



Chapter 3

Functions

As implemented by Oracle, SQL contains more than 80 functions that each manipulate data values and return results. These functions provide a rich opportunity for creative query writing. This chapter examines some of the ways that you can use functions individually or in combination to achieve special effects with your queries.

Functions come in two varieties based on how many rows they process. As the name implies, single-row functions operate on data one row at a time. An example is the `UPPER` function that converts character data to uppercase, doing so for a specified data column in each row of the query result set. Conversely, group functions operate on multiple rows grouped according to some criterion. They return a characteristic of the entire group. Examples include functions that compute averages, sums, counts, and variances.

You generally see single-row functions more finely categorized by the datatype accepted in the function argument. Number functions process `NUMBER` datatypes, date functions process `DATE` datatypes, and so on. When classified like this, there are five major types of single-row functions: number, character, date, conversion, and miscellaneous. Conversion functions convert data values of one datatype to values of another datatype, often using optional formats. Miscellaneous functions include those functions which cannot be categorized elsewhere.

Query writers most frequently ask questions about two types of SQL functions, with a miscellaneous set of remaining questions. The two principal query concerns are `DATE` functions and the `DECODE` function. Both receive considerable attention in this chapter.

Dates

If sheer quantity of questions indicates an area of uncertainty among query writers, then dates and date arithmetic evidently pose considerable query problems. Many of the questions that appear in this section arise repeatedly among query writers.

How do I determine if a date occurred on a Monday?

I'm writing a query to select events that occurred on Mondays. I know that some of the events did occur on a Monday, yet my query returns no rows. The `WHERE` condition in my query uses the `TO_CHAR` function to convert the event dates to the day of the week; i.e., `TO_CHAR(event_date, 'Day') = 'Monday'`. It seems real simple. What's going on here?

Approach

This is one of those “gotcha” problems that torment query writers. Just when you think you understand query writing, a seemingly illogical element in SQL or SQL*PLUS pops up to humble you. Maybe that's good.

When the `TO_CHAR` function converts dates using a 'Day' format, it right-pads the name of the day to a length of 9. Thus the condition in the question above fails because the `TO_CHAR` function returns Monday right-padded with three blank spaces. Any of the following conditions will work:

```
TO_CHAR(event_date, 'Day') = 'Monday   '
RTRIM(TO_CHAR(event_date, 'Day'),' ') = 'Monday'
TO_CHAR(event_date, 'FMDay') = 'Monday'
```

In the first case, the literal (i.e., Monday) is right-padded with three spaces; in the second case, the `RTRIM` function trims trailing blank spaces; in the third case, the `FM` (Fill Mode) format modifier suppresses blank spaces returned to the `TO_CHAR` function.

See Also

1. The `DAY` format model can also be used to achieve other effects. For example, if you want to recode days into weeks you can use the `DAY` format with the `TRUNC` function. This collapses all days during a week into the first day of the week. See page 84 in this chapter or Chapter 10 on page 298 for a discussion of this recoding technique.

How do I create begin-end date ranges from a series of begin dates?

Our training classes run monthly beginning the first day of each month. The table of class information includes a begin date but no end date. I'd like to write a query that displays both a begin and an end date for the training classes. Is there an easy way to do this?

Approach

The `ADD_MONTHS` function allows you to add or subtract a specified number of months from a date. With just the slightest tweak, you can create monthly date ranges using this function. Figure 3-1 illustrates how this is done.

```
SELECT
  training_code "class",
  training_start_date "begin",
  ADD_MONTHS(training_start_date,1) -1 "end"
FROM
  training
ORDER BY 2;
```

class	begin	end
A101	01-FEB-96	29-FEB-96
A100	01-JAN-97	31-JAN-97
R950	01-APR-97	30-APR-97

Figure 3-1: Creating monthly date ranges with `add_months`.

Note that adding a month normally yields the same day one month later (e.g., adding one month to `15-APR-97` returns `15-MAY-97`). However, if the begin date is near the end of a month and there is no comparable date the next month, the last day of the next month is returned (e.g., adding a month to `30-JAN-97` yields `28-FEB-97` except in a leap year).

Comments

► The general form of the `ADD_MONTHS` function is `ADD_MONTHS(date, n)`, where `n` is any integer. In Figure 3-1, subtracting 1 in the end date expression produces the correct date as the last day of the month.

See Also

1. The `ADD_MONTHS` function is one of many date functions. Check the index under *date functions* for examples of other date functions.

How do I calculate age based on SYSDATE and a birth date?

I've written a query that computes a person's age using the MONTHS_BETWEEN function and the following algorithm:

```
FLOOR(MONTHS_BETWEEN(SYSDATE, person_birthdate)/12)
```

Will this formula produce accurate ages in years?

Approach

The algorithm works fine for computing years between two dates. People normally report their age in years by truncating the months and days, so someone is 10 years old until the next birthday that makes him 11. The FLOOR function in the formula corresponds to this common convention. You also could use TRUNC.

The MONTHS_BETWEEN function always returns integers for two dates having the same day of the month (e.g., 11-SEP-97 and 11-AUG-97 are 1 month apart). It also produces integers for two dates that are both the end of the month (e.g., 30-APR-97 and 31-MAR-97). Fractional months are based on a 31-day month. Thus you can get some strange answers when examining results in terms of days. For example, 1996 was a leap year. The MONTHS_BETWEEN 31-MAR-96 and 28-FEB-96 is 1.09677419; that is, there is 1 month between 28-MAR-96 and 28-FEB-96, plus another $0.09677419 * 31 = 3$ days. The days component is wrong, of course, but then this is not the purpose of the function. Used to compute months or larger units like quarters or years, the MONTHS_BETWEEN function will provide accurate results.

See Also

1. FLOOR(arg) is a number function that returns the largest integer equal to or less than arg (e.g., floor(4.2) returns 4). For another example of its use, see Figure 3-4 on page 82.

How do I specify a date value in the year 2000 or later?

I've written several queries that prompt users for dates in a DD-MON-YY date format. Will I have problems with these queries when the next century starts?

Approach

Data values stored with DATE datatypes include the century as well as the year, month, day, hour, minute, and second. Thus the year 2000 won't present any insurmountable problems. However, depending on how your queries are constructed, they may return inaccurate results.

Figure 3-2 includes two training classes that both begin at midnight on 01-APR, but one in the year 1900 and the second in the year 2000. The query prompts the user for a begin date. Suppose you want the row for the training class that begins in the year 2000. The results depend on how the query is constructed.

```

class   begin                end
-----
X20a    01-APR-1900          01-APR-1900
R111    01-APR-2000          01-APR-2000

SELECT * FROM seminars
WHERE TO_CHAR(seminar_start_date, 'dd-MON-yyyy') = '&date';
Enter value for date: 01-APR-2000
class   begin                end
-----
R111    01-APR-2000          01-APR-2000

SELECT * FROM seminars
WHERE seminar_start_date = '&date';
Enter value for date: 01-APR-00
class   begin                end
-----
X20a    01-APR-1900          01-APR-1900

SELECT * FROM seminars
WHERE seminar_start_date = to_date('&date', 'dd-MON-rr');
Enter value for date: 01-APR-00
class   begin                end
-----
R111    01-APR-2000          01-APR-2000

SELECT * FROM seminars
WHERE TO_CHAR(seminar_start_date, 'dd-MON-rr') = '&date';
Enter value for date: 01-APR-00
class   begin                end
-----
X20a    01-APR-1900          01-APR-1900
R111    01-APR-2000          01-APR-2000
    
```

Figure 3-2: Specifying dates after the year 2000.

Comments

- ▶ The data includes two seminars that begin on 01-APR. One begins in the year 1900 and the second in the year 2000.
- ▶ If you use a TO_CHAR function to format the date column with a four-digit year (i.e., 'YYYY'), then the query will always return accurate results. This means prompting users with the new date format, however.
- ▶ If the query is constructed in its simplest form with no conversion functions like TO_CHAR or TO_DATE and the user specifies a date with a two-digit year, then the query returns rows for dates in the current century (assuming that NLS_DATE_FORMAT is 'DD-MON-YY' or some variation that uses a two-digit 'YY' year format). In Figure 3-2, the training class in 1900 was returned because the query was run in 1996. This query is obviously wrong if you want the class that begins in 2000. You would experience a similar problem after the year 2000 if the user specified a two-digit year and the training class you want is in the 1900s.
- ▶ The two-digit 'RR' year format solves several problems associated with dates in different centuries. How it behaves depends on SYSDATE. If the present year ends in 00 to 49 and the user specifies a two-digit year between 00 and 49, the present century is implied. The same holds true if the present year ends in 50 to 99 and the user specifies a two-digit year within this range. Thus, if the year is 1996 and the user specifies a date of 12-DEC-99 with a 'DD-MON-RR' format, it is assumed the user means 1999.

However, if the present year and the two-digit year specified by the user fall within different half-century categories, different centuries are implied. For example, assume that the present year is 1997. Specifying a date as 25-AUG-02 implies the twenty-first century. Similarly, if the present year is 2005, specifying a date of 19-SEP-92 implies the twentieth century.

A query constructed with the TO_DATE function using an 'RR' year format will return the correct results most of the time. It will produce inaccurate results whenever SYSDATE and the date you want are in different centuries and more than 50 years apart. For example, assume that the SYSDATE year is 2005 and the user specifies a date as 10-OCT-23, where she means the year 1923. The query will search for dates in the year 2023 and return the wrong rows.

Note that if your database administrator sets the parameter `NLS_DATE_FORMAT` as `'DD-MON-RR'`, you won't need to use the `TO_DATE` function to get the same results as shown in this query in Figure 3-2.

► The two-digit `'RR'` year format does not behave differently from the `'YY'` format when used in the `TO_CHAR` function. In either case, the query returns the wrong results.

The long and the short of all this discussion about dates and the year 2000 is that you're better off revising the queries so that they require the user to specify a four-digit year. Then there is no ambiguity. However, if your queries are unlikely to ever confront the date problems possible with the two-digit `'RR'` year format, then encourage your database administrator to change the `NLS_DATE_FORMAT` to use this format.

See Also

1. The `NLS` in `NLS_DATE_FORMAT` means National Language Support. Throughout the world, common conventions for the display of dates vary. The parameters that control the default display of dates can be found in the `SYS` view `NLS_SESSION_PARAMETERS`.
2. In Chapter 2, the utility program that computes minimum linesize needed for a report used the `NLS_DATE_FORMAT` to determine the space required for date columns. See Chapter 2 on page 55 for a discussion of this.
3. See the discussion later in this chapter (page 84) on `NLS_TERRITORY` and the definitions which identify the day a week begins (e.g., in the `AMERICA` territory a week begins on Sunday).

How do I return all rows where a column equals a specified date?

I'm writing a query to retrieve all training classes that begin on a specific date. The WHERE conditions seem simple enough, but some rows fall through the query and don't display. Why does this happen?

Approach

Data columns with a DATE datatype have both a date and time component. If your WHERE clause specifies only a date, the time component defaults to 00:00:00. Thus you may think a query reads

```
WHERE begin_date = '20-MAR-97'
```

In fact, however, the condition actually being executed retrieves only those rows where the begin_date is midnight on 20-MAR-97.

You can eliminate this problem in several ways. Figure 3-2 shows two. One uses the TO_CHAR function to format the begin_date; the second uses relational operators. You could also use the LIKE operator to

class	begin	end
BX1a	15-JAN-1997 09:30:00	15-JAN-1997 11:45:00
T1A	15-JAN-1997 00:00:00	16-JAN-1997 00:00:00
RT12c	10-FEB-1997 14:30:00	13-FEB-1997 17:00:00
Y1C	31-DEC-1999 22:15:30	01-JAN-2000 08:05:50


```
SELECT * FROM seminars
WHERE seminar_start_date = '15-JAN-97';
```

class	begin	end
T1A	15-JAN-97	16-JAN-97


```
SELECT * FROM seminars WHERE
TO_CHAR(seminar_start_date, 'dd-MON-yy') = '15-JAN-97';
```

class	begin	end
BX1a	15-JAN-97	15-JAN-97
T1A	15-JAN-97	16-JAN-97


```
SELECT * FROM seminars
WHERE seminar_start_date >= '15-JAN-97'
AND seminar_start_date < '16-JAN-97';
```

ENTER to continue ...

class	begin	end
BX1a	15-JAN-97	15-JAN-97
T1A	15-JAN-97	16-JAN-97

Figure 3-3: Two methods for specifying dates in WHERE conditions.

retrieve rows WHERE seminar_start_date LIKE '15-JAN-97%'. Or you could use the TRUNC function: WHERE TRUNC(seminar_start_date) = '15-JAN-97'.

Comments

- ▶ Two seminar classes begin on 15-JAN-97 but only one of them at midnight.
- ▶ The query that seems logically correct only returns the class that begins at midnight.
- ▶ The query works fine if you use TO_CHAR to format the begin date so that it excludes the time component.
- ▶ Creating a date range with relational operators also works. In this case, the query includes all courses that begin at or later than midnight on 15-JAN-97 and end before midnight on 16-JAN-97. Both courses fall within this range.

See Also

1. Having alternative methods of solving a problem is extremely valuable. Query writers can exercise a range of personal preference and still get out of trouble when problems occur in a query. Of course, not all solutions are equal. Elegance and performance impact are two criteria that should be used to determine which method to employ for a query problem. Later chapters stress the importance of alternative solutions to problems. See, for example, Chapter 7 that discusses several methods for testing and debugging join problems.
2. See page 82 for an example where the TRUNC function is used with dates.

How do I compute duration based on begin and end dates?

I'm trying to write a query to show the duration of training classes in hours and minutes, yet I can't locate an SQL function that provides this information based on starting and end dates. How can I compute duration myself?

Approach

When you subtract two dates, the result is a real number expressed in days (e.g., 1.2 days). The trick lies in formatting the computation to provide duration in more meaningful units than a decimal number of days. Figure 3-4 shows begin and end dates for three hypothetical training classes (including one that runs on New Year's eve 1999). The output formats the duration of the classes in days, hours, minutes, and seconds.

class	begin	end
BX1a	15-JAN-1997 09:30:00	15-JAN-1997 11:45:00
Y1C	31-DEC-1999 22:15:30	01-JAN-2000 08:05:50
RT12c	10-FEB-1997 14:30:00	13-FEB-1997 17:00:00

```

COLUMN days FORMAT 9999
COLUMN hhmiss HEADING 'hh24:mi:ss' FORMAT a10
SELECT
  1 FLOOR(seminar_end_date - seminar_start_date) "days",
    TO_CHAR((seminar_end_date - seminar_start_date) +
      TRUNC(SYSDATE), 'hh24:mi:ss') "hhmiss"
FROM
  seminars;
2
  days hh24:mi:ss
-----
  0 02:15:00
  0 09:50:20
  3 02:30:00
3
```

Figure 3-4: Calculation of the duration of an event.

Comments

- ▶ The **FLOOR** function returns the greatest integer equal to or less than the function's argument. Thus, if the duration is 1.2 days, **FLOOR(1.2)** returns 1. This provides the days between start and end dates.
- ▶ The **TRUNC** function truncates a date to a specified unit (e.g., 'MON' truncates to the first day in the nearest month). If the truncating unit is not specified (as here), the truncation occurs to the nearest day. This makes the time component of the truncated

date 00:00:00. The calculation then adds the duration of the training class to the truncated date, producing a new date that is formatted to display only the hours, minutes, and seconds. For example, suppose that `TRUNC(SYSDATE)` produces `07-APR-97 00:00:00` and that the duration is 1.2 days. Adding the two elements produces the new date `08-APR-97 04:48:00`. Formatting the new date with `'HH24:MI:SS'` completes the calculation.

▶ Note that the output displays duration as the number of days, hours, minutes, and seconds.

Having gone through the exercise of computing duration, you don't want to reinvent the wheel for each query that requires a similar calculation. Save the expressions as a chunk of SQL (a *segment*, as discussed in Chapter 1) that you can reuse. Generalize the segment by including substitution variables for the begin and end dates. Store this code fragment in a subdirectory devoted to SQL segments. Update your segment inventory (e.g., a list or an Oracle table devoted to the management of SQL segments you've created). Then you can easily retrieve the SQL segment and insert it into your next query that computes event durations.

See Also

1. Chapter 1 on page 15 discusses the usefulness of SQL segments in simplifying the query writer's job. Over time you'll gather an impressive collection of these code fragments, each of which performs a specific task. When that happens, query writing becomes less like programming and more like writing, where you search for the right word or phrase to use.

How do I get counts by week?

I'd like to count the number of sales by week based on a sale date, but I'm having trouble grouping the sales into weeks. How do I group date columns by week?

Approach

Several DATE functions provide ways to group dates by week. Choosing the right function depends on how you want the dates grouped. The NLS_TERRITORY parameter implicitly determines the start day of a week. For example, weeks in the AMERICA territory start by definition on Sunday and run through Saturday. Thus one way to group dates is by the week defined by NLS_TERRITORY. However, you also might choose to group dates by a 7-day period that extends across a Sunday-to-Saturday week. For example, suppose you wanted to count the sales that occurred in the 7-day period ending on a Friday. This defines your week as beginning on Saturday and running through Friday.

Let's examine functions that group dates into both NLS_TERRITORY and user-defined weeks. Figure 3-5 shows sales data for five dates. The figure shows one method to group these sales so that the display date is the Friday in a week defined by NLS_TERRITORY. It also illustrates how to group sales into a week extending from Saturday through Friday.

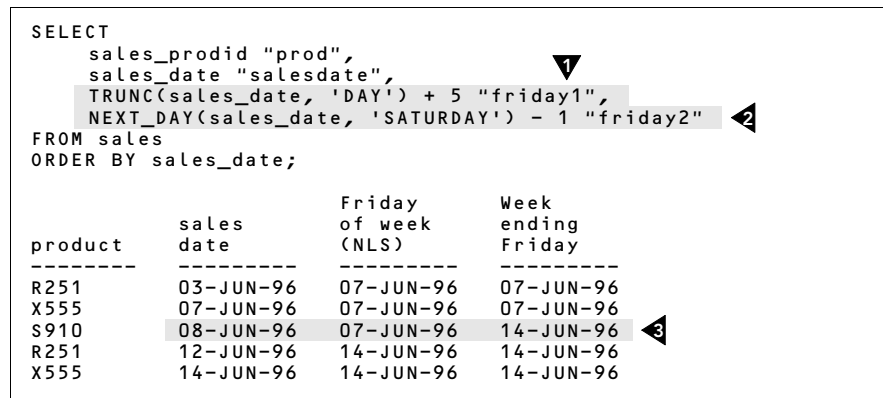


Figure 3-5: Two methods to group dates by week.

Comments

▶ Truncating a date using the 'DAY' format yields the first day in a week as defined by NLS_TERRITORY. In this case, the first day

of the week is Sunday, so adding 5 days produces Friday dates. Note that this algorithm would differ if the `NLS_TERRITORY` defined another day of the week as the start date.

▶ The `NEXT_DAY` function returns the first date that falls on the specified `DAY` and occurs after the argument date. In this case, the `NEXT_DAY` function returns the first Saturday after the sale date. If the sales date happens to occur on a Saturday, it returns the next Saturday (i.e., one week later). Subtracting 1 produces Friday dates.

▶ Note that the two grouping procedures differ in their week definitions. One uses a week extending from Sunday to Saturday; the other from Saturday to Friday. The sale that occurred on Saturday 08-JUN-96 illustrates the difference. Using the `TRUNC` procedure groups this sale with the 07-JUN-96 Friday; using the `NEXT_DAY` procedure groups this sale with the 14-JUN-96 Friday.

See Also

1. The `NLS` in `NLS_TERRITORY` means National Language Support. For related discussions, see the `SYS` view `NLS_SESSION_PARAMETERS` and `NLS_DATE_FORMAT`. The `NLS_DATE_FORMAT`, for example, featured prominently in the discussion earlier in this chapter on query problems that might arise in the year 2000. See page 77 for this discussion.

How do get a due date that is x business days after an end date?

Faculty members have 14 business days after completion of a seminar to submit final grades for students. How do I determine the due date for grades based on the seminar end date?

Approach

This is a very common question applicable to a variety of situations where one event triggers another event a specified number of business days in the future. Another example is the end of a billing period triggering a payment due date. Let's first deal with the problem without considering holidays and then include holidays as a further complication.

The term *business days* means weekdays as opposed to weekends. What constitutes a weekday varies throughout the world. For purposes of illustration, let's assume that Saturday and Sunday are weekend days. If this is not the case for you, then modify the SQL found below.

SQL Approach

Assuming 5 days in a business week, any N number of business days in the future can be defined by the number of weeks ($TRUNC(N/5)$) and the remaining number of days ($MOD(N,5)$). For example, 14 business days is 2 weeks and 4 days. What makes this question so interesting is that a due date depends on two factors: (1) the day that the trigger event occurred and (2) the MOD remaining days. The following table illustrates this.

Mod	Day that trigger event occurs							
	Sun	Mon	Tue	Wed	Thu	Fri	Sat	
Day	Day	Day	Day	Day	Day	Day	Day	Day
Chg	Chg	Chg	Chg	Chg	Chg	Chg	Chg	Chg
4	Thu 4	Fri 4	Mon 6	Tue 6	Wed 6	Thu 6	Fri 6	Thu 5
3	Wed 3	Thu 3	Fri 3	Mon 5	Tue 5	Wed 5	Thu 5	Wed 4
2	Tue 2	Wed 2	Thu 2	Fri 2	Mon 4	Tue 4	Wed 4	Tue 3
1	Mon 1	Tue 1	Wed 1	Thu 1	Fri 1	Mon 3	Tue 3	Mon 2
0	Fri -2	Mon 0	Tue 0	Wed 0	Thu 0	Fri 0	Sat 0	Fri -1

Table 3-1: Due dates based on business days and day of trigger event.

Comments

► First a word about the notation in the table. Consider the cell in the first row (MOD = 4) and the fifth column (Thursday). For example, 14 business days (MOD = 4) after a Thursday trigger date occurs on a Wednesday because of the 2 intervening

weekend days. The total number of days between Thursday and the following Wednesday is 6; this number appears in the Chg (Change) column.

► Note that the end dates occurring on weekdays are well-behaved and produce easily computed due dates but that end dates falling on weekends provide more problems. For example, 10 business days (MOD = 0) after a Sunday end date occurs on a Friday 2 weeks minus 2 days from the end date.

Any due date can be calculated with the following formula:

$$\text{end_date} + 7 * \text{TRUNC}(N/5) + \text{cell_value_from}(\text{MOD}(N,5), \text{Day})$$

where cell_value_from is a function that returns a value from Table 3-1 based on the two parameters defined by the rows and columns in the table. Figure 3-6 illustrates an SQL solution.

```

SELECT
  seminar_code "class",
  TO_CHAR(seminar_end_date, 'dd-MON-yyyy, Day') "enddate",
  TO_CHAR(
    seminar_end_date +
    TRUNC(&&days/5)*7 +
    DECODE(
      mod(&&days,5),
      4, DECODE(TO_CHAR(seminar_end_date, 'D'),1,4,2,4,7,5,6),
      3, DECODE(TO_CHAR(seminar_end_date, 'D'),1,3,2,3,3,3,7,4,5),
      2, DECODE(TO_CHAR(seminar_end_date, 'D'),5,4,6,4,7,3,2),
      1, DECODE(TO_CHAR(seminar_end_date, 'D'),6,3,7,2,1),
      0, DECODE(TO_CHAR(seminar_end_date, 'D'),1,-2,7,-1,0)
    ),
    'dd-MON-yyyy, Day') "duedate"
FROM
  seminars
ORDER BY
  TO_CHAR(seminar_end_date, 'D');

```

Enter value for days: 14

class	enddate	duedate
DCx5	16-JUN-1996, Sunday	04-JUL-1996, Thursday
AAx9	24-JUN-1996, Monday	12-JUL-1996, Friday
CD9b	04-JUN-1996, Tuesday	24-JUN-1996, Monday
BX1a	15-JAN-1997, Wednesday	04-FEB-1997, Tuesday
RT12c	13-FEB-1997, Thursday	05-MAR-1997, Wednesday
T1A	17-JAN-1997, Friday	06-FEB-1997, Thursday
R111	01-APR-2000, Saturday	20-APR-2000, Thursday

Figure 3-6: SQL calculating due dates from business days and trigger events.

Comments

► This element in the algorithm adds 7 days to the event date for each 5-day business week. For example, 14 business days is 2

weeks and 4 days. This element adds $2*7 = 14$ days to the event date to take care of the week component.

▶ The `DECODE`s consider the day component after breaking the specified number of business days into weeks and days. The highlighted `DECODE` deals with situations where there are no remainder days. Note that if the event date occurs on a Saturday or Sunday, the `DECODE` causes the due date to fall on a Friday.

PL/SQL Approach

Procedural languages deal with the problem of due dates much more elegantly than SQL. Figure 3-7 shows a PL/SQL function that computes due dates in a very direct manner.

```
CREATE OR REPLACE FUNCTION due_date
(event_date IN DATE, business_days IN NUMBER)
RETURN DATE
IS
    due_date DATE := event_date + TRUNC(business_days/5)*7;
    days_left NUMBER := MOD(business_days,5);
BEGIN
    WHILE days_left > 0
    LOOP
        due_date := due_date + 1;

        IF TO_CHAR(due_date, 'D') BETWEEN 2 AND 6 THEN
            days_left := days_left - 1;
        END IF;
    END LOOP;

    RETURN due_date;
END;
/
```

Figure 3-7: PL/SQL function that determines due dates.

Comments

▶ Any number of days can be broken into weeks and remainder days. For example, 14 business days is 2 weeks and 4 days. The weeks component gets handled when initializing `due_date`; the remainder days get handled in the `LOOP` by checking to ensure that the incremented `due_date` falls on a weekday.

Using a function to retrieve due dates also makes the query much cleaner, as demonstrated in Figure 3-8.

```

SELECT
  seminar_code "class",
  TO_CHAR(seminar_end_date, 'dd-MON-yyyy, Day') "enddate",
  TO_CHAR(due_date(seminar_end_date, 14), 'dd-MON-yyyy, Day') "duedate"
FROM seminars
ORDER BY
  TO_CHAR(seminar_end_date, 'D');

```

class	enddate	duedate
DCx5	16-JUN-1996, Sunday	04-JUL-1996, Thursday
AAx9	24-JUN-1996, Monday	12-JUL-1996, Friday
CD9b	04-JUN-1996, Tuesday	24-JUN-1996, Monday
BX1a	15-JAN-1997, Wednesday	04-FEB-1997, Tuesday
RT12c	13-FEB-1997, Thursday	05-MAR-1997, Wednesday
T1A	17-JAN-1997, Friday	06-FEB-1997, Thursday
R111	01-APR-2000, Saturday	20-APR-2000, Thursday

Figure 3-8: Query using a PL/SQL function to return due dates.

Comments

► All the DECODES in the SQL solution shown in Figure 3-6 on page 87 get replaced by a simple PL/SQL function. In this case, the due dates returned are 14 business days after the seminars end.

See Also

1. You'd like your queries to be both elegant and fast. Sometimes there's a trade-off that you must make between the two goals. Chapter 9 (Table 9-2 on page 255) shows results of performance comparisons between the SQL and PL/SQL approaches to the due date question.

How do I determine due dates and account for holidays?

Faculty members have 14 business days after a seminar ends to submit grades. How do I compute a due date for grades that also accounts for holidays?

Approach

Holidays differ from location to location even within the same country. If you store holiday dates in a table, there's a convenient variation of the PL/SQL function in Figure 3-7 on page 88 that adjusts for holidays. Figure 3-9 shows the revised algorithm.

```

CREATE OR REPLACE FUNCTION due_date1
(event_date IN DATE, business_days IN NUMBER)
RETURN DATE
IS
    due_date DATE := event_date + TRUNC(business_days/5)*7;
    days_left NUMBER := MOD(business_days,5);
    extra_days NUMBER;

BEGIN
    WHILE days_left > 0
    LOOP
        due_date := due_date + 1;

        IF TO_CHAR(due_date, 'D') BETWEEN 2 AND 6 THEN
            days_left := days_left - 1;
        END IF;
    END LOOP;

    -- determine holidays between event_date and due_date ◀1
    SELECT COUNT(*) INTO extra_days FROM holidays
    WHERE holidays_date > event_date
    AND holidays_date <= due_date
    AND TO_CHAR(holidays_date, 'D') BETWEEN 2 AND 6;

    -- recursively call if extra_days contain holidays ◀2
    IF extra_days > 0 THEN
        due_date := due_date1(due_date, extra_days);
    END IF;

    RETURN due_date;
END;
/

```

Figure 3-9: PL/SQL due date function that accounts for holidays.

Comments

▶ This function, called `xdue_date`, checks the *HOLIDAY* table to determine how many holidays occurred between the event date and the computed due date. The due date must be adjusted by

these extra days.

➤ However, the adjustment to the due date could also contain holidays. Calling the function recursively eliminates this problem.

Let's consider an example. In Figure 3-8 on page 89, one of the seminars had a completion date of 16-JUN-96, making grades due on 04-JUL-96. In the United States this is a national holiday. It happened to fall on a Thursday in 1996. Suppose that the company also gave its employees the Friday after 04-JUL-96 as a holiday, figuring that many employees would take the day anyway to make a long weekend. A *HOLIDAY* table for this company would contain both 04-JUL-96 and 05-JUL-96. Using these holidays, the PL/SQL function in Figure 3-9 produces a new due date for course grades on Monday 08-JUL-96. To arrive at this calculation, the function must adjust the initial due date by one holiday. But because 05-JUL-96 is also a holiday the function gets called recursively until it settles on 08-JUL-96 as the next available weekday that is not also a holiday.

See Also

1. Trying to decide between SQL and PL/SQL approaches to query problems can frequently be difficult. Chapter 9 discusses trade-offs that must sometimes be made between performance and query simplicity. However, there's another component in this trade-off that is often overlooked. The time a query writer must take to write, test, and debug a query should also be considered. If the contortions needed to arrive at a SQL solution reach some ill-defined limit, and the PL/SQL approach saves you time, then factor that into your decision as well.

DECODE Function

The `DECODE` function is a wonderful addition to Oracle's implementation of SQL because it provides the ability to introduce conditional if . . . then . . . else logic in queries. The traditional `DECODE` looks for matches — with a syntax that's equivalent to “if $x = y$, then z , else r ”. In this section you'll see two variations and extensions of the traditional `DECODE`.

How can I use `DECODE` with a greater than (>) operator?

Based on a sales amount, I want to assign a discount rate. I'd like to replicate the if . . . then . . . else functionality in procedural languages. I assume that this needs to be done in the `DECODE` function, but how?

Approach

Let's assume the following situation: If the sale amount is less than \$100, the discount rate is 3%; if the sale amount is greater than or equal to \$100 but less than \$200, the discount rate is 4%; if the sale amount is greater than or equal to \$200 and less than or equal to \$350, the discount rate is 5%; and if the sale amount is greater than \$350, the discount rate is 6%.

Several techniques produce the desired results.

SQL Approach

The classic solution in SQL uses the `DECODE` and `SIGN` functions. The latter function takes the form `SIGN(n)`. If the argument $n < 0$, the function returns -1 ; if $n > 0$, the function returns $+1$; and if $n = 0$, the function returns 0 . This feature of the `SIGN` function makes it useful for determining whether one number is larger, smaller, or equal to a second number.

Figure 3-10 shows the `DECODE` and `SIGN` functions used to create the discount categories required for the query.

```
SELECT
  sales_total "sales",
  DECODE(SIGN(sales_total - 100), -1, 0.03,
        DECODE(SIGN(sales_total - 200), -1, 0.04,
              DECODE(SIGN(sales_total - 350), -1, 0.05, 0, 0.05,
                    0.06))) "discount1"
FROM
  sales
ORDER BY 1;
```

Figure 3-10: Relational expressions using `DECODE` and `SIGN` functions.

Comments

▶ Pay particular attention to the start and end values in a range of values. Use the format shown here when the ending value (i.e., 100) is not included in the range. If the `sales_total` is \$100, then $100 - 100 = 0$, the `SIGN` is 0, and the value would not be assigned a 3% discount. Note that the same format applies to the next `DECODE` and `SIGN`, where the value \$200 is not included in the range.

▶ Unlike the preceding example, the value \$350 is included in the range of values. Use the format shown here when the ending value (i.e., 350) is included in the range. If the `sales_total` is \$350, then $350 - 350 = 0$, the `SIGN` is 0, and the value does get assigned the discount rate of 5% because the `DECODE` includes a 0 condition.

PL/SQL Approach

`DECODE`s quickly become complex. In Figure 3-10 the query nested the `DECODE` functions three levels deep. It can be very difficult sometimes to figure out exactly what a `DECODE` does, particularly if you didn't write the expression and it is not well documented. It's far easier to use a procedural language to implement the relational criteria.

Figure 3-11 shows a `PL/SQL` solution that provides exactly the same functionality as the `DECODE` and `SIGN` functions in Figure 3-10. However, it is much easier to understand and to maintain.

```
CREATE OR REPLACE FUNCTION discount
  (sales_total IN NUMBER)
RETURN NUMBER
IS
  discount_rate NUMBER(3,2);
BEGIN
  IF sales_total > 350 THEN
    discount_rate := 0.06;
  ELSIF sales_total >= 200 THEN
    discount_rate := 0.05;
  ELSIF sales_total >= 100 THEN
    discount_rate := 0.04;
  ELSIF sales_total > 0 THEN
    discount_rate := 0.03;
  END IF;

  RETURN discount_rate;
END;
/
```

Figure 3-11: `PL/SQL` equivalent of the `SQL` in Figure 3-10.

Comments

▶ The sales ranges that determine discount rates are mutually exclusive. The `IF . . . THEN . . . ELSIF` logic ensures exclusive ranges by using a hierarchy of the end value in each range. For example, all sales over \$350 qualify for a 6% discount. Rows meeting this criterion will not be considered in the remainder of the `IF . . . ELSIF` structure. This makes it easy to specify the 5% discount as including any remaining sale equal to or greater than \$200. For this logic to work correctly, you must order the `IF` or `ELSIF` criteria in ascending or descending order based on the sales ranges.

As shown in Figure 3-12, the `SQL` and `PL/SQL` approaches return identical results.

```

SELECT
  sales_total "sales",
  DECODE(SIGN(sales_total - 100), -1, 0.03,
        DECODE(SIGN(sales_total - 200), -1, 0.04,
              DECODE(SIGN(sales_total - 350), -1, 0.05, 0, 0.05,
                    0.06))) "discount1",
  discount(sales_total) "discount2"
FROM sales
ORDER BY 1;

```

sales	discount1	discount2
50	.03	.03
100	.04	.04
200	.05	.05
250	.05	.05
350	.05	.05
500	.06	.06

Figure 3-12: Query using both `SQL` and `PL/SQL` sales discounts.

Comments

▶ The `PL/SQL` function makes the query easier to understand.
 ▶ If a sale exceeds \$350, the appropriate discount is 6%, but a sale of exactly \$350 only qualifies for the 5% discount. Both the `SQL` and the `PL/SQL` solutions correctly dealt with this value that occurred at the endpoint of one of the specified sales ranges.

See Also

1. Chapter 9 (Table 9-3 on page 257) shows results of performance comparisons between the `SQL` and `PL/SQL` approaches to the recode question. Again, you'll be faced with a trade-off.

How do I recode numeric data into categories for frequency reports?

For ease of presentation, I'd like to create sales ranges and then report how many sales fell within each range? Is this possible?

Approach

Actually, this question is very similar to the preceding one. Both situations require that numeric data be grouped into categorical ranges. In the preceding question these ranges determine discount rates to apply to the sales. In this question, however, you only want to code the ranges so that they can be used in a standard frequency distribution report.

The easiest approach uses a PL/SQL function that is nearly identical to the one shown in Figure 3-11 on page 93. Instead of discount rates in the IF statements, however, the new function (called `sales_category`) merely assigns some dummy code. For completeness, the function appears in Figure 3-13.

```
CREATE OR REPLACE FUNCTION sales_category
  (sales_total IN NUMBER)
RETURN NUMBER
IS
  sales_category NUMBER(2);

BEGIN
  IF sales_total > 350 THEN
    sales_category := 4;
  ELSIF sales_total >= 200 THEN
    sales_category := 3;
  ELSIF sales_total >= 100 THEN
    sales_category := 2;
  ELSIF sales_total > 0 THEN
    sales_category := 1;
  END IF;

  RETURN sales_category;
END;
```

Figure 3-13: PL/SQL function recoding numeric sales data in categories.

Comments

► Each range of sales is assigned a code. This gets used in the query but has no significance. If you intend to sort the query results by the assigned codes, make sure you assign them in a meaningful way. Here the assignments are consistent with the sales ranges; the highest sales range gets assigned the highest code value and the lowest sales range gets assigned the lowest value.

The query then uses the `sales_category` function to group numeric sales data into categories. Using the `DECODE` function provides an easy way to assign descriptive names to the categories. Figure 3-14 shows how this is done.

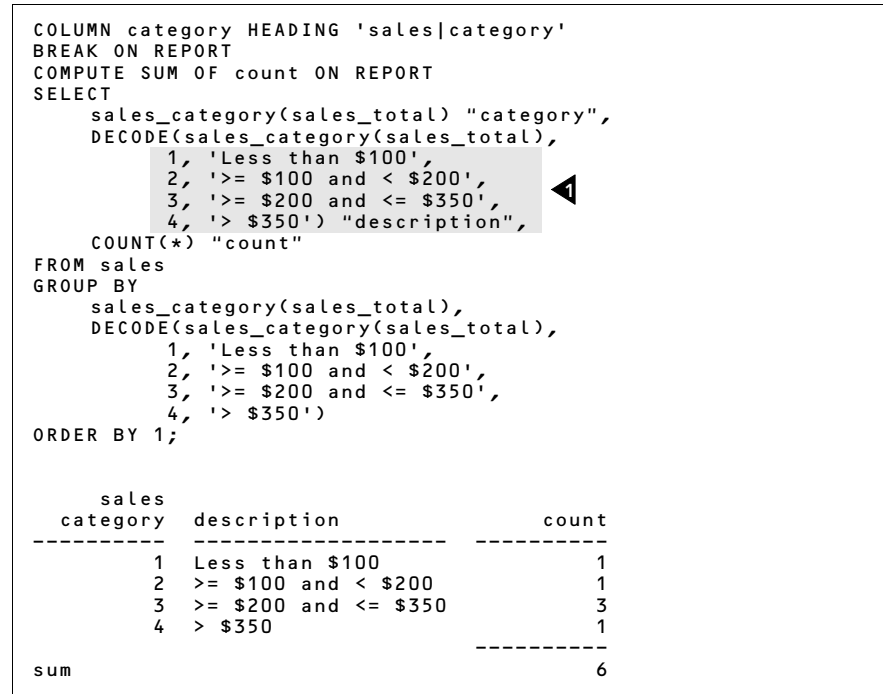


Figure 3-14: Frequency report using recoded sales amounts.

Comments

▶ A `DECODE` function assigns descriptions to each of the dummy sales categories.

See Also

1. `DECODE` can be a very versatile tool for query writers. The index contains many references to examples where the `DECODE` function is used in a query, including everything from computing a median to concatenating strings when `NULL` values exist. See the index under the heading *DECODE function*.

Other Functions

Many useful functions exist besides the DATE and DECODE functions discussed earlier. In this section several of these additional functions appear. Included are CHR, REPLACE, TO_NUMBER, ROUND, and TO_CHAR, which all operate on individual data values. The COUNT function that operates on groups of values is also discussed.

How do I left-justify numeric data?

The identification numbers in my tables are all numeric. I'd like my reports to display these numbers left-justified in a column. Is this possible?

Approach

Oracle includes several conversion functions that convert data values between datatypes — e.g., from character to number, from character to date, from date to character, and so on. One of the functions, TO_CHAR, converts numeric data to character. It accepts as an optional argument a format to use in the conversion.

Displaying numeric data left-justified only requires that TO_CHAR be used to convert the numbers to character data. Do not use the optional conversion format or the characters also will appear right-justified.

Figure 3-15 illustrates how the TO_CHAR function works.

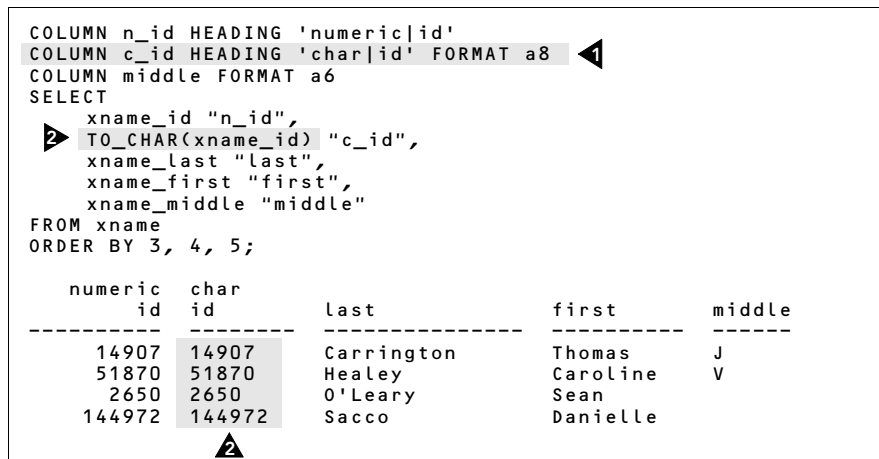


Figure 3-15: Left-justifying numeric data.

Comments

- ▶ Use the `COLUMN` command to control the display width of the converted numeric data.
- ▶ Using the `TO_CHAR` conversion function without the format argument produces a character string that displays left-justified.

See Also

1. For examples of other conversion functions, see the index under *conversion functions*.
2. Chapter 6 (Figure 6-9 on page 186) shows another example where the `TO_CHAR` function is used to convert a number to a character string.

How do I get COUNT() to return a zero?*

I'd like to know how many test scores exist for each person in a report population. If no scores exist, I want 0 to appear in the report. But when I use COUNT(*), I get a 1. How can I get zeroes returned if no rows exist?

Approach

COUNT(*) is the only group function that does not ignore NULL values. Since an outer join creates NULL rows if necessary, COUNT(*) will include the NULL rows in counts. Use instead the COUNT function with an argument that is a NOT NULL data column. Figure 3-16 shows how this works.

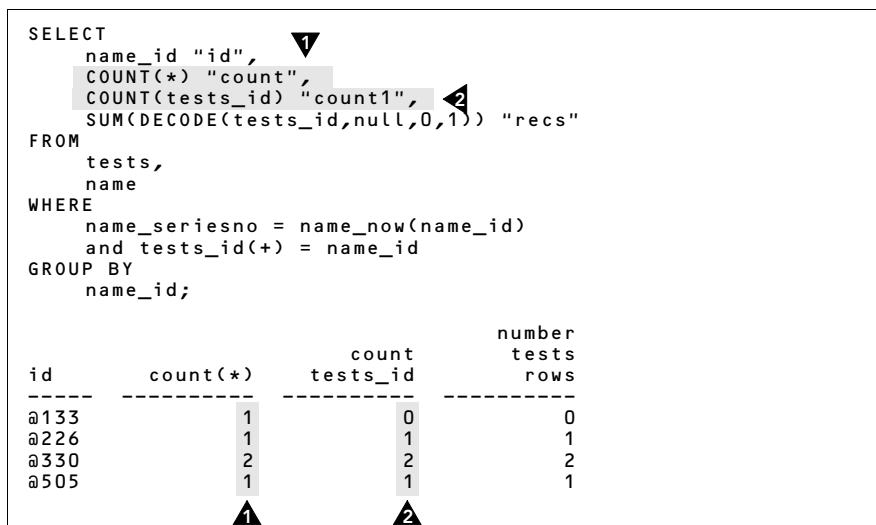


Figure 3-16: Query illustrating counts that return zeroes.

Comments

- ▶ COUNT(*) includes NULL rows created by outer joins.
- ▶ The tests_id column is a NOT NULL column in the TESTS table; it could only be NULL if temporarily created by an outer join. COUNT(tests_id) ignores these NULL rows and returns a 0 count.

See Also

1. See Chapter 1 (Figure 1-9 on page 13) for the name_now function.
2. Beginning on page 156, Chapter 5 discusses the special problems that outer joins sometimes present query writers.

How do I remove control characters embedded in data values?

One of my tables has a data column with lengthy values. When the data were input, carriage returns got embedded in the values. I'd like to get rid of the carriage returns. How can I do this?

Approach

In the 7-bit ASCII character set, a carriage return has the decimal value 10. Assuming that you're using the ASCII character set, you could REPLACE all CHR(10) with a NULL or single space. Figure 3-17 shows an example before and after using the REPLACE function.

```

COLUMN class FORMAT a5
COLUMN desc FORMAT a50
SELECT
    seminar_code "class",
    seminar_desc "desc"
FROM
    seminars
WHERE
    seminar_code = 'Y1C';

class desc
-----
Y1C   An all-night photo shoot
      at New York City Times Square
      as the next millenium arrives

UPDATE seminars SET seminar_desc =
    REPLACE(seminar_desc,chr(10),' ')
WHERE seminar_code = 'Y1C';

class desc
-----
Y1C   An all-night photo shoot at New York City Times Sq
      uare as the next millenium arrives

```

Figure 3-17: Removing control characters from data values.

Comments

- ▶ This data value for seminar_desc has two embedded carriage returns.
- ▶ As used here, the REPLACE function replaces all carriage returns (i.e., CHR(10)) with a single space.

See Also

1. All tables and views used in this book appear in the index under

tables and views. Other examples where the *SEMINARS* table was used include the discussion of due dates that appeared earlier in this chapter (page 86).

2. *REPLACE* is a character function. Other examples of these functions appear in the index under *character functions*.

How do I determine if character data values are positive integers?

One of my tables has a VARCHAR2 data column that contains mostly positive integers. I want to convert the column to a NUMBER datatype, but first I must identify all values that are not positive integers so that I can update these values. How can I find these data values that are not positive integers?

Approach

Checking that a number is a positive integer is straightforward in SQL. You can use the SIGN function to check that the number exceeds zero (i.e., SIGN = 1). You can check that the number is an integer using the MOD function (i.e., MOD(n,1) = 0 for integers). However, converting character strings to numbers with the TO_NUMBER function generates an error if the string is not actually a number (e.g., '100a'), so the query fails without identifying data values containing characters.

It's easier to use a PL/SQL function that includes a section to handle any errors that occur when converting character data to numeric data with the TO_NUMBER function. Figure 3-18 shows one possible PL/SQL solution.

```

CREATE OR REPLACE FUNCTION is_pos_integer
  (in_string IN VARCHAR2)
RETURN VARCHAR2
IS
  pos_integer VARCHAR2(1) := 'N';
  num_string NUMBER;
BEGIN
  num_string := TO_NUMBER(in_string, '99999990');
  IF SIGN(num_string) = 1 THEN
    IF MOD(num_string,1) = 0 THEN
      pos_integer := 'Y';
    END IF;
  END IF;
  RETURN pos_integer;
EXCEPTION
  WHEN OTHERS THEN NULL;
  RETURN pos_integer;
END;
/

```

Figure 3-18: Identifying character strings that are positive integers.

Comments

► The format included in the TO_NUMBER function should be

consistent with the definition of the character data column. In this case, `xtemp_code` is `VARCHAR2(8)`, so a number format of '99999990' was used.

- ▶ Use the `SIGN` function to filter only positive numbers.
- ▶ Use the `MOD` function to find integers.
- ▶ The `EXCEPTION` section traps all errors generated from an invalid `TO_NUMBER` conversion. It returns the value `pos_integer` that is declared on initiation to be 'N'.

Figure 3-19 illustrates how the PL/SQL function `is_pos_integer` can be used to identify character data values that are not positive integers.

```

COLUMN int HEADING 'positive|integer?' FORMAT a8
SELECT
    xtemp_code "char",
    is_pos_integer(xtemp_code) "int"
FROM xtemp;

char          positive
-----      -
100a         N
209          Y
875.1        N
-115         N

```

Figure 3-19: Query identifying character strings that are not positive integers.

See Also

1. The `MOD` function can be used in other creative ways. Chapter 8 (Figure 8-5 on page 238) shows `MOD` being used to systematically sample rows from a population.

How do I prevent inaccurate totals when summing rounded values?

One of my queries computes percentages of total sales by product category. I rounded the percentages to two decimal places, but when I add the rounded values, they don't total 100%. How can I prevent this from happening?

Approach

There are two ways to round numeric values in Oracle's implementation of SQL. You can use the ROUND function to explicitly round values based on the scientific method of rounding. If the digit to be dropped is a 0, 1, 2, 3, or 4, the value is rounded down; if the digit to be dropped is 5, 6, 7, 8, or 9, the value is rounded up. You also can rely on the numeric formats available in the COLUMN command, which implicitly round using the scientific method before displaying the value. For example, the format '999.99' rounds to two decimal places.

Used in a summation, the two methods of rounding can produce different results. Figure 3-20 illustrates this difference.

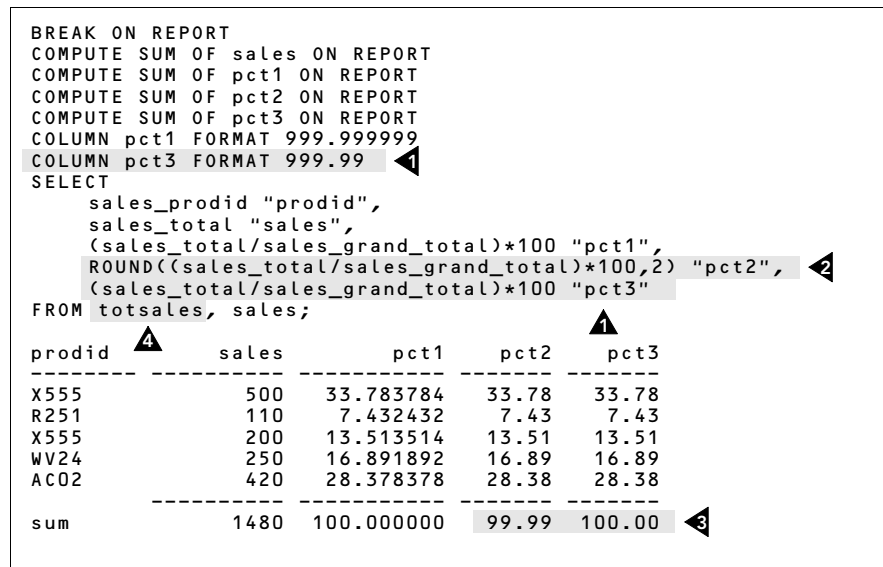


Figure 3-20: Rounding with ROUND function or NUMBER formats.

Comments

- ▶ Round implicitly by applying a NUMBER format in the

COLUMN command. In this case the percentage calculation will be displayed using the format '999.99', which rounds to two decimal places.

▶ Explicitly round by using the ROUND function. In this example, values are again rounded to two decimal places. Note that no column formatting was applied.

▶ Sums of the values rounded with the two methods differ (i.e., 100.00 versus 99.99). In the explicit method, the rounded values are summed to produce 99.99. In the implicit method, the individual (nonrounded) values are summed and then rounded before display. This latter method produces 100.00.

▶ The view *TOTSALES* includes only one data column (sales_grand_total) that contains the total of all sales. This is an instance where a cartesian product created by the absence of join WHERE conditions actually produces the desired result.

Either method of rounding presents problems for the cautious query writer. Using the implicit method, the total appears correct (100.00), but the individually rounded values do not sum to that total. Using the explicit method, the total appears incorrect (99.99), but the individual rounded values do sum to that total. Take your pick; either requires a footnote in a fully documented report.

Note that by both applying the ROUND function and formatting in the COLUMN command, you effectively double round. For example, a ROUND(n,3) rounds to three decimal places. If displayed with a format of '999.99', the rounded three-decimal value is rounded again — this time to two decimal places.

See Also

1. Another example showing the use of the ROUND function appears in Chapter 6 (Figure 6-20 on page 199).
2. Cartesian products occur when you do not join two tables in a query. Frequently this happens by mistake. But occasionally a cartesian product can be put to good use. In Chapter 10 (Figure 10-13 on page 303) a cartesian product is used to distinctly select two objects from *N* objects.

