CHAPTER 6 Aggregate Functions, GROUP BY, and HAVING Clauses

Oracle SQL By Example, Fourth Edition by Alice Rischert. Published by Prentice Hall.

CHAPTER OBJECTIVES

In this chapter, you will learn about:

- Aggregate Functions
- The GROUP BY and HAVING Clauses

In the last two chapters, you learned about character functions, number functions, date functions, and miscellaneous functions—all single-row functions. In this chapter, you will learn about aggregate functions, which work on groups of rows. The most commonly used aggregate functions are discussed. Aggregate functions allow you to generate summary data for a group of rows to obtain totals, averages, counts, minimum values, and maximum values. In Chapter 17, "Exploring Data Warehousing Features," you will learn about advanced SQL aggregation topics involving the ROLLUP and CUBE operators.

LAB 6.1 Aggregate Functions

LAB OBJECTIVES

After this lab, you will be able to:

- Use Aggregate Functions in a SQL Statement
- Understand the Effect of Nulls on Aggregate Functions

Aggregate functions do just as you would expect: They *aggregate*, or group together, data to produce a single result. Questions such as "How many students are registered?" and "What is the average cost of a course?" can be answered by using aggregate functions. You count the individual students to answer the first question, and you calculate the average cost of all courses to answer the second. In each case, the result is a single answer, based on several rows of data.

The COUNT Function

One of the most common aggregate functions is the COUNT function, which lets you count values in a table. The function takes a single parameter, which can be a column in a table of any data type and can even be the

asterisk (*) wildcard. The following SELECT statement returns the number of rows in the ENROLLMENT table:

```
SELECT COUNT(*)
  FROM enrollment
 COUNT(*)
---------
      226

1 row selected.
```

COUNT AND NULLS

The COUNT function is useful for determining whether a table has data. If the result returns the number 0 when you use COUNT(*), it means there are no rows in the table, even though the table exists.

Following is an example of the COUNT function used with a database column as a parameter. The difference is that COUNT(*) counts rows that contain null values, whereas COUNT with a column excludes rows that contain nulls.


```
SELECT COUNT(DISTINCT section_id), COUNT(section_id)
  FROM enrollment
COUNT(DISTINCTSECTION_ID) COUNT(SECTION_ID)
------------------------- -----------------
                       64               226

1 row selected.
```

The FINAL_GRADE column in the ENROLLMENT table allows null values, and there is only one row with a value in the FINAL_GRADE column. Therefore, the result of the function is 1. COUNT(section_id) returns the same number as COUNT(*) because the SECTION_ID column contains no nulls.

COUNT AND DISTINCT

DISTINCT is often used in conjunction with aggregate functions to determine the number of distinct values. There are 226 rows in the ENROLLMENT table but 64 distinct section IDs. Several students are enrolled in the same section; therefore, individual section IDs usually exist more than once in the ENROLLMENT table.

```
SELECT COUNT(DISTINCT section_id), COUNT(section_id)
  FROM enrollment
COUNT(DISTINCTSECTION_ID) COUNT(SECTION_ID)
------------------------- -----------------
                       64               226

1 row selected.
```

You can use the UNIQUE keyword in place of DISTINCT.

The SUM Function

The SUM function adds values together for a group of rows. The following example adds up all the values in the CAPACITY column of the SECTION table. The result is the total capacity of all sections. If any value in the CAPACITY column contains a null, these values are ignored.

```
SELECT SUM(capacity)
  FROM section
SUM(CAPACITY)
-------------
         1652

1 row selected.
```


The AVG Function

The AVG function returns the average of a group of rows. The following example computes the average capacity of each section. Any nulls in the capacity column are ignored. To substitute a zero for a null, use the NVL or COALESCE function, as discussed in Chapter 4, “Character, Number, and Miscellaneous Functions.”

```
SELECT AVG(capacity), AVG(NVL(capacity,0))
  FROM section
AVG(CAPACITY) AVG(NVL(CAPACITY,0))
------------- --------------------
    21.179487            21.179487

1 row selected.
```

In this example, there are no sections with null values in the CAPACITY column; therefore, the results of the two functions are identical.

The MIN and MAX Functions

The MIN and MAX functions are opposites of each other, providing the minimum and maximum values, respectively, in a group of rows. The result shows the lowest value in the CAPACITY column in the SECTION table; this value is 10, and the highest value is 25.

```
SELECT MIN(capacity), MAX(capacity)
  FROM section
MIN(CAPACITY) MAX(CAPACITY)
------------- -------------
           10            25

1 row selected.
```

MIN and MAX with Different Data Types

The previous example operates on the CAPACITY column, which is of the NUMBER data type. The MIN and MAX functions can take other data types as parameters. The next example shows the use of these functions with the DATE data type and displays the first and last registration dates in the STUDENT table.

```
SELECT MIN(registration_date) "First",
       MAX(registration_date) "Last"
  FROM student
First     Last
--------- ---------
22-JAN-07 23-FEB-07

1 row selected.
```


A less frequently used data type for the MIN and MAX functions is the VARCHAR2 data type. The following query shows the minimum or maximum value of the DESCRIPTION column and returns the first and last values, in an alphabetized list of values.

```
SELECT MIN (description) AS MIN, MAX (description) AS MAX
  FROM course
MIN                       MAX
------------------------- -----------------------
Advanced Java Programming  Unix Tips and Techniques

1 row selected.
```

The capital letter A is equal to the ASCII value 65, B is 66, and so on. Lowercase letters, numbers, and characters all have their own ASCII values. Therefore, MIN and MAX can be used to evaluate a character's respective first and last letters, in alphabetical order.

Aggregate Functions and Nulls

Except for the COUNT(*) function, all the aggregate functions you have learned about so far ignore null values. You use the NVL or COALESCE function to substitute for any null values. An aggregate function always returns a row. Even if the query returns no rows, the result is simply one row with a null value; the COUNT function always returns either a zero or a number.

Aggregate Functions and CASE

Placing a CASE expression within an aggregate function can be useful if you want to manipulate or select specific values before applying the aggregate function. For example, the following SQL statement shows the computation of the average course cost. If the value in the PREREQUISITE column is null, the value in the COST column is multiplied by 1.1; if the value is equal to 20, it is multiplied by 1.2; in all other cases, the value retrieved in the COST column remains unchanged.

```
SELECT AVG(CASE WHEN prerequisite IS NULL THEN cost*1.1
                WHEN prerequisite = 20 THEN cost*1.2
                ELSE cost
           END) AS avg
  FROM course
       AVG
----------
1256.13793

1 row selected.
```

Aggregate Function Syntax

Table 6.1 lists the most commonly used aggregate functions and their corresponding syntax. As you may notice, you can use the DISTINCT keyword with all these functions to evaluate only the distinct (or unique) values. The ALL keyword is the default option and evaluates all rows. The DISTINCT keyword is really useful only for the AVG, SUM, and COUNT functions. In place of DISTINCT, you can substitute the UNIQUE keyword.

TABLE 6.1 Commonly Used Aggregate Functions

FUNCTION	PURPOSE
COUNT({* | [DISTINCT|ALL])	Counts the number of rows. The wildcard (*) option includes expression) duplicates and null values.
SUM([DISTINCT|ALL] value)	Computes the total of a value; ignores nulls.
AVG([DISTINCT|ALL] value)	Finds the average value; ignores nulls.
MIN(expression)	Determines the minimum value of an expression; ignores nulls.
MAX(expression)	Determines the maximum value of an expression; ignores nulls.

LAB 6.1 EXERCISES

a) Write a SELECT statement that determines how many courses do not have a prerequisite.

b) Write a SELECT statement that determines the total number of students enrolled. Count each student only once, no matter how many courses the student is enrolled in.

c) Determine the average cost for all courses. If the course cost contains a null value, substitute the value 0.

d) Write a SELECT statement that determines the date of the most recent enrollment.

LAB 6.1 EXERCISE ANSWERS

a) Write a SELECT statement that determines how many courses do not have a prerequisite.

ANSWER: The COUNT function is used to count the number of rows in the COURSE table where the values in the PREREQUISITE column are null.

```
SELECT COUNT(*)
  FROM course
 WHERE prerequisite IS NULL
 COUNT(*)
---------
         4

1 row selected.
```

b) Write a SELECT statement that determines the total number of students enrolled. Count each student only once, no matter how many courses the student is enrolled in.

ANSWER: Use DISTINCT in conjunction with the COUNT function to count distinct students, regardless of how many times they appear in the ENROLLMENT table.

```
SELECT COUNT(DISTINCT student_id)
  FROM enrollment
COUNT(DISTINCTSTUDENT_ID)
-------------------------
                      165

1 row selected.
```

c) Determine the average cost for all courses. If the course cost contains a null value, substitute the value 0.

ANSWER: Both the NVL function and the COALESCE function substitute any null value with a zero. The NVL or COALESCE function must be nested inside the AVG function.

```
SELECT AVG(NVL(cost, 0))
  FROM course
AVG(NVL(COST,0))
----------------
          1158.5

1 row selected.
```

The following is another possibility.

```
SELECT AVG(COALESCE(cost, 0))
  FROM course
```

If you do not substitute the nulls for the zero value, the average course cost returns a different, more accurate, result.

```
SELECT AVG(cost)
  FROM course
AVG(COST)
---------
1198.4483

1 row selected.
```

d) Write a SELECT statement that determines the date of the most recent enrollment.

ANSWER: The MAX function determines the most recent value in the ENROLL_DATE column of the ENROLLMENT table.

```
SELECT MAX(enroll_date)
  FROM enrollment
MAX(ENROL
---------
21-FEB-07

1 row selected.
```


Lab 6.1 Quiz

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

1) How many of these functions are aggregate functions? AVG, COUNT, SUM, ROUND.

   a) _____One

   b) _____Two

   c) _____Three

   d) _____Four

2) Which problem does the following SQL statement solve?

```
SELECT NVL(MAX(modified_date),
   TO_DATE('12-MAR-2012', 'DD-MON-
YYYY'))
 FROM enrollment
```

a) _____Display the date when a STUDENT table was last modified.

b) _____Display the date when a STUDENT record was last modified. Replace any null value with the date March 12, 2012.

c) _____Show the date when a record in the ENROLLMENT table was last modified. If the result returns a null value, display March 12, 2012.

d) _____For all the ENROLLMENT records, show the date 12-Mar-2012.

3) An aggregate function can be applied on a single row.

a) _____True

b) _____False

4) The following SQL statement contains an error.

```
SELECT AVG(*)
 FROM course
```

a) _____True

b) _____False

5) The following SQL statement determines the average of all capacities in the SECTION table.

```
SELECT AVG(UNIQUE capacity)
  FROM section
```

a) _____True

b) _____False

6) The following SQL statement contains an error.

```
SELECT SUM(capacity*1.5)
  FROM section
```

a) _____True

b) _____False

ANSWERS APPEAR IN APPENDIX A.

LAB 6.2 The GROUP BY and HAVING Clauses

LAB OBJECTIVES

After this lab, you will be able to:

- Use the GROUP BY Clause
- Apply the HAVING Clause

The GROUP BY and HAVING clauses allow you to categorize and aggregate data further. This lab analyzes the data in the STUDENT schema tables, using various aggregation functions.

The GROUP BY Clause

The GROUP BY clause determines how rows are grouped. The aggregate function together with the GROUP BY clause shows the aggregate value for each group. For example, for all the different locations, you can determine the number of rows or the average, minimum, maximum, or total capacity.

To understand the result of the GROUP BY clause without an aggregate function, compare it to that of the DISTINCT clause.

The following two queries return the same result, which is a distinct list of the values in the LOCATION column.

```
SELECT DISTINCT location
  FROM section

SELECT location
  FROM section
 GROUP BY location
LOCATION
--------
H310
L206
...
M311
M500

12 rows selected.
```

If you want to expand on this example and now include how many times each respective location value is listed in the SECTION table, you add the COUNT(*) function in the query.

```
SELECT location, COUNT(*)
  FROM section
 GROUP BY location
LOCATION   COUNT(*)
-------- ----------
H310              1
L206              1
L210             10
L211              3
L214             15
...
M500              1

12 rows selected.
```

Essentially, the GROUP BY clause and the aggregate function work hand-in-hand. Based on the distinct values, as listed in the GROUP BY clause, the aggregate function returns the result.

You can change the SQL query to determine other values for the distinct LOCATION column. For example, the following statement includes the aggregate functions SUM, MIN, and MAX in the SELECT list. For each distinct location, you see the total capacities with the

SUM function, which adds up all the values in the CAPACITY column. The MIN and MAX functions return the minimum and maximum capacities for each respective location.

```
SELECT location, COUNT(*), SUM(capacity) AS sum,
       MIN(capacity) AS min, MAX(capacity) AS max
  FROM section
 GROUP BY location
```

LOCATION	COUNT(*)	SUM	MIN	MAX
H310	1	15	15	15
L206	1	15	15	15
L210	10	200	15	25
L211	3	55	15	25
L214	15	275	15	25
...				
M500	1	25	25	25

12 rows selected.

You can validate the result of the query by looking at one of the rows. For example, the row with the LOCATION value L211 has three rows, according to the COUNT function. The total of all the values in the CAPACITY column is 55 (25 + 15 + 15). The minimum value of the CAPACITY column is 15, and the maximum value is 25.


```
SELECT location, capacity, section_id
  FROM section
 WHERE location = 'L211'
```

LOCATION	CAPACITY	SECTION_ID
L211	25	119
L211	15	133
L211	15	153

3 rows selected.

GROUPING BY MULTIPLE COLUMNS

The following query applies the aggregate functions to the distinct values of the LOCATION and the INSTRUCTOR_ID columns; therefore, the statement returns more rows than the previous GROUP BY query.

```
SELECT location, instructor_id,
       COUNT(*), SUM(capacity) AS sum,
       MIN(capacity) AS min, MAX(capacity) AS max
  FROM section
 GROUP BY location, instructor_id
 ORDER BY 1
```

LOCATION	INSTRUCTOR_ID	COUNT(*)	SUM	MIN	MAX
H310	103	1	15	15	15
L206	108	1	15	15	15
L210	101	1	15	15	15
L210	103	2	40	15	25
L210	104	1	25	25	25
L210	105	2	40	15	25
L210	106	1	25	25	25
L210	108	3	55	15	25
L214	102	4	70	15	25
...					
M500	102	1	25	25	25

39 rows selected.

When you examine the output, you notice that there are six rows for the L210 location. For this location, each row has a different INSTRUCTOR_ID value. On each

of these six distinct LOCATION and INSTRUCTOR_ID combinations, the aggregate functions are applied. For example, the first row has only one row with this LOCATION and INSTRUCTOR_ID combination, and the second row has two rows, as you see from the number in the COUNT(*) column. Once again, you can validate the result by issuing an individual query against the SECTION table.

273

274

```
SELECT location, instructor_id, capacity, section_id
  FROM section
 WHERE location = 'L210'
 ORDER BY 1, 2
```

LOCATION	INSTRUCTOR_ID	CAPACITY	SECTION_ID
L210	101	15	117
L210	103	15	81
L210	103	25	150
L210	104	25	96
L210	105	25	91
L210	105	15	129
L210	106	25	84
L210	108	15	86
L210	108	15	155
L210	108	25	124

10 rows selected.

ORACLE ERROR ORA-00979

Every column you list in the SELECT list, except the aggregate function column itself, must be repeated in the GROUP BY clause. Following is the error Oracle returns when you violate this rule.

```
SQL> SELECT location,
instructor_id,
  2 COUNT(*), SUM(capacity) AS sum,
  3 MIN(capacity) AS min,
MAX(capacity) AS max
  4 FROM section
  5 GROUP BY location
  6 /
SELECT location, instructor_id,
*ERROR at line 1:
ORA-00979: not a GROUP BY
expression
```

The error message indicates that Oracle does not know how to process this query. The query lists the LOCATION and INSTRUCTOR_ID columns in the SELECT list but only the LOCATION column in the GROUP BY clause. Essentially, Oracle is confused about the SQL statement. The GROUP BY clause lists only the LOCATION column, which determines the

distinct values. But the statement fails to specify what to do with the INSTRUCTOR_ID column.

SORTING DATA

The GROUP BY clause groups the rows, but it does not necessarily sort the results in any particular order. To change the order, use the ORDER BY clause, which follows the GROUP BY clause. The columns used in the ORDER BY clause must appear in the SELECT list, which is unlike the normal use of ORDER BY. In the following example, the result is sorted in descending order by the total capacity. You can use the column alias in the ORDER BY clause.

274

275

```
SELECT location ”Location”,
instructor_id,
   COUNT(location) ”Total
Locations”,
   SUM(capacity) ”Total Capacity”
  FROM section
GROUP BY location, instructor_id
ORDER BY ”Total Capacity” DESC
```

The HAVING Clause

The purpose of the HAVING clause is to eliminate groups, just as the WHERE clause is used to eliminate

rows. Expanding on the previous SQL statement, the following query showing the applied HAVING clause restricts the result set to locations with a total capacity value of more than 50 students.

```
SELECT location "Location", instructor_id,
       COUNT(location) "Total Locations",
       SUM(capacity) "Total Capacity"
  FROM section
 GROUP BY location, instructor_id
HAVING SUM(capacity) > 50j277
 ORDER BY "Total Capacity" DESC
```

Location	INSTRUCTOR_ID	Total Locations	Total Capacity
L509	106	5	115
L509	101	4	85
L507	101	3	75
L507	107	3	75
L509	105	3	75
...			
L214	106	3	55

14 rows selected.

THE WHERE AND HAVING CLAUSES

As previously mentioned, the HAVING clause eliminates groups that do not satisfy its condition. This is in contrast to the WHERE clause, which eliminates rows even before the aggregate functions and the GROUP BY and HAVING clauses are applied.

```
SELECT location "Location", instructor_id,
       COUNT(location) "Total Locations",
       SUM(capacity) "Total Capacity"
  FROM section
 WHERE section_no IN (2, 3)
 GROUP BY location, instructor_id
HAVING SUM(capacity) > 50
Location INSTRUCTOR_ID Total Locations Total Capacity
-------- ------------ --------------- --------------
L214                104               3             55

1 row selected.
```

275

276

The WHERE clause is executed by the database first, narrowing the result set to rows in the SECTION table where SECTION_NO equals either 2 or 3 (that is, the second or third section of a course). The next step is to group the results by the columns listed in the GROUP BY clause and to apply the aggregate functions. Finally, the HAVING condition is tested against the groups. Only rows with a total capacity of greater than 50 are returned in the result.

MULTIPLE CONDITIONS IN THE HAVING CLAUSE

The HAVING clause can use multiple operators to further eliminate any groups, as in the following example. Either the columns used in the HAVING clause must be found in the GROUP BY clause or they

must be aggregate functions. In the following example, the aggregate COUNT function is not mentioned in the SELECT list, yet the HAVING clause refers to it. The second condition of the HAVING clause chooses only location groups with a starting value of L5. In this particular example, it is preferable to move this condition to the WHERE clause because doing so eliminates the rows even before the groups are formed, and the statement will therefore execute faster.

```
SELECT location "Location",
       SUM(capacity) "Total Capacity"
  FROM section
 WHERE section_no = 3
 GROUP BY location
HAVING (COUNT(location) > 3
        AND location LIKE 'L5%')
```

```
Location   Total Capacity
---------- --------------
L507                  100
L509                  175

2 rows selected.
```

CONSTANTS AND FUNCTIONS WITHOUT PARAMETERS

Any constant, such as a text or number literal or a function that does not take any parameters, such as the SYSDATE function, may be listed in the SELECT list without being repeated in the GROUP BY clause. This does not cause the ORA-00979 error message. The following query shows the text literal 'Hello', the number literal 1, and the SYSDATE function in the SELECT list of the query. These expressions do not need to be repeated in the GROUP BY clause.

```
SELECT 'Hello', 1, SYSDATE,
course_no "Course #",
   COUNT(*)
  FROM section
 GROUP BY course_no
HAVING COUNT(*) = 5
```



```
SELECT MAX(COUNT(*))
  FROM enrollment
 GROUP BY section_id
MAX(COUNT(*))
-------------
           12

1 row selected.
```

THE ORDER OF THE CLAUSES

The HAVING clause can appear before the GROUP BY clause, as shown in the following example, but this is rarely seen in practice.

```
SELECT course_no "Course #",
   AVG(capacity) "Avg. Capacity",
   ROUND(AVG(capacity)) "Rounded
Avg. Capacity"
  FROM section
HAVING COUNT(*) = 2
 GROUP BY course_no
```

Nesting Aggregate Functions

Aggregate functions can be nested, as in the following example. The query returns the largest number of students that enrolled in an individual section. The COUNT function determines a count for all the sections, based on the GROUP BY clause, which lists SECTION_ID. When the MAX function is applied against this result, it returns 12 as the largest of the values. In other words, 12 students is the largest number of students enrolled in an individual section.

```
SELECT MAX(COUNT(*))
  FROM enrollment
 GROUP BY section_id
MAX(COUNT(*))
-------------
           12

1 row selected.
```

Taking Aggregate Functions and Groups to the Next Level

Each of the SQL statements described so far in this chapter has focused on a single table. You will learn how to avoid potential pitfalls when joining tables and applying aggregate functions in Chapter 8, "Subqueries."

In Chapter 17, you will learn about many additional aggregate and analytical functions, such as RANK, that allow you to analyze data even further. Some of this functionality helps you avoid the ORA-00979 error through the use of a special analytical clause. Chapter 17 also introduces the ROLLUP and CUBE operators so that you can perform multilevel aggregations.

277

LAB 6.2 EXERCISES

a) Show a list of prerequisites and count how many times each appears in the COURSE table. Order the result by the PREREQUISITE column.

b) Write a SELECT statement that shows student IDs and the number of courses each student is enrolled in. Show only those enrolled in more than two classes.

c) Write a SELECT statement that displays the average room capacity for each course. Display the average, expressed to the nearest whole number, in another column. Use a column alias for each column selected.

d) Write the same SELECT statement as in Exercise c, except consider only courses with exactly two sections. Hint: Think about the relationship between the COURSE and SECTION tables—specifically, how many times a course can be represented in the SECTION table.

LAB 6.2 EXERCISE ANSWERS

a) Show a list of prerequisites and count how many times each appears in the COURSE table. Order the result by the PREREQUISITE column.

ANSWER: The COUNT function and GROUP BY clause are used to count distinct prerequisites. The last row of the result set shows the number of prerequisites with a null value.

```
SELECT prerequisite, COUNT(*)
  FROM course
 GROUP BY prerequisite
 ORDER BY prerequisite
PREREQUISITE  COUNT(*)
------------ ---------
          10         1
          20         5
...
         350         2
         420         1
                     4

17 rows selected.
```

NULLS AND THE GROUP BY CLAUSE

If there are null values in a column, and you group on the column, all the null values are considered equal.

This is different from the typical handling of nulls, where one null is not equal to another. The aforementioned query and result show that there are four null prerequisites. The nulls always appear last in the default ascending sort order.

You can change the default ordering of the nulls with the NULLS FIRST option in the ORDER BY clause, as shown in the following statement. When you look at the result set, you see that the nulls are now first, followed by the default ascending sort order.


```
SELECT prerequisite, COUNT(*)
  FROM course
 GROUP BY prerequisite
 ORDER BY prerequisite NULLS FIRST
PREREQUISITE  COUNT(*)
------------ ---------
                    4
          10        1
...
         350        2
         420        1

17 rows selected.
```

b) Write a SELECT statement that shows student IDs and the number of courses each student is enrolled in. Show only those enrolled in more than two classes.

ANSWER: To obtain the distinct students, use the STUDENT_ID column in the GROUP BY clause. For each of the groups, use the COUNT function to count records for each student. Use the HAVING clause to include only those students enrolled in more than two sections.

```
SELECT student_id, COUNT(*)
  FROM enrollment
 GROUP BY student_id
HAVING COUNT(*) > 2
STUDENT_ID  COUNT(*)
---------- ---------
       124         4
       184         3
...
       238         3
       250         3

7 rows selected.
```

c) Write a SELECT statement that displays the average room capacity for each course. Display the average, expressed to the nearest whole number, in another column. Use a column alias for each column selected.

ANSWER: The SELECT statement uses the AVG function and the ROUND function. The GROUP

BY clause ensures that the average capacity is displayed for each course.


```
SELECT course_no "Course #",
       AVG(capacity) "Avg. Capacity",
       ROUND(AVG(capacity)) "Rounded Avg. Capacity"
  FROM section
 GROUP BY course_no
Course # Avg Capacity Rounded Avg Capacity
-------- ------------ ---------------------
      10           15                    15
      20           20                    20
      25    22.777778                    23
...
     350    21.666667                    22
     420           25                    25
     450           25                    25

28 rows selected.
```

The SQL statement uses nested functions. Nested functions always work from the inside out, so the AVG(capacity) function is evaluated first, and its result is the parameter for the ROUND function. ROUND's optional precision parameter is not used, so the result of AVG(capacity) rounds to a precision of zero, or no decimal places.

COLUMN ALIASES IN THE GROUP BY CLAUSE

Sometimes you might copy the columns from a SELECT list—with the exception of the aggregate function, of course—down to the GROUP BY clause. After all, cutting and pasting saves a lot of typing. You might then end up with an error such as the following one. The ORA-00933 error message ("SQL command not properly ended") may leave you clueless as to how to solve the problem.

```
SELECT course_no "Course #",
   AVG(capacity) "Avg. Capacity",
   ROUND(AVG(capacity)) "Rounded
Avg. Capacity"
  FROM section
/
 GROUP BY course_no "Course #"
  *
ERROR at line 5:
ORA-00933: SQL command not properly
ended
```

To resolve the error, you must exclude the column aliase **"Course #"**in the GROUP BY clause.

Column aliases are not allowed in the GROUP BY clause.

d) Write the same SELECT statement as in Exercise c, except consider only courses with exactly two sections. Hint: Think about the relationship between the COURSE and SECTION tables—specifically, how many times a course can be represented in the SECTION table.

ANSWER: The HAVING clause is added to limit the result set to courses that appear exactly twice.

```
SELECT course_no "Course #",
   AVG(capacity) "Avg. Capacity",
   ROUND(AVG(capacity)) "Rounded
Avg. Capacity"
 FROM section
  GROUP BY course_no
HAVING COUNT(*) = 2
```



```
 Course # Avg. Capacity Rounded Avg. Capacity
--------- ------------- ---------------------
      132            25                    25
      145            25                    25
      146            20                    20
      230          13.5                    14
      240          12.5                    13

5 rows selected.
```

The COUNT(*) function in the HAVING clause does not appear as part of the SELECT list. You can include in the HAVING clause any aggregate function, even if this aggregate function is not mentioned in the SELECT list.

Lab 6.2 Quiz

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

1) Which column(s) must be included in the GROUP BY clause of the following SELECT statement?

```
SELECT NVL(MAX(final_grade),0),
section_id,
   MAX(created_date)
 FROM enrollment
GROUP BY ________
```

a) _____final_grade

b) _____section_id

c) _____created_date

d) _____All three

e) _____None of the above

2) You can combine DISTINCT and a GROUP BY clause in the same SELECT statement.

a) _____True

b) _____False

3) There is an error in the following SELECT statement.

```
SELECT COUNT(student_id)
  FROM enrollment
WHERE COUNT(student_id) > 1
```

a) _____True

b) _____False

4) How many rows in the following SELECT statement will return a null prerequisite? SELECT prerequisite, COUNT(*)

```
FROM course
  WHERE prerequisite IS NULL
GROUP BY prerequisite
```

a) _____None

b) _____One

c) _____Multiple

5) Where is the error in the following SELECT statement?

```
SELECT COUNT(*)
  FROM section
GROUP BY course_no
```

a) _____There is no error.

b) _____Line 1.

c) _____Line 2.

d) _____Line 3.

ANSWERS APPEAR IN APPENDIX A.

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) List the order in which the WHERE, GROUP BY, and HAVING clauses are executed by the database in the following SQL statement.

```
SELECT section_id, COUNT(*),
final_grade
```

```
FROM enrollment
WHERE TRUNC(enroll_date) >
TO_DATE('2/16/2007', 'MM/DD/
YYYY')
GROUP BY section_id, final_grade
HAVING COUNT(*) > 5
```

2) Display a count of all the different course costs in the COURSE table.

3) Determine the number of students living in zip code 10025.

4) Show all the different companies for which students work. Display only companies in which more than four students are employed.

5) List how many sections each instructor teaches.

6) What problem does the following statement solve?

```
SELECT COUNT(*), start_date_time,
location
FROM section
GROUP BY start_date_time,
location
HAVING COUNT(*) > 1
```

7) Determine the highest grade achieved for the midterm within each section.

8) A table called CUSTOMER_ORDER contains 5,993 rows, with a total order amount of $10,993,333.98, based on the orders from 4,500 customers. Given this scenario, how many rows does the following query return?

```
SELECT SUM(order_amount) AS
"Order Total"
FROM customer_order
```