PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

CHAPTER 17 Exploring Data Warehousing Features

Oracle SQL By Example, Fourth Edition by Alice Rischert. Published by Prentice Hall. Copyright © 2008 by Pearson Education, Inc.

CHAPTER OBJECTIVES

In this chapter, you will learn about:

- Advanced SQL Concepts, Analytical Functions, and the WITH Clause
- ROLLUP and CUBE Operators

This chapter revisits some of the concepts and functionality discussed in previous chapters, including the DECODE function, the CASE expression, and aggregate functions. You will see how to use the PIVOT and UNPIVOT clauses to effortlessly produce cross-tab queries. This will help expand on your existing knowledge and enable you to solve more complex queries.

Analytical functions allow you to explore information in ways never imagined before. You can use analytical functions to determine rankings, perform complex aggregate calculations, and reveal period-to-period changes.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The WITH clause allows you to reuse a query result without having to re-execute the statement; this greatly improves execution time and resource utilization.

The CUBE and ROLLUP operators perform aggregation on multiple levels at once.

This chapter offers you a glimpse at some of the incredibly powerful capabilities of Oracle's data warehousing features, which allow users to query large volumes of summarized data.

LAB 17.1 Advanced SQL Concepts, Analytical Functions, and the WITH Clause

LAB OBJECTIVES

After this lab, you will be able to:

- Transpose a Result Set
- Utilize Analytical Functions and the WITH Clause

Oracle allows you to write queries to generate cross-tabular data in various ways. You've already learned about the DECODE function and the CASE expression.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The PIVOT clauses introduced in Oracle 11*g* simplifies the writing of these queries even further.

Transposing Results

Transposing SQL results into a cross-tab is very useful when you're reporting data. End users demand this layout for improved readability of reports and analysis of data. As you will discover, there are various ways to transpose data into a cross-tab. The new PIVOT functionality to makes this task even easier.

USING THE DECODE FUNCTION

The DECODE function not only permits you to perform powerful *if-then-else* comparisons but also allows you to transpose or pivot the results of queries. For example, the following query returns a list of the number of classes held for each day of the week, with the day of the week formatted using the DY format mask.

742

743

```
SELECT TO_CHAR(start_date_time, 'DY') Day,
       COUNT(*) num_of_classes
  FROM section
 GROUP BY TO_CHAR(start_date_time, 'DY')
 ORDER BY 2
DAY  NUM_OF_CLASSES
---  --------------
FRI               4
...
MON              15
SAT              17
TUE              17

7 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

You can transpose the result, effectively producing a cross-tab to display the result horizontally, with the days of the week as columns and a count below. You do this by nesting the DECODE function in the COUNT function.

```
SELECT COUNT(DECODE(
       TO_CHAR(start_date_time, 'DY'), 'MON', 1)) MON,
       COUNT(DECODE(
       TO_CHAR(start_date_time, 'DY'), 'TUE', 1)) TUE,
       COUNT(DECODE(
       TO_CHAR(start_date_time, 'DY'), 'WED', 1)) WED,
       COUNT(DECODE(
       TO_CHAR(start_date_time, 'DY'), 'THU', 1)) THU,
       COUNT(DECODE(
       TO_CHAR(start_date_time, 'DY'), 'FRI', 1)) FRI,
       COUNT(DECODE(
       TO_CHAR(start_date_time, 'DY'), 'SAT', 1)) SAT,
       COUNT(DECODE(
       TO_CHAR(start_date_time, 'DY'), 'SUN', 1)) SUN
  FROM section
  MON   TUE   WED   THU   FRI   SAT   SUN
----- ----- ----- ----- ----- ----- -----
   15    17     7     5     4    17    13

1 row selected.
```

The syntax of the DECODE function is as follows.

```
DECODE (if_expr, equals_search,
        then_result [,else_default])
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Search and result values can be repeated.

When each row of the expression TO_CHAR(start_date_time, 'DY') is evaluated, Oracle returns the day of the week, in the format DY, which is MON for Monday, TUE for Tuesday, and so on. If the DECODE expression is equal to the search value, the value 1 is returned. Because no ELSE condition is specified, a NULL value is returned.

The COUNT function without an argument does not count NULL values; NULLs are counted only with the wildcard COUNT(*). Therefore, when the COUNT function is applied to the result of either NULL or 1, it counts only records with NOT NULL values. Alternatively, you can also use the SUM function.

743

744

USING CASE

Instead of using the DECODE function, you can write the statement with the equivalent CASE expression for an identical result.

```
SELECT COUNT(CASE WHEN
TO_CHAR(start_date_time, 'DY')
   = 'MON' THEN 1 END) MON,
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
   COUNT(CASE WHEN
TO_CHAR(start_date_time, 'DY')
   = 'TUE' THEN 1 END) TUE,
   COUNT(CASE WHEN
TO_CHAR(start_date_time, 'DY')
   = 'WED' THEN 1 END) WED,
   COUNT(CASE WHEN
TO_CHAR(start_date_time, 'DY')
   = 'THU' THEN 1 END) THU,
   COUNT(CASE WHEN
TO_CHAR(start_date_time, 'DY')
   = 'FRI' THEN 1 END) FRI,
   COUNT(CASE WHEN
TO_CHAR(start_date_time, 'DY')
   = 'SAT' THEN 1 END) SAT,
   COUNT(CASE WHEN
TO_CHAR(start_date_time, 'DY')
   = 'SUN' THEN 1 END) SUN
FROM section
```

USING A SCALAR SUBQUERY

The following query shows yet another way to accomplish the same output. The drawback of this solution is that you execute seven individual queries; this is not as efficient as using DECODE or CASE, which execute only once against the table.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
SELECT (SELECT COUNT(*)
   FROM section
   WHERE TO_CHAR(start_date_time,
'DY') = 'MON') MON,
   (SELECT COUNT(*)
   FROM section
   WHERE TO_CHAR(start_date_time,
'DY') = 'TUE') TUE,
   (SELECT COUNT(*)
   FROM section
   WHERE TO_CHAR(start_date_time,
'DY') = 'WED') WED,
   (SELECT COUNT(*)
   FROM section
   WHERE TO_CHAR(start_date_time,
'DY') = 'THU') THU,
   (SELECT COUNT(*)
   FROM section
   WHERE TO_CHAR(start_date_time,
'DY') = 'FRI') FRI,
   (SELECT COUNT(*)
   FROM section
   WHERE TO_CHAR(start_date_time,
'DY') = 'SAT') SAT,
   (SELECT COUNT(*)
   FROM section
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
   WHERE TO_CHAR(start_date_time,
'DY') = 'SUN') SUN
FROM dual
```

744

745

USING THE PIVOT AND UNPIVOT CLAUSES

The new PIVOT clause in Oracle 11*g* greatly simplifies the writing of cross-tab queries. It rotates the rows into columns. The following example shows the same query listed in the FROM clause as in the beginning of the chapter. If you apply the PIVOT clause to this listing of sections by day of the week, Oracle creates a horizontal row by listing and summing all the values shown the specified IN list.

```
SELECT * FROM
  (
  SELECT TO_CHAR(start_date_time, 'DY') day,
         COUNT(*) num_of_sections
    FROM section
   GROUP BY TO_CHAR(start_date_time, 'DY')
   ) classes_by_day
 PIVOT (SUM(num_of_sections)
 FOR day IN ('MON','TUE', 'WED','THU',
             'FRI','SAT','SUN'))
'MON' 'TUE' 'WED' 'THU' 'FRI' 'SAT' 'SUN'
----- ----- ----- ----- ----- ----- -----
15    17    7     5     4     17    13

1 rows selected
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The following statement creates a table called LOCATION_BY_DATE. The difference from the previous query is that this example also includes the LOCATION column in the inner query. Now the cross-tabulated result is computed not only by the days of the week but also by the location. Another difference is the removal of quotes in the column heading; the columns shown in the IN list have an alias.

```
CREATE TABLE LOCATION_BY_DAY AS
SELECT * FROM
  (
  SELECT TO_CHAR(start_date_time, 'DY') Day,
         s.location,
         COUNT(*) num_of_classes
    FROM section s
   GROUP BY TO_CHAR(start_date_time, 'DY'), location
  ) sections_by_day
 PIVOT (SUM(num_of_classes) FOR day IN
  ('MON' AS MON,'TUE' AS TUE,'WED' AS WED,'THU' AS THU,
   'FRI' AS FRI,'SAT' AS SAT,'SUN' AS SUN))
```

LOCATION	MON	TUE	WED	THU	FRI	SAT	SUN
H310		1					
L210	2	3	1		1	2	1
L214	2	2		2	1	4	4
M500						1	
...							
L509	4	5	3	2	1	4	6
L211		1		1		1	

```
12 rows selected
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Using the LOCATION_BY_DAY table as a source, the UNPIVOT clause allows the rotation of columns back to rows. NUM_OF_CLASSES is a numeric column created as part of the result, and it contains the number of records for the combination of location and day of the week. The DAY column holds the values shown in the IN clause.

The default option of the UNPIVOT clause is EXCLUDE NULLS, but in this example you see the inclusion of null values through the use of the INCLUDE NULLS option. This generates null values, such as for location L210 and THU.

```
SELECT *
  FROM location_by_day
UNPIVOT INCLUDE NULLS (num_of_classes
   FOR day IN (mon, tue, wed, thu, fri, sat, sun))
LOCATION        DAY  NUM_OF_CLASSES
--------------- ---- --------------
L210            MON               2
L210            TUE               3
L210            WED               1
L210            THU
L210            FRI               1
L210            SAT               2
L210            SUN               1
L211            MON

...
M500            SUN

84 rows selected
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

For very difficult queries, where the result cannot be performed using any of the previously mentioned solutions, you might want to consider creating a temporary table to hold intermediate results. Creating temporary tables is discussed in Chapter 12,"Create, Alter, and Drop Tables."

Analytical Functions

Oracle includes a number of very useful functions that allow you to analyze, aggregate, and rank vast amounts of stored data. You can use these analytical functions to find the top-*n* revenue-generating courses, compare revenues of one course with another, or compute various statistics about students' grades.

745

747

Although this lab does not discuss all the available analytical functions, it does provide an overview of the most commonly used functions. You will gain an appreciation of their core functionality and usefulness, particularly with regard to the calculation of rankings or generation of moving averages, moving sums, and so on.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Analytical functions execute queries fairly quickly because they allow you to make one pass through the data rather than write multiple queries or complicated SQL to achieve the same result. This significantly speeds up query performance.

The general syntax of analytical functions is as follows.

```
analytic_function([arguments]) OVER
(analytic_clause)
```

The OVER keyword indicates that the function operates after the results of the FROM, WHERE, GROUP BY, and HAVING clauses have been formed.

ANALYTIC_CLAUSE can contain three other clauses: QUERY_PARTITIONING, ORDER_BY, or WINDOWING.

```
[query_partition_clause]
[order_by_clause
[windowing_clause]]
```

There are slight variations in the general syntax for certain functions, with some requiring specific clauses and others not. The QUERY_PARTIONING clause allows you to split a result into smaller subsets on which you can apply the analytical functions. The ORDER_BY_CLAUSE is much like the familiar

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

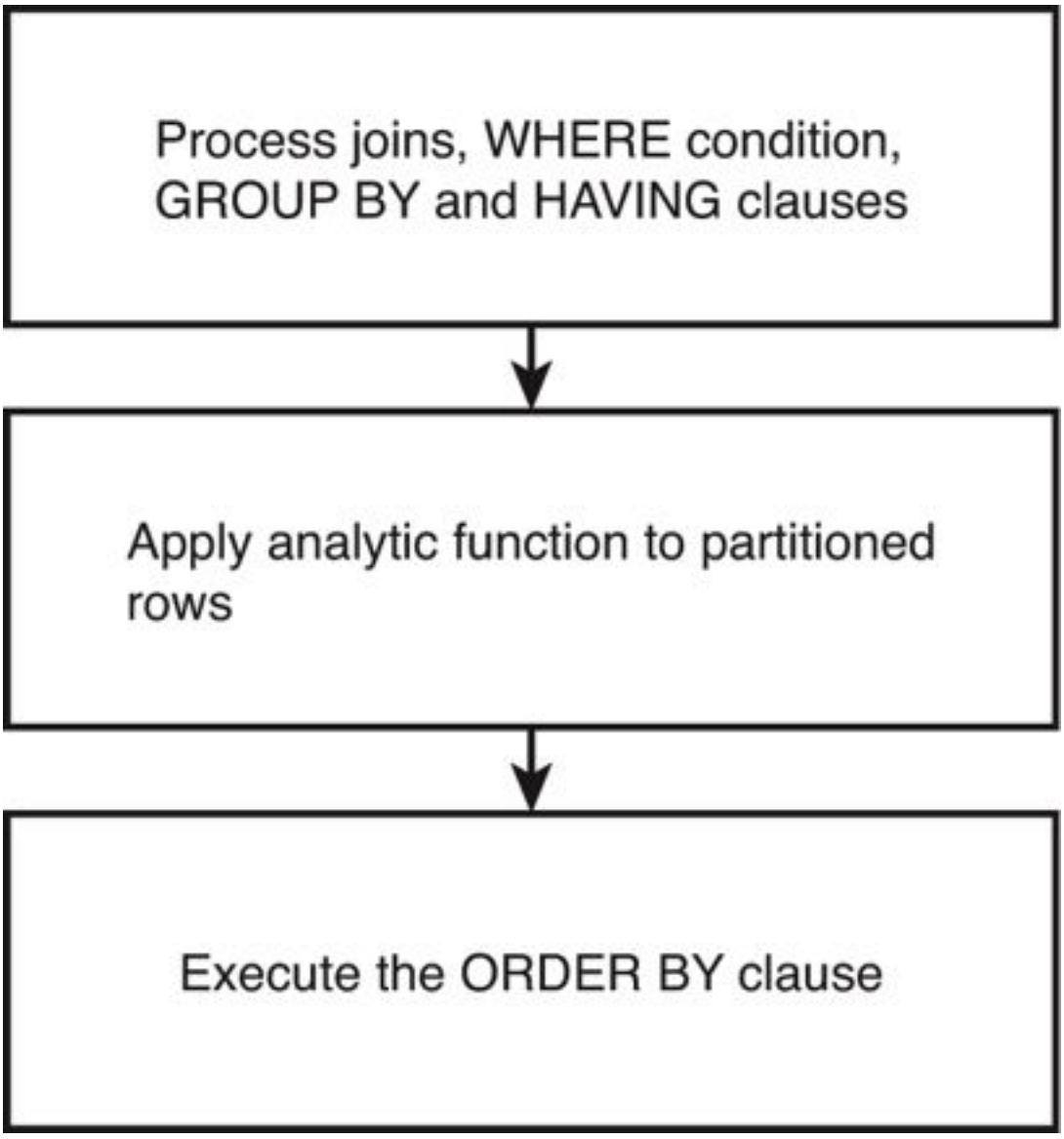
ordering clause; however, it is applied to the result of an analytical function. WINDOWING_CLAUSE lets you compute moving and accumulative aggregates—such as moving averages, moving sums, or cumulative sums—by choosing only certain data within a specified window.

QUERY PROCESSING WITH ANALYTICAL FUNCTIONS

You perform query processing with analytical functions in several steps (see Figure 17.1). First, joins, WHERE, GROUP BY, and HAVING clauses are carried out. The analytical functions then use these results. If any partitioning clause is listed, the rows are split into the appropriate partitions. These partitions are formed after the GROUP BY clause, so you may be able to analyze data by partition, not just the expressions of the GROUP BY clause. If a windowing clause is involved, it determines the ranges of sliding windows of rows. The analytical functions are based against the specified window and allow moving averages, sums, and so on. Analytical functions may have an ORDER BY clause as part of the function specification that allows you to order the result before the analytical function is applied. Finally, if an ORDER BY clause is present at the end of the statement, the result set is sorted accordingly.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

FIGURE 17.1 Query processing steps with analytical functions



PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

ANALYTICAL FUNCTION TYPES

Analytical functions can be categorized into various types; Table 17.1 provides an overview of the different types. Ranking functions determine the ranking of a value (for example, to determine the top three students, based on their grade averages or to determine the first and last values of an ordered group). The reporting functions take the familiar aggregate function capabilities a step further, allowing you to aggregate values without the need for a GROUP BY clause. The windowing capability allows you to generate moving averages, cumulative sums, and the like. The LAG/LEAD functions allow you to easily see how much values changed from one period to another.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

TABLE 17.1 Type and Purpose of Analytical Functions

FUNCTION	TYPE PURPOSE
Ranking	Compute ranking. Function examples are RANK, DENSE_RANK, NTILE, and ROW_NUMBER.
Hypothetical ranking	Determine the rank of hypothetical data values within a result set.
FIRST/LAST	Find the FIRST and LAST values in an ordered group.
Reporting	Use aggregate functions such as SUM, AVG, MIN, MAX, COUNT, VARIANCE, or STDDEV. Also calculate ratios using functions as RATIO_TO_REPORT.
Windowing	Calculate moving averages and cumulative values, using AVG, SUM, MIN, MAX, COUNT, FIRST_VALUE, and LAST_VALUE. (Note that FIRST_VALUE and LAST_VALUE, unlike the FIRST/LAST function, are available only within WINDOWING_CLAUSE.)
Statistical	Calculate statistics. For example, the MEDIAN function and the STATS_MODE function allow calculation of the median and the most frequently occurring value.

748

749

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

LAG/LEAD	Allows you to specify an individual row relative to before or after the current row. The functionality is somewhat similar to windowing and is very useful for comparing period-to-period changes.
Inverse percentile	Determine the value in a data set that is equal to a specific percentile. This functionality is beyond the scope of this book.
Linear regression	Compute linear regression and other related statistics. These functions are beyond the scope of this book.

RANKING FUNCTIONS

Chapter 8, "Subqueries," explores the subject of top-*n* queries, using an inline view and the ROWNUM pseudocolumn. Ranking functions allow for even more advanced functionality. In this lab, you will learn about the differences between the ranking functions.

Let's look first at the DENSE_RANK ranking function. The following query shows the ranking of the grades for student ID 254 in section 87. The grades are ranked by the lowest grade first. Notice the use of the ORDER BY clause in the analytical function.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
SELECT numeric_grade,
       DENSE_RANK() OVER (ORDER BY numeric_grade) AS rank
  FROM grade
 WHERE student_id = 254
   AND section_id = 87
NUMERIC_GRADE       RANK
------------- ----------
           71          1
           71          1
           75          2
...
           91          5
           98          6
           98          6

12 rows selected.
```

The NUMERIC_GRADE value of 71 is the lowest grade of the student; it holds rank number 1. The next highest grade, which is 75, holds rank number 2, and so on. The ORDER BY clause controls the ordering of the ranking. If you want the highest grade to have rank number 1, use DESCENDING in the ORDER BY clause. The default is ASCENDING. You might already have noticed one difference from the inline view; the DENSE_RANK function allows identical values to share the same rank.

To find out the three lowest grades of the student, rather than all the grades, you can modify the query by using

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

the ranking function and an inline view, as in the following example.

```
SELECT *
  FROM (SELECT numeric_grade,
               DENSE_RANK() OVER (ORDER BY numeric_grade)
               AS rank
          FROM grade
         WHERE student_id = 254
           AND section_id = 87)
 WHERE rank <= 3
NUMERIC_GRADE       RANK
------------- ----------
           71          1
           71          1
           75          2
           76          3

4 rows selected.
```

The lowest grade occurs twice. The DENSE_RANK function, unlike the ROWNUM pseudocolumn (discussed in Chapter 8), does not distinguish between grades that share the same values. Therefore, the next query returns only three rows.

```
SELECT *
  FROM (SELECT numeric_grade
          FROM grade
         WHERE student_id = 254
           AND section_id = 87
         ORDER BY numeric_grade)
 WHERE rownum <=3
NUMERIC_GRADE
-------------
           71
           71
           75

3 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The next query shows you the revenue generated per course. It is based on a table named COURSE_REVENUE, and its columns are defined as follows.

```
SQL> DESCR course_revenue
 Name                      Null?    Type
 ------------------------- -------- ------------
 COURSE_NO                 NOT NULL NUMBER(8)
 REVENUE                            NUMBER
 COURSE_FEE                         NUMBER(9,2)
 NUM_ENROLLED                       NUMBER
 NUM_OF_SECTIONS                    NUMBER
```


The REVENUE column holds the revenue generated by the respective COURSE_NO. The COURSE_FEE column shows the amount charged for enrollment in one individual course, and the NUM_ENROLLED column stores the number of students enrolled in a specific course. The NUM_OF_SECTIONS column holds the number of sections per course.

This table is not part of the STUDENT schema but can be created from the additional script available for download from the companion Web site, located at **www.oraclesqlbyexample.com.**

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

RANK, DENSE_RANK, AND ROW_NUMBER

The following query illustrates the differences between the three ranking functions RANK, DENSE_RANK, and ROW_NUMBER. The simplest function of the three is ROW_NUMBER, which is listed as the last column in the result; it has similar functionality to the ROWNUM pseudocolumn. It sequentially assigns a unique number to each row, starting with the number 1, based on the ORDER BY clause ranking of the revenue. When rows share duplicate revenue values, such as the course numbers 20 and 350, one of them arbitrarily gets the next number assigned.

```
SELECT course_no, revenue,
       RANK() OVER (ORDER BY revenue DESC)
         rev_rank,
       DENSE_RANK() OVER (ORDER BY revenue DESC)
         rev_dense_rank,
       ROW_NUMBER() OVER (ORDER BY revenue DESC)
         row_number
  FROM course_revenue
```

COURSE_NO	REVENUE	REV_RANK	REV_DENSE_RANK	ROW_NUMBER
25	53775	1	1	1
122	28680	2	2	2
120	27485	3	3	3
...				
240	14235	7	7	7
20	10755	8	8	8
350	10755	8	8	9
124	9560	10	9	10
125	9560	10	9	11
130	9560	10	9	12
142	8365	13	10	13
147	5975	14	11	14
310	4780	15	12	15
...				
204	1195	23	16	24

```
24 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The RANK function assigns each row a unique number. However, duplicate rows receive the identical ranking, and a gap appears in the sequence before the next rank. In the column labeled REV_RANK, course numbers 20 and 350 share the identical revenue and therefore obtain the same rank. You can see a gap before the next rank.

The ranking function DENSE_RANK assigns duplicate values the same rank. The result of this function is displayed in the column labeled REV_DENSE_RANK.

The syntax of the three functions is as follows.

```
ROW_NUMBER() OVER
([query_partition_clause]
order_by_clause)
   RANK() OVER
([query_partition_clause]
order_by_clause)
DENSE_RANK() OVER
([query_partition_clause]
order_by_clause)
```

The ORDER_BY_CLAUSE is required because it determines the ordering of the rows and therefore ranking. Although in the previous example no null values were present, you should understand that a null is assumed to be equal to another null value. As with

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

the ORDER BY clause at the end of a SQL statement, you can include the NULLS FIRST or NULLS LAST clause to indicate the position of any nulls in the ordered sequence. If you need a refresher on NULLS FIRST or NULLS LAST, refer to Lab 6.2 in Chapter 6, "Aggregate Functions, GROUP BY, and HAVING Clauses."

The syntax includes the optional QUERY_PARTITION_CLAUSE, which allows you to rank across portions of the result set, as in the following examples.

PARTITIONING THE RESULT

The previous query generated the ranking over the entire result. The optional partitioning clause lets you create independent rankings and resets the rank whenever the partitioned values change. In the following query, the COURSE_FEE column is added to show the respective fee per course number. The ranking is now partitioned by a course's fee instead of the entire result. The ranking changes after each value change in the COURSE_FEE column.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
SELECT course_no, course_fee fee, revenue,
       RANK() OVER (PARTITION BY course_fee
          ORDER BY revenue DESC) rev_rank,
       DENSE_RANK() OVER (PARTITION BY course_fee
          ORDER BY revenue DESC) rev_dense_rank,
       ROW_NUMBER() OVER (PARTITION BY course_fee
          ORDER BY revenue DESC) row_number
  FROM course_revenue
```

COURSE_NO	FEE	REVENUE	REV_RANK	REV_DENSE_RANK	ROW_NUMBER
230	1095	15330	1	1	1
240	1095	14235	2	2	2
135	1095	4380	3	3	3
25	1195	53775	1	1	1
122	1195	28680	2	2	2
120	1195	27485	3	3	3
140	1195	17925	4	4	4
100	1195	15535	5	5	5
20	1195	10755	6	6	6
350	1195	10755	6	6	7
124	1195	9560	8	7	8
125	1195	9560	8	7	9
130	1195	9560	8	7	10
...					
204	1195	1195	20	13	21

```
24 rows selected.
```

The first step in the query execution is the formation of the partition, and then for each distinct partition value, the ORDER BY clause is executed. This example demonstrates the use of a single partitioned value, the COURSE_FEE column. You can partition over multiple values/columns by listing each individual expression

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

and separating them with commas in the partitioning clause.

Do not confuse the partitioning clause in analytical functions with the concept of physically splitting very large tables or indexes into smaller partitioned tables and indexes. Table and index partitioning functionality is beyond the scope of this book and independent of analytical functions discussed in this lab.

NTILE

The NTILE function is another ranking function you can use to divide data into buckets of fourth, thirds, or any other groupings. The next SELECT statement shows the result split into four buckets (four quartiles, or 4 × 25 percent buckets). Those in the first quartile of the revenue receive the number 1 in the NTILE column. The next quartile displays the number 2, and so on.

```
SELECT course_no, revenue,
       NTILE(4) OVER (ORDER BY revenue DESC) ntile
  FROM course_revenue
 COURSE_NO REVENUE      NTILE
---------- ------- ----------
        25   53775          1
       122   28680          1
       120   27485          1
       140   17925          1
       100   15535          1
       230   15330          1
       240   14235          2
        20   10755          2
 ...
       204    1195          4

24 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The syntax of the NTILE function is as follows.

```
NTILE(expr) OVER
([query_partition_clause]
order_by_clause)
```


Other less frequently used ranking functions are CUME_DIST and PERCENT_RANK. CUME_DIST determines the position of a specific value relative to a set of values, and PERCENT_RANK calculates the percent rank relative to the number of rows.

HYPOTHETICAL RANKING

Sometimes, you might want to find out how a specific data value would rank if it were part of the result set. You can perform this type of what-if analysis with the hypothetical ranking syntax, which uses the WITHIN GROUP keywords. The following query determines the rank of the value 20,000 if it was present in the REVENUE column of the COURSE_REVENUE table. As you can see from the result of the query, it would have a rank of 4.

```
SELECT RANK(20000) WITHIN GROUP (ORDER BY revenue DESC)
       "Hypothetical Rank"
  FROM course_revenue
Hypothetical Rank
-----------------
                4

1 row selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The syntax for hypothetical ranking is as follows.

```
[RANK|DENSE_RANK|PERCENT_RANK|
CUME_DIST](constant[, ...])
WITHIN GROUP (order_by_clause)
```

FIRST AND LAST FUNCTIONS

The FIRST and LAST functions operate on a set of values to show the lowest or highest value within a result. The syntax of these functions is as follows.

```
aggregate_function KEEP
(DENSE_RANK {LAST|FIRST}
order_by_clause)
[OVER query_partitioning_clause]
```

The following query displays for the GRADE table and SECTION_ID 99 a count of the number of rows with the highest and the lowest grades.

```
SELECT COUNT(*),
       MIN(numeric_grade) min, MAX(numeric_grade) max,
       COUNT(*) KEEP (DENSE_RANK FIRST ORDER BY numeric_grade)
        lowest,
       COUNT(*) KEEP (DENSE_RANK LAST ORDER BY numeric_grade)
        highest
  FROM grade g
 WHERE section_id = 99
  COUNT(*)        MIN        MAX     LOWEST    HIGHEST
---------- ---------- ---------- ---------- ----------
       108         73         99          2          9

1 row selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

This result indicates a total of 108 rows, or individual grades. Of these rows, the lowest grade is 73 and the highest is 99, as computed with the familiar MIN and MAX functions. The query's last two columns apply the FIRST and LAST functions; two grade rows exist for the lowest grade of 73, and nine rows have 99 as the highest grade.

The FIRST and LAST functions allow you to order by one column but apply the aggregate to another column. This effectively eliminates the writing of a subquery that reads the same table yet again.

The following query is an equivalent statement to determine the result of the FIRST and LAST ranking functions. It makes multiple passes through the GRADE table.

```
SELECT numeric_grade, COUNT(*)
  FROM grade
 WHERE section_id = 99
   AND (numeric_grade IN (SELECT MAX(numeric_grade)
                            FROM grade
                           WHERE section_id = 99)
        OR
        numeric_grade IN (SELECT MIN(numeric_grade)
                            FROM grade
                           WHERE section_id = 99))
 GROUP BY numeric_grade
NUMERIC_GRADE   COUNT(*)
------------- ----------
           73          2
           99          9

2 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

MEDIAN

The MEDIAN function returns the median, or middle, value. This function has the following syntax.

```
MEDIAN (expression) [OVER
(query_partitioning_clause)]
```

As indicated by the square brackets in this syntax, the OVER partitioning clause is optional. The following statement excludes the clause. It lists all the distinct grade type codes and the respective median and works just like any other aggregate function.

```
SELECT grade_type_code, MEDIAN(numeric_grade)
  FROM grade
 GROUP BY grade_type_code
GR MEDIAN(NUMERIC_GRADE)
-- --------------------
FI                    85
HM                    85
MT                    88
PA                    87
PJ                    88
QZ                  85.5

6 rows selected.
```

The following SELECT does not use an aggregate function to compute the median. The statement returns the GRADE_TYPE_CODE and the

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

NUMERIC_GRADE columns for section ID 150. The third column, MEDIAN, displays the median of the NUMERIC_GRADE column, partitioned by GRADE_TYPE_CODE. The median for GRADE_TYPE_CODE FI is 88 and for MT is 77 for the respective section ID.

```
SELECT grade_type_code, numeric_grade,
       MEDIAN(numeric_grade) OVER
        (PARTITION BY grade_type_code)
       AS median
  FROM grade
 WHERE section_id = 150
GR NUMERIC_GRADE     MEDIAN
-- ------------- ----------
FI             77         88
FI             88         88
FI             99         88
MT             76         77
MT             77         77
MT             88         77

6 rows selected.
```

STATS_MODE

The STATS_MODE function is another useful statistical function. The following statement illustrates the use of this function, which returns the value that occurs with the greatest frequency. In this instance, the

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

function's parameter is the COST column, and it returns the value 1195.

```
SELECT STATS_MODE(cost)
  FROM course
STATS_MODE(COST)
----------------
            1195

1 row selected.
```

To verify the result that 1195 is the most frequently occurring COST column value, you can run the following query.

```
SELECT cost, COUNT(*)
  FROM course
 GROUP BY cost
 ORDER BY COUNT(*)
      COST   COUNT(*)
---------- ----------
      1595          1
                    1
      1095          3
      1195         25

4 rows selected.
```

The syntax of the STATS_MODE function is as follows.

```
STATS_MODE(expr)
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

REPORTING FUNCTIONALITY

The reporting functionality allows you to compute aggregates for a row in a partition. The syntax is as follows.

```
{SUM|AVG|MAX|MIN|COUNT|STDDEV|
VARIANCE}
    ([ALL|DISTINCT] {expression|*})
    OVER ([PARTITION BY
expression2[,...]])
```

The following example lists the individual grades and the grade type for STUDENT_ID 254 enrolled in SECTION_ID 87. The last column, labeled AVG, displays the grade average for each grade type.

```
SELECT numeric_grade, grade_type_code,
       AVG(numeric_grade)
        OVER(PARTITION BY grade_type_code) AS avg
  FROM grade
 WHERE student_id = 254
   AND section_id = 87
NUMERIC_GRADE GR        AVG
------------- -- ----------
           91 FI         91
           91 HM       84.8
           75 HM       84.8
           98 HM       84.8
...
           91 HM       84.8
           76 MT         76

12 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

There is no GROUP BY clause, even though an aggregate function is used in the SELECT statement. The aggregate function is processed last and works over the GRADE_TYPE_CODE column partition. The partitioning clause works similarly to the GROUP BY clause, grouping together rows and building the aggregate for each of the distinct values of the partition.

If you omit the partition, as indicated in the following example by the empty set of parentheses, the aggregate is computed for all the rows of the result set.

755

758

```
SELECT numeric_grade, grade_type_code,
       AVG(numeric_grade) OVER() AS avg
  FROM grade
 WHERE student_id = 254
   AND section_id = 87
NUMERIC_GRADE GR        AVG
------------- -- ----------
           91 FI 84.5833333
           91 HM 84.5833333
           75 HM 84.5833333
           98 HM 84.5833333
...
           76 MT 84.5833333


12 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

RATIO_TO_REPORT

The RATIO_TO_REPORT function is a reporting function that computes the ratio of a value to the sum of a set of values. The syntax is as follows.

```
RATIO_TO_REPORT(expression) OVER
([query_partition_clause])
```

The following SQL statement illustrates the use of the RATIO_TO_REPORT function. The result of the function, displayed in the RATIO column, represents the ratio of the entire revenue because the partitioning clause is absent. The first row shows the total revenue of COURSE_NO 10 for 1195; the RATIO column indicates that the computed value represents 4.496284 percent of the entire revenue.

```
SELECT course_no, revenue,
       RATIO_TO_REPORT(revenue) OVER () AS ratio
  FROM course_revenue
 COURSE_NO    REVENUE      RATIO
---------- ---------- ----------
        10       1195 .004496284
        20      10755  .04046656
        25      53775   .2023328
       100      15535 .058451698
       120      27485 .103414542
...
       350      10755  .04046656
       420       2390 .008992569

24 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

WINDOWING

The WINDOWING clause allows you to compute cumulative, moving, and centered aggregates. A window has a defining starting point and ending point. The parameters in the windowing clause are always relative to the current row. A sliding window changes the starting point or ending point, depending on the definition of the window.

758
759

A window that defines a cumulative sum starts with the first row and then slides forward with each subsequent row. A moving average has sliding starting and ending rows for a constant logical or physical range.

The following SELECT statement illustrates the computation of a cumulative average and a cumulative sum that is based on the values from the first row until and including the current row. The result shows the individual course numbers and their respective revenues. The CumAvg column shows the cumulative average, and the CumSum column shows the cumulative sum.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
SELECT course_no, revenue,
       AVG(revenue) OVER (ORDER BY course_no
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
         "CumAvg",
       SUM(revenue) OVER (ORDER BY course_no
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
         "CumSum"
  FROM course_revenue
 COURSE_NO    REVENUE     CumAvg     CumSum
---------- ---------- ---------- ----------
        10       1195       1195       1195
        20      10755       5975      11950
        25      53775 21908.3333      65725
       100      15535      20315      81260
       120      27485      21749     108745
...
       350      10755 11451.5217     263385
       420       2390 11073.9583     265775

24 rows selected.
```

Examine the third output row, where COURSE_NO is equal to 25. The average was built based on the revenue values of COURSE_NO 10, 20, and 25, which have REVENUE column values of 1195, 10755, and 53775, respectively. The average of these three values is 21908.3333. The next row builds the average from the previously mentioned values plus the current value, which is 15535; divided by four, this yields 20315. The value in the CumAvg column changes for each subsequent row.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The CumSum column is the cumulative sum, and for each subsequent row it adds the revenue value to the previously computed sum.

The next example shows a centered average; it is computed with the row preceding the current row and the row following the current row. The column is labeled CentAvg. A moving average takes the current row and the previous row, and the result is shown in the MovAvg column.


```
SELECT course_no, revenue,
       AVG(revenue) OVER (ORDER BY course_no
         ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
       "CentAvg",
       AVG(revenue) OVER (ORDER BY course_no
         ROWS 1 PRECEDING)
       "MovAvg"
  FROM course_revenue
 COURSE_NO    REVENUE    CentAvg      MovAvg
---------- ---------- ---------- ----------
        10       1195       5975       1195
        20      10755 21908.3333       5975
        25      53775 26688.3333      32265
       100      15535      32265      34655
       120      27485      23900      21510
...
       420       2390     6572.5     6572.5

24 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

You can expand this functionality for any of the aggregate functions. This allows you to compute moving sums, centered sums, moving min and max values, and so on.

The syntax of the windowing clause is as follows.

```
order_by_clause {ROWS|RANGE}
{BETWEEN
   {UNBOUNDED PRECEDING|CURRENT
ROW|
   expression {PRECEDING|
FOLLOWING}}
AND
   {UNBOUNDED FOLLOWING|CURRENT
ROW|
   expression {PRECEDING|
FOLLOWING}}|
   {UNBOUNDED PRECEDING|CURRENT
ROW|
   expression PRECEDING}}
```

The ROWS and RANGE keywords allow you to define a window, either *physically* through the number of rows or *logically*, such as a time interval or a positive numeric value in the RANGE keyword. The BETWEEN…AND clause defines the starting and ending points of the window, and if none are specified,

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Oracle defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

UNBOUNDED PRECEDING indicates that the window starts at the first row of the partition, and UNBOUNDED FOLLOWING indicates that the window ends at the last row of the partition.

Besides the aggregate functions, such as AVG, COUNT, MIN, MAX, SUM, STDDEV, and VARIANCE, you can use the FIRST_VALUE and LAST_VALUE functions, which return the first value and last value in the window, respectively.

LOGICAL AND PHYSICAL WINDOWS

As mentioned previously, a window can be defined as either a logical window or a physical window. A physical window is defined with the ROWS keyword. A logical window uses the RANGE keyword. Table 17.2 highlights the main differences between logical and physical windows. You will explore these differences in the following exercises.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

TABLE 17.2 Differences Between Physical and Logical Windows

PHYSICAL WINDOW	LOGICAL WINDOW
Specify window with the ROWS keyword.	Specify the window with the RANGE keyword.
Ability to specify the exact number of rows.	Logical offset that determines the starting and ending points of the window; this can be a constant (for example, RANGE 5 PRECEDING), an expression that evaluates to a constant, or an interval (for example, RANGE INTERVAL 10 DAYS PRECEDING).
Duplicate values in the ORDER BY clause do not affect the definition of the current row.	Duplicate values are considered the same for the purpose of defining the current row; therefore, the aggregate function includes all duplicate values, even if they follow the current physical row.
Allows multiple ORDER BY expressions.	Only one ORDER BY expression is allowed.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

THE ORDER BY CLAUSE

ORDER BY in a windowing clause is a mandatory clause that determines the order in which the rows are sorted. Based on this order, the starting and ending points of the window are defined.

```
SELECT numeric_grade, grade_type_code,
       AVG(numeric_grade) OVER(ORDER BY grade_type_code)
       AS cumavg
  FROM grade
 WHERE student_id = 254
   AND section_id = 87
NUMERIC_GRADE GR     CUMAVG
------------- -- ----------
           91 FI         91
           91 HM 85.3636364
           75 HM 85.3636364
           98 HM 85.3636364
...
           76 MT 84.5833333

12 rows selected.
```

761

762

The SELECT statement computes the average of grades, based on the GRADE_TYPE_CODE column. The moving average changes with each change in the GRADE_TYPE_CODE value. Because no windowing clause is specified, the window defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. This is effectively a logical window.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Therefore, the cumulative average value changes with the change of the value in the GRADE_TYPE_CODE column.

One of the keys to understanding logical windows is that the current row equals all the rows with the same ORDER BY values.

In the following statement, the windowing clause is missing, but now there are four columns in the ORDER BY clause. These columns represent the primary key and make the values in the ORDER BY clause unique. In the CUMAVG_OVER_PK column, the average changes with each row. The PARTITION column is formed with a combination of the PARTITIONING clause and the WINDOWING clause. The rows are partitioned by GRADE_TYPE_CODE, and the cumulative average change is reset with each change in the partition.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
SELECT numeric_grade, grade_type_code,
       grade_code_occurrence AS occur,
       AVG(numeric_grade) OVER(ORDER BY student_id,
         section_id, grade_type_code,
         grade_code_occurrence) AS cumavg_over_pk,
       AVG(numeric_grade) OVER(PARTITION BY
        grade_type_code
        ORDER BY student_id, section_id,
        grade_type_code, grade_code_occurrence)
        AS partition
  FROM grade
 WHERE student_id = 254
   AND section_id = 87
```

NUMERIC_GRADE	GR	OCCUR	CUMAVG_OVER_PK	PARTITION
91	FI	1	91	91
91	HM	1	91	91
75	HM	2	85.6666667	83
98	HM	3	88.75	88
98	HM	4	90.6	90.5
81	HM	5	89	88.6
71	HM	6	86.4285714	85.6666667
71	HM	7	84.5	83.5714286
81	HM	8	84.1111111	83.25
91	HM	9	84.8	84.1111111
91	HM	10	85.3636364	84.8
76	MT	1	84.5833333	76

```
12 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

INTERVALS AND THE LOGICAL WINDOW

The following SQL statement shows another example of the functionality of a logical window and the RANGE keyword. The resulting output of the statement lists the number of students who enrolled on specific dates. The sliding windowing functionality with the moving AVG function is applied to the last three columns, named PREV 10 DAYS, NEXT 10 DAYS, and 20-DAY WINDOW.

The column PREV 10 DAYS indicates the average number of students who enrolled 10 days prior to the listed ENROLL_DATE. The starting point of the window is 10 days prior to the ENROLL_DATE of the current row, and the ending point of the window is the current row. The next column, labeled NEXT 10 DAYS, is a window that defines itself from the current row until 10 days after ENROLL_DATE. The column 20-DAY WINDOW shows a 20-day sliding window, starting with 10 days prior to the current row and ending 10 days after the current row.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
SELECT TRUNC(enroll_date) ENROLL_DATE, COUNT(*) "# ENROLLED",
       AVG(COUNT(*))OVER(ORDER BY TRUNC(enroll_date)
        RANGE INTERVAL '10' DAY PRECEDING) "PREV 10 DAYS",
       AVG(COUNT(*))OVER(ORDER BY TRUNC(enroll_date)
        RANGE BETWEEN CURRENT ROW
        AND INTERVAL '10' DAY FOLLOWING) "NEXT 10 DAYS",
       AVG(COUNT(*))OVER(ORDER BY TRUNC(enroll_date)
        RANGE BETWEEN INTERVAL '10' DAY PRECEDING
        AND INTERVAL '10' DAY FOLLOWING) "20-DAY WINDOW"
  FROM enrollment
 GROUP BY TRUNC(enroll_date)
```

ENROLL_DA	# ENROLLED	PREV 10 DAYS	NEXT 10 DAYS	20-DAY WINDOW
30-JAN-07	11	11	17	17
02-FEB-07	14	12.5	20.6	19
04-FEB-07	23	16	22.8	19.8571429
07-FEB-07	20	17	23.4	20.625
10-FEB-07	22	19.75	24.4	22.375
11-FEB-07	24	20.6	27.2	23.8888889
13-FEB-07	25	22.8	28	25.125
16-FEB-07	26	23.4	29	25.4285714
19-FEB-07	25	24.4	30.5	26.3333333
21-FEB-07	36	27.2	36	27.2

```
10 rows selected.
```

Examine the first row; it displays 30-JAN-07 in the ENROLL_DATE column. You see that 11 students enrolled on this date. The average number of enrollments for the previous 10 days is computed by the nested AVG(COUNT(*)) function, which computes the average of the number of enrolled students per ENROLL_DATE within the last 10 days. Because this is the first row and there are no prior values, the average is equal to the number of enrolled students. All

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

the values in the ENROLL_DATE column are truncated to ensure that only date, not time, values are considered.

All the cumulative window values change as you move forward within each subsequent enrollment date. For example, the row with the ENROLL_DATE value 10-FEB-07 shows the average number of enrollments for the previous 10 days (including the current date) as 19.75. This value is computed by averaging the number of enrollments up to and including the 02-FEB-07 value. The value in the NEXT 10 DAYS column is computed once again, through the sliding window of 10 days after the current date and inclusive of the current row. This includes all the enrollments up to and including 19-FEB-07. The value in the 20-DAY WINDOW column includes the prior 10 days and the 10 days following the current date of the row.

The query shows the use of interval literals; remember that interval literals are expressed in the following format.

```
INTERVAL n DAY|MONTH|YEAR
```

If you need to compute the time interval between two dates, use the NUMTOYMINTERVAL function or the NUMTODSINTERVAL function, discussed in Chapter 5, “Date and Conversion Functions.”

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

LAG/LEAD FUNCTIONS

The LAG and LEAD functions allow you to get values from other rows relative to the position of the current row. The syntax is as follows.

```
{LAG|LEAD}(expression[,offset]
[,default])
OVER ([query_partition_clause]
order_by_clause)
```

The LAG function returns one of the values of the previous rows, and the LEAD function returns one of the values of the next rows. The optional OFFSET parameter identifies the relative position of the row; if no parameter is specified, it defaults to 1. The optional default parameter returns the value if the offset falls outside the boundaries of the table or the partition, such as the last and first rows. The LAG and LEAD functions do not have a windowing clause because the offset indicates the exact row.

The following SQL statement shows a useful example of the LAG function. The This Month's Revenue column displays the revenue generated for the month in which an individual section begins. The Previous Month column is computed using the LAG function.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The offset number is specified as 1, which indicates to always use the previous row's value. The Monthly Change column computes the change from the previous month by subtracting the value of the This Month's Revenue column from the value in the Previous Month column.

```
SELECT TO_CHAR(start_date_time, 'MM') "Month",
       SUM(cost) "This Month's Revenue",
       LAG(SUM(cost),1) OVER
        (ORDER BY TO_CHAR(start_date_time, 'MM'))
        "Previous Month",
       SUM(cost)-LAG(SUM(cost),1) OVER
        (ORDER BY TO_CHAR(start_date_time, 'MM'))
        "Monthly Change"
 FROM enrollment e, section s, course c
 WHERE e.section_id = s.section_id
   AND s.course_no = c.course_no
   AND c.cost IS NOT NULL
 GROUP BY TO_CHAR(start_date_time, 'MM')
```

Mo	This Month's Revenue	Previous Month	Monthly Change
04	59745		
05	98780	59745	39035
06	48695	98780	-50085
07	58555	48695	9860

4 rows selected.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

ADVANTAGES OF ANALYTICAL FUNCTIONS

Analytical functions have a number of advantages. Unlike SELECT statements containing aggregate functions and the GROUP BY clause, they allow you to display summary and detail data together rather than write separate queries. The following SELECT statements illustrate this advantage.

This query shows the average revenue per number of sections in a course.

```
SELECT num_of_sections, AVG(revenue)
  FROM course_revenue
 GROUP BY num_of_sections
 ORDER BY 1
NUM_OF_SECTIONS AVG(REVENUE)
--------------- ------------
              1       2987.5
              2   10369.2857
              3   7833.33333
              4      10157.5
              5      22107.5
              6        27485
              8        53775

7 rows selected.
```

The next statement allows you to show any of the table's columns; you are not limited to only the

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

columns listed in the GROUP BY clause, and you avoid the ORA-00937 and ORA-00979 errors. The result shows a list of both summary and detail data.

```
SELECT course_no, revenue, num_of_sections,
       AVG(revenue) OVER (PARTITION BY
         num_of_sections) AS avg_rev_per_cour
 FROM course_revenue
 COURSE_NO    REVENUE NUM_OF_SECTIONS AVG_REV_PER_COUR
---------- ---------- --------------- ----------------
        10       1195               1           2987.5
       132       2390               1           2987.5
       145       2390               1           2987.5
...
       147       5975               1           2987.5
       134       2390               2       10369.2857
       350      10755               2       10369.2857
...
       146       3585               2       10369.2857
...
       135       4380               3       7833.33333
...
        25      53775               8            53775

24 rows selected.
```

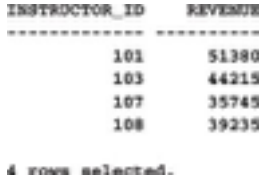
Analytical functions perform postprocessing on the result, which makes them very efficient and simple to use. Some analytical functions cannot be duplicated using any other SQL syntax. For example, the DENSE_RANK or moving and cumulative values cannot be computed without the use of the analytical clause in a statement.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The WITH Clause

The WITH clause, also referred to as the *subquery factoring clause*, offers the benefit of reusing a query when it occurs more than once within the same statement. Instead of storing the query results in a temporary table and performing queries against this temporary table, you can use the WITH clause; it gives the query a name, and you can reference it multiple times. This avoids a reread and re-execution of the query, which improves overall query execution time and resource utilization, particularly when very large tables and/or joins are involved. The WITH clause also simplifies the writing of SQL statements. You most frequently use this type of query when querying against large volumes of data, such as in data warehouses.

The WITH keyword indicates that multiple SQL statements are involved. The following example determines the revenue generated by each instructor. The query result returns revenue for only those instructors who have revenue greater than the average generated by all instructors combined.



```
INSTRUCTOR_ID    REVENUE
------------- ----------
          101      51380
          103      44215
          107      35745
          108      39235

4 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The WITH clause creates a name for the "temporary" result REVENUE_PER_INSTRUCTOR. This result is then referred to in the subsequent SELECT statements in the context of the original query.

Because the REVENUE_PER_INSTRUCTOR query involves a join and an aggregate function, it is useful to examine the result of the join to ensure the accuracy of the aggregate function.

```
SELECT instructor_id, cost, s.section_id, student_id
  FROM section s, course c, enrollment e
 WHERE s.section_id = e.section_id
   AND c.course_no = s.course_no
 ORDER BY instructor_id
INSTRUCTOR_ID       COST SECTION_ID STUDENT_ID
------------- ---------- ---------- ----------
          101       1195         87        256
...
          105       1195        152        138
          105       1195        152        144
          105       1195        152        206
          105       1195        152        207
          105       1195        144        153
          105       1195        144        200
          105       1095        113        129
          105       1195        105        202
          105       1195         91        232
          105       1195        105        263
          105       1195        105        261
          105       1195        105        259
          105       1195        105        260
          105       1195         83        124
          105       1195         91        271
...
          108       1195         86        102

226 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

For example, review INSTRUCTOR_ID 105. Effectively, the SUM function adds up all the individual values of the COST column, resulting in a total of 19020. Adding the GROUP BY clause and the SUM function to the joined tables produces this output, which is identical to the result achieved by the REVENUE_PER_INSTRUCTOR query.

764
768

```
SELECT instructor_id, SUM(cost)
  FROM section s, course c, enrollment e
 WHERE s.section_id = e.section_id
   AND c.course_no = s.course_no
 GROUP BY instructor_id
INSTRUCTOR_ID  SUM(COST)
------------- ----------
          101      51380
          102      24995
          103      44215
          104      29675
          105      19020
          106      21510
          107      35745
          108      39235

8 rows selected.
```

To determine the average revenue for all instructors, you nest the two aggregate functions AVG and SUM. REVENUE_PER_INSTRUCTOR, however, reuses the previous result instead of executing the following query.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
SELECT AVG(SUM(cost))
  FROM section s, course c, enrollment e
 WHERE s.section_id = e.section_id
   AND c.course_no = s.course_no
 GROUP BY instructor_id
AVG(SUM(COST))
--------------
     33221.875

1 row selected.
```

Without the WITH clause, you need to write the following statement, which effectively performs the reading of the tables and the join twice--once for the subquery and once for the outer query. This requires more resources and consumes more time than writing the query with the subquery factoring clause, particularly when large tables, many joins, and complex aggregations are involved.

```
SELECT instructor_id, SUM(cost) AS
revenue
   FROM section s, course c,
enrollment e
WHERE s.section_id = e.section_id
   AND c.course_no = s.course_no
GROUP BY instructor_id
HAVING SUM(cost) > (SELECT
AVG(SUM(cost))
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
    FROM section s, course c,
enrollment e
WHERE s.section_id = e.section_id
    AND c.course_no = s.course_no
GROUP BY instructor_id)
```

768

769

Do not confuse the WITH clause in subqueries with the START WITH clause used in hierarchical queries, discussed in Chapter 16, "Regular Expressions and Hierarchical Queries."

You can also write this query using the analytical function discussed previously. As you have discovered throughout this book, there are often many ways to formulate a SQL statement. Knowing the options is useful and allows you to choose the most efficient statement.

```
SELECT *
  FROM (SELECT instructor_id, SUM(cost) AS revenue,
               AVG(SUM(cost)) OVER() AS avg
          FROM section s, course c, enrollment e
         WHERE s.section_id = e.section_id
           AND c.course_no = s.course_no
         GROUP BY instructor_id) t
 WHERE revenue > avg
INSTRUCTOR_ID    REVENUE        AVG
------------- ---------- ----------
          101      51380  33221.875
          103      44215  33221.875
          107      35745  33221.875
          108      39235  33221.875

4 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Inter-Row Calculations

Oracle can perform spreadsheet-like calculations in queries. This capability can be useful for budgeting and forecasting. It allows the display of additional rows or calculations in the query result through the application of formulas in the MODEL clause of a SELECT statement. Following is a query example based on the MODEL_EXAMPLE table, consisting of the columns COURSE, GENDER, YEAR, and ENROLL_NO, which store data about courses, gender, the enrollment year, and the enrollment figures, respectively. (This table can be created based on the additional script available from the companion Web site, located at **www.oraclesqlbyexample.com.)**

This table contains only data for the year 2008, but the following sample query result shows additional rows (shaded in gray) for the years 2009 and 2010. For those years, the query determines the projected enrollment numbers in the courses Spanish II and Spanish III, based on the previous year's enrollment numbers. You need to keep in mind that these rows represent a query result, and the query does not update the MODEL_EXAMPLE table.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
SELECT course, gender, year, s
    FROM model_example
   MODEL PARTITION BY (gender)
         DIMENSION BY (year, course)
         MEASURES (enroll_no s)
   (
    s[2009,'Spanish II'] = s[2008,'Spanish I'],
    s[2010,'Spanish III'] = ROUND((s[2009, 'Spanish II'])*0.80)
   )
   ORDER BY year, gender, course
```

COURSE	G	YEAR	S
Spanish I	F	2008	37
Spanish II	F	2008	59
Spanish III	F	2008	3
Spanish I	M	2008	3
Spanish II	M	2008	35
Spanish III	M	2008	34
Spanish II	F	2009	37
Spanish II	M	2009	3
Spanish III	F	2010	30
Spanish III	M	2010	2

10 rows selected.

769

770

The MODEL clause lists a PARTITION element that is much like the familiar partition in the previously discussed analytical functions query and defines the top-level grouping. The DIMENSION element identifies the key to the MEASURES cells, which hold numeric values. In this example, the GENDER column is the partitioned column, and the DIMENSION columns YEAR and COURSE identify the groups within each partition. The MEASURE column is ENROLL_NO and is uniquely identified by the combination of partition and dimension columns.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The rule s[2009, 'Spanish II'] = s[2008, 'Spanish I'] is a definition of an assignment. A rule consists of references. The two-dimensional [2009, 'Spanish II'] reference defines a single cell for the dimensions YEAR and COURSE. The measure s is an alias, as defined in the MEASURE element. The left side of the assignment indicates the destination cell; the right side determines the values for it. In this instance, the destination cell with the year 2009 and the course Spanish II does not exist in the MODEL_EXAMPLE table and is created in the result set by default; this action is referred to as UPSERT. The right side of the assignment states that the measure cell should be the same value as the value in year 2008 and for the course Spanish I. Two rows are created for this year: one for male and another for female.

The next rule, s[2010, 'Spanish III'] = ROUND((s[2009, 'Spanish II'])*0.80), defines that the enrollment number for Spanish III in year 2010 should be 80 percent of the enrollment for Spanish II in 2009, rounded to nearest whole number.

The previous statement’s calculations refer to specific individual cells. You can also specify ranges of cells, create loops, and so on. The following example

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

illustrates the projection of enrollment figures for Spanish II for the years 2009 through 2011 (shaded in gray). The FOR loop creates these years. The measure values are based on the previous year's value of Spanish II. The CURRENTV( ) function returns the current value of the YEAR dimension column, but because the function reads CURRENTV( )-1, it looks at the previous year's value. The calculation ROUND((s[CURRENTV( )-1, 'Spanish II'])*1.1) assumes a 10 percent increase each subsequent year and is rounded to the nearest whole number.

770
771

```
SELECT course, gender, year, s
  FROM model_example
 WHERE course = 'Spanish II'
 MODEL PARTITION BY (gender)
       DIMENSION BY (year, course)
       MEASURES (enroll_no s)
 (
  s[FOR year FROM 2009 TO 2011 INCREMENT 1,
    'Spanish II']
  = ROUND((s[CURRENTV()-1,'Spanish II'])*1.1)
 )
 ORDER BY year, gender, course
```

COURSE	G	YEAR	S
Spanish II	F	2008	59
Spanish II	M	2008	35
Spanish II	F	2009	65
Spanish II	M	2009	39
Spanish II	F	2010	72
Spanish II	M	2010	43
Spanish II	F	2011	79
Spanish II	M	2011	47

8 rows selected.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Creating Your Own Custom Function

So far, you have used a variety of the rich offerings of Oracle's built-in functions. Using the PL/SQL language, you can write your own functions. Although this book does not discuss PL/SQL in detail, this brief section offers a glimpse into what a customized PL/SQL function can accomplish when used in a SQL statement.

Why would you write your own PL/SQL function? This functionality is quite useful in the case of complicated query logic because it allows you to easily call the function that hides the complexity of the logic. The following example shows a custom function that chooses the next business day if the passed date falls on a weekend day. The function queries the HOLIDAY table to make sure the next business day does not fall on a company holiday. You can use this function much as you would any of the single-row functions you have learned about.

The next example shows how the function called NEXT_BUSINESS_DAY works. As you can see, it hides all the complexity of figuring out the date for you and simply returns the next business day.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Because August 7, 2010, falls on a Saturday, the next business day is a Monday. Therefore, if you're trying to write a report to list the due dates of invoices and you must always display a business day as the due date, it is easy to use this function. Furthermore, it is simpler than writing a long CASE expression. Other statements can use the stored function and take advantage of the functionality. In addition, a stored function ensures that subsequent changes to the logic are automatically applied to any statement that calls the function.

Following is the PL/SQL code that implements and stores NEXT_BUSINESS_DAY function in the database.

```
CREATE OR REPLACE FUNCTION
next_business_day(i_date DATE)
   RETURN DATE IS
   v_date DATE;
BEGIN
   v_date:=i_date;
SELECT NVL(MAX(holiday_end_date)+1,
v_date)
   INTO v_date
   FROM holiday
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
   WHERE v_date BETWEEN
holiday_start_date AND
holiday_end_date;
   IF TO_CHAR(v_date, 'DY') = 'SAT'
THEN
   v_date:=v_date+2;
   ELSIF TO_CHAR(v_date, 'DY') =
'SUN' THEN
   v_date:=v_date+1;
   END IF;
   RETURN v_date;
EXCEPTION
   WHEN OTHERS THEN
   RETURN NULL;
END next_business_day;
/
```

In previous Oracle versions, functions executed from a SQL statement had to be wrapped inside a package—a type of PL/SQL object.

Clearly, writing your own custom functions simplifies the logic of complicated business rules and can overcome SQL limitations. However, you must keep in mind that a function will be executed for every row of the result set; the key is to eliminate as many rows as possible before applying the function.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

To learn more about PL/SQL, refer to *Oracle PL/SQL by Example*, 4th Edition, by Benjamin Rosenzweig and Elena Silvestrova Rakhimov (Prentice Hall, 2008).

LAB 17.1 EXERCISES

a) The following query result is a list of all the distinct course costs and a count of each. Write the query that achieves this result.

```
      1095       1195       1595       NULL
---------- ---------- ---------- ----------
         3         25          1          1

1 row selected.
```

b) Modify the following query to display the top three revenue-generating courses. If there is a tie in the revenue, include the duplicates.

```
SELECT course_no, revenue,
   RANK() OVER (ORDER BY revenue
DESC)
   rev_rank,
   DENSE_RANK() OVER (ORDER BY
revenue DESC)
   rev_dense_rank,
   ROW_NUMBER() OVER (ORDER BY
revenue DESC)
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
    row_number
FROM course_revenue
```

c) Based on the following statement, explain how the result of the AVG column is achieved.

```
SELECT numeric_grade AS grade, grade_type_code,
       grade_code_occurrence AS occurrence,
       AVG(numeric_grade) OVER(PARTITION BY grade_type_code
       ORDER BY grade_code_occurrence) AS avg
  FROM grade
 WHERE student_id = 254
   AND section_id = 87
```

GRADE	GR	OCCURRENCE	AVG
91	FI	1	91
91	HM	1	91
75	HM	2	83
98	HM	3	88
98	HM	4	90.5
81	HM	5	88.6
71	HM	6	85.6666667
71	HM	7	83.5714286
81	HM	8	83.25
91	HM	9	84.1111111
91	HM	10	84.8
76	MT	1	76

12 rows selected.

d) How would you formulate the problem this query is attempting to solve? SELECT e.*, SUM(diff) OVER (ORDER BY 1 ROWS BETWEEN

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

e) Explain the result of the following query.

```
WITH
num_enroll AS
(SELECT COUNT(*) num_students, course_no
  FROM enrollment e JOIN section s
 USING (section_id)
 GROUP BY course_no),
avg_stud_enroll AS
(SELECT AVG(num_students) avg#_of_stud
  FROM num_enroll
 WHERE num_students <> (SELECT MAX(num_students)
                         FROM num_enroll))
SELECT course_no, num_students
  FROM num_enroll
 WHERE num_students > (SELECT avg#_of_stud
                        FROM avg_stud_enroll)
   AND num_students < (SELECT MAX(num_students)
                        FROM num_enroll)
```

COURSE_NO	NUM_STUDENTS
20	9
100	13
120	23
122	24
124	8
125	8
130	8
140	15
230	14
240	13
350	9

11 rows selected.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

LAB 17.1 EXERCISE ANSWERS

a) The following query result is a list of all the distinct course costs and a count of each. Write the query that achieves this result.

```
      1095       1195       1595       NULL
---------- ---------- ---------- ----------
         3         25          1          1

1 row selected.
```

ANSWER: There are various ways to write the query. You can use PIVOT, CASE, or DECODE to obtain the same output.

If you use the PIVOT clause, you can write the following query.

```
SELECT *
   FROM (SELECT cost, COUNT(*)
num_of_rows
   FROM course
   GROUP BY cost) course_cost
PIVOT (SUM(num_of_rows)
FOR cost IN (1095, 1195, 1595,
null))
```

Or use the CASE expression, as follows.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
SELECT COUNT(CASE WHEN cost =
1095 THEN 1 END) ”1095”,
   COUNT(CASE WHEN cost = 1195
THEN 1 END) ”1195”,
   COUNT(CASE WHEN cost = 1595
THEN 1 END) ”1595”,
   COUNT(CASE WHEN cost IS NULL
THEN 1 END) ”NULL”
FROM course
```

Or use the DECODE function, as follows.

```
SELECT COUNT(DECODE(cost, 1095,
1)) ”1095”,
   COUNT(DECODE(cost, 1195, 1))
”1195”,
   COUNT(DECODE(cost, 1595, 1))
”1595”,
   COUNT(DECODE(cost, NULL, 1))
”NULL”
FROM course
```

The transposed result uses the COUNT function to count the row only if it meets the search criteria of the DECODE function or CASE expression. The first column of the SELECT statement tests for courses with a cost of 1095. If this expression is equal to 1095, the DECODE

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

function or CASE expression returns the value 1; otherwise, it returns a NULL value. The COUNT function counts NOT NULL values; the NULL values are not included. This is different from the way the COUNT(*) function works, which includes NULL values in the count.

The last column in the SELECT statement tests for courses with a NULL cost. If this condition of a NULL course cost is true, the DECODE function or the CASE expression returns a 1, and the row is included in the count.

If your Oracle version does not support the PIVOT clause, you might want to use the CASE expression because it is easier to understand than DECODE functionality. It also has the added benefit of being ANSI compatible.

b) Modify the following query to display the top three revenue-generating courses. If there is a tie in the revenue, include the duplicates.

```
SELECT course_no, revenue,
   RANK() OVER (ORDER BY revenue
DESC)
   rev_rank,
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
   DENSE_RANK() OVER (ORDER BY
revenue DESC)
   rev_dense_rank,
   ROW_NUMBER() OVER (ORDER BY
revenue DESC)
   row_number
FROM course_revenue
```

ANSWER: Using an inline view, you restrict the rows to only those where the values in the REV_DENSE_RANK column are 3 or less. The DENSE_RANK function is the better choice, just in case some courses share the same revenue.

```
SELECT course_no, revenue, rev_dense_rank
  FROM (SELECT course_no, revenue,
               DENSE_RANK() OVER (ORDER BY revenue ASC)
                 rev_dense_rank
          FROM course_revenue) t
 WHERE rev_dense_rank <= 3
```

COURSE_NO	REVENUE	REV_DENSE_RANK
10	1195	1
204	1195	1
132	2390	2
145	2390	2
420	2390	2
134	2390	2
146	3585	3
330	3585	3

```
8 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

BOTTOM-*N*RANKING

The bottom-*n* ranking is similar to top-*n* except that you change the order of the ranking. Instead of ordering the revenue in descending order, the ORDER BY clause is now in ascending order. The query to determine the bottom three revenue-ranking courses is shown here.

```
SELECT course_no, revenue, rev_dense_rank
  FROM (SELECT course_no, revenue,
               DENSE_RANK() OVER (ORDER BY revenue ASC)
                 rev_dense_rank
          FROM course_revenue) t
 WHERE rev_dense_rank <= 3
 COURSE_NO    REVENUE REV_DENSE_RANK
---------- ---------- --------------
        10       1195              1
       204       1195              1
       132       2390              2
       145       2390              2
       420       2390              2
       134       2390              2
       146       3585              3
       330       3585              3

8 rows selected.
```

c) Based on the following statement, explain how the result of the AVG column is achieved.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
SELECT numeric_grade AS grade, grade_type_code,
       grade_code_occurrence AS occurrence,
       AVG(numeric_grade) OVER(PARTITION BY grade_type_code
       ORDER BY grade_code_occurrence) AS avg
  FROM grade
 WHERE student_id = 254
   AND section_id = 87
```

GRADE	GR	OCCURRENCE	AVG
91	FI	1	91
91	HM	1	91
75	HM	2	83
98	HM	3	88
98	HM	4	90.5
81	HM	5	88.6
71	HM	6	85.6666667
71	HM	7	83.5714286
81	HM	8	83.25
91	HM	9	84.1111111
91	HM	10	84.8
76	MT	1	76

```
12 rows selected.
```

ANSWER: The statement computes a cumulative average for each partition. The average is reset after the values listed in the partition clause change. The ORDER BY clause determines the definition of the window.

After the WHERE clause is executed, postprocessing with the analytical function takes over. The partitions are built first. The result shows three distinct values: FI, HM, and MT, for final, homework, and midterm, respectively. After a change in partition, the average is reset.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

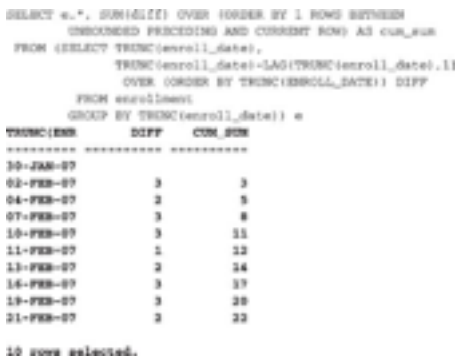
It is easiest to follow the logic by examining the individual computations for each respective row. They are listed next to the result.

```
   GRADE GR OCCURRENCE        AVG
------- -- ---------- ----------
     91 FI          1         91  /* (91)/1 */
     91 HM          1         91  /* (91)/1 partition change*/
     75 HM          2         83  /* (91+83)/2 */
     98 HM          3         88  /* (91+83+98)/3 */
     98 HM          4       90.5  /* (91+83+98+98)/4 */
...
     76 MT          1         76  /* (76)/1 partition change*/

   12 rows selected.
```

For the HM partition there are duplicates for the NUMERIC_GRADE column values (rows 4 and 5 show 98). Because the column GRADE_CODE_OCCURRENCE is part of the ORDER BY clause and has different and ever-changing values, the cumulative average is computed for each row.

d) How would you formulate the problem this query is attempting to solve?



```
SELECT e.*, SUM(diff) OVER (ORDER BY 1 ROWS BETWEEN
       UNBOUNDED PRECEDING AND CURRENT ROW) AS cum_sum
 FROM (SELECT TRUNC(enroll_date),
              TRUNC(enroll_date)-LAG(TRUNC(enroll_date),1)
               OVER (ORDER BY TRUNC(ENROLL_DATE)) DIFF
         FROM enrollment
        GROUP BY TRUNC(enroll_date)) e
TRUNC(ENR      DIFF    CUM_SUM
--------- ---------- ----------
30-JAN-07
02-FEB-07          3          3
04-FEB-07          2          5
07-FEB-07          3          8
10-FEB-07          3         11
11-FEB-07          1         12
13-FEB-07          2         14
16-FEB-07          3         17
19-FEB-07          3         20
21-FEB-07          2         22

10 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

ANSWER: The query determines the difference in days between the distinct ENROLL_DATE values; it considers only the date, not the time values. In the query result, you see the distinct ENROLL_DATE values, the difference in days between each value, and the cumulative sum of days in the last column.

e) Explain the result of the following query.

```
WITH
num_enroll AS
(SELECT COUNT(*) num_students, course_no
  FROM enrollment e JOIN section s
 USING (section_id)
 GROUP BY course_no),
avg_stud_enroll AS
(SELECT AVG(num_students) avg#_of_stud
  FROM num_enroll
 WHERE num_students <> (SELECT MAX(num_students)
                         FROM num_enroll))
SELECT course_no, num_students
  FROM num_enroll
 WHERE num_students > (SELECT avg#_of_stud
                        FROM avg_stud_enroll)
   AND num_students < (SELECT MAX(num_students)
                        FROM num_enroll)
```

COURSE_NO	NUM_STUDENTS
20	9
100	13
120	23
122	24
124	8
125	8
130	8
140	15
230	14
240	13
350	9

11 rows selected.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

ANSWER: The query returns courses and the respective enrollments above the average enrollment per course, excluding courses with the highest enrollment.

This query uses two inline queries, NUM_ENROLL and AVG_STUD_ENROLL. The first query, NUM_ENROLL, computes the number of enrolled students per course. The second query, AVG_STUD_ENROLL, uses the previous query to determine the average number of students enrolled, excluding the course with the highest enrollment. The last query shows the COURSE_NO column together with the number of enrolled students where the course has an enrollment that is higher than the average enrollment. Remember that this average excludes the course with the highest enrollment. The last condition specifically excludes the course with the highest enrollment in the result, as it would otherwise be included, because it obviously has a higher-than-average enrollment.

As you have discovered, SQL allows you to express a query in many different ways to achieve the same result. The differences often lie in the

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

efficiency of the statement. The result you see can also be obtained by using the following query; several joins are required with each execution of the condition, thus requiring more resources and time than simply reusing the temporarily stored query result. Furthermore, the WITH clause breaks down the problem into individual pieces, therefore simplifying the writing of complex queries.

```
SELECT course_no, COUNT(*)
num_students
FROM enrollment e JOIN section s
USING (section_id)
GROUP BY course_no
HAVING COUNT(*) > (SELECT
AVG(COUNT(*))
   FROM enrollment e JOIN section
s
   USING (section_id)
   GROUP BY course_no
   HAVING COUNT(*) <> (SELECT
MAX(COUNT(*))
   FROM enrollment e JOIN section
s
   USING (section_id)
   GROUP BY course_no))
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
AND COUNT(*) <>(SELECT
MAX(COUNT(*))
   FROM enrollment e JOIN section
s
   USING (section_id)
   GROUP BY course_no)
```

780

780

781

LAB 17.1 Quiz

In order to test your progress, you should be able to answer the following questions.

1) The difference between the RANK and DENSE_RANK functions is that DENSE_RANK leaves no gaps in ranking sequence when there are ties in the values.

   _____**a)** True

   _____**b)** False

2) If you use the RANK function without a partitioning clause, the ranking works over the entire result set.

   _____**a)** True

   _____**b)** False

3) The ROWS keyword in a windowing clause defines a logical window.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

_____**a)** True

_____**b)** False

4) The LAG and LEAD functions can contain a windowing clause.

_____**a)** True

_____**b)** False

5) The FIRST_VALUE and LAST_VALUE functions work only with a windowing clause.

_____**a)** True

_____**b)** False

ANSWERS APPEAR IN APPENDIX A.

LAB 17.2 ROLLUP and CUBE Operators

LAB OBJECTIVES

After this lab, you will be able to:

- Use the ROLLUP and CUBE Operators
- Understand the GROUPING and GROUPING SETS Capabilities

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Oracle includes many enhancements to the GROUP BY clause, which makes aggregating data from many different perspectives simpler and more efficient. These enhancements come in the form of the ROLLUP and CUBE operators, the GROUPING function, and GROUPING SETS capability. The ROLLUP and CUBE operators allow you to create subtotals and grand totals by simply eliminating the need to run multiple queries against the data.

You will see how useful these capabilities are for analyzing data and discovering relationships between data elements. This functionality is primarily used in data warehousing environments, with the goal of providing users with reporting and decision support functionality against summarized data.

End-user access is often accomplished with various querying tools that read this summarized data and allow users to “slice and dice” the information in any way they desire. As many companies are increasingly using their databases to gain a competitive market advantage and to better support their customers as well as reduce costs, this capability allows you to uncover much information about the data.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Many software vendors offer various types of tools that present the summarized data in an easily understandable format; Oracle also supplies its own version of an end-user decision support software tool, called Oracle BI Discoverer. While we will not discuss the capabilities and merits of such tools, we illustrate the summarization capabilities of Oracle to discover relationships and information about the data. You will see that this functionality is quite powerful and extremely valuable, yet very easy to use.

The ROLLUP Operator

The ROLLUP operator allows you to create subtotals and grand totals, also referred to as *super aggregate rows*, for various groupings and for all rows.

782

783

The table used for the exercises in this lab is called INSTRUCTOR_SUMMARY and is included in the supplemental tables you can download from the companion Web site, located at **www.oraclesqlbyexample.com.** It contains summary data generated from the various tables in the STUDENT schema.

The DESCRIBE command lists the following columns. The primary key of the table consists of the

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

INSTRUCTOR_ID, SEMESTER_YEAR, and SEMESTER_MONTH columns.

```
SQL> DESCR instructor_summary
 Name                          Null?    Type
 ----------------------------- -------- ---------------
 INSTRUCTOR_ID                 NOT NULL NUMBER(8)
 GENDER                                 CHAR(1)
 CAMPUS                                 VARCHAR2(11)
 SEMESTER_YEAR                 NOT NULL VARCHAR2(4)
 SEMESTER_MONTH                NOT NULL VARCHAR2(2)
 NUM_OF_CLASSES                         NUMBER
 NUM_OF_STUDENTS                        NUMBER
 REVENUE                                NUMBER
```

INSTRUCTOR_ID is identical to the familiar column in the INSTRUCTOR table. The GENDER column identifies the gender as male (M), female (F), or unknown (U) (if the title of an instructor is different than Mr., Mrs., or Ms.). The CAMPUS column indicates the name of the campus where the instructor's office is located. The SEMESTER_YEAR and SEMESTER_MONTH columns display the year and month the instructor worked. The column NUM_OF_CLASSES holds the number of sections the instructor taught, the NUM_OF_STUDENTS shows the column number of students for all the sections, and the REVENUE column contains the revenue generated by the individual instructor for these classes.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Using the familiar GROUP BY clause, the following query produces a list of instructors, grouped by the GENDER, SEMESTER_YEAR, and SEMESTER_MONTH columns, including the total number of students taught.

```
SELECT gender, semester_year AS year,
       semester_month AS month,
       SUM(num_of_students) AS total
  FROM instructor_summary
 GROUP BY gender, semester_year, semester_month
G YEAR MO      TOTAL
- ---- -- ----------
F 2007 05           0
F 2007 06          16
F 2008 07          37
M 2007 06          45
M 2007 07          79
U 2007 05           0
U 2007 07          49

7 rows selected.
```

783

784

All the distinct occurrences of these three columns are summarized.

Instead of using the GROUP BY clause, the next query uses the ROLLUP operator. With it, you discover the formation of subtotals for each of the groups. Your individual result does not display the shading shown

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

here; it is used here to illustrate the location of the formed super aggregate rows.

```
SELECT gender, semester_year AS year,
       semester_month AS month,
       SUM(num_of_students) AS total
  FROM instructor_summary
 GROUP BY ROLLUP(gender, semester_year, semester_month)
G YEAR MO      TOTAL
- ---- -- ----------
F 2007 05           0
F 2007 06          16
F 2007             16 Subtotal for female in year 2007
F 2008 07          37
F 2008             37 Subtotal for female in year 2008
F                  53 Subtotal for entire female gender
M 2007 06          45
M 2007 07          79
M 2007            124 Subtotal for male in year 2007
M                 124 Subtotal for entire male gender
U 2007 05           0
U 2007 07          49
U 2007             49 Subtotal for unknown gender 2007
U                  49 Subtotal for entire unknown gender
                  226 Grand Total

15 rows selected.
```

In this result, you notice some of the same rows as in the previous GROUP BY query. The difference is that here you see additional rows. These additional rows indicate subtotals for the GENDER and YEAR columns, a subtotal for each GENDER column only, and a grand total column. The subtotals are formed for each change in value.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The first shaded set of summary rows indicates 16 female students in 2007, 37 females in 2008, and 53 female students total.

You needed only one query to generate different groupings of data. The individual groupings are group 1: gender, year, and month; group 2: gender and year; group 3: gender; and group 4: grand total.

The number of columns or expressions appearing in the ROLLUP clause determines the number of groupings. The formula is *n* + 1, where *n* is the number of columns listed in the ROLLUP clause. Without the ROLLUP clause, you would need to write four individual queries.

784

785

The first query is already listed at the beginning of this lab, but it is repeated here, and also shown with partial output. This query represents the different individual rows grouped by the columns GENDER, SEMESTER_YEAR, and SEMESTER_MONTH.

```
SELECT gender, semester_year AS year,
       semester_month AS month,
       SUM(num_of_students) AS total
  FROM instructor_summary
 GROUP BY gender, semester_year, semester_month
G YEAR MO      TOTAL
- ---- -- ----------
F 2007 05          0
F 2007 06         16
...
U 2007 07         49

7 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The second query lists the GENDER and SEMESTER_YEAR columns and computes the respective summary data.

```
SELECT gender, semester_year AS year,
       SUM(num_of_students) AS total
  FROM instructor_summary
 GROUP BY gender, semester_year
G YEAR      TOTAL
- ---- ----------
F 2007          16
F 2008          37
M 2007         124
U 2007          49

4 rows selected.
```

The third query is a listing grouped to give you a subtotal for the gender.

```
SELECT gender, SUM(num_of_students) AS total
  FROM instructor_summary
 GROUP BY gender
G      TOTAL
- ----------
F          53
M         124
U          49

3 rows selected.
```

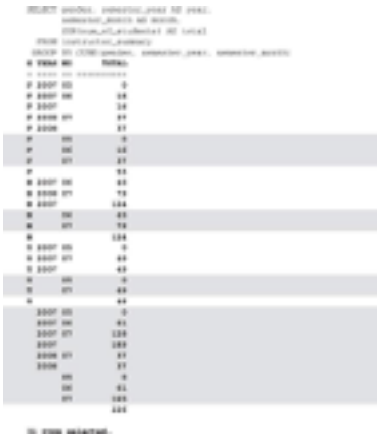

The last query is the grand total for all the rows.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The idea of the ROLLUP operator is that you don't need to write multiple queries, and Oracle doesn't need to process the table multiple times but rather does all the necessary work in one pass through the table. This is a very efficient and quick way to accomplish the desired result.

The CUBE Operator

The CUBE operator takes the formation of super aggregates another step further: It allows you to generate all the possible combinations of groups. If you have *n* columns or expressions in the GROUP BY clause, the CUBE operator generates 2^n groupings. The CUBE operator gets its name from the different combinations that can be achieved from an *n*-dimensional cube. The following example illustrates the combinations based on the previously used query. The CUBE operator is now substituted for the ROLLUP operator.



PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The shading around the rows indicates the new additionally formed subtotals. You see a subtotal for GENDER and SEMESTER_MONTH, another for SEMESTER_YEAR and SEMESTER_MONTH, and another for SEMESTER_YEAR only; you also see a total by SEMESTER_MONTH.

CUBE determines the 2???????????? different combinations for the three columns, which results in a total of eight different subtotals. ROLLUP already determined four, and CUBE adds four more combinations.

Determining the ROLLUP and CUBE Combinations

Assume that you have three rollup groups in your GROUP BY ROLLUP clause, listed like the following hypothetical columns named YEAR, MONTH, and WEEK.

```
GROUP BY ROLLUP (year, month, week)
```

You get the following four rollup groups, according to the *n* + 1 formula: group 1 consists of year, month, and week; group 2 shows year and month; group 3 shows year; and group 4 shows the grand total. Hierarchies such

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

as time periods or sales territories (for example, continent, country, state, county) lend themselves naturally to the ROLLUP operator, although you can obviously create your own or use your own combination of columns to roll up.

If you use the CUBE operator instead of ROLLUP, you generate eight different combinations, all of which are listed in Table 17.3. The empty set of parentheses, ( ), indicates the grand total.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

TABLE 17.3 Grouping Combinations

OPERATOR	FORMULA GROUPING	COMBINATIONS	
ROLLUP (year, month, week)	$n + 1$	3 + 1 = 4	(year, month, week), (year, month), (year), ( )
CUBE (year, month, week)	2^n	$2^3 = 8$	(year, month, week), (year, month), (year), $2^3 = 8$ (month, week), (year, week), (week), (month), ( )

786
788

To exclude certain subtotals from the CUBE result or the ROLLUP result, you can selectively remove columns from the CUBE or ROLLUP clause and place them into the GROUP BY clause or generate summaries based on

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

composite columns. Although it is useful to know about these options, you can simplify this with the GROUPING SETS clause, discussed shortly.

Table 17.4 lists the partial ROLLUP and CUBE results when a column moves into the GROUP BY clause. The example uses three columns called YEAR, MONTH, and WEEK. Most notably, the summary grand total is missing from all the partial rollups.

TABLE 17.4 Partial ROLLUP and CUBE Operations

GROUP BY CLAUSE	GROUPING COMBINATIONS
year, ROLLUP (month, week)	(year, month, week), (year, month), (year)
year, month ROLLUP (week)	(year, month, week), (year, month)
year, CUBE(month, week)	(year, month, week), (year, month), (year, week), (year)
year, month, CUBE (week)	(year, month, week), (year, month)

You can further group on composite columns. A composite column defined in this context is a collection of columns and is listed within a set of parentheses. As such, a composite column is treated as a single unit; this

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

avoids any unnecessary aggregations for specific levels. Table 17.5 lists the results of operations on composite columns.

TABLE 17.5 Composite Column ROLLUP and CUBE Operations

COMPOSITE COLUMNS	GROUPING COMBINATIONS
ROLLUP ((year, month), week)	(year, month, week), (year, month), ( )
ROLLUP (year), (month, week)	(year, month, week), (month, week)
CUBE ((year, month), week)	(year, month, week), (year, month), (week), ( )
CUBE (year), (month, week)	(year, month, week), (month, week)

GROUPING SETS

Computing and displaying only selective results can actually be simplified with the GROUPING SETS extension of the GROUP BY clause. You explicitly state which summaries you want to generate. The following query applies the GROUPING SETS functionality to the query drawn on previously.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.


```
SELECT gender, semester_year AS YEAR,
       semester_month AS month,
       SUM(num_of_students) AS total
  FROM instructor_summary
 GROUP BY GROUPING SETS
       ((gender, semester_year),   -- 1st Group
        (semester_month),          -- 2nd Group
        ())                        -- 3rd Group
```

```
G YEAR MO      TOTAL
- ---- -- ----------
F 2007            16
F 2008            37
M 2007           124
U 2007            49
       05          0
       06         61
       07        165
                 226

8 rows selected.
```

The query produces three sets: one for the GENDER and SEMESTER_YEAR columns, a second for SEMESTER_MONTH, and the last one for the grand total. Each individual set must be enclosed in parentheses; the empty set of parentheses indicates the grand total. The GROUPING SETS clause provides the advantage of reading the table once and generating the results immediately and only for those summaries in which you are interested. GROUPING SETS

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

functionality is very efficient and yet selective about the results you choose to report.

If you have many hierarchy groupings, you may not want to specify all the different groupings individually. You can combine multiple GROUPING SETS clauses to generate yet more combinations.

The following example lists two GROUPING SETS clauses in the GROUP BY clause.

```
GROUP BY GROUPING SETS (year,
month),
   GROUPING SETS (week, day)
```

The cross-product results in the following equivalent groupings.

```
GROUP BY GROUPING SETS (
   (year, week),
   (year, day),
   (month, week),
   (month, day))
```

The GROUPING Function

One of the purposes of the GROUPING function is that it helps you distinguish summary rows from any rows that are a result of null values. The following query shows a CUBE operation on the columns

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

SEMESTER_YEAR and CAMPUS. As it turns out, the CAMPUS column contains null values, and it is difficult to distinguish between the summary row and an individual row holding a null value.

```
SELECT semester_year AS year, campus,
       SUM(num_of_classes) AS num_of_classes
```



```
  FROM instructor_summary
 GROUP BY CUBE (semester_year, campus)
 ORDER BY 1
```

```
YEAR CAMPUS      NUM_OF_CLASSES
---- ----------- --------------
2007 DOWNTOWN                10
2007 LIBERTY                 19
2007 MORNINGSIDE             29
2007                         10 Summary row or null?
2007                         68
2008 MORNINGSIDE             10
2008                         10
     DOWNTOWN                10
     LIBERTY                 19
     MORNINGSIDE             39
                             10 Summary row or null?
                             78

12 rows selected.
```

The GROUPING function eliminates any ambiguities. Whenever you see a value of 1 in a column where the GROUPING function is applied, it indicates a super aggregate row, such as a subtotal or grand total row created by the ROLLUP or CUBE operator.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

When examining the result, you observe the columns where the GROUPING function is applied and has a value of 0 or 1. The number 1 indicates that this column is a super aggregate row.

The first shaded area on the resulting output shows a 0 in the GP_CAMPUS column; this indicates that the NULL value in the CAMPUS column is indeed a NULL. The next row lists the number 1 in the GP_CAMPUS column; this designates the row as a summary row. It lists 68 classes for all campus locations in 2007.

The second shaded row shows the number 1 in the GP_YEAR column. This indicates that the SEMESTER_YEAR column is an aggregate, just like the previous three rows. That means the rows display the aggregate values for each individual campus for all years.

The last row contains the number 1 for both the GP_YEAR and the GP_CAMPUS columns. This indicates the grand total.

You can use the GROUPING function not only to determine whether it’s a generated row or NULL value but to return certain rows from the result set with the

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

HAVING clause. This is yet another way to selectively choose certain summary rows only.

```
HAVING GROUPING(campus) = 1
```

You can use the GROUPING function to add labels to the super aggregate rows. Instead of a blank column, you can display a label such as GRAND TOTAL:.

```
SELECT CASE WHEN GROUPING(semester_year) = 1
                AND GROUPING(campus) = 1 THEN 'GRAND TOTAL:'
            ELSE semester_year
       END AS year,
       CASE WHEN GROUPING(semester_year) = 1
                AND GROUPING(campus) = 1 THEN NULL
            ELSE campus
       END AS campus,
       SUM(num_of_classes) AS num_of_classes,
       GROUPING (semester_year) GP_YEAR,
       GROUPING (campus) GP_CAMPUS
  FROM instructor_summary
 GROUP BY CUBE (semester_year, campus)
 ORDER BY 4, 2, 5
```

YEAR	CAMPUS	NUM_OF_CLAS	GP_YEAR	GP_CAMPUS
2007	DOWNTOWN	10	0	0
2007	LIBERTY	19	0	0
2007	MORNINGSIDE	29	0	0
2007		10	0	0
2007		68	0	1
2008	MORNINGSIDE	10	0	0
2008		10	0	1
	DOWNTOWN	10	1	0
	LIBERTY	19	1	0
	MORNINGSIDE	39	1	0
		10	1	0
GRAND TOTAL:		78	1	1

12 rows selected.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Remember that you can also use the NVL or COALESCE function to test for null values and return a substitute value.

The GROUPING_ID Function

If a query includes many GROUPING functions, you might want to consider consolidating the columns with the GROUPING_ID function. This function is not only similar in name and functionality to GROUPING, it also allows multiple columns as a parameter; it returns a number indicating the level of aggregation in the rollup or cube.

The GROUPING_ID function returns a single number that identifies the exact aggregation level of every row.

```
SELECT semester_year AS year,
       campus,
       SUM(num_of_classes) AS num_of_classes,
       GROUPING (semester_year) GP_YEAR,
       GROUPING (campus) GP_CAMPUS,
       GROUPING_ID(semester_year, campus)
       AS GROUPING_ID
  FROM instructor_summary
 GROUP BY CUBE (semester_year, campus)
YEAR CAMPUS      NUM_OF_C GP_YEAR GP_CAMPUS GROUPING_ID
---- ---------- -------- ------- --------- -----------
2007 DOWNTOWN         10       0         0           0
2007 LIBERTY          19       0         0           0
2007 MORNINGSIDE      29       0         0           0
2007                  10       0         0           0
2007                  68       0         1           1
2008 MORNINGSIDE      10       0         0           0
2008                  10       0         1           1
     DOWNTOWN         10       1         0           2
     LIBERTY          19       1         0           2
     MORNINGSIDE      39       1         0           2
                      10       1         0           2
                      78       1         1           3

12 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The GROUPING_ID function works just like the GROUPING function that generates zeros and ones. However, the GROUPING_ID function concatenates the zeros and ones and forms a bit vector, which is treated as a binary number. The GROUPING_ID function returns the binary number's base-10 value. A value of 1 for each GROUPING column indicates that this is the grand total. The binary number 11 for the two-level column aggregation represents the number 3, which is returned by the GROUPING_ID function. Zeros in all the columns of the GROUPING functions indicate that this is the lowest aggregation level.

792
793

Table 17.6 lists binary numbers and their numeric equivalents on the example of a four-column CUBE, representative of a GROUP BY clause such as CUBE(year, month, week, day). The column labeled GROUPING_ID displays the result of the GROUPING_ID function for each individual column; the Bit-Vector column indicates which bits are turned on and off. The BIN_TO_NUM function allows you to convert a binary value to a number. Every argument represents a bit in the bit vector.

```
SELECT BIN_TO_NUM(1,1)
  FROM dual
BIN_TO_NUM(1,1)
---------------
              3

1 row selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

TABLE 17.6 Bit to Numeric Representation on the Example of a Four-Column Cube

GROUPING_ID	NUMERIC EQUIVALENT	BIT-VECTOR AGGREGATION LEVEL
0	0 0 0 0	(year, month, week, day)
1	0 0 0 1	(year, month, week)
2	0 0 1 0	(year, month, day)
3	0 0 1 1	(year, month)
4	0 1 0 0	(year, week, day)
5	0 1 0 1	(year, week)
6	0 1 1 0	(year, day)
7	0 1 1 1	(year)
8	1 0 0 0	(month, week, day)
9	1 0 0 1	(month, week)
10	1 0 1 0	(month, day)
11	1 0 1 1	(month)
12	1 1 0 0	(week, day)
13	1 1 0 1	(week)
14	1 1 1 0	(day)
15	1 1 1 1	( )

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

You can use the GROUPING_ID function for labeling columns, as discussed previously. However, its primary use is in *materialized views*, which are Oracle objects that allow you to create and maintain aggregate summary tables. Storing pre-aggregate summary information is an important technique for maximizing query performance in large decision support applications. The effect of the creation of this materialized view is that data is physically stored in a table. Any changes to the underlying tables are reflected in the materialized view through various refresh methods. (Unlike views, discussed in Chapter 13, "Indexes, Sequences, and Views," materialized views require physical storage.)

793
794

```
CREATE MATERIALIZED VIEW
instructor_sum_mv
STORAGE(INITIAL 5 M PCTINCREASE 0)
AS
SELECT semester_year AS year,
   campus,
   SUM(num_of_classes) AS
num_of_classes,
   GROUPING_ID(semester_year,
campus)
   AS GROUPING_ID
FROM instructor_summary
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
GROUP BY CUBE (semester_year,
campus)
Materialized view created.
```

To perform this operation, you must have the CREATE MATERIALIZED VIEW privilege.

THE GROUP_ID FUNCTION

The GROUP_ID function lets you distinguish among duplicate groupings; they may be generated as a result of combinations of columns listed in the GROUP BY clause. GROUP_ID returns the number 0 to the first row in the set that is not yet duplicated; any subsequent duplicate grouping row receives a higher number, starting with the number 1.

```
SELECT semester_year AS year, campus,
       SUM(num_of_classes) AS num_of_classes,
       GROUPING_ID(semester_year, campus) GROUPING_ID,
       GROUP_ID()
  from instructor_summary
 group by GROUPING SETS
       (semester_year, ROLLUP(semester_year, campus))
YEAR CAMPUS      NUM_OF_CLASSES GROUPING_ID GROUP_ID()
---- ----------- -------------- ----------- ----------
2007 DOWNTOWN                10           0          0
2007 LIBERTY                 19           0          0
2007 MORNINGSIDE             29           0          0
2007                         10           0          0
2008 MORNINGSIDE             10           0          0
                             78           3          0
2007                         68           1          0
2008                         10           1          0
2007                         68           1          1
2008                         10           1          1

10 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

This query illustrates the result of the GROUP_ID function, which returns a 0 for the first row; the subsequent identical group returns the number 1 in the GROUP_ID() column. You can see this in the last two rows of the result.

If you have complicated queries that may generate duplicate values, you can eliminate those rows by including the condition HAVING GROUP_ID( ) = 0.

LAB 17.2 EXERCISES

a) Describe the effect of the following SQL statement and its resulting output.

```
SELECT salutation AS SALUTATION, SUBSTR(phone, 1,3)
       AS "Area Code",
       TO_CHAR(registration_date, 'MON') AS "Reg.Month",
       COUNT(*)
  FROM student
 WHERE SUBSTR(phone, 1,3) IN ('201','212','718')
   AND salutation IN ('Mr.', 'Ms.')
 GROUP BY ROLLUP (salutation, SUBSTR(phone, 1,3),
       TO_CHAR(registration_date, 'MON'))
```

SALUT	Area Code	Reg.Month	COUNT(*)
Mr.	201	FEB	34
Mr.	201	JAN	9
Mr.	201		43
Mr.	212	FEB	1
Mr.	212	JAN	1
Mr.	212		2
Mr.	718	FEB	72
Mr.	718	JAN	17
Mr.	718		89
Mr.			134
Ms.	201	FEB	27
Ms.	201	JAN	5
Ms.	201		32
Ms.	212	FEB	2
Ms.	212	JAN	1
Ms.	212		3
Ms.	718	FEB	52
Ms.	718	JAN	13
Ms.	718		65
Ms.			100
			234

21 rows selected.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

b) Answer the following questions about the result set.

How many female students are there in total?

How many male students live in area code 212?

What is the total number of students?

How many female students live in the area code 718 and registered in January?

c) If the CUBE operator is used on the query in exercise a instead of ROLLUP, how many different combinations of groups do you get? List the groups.

d) Describe the result of the following query, which uses the GROUPING SET extension to the GROUP BY clause.

```
SELECT SALUTATION, SUBSTR(phone, 1,3) "Area Code",
       TO_CHAR(registration_date, 'MON') "Reg.Month",
       COUNT(*)
  FROM student
 WHERE SUBSTR(phone, 1,3) IN ('201','212','718')
   AND salutation IN ('Mr.', 'Ms.')
 GROUP BY
       GROUPING SETS
       ((SALUTATION, SUBSTR(phone, 1,3)),
        (SALUTATION, TO_CHAR(registration_date, 'MON')),
        ()
       )
SALUT Are Reg  COUNT(*)
----- --- --- ----------
Mr.   201            43
Mr.   212             2
Mr.   718            89
Ms.   201            32
Ms.   212             3
Ms.   718            65
Mr.       FEB       107
Mr.       JAN        27
Ms.       FEB        81
Ms.       JAN        19
                    234

11 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

e) Write the individual queries to proof the numbers found in exercise d.

LAB 17.2 EXERCISE ANSWERS

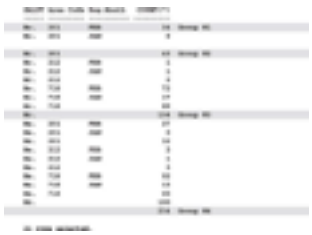
a) Describe the effect of the following SQL statement and its resulting output. (For space reasons, the code is not repeated here.)

ANSWER: The query generates summary totals by using the ROLLUP operator. This effectively creates groupings, which are counting rows in the STUDENT table. Four rollup groups are generated based on the columns: salutation, area code, and registration month (labeled "Reg.Month").

796

797

The first ROLLUP group is salutation, area code, and registration date; the second group is salutation and area code; the third is salutation; and the last is a grand total of all the rows. Each shaded row in the following output indicates the first occurrence of a group.



PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

b) Answer the following questions about the result set.

How many female students are there in total?

How many male students live in area code 212?

What is the total number of students?

How many female students live in the area code 718 and registered in January?

ANSWER: You can obtain all these answers by examining the result set.

You can easily answer the first question (How many female students are part of the result set?) by looking at the result set. The correct answer is that there are 100 female students. Following is an excerpt of the output.

```
SALUT Area Code Reg.Month   COUNT(*)
----- --------- ---------- ---------
...
Ms.                              100
```

You can obtain the answer to the next question, about the male students living in area code 212, from the following row. As you can see, the number of students is 2.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

You can obtain the total number of students satisfying the WHERE clause of the query with the last row, the grand total row. There are 234 in total.

```
SALUT Area Code Reg.Month   COUNT(*)
----- --------- ---------- ---------
...
                                  234
```

Finally, there are 13 female students who live in area code 718 and registered in January.

```
SALUT Area Code Reg.Month   COUNT(*)
----- --------- ---------- ---------
...
Ms.   718       JAN                13
```

c) If the CUBE operator is used on the query in exercise a instead of ROLLUP, how many different combinations of groups do you get? List the groups.

ANSWER: There are three columns involved; therefore, there are 2^3 possible combinations, which translates to eight different groupings.

Group 1: Salutation, area code, and registration date

Group 2: Salutation and area code

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

Group 3: Salutation

Group 4: Area code and registration date

Group 5: Registration date

Group 6: Salutation and registration date

Group 7: Area code

Group 8: Grand total

d) Describe the result of the following query, which uses the GROUPING SET extension to the GROUP BY clause.

```
SELECT SALUTATION, SUBSTR(phone, 1,3) "Area Code",
       TO_CHAR(registration_date, 'MON') "Reg.Month",
       COUNT(*)
  FROM student
 WHERE SUBSTR(phone, 1,3) IN ('201','212','718')
   AND salutation IN ('Mr.', 'Ms.')
 GROUP BY
       GROUPING SETS
       ((SALUTATION, SUBSTR(phone, 1,3)),
        (SALUTATION, TO_CHAR(registration_date, 'MON')),
        ()
       )
       SALUT Are Reg  COUNT(*)
       ----- --- --- ---------
       Mr.   201            43
       Mr.   212             2
       Mr.   718            89
       Ms.   201            32
       Ms.   212             3
       Ms.   718            65
       Mr.       FEB       107
       Mr.       JAN        27
       Ms.       FEB        81
       Ms.       JAN        19
                           234

       11 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

ANSWER: The query result shows students with the salutations Mr. and Ms. who live in area codes 201, 212, and 718. The GROUPING SETS extension to the GROUP BY clause allows you to selectively group specific information. There are three individual groups. The first set is salutation and area code and is indicated as a set with the enclosed parentheses. The next set is the registration month, and the salutation is again enclosed in parentheses. The final set is an empty set of parentheses, indicating the grand total. The GROUPING SETS clause is very efficient as it queries the table once to generate the result.

e) Write the individual queries to proof the numbers found in exercise d.

ANSWER: In exercise d, you already identified the three groupings, which then translate into three individual queries.

The following is query 1.

```
SELECT salutation, SUBSTR(phone, 1, 3), COUNT(*)
  FROM student
 WHERE SUBSTR(phone, 1,3) IN ('201','212','718')
   AND salutation IN ('Mr.', 'Ms.')
 GROUP BY salutation, SUBSTR(phone, 1, 3)
SALUT SUB  COUNT(*)
----- --- ----------
Mr.   201         43
Mr.   212          2
Mr.   718         89
Ms.   201         32
Ms.   212          3
Ms.   718         65

6 rows selected.
```

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The following is query #2.

```
SELECT salutation, TO_CHAR(registration_date, 'MON'),
       COUNT(*)
  FROM student
 WHERE SUBSTR(phone, 1,3) IN ('201','212','718')
   AND salutation IN ('Mr.', 'Ms.')
 GROUP BY salutation, TO_CHAR(registration_date, 'MON')
SALUT TO_  COUNT(*)
----- --- ---------
Mr.   FEB       107
Mr.   JAN        27
Ms.   FEB        81
Ms.   JAN        19

4 rows selected.
```

The following is query #3.

```
SELECT COUNT(*)
  FROM student
 WHERE SUBSTR(phone, 1,3) IN ('201','212','718')
   AND salutation IN ('Mr.', 'Ms.')
 COUNT(*)
---------
      234

1 row selected.
```


Lab 17.2 Quiz

In order to test your progress, you should be able to answer the following questions.

1) A query has the following GROUP BY clause. How many different groupings are visible in the result?

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
GROUP BY CUBE (color, price,
material, store_location)
```

_____**a)** 4

_____**b)** 5

_____**c)** 16

_____**d)** 24

_____**e)** Unknown

2) A query has the following GROUP BY clause. How many different groupings are visible in the result?

```
GROUP BY ROLLUP (color, price,
material, store_location)
```

_____**a)** 4

_____**b)** 5

_____**c)** 16

_____**d)** 24

_____**e)** Unknown

3) A return value of 1 from the GROUPING function indicates an aggregate row.

_____**a)** True

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

_____**b)** False

4) How many groups are generated by the following query?

```
GROUP BY GROUPING SETS((color,
price), material, store_location)
```

_____**a)** 3

_____**b)** 4

_____**c)** 5

_____**d)** 16

_____**e)** Unknown

ANSWERS APPEAR IN APPENDIX A.

801

802

WORKSHOP

The projects in this section are meant to prompt you to utilize all the skills you have acquired throughout this chapter. The answers to these projects can be found at the companion Web site to this book, located at **www.oraclesqlbyexample.com**.

1) Write the question for the following query and answer.

PRINTED BY: Monica Gonzalez <monicazalez1@hotmail.com>. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

```
SELECT COUNT(DECODE(SIGN(total_capacity-20),
             -1, 1, 0, 1)) "<=20",
       COUNT(DECODE(SIGN(total_capacity-21),
             0, 1, -1, NULL,
             DECODE(SIGN(total_capacity-30), -1, 1)))
             "21-30",
       COUNT(DECODE(SIGN(total_capacity-30), 1, 1)) "31+"
  FROM (SELECT SUM(capacity) total_capacity, course_no
          FROM SECTION
         GROUP BY COURSE_NO)
    <=20      21-30        31+
--------- --------- ---------
         2        10        16

1 row selected.
```

2) Using an analytical function, determine the top three zip codes where most of the students live.

3) Explain the result of the following query.

```
SELECT 'Q'||TO_CHAR(start_date_time, 'Q') qtr,
       TO_CHAR(start_date_time, 'DY') day, count(*),
       DENSE_RANK() OVER (
         PARTITION BY 'Q'||TO_CHAR(start_date_time, 'Q')
         ORDER BY COUNT(*) DESC) rank_qtr,
       DENSE_RANK() OVER (ORDER BY COUNT(*) DESC) rank_all
  FROM enrollment e, section s
 WHERE s.section_id = e.section_id
 GROUP BY 'Q'||TO_CHAR(start_date_time, 'Q'),
       TO_CHAR(start_date_time, 'DY')
 ORDER BY 1, 4
QT  DAY COUNT(*)   RANK_QTR   RANK_ALL
--- --- ---------- -------- ----------
Q2  MON         42        1          1
Q2  TUE         35        2          2
Q2  SAT         30        3          3
Q2  SUN         29        4          4
Q2  WED         15        5          6
Q2  FRI         13        6          7
Q2  THU         13        6          7
Q3  SAT         29        1          4
Q3  TUE         20        2          5

9 rows selected
```