CHAPTER 8 Subqueries

CHAPTER OBJECTIVES

In this chapter, you will learn about:

- Simple Subqueries
- Correlated Subqueries
- Inline Views and Scalar Subquery Expressions
- ANY, SOME, and ALL Operators in Subqueries

A subquery is a SELECT statement nested in various clauses of a SQL statement. It allows you to use the output from one query as the input of another SQL statement. Subqueries make it easy to break down problems into logical and manageable pieces.

LAB 8.1 Simple Subqueries

LAB OBJECTIVES

After this lab, you will be able to:

- Write Subqueries in the WHERE and HAVING Clauses
- Write Subqueries That Return Multiple Rows
- Write Subqueries That Return Multiple Columns

As mentioned earlier, a subquery allows you to break down a problem into individual components and solve it by nesting the queries. Although subqueries can be nested several levels deep, nesting beyond four or five levels is impractical. Subqueries are sometimes also referred to as *sub-SELECTs* or *nested SELECTs*.

Subqueries are not used just in SELECT statements but also in other SQL statements that allow subqueries (for example, the WHERE clause of DELETE statements, the SET and WHERE clauses of UPDATE statements, or part of the SELECT clause of INSERT statements). You use these SQL statements in Chapter 11, "Insert, Update, and Delete."

In this book, a *subquery* is referred to as an *inner query*, and the surrounding statement is known as the *outer query*. In a simple subquery, the inner query is executed once, before the execution of the outer query. (This is in contrast to the correlated subquery, where the inner

query executes repeatedly. You will learn to write correlated subqueries in Lab 8.2.)

Scalar Subqueries

A *scalar subquery*, also called a *single-row subquery*, returns a single column with one row. When you want to show the courses with the lowest course cost, you can write two separate queries. First, determine the lowest cost by applying the aggregate function MIN to the COST column of the COURSE table.

```
SELECT MIN(cost)
  FROM course
MIN(COST)
---------
     1095

1 row selected.
```


Then write another SELECT statement that retrieves courses equaling the cost.

```
SELECT course_no, description, cost
  FROM course
 WHERE cost = 1095
COURSE_NO DESCRIPTION                              COST
--------- ---------------------------------------- ----
      135 Unix Tips and Techniques                 1095
      230 Intro to the Internet                    1095
      240 Intro to the BASIC Language              1095

3 rows selected.
```

The subquery construct simplifies the writing of two separate queries and the recording of the intermediate result. The following SQL statement nests the subquery in the WHERE clause of the outer query. The inner query, which is the query that determines the lowest cost from the COURSE table, is executed first. The result is fed to the outer query, which retrieves all the values that qualify.

```
SELECT course_no, description, cost
  FROM course
 WHERE cost =
       (SELECT MIN(cost)
          FROM course)
COURSE_NO DESCRIPTION                              COST
--------- ---------------------------------------- ----
      135 Unix Tips and Techniques                 1095
      230 Intro to the Internet                    1095
      240 Intro to the BASIC Language              1095

3 rows selected.
```

Instead of performing equality conditions, you might need to construct >, <, >=, <=, or <> comparisons against a result. Just like the aforementioned statement, these comparisons will work only if the subquery returns a single row.

Subqueries That Return Multiple Rows

Subqueries can return one or multiple rows. If a subquery returns a single row, the =, <, >, <=, >=, or <> operator can be used for comparison with the subquery. If multiple records are returned, the IN, ANY, ALL, or SOME operator must be used; otherwise, Oracle returns an error message.

The following query displays the course number, description, and cost of courses with a cost equal to the highest cost of all the courses. The highest cost requires the use of the aggregate function MAX. As discussed in Chapter 6, "Aggregate Functions, GROUP BY, and HAVING Clauses," aggregate functions, when used alone, without the presence of any nonaggregate expressions in the SELECT list, always return one row. The subquery returns the single value 1595. All the rows of the COURSE table are compared to this value to see if any rows have the same course cost. Only one record in the COURSE table equals this cost.

325 326

```
SELECT course_no, description, cost
  FROM course
 WHERE cost =
       (SELECT MAX(cost)
          FROM course)
COURSE_NO DESCRIPTION                       COST
--------- --------------------------------- ------
       80 Programming Techniques            1595

1 row selected.
```

The next SQL statement is an example of a subquery that returns several rows.

```
SELECT course_no, description, cost
FROM course
WHERE cost =
   (SELECT cost
   FROM course
   WHERE prerequisite = 20)
ERROR at line 4:ORA-01427: single-row subquery returns more than one row
```

Multiple rows of the subquery satisfy the criteria of a prerequisite course number equal to 20. Therefore, Oracle returns an error message. To eliminate the error, change the = operator of the outer query to the IN operator. The IN operator compares a list of values for equivalency. If any of the values in the list satisfy the condition, the record is included in the result set.

```
SELECT course_no, description, cost
  FROM course
 WHERE cost IN
       (SELECT cost
          FROM course
         WHERE prerequisite = 20)
COURSE_NO DESCRIPTION                       COST
--------- ---------------------------------- ----
       10 Technology Concepts                1195
       20 Intro to Information Systems       1195
...
      122 Intermediate Java Programming      1195
      100 Hands-On Windows                   1195

25 rows selected.
```

You can also negate the criteria of the subquery and include only records with values that are not in the subquery's result. You accomplish this by applying the NOT IN operator.


```
SELECT course_no, description, cost
  FROM course
 WHERE cost NOT IN
       (SELECT cost
          FROM course
         WHERE prerequisite = 20)
COURSE_NO DESCRIPTION                              COST
--------- ---------------------------------------- ----
       80 Programming Techniques                   1595
      135 Unix Tips and Techniques                 1095
      230 Intro to the Internet                    1095
      240 Intro to the BASIC Language              1095

4 rows selected.
```

If the subquery returns multiple rows and you want to perform a comparison other than equality or inequality, use the ALL, ANY, and SOME operators, discussed in Lab 8.4, to perform such comparisons.

Table 8.1 provides an overview of the various comparison operators available for subqueries. If your

subquery returns more than one row, you have to choose a different operator than if your subquery retrieves at most one row only.

TABLE 8.1 Comparison Operators for Subqueries

COMPARISON OPERATOR	SUBQUERY RETURNS ONE ROW	SUBQUERY RETURNS MULTIPLE ROWS
Equality	=	IN.
Inequality	<>	NOT IN.
Greater than	>	Use the ANY, ALL, and SOME operators
Less than	<	(see Lab 8.4).
Greater than or equal to	>=	
Less than or equal to	<=	

Nesting Multiple Subqueries

You can nest one subquery within another subquery. The innermost query is always evaluated first, then the next highest one, and so on. The result of each subquery is fed into the enclosing statement.

The next query determines the last and first names of students enrolled in section number 8 of course number 20.


```
SELECT last_name, first_name
  FROM student
 WHERE student_id IN
       (SELECT student_id
          FROM enrollment
         WHERE section_id IN
               (SELECT section_id
                  FROM section
                 WHERE section_no = 8
                   AND course_no = 20))
```

```
LAST_NAME                      FIRST_NAME
------------------------------ ----------
Limate                         Roy
Segall                         J.

2 rows selected.
```

The innermost nested subquery, the last subquery in the example, is executed first; it determines the SECTION_ID for section number 8 and course number 20. The surrounding query uses this resulting SECTION_ID in the WHERE clause to select student

IDs from the ENROLLMENT table. These STUDENT_ID rows are fed to the outermost SELECT statement, which then displays the first and last names from the STUDENT table.

Subqueries and Joins

A subquery that uses the IN or = operator can often be expressed as an equijoin if the subquery does not contain an aggregate function. The following query can be transformed into an equijoin.

```
SELECT course_no, description
  FROM course
 WHERE course_no IN
       (SELECT course_no
          FROM section
         WHERE location = 'L211')
COURSE_NO DESCRIPTION
--------- ------------------------------
      142 Project Management
      125 Java Developer I
      122 Intermediate Java Programming

3 rows selected.
```

The following is the same query now expressed as an equijoin.

```
SELECT c.course_no, c.description
  FROM course c, section s
 WHERE c.course_no = s.course_no
   AND s.location = 'L211'
```


Subqueries That Return Multiple Columns

SQL allows you to compare multiple columns in the WHERE clause to multiple columns of a subquery. The values in the columns must match both sides of the equation in the WHERE clause for the condition to be true. This means the data type must be compatible and the number and order of columns must match.

For example, for each section, determine the students with the highest grades for their project (PJ). The following query does not accomplish this goal. It returns the highest project grade for each section but does not list the individual student(s).

```
SELECT section_id, MAX(numeric_grade)
  FROM grade
 WHERE grade_type_code = 'PJ'
 GROUP BY section_id
SECTION_ID MAX(NUMERIC_GRADE)
---------- ------------------
        82                 77
        88                 99
...
       149                 83
       155                 92

8 rows selected.
```

The following query obtains the desired result by transforming the query into a subquery. The outer query displays the desired STUDENT_ID column, and the WHERE clause compares the column pairs against the column pairs in the subquery.

```
SELECT student_id, section_id, numeric_grade
  FROM grade
 WHERE grade_type_code = 'PJ'
   AND (section_id, numeric_grade) IN
       (SELECT section_id, MAX(numeric_grade)
          FROM grade
         WHERE grade_type_code = 'PJ'
         GROUP BY section_id)
STUDENT_ID SECTION_ID NUMERIC_GRADE
---------- ---------- -------------
       245         82            77
       166         88            99
...
       232        149            83
       105        155            92

8 rows selected.
```

The execution steps are just like those for the previous simple subqueries. First, the innermost query determines the highest grade for each section. Then the pairs of columns are compared. If the column pair matches, Oracle displays the record.

If you were to write the query using literals instead of a subquery, the query would look like the following. It shows column pairs whereby the values of each expression pair is surrounded by parentheses.

```
SELECT student_id, section_id, numeric_grade
  FROM grade
 WHERE grade_type_code = 'PJ'
   AND (section_id, numeric_grade) IN
       ((82, 77),
       (88, 99),
...
       (149, 83),
       (155, 92))
```

Subqueries and Nulls

One easily overlooked behavior of subqueries is the occurrence of null values. The next example illustrates this concept on the COURSE table and the PREREQUISITE column. The first query shows a subquery that returns all the COURSE_NO and PREREQUISITE column values for courses with COURSE_NO 120, 220, and 310 in the COURSE table. Course number 310 has a null value in the

PREREQUISITE column, meaning that the individual course does not have any prerequisites.

```
SELECT course_no, prerequisite
  FROM course
 WHERE course_no IN (120, 220, 310)
COURSE_NO PREREQUISITE
--------- ------------
      120           80
      220           80
      310

3 rows selected.
```

If you use this result to formulate a subquery for these rows specifically and negate it with NOT, you obtain an interesting result: The outer query does not return any rows, despite the fact that there are rows with PREREQUISITE column values other than 80 and null.

```
SELECT course_no, prerequisite
FROM course
WHERE prerequisite NOT IN
(SELECT prerequisite
FROM course
WHERE course_no IN (310, 220))
no rows selected
```


If you translate the result of the subquery into a list of values, you see the same result: No rows are returned

from the query because the condition evaluates to unknown when any value in the list is a null.

```
SELECT course_no, prerequisite
  FROM course
 WHERE prerequisite NOT IN (80,
                            NULL)
```

```
no rows selected
```

You typically come across this type of scenario only in subqueries; therefore, you must be aware of any NOT IN operator subqueries that can potentially return null values. The way to solve this null dilemma is to use the NOT EXISTS operator discussed in Lab 8.2. The next query returns the desired result.

```
SELECT course_no, prerequisite
  FROM course c
 WHERE NOT EXISTS
       (SELECT '*'
       FROM course
       WHERE course_no IN (310,
       220)
       AND c.prerequisite =
       prerequisite)
```

The NVL and COALESCE functions are useful in dealing with null values. You can substitute a default value and apply the function to both the subquery and the WHERE clause condition.

ORDER BY Clause in Subqueries

With the exception of the inline view discussed in Lab 8.3, the ORDER BY clause is not allowed inside a subquery. If you attempt to include an ORDER BY clause, you receive an error message.

```
SELECT course_no, description, cost
  FROM course
 WHERE cost IN
       (SELECT cost
       FROM course
       WHERE prerequisite = 420
       ORDER BY cost)
       ORDER BY cost)
       *

ERROR at line 7:
ORA-00907: missing right
parenthesis
```

It is not immediately apparent where the problem lies unless you already know about this rule. The message essentially indicates that an ORDER BY clause is not permitted in a subquery and that Oracle is expecting to see the right parenthesis, signifying the closing of the subquery. An ORDER BY clause is certainly valid for the outer query—just not for the nested subquery.

LAB 8.1 EXERCISES

a) Write a SQL statement that displays the first and last names of the students who registered first.

b) Show the sections with the lowest course cost and a capacity equal to or lower than the average capacity. Also display the course description, section number, capacity, and cost.

c) Select the course number and total capacity for each course. Show only the courses with a total capacity less than the average capacity of all the sections.

d) Choose the most ambitious students: Display the STUDENT_ID for the students enrolled in the most sections.

e) Select the STUDENT_ID and SECTION_ID of enrolled students living in zip code 06820.

f) Display the course number and course description of the courses taught by instructor Fernand Hanks.

g) Select the last names and first names of students not enrolled in any class.

h) Determine the STUDENT_ID and last name of students with the highest FINAL_GRADE for each section. Also include the SECTION_ID and the FINAL_GRADE columns in the result.

i) Select the sections and their capacity, where the capacity equals the number of students enrolled.

LAB 8.1 EXERCISE ANSWERS

a) Write a SQL statement that displays the first and last names of the students who registered first.

ANSWER: You break down the query into logical pieces by first determining the earliest registration date of all students. The aggregate function MIN obtains the result in the subquery. The earliest date is compared to the

REGISTRATION_DATE column for each student, and only records that are equal to the same date and time are returned.

```
SELECT first_name, last_name
  FROM student
 WHERE registration_date =
       (SELECT MIN(registration_date)
          FROM student)
FIRST_NAME                     LAST_NAME
------------------------- ---------
J.                             Landry
Judith                         Olvsade
...
Larry                          Walter
Catherine                      Mierzwa

8 rows selected.
```

b) Show the sections with the lowest course cost and a capacity equal to or lower than the average capacity. Also display the course description, section number, capacity, and cost.

ANSWER: First, split the problem into individual queries. Start by determining the average capacity of all sections and the lowest course cost of all courses. To compare both cost and capacity against the subqueries, add a join to the SECTION and COURSE tables.

```
SELECT c.description, s.section_no, c.cost, s.capacity
  FROM course c, section s
 WHERE c.course_no = s.course_no
   AND s.capacity <=
       (SELECT AVG(capacity)
          FROM section)
   AND c.cost =
       (SELECT MIN(cost)
          FROM course)
```

DESCRIPTION	SECTION_NO	COST	CAPACITY
Intro to the Internet	1	1095	12
Intro to the Internet	2	1095	15
...			
Unix Tips and Techniques	2	1095	15
Unix Tips and Techniques	4	1095	15

6 rows selected.

c) Select the course number and total capacity for each course. Show only the courses with a total capacity less than the average capacity of all the sections.

ANSWER: To determine the total capacity per course, use the SUM function to add the values in the SECTION table's CAPACITY column. Compare the total capacity for each course to the average capacity for all sections and return those courses that have a total capacity less than the average capacity.

```
SELECT course_no, SUM(capacity)
  FROM section
 GROUP BY course_no
HAVING SUM(capacity) <
       (SELECT AVG(capacity)
          FROM section)
COURSE_NO SUM(CAPACITY)
--------- -------------
       10            15
      144            15

2 rows selected.
```

The solution shows only courses and their respective capacities that satisfy the condition in the HAVING clause.

To determine the solution, first write the individual queries and then combine them. The following query first determines the total capacity for each course.

```
COURSE_NO SUM(CAPACITY)
--------- -------------
       10            15
       20            80
...
      420            25
      450            25

28 rows selected.
```

You can easily obtain the average capacity for all sections by using the AVG function. SELECT AVG(capacity)

```
SELECT AVG(capacity)
  FROM section
AVG(CAPACITY)
-------------
    21.179487

1 row selected.
```

d) Choose the most ambitious students: Display the STUDENT_ID for the students enrolled in the most sections.

ANSWER: A count of records for each student in the ENROLLMENT table shows how many sections each student is enrolled in. Determine the highest number of enrollments per student by nesting the aggregate functions MAX and COUNT.

```
SELECT student_id, COUNT(*)
  FROM enrollment
 GROUP BY student_id
HAVING COUNT(*) =
       (SELECT MAX(COUNT(*))
          FROM enrollment
         GROUP BY student_id)
STUDENT_ID  COUNT(*)
---------- --------
       124        4
       214        4

2 rows selected.
```

To reach the subquery solution, determine the number of enrollments for each student. The STUDENT_ID column is not listed in the SELECT list. Therefore, only the result of the COUNT function is shown.

```
SELECT COUNT(*)
  FROM enrollment
 GROUP BY student_id
COUNT(*)
---------
        2
        1
...
        2
        2

165 rows selected.
```

The second query combines two aggregate functions to determine the highest number of sections any student is enrolled in. This subquery is then applied in the HAVING clause of the solution.

```
SELECT MAX(COUNT(*))
  FROM enrollment
 GROUP BY student_id
MAX(COUNT(*))
-------------
            4

1 row selected.
```

e) Select the STUDENT_ID and SECTION_ID of enrolled students living in zip code 06820.

ANSWER: The IN operator is necessary because the subquery returns multiple rows. SELECT student_id, section_id

```
SELECT student_id, section_id
  FROM enrollment
 WHERE student_id IN
       (SELECT student_id
          FROM student
         WHERE zip = '06820')
STUDENT_ID SECTION_ID
---------- ----------
       240         81

1 row selected.
```

Alternatively, you can write the query as a join and achieve the same result.

```
SELECT e.student_id, e.section_id
  FROM enrollment e, student s
 WHERE e.student_id =
       s.student_id
   AND s.zip = '06820'
```

f) Display the course number and course description of the courses taught by instructor Fernand Hanks.

ANSWER: To determine the courses taught by this instructor, nest multiple subqueries. SELECT course_no, description

```
SELECT course_no, description
  FROM course
 WHERE course_no IN
       (SELECT course_no
          FROM section
         WHERE instructor_id IN
               (SELECT instructor_id
                  FROM instructor
                       WHERE last_name = 'Hanks'
                         AND first_name = 'Fernand'))
COURSE_NO DESCRIPTION
--------- ------------------------------
       25 Intro to Programming
      240 Intro to the BASIC Language
...
      120 Intro to Java Programming
      122 Intermediate Java Programming
```


You can also solve this problem by using an equijoin.

```
SELECT c.course_no, c.description
  FROM course c, section s,
       instructor i
 WHERE c.course_no =
       s.course_no
   AND s.instructor_id =
       i.instructor_id
   AND i.last_name = 'Hanks'
   AND i.first_name = 'Fernand'
```

g) Select the last names and first names of students not enrolled in any class.

ANSWER: Use the NOT IN operator to eliminate student IDs not found in the ENROLLMENT table. The result is a list of students with no rows in the ENROLLMENT table. They may be newly registered students who have not yet enrolled in any courses.

```
SELECT last_name, first_name
  FROM student
 WHERE student_id NOT IN
       (SELECT student_id
          FROM enrollment)
LAST_NAME                 FIRST_NAME
------------------------- ----------
Eakheit                   George
Millstein                 Leonard
...
Larcia                    Preston
Mastandora                Kathleen

103 rows selected.
```

You might wonder why the solution does not include the DISTINCT keyword in the subquery. This keyword is not required and does not alter the result, nor does it change the efficiency of the execution. Oracle automatically eliminates duplicates in a list of values as a result of the subquery.

h) Determine the STUDENT_ID and last name of students with the highest FINAL_GRADE for each section. Also include the SECTION_ID and the FINAL_GRADE columns in the result.

ANSWER: The solution requires pairs of columns to be compared. First, determine the subquery to show the highest grade for each section. Then match the result to the columns in the outer query.

336

337

```
SELECT s.student_id, s.last_name, e.final_grade,
       e.section_id
  FROM enrollment e, student s
 WHERE e.student_id = s.student_id
   AND (e.final_grade, e.section_id) IN
       (SELECT MAX(final_grade), section_id
          FROM enrollment
         GROUP BY section_id)
STUDENT_ID LAST_NAME  FINAL_GRADE SECTION_ID
---------- ---------- ----------- ----------
       102 Crocitto            92         89

1 row selected.
```

There is no need to add a table alias to the subquery. Table aliases in subqueries are typically used only in correlated subqueries or in subqueries that contain joins. Correlated subqueries are discussed in Lab 8.2.

i) Select the sections and their capacity, where the capacity equals the number of students enrolled.

ANSWER: Use a subquery to determine the number of enrolled students per section. Then compare the resulting set to the column pair SECTION_ID and CAPACITY.

```
SELECT section_id, capacity
  FROM section
 WHERE (section_id, capacity) IN
       (SELECT section_id, COUNT(*)
          FROM enrollment
         GROUP BY section_id)
SECTION_ID  CAPACITY
----------  --------
        99        12

1 row selected.
```

337

Lab 8.1 Quiz

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

1) The ORDER BY clause is not allowed in subqueries.

_____**a)** True

_____**b)** False

2) Subqueries are used only in SELECT statements.

_____**a)** True

_____**b)** False

3) The most deeply nested, noncorrelated subquery always executes first.

_____**a)** True

_____**b)** False

4) What operator would you choose to prevent Oracle error ORA-01427: single-row subquery returns more than one row?

_____**a)** >=

_____**b)** =

_____**c)** IN

_____**d)** <=

5) Subqueries can return multiple rows and columns.

_____**a)** True

_____**b)** False

ANSWERS APPEAR IN APPENDIX A.

LAB 8.2 Correlated Subqueries

LAB OBJECTIVES

After this lab, you will be able to:

- Write Correlated Subqueries
- Use the EXISTS and NOT EXISTS Operators

Correlated subqueries are probably one of the most powerful, yet initially very difficult, concepts of the SQL language. Correlated subqueries are different from the simple subqueries discussed so far. First, they allow you to reference columns from the outer query in the subquery. Second, they execute the inner query repeatedly.

Reviewing Rows with Correlated Subqueries

A correlated subquery is so named because of the reference of a column from the outer query. You use a correlated subquery when you need to review every row of the outer query against the result of the inner query. The inner query is executed repeatedly, each time specific to the correlated value of the outer query. In contrast, in previous subquery examples, the inner query is executed only once.

One example in Lab 8.1 illustrates how to determine the students with the highest grades for their project (PJ), within their respective sections. The solution is accomplished with the IN operator, which compares the column pairs. The following SELECT statement repeats the solution.

```
SELECT student_id, section_id,
       numeric_grade
  FROM grade
 WHERE grade_type_code = 'PJ'
   AND (section_id, numeric_grade) IN
       (SELECT section_id,
       MAX(numeric_grade)
       FROM grade
       WHERE grade_type_code =
       'PJ'
       GROUP BY section_id)
```

Here is the query rewritten as a correlated subquery.

```
SELECT student_id, section_id, numeric_grade
  FROM grade outer
    WHERE grade_type_code = 'PJ'
      AND numeric_grade =
          (SELECT MAX(numeric_grade)
             FROM grade
            WHERE grade_type_code = outer.grade_type_code
              AND section_id = outer.section_id)
STUDENT_ID SECTION_ID NUMERIC_GRADE
---------- ---------- -------------
       245         82            77
       166         88            99
...
       232        149            83
       105        155            92

8 rows selected.
```

This query is a correlated subquery because the inner query refers to columns from the outer query. The GRADE table is the parent query, or the outer query. For simplicity, the table alias OUTER is used.

You can refer to columns of the outer query by using the alias. In this example, the values of the columns SECTION_ID and GRADE_TYPE_CODE of the outer query are compared to the values of the inner query. The inner query determines the highest project grade for the SECTION_ID and GRADE_TYPE_CODE values of the current outer row.

STEPS PERFORMED BY THE CORRELATED SUBQUERY

To select the correct records, Oracle performs the following steps.

1. Select a row from the outer query.
2. Determine the value of the correlated column(s).
3. Execute the inner query for each record of the outer query.
4. Feed the result of the inner query to the outer query and evaluate it. If it satisfies the criteria, return the row for output.

5. Select the next record of the outer query and repeat steps 2 through 4 until all the records of the outer query are evaluated.

Let's look at these steps in more detail.

STEP 1: SELECT A ROW FROM THE OUTER QUERY

Choose a record in the outer query where GRADE_TYPE_CODE equals 'PJ'. The row returned in this step will be further evaluated in the steps that follow.

339

341

```
SELECT student_id, section_id, numeric_grade
  FROM grade outer
 WHERE grade_type_code = 'PJ'

STUDENT_ID SECTION_ID NUMERIC_GRADE
---------- ---------- -------------
       105        155            92
       111        133            90
...
       245         82            77
       248        155            76

21 rows selected.
```

STEP 2: DETERMINE THE VALUE OF THE CORRELATED COLUMN(S)

Starting with the first returned row with STUDENT_ID 105, the value of the correlated column OUTER.SECTION_ID equals 155. For the column OUTER.GRADE_TYPE_CODE, the value is 'PJ'.

STEP 3: EXECUTE THE INNER QUERY

Based on the correlated column values, the inner query is executed. It shows the highest grade for the respective section ID and grade type code.

```
SELECT MAX(numeric_grade)
  FROM grade
 WHERE grade_type_code = 'PJ'
   AND section_id = 155
MAX(NUMERIC_GRADE)
------------------
                92

1 row selected.
```

STEP 4: EVALUATE THE CONDITION

Because NUMERIC_GRADE equals 92, the row for STUDENT_ID 105 evaluates to true and is included in the result.

STEP 5: REPEAT STEPS 2 THROUGH 4 FOR EACH SUBSEQUENT ROW OF THE OUTER QUERY

Evaluate the next row containing values STUDENT_ID 111 and SECTION_ID 133. The highest grade for the section and grade type code is 92, but student 111 does not have a NUMERIC_GRADE equal to this value. Therefore, the row is not returned. Each row of the outer query repeats these steps until all the rows are evaluated.

```
SELECT MAX(numeric_grade)
  FROM grade
 WHERE grade_type_code = 'PJ'
   AND section_id = 133
MAX(NUMERIC_GRADE)
------------------
                92

1 row selected.
```

341

Unlike the subqueries discussed in Lab 8.1, where the inner query is evaluated once, a correlated subquery executes the inner query repeatedly, once for each row in the outer query.

The EXISTS Operator

The EXISTS operator is used for correlated subqueries. It tests whether the subquery returns at least one row. The EXISTS operator returns either true or false, never unknown. Because EXISTS tests only whether a row exists, the columns shown in the SELECT list of the subquery are irrelevant. Typically, you use a single-character text literal, such as '1' or 'X', or the keyword NULL.

The following correlated subquery displays instructors where the INSTRUCTOR_ID has a matching row in the SECTION table. The result shows the INSTRUCTOR_ID, FIRST_NAME, LAST_NAME, and ZIP column values of instructors assigned to at least one section.

```
SELECT instructor_id, last_name, first_name, zip
  FROM instructor i
 WHERE EXISTS
       (SELECT 'X'
          FROM section
         WHERE i.instructor_id = instructor_id)
INSTRUCTOR_ID LAST_NAME      FIRST_NAME      ZIP
------------- -------------- --------------- -----
          101 Hanks          Fernand         10015
          102 Wojick         Tom             10025
          103 Schorin        Nina            10025
          104 Pertez         Gary            10035
          105 Morris         Anita           10015
          106 Smythe         Todd            10025
          108 Lowry          Charles         10025
          107 Frantzen       Marilyn         10005

8 rows selected.
```

For every row of the INSTRUCTOR table, the outer query evaluates the inner query. It checks whether the current row's INSTRUCTOR_ID value exists for the SECTION table's INSTRUCTOR_ID column. If a row with the appropriate value is found, the condition is true, and the outer row is included in the result.

The query can also be written using the IN operator.

```
SELECT instructor_id, last_name,
       first_name, zip
  FROM instructor
 WHERE instructor_id IN
       (SELECT instructor_id FROM
       section)
```

342

343

Alternatively, you can write this query with an equijoin.

```
SELECT DISTINCT i.instructor_id,
       i.last_name,
       i.first_name, i.zip
  FROM instructor i JOIN section s
    ON i.instructor_id = s.instructor_id
```

This equijoin solution allows you to list columns found on the SECTION table, whereas the subquery solution does not. The subquery has the advantage of breaking problems into individual pieces. Not all subqueries can be transformed into joins; for example, a subquery containing an aggregate function cannot be transformed into a join.

The NOT EXISTS Operator

The NOT EXISTS operator is the opposite of the EXISTS operator; it tests whether a matching row *cannot* be found. The operator is the most frequently used type

of correlated subquery construct. The following query displays the instructors not assigned to any section.

```
SELECT instructor_id, last_name, first_name, zip
  FROM instructor i
 WHERE NOT EXISTS
       (SELECT 'X'
          FROM section
         WHERE i.instructor_id = instructor_id)
INSTRUCTOR_ID LAST_NAME       FIRST_NAME      ZIP
------------- --------------- --------------- -----
          109 Chow            Rick            10015
          110 Willig          Irene

2 rows selected.
```

You cannot rewrite this particular query using an equijoin, but you can rewrite it with the NOT IN operator. However, the NOT IN operator does not always yield the same result if null values are involved, as you will see in the following example.

NOT EXISTS Versus NOT IN

Display the INSTRUCTOR_ID, FIRST_NAME, LAST_NAME, and ZIP columns from the INSTRUCTOR table where there is no corresponding zip code in the ZIPCODE table. Note that the ZIP column in the INSTRUCTOR table allows NULL values.

USING NOT EXISTS

The query with the NOT EXISTS operator retrieves instructor Irene Willig.

```
SELECT instructor_id, last_name, first_name, zip
  FROM instructor i
 WHERE NOT EXISTS
       (SELECT 'X'
           FROM zipcode
          WHERE i.zip = zip)
INSTRUCTOR_ID LAST_NAME       FIRST_NAME      ZIP
------------- --------------- --------------- ---
          110 Willig          Irene

1 row selected.
```

USING NOT IN

The same query rewritten with NOT IN does not return a record.

```
SELECT instructor_id, last_name,
       first_name, zip
  FROM instructor
 WHERE zip NOT IN
       (SELECT zip FROM zipcode)
```

```
no rows selected
```

As you can see, the difference between NOT EXISTS and NOT IN lies in the way NULL values are treated. Instructor Irene Willig's ZIP column contains a NULL value. The NOT EXISTS operator tests for NULL values; the NOT IN operator does not.

Avoiding Incorrect Results Through the Use of Subqueries

Many SQL statements may contain joins and aggregate functions. When you join tables together, some values may be repeated as a result of a one-to-many relationship between the joined tables. If you apply an aggregate function to the resulting repeating values, the result of the calculation may be incorrect. The following example shows a list of the total capacity for courses with enrolled students.

```
SELECT s.course_no, SUM(s.capacity)
  FROM enrollment e, section s
 WHERE e.section_id = s.section_id
 GROUP BY s.course_no
COURSE_NO SUM(S.CAPACITY)
--------- ---------------
       10              15
       20             175
...
      420              50
      450              25

25 rows selected.
```

To illustrate that the result is incorrect, look at the value for the CAPACITY column of COURSE_NO 20. The following query shows the capacity for each section, resulting in a total capacity of 80 students, rather than 175 students, as shown in the previous result.

343

345

```
SELECT section_id, capacity
  FROM section
 WHERE course_no = 20
SECTION_ID  CAPACITY
---------- --------
        81       15
        82       15
        83       25
        84       25

4 rows selected.
```

A closer look at the effect of the join without the aggregate function reveals the problem.

```
SELECT s.section_id, s.capacity, e.student_id,
       s.course_no
  FROM enrollment e, section s
 WHERE e.section_id = s.section_id
   AND s.course_no = 20
 ORDER BY section_id
SECTION_ID CAPACITY STUDENT_ID COURSE_NO
---------- -------- ---------- ---------
        81       15        103        20
        81       15        104        20
        81       15        240        20
        82       15        244        20
        82       15        245        20
        83       25        124        20
        83       25        235        20
        84       25        158        20
        84       25        199        20

9 rows selected.
```

For each enrolled student, the capacity record is repeated as the result of the join. This is correct, because for every row in the ENROLLMENT table, the corresponding SECTION_ID is looked up in the SECTION table. But when the SUM aggregate function is applied to the capacity, the capacity value of every returned record is added to the total capacity for each course. To achieve the correct result, the query needs to be written as follows.

```
SELECT course_no, SUM(capacity)
  FROM section s
 WHERE EXISTS
       (SELECT NULL
          FROM enrollment e, section sect
         WHERE e.section_id = sect.section_id
           AND sect.course_no = s.course_no)
 GROUP BY course_no

COURSE_NO SUM(CAPACITY)
--------- -------------
       10            15
       20            80
...
      420            25
      450            25

25 rows selected.
```

The outer query checks for every row if there is a matching COURSE_NO value in the subquery. If the

COURSE_NO exists, the EXISTS operator evaluates to true and will include the row of the outer query in the result. The outer query sums up the values for every row of the SECTION table. The EXISTS operator solves this particular problem, but not all queries can be solved this way; some may need to be written using inline views (see Lab 8.3).

Unnesting Queries

Sometimes Oracle implicitly transforms a subquery into a join by unnesting the query as part of its optimization strategy. For example, in the background Oracle rewrites the query if the primary and foreign keys exist on the tables and the join does not cause a Cartesian product or incorrect results because of an aggregate function in the SELECT clause.

Some subqueries cannot be unnested because they contain an aggregate function, a ROWNUM pseudocolumn (discussed in Lab 8.3), a set operator (see Chapter 9, “Set Operators”), or because it is a correlated subquery that references a query that is not in the immediate outer query block.

Subquery Performance Considerations

When you initially learn SQL, your utmost concern must be to obtain the correct result. Performance considerations should be secondary. However, as you get more experienced with the language or as you work with large data sets, you want to consider some of the effects of your constructed statements. As you have seen, sometimes you can achieve the same result with a join, a correlated subquery, or a noncorrelated subquery.

As previously mentioned, under specific circumstances, Oracle may automatically optimize your statement and implicitly transform your subquery to a join. This implicit transformation frequently results in better performance without your having to worry about applying any optimization techniques.

Performance benefits of one type of subquery over another type may be noticeable when you are working with very large volumes of data. To optimize subqueries, you must understand the key difference between correlated and noncorrelated subqueries.

A correlated subquery evaluates the inner query for every row of the outer query. Therefore, your optimization strategy should focus on eliminating as many rows as

possible from the outer query. You can do this by adding additional restricting criteria in the WHERE clause of the statement. The advantage of correlated subqueries is that they can use indexes on the correlated columns, if any exist.

A noncorrelated subquery executes the inner query first and then feeds this result to the outer query. The inner query is executed once. Generally speaking, this query is best suited for situations in which the inner query does not returns a very large result set and where no indexes exist on the compared columns.

If your query involves a NOT EXISTS condition, you cannot modify it to a NOT IN condition if the subquery can return null values. In many circumstances, NOT EXISTS offers better performance because indexes are usually used.

Because the STUDENT schema contains a fairly small number of records, the difference in execution time is minimal. The illustrated various solutions throughout this book allow you to look at different ways to approach and solve problems. You might want to use those solutions as starting points for ideas and perform your own tests, based on your distinct environment, data volume, and requirements.

Good performance is subject to many variables that can have a significant impact on the execution time of your statements. In Chapter 18, "SQL Optimization," you will learn more about the topic of SQL performance optimization.

LAB 8.2 EXERCISES

a) Explain what the following correlated subquery accomplishes.

```
SELECT section_id, course_no
  FROM section s
 WHERE 2 >
       (SELECT COUNT(*)
          FROM enrollment
         WHERE section_id = s.section_id)
SECTION_ID COURSE_NO
---------- ---------
        79       350
        80        10
...
       145       100
       149       120

27 rows selected.
```

b) List the sections where the enrollment exceeds the capacity of a section and show the number of

enrollments for the section using a correlated subquery.

c) Write a SQL statement to determine the total number of students enrolled, using the EXISTS operator. Count students enrolled in more than one course as one.

d) Show the STUDENT_ID, last name, and first name of each student enrolled in three or more classes.

e) Which courses do not have sections assigned? Use a correlated subquery in the solution.

f) Which sections have no students enrolled? Use a correlated subquery in the solution and order the result by the course number, in ascending order.

LAB 8.2 EXERCISE ANSWERS

a) Explain what the following correlated subquery accomplishes.

```
SELECT COUNT(*)
  FROM enrollment
 WHERE section_id = 80
 COUNT(*)
---------
         1

1 row selected.
```

ANSWER: The correlated subquery displays the SECTION_ID and COURSE_NO of sections with fewer than two students enrolled. It includes sections that have no students enrolled.

For each row of the SECTION table, the number literal 2 is compared to the result of the COUNT(*) function of the inner subquery. For each row of the outer SECTION table with the S.SECTION_ID column being the correlated column, the inner query counts the number of rows for this individual SECTION_ID. If no enrollment is found for the particular section, the COUNT function returns a zero; the row satisfies the criteria that 2 is greater than zero and is included in the result set.

Let's look at one of the rows of the SECTION table, specifically the row with the SECTION_ID value 80. The inner query returns a count of 1.

```
SELECT COUNT(*)
  FROM enrollment
 WHERE section_id = 80
 COUNT(*)
---------
         1

1 row selected.
```

When the number 1 is compared in the WHERE clause of the outer query, you see that 2 > 1 is true; therefore, this section is returned in the result.

You can write two queries to verify that the result of the correlated query is correct. First, write a query that shows sections where the enrollment is fewer than 2 students. This query returns 13 rows.

```
SELECT section_id, COUNT(*)
  FROM enrollment
 GROUP BY section_id
HAVING COUNT(*) < 2
SECTION_ID  COUNT(*)
---------- --------
        80        1
        96        1
...
       145        1
       149        1

13 rows selected.
```

Then write a second query to show the sections that have no enrollments (that is, the SECTION_ID does not exist in the ENROLLMENT table). To determine these sections, you can use the NOT IN operator

because the SECTION_ID in the ENROLLMENT table is defined as NOT NULL.

```
SELECT section_id
  FROM section
 WHERE section_id NOT IN
       (SELECT section_id
          FROM enrollment)
SECTION_ID
----------
        79
        93
...
       136
       139

14 rows selected.
```

The combination of the 13 and 14 rows from the last two queries returns a total of 27.

Alternatively, you can combine the results of the two queries with the UNION operator, as discussed in Chapter 9.

b) List the sections where the enrollment exceeds the capacity of a section and show the number of enrollments for the section using a correlated subquery.

ANSWER: The correlated query solution executes the outer query's GROUP BY clause

first; then, for every group, the subquery is executed to determine whether it satisfies the condition in the HAVING clause. Only sections where the number of enrolled students exceeds the capacity for the respective section are returned for output.

```
SELECT section_id, COUNT(*)
  FROM enrollment e
 GROUP BY section_id
  HAVING COUNT(*) >
         (SELECT capacity
            FROM section
           WHERE e.section_id = section_id)
SECTION_ID  COUNT(*)
---------- --------
       101       12

1 row selected.
```

Alternatively, you can solve this problem by using an equijoin and an aggregate function. You evaluate the enrollment count in the HAVING clause and compare it with the capacity. The additional CAPACITY column in the output validates the correct result.

```
SELECT COUNT(*)
  FROM student s
 WHERE EXISTS
       (SELECT NULL
          FROM enrollment
         WHERE student_id = s.student_id)
 COUNT(*)
---------
      165

1 row selected.
```

When you join tables and apply aggregate functions, be sure the resulting rows provide the correct result of the aggregate function.

c) Write a SQL statement to determine the total number of students enrolled, using the EXISTS operator. Count students enrolled in more than one course as one.

ANSWER: For every student, the query checks whether a row exists in the ENROLLMENT table. If this is true, the record is part of the result set. After Oracle determines all the rows that satisfy the EXISTS condition, the aggregate function COUNT is applied to determine the total number of students.

```
SELECT COUNT(*)
  FROM student s
 WHERE EXISTS
       (SELECT NULL
          FROM enrollment
         WHERE student_id = s.student_id)
 COUNT(*)
---------
       165

1 row selected.
```

The same result can be obtained with the following query. Because the ENROLLMENT table may contain multiple STUDENT_IDs if the student is enrolled in several sections, you need to count the distinct occurrences of STUDENT_ID to obtain the correct result.

349
351

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

1 row selected.
```

d) Show the STUDENT_ID, last name, and first name of each student enrolled in three or more classes.

ANSWER: There are four possible solutions to illustrate alternate ways to obtain the same result set.

SOLUTION 1: CORRELATED SUBQUERY

For each record in the STUDENT table, the inner query is executed to determine whether the STUDENT_ID occurs three or more times in the ENROLLMENT

table. The inner query's SELECT clause lists the NULL keyword, whereas in the previous examples, a text literal was selected. It is completely irrelevant what columns are selected in the subquery with the EXISTS and NOT EXISTS operators because these operators only check for the existence or nonexistence of rows.

```
SELECT first_name, last_name, student_id
  FROM student s
 WHERE EXISTS
       (SELECT NULL
          FROM enrollment
         WHERE s.student_id = student_id
         GROUP BY student_id
        HAVING COUNT(*) >= 3)
FIRST_NAME                LAST_NAME      STUDENT_ID
------------------------- -------------- ----------
Daniel                    Wicelinski            124
Roger                     Snow                  238
...
Salewa                    Zuckerberg            184
Yvonne                    Williams              214

7 rows selected.
```

SOLUTION 2: EQUIJOIN

The next solution joins the STUDENT and ENROLLMENT tables. Students enrolled multiple times are grouped into one row, and the COUNT function counts the occurrences of each student's

enrollment record. Only those having three or more records in the ENROLLMENT table are included.

```
SELECT first_name, last_name,
       s.student_id
  FROM enrollment e, student s
 WHERE e.student_id = s.student_id
 GROUP BY first_name, last_name,
       s.student_id
HAVING COUNT(*) >= 3
```

351

352

Although Solution 2 achieves the correct result, you need to be aware of the dangers of aggregate functions in joins.

SOLUTION 3: IN SUBQUERY

This subquery returns only STUDENT_IDs with three or more enrollments. The result is then fed to the outer query.

```
SELECT first_name, last_name,
       student_id
  FROM student
 WHERE student_id IN
       (SELECT student_id
       FROM enrollment
       GROUP BY student_id
       HAVING COUNT(*) >= 3)
```

SOLUTION 4: ANOTHER CORRELATED SUBQUERY

The number literal 3 is compared to the result of the correlated subquery. It counts the enrollment records for the individual students. This solution is similar to the solution in exercise a in this lab.

```
SELECT last_name, first_name,
       student_id
  FROM student s
 WHERE 3 <= (SELECT COUNT(*)
       FROM enrollment
       WHERE s. student_id =
       student_id)
```

e) Which courses do not have sections assigned? Use a correlated subquery in the solution.

ANSWER: For every course in the COURSE table, the NOT EXISTS condition probes the SECTION table to determine whether a row with the same course number *does not* exist. If the course number is not found, the WHERE clause evaluates to true, and the record is included in the result set.

```
SELECT course_no, description
  FROM course c
 WHERE NOT EXISTS
       (SELECT 'X'
          FROM section
         WHERE c.course_no = course_no)
COURSE_NO DESCRIPTION
--------- ------------------------------
       80 Programming Techniques
      430 Java Developer III

2 rows selected.
```

Note you can also write the query as follows.

```
SELECT course_no, description
  FROM course c
 WHERE NOT EXISTS
       (SELECT 'X' FROM section
       s WHERE c.course_no = s.
       course_no)
```

352

353

The SECTION table uses the table alias named s, which the S.COURSE_NO column refers to. This alias is not required; it simply clarifies the column's source table. When you use a columns without an alias, it is understood that the columns refers to the table in the current subquery. However, you must use a table alias for the C.COURSE_NO column, referencing the COURSE_NO in the outer query; otherwise, the query is not correlated.

As an alternative, you can obtain the same result by using the NOT IN operator. Because the COURSE_NO column in the SECTION table is defined as NOT NULL, the query returns the same result.

```
SELECT course_no, description
  FROM course
 WHERE course_no NOT IN
       (SELECT course_no FROM
       section)
```

f) Which sections have no students enrolled? Use a correlated subquery in the solution and order the result by the course number, in ascending order.

ANSWER: The result contains only rows where the SECTION_ID does not exist in the ENROLLMENT table. The inner query executes for each row of the outer query. SELECT course_no, section_id

```
SELECT course_no, section_id
  FROM section s
 WHERE NOT EXISTS
       (SELECT NULL
          FROM enrollment
         WHERE s.section_id = section_id)
 ORDER BY course_no
COURSE_NO SECTION_ID
--------- ----------
       25         93
      124        129
...
      350         79

14 rows selected.
```

You can achieve the same result by using the NOT IN operator because the SECTION_ID column in the ENROLLMENT table is defined as NOT NULL.

```
SELECT course_no, section_id
  FROM section
 WHERE section_id NOT IN
       (SELECT section_id FROM
       enrollment)
ORDER BY course_no
```


Lab 8.2 Quiz

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

1) The NOT EXISTS operator tests for occurrences of nulls.

   _____**a)** True

   _____**b)** False

2) In a correlated subquery, the inner query is executed repeatedly.

   _____**a)** True

   _____**b)** False

3) The operators IN and EXISTS are somewhat equivalent.

_____**a)** True

_____**b)** False

4) What problem does the following SQL statement solve?

```
SELECT student_id, section_id
  FROM enrollment e
 WHERE NOT EXISTS
       (SELECT '1' FROM grade g
       WHERE e.section_id =
       section_id AND
       e.student_id = student_id)
```

_____**a)** Show the enrolled students and their respective sections that have grades assigned.

_____**b)** Determine the students and their sections where no grades have been assigned.

_____**c)** Determine which students are not enrolled.

_____**d)** Determine which students are not enrolled and do not have grades.

_____**e)** This is an invalid query.

5) Always evaluate the result of a join first, before applying an aggregate function.

_____**a)** True

_____**b)** False

ANSWERS APPEAR IN APPENDIX A.

LAB 8.3 Inline Views and Scalar Subquery Expressions

LAB OBJECTIVES

After this lab, you will be able to:

- Write Inline Views
- Write Scalar Subquery Expressions

Inline views and scalar subquery expressions help you simplify the writing of SQL statements. They allow you to break down complicated query requests into individual queries and then combine the results.

Inline Views

Inline views, also referred to as queries in the FROM clause, allow you to treat a query as a virtual table or view. The following is an example of an inline view.

```
SELECT e.student_id, e.section_id, s.last_name
  FROM (SELECT student_id, section_id, enroll_date
          FROM enrollment
         WHERE student_id = 123) e,
        student s
 WHERE e.student_id = s.student_id
STUDENT_ID SECTION_ID LAST_NAME
---------- ---------- ---------
       123         87 Radicola

1 row selected.
```

An inline view is written in the FROM clause of a query and enclosed in a set of parentheses; this query has the alias e. The result of this query is evaluated and executed first, and then the result is joined to the STUDENT table.

An inline view acts just like a virtual table, or, for that matter, like a view. A view is a query definition stored in the database that looks just like a table. It does not have any physical rows because a view is actually a stored query that is executed only when the view is accessed. You'll learn more about views in Chapter 13, "Indexes, Sequences, and Views."

One of the differences between a view and an inline view is that an inline view does not need to be created and stored in the data dictionary. You can create an inline view or a virtual table by placing your query in the FROM clause of a SQL statement.

Inline view queries may look complicated, but they are easy to understand. They allow you to break down complex problems into simple queries. The following query uses two inline views to return the actual number of enrollments for course number 20 and joins this result to the capacity of the course. The actual and potential revenue are then computed by multiplying the course cost by the number of enrollments and the by the respective capacity of the course.

```
SELECT enr.num_enrolled "Enrollments",
       enr.num_enrolled * c.cost "Actual Revenue",
       cap.capacity "Total Capacity",
       cap.capacity * c.cost "Potential Revenue"
  FROM (SELECT COUNT(*) num_enrolled
          FROM enrollment e, section s
         WHERE s.course_no = 20
           AND s.section_id = e.section_id) enr,
       (SELECT SUM(capacity) capacity
          FROM section
         WHERE course_no = 20) cap,
       course c
 WHERE c.course_no = 20
Enrollments Actual Revenue Total Capacity Potential Revenue
----------- -------------- -------------- -----------------
          9          10755             80             95600

1 row selected.
```

The easiest way to understand the query is to look at the result set for each inline view. The first query, referenced with the alias enr, returns the number of students enrolled in course number 20. It requires a join between the ENROLLMENT table and the SECTION table, because the number of students enrolled per section is in the ENROLLMENT table, and the COURSE_NO column is found in the SECTION table. The column joining the two tables is the SECTION_ID. The query returns one row and indicates that nine students are enrolled in course number 20.

```
SELECT COUNT(*) num_enrolled
  FROM enrollment e, section s
 WHERE s.course_no = 20
   AND s.section_id = e.section_id
NUM_ENROLLED
------------
           9

1 row selected.
```

The second query, with the alias cap, uses the aggregate function SUM to add all the values in the CAPACITY column for course number 20. Because the SUM function is an aggregate function, it returns one row with the total capacity of 80 for all the sections for course number 20.

```
SELECT SUM(capacity) capacity
  FROM section
 WHERE course_no = 20
CAPACITY
---------
       80

1 row selected.
```

The last table in the FROM clause of the query is the COURSE table. This table holds the course cost to compute the actual revenue and the potential revenue. The query also retrieves one row.

The results of the inline views, which are identified with the aliases enr and cap, are not joined together with the COURSE table, thus creating a Cartesian product. Because a multiplication of the number of rows from each involved inline view and table, 1*1*1, results in one row, this query returns the one row for course number 20. A join condition is not required in this case, but it can be added for clarification.

TOP-N QUERY

An example of a top-n query is a query that allows you to determine the top three students for a particular section. To accomplish this, you need to understand the use of the ROWNUM pseudocolumn.

This column returns a number indicating the order in which Oracle returns the rows from a table or set of tables. You can use ROWNUM to limit the number of rows returned, as in the following example. It returns the first five rows.

```
SELECT last_name, first_name
  FROM student
 WHERE ROWNUM <=5
LAST_NAME                 FIRST_NAME
------------------------- ----------
Cadet                     Austin V.
M. Orent                  Frank
Winnicki                  Yvonne
Madej                     Mike
Valentine                 Paula

5 rows selected.
```

A pseudocolumn is not a real column in a table; you can use SELECT on this column, but you cannot manipulate its values. You will learn more about other pseudocolumns (for example, LEVEL, NEXTVAL, CURRVAL, ROWID) throughout this book.

You can combine the ROWNUM pseudocolumn and an inline view to determine the three highest final examination grades of section 101, as illustrated in the following query.

```
SELECT ROWNUM, numeric_grade
  FROM (SELECT DISTINCT numeric_grade
          FROM grade
         WHERE section_id = 101
           AND grade_type_code = 'FI'
         ORDER BY numeric_grade DESC)
  WHERE ROWNUM <= 3
   ROWNUM NUMERIC_GRADE
--------- -------------
        1            99
        2            92
        3            91

3 rows selected.
```

The inline view selects the distinct values in the NUMERIC_GRADE column for all final examination grades where SECTION_ID equals 101. This result is ordered by the NUMERIC_ GRADE, in descending order, with the highest NUMERIC_GRADE listed first. The outer query uses the ordered result of the inline view and the ROWNUM column to return only the first three. By ordering the results within the inline view, this construct provides a method to both limit and order the number of rows returned. For even more sophisticated ranking functionality and for analytical and statistical functions, see Chapter 17, “Exploring Data Warehousing Features.”

PRACTICAL USES OF INLINE VIEWS

If you face a problem that is complex and challenging, using inline views may be the best way to solve the problem without violating any of the SQL syntax restrictions. Inline views allow you to break down the problem into individual queries and then combine the results through joins. If you want to write top-n queries without using any of the ranking functions (discussed in Chapter 17), you need to use an inline view.

Scalar Subquery Expressions

You have already learned about the scalar subquery, which is a query that returns a single-column, single-row value. You can use a scalar subquery expression in most syntax that calls for an expression. The next examples show you how to use this functionality in the SELECT list, in the WHERE clause, in the ORDER BY clause of a query, in a CASE expression, or as part of a function call.

SCALAR SUBQUERY EXPRESSION IN THE SELECT CLAUSE

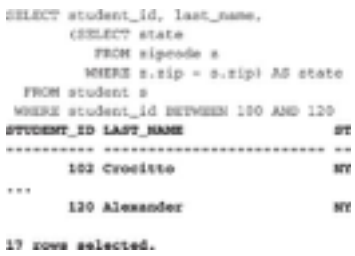
The following query returns all the Connecticut zip codes and a count of how many students live in each zip

code. The query is correlated as the scalar subquery, and it is executed for each individual zip code. For some zip codes, no students are in the STUDENT table; therefore, the subquery's COUNT function returns a zero. (The query can also be written as an outer join, as discussed in Chapter 10, "Complex Joins.")


```
SELECT city, state, zip,
       (SELECT COUNT(*)
          FROM student s
         WHERE s.zip = z.zip) AS student_count
  FROM zipcode z
 WHERE state = 'CT'
CITY                 ST ZIP   STUDENT_COUNT
-------------------- -------- -------------
Ansonia              CT 06401             0
...
Stamford             CT 06907             1

19 rows selected.
```

Scalar subquery expressions can become notoriously inefficient because Oracle can often execute table joins faster, particularly when scans of the entire result set are involved. Following is an example where the result of an equijoin is achieved by using a scalar subquery.



SCALAR SUBQUERY EXPRESSION IN THE WHERE CLAUSE

The following query is an example of a scalar subquery expression in the WHERE clause of a SELECT statement. The WHERE clause limits the result set to those students who enrolled in more courses than the average student. The equivalent equijoin is probably more efficient.

```
SELECT student_id, last_name
  FROM student s
 WHERE (SELECT COUNT(*)
          FROM enrollment e
         WHERE s.student_id = e.student_id) >
               (SELECT AVG(COUNT(*))
                  FROM enrollment
                 GROUP BY student_id)
 ORDER BY 1
STUDENT_ID LAST_NAME
---------- ---------
       102 Crocitto
...
       283 Perkins

52 rows selected.
```

SCALAR SUBQUERY EXPRESSION IN THE ORDER BY CLAUSE

You might wonder why you would ever need to execute a scalar subquery expression in the ORDER BY clause. The following example illustrates that you can sort by a column that does not even exist in the STUDENT table. The query lists the STUDENT_ID and LAST_NAME columns of those students with an ID between 230 and 235. The result is ordered by the number of sections a respective student is enrolled in. If you execute a separate query to verify the result, you see that student Brendler is enrolled in one section and student Jung in three sections.

```
SELECT student_id, last_name
  FROM student s
 WHERE student_id BETWEEN 230 AND 235
 ORDER BY (SELECT COUNT(*)
             FROM enrollment e
            WHERE s.student_id = e.student_id) DESC
STUDENT_ID LAST_NAME
---------- ---------
       232 Jung
...
       234 Brendler

5 rows selected.
```

SCALAR SUBQUERY EXPRESSION AND THE CASE EXPRESSION

Scalar subquery expressions are particularly handy in CASE expressions and within the DECODE function. The following example demonstrates their extraordinarily powerful functionality. The SELECT statement lists the costs of COURSE_NO 20 and 80. The column labeled Test CASE illustrates the result of the CASE expression.

Depending on the value of the COST column, a comparison against a scalar subquery expression is executed. For example, the COST column is compared to the average COST of all courses, and if the value in the COST column is less than or equal to that average, the value in the COST column is multiplied by 1.5.

The next WHEN comparison checks whether the cost is equal to the highest course cost. If so, it displays the value of the COST column for COURSE_NO 20. If the scalar subquery expression determines that the row with COURSE_NO 20 does not exist, the scalar subquery expression evaluates to NULL.

```
SELECT course_no, cost,
       CASE WHEN cost <= (SELECT AVG(cost) FROM course) THEN
                          cost *1.5

            WHEN cost =  (SELECT MAX(cost) FROM course) THEN
                         (SELECT cost FROM course
                            WHERE course_no = 20)
            ELSE cost
       END "Test CASE"
  FROM course
 WHERE course_no IN (20, 80)
 ORDER BY 2
```

COURSE_NO	COST	Test CASE
20	1195	1792.5
80	1595	1195

2 rows selected.

The next example shows the use of the scalar subquery expression in the condition part of the CASE expression. The cost of course number 134, which is 1195, is multiplied by 2, effectively doubling the cost. This result is then compared to see if it's less than or equal to the average cost of all courses.

```
SELECT course_no, cost,
       CASE WHEN (SELECT cost*2
                    FROM course
                   WHERE course_no = 134)
                      <= (SELECT AVG(cost) FROM course) THEN
                      cost *1.5
            WHEN cost = (SELECT MAX(cost) FROM course) THEN
                      (SELECT cost FROM course
                        WHERE course_no = 20)
            ELSE cost
       END "Test CASE"
  FROM course
 WHERE course_no IN (20, 80)
 ORDER BY 2
```

COURSE_NO	COST	Test CASE
20	1195	1195
80	1595	1195

2 rows selected.

SCALAR SUBQUERY EXPRESSIONS AND FUNCTIONS

The next example shows the use of a scalar subquery expression within a function. For every retrieved row, the UPPER function is executed, which in turn retrieves the respective student's last name from the STUDENT table. A join between the STUDENT and ENROLLMENT tables to obtain the same information is typically more efficient, but the example illustrates another of the many versatile uses of scalar subquery expressions.

359

362

```
SELECT student_id, section_id,
       UPPER((SELECT last_name
                FROM student
               WHERE student_id = e.student_id))
       "Last Name in Caps"
  FROM enrollment e
 WHERE student_id BETWEEN 100 AND 110
STUDENT_ID SECTION_ID Last Name in Caps
---------- ---------- ----------------
       102         86 CROCITTO
       102         89 CROCITTO
...
       110         95 MARTIN
       110        154 MARTIN

13 rows selected.
```

ERRORS IN SCALAR SUBQUERY EXPRESSIONS

Just as you learned in Lab 8.1, the scalar subquery expression must always return one row and one column. Otherwise, Oracle returns error ORA-01427: single-row subquery returns more than one row. If you list multiple columns, you receive error ORA-00913: "too many values." If your subquery does not return any row, a null value is returned.

PERFORMANCE CONSIDERATIONS

While you have seen that scalar subquery expressions can be used anywhere expressions are allowed, their application may not be practical under all circumstances. For example, to display a value from another table, a join is frequently more efficient than a scalar subquery in the SELECT list. Because the scalar subquery expression will be evaluated for each row, you should try to eliminate as many rows as possible prior to the execution of the scalar subquery expression step. This can be achieved by adding additional restricting conditions to the WHERE clause. As you might expect, there are many ways to achieve the same result, using various SQL syntax options; this chapter illustrates

some of these possibilities. Unfortunately, there is no specific set of guidelines you can follow to ensure that your SQL statement executes in a timely manner. Many variables affect performance; however, the Oracle optimizer (discussed in Chapter 18) typically does a pretty good job of efficiently processing your statement.

LAB 8.3 EXERCISES

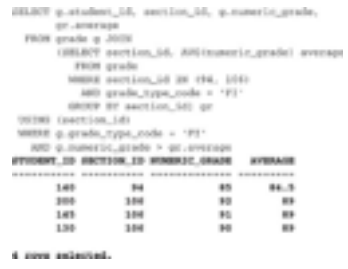
a) Write a query that displays the SECTION_ID and COURSE_NO columns, along with the number of students enrolled in sections with the IDs 93, 101, and 103. Use a scalar subquery to write the query. The result should look similar to the following output.

362
363

```
SECTION_ID COURSE_NO NUM_ENROLLED
---------- --------- ------------
        93        25            0
       103       310            4
       101       240           12

3 rows selected.
```

b) What problem does the following query solve?



c) For each course number, display the total capacity of the individual sections. Include the number of students enrolled and the percentage of the course that is filled. The result should look similar to the following output.

```
COURSE_NO TOTAL_CAPACITY TOTAL_STUDENTS Filled Percentage
--------- -------------- -------------- -----------------
      240             25             13                52
      230             27             14             51.85
...
      450             25              1                 4
      134             65              2              3.08

25 rows selected.
```

d) Determine the top five courses with the largest numbers of enrollments.

LAB 8.3 EXERCISE ANSWERS

a) Write a query that displays the SECTION_ID and COURSE_NO columns, along with the number of students enrolled in sections with the IDs 93, 101, and 103. Use a scalar subquery to write the query. The result should look similar to the following output.

```
SECTION_ID COURSE_NO NUM_ENROLLED
---------- --------- ------------
        93        25            0
       103       310            4
       101       240           12

3 rows selected.
```

ANSWER: This query uses a scalar subquery in the SELECT clause of the SQL statement. The scalar subquery is correlated and determines for each of the three SECTION_ID values the number of rows in the ENROLLMENT table.

```
SELECT section_id, course_no,
       (SELECT COUNT(*)
        FROM enrollment e
        WHERE s.section_id =
        e.section_id)
        AS num_enrolled
  FROM section s
 WHERE section_id IN (101, 103,
 93)
```

b) What problem does the following query solve?

```
SELECT g.student_id, section_id, g.numeric_grade,
       gr.average
  FROM grade g JOIN
       (SELECT section_id, AVG(numeric_grade) average
          FROM grade
         WHERE section_id IN (94, 106)
           AND grade_type_code = 'FI'
         GROUP BY section_id) gr
 USING (section_id)
 WHERE g.grade_type_code = 'FI'
   AND g.numeric_grade > gr.average
STUDENT_ID SECTION_ID NUMERIC_GRADE   AVERAGE
---------- ---------- -------------- ---------
       140         94             85      84.5
       200        106             92        89
       145        106             91        89
       130        106             90        89

4 rows selected.
```

ANSWER: The query show for sections 94 and 106 those students with a final examination grade higher than the average for each respective section.

The inline view determines the average final examination grade for each of the sections 94 and 106. This query is executed first. The result is then joined with the GRADE table, where the SECTION_ID column agrees. The filtering criteria is that the GRADE_TYPE_CODE column equals 'FI', which stands for final examination grade, and the last condition chooses only those rows that have a grade higher than the average for each respective section.

364
365

c) For each course number, display the total capacity of the individual sections. Include the number of students enrolled and the percentage of the course that is filled. The result should look similar to the following output.

```
COURSE_NO TOTAL_CAPACITY TOTAL_STUDENTS Filled Percentage
--------- -------------- -------------- -----------------
      240             25             13                52
      230             27             14             51.85
...
      450             25              1                 4
      134             65              2              3.08

25 rows selected.
```

ANSWER: This query uses inline views to retrieve the total capacity and number of students enrolled. The Filled Percentage column is calculated using the resulting values from the inline views and is used to order the result.

```
SELECT a.course_no, total_capacity,
       total_students,
       ROUND(100/total_capacity*total_students,
       2)
       ”Filled Percentage”
  FROM (SELECT COUNT(*) total_students,
       s.course_no FROM enrollment e, section s
       WHERE e.section_id = s.section_id
       GROUP BY s.course_no) a,
       (SELECT SUM(capacity) total_capacity,
       course_no FROM section GROUP BY
       course_no) b
 WHERE b.course_no = a.course_no
 ORDER BY ”Filled Percentage” DESC
```

It helps to build the query step by step, looking at the individual queries. The first query, with the alias a, returns the total number of students enrolled in each course.

```
SELECT COUNT(*) total_students, s.course_no
  FROM enrollment e, section s
 WHERE e.section_id = s.section_id
 GROUP BY s.course_no
TOTAL_STUDENTS COURSE_NO
-------------- ---------
             1        10
             9        20
...
             2       420
             1       450

25 rows selected.
```

365

366

The second query, with the alias b, returns the total capacity for each course.

```
SELECT SUM(capacity) total_capacity, course_no
  FROM section
 GROUP BY course_no
TOTAL_CAPACITY COURSE_NO
-------------- ---------
            15        10
            80        20
...
            25       420
            25       450

28 rows selected.
```

Then you join the two queries by the common column, COURSE_NO, using the aliases a and b assigned in the inline view queries. The outer

query references the columns TOTAL_STUDENTS and TOTAL_CAPACITY. The ROUND function computes the percentage, with two-digit precision after the comma. The result is sorted by this percentage, in descending order.

d) Determine the top five courses with the largest numbers of enrollments.

ANSWER: This question is solved with an inline view and the ROWNUM pseudocolumn. You will learn about more advanced ranking functionality in Chapter 17.

```
SELECT ROWNUM Ranking, course_no, num_enrolled
  FROM (SELECT COUNT(*) num_enrolled, s.course_no
          FROM enrollment e, section s
         WHERE e.section_id = s.section_id
         GROUP BY s.course_no
         ORDER BY 1 DESC)
 WHERE ROWNUM <= 5
  RANKING COURSE_NO NUM_ENROLLED
--------- --------- ------------
        1        25           45
        2       122           24
        3       120           23
        4       140           15
        5       230           14

5 rows selected.
```

366

Lab 8.3 Quiz

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

1) The ORDER BY clause is allowed in an inline view.

   _____**a)** True

   _____**b)** False

2) Scalar subqueries return one or more rows.

   _____**a)** True

   _____**b)** False

3) Inline views are stored in the data dictionary.

   _____**a)** True

   _____**b)** False

4) ROWNUM is an actual column in a table.

   _____**a)** True

   _____**b)** False

ANSWERS APPEAR IN APPENDIX A.

LAB 8.4 ANY, SOME, and ALL Operators in Subqueries

LAB OBJECTIVES

After this lab, you will be able to:

- Use the ANY, SOME, and ALL Operators in Subqueries
- Understand the Differences between These Operators

You are already familiar with the IN operator, which compares a list of values for equality. The ANY, SOME, and ALL operators are related to the IN operator as they also compare against a list of values. In addition, these operators allow >, <, >=, and <= comparisons.

The ANY operator checks whether any value in the list makes the condition true. The ALL operator returns rows if the condition is true for all the values in the list. The SOME operator is identical to ANY, and the two can be used interchangeably. Before applying these operators to subqueries, examine their effect on a simple list of values.

The following query retrieves all the grades for SECTION_ID 84.

```
SELECT section_id, numeric_grade
  FROM grade
 WHERE section_id = 84
SECTION_ID NUMERIC_GRADE
---------- -------------
        84            88
        84            99
        84            77
        84            88

4 rows selected.
```

The familiar IN operator in the next SQL statement chooses all the grades that are equal to either 77 or 99.

```
SELECT section_id, numeric_grade
  FROM grade
 WHERE section_id = 84
   AND numeric_grade IN (77, 99)
   SECTION_ID NUMERIC_GRADE
   ---------- -------------
           84            99
           84            77

   2 rows selected.
```

If you want to perform a comparison such as less than (<) against a list of values, use either the ANY, SOME, or ALL operator.

ANY and SOME

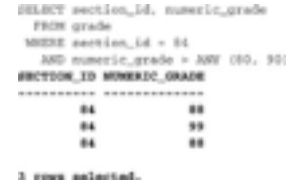
The following SQL query looks for any rows where the value in the NUMERIC_GRADE column is less than either value in the list.

```
SELECT section_id, numeric_grade
  FROM grade
 WHERE section_id = 84
   AND numeric_grade < ANY (80, 90)
SECTION_ID NUMERIC_GRADE
---------- -------------
        84            88
        84            88
        84            77

3 rows selected.
```

The query returns the NUMERIC_GRADE values 77 and 88. For the rows with NUMERIC_ GRADE 88, the condition is true because 88 is less than 90, but the condition is not true for the value 80. However, because the condition needs to be true for any of the records compared in the list, the row is included in the result.

The following query performs a greater-than comparison with the ANY operator.



Because the records with NUMERIC_GRADE 88 are greater than 80, they are included. NUMERIC_GRADE 99 is greater than both 80 and 90 and is therefore also included in the result set, although just one of the conditions is sufficient to be included in the result set.

368

370

The ANY operator with the = operator is the equivalent of the IN operator. There are no rows that have a NUMERIC_GRADE of either 80 or 90.

```
SELECT section_id, numeric_grade
  FROM grade
WHERE section_id = 84
    AND numeric_grade = ANY (80, 90)
```

```
no rows selected
```

The following query is the logical equivalent to the = ANY condition.

```
SELECT section_id, numeric_grade
  FROM grade
WHERE section_id = 84
    AND numeric_grade IN (80, 90)
```

```
no rows selected
```

The ALL Operator

The ALL operator returns true if every value in the list satisfies the condition. In the following example, all the records in the GRADE table must be less than 80 and 90. This condition is true only for the row with the NUMERIC_GRADE value 77, which is less than both 80 and 90.

```
SELECT section_id, numeric_grade
  FROM grade
 WHERE section_id = 84
   AND numeric_grade < ALL (80, 90)
SECTION_ID NUMERIC_GRADE
---------- -------------
        84            77

1 row selected.
```

A SQL statement using <> ALL is equivalent to NOT IN.

```
SELECT section_id, numeric_grade
  FROM grade
 WHERE section_id = 84
   AND numeric_grade <> ALL (80, 90)
SECTION_ID NUMERIC_GRADE
---------- -------------
        84            88
        84            99
        84            77
        84            88

4 rows selected.
```

Whenever a subquery with the ALL operator fails to return a row, the query is automatically true. This is different from the ANY operator, which returns false.

LAB 8.4 EXERCISES

a) Write a SELECT statement to display the STUDENT_ID, SECTION_ID, and grade for every student who received a final examination grade better than *all* of his or her individual homework grades.

b) Based on the result of exercise a, what do you observe about the row with STUDENT_ID 102 and SECTION_ID 89?

c) Select the STUDENT_ID, SECTION_ID, and grade of every student who received a final examination grade better than *any* of his or her individual homework grades.

d) Based on the result of exercise c, explain the result of the row with STUDENT_ID 102 and SECTION_ID 89.

LAB 8.4 EXERCISE ANSWERS

a) Write a SELECT statement to display the STUDENT_ID, SECTION_ID, and grade for every student who received a final examination grade better than *all* of his or her individual homework grades.

ANSWER: Use a correlated subquery to compare each individual student's final examination grade with his or her homework grades for a particular section. The output includes only those records where the final examination grade is higher than all the homework grades.

```
SELECT student_id, section_id, numeric_grade
  FROM grade g
 WHERE grade_type_code = 'FI'
   AND numeric_grade > ALL
       (SELECT numeric_grade
          FROM grade
         WHERE grade_type_code = 'HM'
           AND g.section_id = section_id
           AND g.student_id = student_id)
STUDENT_ID SECTION_ID NUMERIC_GRADE
---------- ---------- -------------
       102         89            92
       124         83            99
       143         85            92
...
       215        156            90
       283         99            85

96 rows selected.
```

To verify the result, use STUDENT_ID 143 and SECTION_ID 85 as an example. The highest grade for all of the homework is 91, and the lowest is 81. The grade achieved on the final examination is 92.


```
SELECT student_id, section_id, grade_type_code,
       MAX(numeric_grade) max, MIN(numeric_grade) min
  FROM grade
 WHERE student_id = 143
   AND section_id = 85
   AND grade_type_code IN ('HM', 'FI')
 GROUP BY student_id, section_id, grade_type_code
STUDENT_ID SECTION_ID GR         MAX   MIN
---------- ---------- -------- ----- -----
       143         85 FI          92    92
       143         85 HM          91    81

2 rows selected.
```

The student with ID 143 enrolled in section 85 is correctly selected for output. As shown in the result, the condition that the final examination grade be greater than all the homework grades is satisfied.

The following query verifies that the student with ID 179 enrolled in section 116 has a lower grade in the final exam than in all the homework grades. Therefore, the row is not included in the set.

```
SELECT student_id, section_id, grade_type_code,
       MAX(numeric_grade) max, MIN(numeric_grade) min
  FROM grade
 WHERE student_id = 179
   AND section_id = 116
   AND grade_type_code IN ('HM', 'FI')
 GROUP BY student_id, section_id, grade_type_code
STUDENT_ID SECTION_ID GR MAX MIN
---------- ---------- -- --- ---
       179        116 FI  90  90
       179        116 HM  99  99

2 rows selected.
```

b) Based on the result of exercise a, what do you observe about the row with STUDENT_ID 102 and SECTION_ID 89?

ANSWER: When the subquery with the ALL operator fails to return a row, the query is automatically true. Therefore, this student is also included in the result set.

The interesting aspect of the relationship between ALL and NULL is that here the student for this section has no homework grades, yet the row is returned for output.

```
SELECT student_id, section_id, grade_type_code,
       MAX(numeric_grade) max, MIN(numeric_grade) min
  FROM grade
 WHERE student_id = 102
   AND section_id = 89
   AND grade_type_code IN ('HM', 'FI')
 GROUP BY student_id, section_id, grade_type_code

STUDENT_ID SECTION_ID GR MAX MIN
---------- ---------- -- --- ---
       102         89 FI  92  92

1 row selected.
```

c) Select the STUDENT_ID, SECTION_ID, and grade of every student who received a final examination grade better than *any* of his or her individual homework grades.

ANSWER: Using the ANY operator together with the correlated subquery achieves the desired result.

```
SELECT student_id, section_id, numeric_grade
  FROM grade g
 WHERE grade_type_code = 'FI'
   AND numeric_grade > ANY
       (SELECT numeric_grade
          FROM grade
         WHERE grade_type_code = 'HM'
           AND g.section_id = section_id
           AND g.student_id = student_id)
STUDENT_ID SECTION_ID NUMERIC_GRADE
---------- ---------- -------------
       102         86            85
       103         81            91
       143         85            92
...
       283         99            85
       283        101            88

157 rows selected.
```

Examine the grades for the homework and the final examination for STUDENT_ID 102 and SECTION_ID 86. This student's final

examination grade of 85 is better than the homework grade of 82. The ANY operator tests for an OR condition, so the student and section are returned because only one of the homework grades has to satisfy the condition.

```
SELECT student_id, section_id, grade_type_code,
       numeric_grade
  FROM grade
 WHERE student_id = 102
   AND section_id = 86
   AND grade_type_code IN ('HM', 'FI')
 GROUP BY student_id, section_id, grade_type_code,
       numeric_grade
STUDENT_ID SECTION_ID GR NUMERIC_GRADE
---------- ---------- -- -------------
       102         86 FI             85
       102         86 HM             82
       102         86 HM             82
       102         86 HM             90
       102         86 HM             99

5 rows selected.
```

372
374

d) Based on the result of exercise c, explain the result of the row with STUDENT_ID 102 and SECTION_ID 89.

ANSWER: This record is not returned because unlike the ALL operator, the ANY operator returns false.

The following example illustrates the effect of no records in the subquery on the ANY operator. STUDENT_ID 102 enrolled in SECTION_ID 89 has no homework grades, and, therefore, does not appear in the result set for question c.

```
SELECT student_id, section_id, grade_type_code,
       numeric_grade
  FROM grade
 WHERE student_id = 102
   AND section_id = 89
   AND grade_type_code IN ('HM', 'FI')
STUDENT_ID SECTION_ID GR NUMERIC_GRADE
---------- ---------- -- -------------
       102         89 FI             92

1 row selected.
```

374

375

Lab 8.4 Quiz

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

1) Are the operators NOT IN and <> ANY equivalent, as illustrated in the following example?

SELECT “TRUE’

FROM dual

WHERE 6 <> ANY (6, 9)

```
SELECT 'TRUE'
  FROM dual
WHERE 6 NOT IN (6, 9)
```

_____**a)** Yes

_____**b)** No

2) The following queries are logically equivalent.

```
SELECT 'TRUE'
  FROM dual
WHERE 6 IN (6, 9)
```

```
SELECT 'TRUE'
  FROM dual
WHERE 6 = ANY (6,9)
```

_____**a)** True

_____**b)** False

3) The operators ANY and SOME are equivalent.

_____**a)** True

_____**b)** False

4) To perform any >=, <=, >, or < comparison with a subquery returning multiple rows, you need to use either the ANY, SOME, or ALL operator.

_____**a)** True

_____**b)** False

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) Using a subquery construct, determine which sections the student Henry Masser is enrolled in.

2) What problem does the following SELECT statement solve?

```
SELECT zip
  FROM zipcode z
 WHERE NOT EXISTS
       (SELECT '*' FROM
       student WHERE z.zip =
       zip)
   AND NOT EXISTS (SELECT '*'
       FROM instructor WHERE
       z.zip = zip)
```

3) Display the course number and description of courses with no enrollment. Also include courses that have no section assigned.

4) Can the ANY and ALL operators be used on the DATE data type? Write a simple query to prove your answer.

5) If you have a choice to write either a correlated subquery or a simple noncorrelated subquery, which one would you choose? Why?

6) Determine the top three zip codes where most of the students live.