NLS_DATE_FORMAT is YY, this correctly returns 2/28, because it is using the year 1900 to evaluate the expression. The value 2/28 is then returned to the next outer function. So, the TO_DATE and TO_CHAR functions format the value 02/28/00 using the 'RR' format mask and display the result as 02/28/2000.

If SELECT LAST_DAY('01-FEB-00') FROM DUAL is issued, the result changes depending on the NLS_DATE_FORMAT. With 'YY', the LAST_DAY returned is 28-Feb-00 because the year is interpreted as 1900. With 'RR', the LAST_DAY returned is 29-Feb-00 because the year is interpreted as 2000. The year 1900 is not a leap year, but the year 2000 is.

### Y2K Example: Implicit Date Conversion within DECODE

When the DECODE function is used and if the third argument has data type CHAR, VARCHAR2, or if it is NULL, then Oracle converts the return value to datatype VARCHAR2. Therefore, the following statement:

```
INSERT INTO destination_table (date_column)
    SELECT DECODE('31.12.2000', '00000000', NULL,
        TO_DATE('31.12.2000','DD.MM.YYYY'))
    FROM DUAL;
```

inserts date 31.12.1900.

Another sample statement:

```
INSERT INTO destination_table (date_column)
    SELECT DECODE('01.11.1999', '00000000', NULL, sysdate+1000)
FROM DUAL;
```

inserts date 04.10.1901.

In the above examples, the third argument in the DECODE argument list is a NULL value, so Oracle implicitly converted the DATE value to a VARCHAR2 string using the default format mask. This is DD-MON-YY, hence loses the first two digits of the year.

*Note: When inserting the record into a table, Oracle implicitly converts the string into a date, using the first 2-digits of the current year. To ensure the correct year is interpreted, set NLS_DATE_FORMAT using 'RR' or 'YYYY'.*

### Y2K Example: Partitioning Tables Based on DATE Columns

If creating a partitioned table using a DATE data type column in the partition key, use a 4-digit year when specifying date ranges. For example:

```
CREATE TABLE stock_xactions (stock_symbol CHAR(5),
    stock_series CHAR(1),
    num_shares NUMBER(10),
    price NUMBER(5,2),
    trade_date DATE)
    STORAGE (INITIAL 100K NEXT 50K) LOGGING
    PARTITION BY RANGE (trade_date)
      (PARTITION sx1992 VALUES LESS THAN (TO_DATE('01-JAN-1993','DD-MON-YYYY'))
    TABLESPACE ts0
        NOLOGGING,
       PARTITION sx1993 VALUES LESS THAN (TO_DATE('01-JAN-1994','DD-MON-YYYY'))
    TABLESPACE ts1,
      PARTITION sx1994 VALUES LESS THAN (TO_DATE('01-JAN-1995','DD-MON-YYYY'))
    TABLESPACE ts2);
```

### Y2K Example: Views Defined Using 2-Digit Years

Oracle views depend on the session state. In particular, a predicate with a 2-digit year, such as:

```
WHERE col > '12-MAY-99'
```

is allowed in a view. Interpretation of the full 4-digit year depends on the setting of NLS_DATE_FORMAT.

## Representing Geographic Coordinate Data

To represent Geographic Information System (GIS) or spatial data in the database, you can use the Oracle Spatial features, including the type `MDSYS.SDO_GEOMETRY`. You can store the data in the database using either an object-relational or a relational model, and manipulate and query the data using a set of PL/SQL packages.

For more information, see *Oracle Spatial User's Guide and Reference.*

## Representing Image, Audio, and Video Data

Whether you store such multimedia data inside the database as `BLOB`s or `BFILE`s, or store it externally on a web or other kind of server, you can use interMedia to access the data using either an object-relational or a relational model, and manipulate and query the data using a set of object types.

For more information, see *Oracle interMedia User's Guide and Reference.*

## Representing Searchable Text Data

Rather than writing low-level code to do full-text searches, you can use Oracle9*i* Text, formerly known as ConText and *inter*Media Text. It stores the search data in a special kind of index, and lets you query it with operators and PL/SQL packages. This makes it simple to create your own search engine using data from tables, files, or URLs, and combine the search logic with relational queries. You can also search XML data this way, using XPath notation.

For more information, see *Oracle Text Application Developer's Guide.*

## Representing Large Data Types

In times gone by, the way to represent large data objects in the database was to use the `LONG`, `RAW`, and `LONG RAW` types. Oracle recommends that current applications use the various LOB types, such as `CLOB`, `BLOB` and `BFILE`, for this data.

See the *Oracle9i Application Developer's Guide - Large Objects (LOBs)* , for information about LOB datatypes.

The following sections deal with ways to migrate data from the older datatypes to the LOB types. The LOB types can be used in many situations that formerly required other types such as `LONG` or `VARCHAR2`.

## Migrating LONG Datatypes to LOB Datatypes

The LONG datatype can store variable-length character data containing up to two gigabytes of information, depending upon available memory. LONG columns have many of the characteristics of VARCHAR2 columns. You can use them in SELECT lists, SET clauses of UPDATE statements, and VALUES clauses of INSERT statements.

Oracle Corporation recommends using the LONG datatype only for backward compatibility with old applications. For new applications, you should use the CLOB and NCLOB datatypes for large amounts of character data. Typically, you can change LONG data to LOBs in your tables without changing existing applications. SQL, PL/SQL, and OCI interfaces for LONG data can all work on LOB data as well.

**See Also:**

- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for information about the CLOB and NCLOB datatypes.
- *Oracle Call Interface Programmer's Guide* for details about each of the OCI functions.
- *Oracle9i SQL Reference* for syntax of the ALTER TABLE command.

### Changing a LONG or LONG RAW Column to a LOB Datatype

You can use the ALTER TABLE command to change the underlying datatype of a column from LONG to CLOB, or LONG RAW to BLOB. For example:

```
ALTER TABLE employees MODIFY (resume BLOB) DISABLE STORAGE IN ROW;
ALTER TABLE newspaper MODIFY (article CLOB DEFAULT 'Has not been written yet');
```

This technique preserves all the constraints and triggers on the table. All indexes must be rebuilt. Any domain indexes on a long column, such as indexes for data cartridge or interMedia applications, must be dropped before changing the type of the column.

#### Restrictions on Changing LONG or LONG RAW Columns to LOB Datatypes

1. LOBs are not allowed in clustered tables. So if a table is a part of a cluster, its LONG or LONG RAW column cannot be changed to LOB.
2. If a table is replicated or has materialized views, and its LONG column is changed to LOB, you might have to manually fix the replicas.
3. Not all triggers are preserved when the column is changed to a LOB datatype. LOB columns are not allowed in the column list of an UPDATE trigger. For example, the following trigger becomes invalid after changing the type of the column to a LOB, and cannot be recompiled:

```
CREATE TABLE t(changed_col LONG);
CREATE TRIGGER trig BEFORE UPDATE OF lobcol ON t ...;
```

### Transparent Access to LOBs from Applications that Use LONG and LONG RAW Datatypes

If your application uses DML (INSERT, UPDATE, DELETE) statements from SQL or PL/SQL for LONG or LONG RAW data, these statements work the same after the column is converted to a LOB datatype. You can use input parameters and output buffers of various character types, and they are converted to and from the corresponding LOB datatypes, and truncated if the output type is not large enough to hold the entire result. For example, you can SELECT a CLOB into a character variable, or a BLOB into a RAW variable.

The following SQL functions that accept or output character types now accept or output CLOB data as well:

||, CONCAT, INSTR, INSTRB, LENGTH, LENGTHB, LIKE, LOWER, LPAD, LTRIM, NLS_LOWER, NLS_UPPER, NVL, REPLACE, RPAD, RTRIM, SUBSTR, SUBSTRB, TRIM, UPPER

In PL/SQL, all the SQLfunctions listed above and the comparison operators (>, =, < and !=), and all user-defined procedures and functions, accept CLOB datatypes as parameters or output types. You can also assign a CLOB to a character variable and vice versa in PL/SQL.

If your application uses OCI calls to perform piecewise inserts, updates, or fetches of LONG data, these calls work the same after the column is converted to a LOB datatype. You can define a CLOB column as SQLT_CHR or a BLOB column as SQLT_BIN and select the LOB data directly into a CHARACTER or RAW buffer without selecting out the locator first. The OCI functions that provide this transparent access, by accepting datatypes of SQLT_LNG, SQLT_CHR, SQLT_BIN, and SQLT_LBI) are:

- OCIBindByName()
- OCIBindByPos()
- OCIDefineByPos()

### Restrictions on LONG and LONG RAW Datatypes

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 some 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.
- A stored function cannot return a LONG value.
- You can declare a variable or argument of a PL/SQL program unit using the LONG datatype. 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, 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.
- A table with LONG columns cannot be stored in a tablespace with automatic segment-space management.

LONG columns cannot appear in certain 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 datatype 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 datatype (such as CHAR and VARCHAR2), a LONG column can be referenced in a SQL statement within a trigger.
- Variables in triggers cannot be declared using the LONG datatype.
- :NEW and :OLD cannot be used with LONG columns.

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

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

> **Note:** If you design tables containing LONG or LONG RAW data, then you should place each LONG or LONG RAW column in its own table, along with a primary key value that lets you retrieve the LONG or LONG RAW columns through joins. This way, you can query the other columns without reading large amounts of irrelevant data.

### Example of LONG Datatype

To store information on magazine articles, including the texts of each article, create two tables. For example:

```
CREATE TABLE Article_header
    (Id               NUMBER  PRIMARY KEY,
    Title             VARCHAR2(200),
    First_author      VARCHAR2(30),
    Journal           VARCHAR2(50),
    Pub_date          DATE);

CREATE TABLE article_text
    (Id               NUMBER
                      REFERENCES
                      Article_header,
    Text              LONG);
```

The ARTICLE_TEXT table stores only the text of each article. The ARTICLE_HEADER table stores all other information about the article, including the title, first author, and journal and date of publication. The two tables are related by the referential integrity constraint on the ID column of each table.

This design allows SQL statements to query data other than the text of an article without reading through the text. If you want to select all first authors published in Nature magazine during July 1991, then you can issue this statement that queries the ARTICLE_HEADER table:

```
SELECT First_author
    FROM Article_header
    WHERE Journal = 'NATURE'
        AND TO_CHAR(Pub_date, 'MM YYYY') = '07 1991';
```

If the text of each article were stored in the same table with the first author, publication, and publication date, then Oracle would need to read through the text to perform this query.

## Using RAW and LONG RAW Datatypes

---

**Note:** The RAW and LONG RAW datatypes are provided for backward compatibility with existing applications. For new applications, you should use the BLOB and BFILE datatypes for large amounts of binary data.

---

**See Also:** See *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for information about the BLOB and BFILE datatypes.

The RAW and LONG RAW datatypes store data that is not interpreted by Oracle (that is, not converted when moving data between different systems). These datatypes are intended for binary data and byte strings. For example, LONG RAW can store graphics, sound, documents, and arrays of binary data; the interpretation is dependent on the use.

Oracle Net and the Export and Import utilities do not perform character conversion when transmitting RAW or LONG RAW data. When Oracle automatically converts RAW or LONG RAW data to and from CHAR data (as is the case when entering RAW data as a literal in an INSERT statement), the data is represented as one hexadecimal

character representing the bit pattern for every four bits of RAW data. For example, one byte of RAW data with bits 11001011 is displayed and entered as 'CB'.

LONG RAW data cannot be indexed, but RAW data can be indexed.

> **See Also:** For more information about restrictions on LONG RAW data, see "Restrictions on LONG and LONG RAW Datatypes" on page 3-29.

# Addressing Rows Directly with the ROWID Datatype

Every row in an Oracle table is assigned a ROWID that corresponds to the physical address of a row. If the row is too large to fit within a single data block, the ROWID identifies the initial row piece. Although ROWIDs are usually unique, different rows can have the same ROWID if they are in the same data block, but in different clustered tables.

Each table in an Oracle database has a pseudocolumn named ROWID.

> **See Also:** *Oracle9i Database Concepts* for general information about the ROWID pseudocolumn and the ROWID datatype.

## Extended ROWID Format

The Oracle Server uses an *extended ROWID* format, which supports features such as table partitions, index partitions, and clusters.

The extended ROWID includes the following information:

- Data object (segment) identifier
- Datafile identifier
- Block identifier
- Row identifier

The data object identifier is an identification number that Oracle assigns to schema objects in the database, such as nonpartitioned tables or partitions. For example:

```
SELECT DATA_OBJECT_ID FROM ALL_OBJECTS
       WHERE OWNER = 'SCOTT' AND OBJECT_NAME = 'EMP_TAB';
```

This query returns the data object identifier for the EMP_TAB table in the SCOTT schema.

> **See Also:** *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about other ways to get the data object identifier, using the DBMS_ROWID package functions.

### Different Forms of the ROWID

Oracle documentation uses the term ROWID in different ways, depending on context.

**ROWID Pseudocolumn** Each table and nonjoined view has a pseudocolumn called ROWID. For example:

```
CREATE TABLE T_tab (col1 Rowid);
INSERT INTO T_tab SELECT Rowid FROM Emp_tab WHERE Empno = 7499;
```

This command returns the ROWID pseudocolumn of the row of the EMP_TAB table that satisfies the query, and inserts it into the T1 table.

**Internal ROWID** The internal ROWID is an internal structure that holds information that the server code needs to access a row. The restricted internal ROWID is 6 bytes on most platforms; the extended ROWID is 10 bytes on these platforms.

**External Character ROWID** The extended ROWID pseudocolumn is returned to the client in the form of an 18-character string (for example, "AAAA8mAALAAAAQkAAA"), which represents a base 64 encoding of the components of the extended ROWID in a four-piece format, OOOOOOFFFBBBBBBRRR:

- OOOOOO: The **data object number** identifies the database segment (AAAA8m in the example). Schema objects in the same segment, such as a cluster of tables, have the same data object number.
- FFF: The **datafile** that contains the row (file AAL in the example). File numbers are unique within a database.
- BBBBBB: The **data block** that contains the row (block AAAAQk in the example). Block numbers are relative to their datafile, *not* tablespace. Therefore, two rows with identical block numbers could reside in two different datafiles of the same tablespace.
- RRR: The **row** in the block (row AAA in the example).

There is no need to decode the external ROWID; you can use the functions in the DBMS_ROWID package to obtain the individual components of the extended ROWID.

> **See Also:** *Oracle9i Supplied PL/SQL Packages and Types Reference* for information about the DBMS_ROWID package.

The restricted ROWID pseudocolumn is returned to the client in the form of an 18-character string with a hexadecimal encoding of the datablock, row, and datafile components of the ROWID.

**External Binary ROWID** Some client applications use a binary form of the ROWID. For example, OCI and some precompiler applications can map the ROWID to a 3GL structure on bind or define calls. The size of the binary ROWID is the same for extended and restricted ROWIDs. The information for the extended ROWID is included in an unused field of the restricted ROWID structure.

The format of the extended binary ROWID, expressed as a C struct, is:

```
struct riddef {
    ub4    ridobjnum; /* data obj#--this field is
                         unused in restricted ROWIDs */
    ub2    ridfilenum;
    ub1    filler;
    ub4    ridblocknum;
    ub2    ridslotnum;
}
```

### ROWID Migration and Compatibility Issues

For backward compatibility, the restricted form of the ROWID is still supported. These ROWIDs exist in Oracle7 data, and the extended form of the ROWID is required only in global indexes on partitioned tables. New tables always get extended ROWIDs.

> **See Also:** *Oracle9i Database Administrator's Guide.*

It is possible for an Oracle7 client to access a more recent database, and vice versa. A client in this sense can include a remote database accessing a server using database links, as well as a client 3GL or 4GL application accessing a server.

> **See Also:** There is more information on the ROWID_TO_EXTENDED function in *Oracle9i Supplied PL/SQL Packages and Types Reference* and *Oracle9i Database Migration*.

**Accessing an Oracle7 Database from an Oracle9i Client** The ROWID values that are returned are always restricted ROWIDs. Also, Oracle9*i* uses restricted ROWIDs when returning a ROWID value to an Oracle7 or earlier server.

The following ROWID functionality works when accessing an Oracle7 Server:

- Selecting a ROWID and using the obtained value in a WHERE clause
- WHERE CURRENT OF cursor operations
- Storing ROWIDs in user columns of ROWID or CHAR type
- Interpreting ROWIDs using the hexadecimal encoding (not recommended, use the DBMS_ROWID functions)

**Accessing an Oracle9i Database from an Oracle7 Client** Oracle9*i* returns ROWIDs in the extended format. This means that you can only:

- Select a ROWID and use it in a WHERE clause
- Use WHERE CURRENT OF cursor operations
- Store ROWIDs in user columns of CHAR(18) datatype

**Import and Export** It is not possible for an Oracle7 client to import a table from a later version that has a ROWID column (not the ROWID pseudocolumn), if any row of the table contains an extended ROWID value.

# ANSI/ISO, DB2, and SQL/DS Datatypes

You can define columns of tables in an Oracle database using ANSI/ISO, DB2, and SQL/DS datatypes. Oracle internally converts such datatypes to Oracle datatypes.

The ANSI datatype conversions to Oracle datatypes are shown in Table 3–3. The ANSI/ISO datatypes NUMERIC, DECIMAL, and DEC can specify only fixed-point numbers. For these datatypes, s defaults to 0.

***Table 3–3 ANSI Datatype Conversions to Oracle Datatypes***

| ANSI SQL Datatype | Oracle Datatype |
|---|---|
| CHARACTER (n), CHAR (n) | CHAR (n) |
| NUMERIC (p,s), DECIMAL (p,s), DEC (p,s) | NUMBER (p,s) |
| INTEGER, INT, SMALLINT | NUMBER (38) |
| FLOAT (p) | FLOAT (p) |
| REAL | FLOAT (63) |
| DOUBLE PRECISION | FLOAT (126) |
| CHARACTER VARYING(n), CHAR VARYING(n) | VARCHAR2 (n) |
| TIMESTAMP | TIMESTAMP |
| TIMESTAMP WITH TIME ZONE | TIMESTAMP WITH TIME ZONE |

The IBM products SQL/DS, and DB2 datatypes TIME, GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC have no corresponding Oracle datatype and cannot be used.

Table 3–4 shows the DB2 and SQL/DS conversions.

***Table 3–4 SQL/DS, DB2 Datatype Conversions to Oracle Datatypes***

| DB2 or SQL/DS Datatype | Oracle Datatype |
|---|---|
| CHARACTER (n) | CHAR (n) |
| VARCHAR (n) | VARCHAR2 (n) |
| LONG VARCHAR | LONG |
| DECIMAL (p,s) | NUMBER (p,s) |
| INTEGER, SMALLINT | NUMBER (38) |
| FLOAT (p) | FLOAT (p) |

*Table 3–4 SQL/DS, DB2 Datatype Conversions to Oracle Datatypes*

| DB2 or SQL/DS Datatype | Oracle Datatype |
|---|---|
| DATE | DATE |
| TIMESTAMP | TIMESTAMP |

# How Oracle Converts Datatypes

In some cases, Oracle allows data of one datatype where it expects data of a different datatype. Generally, an expression cannot contain values with different datatypes. However, Oracle can use the following functions to automatically convert data to the expected datatype:

- TO_NUMBER()
- TO_CHAR()
- TO_NCHAR()
- TO_DATE()
- HEXTORAW()
- RAWTOHEX()
- RAWTONHEX()
- ROWIDTOCHAR()
- ROWIDTONCHAR()
- CHARTOROWID()
- TO_CLOB()
- TO_NCLOB()
- TO_BLOB()
- TO_RAW()

Implicit datatype conversions work according to the rules explained below.

## Datatype Conversion During Assignments

For assignments, Oracle can automatically convert the following:

- VARCHAR2, NVARCHAR2, CHAR, or NCHAR to NUMBER

- NUMBER to VARCHAR2 or NVARCHAR2
- VARCHAR2, NVARCHAR2, CHAR, or NCHAR to DATE
- DATE to VARCHAR2 or NVARCHAR2
- VARCHAR2, NVARCHAR2, CHAR, or NCHAR to ROWID
- ROWID to VARCHAR2 or NVARCHAR2
- VARCHAR2, NVARCHAR2, CHAR, NCHAR, or LONG to CLOB
- VARCHAR2, NVARCHAR2, CHAR, NCHAR, or LONG to NCLOB
- CLOB to CHAR, NCHAR, VARCHAR2, NVARCHAR2, and LONG
- NCLOB to CHAR, NCHAR, VARCHAR2, NVARCHAR2, and LONG
- NVARCHAR2, NCHAR, or BLOB to RAW
- RAW to BLOB
- VARCHAR2 or CHAR to HEX
- HEX to VARCHAR2

The assignment succeeds if Oracle can convert the datatype of the value used in the assignment to that of the assignment's target.

For the examples in the following list, assume a package with a public variable and a table declared as in the following statements:

---

**Note:** You may need to set up the following data structures for certain examples to work:

```
CREATE PACKAGE Test_Pack AS var1 CHAR(5); END;
CREATE TABLE Table1_tab (col1 NUMBER);
```

---

- `variable := expression`

  The datatype of *expression* must be either the same as, or convertible to, the datatype of *variable*. For example, Oracle automatically converts the data provided in the following assignment within the body of a stored procedure:

  ```
  VAR1 := 0;
  ```

- `INSERT INTO table VALUES (expression1, expression2, ...)`

The datatypes of *expression1*, *expression2*, and so on, must be either the same as, or convertible to, the datatypes of the corresponding columns in *table*. For example, Oracle automatically converts the data provided in the following INSERT statement for TABLE1 (see table definition above):

```
INSERT INTO Table1_tab VALUES ('19');
```

- UPDATE *table* SET *column* = *expression*

  The datatype of *expression* must be either the same as, or convertible to, the datatype of *column*. For example, Oracle automatically converts the data provided in the following UPDATE statement issued against TABLE1:

  ```
  UPDATE Table1_tab SET col1 = '30';
  ```

- SELECT *column* INTO *variable* FROM *table*

  The datatype of *column* must be either the same as, or convertible to, the datatype of *variable*. For example, Oracle automatically converts data selected from the table before assigning it to the variable in the following statement:

  ```
  SELECT Col1 INTO Var1 FROM Table1_tab WHERE Col1 = 30;
  ```

## Datatype Conversion During Expression Evaluation

For expression evaluation, Oracle can automatically perform the same conversions as for assignments. An expression is converted to a type based on its context. For example, operands to arithmetic operators are converted to NUMBER, and operands to string functions are converted to VARCHAR2.

Oracle can automatically convert the following:

- VARCHAR2 or CHAR to NUMBER
- VARCHAR2 or CHAR to DATE

Character to NUMBER conversions succeed only if the character string represents a valid number. Character to DATE conversions succeed only if the character string satisfies the session default format, which is specified by the initialization parameter NLS_DATE_FORMAT.

Some common types of expressions follow:

- Simple expressions, such as:

  ```
  commission + '500'
  ```

- Boolean expressions, such as:

  ```
  bonus > salary / '10'
  ```

- Function and procedure calls, such as:

  ```
  MOD (counter, '2')
  ```

- WHERE clause conditions, such as:

  ```
  WHERE hiredate = TO_DATE('1997-01-01','yyyy-mm-dd')
  ```

- WHERE clause conditions, such as:

  ```
  WHERE rowid = 'AAAAaoAATAAAADAAA'
  ```

In general, Oracle uses the rule for expression evaluation when a datatype conversion is needed in places not covered by the rule for assignment conversions.

In assignments of the form:

```
variable := expression
```

Oracle first evaluates *expression* using the conversion rules for expressions; *expression* can be as simple or complex as desired. If it succeeds, then the evaluation of *expression* results in a single value and datatype. Then, Oracle tries to assign this value to the target variable using the conversion rules for assignments.

## Representing Dynamically Typed Data

You might be familiar with features in some languages that allow datatypes to change at runtime, or 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`. Oracle9*i* includes features that let you create variables and columns that can hold data of any type, and test such data values to see their underlying representation. Using these features, a single table column can represent a numeric value in one row, a string value in another row, and an object in another row.

You can use the built-in type `SYS.ANYDATA` to represent values of any scalar or object type. This type is an object type with methods to bring in a scalar value of any type, and turn the value back into a scalar or object.

In the same way, you can use the built-in type `SYS.ANYDATASET` to represent values of any collection type.

To manipulate and check type information, you can use the built-in type SYS.ANYTYPE in combination with the DBMS_TYPES package. For example, the following program represents data of different underlying types in a table, then interprets the underlying type of each row and processes each value appropriately:

```
-- The example below defines and executes a PL/SQL procedure that
-- uses methods built into SYS.ANYDATA to access information about
-- data stored in a SYS.ANYDATA table column.

DROP TYPE Employee FORCE;
DROP TABLE mytab;
CREATE OR REPLACE TYPE Employee AS OBJECT ( empno NUMBER,
  ename VARCHAR2(10) );
/
CREATE TABLE mytab ( id NUMBER, data SYS.ANYDATA );

INSERT INTO mytab VALUES (1, SYS.ANYDATA.ConvertNumber(5));
INSERT INTO mytab VALUES (2, SYS.ANYDATA.ConvertObject(Employee(5555, 'john')));
commit;

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;
  non_null_anytype_for_NUMBER exception;
  unknown_typename            exception;
BEGIN
  OPEN cur;
  LOOP
    FETCH cur INTO v_id, v_data;
    EXIT WHEN cur%NOTFOUND;

/* The typecode is a number that signifies what type is 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 */ );

/* Now we compare the typecode against constants from DBMS_TYPES to see what
   kind of data we have, and decide how to display it. */
```

```
    CASE v_typecode

      WHEN Dbms_Types.Typecode_NUMBER THEN
        IF v_type IS NOT NULL
-- This condition should never happen, but we check just in case.
          THEN 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();
-- An object type's name is qualified with the schema name.
        IF v_typename NOT IN ( 'SCOTT.EMPLOYEE' )
-- If we encounter any object type besides EMPLOYEE, raise an exception.
          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;
  CLOSE cur;
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 SCOTT.EMPLOYEE' );
END;
/
-- The query and the procedure P in the preceding code sample
-- produce output like the following:

SQL> SELECT t.data.gettypename() FROM mytab t;

T.DATA.GETTYPENAME()
--------------------------------------------------------------------------------
SYS.NUMBER
SCOTT.EMPLOYEE

SQL> EXEC P;
```

```
1: NUMBER = 5
2: user-defined type = SCOTT.EMPLOYEE ( 5555, john )
```

You can access the same features through the OCI interface, using the `OCIType`, `OCIAnyData`, and `OCIAnyDataSet` interfaces.

**See Also:**

*Oracle9i Supplied PL/SQL Packages and Types Reference* for details about the `DBMS_TYPES` package.

*Oracle9i Application Developer's Guide - Object-Relational Features* for information and examples using the `ANYDATA`, `ANYDATASET`, and `ANYTYPE` types.

*Oracle Call Interface Programmer's Guide* for details about the OCI interfaces.

## Representing XML Data

If you have information stored as files in XML format, or if you want to take an object type and store it as XML, you can use the `XMLType` built-in type.

`XMLType` columns store their data as `CLOB`s. You can take an existing `CLOB`, `VARCHAR2`, or any object type, and call the `XMLType` constructor to turn it into an XML object.

Once an XML object is inside the database, you can use queries to traverse it (using the XML XPath notation) and extract all or part of its data.

You can also produce XML output from existing relational data, and split XML documents across relational tables and columns. You can use the `DBMS_XMLQUERY`, `DBMS_XMLGEN`, and `DBMS_XMLSAVE` packages, and the `SYS_XMLGEN` and `SYS_XMLAGG` functions to transfer XML data into and out of relational tables.

**See Also:**

- *Oracle9i XML Developer's Kits Guide - XDK* for information about all aspects of working with XML.
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for details about the `XMLType` type and the `DBMS_XMLQuery`, `DBMS_XMLGEN`, and `DBMS_XMLSave` packages.
- *Oracle9i SQL Reference* for information about the `SYS_XMLGEN` and `SYS_XMLAGG` functions.

# 4

# Maintaining Data Integrity Through Constraints

This chapter explains how to enforce the business rules associated with your database and prevent the entry of invalid information into tables by using integrity constraints. Topics include the following:

# Overview of Integrity Constraints

You can define integrity constraints to enforce business rules on data in your tables. Business rules specify conditions and relationships that must always be true, or must always be false. Because each company defines its own policies about things like salaries, employee numbers, inventory tracking, and so on, you can specify a different set of rules for each database table.

When an integrity constraint applies to a table, all data in the table must conform to the corresponding rule. When you issue a SQL statement that modifies data in the table, Oracle ensures that the new data satisfies the integrity constraint, without the need to do any checking within your program.

## When to Enforce Business Rules with Integrity Constraints

You can enforce rules by defining integrity constraints more reliably than by adding logic to your application. Oracle can check that all the data in a table obeys an integrity constraint faster than an application can.

### Example of an Integrity Constraint for a Business Rule

To ensure that each employee works for a valid department, first create a rule that all values in the department table are unique :

```
ALTER TABLE Dept_tab
    ADD PRIMARY KEY (Deptno);
```

Then, create a rule that every department listed in the employee table must match one of the values in the department table:

```
ALTER TABLE Emp_tab
    ADD FOREIGN KEY (Deptno) REFERENCES Dept_tab(Deptno);
```

When you add a new employee record to the table, Oracle automatically checks that its department number appears in the department table.

To enforce this rule without integrity constraints, you can use a trigger to query the department table and test that each new employee's department is valid. But this method is less reliable than the integrity constrain, because SELECT in Oracle uses "consistent read" and so the query might miss uncommitted changes from other transactions.

## When to Enforce Business Rules in Applications

You might enforce business rules through application logic as well as through integrity constraints, if you can filter out bad data before attempting an insert or update. This might let you provide instant feedback to the user, and reduce the load on the database. This technique is appropriate when you can determine that data values are wrong or out of range, without checking against any data already in the table.

## Creating Indexes for Use with Constraints

All enabled unique and primary keys require corresponding indexes. You should create these indexes by hand, rather than letting the database create them for you. Note that:

- Constraints use existing indexes where possible, rather than creating new ones.
- Unique and primary keys can use non-unique as well as unique indexes. They can even use just the first few columns of non-unique indexes.
- At most one unique or primary key can use each non-unique index.
- The column orders in the index and the constraint do not need to match.
- If you need to check whether an index is used by a constraint, for example when you want to drop the index, the object number of the index used by a unique or primary key constraint is stored in CDEF$.ENABLED for that constraint. It is not shown in any catalog view.

You should almost always index foreign keys, and the database does not do this for you.

## When to Use NOT NULL Integrity Constraints

By default, all columns can contain nulls. Only define `NOT NULL` constraints for columns of a table that absolutely require values at all times.

For example, a new employee's manager or hire date might be temporarily omitted. Some employees might not have a commission. Columns like these should not have `NOT NULL` integrity constraints. However, an employee name might be required from the very beginning, and you can enforce this rule with a `NOT NULL` integrity constraint.

`NOT NULL` constraints are often combined with other types of integrity constraints to further restrict the values that can exist in specific columns of a table. Use the combination of `NOT NULL` and `UNIQUE` key integrity constraints to force the input of

values in the UNIQUE key; this combination of data integrity rules eliminates the possibility that any new row's data will ever attempt to conflict with an existing row's data.

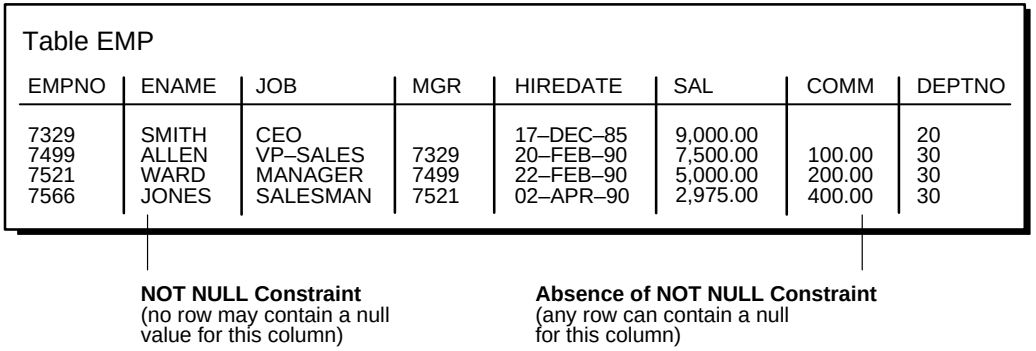
Because Oracle indexes do not store keys that are all null, if you want to allow index-only scans of the table or some other operation that requires indexing all rows, put a NOT NULL constraint on at least one indexed column.

**See Also:** "Defining Relationships Between Parent and Child Tables" on page 4-11

A NOT NULL constraint is specified like this:

```
ALTER TABLE emp MODIFY ename NOT NULL;
```

***Figure 4–1 Table with NOT NULL Integrity Constraints***




## When to Use Default Column Values

Assign default values to columns that contain a typical value. For example, in the DEPT_TAB table, if most departments are located at one site, then the default value for the LOC column can be set to this value (such as NEW YORK).

Default values can help avoid errors where there is a number, such as zero, that applies to a column that has no entry. For example, a default value of zero can simplify testing, by changing a test like this:

```
IF sal IS NOT NULL AND sal < 50000
```

to the simpler form:

```
IF sal < 50000
```

Depending upon your business rules, you might use default values to represent zero or false, or leave the default values as NULL to signify an unknown value.

Defaults are also useful when you use a view to make a subset of a table's columns visible. For example, you might allow users to insert rows through a view. The base table might also have a column named INSERTER, not included in the definition of the view, to log the user that inserts each row. To record the user name automatically, define a default value that calls the USER function:

```
CREATE TABLE audit_trail
(
    value1   NUMBER,
    value2   VARCHAR2(32),
    inserter VARCHAR2(30) DEFAULT USER
);
```

> **See Also:** For another example of assigning a default column value, refer to the section "Creating Tables".

## Setting Default Column Values

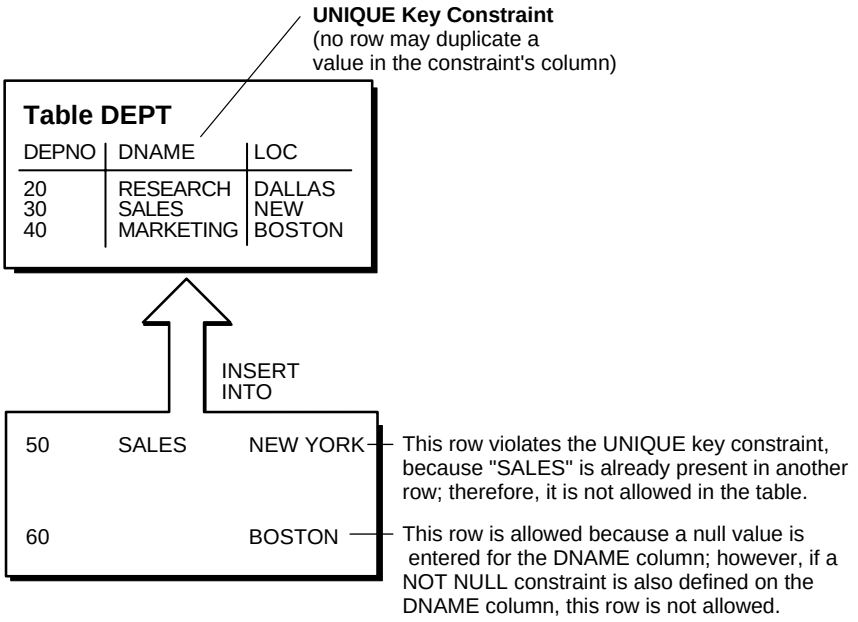
Default values can include any literal, or almost any expression, including calls to SYSDATE, SYS_CONTEXT, USER, USERENV, and UID. Default values cannot include expressions that refer to a sequence, PL/SQL function, column, LEVEL, ROWNUM, or PRIOR. The datatype of a default literal or expression must match or be convertible to the column datatype.

Sometimes the default value is the result of a SQL function. For example, a call to SYS_CONTEXT can set a different default value depending on conditions such as the user name. To be used as a default value, a SQL function must have parameters that are all literals, cannot reference any columns, and cannot call any other functions.

If you do not explicitly define a default value for a column, the default for the column is implicitly set to NULL.

You can use the keyword DEFAULT within an INSERT statement instead of a literal value, and the corresponding default value is inserted.

***Figure 4–2 Table with a UNIQUE Key Constraint***




## Choosing a Table's Primary Key

Each table can have one primary key, which uniquely identifies each row in a table and ensures that no duplicate rows exist. Use the following guidelines when selecting a primary key:

- Whenever practical, use a column containing a sequence number. It is a simple way to satisfy all the other guidelines.
- Minimize your use of composite primary keys. Although composite primary keys are allowed, they do not satisfy all of the other recommendations. For example, composite primary key values are long and cannot be assigned by sequence numbers.
- Choose a column whose data values are unique, because the purpose of a primary key is to uniquely identify each row of the table.
- Choose a column whose data values are never changed. A primary key value is only used to identify a row in the table, and its data should never be used for

any other purpose. Therefore, primary key values should rarely or never be changed.

- Choose a column that does not contain any nulls. A PRIMARY KEY constraint, by definition, does not allow any row to contain a null in any column that is part of the primary key.
- Choose a column that is short and numeric. Short primary keys are easy to type. You can use sequence numbers to easily generate numeric primary keys.

## When to Use UNIQUE Key Integrity Constraints

Choose columns for unique keys carefully. The purpose of these contraints is different from that of primary keys. Unique key constraints are appropriate for any column where duplicate values are not allowed. Primary keys identify each row of the table uniquely, and typically contain values that have no significance other than being unique.

---

**Note:** Although UNIQUE key constraints allow null values, you cannot have identical values in the non-null columns of a composite UNIQUE key constraint.

---

Some examples of good unique keys include:

- An employee's social security number (the primary key is the employee number)
- A truck's license plate number (the primary key is the truck number)
- A customer's phone number, consisting of the two columns AREA and PHONE (the primary key is the customer number)
- A department's name and location (the primary key is the department number)

## Constraints On Views for Performance, Not Data Integrity

The constraints discussed throughout this chapter apply to tables, not views.

Although you can declare constraints on views, such constraints do not help maintain data integrity. Instead, they are used to enable query rewrites on queries involving views, which helps performance with materialized views and other data warehousing features. Such constraints are always declared with the DISABLE keyword, and you cannot use the VALIDATE keyword. The constraints are never enforced, and there is no associated index.

> **See Also:** *Oracle9i Data Warehousing Guide* for information on query rewrite, materialized views, and the performance reasons for declaring constraints on views.

# Enforcing Referential Integrity with Constraints

Whenever two tables contain one or more common columns, Oracle can enforce the relationship between the two tables through a referential integrity constraint. Define a `PRIMARY` or `UNIQUE` key constraint on the column in the parent table (the one that has the complete set of column values). Define a `FOREIGN KEY` constraint on the column in the child table (the one whose values must refer to existing values in the other table).

> **See Also:** Depending on this relationship, you may want to define additional integrity constraints including the foreign key, as listed in the section "Defining Relationships Between Parent and Child Tables" on page 4-11.

Figure 4–3 shows a foreign key defined on the department number. It guarantees that every value in this column must match a value in the primary key of the department table. This constraint prevents erroneous department numbers from getting into the employee table.

Foreign keys can be comprised of multiple columns. Such a **composite foreign key** must reference a composite primary or unique key of the exact same structure, with the same number of columns and the same datatypes. Because composite primary and unique keys are limited to 32 columns, a composite foreign key is also limited to 32 columns.

## About Nulls and Foreign Keys

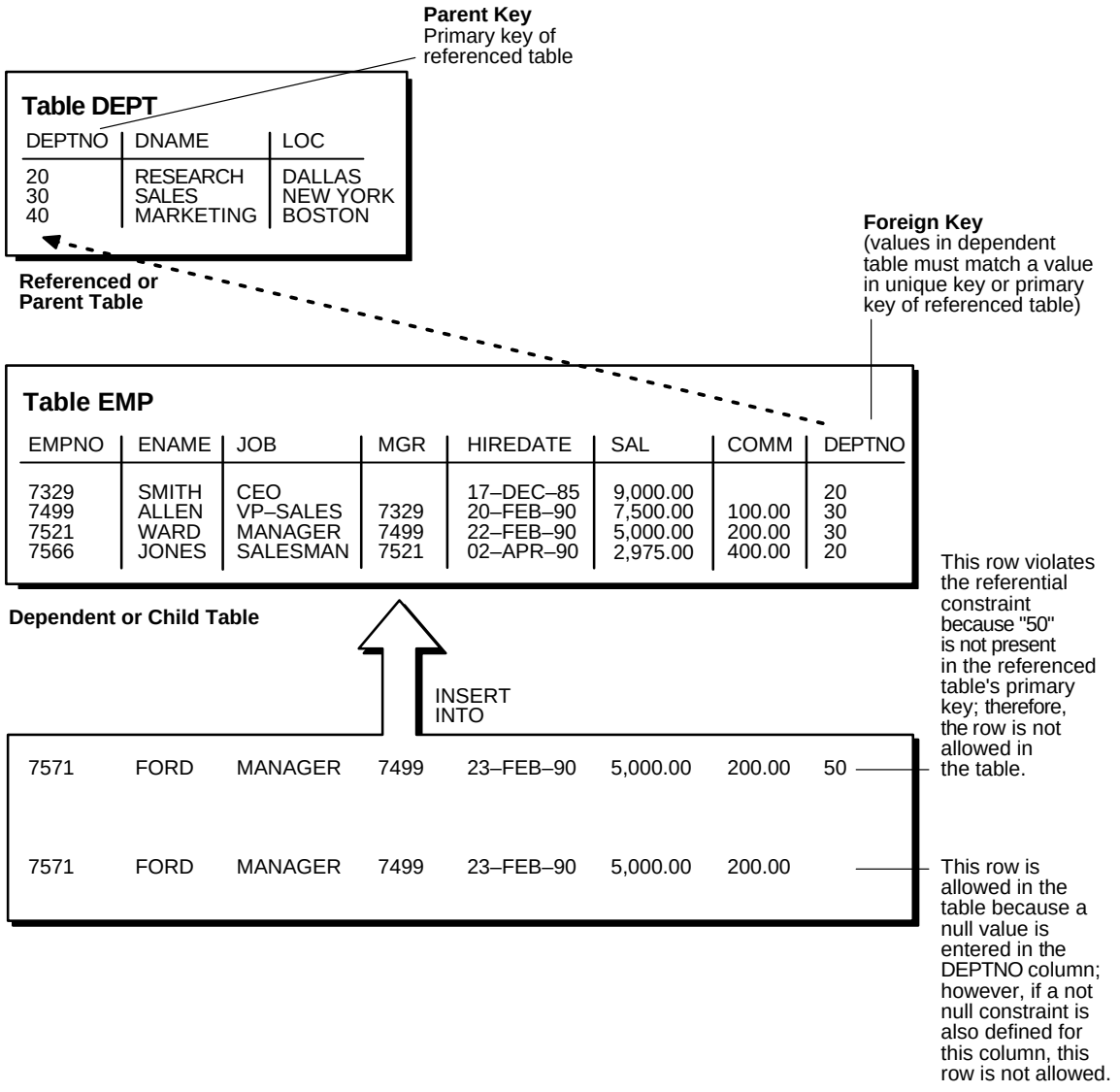
Foreign keys allow key values that are all null, even if there are no matching `PRIMARY` or `UNIQUE` keys.

- By default (without any `NOT NULL` or `CHECK` clauses), the `FOREIGN KEY` constraint enforces the "match none" rule for composite foreign keys in the ANSI/ISO standard.
- To enforce the "match full" rule for nulls in composite foreign keys, which requires that all components of the key be null or all be non-null, define a `CHECK` constraint that allows only all nulls or all non-nulls in the composite foreign key. For example, with a composite key comprised of columns A, B, and C:

```
CHECK ((A IS NULL AND B IS NULL AND C IS NULL) OR
       (A IS NOT NULL AND B IS NOT NULL AND C IS NOT NULL))
```

- In general, it is not possible to use declarative referential integrity to enforce the "match partial" rule for nulls in composite foreign keys, which requires the non-null portions of the key to appear in the corresponding portions in the primary or unique key of a single row in the referenced table. You can often use triggers to handle this case, as described in Chapter 15, "Using Triggers".

***Figure 4–3 Tables with Referential Integrity Constraints***



## Defining Relationships Between Parent and Child Tables

Several relationships between parent and child tables can be determined by the other types of integrity constraints defined on the foreign key in the child table.

**No Constraints on the Foreign Key** When no other constraints are defined on the foreign key, any number of rows in the child table can reference the same parent key value. This model allows nulls in the foreign key.

This model establishes a "one-to-many" relationship between the parent and foreign keys that allows undetermined values (nulls) in the foreign key. An example of such a relationship is shown in Figure 4–3 on page 8 between the employee and department tables. Each department (parent key) has many employees (foreign key), and some employees might not be in a department (nulls in the foreign key).

**NOT NULL Constraint on the Foreign Key** When nulls are not allowed in a foreign key, each row in the child table must explicitly reference a value in the parent key because nulls are not allowed in the foreign key. However, any number of rows in the child table can reference the same parent key value.

This model establishes a "one-to-many" relationship between the parent and foreign keys. However, each row in the child table must have a reference to a parent key value; the absence of a value (a null) in the foreign key is not allowed. The same example in the previous section can be used to illustrate such a relationship. However, in this case, employees must have a reference to a specific department.

**UNIQUE Constraint on the Foreign Key** When a `UNIQUE` constraint is defined on the foreign key, one row in the child table can reference a parent key value. This model allows nulls in the foreign key.

This model establishes a "one-to-one" relationship between the parent and foreign keys that allows undetermined values (nulls) in the foreign key. For example, assume that the employee table had a column named `MEMBERNO`, referring to an employee's membership number in the company's insurance plan. Also, a table named `INSURANCE` has a primary key named `MEMBERNO`, and other columns of the table keep respective information relating to an employee's insurance policy. The `MEMBERNO` in the employee table should be both a foreign key and a unique key:

- To enforce referential integrity rules between the `EMP_TAB` and `INSURANCE` tables (the `FOREIGN KEY` constraint)
- To guarantee that each employee has a unique membership number (the `UNIQUE` key constraint)

**UNIQUE and NOT NULL Constraints on the Foreign Key** When both UNIQUE and NOT NULL constraints are defined on the foreign key, only one row in the child table can reference a parent key value. Because nulls are not allowed in the foreign key, each row in the child table must explicitly reference a value in the parent key.

This model establishes a "one-to-one" relationship between the parent and foreign keys that does not allow undetermined values (nulls) in the foreign key. If you expand the previous example by adding a NOT NULL constraint on the MEMBERNO column of the employee table, in addition to guaranteeing that each employee has a unique membership number, you also ensure that no undetermined values (nulls) are allowed in the MEMBERNO column of the employee table.

## Rules for Multiple FOREIGN KEY Constraints

Oracle allows a column to be referenced by multiple FOREIGN KEY constraints; effectively, there is no limit on the number of dependent keys. This situation might be present if a single column is part of two different composite foreign keys.

## Deferring Constraint Checks

When Oracle checks a constraint, it signals an error if the constraint is not satisfied. You can use the SET CONSTRAINTS statement to defer checking the validity of constraints until the end of a transaction.

---

**Note:** You cannot issue a SET CONSTRAINT statement inside a trigger.

---

The SET CONSTRAINTS setting lasts for the duration of the transaction, or until another SET CONSTRAINTS statement resets the mode.

---

**See Also:** For more details about the SET CONSTRAINTS statement, see the *Oracle9i SQL Reference.*

---

### Guidelines for Deferring Constraint Checks

**Select Appropriate Data** You may wish to defer constraint checks on UNIQUE and FOREIGN keys if the data you are working with has any of the following characteristics:

- Tables are snapshots

- Tables that contain a large amount of data being manipulated by another application, which may or may not return the data in the same order
- Update cascade operations on foreign keys

When dealing with bulk data being manipulated by outside applications, you can defer checking constraints for validity until the end of a transaction.

**Ensure Constraints Are Created Deferrable** After you have identified and selected the appropriate tables, make sure their FOREIGN, UNIQUE and PRIMARY key constraints are created deferrable. You can do so by issuing a statement similar to the following:

```
CREATE TABLE dept (
     deptno NUMBER PRIMARY KEY,
     dname VARCHAR2 (30)
     );
CREATE TABLE emp (
     empno NUMBER,
     ename VARCHAR2 (30),
     deptno NUMBER REFERENCES (dept),
     CONSTRAINT epk PRIMARY KEY (empno) DEFERRABLE,
     CONSTRAINT efk FOREIGN KEY (deptno)
     REFERENCES (dept.deptno) DEFERRABLE);
INSERT INTO dept VALUES (10, 'Accounting');
INSERT INTO dept VALUES (20, 'SALES');
INSERT INTO emp VALUES (1, 'Corleone', 10);
INSERT INTO emp VALUES (2, 'Costanza', 20);
COMMIT;

SET CONSTRAINT efk DEFERRED;
UPDATE dept SET deptno = deptno + 10
     WHERE deptno = 20;

SELECT * from emp ORDER BY deptno;
EMPNO   ENAME          DEPTNO
-----   -------------- -------
   1    Corleone       10
   2    Costanza       20
UPDATE emp SET deptno = deptno + 10
     WHERE deptno = 20;
SELECT * FROM emp ORDER BY deptno;

EMPNO   ENAME          DEPTNO
-----   -------------- -------
   1    Corleone       10
   2    Costanza       30
COMMIT;
```

**Set All Constraints Deferred** Within the application that manipulates the data, you must set all constraints deferred before you begin processing any data. Use the following DML statement to set all deferrable constraints deferred:

```
SET CONSTRAINTS ALL DEFERRED;
```

> **Note:** The `SET CONSTRAINTS` statement applies only to the current transaction. The defaults specified when you create a constraint remain as long as the constraint exists. The `ALTER SESSION SET CONSTRAINTS` statement applies for the current session only.

**Check the Commit (Optional)** You can check for constraint violations before committing by issuing the `SET CONSTRAINTS ALL IMMEDIATE` statement just before issuing the `COMMIT`. If there are any problems with a constraint, this statement will fail and the constraint causing the error will be identified. If you commit while constraints are violated, the transaction will be rolled back and you will receive an error message.

# Managing Constraints That Have Associated Indexes

When you create a `UNIQUE` or `PRIMARY` key, Oracle checks to see if an existing index can be used to enforce uniqueness for the constraint. If there is no such index, Oracle creates one.

## Minimizing Space and Time Overhead for Indexes Associated with Constraints

When Oracle uses a unique index to enforce a constraint, and constraints associated with the unique index are dropped or disabled, the index is dropped. To preserve the statistics associated with the index, or if it would take a long time to re-create it, you can specify the `KEEP INDEX` clause on the `DROP` command for the constraint.

While enabled foreign keys reference a `PRIMARY` or `UNIQUE` key, you cannot disable or drop the `PRIMARY` or `UNIQUE` key constraint or the index.

> **Note:** Deferrable `UNIQUE` and `PRIMARY` keys all must use non-unique indexes.

To reuse existing indexes when creating unique and primary key constraints, you can include USING INDEX in the constraint clause. Fpr example:

```
CREATE TABLE b
(
    b1 INTEGER,
    b2 INTEGER,
    CONSTRAINT unique1 (b1, b2) USING INDEX (CREATE UNIQUE INDEX b_index on
b(b1, b2),
    CONSTRAINT unique2 (b1, b2) USING INDEX b_index
);
```

## Guidelines for Indexing Foreign Keys

You should almost always index foreign keys. The only exception is when the matching unique or primary key is never updated or deleted.

> **See Also:** *Oracle9i Database Concepts* for information on locking mechanisms involving indexes and keys.

## About Referential Integrity in a Distributed Database

The declaration of a referential integrity constraint cannot specify a foreign key that references a primary or unique key of a remote table.

However, you can maintain parent/child table relationships across nodes using triggers.

> **See Also:** For more information about triggers that enforce referential integrity, refer to Chapter 15, "Using Triggers".

**Note:** If you decide to define referential integrity across the nodes of a distributed database using triggers, be aware that network failures can make both the parent table and the child table inaccessible. For example, assume that the child table is in the SALES database, and the parent table is in the HQ database.

If the network connection between the two databases fails, then some DML statements against the child table (those that insert rows or update a foreign key value) cannot proceed, because the referential integrity triggers must have access to the parent table in the HQ database.

# When to Use CHECK Integrity Constraints

Use CHECK constraints when you need to enforce integrity rules based on logical expressions, such as comparisons. Never use CHECK constraints when any of the other types of integrity constraints can provide the necessary checking.

**See Also:** "Choosing Between CHECK and NOT NULL Integrity Constraints" on page 4-18

Examples of CHECK constraints include the following:

- A CHECK constraint on employee salaries so that no salary value is greater than 10000.
- A CHECK constraint on department locations so that only the locations "BOSTON", "NEW YORK", and "DALLAS" are allowed.
- A CHECK constraint on the salary and commissions columns to prevent the commission from being larger than the salary.

## Restrictions on CHECK Constraints

A CHECK integrity constraint requires that a condition be true or unknown for every row of the table. If a statement causes the condition to evaluate to false, then the statement is rolled back. The condition of a CHECK constraint has the following limitations:

- The condition must be a boolean expression that can be evaluated using the values in the row being inserted or updated.
- The condition cannot contain subqueries or sequences.

- The condition cannot include the SYSDATE, UID, USER, or USERENV SQL functions.
- The condition cannot contain the pseudocolumns LEVEL, PRIOR, or ROWNUM.

  **See Also:** *Oracle9i SQL Reference* for an explanation of these pseudocolumns.

- The condition cannot contain a user-defined SQL function.

## Designing CHECK Constraints

When using CHECK constraints, remember that a CHECK constraint is violated only if the condition evaluates to false; true and unknown values (such as comparisons with nulls) do not violate a check condition. Make sure that any CHECK constraint that you define is specific enough to enforce the rule.

For example, consider the following CHECK constraint:

```
CHECK (Sal > 0 OR Comm >= 0)
```

At first glance, this rule may be interpreted as "do not allow a row in the employee table unless the employee's salary is greater than zero or the employee's commission is greater than or equal to zero." But if a row is inserted with a null salary, that row does not violate the CHECK constraint regardless of whether the commission value is valid, because the entire check condition is evaluated as unknown. In this case, you can prevent such violations by placing NOT NULL integrity constraints on both the SAL and COMM columns.

**Note:** If you are not sure when unknown values result in NULL conditions, review the truth tables for the logical operators AND and OR in *Oracle9i SQL Reference*

## Rules for Multiple CHECK Constraints

A single column can have multiple CHECK constraints that reference the column in its definition. There is no limit to the number of CHECK constraints that can be defined that reference a column.

The order in which the constraints are evaluated is not defined, so be careful not to rely on the order or to define multiple constraints that conflict with each other.

### Choosing Between CHECK and NOT NULL Integrity Constraints

According to the ANSI/ISO standard, a NOT NULL integrity constraint is an example of a CHECK integrity constraint, where the condition is the following:

```
CHECK (Column_name IS NOT NULL)
```

Therefore, NOT NULL integrity constraints for a single column can, in practice, be written in two forms: using the NOT NULL constraint or a CHECK constraint. For ease of use, you should always choose to define NOT NULL integrity constraints, instead of CHECK constraints with the IS NOT NULL condition.

In the case where a composite key can allow only all nulls or all values, you must use a CHECK integrity constraint. For example, the following expression of a CHECK integrity constraint allows a key value in the composite key made up of columns C1 and C2 to contain either all nulls or all values:

```
CHECK ((C1 IS NULL AND C2 IS NULL) OR
    (C1 IS NOT NULL AND C2 IS NOT NULL))
```

## Examples of Defining Integrity Constraints

Here are some examples showing how to create simple constraints during the prototype phase of your database design.

Notice how all constraints are given a name. Naming the constraints prevents the database from creating multiple copies of the same constraint, with different system-generated names, if the DDL is run multiple times.

> **See Also:** *Oracle9i Database Administrator's Guide* for information on creating and maintaining constraints for a large production database.

### Defining Integrity Constraints with the CREATE TABLE Command: Example

The following examples of CREATE TABLE statements show the definition of several integrity constraints:

```
CREATE TABLE Dept_tab (
    Deptno  NUMBER(3) CONSTRAINT Dept_pkey PRIMARY KEY,
    Dname   VARCHAR2(15),
    Loc     VARCHAR2(15),
                CONSTRAINT Dname_ukey UNIQUE (Dname, Loc),
                CONSTRAINT Loc_check1
```

```
                    CHECK (loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')));

CREATE TABLE Emp_tab (
    Empno    NUMBER(5) CONSTRAINT Emp_pkey PRIMARY KEY,
    Ename    VARCHAR2(15) NOT NULL,
    Job      VARCHAR2(10),
    Mgr      NUMBER(5) CONSTRAINT Mgr_fkey
                  REFERENCES Emp_tab,
    Hiredate DATE,
    Sal      NUMBER(7,2),
    Comm     NUMBER(5,2),
    Deptno   NUMBER(3) NOT NULL
             CONSTRAINT dept_fkey REFERENCES Dept_tab ON DELETE CASCADE);
```

## Defining Constraints with the ALTER TABLE Command: Example

You can also define integrity constraints using the constraint clause of the ALTER TABLE command. For example, the following examples of ALTER TABLE statements show the definition of several integrity constraints:

```
CREATE UNIQUE INDEX I_dept ON Dept_tab(deptno);
ALTER TABLE Dept_tab
    ADD CONSTRAINT Dept_pkey PRIMARY KEY (deptno);

ALTER TABLE Emp_tab
    ADD CONSTRAINT Dept_fkey FOREIGN KEY (Deptno) REFERENCES Dept_tab;
ALTER TABLE Emp_tab MODIFY (Ename VARCHAR2(15) NOT NULL);
```

You cannot create a validated constraint on a table if the table already contains any rows that would violate the constraint.

## Privileges Required to Create Constraints

The creator of a constraint must have the ability to create tables (the CREATE TABLE or CREATE ANY TABLE system privilege), or the ability to alter the table (the ALTER object privilege for the table or the ALTER ANY TABLE system privilege) with the constraint. Additionally, UNIQUE and PRIMARY KEY integrity constraints require that the owner of the table have either a quota for the tablespace that contains the associated index or the UNLIMITED TABLESPACE system privilege. FOREIGN KEY integrity constraints also require some additional privileges.

> **See Also:** "Privileges Required to Create FOREIGN KEY Integrity Constraints" on page 4-27

## Naming Integrity Constraints

Assign names to NOT NULL, UNIQUE KEY, PRIMARY KEY, FOREIGN KEY, and CHECK constraints using the CONSTRAINT option of the constraint clause. This name must be unique with respect to other constraints that you own. If you do not specify a constraint name, one is assigned by Oracle.

Picking your own name makes error messages for constraint violations more understandable, and prevents the creation of multiple constraints if the SQL statements are run more than once.

See the previous examples of the CREATE TABLE and ALTER TABLE statements for examples of the CONSTRAINT option of the constraint clause. Note that the name of each constraint is included with other information about the constraint in the data dictionary.

> **See Also:** "Viewing Definitions of Integrity Constraints" on page 4-29 for examples of data dictionary views.

# Enabling and Disabling Integrity Constraints

This section explains the mechanisms and procedures for manually enabling and disabling integrity constraints.

**enabled constraint**. When a constraint is enabled, the corresponding rule is enforced on the data values in the associated columns. The definition of the constraint is stored in the data dictionary.

**disabled constraint**. When a constraint is disabled, the corresponding rule is not enforced. The definition of the constraint is still stored in the data dictionary.

An integrity constraint represents an assertion about the data in a database. This assertion is always true when the constraint is enabled. The assertion may or may not be true when the constraint is disabled, because data that violates the integrity constraint can be in the database.

### Why Disable Constraints?

During day-to-day operations, constraints should always be enabled. In certain situations, temporarily disabling the integrity constraints of a table makes sense for performance reasons. For example:

- When loading large amounts of data into a table using SQL*Loader
- When performing batch operations that make massive changes to a table (such as changing everyone's employee number by adding 1000 to the existing number)
- When importing or exporting one table at a time

Turning off integrity constraints temporarily speeds up these operations.

### About Exceptions to Integrity Constraints

If a row of a table disobeys an integrity constraint, then this row is in violation of the constraint and is called an **exception** to the constraint. If any exceptions exist, then the constraint *cannot* be enabled**.** The rows that violate the constraint must be either updated or deleted before the constraint can be enabled.

You can identify exceptions for a specific integrity constraint as you try to enable the constraint.

> **See Also:** This procedure is discussed in the section "Fixing Constraint Exceptions" on page 4-24.

### Enabling Constraints

When you define an integrity constraint in a CREATE TABLE or ALTER TABLE statement, Oracle automatically enables the constraint by default. For code clarity, you can explicitly enable the constraint by including the ENABLE clause in its definition.

Use this technique when creating tables that start off empty, and are populated a row at a time by individual transactions. In such cases, you want to ensure that data are consistent at all times, and the performance overhead of each DML operation is small.

The following CREATE TABLE and ALTER TABLE statements both define and enable integrity constraints:

```
CREATE TABLE Emp_tab (
    Empno NUMBER(5) PRIMARY KEY);
 ALTER TABLE Emp_tab
    ADD PRIMARY KEY (Empno);
```

An ALTER TABLE statement that tries to enable an integrity constraint will fail if any rows of the table violate the integrity constraint. The statement is rolled back and the constraint definition is not stored and not enabled.

> **See Also:** "Fixing Constraint Exceptions" on page 4-24 for more information about rows that violate integrity constraints.

### Creating Disabled Constraints

The following CREATE TABLE and ALTER TABLE statements both define and disable integrity constraints:

```
CREATE TABLE Emp_tab (
    Empno NUMBER(5) PRIMARY KEY DISABLE);

ALTER TABLE Emp_tab
    ADD PRIMARY KEY (Empno) DISABLE;
```

Use this technique when creating tables that will be loaded with large amounts of data before anybody else accesses them, particularly if you need to cleanse data after loading it, or need to fill in empty columns with sequence numbers or parent/child relationships.

An ALTER TABLE statement that defines and disables an integrity constraints never fails, because its rule is not enforced.

## Enabling and Disabling Existing Integrity Constraints

Use the ALTER TABLE command to:

- Enable a disabled constraint, using the ENABLE clause.
- Disable an enabled constraint, using the DISABLE clause.

### Enabling Existing Constraints

Once you have finished cleansing data and filling in empty columns, you can enable constraints that were disabled during data loading.

The following statements are examples of statements that enable disabled integrity constraints:

```
ALTER TABLE Dept_tab
    ENABLE CONSTRAINT Dname_ukey;

ALTER TABLE Dept_tab
    ENABLE PRIMARY KEY
    ENABLE UNIQUE (Dname)
    ENABLE UNIQUE (Loc);
```

An ALTER TABLE statement that attempts to enable an integrity constraint fails when the rows of the table violate the integrity constraint. The statement is rolled back and the constraint is not enabled.

> **See Also:** "Fixing Constraint Exceptions" on page 4-24 for more information about rows that violate integrity constraints.

### Disabling Existing Constraints

If you need to perform a large load or update when the table already contains data, you can temporarily disable constraints to improve performance of the bulk operation.

The following statements are examples of statements that disable enabled integrity constraints:

```
ALTER TABLE Dept_tab
    DISABLE CONSTRAINT Dname_ukey;

ALTER TABLE Dept_tab
    DISABLE PRIMARY KEY
    DISABLE UNIQUE (Dname)
    DISABLE UNIQUE (Loc);
```

### Tip: Using the Data Dictionary to Find Constraints

The preceding examples require that you know constraint names and which columns they affect. To find this information, you can query one of the data dictionary views defined for constraints, USER_CONSTRAINTS or USER_CONS_COLUMNS. For more information about these views, see "Viewing Definitions of Integrity Constraints" on page 4-29 and *Oracle9i Database Reference*.

## Guidelines for Enabling and Disabling Key Integrity Constraints

When enabling or disabling UNIQUE, PRIMARY KEY, and FOREIGN KEY integrity constraints, you should be aware of several important issues and prerequisites. UNIQUE key and PRIMARY KEY constraints are usually managed by the database administrator.

> **See Also:** "Managing FOREIGN KEY Integrity Constraints" on page 4-27 and the *Oracle9i Database Administrator's Guide*

## Fixing Constraint Exceptions

When you try to create or enable a constraint, and the statement fails because integrity constraint exceptions exist, the statement is rolled back. You cannot enable the constraint until all exceptions are either updated or deleted. To determine which rows violate the integrity constraint, include the EXCEPTIONS option in the ENABLE clause of a CREATE TABLE or ALTER TABLE statement.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about fixing constraint exceptions.

# Altering Integrity Constraints

Starting with Oracle8*i*, you can alter the state of an existing constraint with the MODIFY CONSTRAINT clause.

> **See Also:** For information on the parameters you can modify, see the ALTER TABLE section in *Oracle9i SQL Reference.*

### MODIFY CONSTRAINT Example #1

The following commands show several alternatives for whether the CHECK constraint is enforced, and when the constraint checking is done:

```
CREATE TABLE X1_tab (a1 NUMBER CONSTRAINT y CHECK (a1>3) DEFERRABLE DISABLE);

ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt ENABLE;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt RELY;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt INITIALLY DEFERRED;
ALTER TABLE X1_tab MODIFY CONSTRAINT Y_cnstrt ENABLE NOVALIDATE;
```

### MODIFY CONSTRAINT Example #2

The following commands show several alternatives for whether the NOT NULL constraint is enforced, and when the checking is done:

```
CREATE TABLE X1_tab (A1 NUMBER CONSTRAINT Y_cnstrt
NOT NULL DEFERRABLE INITIALLY DEFERRED NORELY DISABLE);

ALTER TABLE X1_tab ADD CONSTRAINT One_cnstrt UNIQUE(A1)
DEFERRABLE INITIALLY IMMEDIATE RELY USING INDEX PCTFREE = 30
```

```
ENABLE VALIDATE;

ALTER TABLE X1_tab MODIFY UNIQUE(A1)
INITIALLY DEFERRED NORELY USING INDEX PCTFREE = 40
ENABLE NOVALIDATE;

ALTER TABLE X1_tab MODIFY CONSTRAINT One_cnstrt
INITIALLY IMMEDIATE RELY;
```

### Modify Constraint Example #3

The following commands show several alternatives for whether the primary key constraint is enforced, and when the checking is done:

```
CREATE TABLE T1_tab (A1 INT, B1 INT);
ALTER TABLE T1_tab add CONSTRAINT P1_cnstrt PRIMARY KEY(a1) DISABLE;
ALTER TABLE T1_tab MODIFY PRIMARY KEY INITIALLY IMMEDIATE
USING INDEX PCTFREE = 30 ENABLE NOVALIDATE;
ALTER TABLE T1_tab MODIFY PRIMARY KEY
USING INDEX PCTFREE = 35 ENABLE;
ALTER TABLE T1_tab MODIFY PRIMARY KEY ENABLE NOVALIDATE;
```

## Renaming Integrity Constraints

Because constraint names must be unique, even across multiple schemas, you can encounter problems when you want to clone a table and all its constraints, but the constraint name for the new table conflicts with the one for the original table. Or, you might create a constraint with a default system-generated name, and later realize that it's better to give the constraint a name that is easy to remember, so that you can easily enable and disable it.

One of the properties you can alter for a constraint is its name. The following SQL*Plus script shows you you can find the system-generated name for a constraint and change it to a name of your choosing:

```
prompt Enter table name to find its primary key:
accept table_name
select constraint_name from user_constraints
  where table_name = upper('&table_name.')
  and constraint_type = 'P';

prompt Enter new name for its primary key:
```

```
accept new_constraint

set serveroutput on

declare
-- USER_CONSTRAINTS.CONSTRAINT_NAME is declared as VARCHAR2(30).
-- Using %TYPE here protects us if the length changes in a future release.
  constraint_name user_constraints.constraint_name%type;
begin
  select constraint_name into constraint_name from user_constraints
    where table_name = upper('&table_name.')
    and constraint_type = 'P';

  dbms_output.put_line('The primary key for ' || upper('&table_name.') || ' is:
' || constraint_name);

  execute immediate
    'alter table &table_name. rename constraint ' || constraint_name ||
    ' to &new_constraint.';
end;
/
```

## Dropping Integrity Constraints

Drop an integrity constraint if the rule that it enforces is no longer true or if the constraint is no longer needed. Drop an integrity constraint using the ALTER TABLE command and the DROP clause. For example, the following statements drop integrity constraints:

```
ALTER TABLE Dept_tab
    DROP UNIQUE (Dname);
ALTER TABLE Dept_tab
    DROP UNIQUE (Loc);

ALTER TABLE Emp_tab
    DROP PRIMARY KEY,
    DROP CONSTRAINT Dept_fkey;

DROP TABLE Emp_tab CASCADE CONSTRAINTS;
```

When dropping UNIQUE, PRIMARY KEY, and FOREIGN KEY integrity constraints, you should be aware of several important issues and prerequisites. UNIQUE and PRIMARY KEY constraints are usually managed by the database administrator.

**See Also:** "Managing FOREIGN KEY Integrity Constraints" on page 4-27 and the *Oracle9i Database Administrator's Guide*.

# Managing FOREIGN KEY Integrity Constraints

General information about defining, enabling, disabling, and dropping all types of integrity constraints is given in the previous sections. The following section supplements this information, focusing specifically on issues regarding FOREIGN KEY integrity constraints, which enforce relationships between columns in different tables.

## Rules for FOREIGN KEY Integrity Constraints

The following topics are of interest when defining FOREIGN KEY integrity constraints.

### Datatypes and Names for Foreign Key Columns

You must use the same datatype for corresponding columns in the dependent and referenced tables. The column names do not need to match.

### Limit on Columns in Composite Foreign Keys

Because foreign keys reference primary and unique keys of the parent table, and PRIMARY KEY and UNIQUE key constraints are enforced using indexes, composite foreign keys are limited to 32 columns.

### Foreign Key References Primary Key by Default

If the column list is not included in the REFERENCES option when defining a FOREIGN KEY constraint (single column or composite), then Oracle assumes that you intend to reference the primary key of the specified table. Alternatively, you can explicitly specify the column(s) to reference in the parent table within parentheses. Oracle automatically checks to verify that this column list references a primary or unique key of the parent table. If it does not, then an informative error is returned.

### Privileges Required to Create FOREIGN KEY Integrity Constraints

To create a FOREIGN KEY constraint, the creator of the constraint must have privileged access to both the parent and the child table.

- **The Parent Table** The creator of the referential integrity constraint must own the parent table or have REFERENCES object privileges on the columns that constitute the parent key of the parent table.
- **The Child Table** The creator of the referential integrity constraint must have the ability to create tables (that is, the CREATE TABLE or CREATE ANY TABLE system privilege) or the ability to alter the child table (that is, the ALTER object privilege for the child table or the ALTER ANY TABLE system privilege).

In both cases, necessary privileges *cannot* be obtained through a role; they must be explicitly granted to the creator of the constraint.

These restrictions allow:

- The owner of the child table to explicitly decide what constraints are enforced on her or his tables and the other users that can create constraints on her or his tables
- The owner of the parent table to explicitly decide if foreign keys can depend on the primary and unique keys in her tables

### Choosing How Foreign Keys Enforce Referential Integrity

Oracle allows different types of referential integrity actions to be enforced, as specified with the definition of a FOREIGN KEY constraint:

- **Prevent Update or Delete of Parent Key** The default setting prevents the update or deletion of a parent key if there is a row in the child table that references the key. For example:

```
CREATE TABLE Emp_tab (
FOREIGN KEY (Deptno) REFERENCES Dept_tab);
```

- **Delete Child Rows When Parent Key Deleted** The ON DELETE CASCADE action allows parent key data that is referenced from the child table to be deleted, but not updated. When data in the parent key is deleted, all rows in the child table that depend on the deleted parent key values are also deleted. To specify this referential action, include the ON DELETE CASCADE option in the definition of the FOREIGN KEY constraint. For example:

```
CREATE TABLE Emp_tab (
    FOREIGN KEY (Deptno) REFERENCES Dept_tab
        ON DELETE CASCADE);
```

- **Set Foreign Keys to Null When Parent Key Deleted** The ON DELETE SET NULL action allows data that references the parent key to be deleted, but not

updated. When referenced data in the parent key is deleted, all rows in the child table that depend on those parent key values have their foreign keys set to null. To specify this referential action, include the ON DELETE SET NULL option in the definition of the FOREIGN KEY constraint. For example:

```
CREATE TABLE Emp_tab (
    FOREIGN KEY (Deptno) REFERENCES Dept_tab
        ON DELETE SET NULL);
```

### Restriction on Enabling FOREIGN KEY Integrity Constraints

FOREIGN KEY integrity constraints cannot be enabled if the referenced primary or unique key's constraint is not present or not enabled.

## Viewing Definitions of Integrity Constraints

The data dictionary contains the following views that relate to integrity constraints:

- ALL_CONSTRAINTS
- ALL_CONS_COLUMNS
- USER_CONSTRAINTS
- USER_CONS_COLUMNS
- DBA_CONSTRAINTS
- DBA_CONS_COLUMNS

You can query these views to find the names of constraints, what columns they affect, and other information to help you manage constraints.

> **See Also:** Refer to *Oracle9i Database Reference* for detailed information about each view.

### Examples of Defining Integrity Constraints

Consider the following CREATE TABLE statements that define a number of integrity constraints:

```
CREATE TABLE Dept_tab (
    Deptno   NUMBER(3) PRIMARY KEY,
    Dname    VARCHAR2(15),
    Loc      VARCHAR2(15),
    CONSTRAINT Dname_ukey UNIQUE (Dname, Loc),
    CONSTRAINT LOC_CHECK1
```

```
        CHECK (Loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')));

CREATE TABLE Emp_tab (
    Empno    NUMBER(5) PRIMARY KEY,
    Ename    VARCHAR2(15) NOT NULL,
    Job      VARCHAR2(10),
    Mgr      NUMBER(5) CONSTRAINT Mgr_fkey
         REFERENCES Emp_tab ON DELETE CASCADE,
    Hiredate DATE,
    Sal      NUMBER(7,2),
    Comm     NUMBER(5,2),
    Deptno   NUMBER(3) NOT NULL
    CONSTRAINT Dept_fkey REFERENCES Dept_tab);
```

**Example 1: Listing All of Your Accessible Constraints** The following query lists all constraints defined on all tables accessible to the user:

```
SELECT Constraint_name, Constraint_type, Table_name,
    R_constraint_name
    FROM User_constraints;
```

Considering the example statements at the beginning of this section, a list similar to the one below is returned:

```
CONSTRAINT_NAME  C TABLE_NAME  R_CONSTRAINT_NAME
---------------  - ----------- -----------------
SYS_C00275       P DEPT_TAB
DNAME_UKEY       U DEPT_TAB
LOC_CHECK1       C DEPT_TAB
SYS_C00278       C EMP_TAB
SYS_C00279       C EMP_TAB
SYS_C00280       P EMP_TAB
MGR_FKEY         R EMP_TAB     SYS_C00280
DEPT_FKEY        R EMP_TAB     SYS_C00275
```

Notice the following:

- Some constraint names are user specified (such as `DNAME_UKEY`), while others are system specified (such as `SYS_C00275`).
- Each constraint type is denoted with a different character in the `CONSTRAINT_TYPE` column. The following table summarizes the characters used for each constraint type.

| Constraint Type | Character |
|---|---|
| PRIMARY KEY | P |
| UNIQUE KEY | U |
| FOREIGN KEY | R |
| CHECK, NOT NULL | C |

**Note:** An additional constraint type is indicated by the character "V" in the CONSTRAINT_TYPE column. This constraint type corresponds to constraints created by the WITH CHECK OPTION for views. See Chapter 2, "Managing Schema Objects" for more information about views and the WITH CHECK OPTION.

**Example 2: Distinguishing NOT NULL Constraints from CHECK Constraints** In the previous example, several constraints are listed with a constraint type of "C". To distinguish which constraints are NOT NULL constraints and which are CHECK constraints in the EMP_TAB and DEPT_TAB tables, issue the following query:

```
SELECT Constraint_name, Search_condition
    FROM User_constraints
    WHERE (Table_name = 'DEPT_TAB' OR Table_name = 'EMP_TAB') AND
        Constraint_type = 'C';
```

Considering the example CREATE TABLE statements at the beginning of this section, a list similar to the one below is returned:

```
CONSTRAINT_NAME  SEARCH_CONDITION
---------------  ---------------------------------------
LOC_CHECK1       loc IN ('NEW YORK', 'BOSTON', 'CHICAGO')
SYS_C00278       ENAME IS NOT NULL
SYS_C00279       DEPTNO IS NOT NULL
```

Notice the following:

- NOT NULL constraints are clearly identified in the SEARCH_CONDITION column.
- The conditions for user-defined CHECK constraints are explicitly listed in the SEARCH_CONDITION column.

**Example 3: Listing Column Names that Constitute an Integrity Constraint** The following query lists all columns that constitute the constraints defined on all tables accessible to you, the user:

```
SELECT Constraint_name, Table_name, Column_name
    FROM User_cons_columns;
```

Considering the example statements at the beginning of this section, a list similar to the one below is returned:

```
CONSTRAINT_NAME  TABLE_NAME  COLUMN_NAME
---------------  ----------- ---------------
DEPT_FKEY        EMP_TAB     DEPTNO
DNAME_UKEY       DEPT_TAB    DNAME
DNAME_UKEY       DEPT_TAB    LOC
LOC_CHECK1       DEPT_TAB    LOC
MGR_FKEY         EMP_TAB     MGR
SYS_C00275       DEPT_TAB    DEPTNO
SYS_C00278       EMP_TAB     ENAME
SYS_C00279       EMP_TAB     DEPTNO
SYS_C00280       EMP_TAB     EMPNO
```

# 5

# Selecting an Index Strategy

This chapter discusses the considerations for using the different types of indexes in an application. The topics include:

- Guidelines for Application-Specific Indexes
- Creating Indexes: Basic Examples
- When to Use Function-Based Indexes

**See Also:**

- *Oracle9i Database Performance Guide and Reference* for detailed information about using indexes.
- *Oracle9i Database Administrator's Guide* for information about creating and managing indexes.
- *Oracle9i SQL Reference* for the syntax of commands to work with indexes.