CHAPTER 10

Dates and Timestamps

Most applications require the storage and manipulation of dates and times. Dates are quite complicated: not only are they highly formatted data, but there are myriad rules for determining valid values and valid calculations (leap days and years, daylight saving time changes, national and company holidays, date ranges, etc.). Fortunately, the Oracle database and PL/SQL provide a set of true datetime datatypes that store both date and time information using a standard internal format.

For any datetime value, the database stores some or all of the following information:

- Year
- Month
- Day
- Hour
- Minute
- Second
- Time zone region
- Time zone hour offset from UTC
- Time zone minute offset from UTC

Support for true datetime datatypes is only half the battle. You also need a language that can manipulate those values in a natural and intelligent manner—as actual dates and times. To that end, Oracle provides you with support for SQL standard interval arithmetic, datetime literals, and a comprehensive suite of functions with which to manipulate date and time information.

Datetime Datatypes

For a long time, the only datetime datatype available was DATE. Oracle9*i* Database shook things up by introducing three new TIMESTAMP and two new INTERVAL datatypes, offering significant new functionality while also bringing Oracle into closer compliance with the ISO SQL standard. I'll talk more about the INTERVAL datatypes later in this chapter. The four datetime datatypes are:

DATE
: Stores a date and time, resolved to the second. Does not include time zone.

TIMESTAMP
: Stores a date and time without respect to time zone. Except for being able to resolve time to the billionth of a second (nine decimal places of precision), TIMESTAMP is the equivalent of DATE.

TIMESTAMP WITH TIME ZONE
: Stores the time zone along with the date and time value, allowing up to nine decimal places of precision.

TIMESTAMP WITH LOCAL TIME ZONE
: Stores a date and time with up to nine decimal places of precision. This datatype is sensitive to time zone differences. Values of this type are automatically converted between the database time zone and the local (session) time zone. When values are stored in the database, they are converted to the database time zone, but the local (session) time zone is not stored. When a value is retrieved from the database, that value is converted from the database time zone to the local (session) time zone.

The nuances of these types, especially the TIMESTAMP WITH LOCAL TIME ZONE type, can be a bit difficult to understand at first. To help illustrate, let's look at the use of TIMESTAMP WITH LOCAL TIME ZONE in a calendaring application for users across multiple time zones. My database time zone is Coordinated Universal Time (UTC). (See the sidebar "Coordinated Universal Time" on page 280 for a description of UTC.) User Jonathan in Michigan (Eastern Daylight Time: UTC –4:00) has scheduled a conference call for 4:00–5:00 p.m. his time on Thursday. Donna in Denver (Mountain Daylight Time: UTC –6:00) needs to know this meeting is at 2:00–3:00 p.m. her time on Thursday. Selva in India (Indian Standard Time: UTC +5:30) needs to know this meeting is at 1:30–2:30 a.m. his time on Friday morning. Figure 10-1 shows how the meeting start time varies as it moves from a user in one time zone through the database to another user in a different time zone.

Figure 10-1 shows user Jonathan in Eastern Daylight Time, which is four hours behind UTC, or UTC –4:00. Jonathan enters the meeting start time as 16:00 using 24-hour notation. This value gets converted to the database time zone (UTC) when the row is inserted. 20:00 is the value stored in the database. Donna is in Denver, where daylight saving time is also observed as Mountain Daylight Time and is six hours behind Coordinated Universal Time (UTC –6:00). When Donna selects the start time, the value is converted to her session time zone and is displayed as 14:00. Selva is in India, which does not observe daylight saving time—India Standard Time is five and a half hours ahead of UTC (UTC +5:30). When Selva selects the meeting start time, the value is converted to his session time zone and is displayed as 1:30 a.m. Friday.

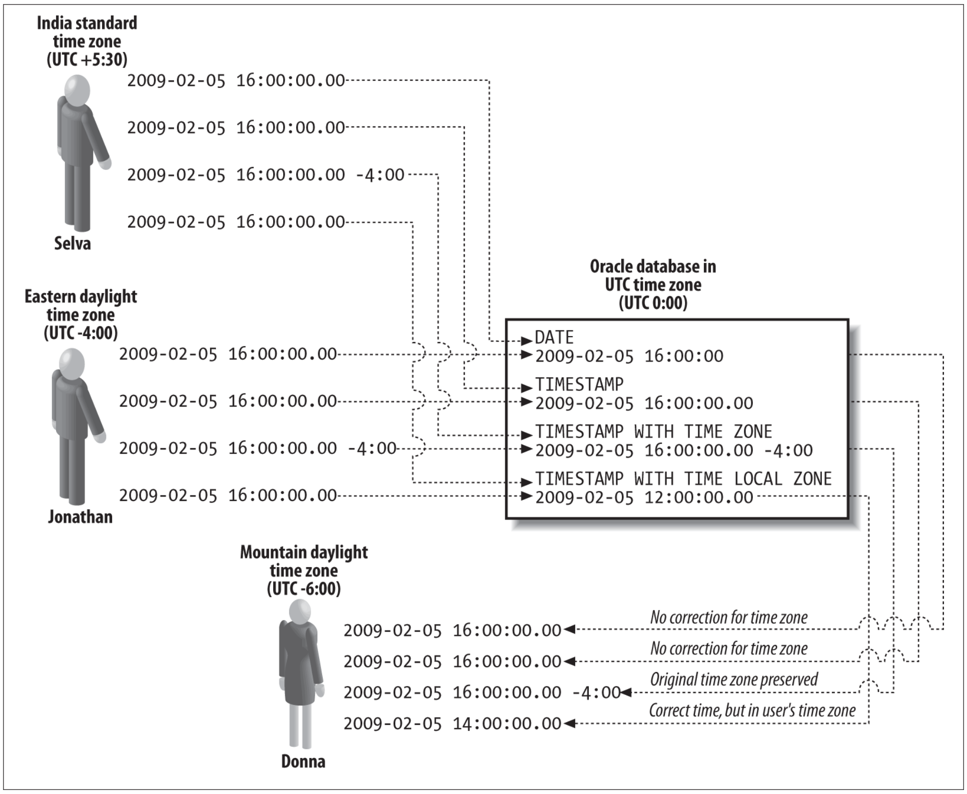

Figure 10-1. Effect of different datetime datatypes

Delegating the time zone management to the database via the TIMESTAMP WITH LOCAL TIME ZONE datatype means you don't have to burden your application with the complex rules surrounding time zones and daylight saving time (which sometimes changes—as it did in the United States in 2007), nor do you have to burden your users with figuring out the time zone conversion. The correct time for the meeting is presented to each user simply and elegantly.

Sometimes you want the database to automatically change the display of the time, and sometimes you don't. When you don't want the display of the timestamp to vary based on session settings, use the TIMESTAMP or TIMESTAMP WITH TIME ZONE datatypes.

Coordinated Universal Time

Coordinated Universal Time, abbreviated UTC, is measured using highly accurate and precise atomic clocks, and it forms the basis of our worldwide system of civil time. Time zones, for example, are all defined with respect to how much they deviate from UTC. UTC is atomic time, and it is periodically adjusted through the mechanism of leap seconds to keep it in sync with time as determined by the rotation of the Earth.

You may be familiar with Greenwich Mean Time (GMT) or Zulu Time. For most practical purposes, these references are equivalent to UTC.

Why the acronym UTC and not CUT? The standards body couldn't agree on whether to use the English acronym CUT or the French acronym TUC, so it compromised on UTC, which matches neither language.

For more information on UTC, see the National Institute of Standards and Technology document on UTC and the FAQ page.

Declaring Datetime Variables

Use the following syntax to declare a datetime variable:

```
var_name [CONSTANT] datetime_type [{:= | DEFAULT} initial_value]
```

Replace *datetime_type* with any one of the following:

```
DATE
TIMESTAMP [(precision)]
TIMESTAMP [(precision)] WITH TIME ZONE
TIMESTAMP [(precision)] WITH LOCAL TIME ZONE
```

The *precision* in these declarations refers to the number of decimal digits allocated for recording values to the fraction of a second. The default precision is 6, which means that you can track time down to 0.000001 seconds. The allowable range for precision is 0 through 9, giving you the ability to store very precise time-of-day values.

Functions such as SYSTIMESTAMP that return timestamp values always return only six digits of subsecond precision.

Following are some example declarations:

```
DECLARE
   hire_date TIMESTAMP (0) WITH TIME ZONE;
   todays_date CONSTANT DATE := SYSDATE;
   pay_date TIMESTAMP DEFAULT TO_TIMESTAMP('20050204','YYYYMMDD');
```

```
BEGIN
   NULL;
END;
/
```

To specify a default *initial_value*, you can use a conversion function such as TO_TIMESTAMP, or you can use a date or timestamp literal. Both are described in "Datetime Conversions" on page 289.

A TIMESTAMP(0) variable behaves like a DATE variable.

Choosing a Datetime Datatype

With such an abundance of riches, I won't blame you one bit if you ask for some guidance as to which datetime datatype to use when. To a large extent, the datatype you choose depends on the level of detail that you want to store:

- Use one of the TIMESTAMP types if you need to track time down to a fraction of a second.
- Use TIMESTAMP WITH LOCAL TIME ZONE if you want the database to automatically convert a time between the database and session time zones.
- Use TIMESTAMP WITH TIME ZONE if you need to keep track of the session time zone in which the data was entered.
- You can use TIMESTAMP in place of DATE. A TIMESTAMP that does not contain subsecond precision takes up 7 bytes of storage, just like a DATE datatype does. When your TIMESTAMP does contain subsecond data, it takes up 11 bytes of storage.

Other considerations might also apply:

- Use DATE when it's necessary to maintain compatibility with an existing application written before any of the TIMESTAMP datatypes were introduced.
- In general, you should use datatypes in your PL/SQL code that correspond to, or are at least compatible with, the underlying database tables. Think twice, for example, before reading a TIMESTAMP value from a table into a DATE variable, because you might lose information (in this case, the fractional seconds and perhaps the time zone).
- If you're using a version older than Oracle9*i* Database, then you have no choice but to use DATE.

- When adding or subtracting years and months, you get different behavior from using ADD_MONTHS, which operates on values of type DATE, than from using interval arithmetic on the timestamp types. See "When to Use INTERVALs" on page 287 for more on this critical yet subtle issue.

Exercise caution when using the DATE and TIMESTAMP datatypes together. Date arithmetic differs significantly between the two. Be careful when applying Oracle's traditional built-in date functions (such as ADD_MONTHS or MONTHS_BETWEEN) to values from any of the timestamp types. See "Datetime Arithmetic" on page 311 for more on this topic.

Getting the Current Date and Time

In any language, it's important to know how to get the current date and time. How to do that is often one of the first questions to come up, especially in applications that involve dates in any way, as most applications do.

Up through Oracle8*i* Database, you had one choice for getting the date and time in PL/SQL: you used the SYSDATE function, and that was it. Beginning with Oracle9*i* Database, you have all the functions in Table 10-1 at your disposal, and you need to understand how they work and what your choices are.

Table 10-1. Comparison of functions that return current date and time

Function	Time zone	Datatype returned
CURRENT_DATE	Session	DATE
CURRENT_TIMESTAMP	Session	TIMESTAMP WITH TIME ZONE
LOCALTIMESTAMP	Session	TIMESTAMP
SYSDATE	Database server	DATE
SYSTIMESTAMP	Database server	TIMESTAMP WITH TIME ZONE

So which function should you use in a given situation? The answer depends on several factors, which you should probably consider in the following order:

1. Whether you are using a release prior to Oracle8*i* Database or need to maintain compatibility with such a release. In either case, your choice is simple: use SYSDATE.
2. Whether you are interested in the time on the database server or for your session. If for your session, then use a function that returns the session time zone. If for the database server, then use a function that returns the database time zone.

3. Whether you need the time zone to be returned as part of the current date and time. If so, then call either SYSTIMESTAMP or CURRENT_TIMESTAMP.

If you decide to use a function that returns the time in the session time zone, be certain that you have correctly specified your session time zone. The functions SESSIONTIMEZONE and DBTIMEZONE will report your session and database time zones, respectively. To report on the time in the database time zone, you must alter your session time zone to DBTIMEZONE and then use one of the session time zone functions. The following example illustrates some of these functions:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE('Session Timezone='||SESSIONTIMEZONE);
   DBMS_OUTPUT.PUT_LINE('Session Timestamp='||CURRENT_TIMESTAMP);
   DBMS_OUTPUT.PUT_LINE('DB Server Timestamp='||SYSTIMESTAMP);
   DBMS_OUTPUT.PUT_LINE('DB Timezone='||DBTIMEZONE);
   EXECUTE IMMEDIATE 'ALTER SESSION SET TIME_ZONE=DBTIMEZONE';
   DBMS_OUTPUT.PUT_LINE('DB Timestamp='||CURRENT_TIMESTAMP);
   -- Revert session time zone to local setting
   EXECUTE IMMEDIATE 'ALTER SESSION SET TIME_ZONE=LOCAL';
END;
```

The output is:

```
Session Timezone=-04:00
Session Timestamp=23-JUN-08 12.48.44.656003000 PM −04:00
DB Server Timestamp=23-JUN-08 11.48.44.656106000 AM −05:00
DB Timezone=+00:00
DB Timestamp=23-JUN-08 04.48.44.656396000 PM +00:00
```

In this example, the session starts in US Eastern Daylight Time (−4:00) while the server is on U.S. Central Daylight Time (−5:00). Although the database server is in Central Daylight Time, the database time zone is GMT (+00:00). To get the time in the database time zone, I first set the session time zone to match the database time zone, then call the session time zone function CURRENT_TIMESTAMP. Finally, I revert my session time zone to the regular local setting that I started with.

What if there's no function to return a value in the datatype that you need? For example, what if you need the server time in a TIMESTAMP variable? You can let the database implicitly convert the types for you, but even better would be to use an explicit conversion with CAST. For example:

```
DECLARE
   ts1 TIMESTAMP;
   ts2 TIMESTAMP;
BEGIN
   ts1 := CAST(SYSTIMESTAMP AS TIMESTAMP);
   ts2 := SYSDATE;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts1,'DD-MON-YYYY HH:MI:SS AM'));
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts2,'DD-MON-YYYY HH:MI:SS AM'));
END;
```

The output is:

```
24-FEB-2002 06:46:39 PM
24-FEB-2002 06:46:39 PM
```

The call to SYSTIMESTAMP uses CAST to make the conversion from TIMESTAMP WITH TIME ZONE to TIMESTAMP explicit. The call to SYSDATE allows the conversion from DATE to TIMESTAMP to happen implicitly.

Be aware of hardware and operating system limitations if you are using these timestamp functions for subsecond timing purposes. The CURRENT_TIMESTAMP, LOCALTIMESTAMP, and SYSTIMESTAMP functions return values in either the TIMESTAMP WITH TIME ZONE or TIMESTAMP datatypes. These datatypes allow you to resolve time down to the billionth of a second.

That's all well and good, but think about where that time comes from. The database gets the time from the operating system via a call to GetTimeOfDay (Unix/Linux), GetSystemTime (Microsoft Windows), or other similar calls on other operating systems. The operating system, in turn, depends at some level on the hardware. If your operating system or underlying hardware tracks time only to the hundredth of a second, the database won't be able to return results any more granular than that. For example, when using Linux on an Intel x86 processor you can resolve time only to the millionth of a second (six digits), whereas you can see resolution only to the thousandth of a second when the database runs on Microsoft Windows XP or Vista on the same hardware. In addition, while the operating system may report a timestamp with six digits of decimal precision, this number may not represent an accuracy of 1 microsecond.

Interval Datatypes

The datetime datatypes let you record specific points in time. Interval datatypes, first introduced in Oracle9*i* Database, are all about recording and computing *quantities* of time. To better understand what the interval datatypes represent, step back a bit and think about the different kinds of datetime data you deal with on a daily basis:

Instants
An *instant* is a point in time with respect to a given granularity. When you plan to wake up at a given hour in the morning, that hour represents an instant. The granularity, then, would be to the hour, or possibly to the minute. DATE and all the TIMESTAMP datatypes allow you to represent instants of time.

Intervals
An *interval* refers not to a specific point in time, but to a specific *amount*, or quantity, of time. You use intervals all the time in your daily life. You work for eight hours a

day (you hope), you take an hour for lunch (in your dreams!), and so forth. Oracle Database's two INTERVAL types allow you to represent time intervals.

Periods
A *period* (our definition) refers to an interval of time that begins or ends at a specific instant. For example: "I woke up at 8:00 a.m. today and worked for eight hours." Here, the eight-hour interval beginning at 8:00 a.m. today would be considered a period. The Oracle database has no datatype to directly support periods, nor does the SQL standard define one.

The database supports two interval datatypes. Both were introduced in Oracle9*i* Database, and both conform to the ISO SQL standard:

INTERVAL YEAR TO MONTH
Allows you to define an interval of time in terms of years and months.

INTERVAL DAY TO SECOND
Allows you to define an interval of time in terms of days, hours, minutes, and seconds (including fractional seconds).

Why Two INTERVAL Datatypes?

I was initially puzzled about the need for two INTERVAL datatypes. I noticed that between the two datatypes, all portions of a TIMESTAMP value were accounted for, but the decision to treat year and month separately from days, hours, minutes, and seconds seemed at first rather arbitrary. Why not simply have one INTERVAL type that covers all possibilities? It turns out that we can blame this state of affairs on the long-dead Roman emperor Julius Caesar, who designed our calendar and determined most of our month lengths.

The reason for having two INTERVAL types with a dividing line at the month level is that months are the only datetime component for which the length of time in question varies. Think about having an interval of one month and 30 days. How long is that, really? Is it less than two months? The same as two months? More than two months? If the one month is January, then 30 days gets you past February and into March, resulting in a 61-day interval that is a bit more than "two months" long. If the one month is February, then the interval is exactly two months (but only 59 or 60 days). If the one month is April, then the interval is slightly less than two months, for a total of 60 days.

Rather than sort out and deal with all the complications that differing month lengths pose for interval comparison, date arithmetic, and normalization of datetime values, the ISO SQL standard breaks the datetime model into two parts: year and month, and everything else. (For more, see C. J. Date's *A Guide to the SQL Standard* [Addison-Wesley].)

Declaring INTERVAL Variables

Compared to other PL/SQL variable declarations, the syntax for declaring INTERVAL variables is a bit unusual. You not only have multiple-word type names, but in one case you specify not one precision, but two:

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

or:

```
    var_name INTERVAL DAY [(day_precision)] TO SECOND [(frac_sec_prec)]
```

where:

var_name
: Is the name of the INTERVAL variable that you want to declare.

year_precision
: Is the number of digits (from 0 to 4) that you want to allow for a year value. The default is 2.

day_precision
: Is the number of digits (from 0 to 9) that you want to allow for a day value. The default is 2.

frac_sec_prec
: Is the number of digits (from 0 to 9) that you want to allow for fractional seconds (i.e., the fractional seconds precision). The default is 6.

It is the nature of intervals that you need only worry about precision at the extremes. INTERVAL YEAR TO MONTH values are always normalized such that the number of months is between 0 and 11. In fact, the database will not allow you to specify a month greater than 11; an interval of 1 year, 13 months, must be expressed as 2 years, 1 month. The *year_precision* fixes the maximum size of the interval. Likewise, the *day_precision* in INTERVAL DAY TO SECOND fixes the maximum size of that interval.

You don't need to specify a precision for the hour, minute, and second values for an INTERVAL DAY TO SECOND variable, for the same reason you don't specify a precision for month in an INTERVAL YEAR TO MONTH. The intervals are always normalized so that any values for hour, minute, and second are within the normal ranges of 0–23 for hours, 0–59 for minutes, and 0–59 for seconds (excluding fractional seconds).

The fractional second precision (*frac_sec_prec*) is necessary because INTERVAL DAY TO SECOND values can resolve intervals down to the fraction of a second. INTERVAL YEAR TO MONTH values don't handle fractional months, so no fractional month precision is necessary.

When to Use INTERVALs

Use the INTERVAL types whenever you need to work with quantities of time. I provide two examples in this section, hoping to spark your natural creativity so that you can begin to think about how you might use INTERVAL types in systems you develop.

Finding the difference between two datetime values

One use for INTERVAL types is when you need to look at the difference between two datetime values. Consider the following example, which computes an employee's length of service:

```
/* File on web: interval_between.sql */
DECLARE
   start_date TIMESTAMP;
   end_date TIMESTAMP;
   service_interval INTERVAL YEAR TO MONTH;
   years_of_service NUMBER;
   months_of_service NUMBER;
BEGIN
   -- Normally, we would retrieve start and end dates from a database.
   start_date := TO_TIMESTAMP('29-DEC-1988','dd-mon-yyyy');
   end_date := TO_TIMESTAMP ('26-DEC-1995','dd-mon-yyyy');

   -- Determine and display years and months of service
   service_interval := (end_date - start_date) YEAR TO MONTH;
   DBMS_OUTPUT.PUT_LINE(service_interval);

   -- Use the new EXTRACT function to grab individual
   -- year and month components.
   years_of_service := EXTRACT(YEAR FROM service_interval);
   months_of_service := EXTRACT(MONTH FROM service_interval);
   DBMS_OUTPUT.PUT_LINE(years_of_service || ' years and '
                     || months_of_service || ' months');
END;
```

The line that performs the actual calculation to get years and months of service is:

```
service_interval := (end_date - start_date) YEAR TO MONTH;
```

The YEAR TO MONTH is part of the interval expression syntax. I talk more about that syntax in "Datetime Arithmetic" on page 311. You can see, however, that computing the interval is as simple as subtracting one timestamp from another. Had I not used an INTERVAL type, I would have had to code something like the following:

```
months_of_service := ROUND(months_between(end_date, start_date));
years_of_service  := TRUNC(months_of_service/12);
months_of_service := MOD(months_of_service,12);
```

I believe the non-INTERVAL solution is more complex to code and understand.

The INTERVAL YEAR TO MONTH type displays rounding behavior, and it's important you understand the ramifications of that. See "Datetime Arithmetic" on page 311 for details about this issue.

Designating periods of time

For this example, I will explore a company with an assembly line. The time required to assemble each product (called *build time* in this example) is an important metric. Reducing this interval allows the assembly line to be more efficient, so management wants to track and report on this interval. In my example, each product has a tracking number used to identify it during the assembly process. The table I use to hold this assembly information looks like this:

```
TABLE assemblies (
    tracking_id NUMBER NOT NULL,
    start_time  TIMESTAMP NOT NULL,
    build_time  INTERVAL DAY TO SECOND
);
```

Next, I need a PL/SQL function to return the build time for a given tracking_id. The build time is calculated from the current timestamp minus the start time. I will cover date arithmetic in greater detail later in this chapter. This build time function is:

```
FUNCTION calc_build_time (
   esn IN assemblies.tracking_id%TYPE
)
   RETURN DSINTERVAL_UNCONSTRAINED
IS
   start_ts assemblies.start_time%TYPE;
BEGIN
   SELECT start_time INTO start_ts FROM assemblies
      WHERE tracking_id = esn;
   RETURN LOCALTIMESTAMP-start_ts;
END;
```

When I pass intervals into and out of PL/SQL programs, I need to use the unconstrained keywords (see "Using Unconstrained INTERVAL Types" on page 318 for an explanation). With the build time recorded in a table, I can analyze the data more easily. I can calculate the minimum, maximum, and mean build times with simple SQL functions. I could answer questions like "Do I build any faster on Monday versus Wednesday?" or "How about first shift versus second shift?" But I'm getting ahead of myself. This straightforward example simply demonstrates the basic concept of a day-to-second interval. Your job as a clever developer is to put these concepts to use in creative ways.

Datetime Conversions

Now that you understand the Oracle database's array of datetime datatypes, it's time to look at how you get dates into and out of datetime variables. Human-readable datetime values are character strings such as "March 5, 2009" and "10:30 a.m.", so this discussion centers on the conversion of datetime values from character strings to Oracle's internal representation, and vice versa.

PL/SQL validates and stores dates that fall from January 1, 4712 B.C.E. through December 31, 9999 A.D. (Oracle documentation indicates a maximum date of December 31, 4712; run the *showdaterange.sql* script, available on the book's website, to verify the range on your version.) If you enter a date without a time (many applications do not require the tracking of time, so PL/SQL lets you leave it off), the time portion of the value defaults to midnight (12:00:00 a.m.).

The database can interpret just about any date or time format you throw at it. Key to that flexibility is the concept of a *date format model*, which is a string of special characters that define a date's format to the database. For example, if your input date is 15-Nov-1961, that—rather obviously in this case—corresponds to the date format dd-mon-yyyy. You then use the string 'dd-mon-yyyy' in calls to conversion functions to convert dates to and from that format.

I show examples of several different format models in the following conversion discussion, and I provide a complete reference to all the format model elements in Appendix C.

From Strings to Datetimes

The first issue you'll face when working with dates is that of getting date (and time) values into your PL/SQL datetime variables. You do so by converting datetime values from character strings to the database's internal format. You can do such conversions implicitly via assignment of a character string directly to a datetime variable, or—better yet—explicitly via one of Oracle's built-in conversion functions.

Implicit conversion is risky, and I don't recommend it. Following is an example of implicit conversion from a character string to a DATE variable:

```
DECLARE
   birthdate DATE;
BEGIN
   birthdate := '15-Nov-1961';
END;
```

Such a conversion relies on the NLS_DATE_FORMAT setting and will work fine until the day your DBA decides to change that setting. On that day, all your date-related code will break. Changing NLS_DATE_FORMAT at the session level can also break such code.

Rather than relying on implicit conversions and the NLS_DATE_FORMAT setting, it's far safer to convert dates explicitly via one of the built-in conversion functions, such as TO_DATE:

```
DECLARE
   birthdate DATE;
BEGIN
   birthdate := TO_DATE('15-Nov-1961','dd-mon-yyyy');
END;
```

Notice here the use of the format string 'dd-mon-yyyy' as the second parameter in the call to TO_DATE. That format string controls how the TO_DATE function interprets the characters in the first parameter.

PL/SQL supports the following functions to convert strings to dates and timestamps:

TO_DATE(`string`[, `format_mask`[, `nls_language`]])
: Converts a character string to a value of type DATE.

TO_DATE(`number`[, `format_mask`[, `nls_language`]])
: Converts a number representing a Julian date into a value of type DATE.

TO_TIMESTAMP(`string`[, `format_mask`[, `nls_language`]])
: Converts a character string to a value of type TIMESTAMP.

TO_TIMESTAMP_TZ(`string`[, `format_mask`[, `nls_language`]])
: Converts a character string to a value of type TIMESTAMP WITH TIME ZONE. Also use this function when your target is TIMESTAMP WITH LOCAL TIME ZONE.

Not only do these functions make it clear in your code that a type conversion is occurring, but they also allow you to specify the exact datetime format being used.

> The second version of TO_DATE can be used only with the format mask of J for Julian date. The Julian date is the number of days that have passed since January 1, 4712 B.C. Only in this use of TO_DATE can a number be passed as the first parameter of TO_DATE.

For all other cases the parameters are as follows:

string
: Is the string variable, literal, named constant, or expression to be converted.

format_mask
: Is the format mask to be used in converting the string. The format mask defaults to the NLS_DATE_FORMAT setting.

nls_language

Optionally specifies the language to be used to interpret the names and abbreviations of both months and days in the string. Here's the format of *nls_language*:

```
'NLS_DATE_LANGUAGE=language'
```

where *language* is a language recognized by your instance of the database. You can determine the acceptable languages by checking Oracle's *Globalization Support Guide*.

The format elements described in Appendix C apply when you're using the TO_ family of functions. For example, the following calls to TO_DATE and TO_TIMESTAMP convert character strings of varying formats to DATE and TIMESTAMP values:

```
DECLARE
   dt DATE;
   ts TIMESTAMP;
   tstz TIMESTAMP WITH TIME ZONE;
   tsltz TIMESTAMP WITH LOCAL TIME ZONE;
BEGIN
   dt := TO_DATE('12/26/2005','mm/dd/yyyy');
   ts := TO_TIMESTAMP('24-Feb-2002 09.00.00.50 PM');
   tstz := TO_TIMESTAMP_TZ('06/2/2002 09:00:00.50 PM EST',
                     'mm/dd/yyyy hh:mi:ssxff AM TZD');
   tsltz := TO_TIMESTAMP_TZ('06/2/2002 09:00:00.50 PM EST',
                     'mm/dd/yyyy hh:mi:ssxff AM TZD');
   DBMS_OUTPUT.PUT_LINE(dt);
   DBMS_OUTPUT.PUT_LINE(ts);
   DBMS_OUTPUT.PUT_LINE(tstz);
   DBMS_OUTPUT.PUT_LINE(tsltz);
END;
```

The output is:

```
26-DEC-05
24-FEB-02 09.00.00.500000 PM
02-JUN-02 09.00.00.500000 PM -05:00
02-JUN-02 09.00.00.500000 PM
```

Note the decimal seconds (.50) and the use of XFF in the format mask. The X format element specifies the location of the radix character, in this case a period (.), separating the whole seconds from the fractional seconds. I could just as easily have specified a period, as in ".FF", but I chose to use X instead. The difference is that when X is specified, the database determines the correct radix character based on the current NLS_TERRITORY setting.

Any Oracle errors between ORA-01800 and ORA-01899 are related to date conversion. You can learn some of the date conversion rule nuances by perusing the different errors and reading about the documented causes of these errors. Some of these nuances are:

- A date literal passed to TO_CHAR for conversion to a date cannot be longer than 220 characters.
- You can't include both a Julian date element (J) and the day of year element (DDD) in a single format mask.
- You can't include multiple elements for the same component of the date/time in the mask. For example, the format mask YYYY-YYY-DD-MM is illegal because it includes two year elements, YYYY and YYY.
- You can't use the 24-hour time format (HH24) and a meridian element (e.g., a.m.) in the same mask.

As the preceding example demonstrates, the TO_TIMESTAMP_TZ function can convert character strings that include time zone information. And while time zones may seem simple on the surface, they are anything but, as you'll see in "Working with Time Zones" on page 295.

From Datetimes to Strings

Getting values into datetime variables is only half the battle. The other half is getting them out again in some sort of human-readable format. Oracle provides the TO_CHAR function for that purpose.

The TO_CHAR function can be used to convert a datetime value to a variable-length string. This single function works for DATE types as well as for all the types in the TIMESTAMP family. TO_CHAR is also used to convert numbers to character strings, as covered in Chapter 9. The following specification describes TO_CHAR for datetime values:

```
FUNCTION TO_CHAR
   (date_in IN DATE
    [, format_mask IN VARCHAR2
    [, nls_language IN VARCHAR2]])
RETURN VARCHAR2
```

where:

date_in
: Is the date to be converted to character format.

format_mask
: Is the mask, made up of one or more of the date format elements. See Appendix C for a list of date format elements.

nls_language
: Is a string specifying a date language.

Both the *format_mask* and *nls_language* parameters are optional.

If you want your results to be in the national character set, you can use TO_NCHAR in place of TO_CHAR. Be certain you provide your date format string in the national character set as well, though. Otherwise, you may receive *ORA-01821: date format not recognized* errors.

If *format_mask* is not specified, the default date format for the database instance is used. This format is 'DD-MON-RR', unless you have nondefault NLS settings, such as NLS_DATE_FORMAT. The best practice, as mentioned elsewhere in this chapter, is to not rely on implicit conversions for dates. Changes to the server NLS settings and, for client-side code, changes to the client NLS settings will cause logic bugs to creep into your programs if you rely on implicit conversions. As an example, in North America you may write a routine assuming that the date 03-04-09 is 4 March 2009, but if your application is later deployed to Japan or Germany the implicit conversion will result in 3 April 2009 or 9 April 2003, depending on the NLS settings. If your application is always explicit in datatype conversions, you will not encounter these logic bugs.

Here are some examples of TO_CHAR being used for date conversion:

- Notice that there are two blanks between month and day and a leading zero for the fifth day:

  ```
  TO_CHAR (SYSDATE, 'Month DD, YYYY') --> 'February  05, 1994'
  ```

- Use the FM fill mode element to suppress blanks and zeros:

  ```
  TO_CHAR (SYSDATE, 'FMMonth DD, YYYY') --> 'February 5, 1994'
  ```

- Note the case difference on the month abbreviations of the next two examples. You get exactly what you ask for with Oracle date formats!

  ```
  TO_CHAR (SYSDATE, 'MON DDth, YYYY') --> 'FEB 05TH, 1994'
  TO_CHAR (SYSDATE, 'fmMon DDth, YYYY') --> 'Feb 5TH, 1994'
  ```

  The TH format is an exception to the capitalization rules. Even if you specify lowercase "th" in a format string, the database will use uppercase TH in the output.

- Show the day of the year, day of the month, and day of the week for the date (with fm used here as a toggle):

  ```
  TO_CHAR (SYSDATE, 'DDD DD D ') --> '036 05 7'
  TO_CHAR (SYSDATE, 'fmDDD fmDD D ') --> '36 05 7'
  ```

- Here's some fancy formatting for reporting purposes:

  ```
  TO_CHAR (SYSDATE, '"In month "RM" of year "YEAR')
     --> 'In month II   of year NINETEEN NINETY FOUR'
  ```

- For TIMESTAMP variables, you can specify the time down to the millisecond using the FF format element:

```
       TO_CHAR (A_TIMESTAMP, 'YYYY-MM-DD HH:MI:SS.FF AM TZH:TZM')
          --> a value like: 2002-02-19 01:52:00.123457000 PM -05:00
```

Be careful when dealing with fractional seconds. The FF format element represents fractional seconds in the output format model, and you'll be tempted to use the number of Fs to control the number of decimal digits in the output. Don't do that! Instead, use FF1 through FF9 to specify one through nine decimal digits. For example, the following block uses FF6 to request six decimal digits of precision in the output:

```
DECLARE
   ts TIMESTAMP WITH TIME ZONE;
BEGIN
   ts := TIMESTAMP '2002-02-19 13:52:00.123456789 -5:00';
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts,'YYYY-MM-DD HH:MI:SS.FF6 AM TZH:TZM'));
END;
```

The output is:

```
2002-02-19 01:52:00.123457 PM -05:00
```

Note the rounding that occurred. The number of seconds input was 00.123456789. That value was rounded (not truncated) to six decimal digits: 00.123457.

It's easy to slip up and specify an incorrect date format, and the introduction of TIMESTAMP types has made this even easier. Format elements that are valid with TIMESTAMP types are not valid for the DATE type. Look at the results in the following example when FF, TZH, and TZM are used to convert a DATE value to a character string:

```
DECLARE
   dt DATE;
BEGIN
   dt := SYSDATE;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(dt,'YYYY-MM-DD HH:MI:SS.FF AM TZH:TZM'));
END;
```

The output is:

```
   dt := SYSDATE;
*
ORA-01821: date format not recognized
ORA-06512: at line 5
```

The error message you get in this case, *ORA-01821: date format not recognized*, is confusing and misleading. The date format is just fine. The problem is that it's being applied to the wrong datatype. Watch for this kind of problem when you write code. If you get an *ORA-01821* error, check *both* the date format and the datatype that you are trying to convert.

Working with Time Zones

The inclusion of time zone information makes the use of TO_TIMESTAMP_TZ more complex than use of the TO_DATE and TO_TIMESTAMP functions. You may specify time zone information in any of the following ways:

- Using a positive or negative displacement of some number of hours and minutes from UTC time; for example, −5:00 is equivalent to U.S. Eastern Standard Time. Displacements must fall into the range −12:59 to +13:59. (I showed examples of this notation earlier in this chapter.)
- Using a time zone region name such as US/Eastern, US/Pacific, and so forth.
- Using a combination of time zone region name and abbreviation, as in US/Eastern EDT for U.S. Eastern Daylight Saving Time.

Let's look at some examples. I'll begin with a simple example that leaves off time zone information entirely:

```
TO_TIMESTAMP_TZ ('12312005 083015.50', 'MMDDYYYY HHMISS.FF')
```

The date and time in this example work out to be 31-Dec-2005 at 15 1/2 seconds past 8:30 a.m. Because no time zone is specified, the database will default to the current session time zone. With the time zone intentionally omitted, this code is less clear than it could be. If the application is designed to use the session time zone (as opposed to an explicit time zone), a better approach would be to first fetch the session time zone using the function SESSIONTIMEZONE and then explicitly use this value in the TO_TIMESTAMP_TZ function call. Being explicit in your intent will help developers (including you) understand and correctly maintain this code two years down the road when some new feature or bug fix is needed.

A Date or a Time?

Be aware that every datetime value is composed of both a date *and* a time. Forgetting this duality may lead to errors in your code. As an example, suppose that I write PL/SQL code to run on the first of the year, 2015:

```
IF SYSDATE = TO_DATE('1-Jan-2015','dd-Mon-yyyy')
THEN
   Apply2015PriceChange;
END IF;
```

The goal of this example is to run a routine to adjust prices for the new year, but the chance of that procedure actually running is minimal. You'd need to run the code block exactly at midnight, to the second. That's because SYSDATE returns a time-of-day value along with the date.

To make the code block work as expected, you can truncate the value returned by SYSDATE to midnight on the day in question:

```
IF TRUNC(SYSDATE) = TO_DATE('1-Jan-2015','dd-Mon-yyyy');
```

Now, both sides of the comparison have a time of day, but that time of day is midnight. The TO_DATE function also returns a time of day, which, because no time of day was given, defaults to midnight (i.e., 00:00:00). Thus, no matter when on 1 Jan 2015 you run this code block, the comparison will succeed, and the Apply2009PriceChange procedure will run.

This use of TRUNCATE to remove the time portion of a date stamp works equally well on timestamps.

Next, let's represent the time zone using a displacement of hours and minutes from UTC. Note the use of TZH and TZM to denote the location of the hour and minute displacements in the input string:

```
TO_TIMESTAMP_TZ ('1231200 083015.50 –5:00', 'MMDDYY HHMISS.FF TZH:TZM')
```

In this example, the datetime value is interpreted as being an Eastern Standard Time value (regardless of your session time zone).

The next example shows the time zone being specified using a time zone region name. The example specifies EST, which is the region name corresponding to Eastern Time in the United States. Note the use of TZR in the format mask to designate where the time zone region name appears in the input string:

```
TO_TIMESTAMP_TZ ('02-Nov-2014 01:30:00 EST',
                 'dd-Mon-yyyy hh:mi:ss TZR')
```

This example is interesting in that it represents Eastern Time, not Eastern Standard Time. The difference is that "Eastern Time" can refer to either Eastern Standard Time or Eastern Daylight Time, depending on whether daylight saving time is in effect. And it might be in effect! I've carefully crafted this example to make it ambiguous. 02-Nov-2014 is the date on which Eastern Daylight Time ends, and at 2:00 a.m. the time rolls back to 1:00 a.m. So on that date, 1:30 a.m. actually comes around twice! The first time it's 1:30 a.m. Eastern Daylight Time, and the second time it's 1:30 a.m. Eastern Standard Time. So what time is it, really, when I say it's 1:30 a.m. on 02-Nov-2014?

If you set the session parameter ERROR_ON_OVERLAP_TIME to TRUE (the default is FALSE), the database will give you an error whenever you specify an ambiguous time because of daylight saving time changes. Note that daylight saving time is also called *summer time* in some parts of the world.

The time zone region name alone doesn't distinguish between standard time and daylight saving time. To remove the ambiguity, you also must specify a time zone abbreviation, which I've done in the next two examples. Use the abbreviation EDT to specify Eastern Daylight Time:

```
TO_TIMESTAMP_TZ ('02-Nov-2014 01:30:00.00 US/Eastern EDT',
                 'dd-Mon-yyyy hh:mi:ssxff TZR TZD')
```

And use the abbreviation EST to specify Eastern Standard Time:

```
TO_TIMESTAMP_TZ ('02-Nov-2014 01:30:00.00 US/Eastern EST',
                 'dd-Mon-yyyy hh:mi:ssxff TZR TZD')
```

To avoid ambiguity, I recommend that you either specify a time zone offset using hours and minutes (as in –5:00) or use a combination of full region name and time zone abbreviation (as in US/Eastern EDT). If you use the region name alone and there's ambiguity with respect to daylight saving time, the database will resolve the ambiguity by assuming that standard time applies.

If you're initially confused by the fact that EST, CST, and PST all can be both a region name and an abbreviation, you're not alone. I was confused by this too. Depending on your time zone file version, EST, CST, MST, and PST may appear as both region and abbreviation. You can further qualify each of those region names using the same string of three characters as a time zone abbreviation. The result (e.g., EST EST or CST CST) is standard time for the region in question. The best practice is to use the full region name, like US/Eastern or America/Detroit, instead of the three-letter abbreviation EST. See Oracle's Metalink Note 340512.1 *Timestamps & time zones—Frequently Asked Questions* for more information.

You can get a complete list of the time zone region names and time zone abbreviations that Oracle supports by querying the V$TIMEZONE_NAMES view. Any database user can access that view. When you query it, notice that time zone abbreviations are not unique (see the sidebar "A Time Zone Standard?").

A Time Zone Standard?

As important as time zones are, you would think there would be some sort of international standard specifying their names and abbreviations. Well, there isn't one. Not only are time zone abbreviations not standardized, but some are also duplicated. For example, EST is used in the U.S. for Eastern Standard Time, and also in Australia for Eastern Standard Time, and I assure you that the two Eastern Standard Times are not at all the same! In addition, BST is the abbreviation for several time zones, including those for Pacific/Midway and Europe/London, which are 12 hours different during daylight saving time and 11 hours different during the rest of the year. This is why the TO_TIMESTAMP functions do not allow you to specify a time zone using the abbreviation alone.

Because there is no time zone standard, you might well ask the source of all those time zone region names in V$TIMEZONE_NAMES. Oracle's source for that information can be found at *ftp://elsie.nci.nih.gov/pub/*. Look for files named something like *tzdata xxx.tar.gz*, where *xxx* is the version of the data. This archive usually has a file named *tz-link.htm* that contains more information and links to other URLs related to time zones.

Requiring a Format Mask to Match Exactly

When converting a character string to a datetime, the TO_* conversion functions normally make a few allowances:

- Extra blanks in the character string are ignored.
- Numeric values, such as the day number or the year, do not have to include leading zeros to fill out the mask.
- Punctuation in the string to be converted can simply match the length and position of punctuation in the format.

This kind of flexibility is great—until you want to actually restrict a user or even a batch process from entering data in a nonstandard format. In some cases, it simply is not OK when a date string has a caret (^) instead of a hyphen (-) between the day and month numbers. For these situations, you can use the FX modifier in the format mask to enforce an exact match between string and format model.

With FX, there is no flexibility in the interpretation of the string. It cannot have extra blanks if none are found in the model. Its numeric values must include leading zeros if the format model specifies additional digits. And the punctuation and literals must exactly match the punctuation and quoted text of the format mask (except for case, which is always ignored). In all of the following examples:

```
TO_DATE ('1-1-4', 'fxDD-MM-YYYY')
TO_DATE ('7/16/94', 'FXMM/DD/YY')
TO_DATE ('JANUARY^1^ the year of 94', 'FXMonth-dd-"WhatIsaynotdo"yy')
```

PL/SQL raises one of the following errors:

```
ORA-01861: literal does not match format string
ORA-01862: the numeric value does not match the length of the format item
```

However, the following example succeeds because case is always irrelevant, and FX does not change that:

```
TO_DATE ('Jan 15 1994', 'fxMON DD YYYY')
```

The FX modifier can be specified in uppercase, lowercase, or mixed case; the effect is the same.

The FX modifier is a toggle, and it can appear more than once in a format model. For example:

```
TO_DATE ('07-1-1994', 'FXDD-FXMM-FXYYYY')
```

Each time it appears in the format, FX changes the effect of the modifier. In this example, an exact match is required for the day number and the year number but not for the month number.

Easing Up on Exact Matches

You can use FM (fill mode) in the format model of a call to a TO_DATE or TO_TIMESTAMP function to fill a string with blanks or zeros so that a date string that would otherwise fail the FX test will pass. For example:

```
TO_DATE ('07-1-94', 'FXfmDD-FXMM-FXYYYY')
```

This conversion succeeds because FM causes the year 94 to be filled out with 00, so the year becomes 0094 (probably not behavior you would ever want). The day 1 is filled out with a single zero to become 01. FM is a toggle, just like FX.

Using FM as I've just described seems at first to defeat the purpose of FX. Why use both? One reason is that you might use FX to enforce the use of specific delimiters while using FM to ease up on the requirement that users enter leading zeros.

Interpreting Two-Digit Years in a Sliding Window

The last millennium change caused an explosion of interest in using four-digit years as people suddenly realized the ambiguity inherent in the commonly used two-digit year. For example, does 1-Jan-45 refer to 1945 or 2045? The best practice is to use unambiguous four-digit years. But despite this realization, habits are tough to break, and existing systems can be difficult to change, so you may find yourself still needing to allow your users to enter dates using two-digit years rather than four-digit years. To help, Oracle provides the RR format element to interpret two-digit years in a sliding window.

In the following discussion, I use the term *century* colloquially. RR's 20th century is composed of the years 1900–1999, and its 21st century is composed of the years 2000–2099. I realize this is not the proper definition of century, but it's a definition that makes it easier to explain RR's behavior.

If the current year is in the first half of the century (years 0 through 49), then:

- If you enter a date in the first half of the century (i.e., from 0 through 49), RR returns the current century.
- If you enter a date in the latter half of the century (i.e., from 50 through 99), RR returns the previous century.

On the other hand, if the current year is in the latter half of the century (years 50 through 99), then:

- If you enter a date in the first half of the century, RR returns the next century.
- If you enter a date in the latter half of the century, RR returns the current century.

Confusing? I had to think about it for a while too. The RR rules are an attempt to make the best guess as to which century is intended when a user leaves off that information. Let's look at some examples of the impact of RR. Notice that for year 88 and year 18, SYSDATE returns a current date in the 20th and 21st centuries, respectively:

```
SQL> SELECT TO_CHAR (SYSDATE, 'MM/DD/YYYY') "Current Date",
  2         TO_CHAR (TO_DATE ('14-OCT-88', 'DD-MON-RR'), 'YYYY') "Year 88",
  3         TO_CHAR (TO_DATE ('14-OCT-18', 'DD-MON-RR'), 'YYYY') "Year 18"
  FROM dual;

    Current Date Year 88 Year 18
    ------------ ------- -------
      02/25/2014    1988    2018
```

When we reach the year 2050, RR will interpret the same dates differently:

```
SQL> SELECT TO_CHAR (SYSDATE, 'MM/DD/YYYY') "Current Date",
  2         TO_CHAR (TO_DATE ('10/14/88', 'MM/DD/RR'), 'YYYY') "Year 88",
  3         TO_CHAR (TO_DATE ('10/14/18', 'MM/DD/RR'), 'YYYY') "Year 18"
  4    FROM dual;

    Current Date Year 88 Year 18
    ------------ ------- -------
      02/25/2050    2088    2118
```

There are a number of ways you can activate the RR logic in your current applications. The cleanest and simplest way is to change the default format mask for dates in your database instance(s). In fact, Oracle has already done this for us. On a default Oracle install, you will find your NLS_DATE_FORMAT equivalent to the result of:

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

Then, if you have not hardcoded the date format mask anywhere else in your screens or reports, any two-digit years will be interpreted according to the windowing rules I've just described.

Converting Time Zones to Character Strings

Time zones add complexity to the problem of converting datetime values to character strings. Time zone information consists of the following elements:

- A displacement from UTC in terms of hours and minutes
- A time zone region name
- A time zone abbreviation

All these elements are stored separately in a TIMESTAMP WITH TIME ZONE variable. The displacement from UTC is always present, but whether you can display the region name or abbreviation depends on whether you've specified that information to begin with. Look closely at this example:

```
DECLARE
   ts1 TIMESTAMP WITH TIME ZONE;
   ts2 TIMESTAMP WITH TIME ZONE;
   ts3 TIMESTAMP WITH TIME ZONE;
BEGIN
     ts1 := TO_TIMESTAMP_TZ('2002-06-18 13:52:00.123456789 −5:00',
                         'YYYY-MM-DD HH24:MI:SS.FF TZH:TZM');
     ts2 := TO_TIMESTAMP_TZ('2002-06-18 13:52:00.123456789 US/Eastern',
                         'YYYY-MM-DD HH24:MI:SS.FF TZR');
     ts3 := TO_TIMESTAMP_TZ('2002-06-18 13:52:00.123456789 US/Eastern EDT',
                         'YYYY-MM-DD HH24:MI:SS.FF TZR TZD');

     DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts1,
        'YYYY-MM-DD HH:MI:SS.FF AM TZH:TZM TZR TZD'));
     DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts2,
        'YYYY-MM-DD HH:MI:SS.FF AM TZH:TZM TZR TZD'));
     DBMS_OUTPUT.PUT_LINE(TO_CHAR(ts3,
        'YYYY-MM-DD HH:MI:SS.FF AM TZH:TZM TZR TZD'));
  END;
```

The output is:

```
2002-06-18 01:52:00.123457000 PM −05:00 −05:00
2002-06-18 01:52:00.123457000 PM −04:00 US/EASTERN EDT
2002-06-18 01:52:00.123457000 PM −04:00 US/EASTERN EDT
```

Note the following with respect to the display of time zone information:

- For ts1, I specified time zone in terms of a displacement from UTC. Thus, when ts1 was displayed, only the displacement could be displayed.
- In the absence of a region name for ts1, the database provided the time zone displacement. This is preferable to providing no information at all.

- For ts2, I specified a time zone region. That region was translated internally into an offset from UTC, but the region name was preserved. Thus, both the UTC offset and the region name could be displayed.
- For ts2, the database correctly recognized that daylight saving time is in effect during the month of June. As a result, the value of ts2 was implicitly associated with the EDT abbreviation.
- For ts3, I specified a time zone region and an abbreviation, and both those values could be displayed. No surprises here.

There's a one-to-many relationship between UTC offsets and time zone regions; the offset alone is not enough to get you to a region name. That's why you can't display a region name unless you specify one to begin with.

Padding Output with Fill Mode

The FM modifier described in "Easing Up on Exact Matches" on page 299 can also be used when converting *from* a datetime *to* a character string, to suppress padded blanks and leading zeros that would otherwise be returned by the TO_CHAR function.

By default, the following format mask results in both padded blanks and leading zeros (there are five spaces between the month name and the day number):

```
TO_CHAR (SYSDATE, 'Month DD, YYYY') --> 'April     05, 1994'
```

With the FM modifier at the beginning of the format mask, however, both the extra blank and the leading zeros disappear:

```
TO_CHAR (SYSDATE, 'FMMonth DD, YYYY') --> April 5, 1994'
```

The modifier can be specified in uppercase, lowercase, or mixed case; the effect is the same.

Remember that the FM modifier is a toggle and can appear more than once in a format model. Each time it appears in the format, it changes the effect of the modifier. By default (that is, if FM is not specified anywhere in a format mask), blanks are not suppressed, and leading zeros are included in the result value.

Date and Timestamp Literals

Date and timestamp literals, as well as the interval literals that appear later in this chapter, are part of the ISO SQL standard and have been supported since Oracle9*i* Database. They represent yet another option for you to use in getting values into datetime variables. A *date literal* consists of the keyword DATE followed by a date (and only a date) value in the following format:

```
DATE 'YYYY-MM-DD'
```

A *timestamp literal* consists of the keyword TIMESTAMP followed by a datetime value in a very specific format:

```
TIMESTAMP 'YYYY-MM-DD HH:MI:SS[.FFFFFFFFF] [{+|-}HH:MI]'
```

The `FFFFFFFFF` represents fractional seconds and is optional. If you specify fractional seconds, you may use anywhere from one to nine digits. The time zone displacement (`+HH:MI`) is optional and may use either a plus or a minus sign as necessary. The hours are always with respect to a 24-hour clock.

If you omit the time zone displacement in a timestamp literal, the time zone will default to the session time zone.

The following PL/SQL block shows several valid date and timestamp literals:

```
DECLARE
   ts1 TIMESTAMP WITH TIME ZONE;
   ts2 TIMESTAMP WITH TIME ZONE;
   ts3 TIMESTAMP WITH TIME ZONE;
   ts4 TIMESTAMP WITH TIME ZONE;
   ts5 DATE;
BEGIN
   -- Two digits for fractional seconds
   ts1 := TIMESTAMP '2002-02-19 11:52:00.00 -05:00';

   -- Nine digits for fractional seconds, 24-hour clock, 14:00 = 2:00 PM
   ts2 := TIMESTAMP '2002-02-19 14:00:00.000000000 -5:00';

   -- No fractional seconds at all
   ts3 := TIMESTAMP '2002-02-19 13:52:00 -5:00';

   -- No time zone, defaults to session time zone
   ts4 := TIMESTAMP '2002-02-19 13:52:00';

   -- A date literal
   ts5 := DATE '2002-02-19';
END;
```

The format for date and timestamp literals is prescribed by the ANSI/ISO standards and cannot be changed by you or by the DBA. Thus, it's safe to use timestamp literals whenever you need to embed a specific datetime value (e.g., a constant) in your code.

Oracle allows the use of time zone region names in timestamp literals —for example: `TIMESTAMP '2002-02-19 13:52:00 EST'`. However, this functionality goes above and beyond the SQL standard.

Interval Conversions

An interval is composed of one or more datetime elements. For example, you might choose to express an interval in terms of years and months, or you might choose to speak in terms of hours and minutes. Table 10-2 lists the standard names for each of the datetime elements used to express intervals. These are the names you must use in conjunction with the conversion functions and expressions described in the subsections that follow. The names are not case sensitive when used with the interval conversion functions. For example, YEAR, Year, and year are all equivalent.

Table 10-2. Interval element names

Name	Description
YEAR	Some number of years, ranging from 1 through 999,999,999
MONTH	Some number of months, ranging from 0 through 11
DAY	Some number of days, ranging from 0 to 999,999,999
HOUR	Some number of hours, ranging from 0 through 23
MINUTE	Some number of minutes, ranging from 0 through 59
SECOND	Some number of seconds, ranging from 0 through 59.999999999

Converting from Numbers to Intervals

The NUMTOYMINTERVAL and NUMTODSINTERVAL functions allow you to convert a single numeric value to one of the interval datatypes. You do this by associating your numeric value with one of the interval elements listed in Table 10-2.

The function NUMTOYMINTERVAL (pronounced "num to Y M interval") converts a numeric value to an interval of type INTERVAL YEAR TO MONTH. The function NUMTODSINTERVAL (pronounced "num to D S interval") likewise converts a numeric value to an interval of type INTERVAL DAY TO SECOND.

Following is an example of NUMTOYMINTERVAL being used to convert 10.5 to an INTERVAL YEAR TO MONTH value. The second argument, 'Year', indicates that the number represents some number of years.

```
DECLARE
   y2m INTERVAL YEAR TO MONTH;
BEGIN
   y2m := NUMTOYMINTERVAL (10.5,'Year');
   DBMS_OUTPUT.PUT_LINE(y2m);
END;
```

The output is:

```
+10-06
```

In this example, 10.5 years was converted to an interval of 10 years, 6 months. Any fractional number of years (in this case, 0.5) will be converted to an equivalent number of months, with the result being rounded to an integer. Thus, 10.9 years will convert to an interval of 10 years, 10 months.

The next example converts a numeric value to an interval of type INTERVAL DAY TO SECOND:

```
DECLARE
  an_interval INTERVAL DAY TO SECOND;
BEGIN
   an_interval := NUMTODSINTERVAL (1440,'Minute');
   DBMS_OUTPUT.PUT_LINE(an_interval);
END;
```

The output is:

```
+01 00:00:00.000000

PL/SQL procedure successfully completed.
```

As you can see, the database has automatically taken care of normalizing the input value of 1,440 minutes to an interval value of 1 day. This is great, because now you don't need to do that work yourself. You can easily display any number of minutes (or seconds or days or hours) in a normalized format that makes sense to the reader. Prior to the introduction of the interval datatypes, you would have needed to write your own code to translate a minute value into the correct number of days, hours, and minutes.

Converting Strings to Intervals

The NUMTO functions are fine if you are converting numeric values to intervals, but what about character string conversions? For those, you can use TO_YMINTERVAL and TO_DSINTERVAL, depending on whether you are converting to an INTERVAL YEAR TO MONTH or an INTERVAL DAY TO SECOND.

TO_YMINTERVAL converts a character string value into an INTERVAL YEAR TO MONTH value and is invoked as follows:

```
TO_YMINTERVAL('Y-M')
```

where *Y* represents some number of years, and *M* represents some number of months. You must supply both values and separate them using a dash.

Likewise, TO_DSINTERVAL converts a character string into an INTERVAL DAY TO SECOND value. Invoke TO_DSINTERVAL using the following format:

```
TO_DSINTERVAL('D HH:MI:SS.FF')
```

where *D* is some number of days, and *HH:MI:SS.FF* represents hours, minutes, seconds, and fractional seconds.

The following example shows an invocation of each of these functions:

```
DECLARE
   y2m INTERVAL YEAR TO MONTH;
   d2s1 INTERVAL DAY TO SECOND;
   d2s2 INTERVAL DAY TO SECOND;
BEGIN
   y2m := TO_YMINTERVAL('40-3'); -- my age
   d2s1 := TO_DSINTERVAL('10 1:02:10');
   d2s2 := TO_DSINTERVAL('10 1:02:10.123'); -- fractional seconds
END;
```

When invoking either function, you must supply all relevant values. You cannot, for example, invoke TO_YMINTERVAL specifying only a year, or invoke TO_DS_INTERVAL leaving off the seconds. You can, however, omit the fractional seconds.

Formatting Intervals for Display

So far in this section on interval conversion, I've relied on the database's implicit conversion mechanism to format interval values for display. And that's pretty much the best that you can do. You can pass an interval to TO_CHAR, but TO_CHAR will ignore any format mask. For example:

```
DECLARE
   y2m INTERVAL YEAR TO MONTH;
BEGIN
   y2m := INTERVAL '40-3' YEAR TO MONTH;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(y2m,'YY "Years" and MM "Months"'));
END;
```

The output is the same as if no format mask had been specified:

```
+000040-03
```

If you're not satisfied with the default conversion of intervals to character strings, you can use the EXTRACT function:

```
DECLARE
   y2m INTERVAL YEAR TO MONTH;
BEGIN
   y2m := INTERVAL '40-3' YEAR TO MONTH;

   DBMS_OUTPUT.PUT_LINE(
      EXTRACT(YEAR FROM y2m) || ' Years and '
      || EXTRACT(MONTH FROM y2m) || ' Months'
   );
END;
```

The output is:

```
40 Years and 3 Months
```

EXTRACT is described in more detail in "CAST and EXTRACT" on page 308.

Interval Literals

Interval literals are similar to timestamp literals and are useful when you want to embed interval values as constants within your code. Interval literals take the following form:

```
INTERVAL 'character_representation' start_element TO end_element
```

where:

character_representation
Is the character string representation of the interval. See "Interval Conversions" on page 304 for a description of how the two interval datatypes are represented in character form.

start_element
Specifies the leading element in the interval.

end_element
Specifies the trailing element in the interval.

Unlike the TO_YMINTERVAL and TO_DSINTERVAL functions, interval literals allow you to specify an interval using any sequence of datetime elements from Table 10-2. There are only two restrictions:

- You must use a consecutive sequence of elements.
- You cannot transition from a month to a day within the same interval.

Following are several valid examples:

```
DECLARE
   y2ma INTERVAL YEAR TO MONTH;
   y2mb INTERVAL YEAR TO MONTH;
   d2sa INTERVAL DAY TO SECOND;
   d2sb INTERVAL DAY TO SECOND;
BEGIN
   /* Some YEAR TO MONTH examples */
   y2ma := INTERVAL '40-3' YEAR TO MONTH;
   y2mb := INTERVAL '40' YEAR;

   /* Some DAY TO SECOND examples */
   d2sa := INTERVAL '10 1:02:10.123' DAY TO SECOND;

   /* Fails in Oracle9i through 11gR2 because of a bug */
   -- d2sb := INTERVAL '1:02' HOUR TO MINUTE;

   /* Following are two workarounds for defining intervals,
      such as HOUR TO MINUTE, that represent only a portion of the
      DAY TO SECOND range */
   SELECT INTERVAL '1:02' HOUR TO MINUTE
   INTO d2sb
   FROM dual;

   d2sb := INTERVAL '1' HOUR + INTERVAL '02' MINUTE;
END;
```

In Oracle9*i* Database through Oracle Database 11*g* Release 2, expressions such as `INTERVAL '1:02' HOUR TO MINUTE` that don't specify a value for each possible element will work from a SQL statement but not from a PL/SQL statement. Furthermore, you'll get an error about using the keyword BULK in the wrong context. This is a bug that I hope to see fixed in a future release.

One very convenient thing that the database will do for you is to normalize interval values. In the following example, 72 hours and 15 minutes is normalized to 3 days, 0 hours, and 15 minutes:

```
DECLARE
   d2s INTERVAL DAY TO SECOND;
BEGIN
   SELECT INTERVAL '72:15' HOUR TO MINUTE INTO d2s FROM DUAL;
   DBMS_OUTPUT.PUT_LINE(d2s);
END;
```

The output is:

```
+03 00:15:00.000000
```

The database will normalize only the high-end value (hours, in this example) of an interval literal. An attempt to specify an interval of 72:75 (72 hours and 75 minutes) results in an error.

CAST and EXTRACT

CAST and EXTRACT are standard SQL functions that are sometimes useful when you are working with datetimes. CAST made its appearance in Oracle8 Database as a mechanism for explicitly identifying collection types, and it was enhanced in Oracle8*i* Database to enable conversion between built-in datatypes. With respect to date and time, you can use CAST to convert datetime values to and from character strings. The EXTRACT function introduced in Oracle9*i* Database allows you to pluck an individual datetime element from a datetime or interval value.

The CAST Function

With respect to date and time, you can use the CAST function to:

- Convert a character string to a datetime value.
- Convert a datetime value to a character string.

- Convert one datetime type (e.g., DATE) to another (e.g., TIMESTAMP).

When used to convert datetimes to and from character strings, CAST respects the NLS parameter settings. Check your settings by querying V$NLS_PARAMETERS, and change them with an ALTER SESSION command. The NLS settings for datetimes are:

NLS_DATE_FORMAT
: When casting to or from a DATE

NLS_TIMESTAMP_FORMAT
: When casting to or from a TIMESTAMP or a TIMESTAMP WITH LOCAL TIME ZONE

NLS_TIMESTAMP_TZ_FORMAT
: When casting to or from a TIMESTAMP WITH TIME ZONE

The following example illustrates the use of CAST for each of these datetime types. The example assumes the default values of 'DD-MON-RR', 'DD-MON-RR HH.MI.SSXFF AM', and 'DD-MON-RR HH.MI.SSXFF AM TZR' for NLS_DATE_FORMAT, NLS_TIMESTAMP_FORMAT, and NLS_TIMESTAMP_TZ_FORMAT, respectively:

```
DECLARE
   tstz TIMESTAMP WITH TIME ZONE;
   string VARCHAR2(40);
   tsltz TIMESTAMP WITH LOCAL TIME ZONE;
BEGIN
   -- convert string to datetime
   tstz := CAST ('24-Feb-2009 09.00.00.00 PM US/Eastern'
                 AS TIMESTAMP WITH TIME ZONE);
   -- convert datetime back to string
   string := CAST (tstz AS VARCHAR2);
   tsltz := CAST ('24-Feb-2009 09.00.00.00 PM'
                  AS TIMESTAMP WITH LOCAL TIME ZONE);

   DBMS_OUTPUT.PUT_LINE(tstz);
   DBMS_OUTPUT.PUT_LINE(string);
   DBMS_OUTPUT.PUT_LINE(tsltz);
END;
```

The output is:

```
24-FEB-09 09.00.00.000000 PM US/EASTERN
24-FEB-09 09.00.00.000000 PM US/EASTERN
24-FEB-09 09.00.00.000000 PM
```

This example generates a TIMESTAMP WITH TIME ZONE from a character string, converts that value to a VARCHAR2, and finally converts a character string to a TIMESTAMP WITH LOCAL TIME ZONE.

You might be asking yourself when you should use CAST. CAST does have some overlap with the TO_DATE, TO_TIMESTAMP, and TO_TIMESTAMP_TZ functions. However, the TO_TIMESTAMP function can take only a string as input, whereas CAST can take a string or a DATE as input and convert it to TIMESTAMP. So, use CAST when you have requirements that the TO_ functions can't handle. However, when there's a TO_ function that will fit the need, you should use the TO_ function, as it generally leads to more readable code.

> In a SQL statement, you can specify the size of a datatype in a CAST, as in CAST (x AS VARCHAR2(40)). However, PL/SQL does not allow you to specify the size of the target datatype.

The EXTRACT Function

The EXTRACT function is used to extract date components from a datetime value. Use the following format when invoking EXTRACT:

```
EXTRACT (component_name, FROM {datetime | interval})
```

In this syntax, *component_name* is the name of a datetime element listed in Table 10-3. Component names are not case sensitive. Replace *datetime* or *interval* with a valid datetime or interval value. The function's return type depends on the component you are extracting.

Table 10-3. Datetime component names for use with EXTRACT

Component name	Return datatype
YEAR	NUMBER
MONTH	NUMBER
DAY	NUMBER
HOUR	NUMBER
MINUTE	NUMBER
SECOND	NUMBER
TIMEZONE_HOUR	NUMBER
TIMEZONE_MINUTE	NUMBER
TIMEZONE_REGION	VARCHAR2
TIMEZONE_ABBR	VARCHAR2

The following example shows EXTRACT being used to check whether the current month is November:

```
BEGIN
   IF EXTRACT (MONTH FROM SYSDATE) = 11 THEN
      DBMS_OUTPUT.PUT_LINE('It is November');
   ELSE
      DBMS_OUTPUT.PUT_LINE('It is not November');
   END IF;
END;
```

Use EXTRACT when you need to use a datetime element to control program flow, as in this example, or when you need a datetime element as a numeric value.

Datetime Arithmetic

Datetime arithmetic in an Oracle database can be reduced to the following types of operations:

- Adding or subtracting an interval to or from a datetime value
- Subtracting one datetime value from another in order to determine the interval between the two values
- Adding or subtracting one interval to or from another interval
- Multiplying or dividing an interval by a numeric value

For historical reasons, because of the way in which the database has been developed over the years, I draw a distinction between datetime arithmetic involving the DATE type and that involving the family of TIMESTAMP and INTERVAL types.

Date Arithmetic with Intervals and Datetimes

Arithmetic with day-to-second intervals is easy when you're working with the TIMESTAMP family of datatypes. Simply create an INTERVAL DAY TO SECOND value and add or subtract it. For example, to add 1,500 days, 4 hours, 30 minutes, and 2 seconds to the current date and time:

```
DECLARE
   current_date TIMESTAMP;
   result_date TIMESTAMP;
BEGIN
   current_date := SYSTIMESTAMP;
   result_date:= current_date + INTERVAL '1500 4:30:2' DAY TO SECOND;
   DBMS_OUTPUT.PUT_LINE(result_date);
END;
```

Date arithmetic with year and month values is not quite as straightforward. All days can be measured as 24 hours or 1,440 minutes or even 86,400 seconds, but not all months have the same number of days. A month may have 28, 29, 30, or 31 days. (I'll ignore the goofy month when the Gregorian calendar was adopted.) Because of this disparity in the number of days in a month, simply adding one month to a date can lead to an ambiguous resulting date. If you want to add one month to the last day of May, should you get the last day of June or the invalid value 31 June? Well, it all depends on what you need the dates or intervals to represent.

The Oracle database gives you the toolkit to build either result into your programs. You —the intelligent, clever developer—get to decide which behavior your system should implement. If you want an end of month to translate into an end of month (31 May + 1 month = 30 June), use the function ADD_MONTHS. If you do not want the database to alter day-of-month values, use an INTERVAL YEAR TO MONTH value. Thus 31May2008 + INTERVAL '1' MONTH will result in 31Jun2008, causing the database to throw an *ORA-01839: date not valid for month specified* error.

Date arithmetic using INTERVAL YEAR TO MONTH values is best reserved for those datetimes that are kept truncated to the beginning of a month, or perhaps to the 15th of the month—it is not appropriate for end-of-month values. If you need to add or subtract a number of months (and also years—you have the same end-of-month problem if you add one year to 29Feb2008) from a datetime that may include end-of-month values, look instead to the function ADD_MONTHS. This function, which returns a DATE datatype, will handle the end-of-month disparity by converting the resultant dates to the last day of the month instead of throwing an error. For example, ADD_MONTHS('31-May-2008', 1) will return 30-Jun-2008. The resulting DATE will not have a time zone (or subsecond granularity), so if you need these components in your result, you will need to code some extra logic to extract and reapply these components to the computed results. The code for this example looks like this:

```
DECLARE
   end_of_may2008 TIMESTAMP;
   next_month TIMESTAMP;
BEGIN
   end_of_may2008 := TO_TIMESTAMP('31-May-2008', 'DD-Mon-YYYY');
   next_month := TO_TIMESTAMP(ADD_MONTHS(end_of_may2008, 1));
   DBMS_OUTPUT.PUT_LINE(next_month);
END;
```

The results are:

```
30-Jun-2008 00:00:00.000000
```

There is no SUBTRACT_MONTHS function, but you can call ADD_MONTHS with negative month values. For example, use ADD_MONTHS(current_date, −1) in the previous example to go back one month to the last day of April.

Date Arithmetic with DATE Datatypes

Date arithmetic with DATE datatypes can use INTERVAL values, or it can use numeric values representing days and fractions thereof. For example, to add one day to the current date and time, specify:

```
SYSDATE + 1
```

And to add four hours to the current date and time:

```
SYSDATE + (4/24)
```

Notice here my use of 4/24 rather than 1/6. I use this approach to make it plain that I am adding four hours to the value returned by SYSDATE. I could use 1/6, but then the next person to maintain the code would have to figure out what was intended by 1/6. By using 4/24, I make my intent of adding four hours more explicit. Even more explicitly, I can use a meaningfully named constant like this:

```
DECLARE
   four_hours NUMBER := 4/24;
BEGIN
   DBMS_OUTPUT.PUT_LINE(
      'Now + 4 hours =' || TO_CHAR (SYSDATE + four_hours));
END;
```

Table 10-4 shows the fractional values that you can use to represent hours, minutes, and seconds when working with DATEs. Table 10-4 also shows some easily understandable expressions that you can use to build those values, in case you prefer to use, say, 60/24/60 instead of 60/1440 to mean 60 minutes.

Table 10-4. Fractional values in date arithmetic

Value	Expression	Represents
1/24	1/24	One hour
1/1,440	1/24/60	One minute
1/86,400	1/24/60/60	One second

Use the values in Table 10-4 consistently, and your code will be easier to understand. Once you learn three denominators, it becomes trivial to recognize that 40/86,400 means 40 seconds. It's not so easy, though, to recognize that 1/21,610 means the same thing.

Computing the Interval Between Two Datetimes

You can compute the interval between two TIMESTAMP family values by simply subtracting one value from the other. The result will always be of type INTERVAL DAY TO SECOND. For example:

```
DECLARE
   leave_on_trip TIMESTAMP := TIMESTAMP '2005-03-22 06:11:00.00';
   return_from_trip TIMESTAMP := TIMESTAMP '2005-03-25 15:50:00.00';
   trip_length INTERVAL DAY TO SECOND;
BEGIN
   trip_length := return_from_trip - leave_on_trip;

   DBMS_OUTPUT.PUT_LINE('Length in days hours:minutes:seconds
  is ' || trip_length);
END;
```

The output is:

```
Length in days hours:minutes:seconds is +03 09:39:00.000000
```

Intervals can be negative or positive. A negative interval indicates that you've subtracted a more recent date from a date further in the past, as in:

```
18-Jun-1961 - 15-Nov-1961 = −150
```

Fundamentally, the sign of the result indicates the directionality of the interval. It's somewhat unfortunate that there is no absolute value function that applies to intervals in the same way that the ABS function applies to numeric values.

If you compute the interval between two DATE values, the result is a number representing how many 24-hour periods (not quite the same as days) are between the two values. If the number is an integer, then the difference is an exact number of days. If the number is a fractional number, then the difference includes some number of hours, minutes, and seconds as well. For example, here is the same computation as the one I specified previously, but this time using DATEs:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE (
      TO_DATE('25-Mar-2005 3:50 pm','dd-Mon-yyyy hh:mi am')
      - TO_DATE('22-Mar-2005 6:11 am','dd-Mon-yyyy hh:mi am')
   );
END;
```

The output is:

```
3.40208333333333333333333333333333333333
```

The three days you can understand, but you're probably wondering what exactly is represented by 0.40208333333333333333333333333333333333. Often the dates are TRUNCed before being subtracted, or the resulting number is truncated. Correctly translating a long decimal string into hours, minutes, and seconds is much easier using the INTERVAL and TIMESTAMP types.

Also useful for computing intervals between two DATEs is the MONTHS_BETWEEN function. The function's syntax is:

```
FUNCTION MONTHS_BETWEEN (date1 IN DATE, date2 IN DATE)
   RETURN NUMBER
```

The following rules apply:

- If *date1* comes after *date2*, MONTHS_BETWEEN returns a positive number.
- If *date1* comes before *date2*, MONTHS_BETWEEN returns a negative number.
- If *date1* and *date2* are in the same month, MONTHS_BETWEEN returns a fraction (a value between −1 and +1).
- If *date1* and *date2* both fall on the last day of their respective months, MONTHS_BETWEEN returns a whole number (no fractional component).

- If *date1* and *date2* are in different months, and at least one of the dates is not the last day of the month, MONTHS_BETWEEN returns a fractional number. The fractional component is calculated on a 31-day-month basis and also takes into account any differences in the time component of *date1* and *date2*.

Here are some examples of uses of MONTHS_BETWEEN:

```
BEGIN
   -- Calculate two ends of month, the first earlier than the second:
   DBMS_OUTPUT.PUT_LINE(
      MONTHS_BETWEEN ('31-JAN-1994', '28-FEB-1994'));

   -- Calculate two ends of month, the first later than the second:
   DBMS_OUTPUT.PUT_LINE(
      MONTHS_BETWEEN ('31-MAR-1995', '28-FEB-1994'));

   -- Calculate when both dates fall in the same month:
   DBMS_OUTPUT.PUT_LINE(
      MONTHS_BETWEEN ('28-FEB-1994', '15-FEB-1994'));

   -- Perform months_between calculations with a fractional component:
   DBMS_OUTPUT.PUT_LINE(
      MONTHS_BETWEEN ('31-JAN-1994', '1-MAR-1994'));
   DBMS_OUTPUT.PUT_LINE(
      MONTHS_BETWEEN ('31-JAN-1994', '2-MAR-1994'));
   DBMS_OUTPUT.PUT_LINE(
      MONTHS_BETWEEN ('31-JAN-1994', '10-MAR-1994'));
END;
```

The output is:

```
−1
13
.41935483870967741935483870967741935483870967741935483870967741935483887
```

```
-1
13
.4193548387096774193548387096774193548387
−1.03225806451612903225806451612903225806
−1.0645161290322580645161290322580645161613
−1.32258064516129032258064516129032258065
```

If you think you detect a pattern here, you are right. As noted, MONTHS_BETWEEN calculates the fractional component of the number of months by assuming that each month has 31 days. Therefore, each additional day over a complete month counts for 1/31 of a month, and:

```
1 divided by 31 = .032258065 -- more or less!
```

According to this rule, the number of months between January 31, 1994, and February 28, 1994, is 1—a nice, clean integer. But the number of months between January 31, 1994, and March 1, 1994, has an additional .032258065 added to it. As with DATE subtraction, the TRUNC function is often used with MONTHS_BETWEEN.

Mixing DATEs and TIMESTAMPs

The result of a subtraction involving two TIMESTAMPs is a value of type INTERVAL DAY TO SECOND. The result of a subtraction involving two DATEs is a numeric value. Consequently, if you want to subtract one DATE from another and return an INTERVAL DAY TO SECOND value, you will need to CAST your DATEs into TIMESTAMPs. For example:

```
DECLARE
   dt1 DATE;
   dt2 DATE;
   d2s INTERVAL DAY(3) TO SECOND(0);
BEGIN
   dt1 := TO_DATE('15-Nov-1961 12:01 am','dd-Mon-yyyy hh:mi am');
   dt2 := TO_DATE('18-Jun-1961 11:59 pm','dd-Mon-yyyy hh:mi am');

   d2s := CAST(dt1 AS TIMESTAMP) - CAST(dt2 AS TIMESTAMP);

   DBMS_OUTPUT.PUT_LINE(d2s);
END;
```

The output is:

```
+149 00:02:00
```

If you mix DATEs and TIMESTAMPs in the same subtraction expression, PL/SQL will implicitly cast the DATEs into TIMESTAMPs. For example:

```
DECLARE
   dt DATE;
   ts TIMESTAMP;
   d2s1 INTERVAL DAY(3) TO SECOND(0);
   d2s2 INTERVAL DAY(3) TO SECOND(0);
BEGIN
   dt := TO_DATE('15-Nov-1961 12:01 am','dd-Mon-yyyy hh:mi am');
   ts := TO_TIMESTAMP('18-Jun-1961 11:59 pm','dd-Mon-yyyy hh:mi am');

   d2s1 := dt - ts;
   d2s2 := ts - dt;

   DBMS_OUTPUT.PUT_LINE(d2s1);
   DBMS_OUTPUT.PUT_LINE(d2s2);
END;
```

The output is:

```
+149 00:02:00
-149 00:02:00
```

As with all datetime datatypes, it's best to use explicit casting and not rely on implicit datatype conversions.

Adding and Subtracting Intervals

Unlike the case with datetime values, it makes perfect sense to add one interval to another. It also makes sense to subtract one interval from another. The one rule you need to keep in mind is that whenever you add or subtract two intervals, they must be of the same type. For example:

```
DECLARE
   dts1 INTERVAL DAY TO SECOND := '2 3:4:5.6';
   dts2 INTERVAL DAY TO SECOND := '1 1:1:1.1';

   ytm1 INTERVAL YEAR TO MONTH := '2-10';
   ytm2 INTERVAL YEAR TO MONTH := '1-1';

   days1 NUMBER := 3;
   days2 NUMBER := 1;
BEGIN
   DBMS_OUTPUT.PUT_LINE(dts1 - dts2);
   DBMS_OUTPUT.PUT_LINE(ytm1 - ytm2);
   DBMS_OUTPUT.PUT_LINE(days1 - days2);
END;
```

The output is:

```
+000000001 02:03:04.500000000
+000000001-09
2
```

This example shows the results of three interval subtractions. The first two involve INTERVAL DAY TO SECOND and INTERVAL YEAR TO MONTH. The third shows the subtraction of two numbers. Remember: when you're working with DATE types, the interval between two DATE values is expressed as a NUMBER. Because months can have 28, 29, 30, or 31 days, if you add or subtract a day-to-second interval to or from a year-to-month interval, the database will raise an *ORA-30081: invalid datatype for datetime/interval arithmetic* exception.

Multiplying and Dividing Intervals

Multiplication and division have no application to dates, but you can multiply an interval by a number and divide an interval by a number. Here are some examples:

```
DECLARE
   dts1 INTERVAL DAY TO SECOND := '2 3:4:5.6';
   dts2 INTERVAL YEAR TO MONTH := '2-10';
   dts3 NUMBER := 3;
BEGIN
   -- Show some interval multiplication
   DBMS_OUTPUT.PUT_LINE(dts1 * 2);
   DBMS_OUTPUT.PUT_LINE(dts2 * 2);
   DBMS_OUTPUT.PUT_LINE(dts3 * 2);
   -- Show some interval division
   DBMS_OUTPUT.PUT_LINE(dts1 / 2);
   DBMS_OUTPUT.PUT_LINE(dts2 / 2);
   DBMS_OUTPUT.PUT_LINE(dts3 / 2);
END;
```

The output is:

```
+000000004 06:08:11.200000000
+000000005-08
6
+000000001 01:32:02.800000000
+000000001-05
1.5
```

Using Unconstrained INTERVAL Types

Intervals can be declared with varying levels of precision, and values of different precisions are not entirely compatible with one another. This becomes especially problematic when you are writing procedures and functions that accept INTERVAL values as parameters. The following example should help you to visualize the problem. Notice the loss of precision when the value of dts is doubled via a call to the function double_my_interval:

```
DECLARE
   dts INTERVAL DAY(9) TO SECOND(9);

   FUNCTION double_my_interval (
      dts_in IN INTERVAL DAY TO SECOND) RETURN INTERVAL DAY TO SECOND
   IS
   BEGIN
      RETURN dts_in * 2;
   END;
BEGIN
   dts := '1 0:0:0.123456789';
   DBMS_OUTPUT.PUT_LINE(dts);
   DBMS_OUTPUT.PUT_LINE(double_my_interval(dts));
END;
```

The output is:

```
+000000001 00:00:00.123456789
+02 00:00:00.246914
```

Not only have I lost digits in my fractional seconds, but I've also lost digits where the number of days is concerned. Had dts been assigned a value of 100 days or more, the call to double_my_interval would have failed with an *ORA-01873: the leading precision of the interval is too small* error.

The issue here is that the default precision for INTERVAL types is not the same as the maximum precision. Usually, the calling program supplies the precision for parameters to a PL/SQL program, but with INTERVAL datatypes, the default precision of 2 is used. To work around this problem, I can use an explicitly unconstrained INTERVAL datatype:

YMINTERVAL_UNCONSTRAINED
: Accepts any INTERVAL YEAR TO MONTH value with no loss of precision

DSINTERVAL_UNCONSTRAINED
: Accepts any INTERVAL DAY TO SECOND value with no loss of precision

Using the DSINTERVAL_UNCONSTRAINED type, I can recode my earlier example as follows:

```
DECLARE
   dts INTERVAL DAY(9) TO SECOND(9);

   FUNCTION double_my_interval (
      dts_in IN DSINTERVAL_UNCONSTRAINED) RETURN DSINTERVAL_UNCONSTRAINED
   IS
   BEGIN
      RETURN dts_in * 2;
   END;
BEGIN
   dts := '100 0:0:0.123456789';
   DBMS_OUTPUT.PUT_LINE(dts);
   DBMS_OUTPUT.PUT_LINE(double_my_interval(dts));
END;
```

The output is:

```
+000000100 00:00:00.123456789
+000000200 00:00:00.246913578
```

Notice that I used DSINTERVAL_UNCONSTRAINED twice: once to specify the type of the formal parameter to double_my_interval, and once to specify the function's return type. As a result, I can now invoke the function on *any* INTERVAL DAY TO SECOND value with no loss of precision or ORA-01873 errors.

Date/Time Function Quick Reference

Oracle implements a number of functions that are useful for working with datetime values. You've seen many of them used earlier in this chapter. I don't document them all here, but I do provide a list in Table 10-5 to help you become familiar with what's available. I encourage you to refer to Oracle's *SQL Reference* manual and read up on those functions that interest you.

Avoid using Oracle's traditional date functions with the new TIMESTAMP types. Instead, use the new INTERVAL functionality whenever possible. Use date functions only with DATE values.

Many of the functions in Table 10-5 accept DATE values as inputs. ADD_MONTHS is an example of one such function. You must be careful when you consider using such functions to operate on any of the new TIMESTAMP datatypes. While you can pass a TIMESTAMP value to one of these functions, the database implicitly and silently converts that value to a DATE. Only then does the function perform its operation. For example:

```
DECLARE
   ts TIMESTAMP WITH TIME ZONE;
BEGIN
   ts := SYSTIMESTAMP;

   -- Notice that ts now specifies fractional seconds
   -- AND a time zone.
   DBMS_OUTPUT.PUT_LINE(ts);

   -- Modify ts using one of the built-in date functions.
   ts := LAST_DAY(ts);

   -- We've now LOST our fractional seconds, and the
   -- time zone has changed to our session time zone.
   DBMS_OUTPUT.PUT_LINE(ts);
END;
```

The output is:

```
13-MAR-05 04.27.23.163826 PM -08:00
31-MAR-05 04.27.23.000000 PM -05:00
```

In this example, the variable ts contained a TIMESTAMP WITH TIME ZONE value. That value was implicitly converted into a DATE when it was passed to LAST_DAY. Because DATEs hold neither fractional seconds nor time zone offsets, those parts of ts's value were silently discarded. The result of LAST_DAY was assigned back to ts, causing a second implicit conversion, this time from DATE to TIMESTAMP WITH TIME ZONE. This second conversion picked up the session time zone, and that's why you see −05:00 as the time zone offset in the final value.

This behavior is critical to understand! It's critical to avoid, too. I'm sure you can imagine the kind of subtle program errors that can be induced by careless application of DATE functions to TIMESTAMP values. Frankly, I can't imagine why Oracle did not overload the built-in DATE functions so that they also worked properly for TIMESTAMPs. Be careful!

Table 10-5. Built-in datetime functions

Name	Description
ADD_MONTHS	Returns a DATE containing the specified DATE incremented by the specified number of months. See the section "Adding and Subtracting Intervals" on page 317.
CAST	Converts between datatypes—for example, between DATE and the various TIMESTAMP datatypes. See the section "CAST and EXTRACT" on page 308.
CURRENT_DATE	Returns a DATE containing the current date and time in the session time zone.
CURRENT_TIMESTAMP	Returns a TIMESTAMP WITH TIME ZONE containing the current date and time in the session time zone.
DBTIMEZONE	Returns the time zone offset (from UTC) of the database time zone in the form of a character string (e.g., '–05:00'). The database time zone is only used with TIMESTAMP WITH LOCAL TIME ZONE datatypes.
EXTRACT	Returns a NUMBER or VARCHAR2 containing the specific datetime element, such as hour, year, or timezone_abbr. See the section "CAST and EXTRACT" on page 308.
FROM_TZ	Converts a TIMESTAMP and time zone to a TIMESTAMP WITH TIME ZONE.
LAST_DAY	Returns a DATE containing the last day of the month for the specified DATE.
LOCALTIMESTAMP	Returns the current date and time as a TIMESTAMP value in the local time zone.
MONTHS_ BETWEEN	Returns a NUMBER containing the number of months between two DATEs. See the section "Computing the Interval Between Two Datetimes" on page 313 for an example.
NEW_TIME	Shifts a DATE value from one time zone to another. This functionality exists to support legacy code. For any new applications, use the TIMESTAMP WITH TIME ZONE or TIMESTAMP WITH LOCAL TIME ZONE types.
NEXT_DAY	Returns the DATE of the first weekday specified that is later than a specified DATE.
NUMTODSINTERVAL	Converts a number of days, hours, minutes, or seconds (your choice) to a value of type INTERVAL DAY TO SECOND.
NUMTOYMINTERAL	Converts a number of years or months (your choice) to a value of type INTERVAL YEAR TO MONTH.
ROUND	Returns a DATE rounded to a specified level of granularity.
SESSIONTIMEZONE	Returns a VARCHAR2 containing the time zone offset (from UTC) of the session time zone in the form of a character string (e.g., '–05:00').
SYS_EXTRACT_UTC	Converts a TIMESTAMP WITH TIME ZONE value to a TIMESTAMP having the same date and time, but normalized to UTC.
SYSDATE	Returns the current date and time from the database server as a DATE value.
SYSTIMESTAMP	Returns the current date and time from the database server as a TIMESTAMP WITH TIME ZONE value.
TO_CHAR	Converts datetime values to their character string representations. See the section "Datetime Conversions" on page 289.
TO_DATE	Converts a character string to a value of type DATE. See the section "Datetime Conversions" on page 289.
TO_DSINTERVAL	Converts a character string to a value of INTERVAL DAY TO SECOND. See the section "Interval Conversions" on page 304.
TO_TIMESTAMP	Converts a character string to a value of type TIMESTAMP. See the section "Datetime Conversions" on page 289.
TO_TIMESTAMP_TZ	Converts a character string to a value of type TIMESTAMP WITH TIME ZONE. See the section "Datetime Conversions" on page 289.

Name	Description
TO_YMINTERVAL	Converts a character string to a value of INTERVAL YEAR TO MONTH. See the section "Interval Conversions" on page 304.
TRUNC	Truncates a DATE or TIMESTAMP value to a specified level of granularity, returning a DATE datatype.
TZ_OFFSET	Returns a VARCHAR2 containing the time zone offset from UTC (e.g., '–05:00') for a given time zone name, abbreviation, or offset.