

INTRODUCTION

Suppose that the management of JustLee Books requests a list of every computer book that has a higher retail price than the book *Database Implementation*. In previous chapters, you would have followed a procedure including two separate queries: 1) Query the database to determine the retail price of *Database Implementation*, and then 2) create a second SELECT statement to find the titles of all computer books retailing for more than *Database Implementation*. In this chapter, you learn how to use an alternative approach, called a subquery, to get the same output by using a single SQL statement.

A **subquery** is a nested query—one complete query inside another query.

The subquery's output can consist of a single value (a single-row subquery), several rows of values (a multiple-row subquery), or even multiple columns of data (a multiple-column subquery). This chapter addresses each type of subquery. In addition, the final section returns to the topic of DML and introduces advanced DML actions, including subqueries and the MERGE statement. With the MERGE statement, you can conditionally process multiple DML actions with a single SQL statement. Table 12-1 gives you an overview of this chapter's contents.

TABLE 12-1 Topics Covered in This Chapter

Subquery	Description
Single-row subquery	Returns to the outer query one row of results consisting of one column
Multiple-row subquery	Returns to the outer query more than one row of results
Multiple-column subquery	Returns to the outer query more than one column of results
Correlated subquery	References a column in the outer query and executes the subquery once for every row in the outer query
Uncorrelated subquery	Executes the subquery first and passes the value to the outer query
DML subquery	Uses a subquery to determine the rows affected by the DML action
MERGE statement	Conditionally processes a series of DML statements

DATABASE PREPARATION

Before attempting to work through the examples in this chapter, run the `JLDB_Build_12.sql` script to make the necessary additions to the JustLee Books database. You should have already executed the `JLDB_Build_8.sql` script as instructed in Chapter 8.

SUBQUERIES AND THEIR USES

As described earlier, getting an answer to a query sometimes requires a multistep operation. First, you must create a query to determine a value you don't know but that's stored in the database. This first query is the subquery. The subquery's results are passed as input to the **Outer query** (also called the **parent query**). The outer query incorporates this value into its calculations to determine the final output.

Although subqueries are used most commonly in the WHERE or HAVING clause of a SELECT statement, at times using a subquery in the SELECT or FROM clause is appropriate. When the subquery is nested in a WHERE or HAVING clause, the results it returns are used as a condition in the outer query. Any type of subquery (single-row, multiple-row, or multiple-column) can be used in the WHERE, HAVING, or FROM clause of a SELECT statement. As you'll see, the only type of subquery that can be used in a SELECT clause is a single-row subquery.

NOTE

The indentation in Figure 12-3's subquery and in other figures in this chapter is used only to improve readability; it isn't required by Oracle 12c.

451

Keep the following rules in mind when working with any type of subquery:

- A subquery must be a *complete query in itself*—in other words, it must have at least a SELECT and a FROM clause.
- A subquery, except one in the FROM clause, can't have an ORDER BY clause. If you need to display output in a specific order, include an ORDER BY clause as the outer query's last clause.
- A subquery must be *enclosed in parentheses* to separate it from the outer query.
- If you place a subquery in the outer query's WHERE or HAVING clause, you can do so only on the *right side* of the comparison operator.

SINGLE-ROW SUBQUERIES

A **single-row subquery** is used when the outer query's results are based on a single, unknown value. Although this query type is formally called "single-row," the name implies that the query returns multiple columns—but only one row—of results. However, a single-row subquery can return *only one row of results consisting of only one column* to the outer query. Therefore, this textbook refers to the output of a single-row subquery as a **single value**.

Single-Row Subquery in a WHERE Clause

To see how subqueries work, you can compare creating multiple queries, which you've studied in previous chapters, with creating one query containing a subquery. In this

chapter's introduction, management requested a list of all computer books with a higher retail price than the book *Database Implementation*. As shown in Figure 12-1, the first step is to create a query to determine the book's retail price, which is \$31.40.

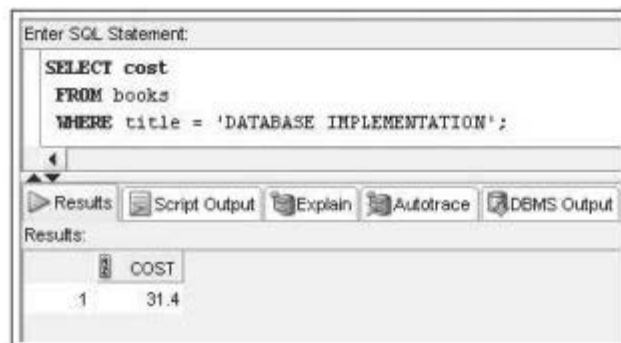


FIGURE 12-1 Query to determine the retail price of *Database Implementation*

To determine which computer books retail for more than \$31.40, you must issue a second query stating the cost of *Database Implementation* in the WHERE clause condition, as shown in Figure 12-2. The WHERE clause includes the retail price of *Database Implementation*, which the query in Figure 12-1 found. The category condition also restricts records to those only in the Computer category.

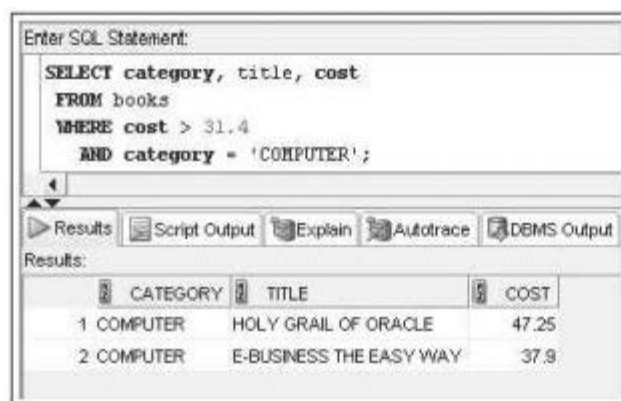
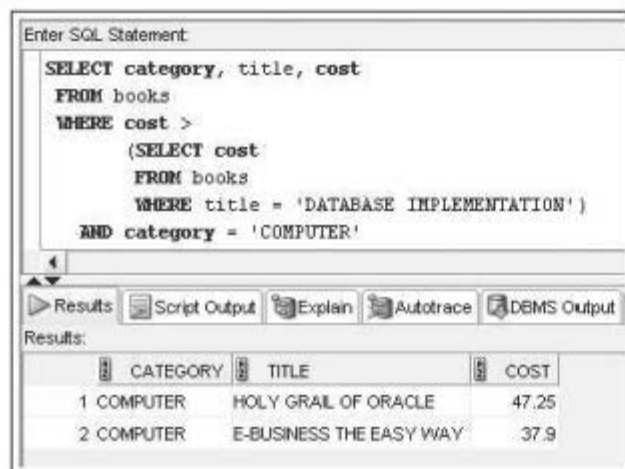


FIGURE 12-2 Query for computer books costing more than \$31.40

You can get these same results with a single SQL statement by using a single-row subquery. A single-row subquery is appropriate in this example because 1) to get the results you need, an unknown value that's stored in the database must be found, and 2) only one value should be returned from the inner query (the retail price of *Database Implementation*).

In Figure 12-3, a single-row subquery is substituted for the `cost > 31.4` condition of the `SELECT` statement in Figure 12-2. This subquery is enclosed in parentheses to distinguish it from the outer query's clauses.



The screenshot shows an SQL IDE window titled "Enter SQL Statement:". The query entered is:

```
SELECT category, title, cost
FROM books
WHERE cost >
  (SELECT cost
   FROM books
   WHERE title = 'DATABASE IMPLEMENTATION')
AND category = 'COMPUTER'
```

Below the query editor, there are buttons for "Results", "Script Output", "Explain", "Autotrace", and "DBMS Output". The "Results" button is selected, and the results are displayed in a table:

	CATEGORY	TITLE	COST
1	COMPUTER	HOLY GRAIL OF ORACLE	47.25
2	COMPUTER	E-BUSINESS THE EASY WAY	37.9

FIGURE 12-3 A single-row subquery

In Figure 12-3, the inner query is executed first, and this query's result, a single value of 31.4, is passed to the outer query. The outer query is then executed, and all books having a retail price greater than \$31.40 and belonging to the Computer category are listed in the output. Using a single SQL statement prevents the need for user intervention to accomplish the task. In addition, the subquery enables this query to always reflect the current price of the *Database Implementation* book.

NOTE

Using a subquery in a `FROM` clause has a specific purpose, as you learn later in "Multiple-Column Subquery in a `FROM` Clause."

Operators indicate to Oracle 12c whether you're creating a single-row subquery or a multiple-row subquery. The single-row operators are `=`, `>`, `<`, `>=`, `<=`, and `<>`. Although other operators, such as `IN`, are allowed, single-row operators instruct Oracle 12c that only one value is expected from the subquery. If more than one value is returned, the `SELECT` statement fails, and you get an error message.

Suppose management makes another request: the title of the most expensive book sold by JustLee Books. The `MAX` function covered in Chapter 11 handles this task. You might be tempted to create the query shown in Figure 12-4.

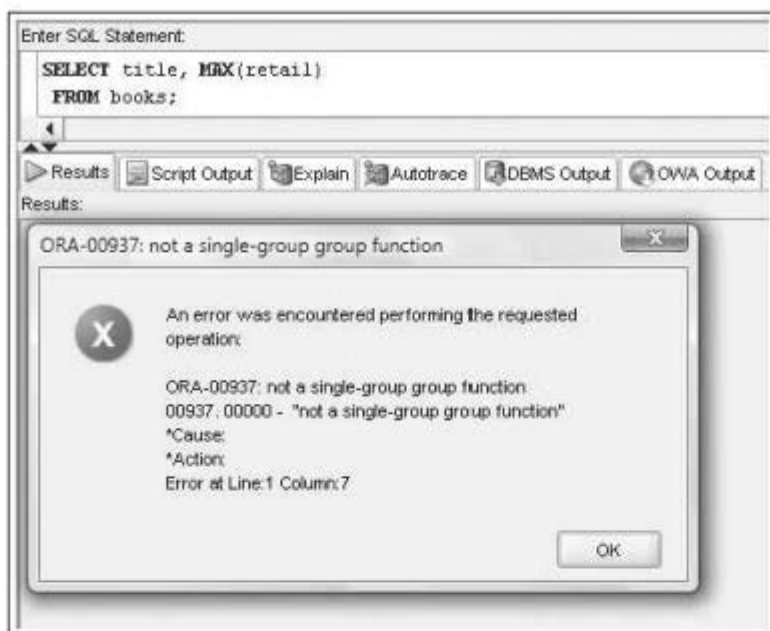


FIGURE 12-4 Flawed query: attempt to determine the book with the highest retail value

You encountered this problem when working with aggregate functions in Chapter 11. Recall this rule when working with group functions: If a nonaggregate field is listed with a group function in the **SELECT** clause, the field must also be listed in a **GROUP BY** clause. In this example, however, adding a **GROUP BY** clause doesn't make sense. If a **GROUP BY** clause containing the **Title** column is added, each book would be its own group because each title is different. In other words, the results would be the same as using **SELECT title, retail** in the query.

Therefore, to retrieve the title of the most expensive book, you can use a subquery to determine the highest retail price of any book. This retail price can then be returned to an outer query and displayed in the results.

As shown in Figure 12-5, the most expensive book sold by JustLee Books is *Painless Child-Rearing*. The book's retail price is included in the query output, but it isn't required. In this case, only one book matches the highest price of \$89.95; however, multiple book titles could be displayed if more than one book matched the high price.

TIP

The query statement serving as the subquery should be created and executed first by itself. In this way, you can verify that the query produces the expected results before embedding it in another query as a subquery.



FIGURE 12-5 Query to determine the title of the most expensive book

CAUTION

If an error message is returned for the query in Figure 12-5, make sure the subquery contains four parentheses—one set around the `retail` argument for the `MAX` function and one set around the subquery.

455

You can include multiple subqueries in a `SELECT` statement. For example, suppose management needs to know the title of all books published by the publisher of *Big Bear and Little Dove* that generate more than the average profit returned by all books sold by JustLee Books. In this case, two values are unknown: the identity of the publisher of *Big Bear and Little Dove* and the average profit of all books. How might you create a query that extracts these values? The `SELECT` statement in Figure 12-6 uses two separate subqueries in the `WHERE` clause to find the information.

Notice that both subqueries in Figure 12-6 are complete because they contain a minimum of one `SELECT` clause and one `FROM` clause. Because they are subqueries, each one is enclosed in parentheses. The first subquery determines the publisher of *Big Bear and Little Dove* and returns the result to the first condition of the `WHERE` clause (`WHERE pubid =`). The second subquery finds the average profit of all books sold by JustLee Books by using the `AVG` function, and then passes this value to the second condition of the `WHERE` clause (`AND retail-cost >`) to be compared against the profit for each book. Because the two conditions of the outer query's `WHERE` clause are combined with the `AND` logical operator, both values returned by the subqueries must be met for a book to be listed in the outer query's output. In this example, the query finds two books published by the publisher of *Big Bear and Little Dove* that return more than the average profit.



FIGURE 12-6 SELECT statement with two single-row subqueries

Single-Row Subquery in a HAVING Clause

As mentioned, you can include a subquery in a HAVING clause. A HAVING clause is used when the group results of a query need to be restricted based on some condition. If a subquery's result must be compared with a group function, you must nest the inner query in the outer query's HAVING clause.

For example, if management needs a list of all book categories returning a higher average profit than the Literature category, you would follow these steps:

1. Calculate the average profit for all Literature books.
2. Calculate the average profit for each category.
3. Compare the average profit for each category with the average profit for the Literature category.

To perform these steps, you use the AVG function and include a subquery in the HAVING clause, as shown in Figure 12-7. The results are restricted to groups having a higher average profit than the Literature category. Because the subquery's results are applied to groups of data, nesting the subquery in the HAVING clause is necessary. (Recall from Chapter 11 that filtering by aggregate data occurs in the HAVING clause, not the WHERE clause.)

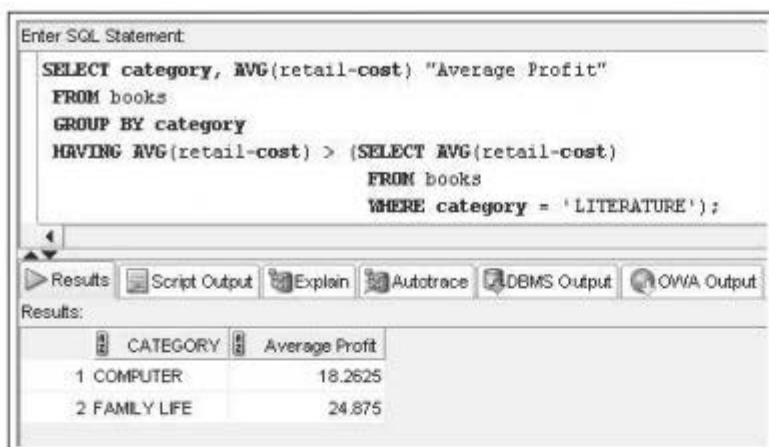


FIGURE 12-7 Single-row subquery nested in a HAVING clause

As Figure 12-7 shows, the subquery in a HAVING clause must follow the same guidelines as for WHERE clauses: It must include at least a SELECT clause and a FROM clause, and it must be enclosed in parentheses.

Single-Row Subquery in a SELECT Clause

A single-row subquery can also be nested in the outer query's SELECT clause. In this case, the value the subquery returns is available for every row of output the outer query generates. Typically, this technique is used to perform calculations with a value produced from a subquery. For example, suppose management wants to compare the price of each book in inventory against the average price of all books in inventory. The query output must show each book's price and the amount above or below the average.

You can accomplish this task by using a subquery in a SELECT clause that calculates the average retail price of all books. When a single-row subquery is included in a SELECT clause, the subquery's results are displayed in the outer query's output. To include a subquery in a SELECT clause, you use a comma to separate the subquery from the table columns, as though you were listing another column. In fact, you can even give the subquery results a column alias. First, use the query in Figure 12-8 to verify that the same average amount the subquery returns is available for each row in the parent query. The TO_CHAR function is used to round the average amount to two decimal places.

Enter SQL Statement:		
<pre>SELECT title, retail, (SELECT TO_CHAR(AVG(retail),999.99) FROM books) "Overall Average" FROM books;</pre>		
Results Script Output Explain Autotrace DBMS Output OWA Output		
Results:		
	TITLE	RETAIL Overall Average
1	BODYBUILD IN 10 MINUTES A DAY	30.95 40.98
2	REVENGE OF MICKEY	22 40.98
3	BUILDING A CAR WITH TOOTHPICKS	59.95 40.98
4	DATABASE IMPLEMENTATION	55.95 40.98
5	COOKING WITH MUSHROOMS	19.95 40.98
6	HOLY GRAIL OF ORACLE	75.95 40.98
7	HANDCRANKED COMPUTERS	25 40.98
8	E-BUSINESS THE EASY WAY	54.5 40.98
9	PAINLESS CHILD-REARING	89.95 40.98
10	THE WOK WAY TO COOK	28.75 40.98
11	BIG BEAR AND LITTLE DOVE	8.95 40.98
12	HOW TO GET FASTER PIZZA	29.95 40.98
13	HOW TO MANAGE THE MANAGER	31.95 40.98
14	SHORTEST POEMS	39.95 40.98

FIGURE 12-8 Single-row subquery in a SELECT clause

To calculate the average price of all books in inventory, the outer query's SELECT clause includes the Title and Retail columns as well as the subquery in the column list. The average calculated by the subquery is displayed for every book included in the output. The column alias, Overall Average, is assigned to the subquery's results to indicate the column's contents. If a column alias isn't used, the actual subquery shows as the column heading, which is somewhat unattractive and not very descriptive. Having the subquery in the SELECT clause enables management to compare each book's retail price to the average retail price for all books by looking at just one list.

Can this subquery be used in a calculation to determine the difference between each book price and the average? Absolutely. Simply move the subquery into a calculation, as shown in Figure 12-9, to calculate the difference between the retail price and the average price.

Enter SQL Statement:			
<pre>SELECT title, retail, retail-(SELECT TO_CHAR(AVG(retail),999.99) FROM books) "Diff from AVG" FROM books;</pre>			
Results Script Output Explain Autotrace DEMS Output OWA Output			
Results:			
	TITLE	RETAIL	Diff from AVG
1	BODYBUILD IN 10 MINUTES A DAY	30.95	-10.03
2	REVENGE OF MICKEY	22	-18.98
3	BUILDING A CAR WITH TOOTHPICKS	59.95	18.97
4	DATABASE IMPLEMENTATION	55.95	14.97
5	COOKING WITH MUSHROOMS	19.95	-21.03
6	HOLY GRAIL OF ORACLE	75.95	34.97
7	HANDCRANKED COMPUTERS	25	-15.98
8	E-BUSINESS THE EASY WAY	54.5	13.52
9	PAINLESS CHILD-REARING	89.95	48.97
10	THE WOK WAY TO COOK	28.75	-12.23
11	BIG BEAR AND LITTLE DOVE	8.95	-32.03
12	HOW TO GET FASTER PIZZA	29.95	-11.03
13	HOW TO MANAGE THE MANAGER	31.95	-9.03
14	SHORTEST POEMS	39.95	-1.03

FIGURE 12-9 Use a subquery in a calculation in the SELECT clause

TIP

Users learning to work with subqueries often don't have much confidence in the output when the subquery is in a WHERE clause because the value the subquery generates isn't displayed. However, if the subquery used in the WHERE clause is also included in the SELECT clause, the value the single-row subquery generates can be compared against the outer query's final output. This comparison is an easy way to validate the output. The subquery is removed from the SELECT clause after validation to generate the requested results. Validation gives you more confidence in the final results and reduces the risk of distributing erroneous data. However, this method works only for single-row subqueries. For other types of subqueries, you must execute the subquery as a separate SELECT statement to determine the values it generates because a SELECT clause can process only single-row subqueries.

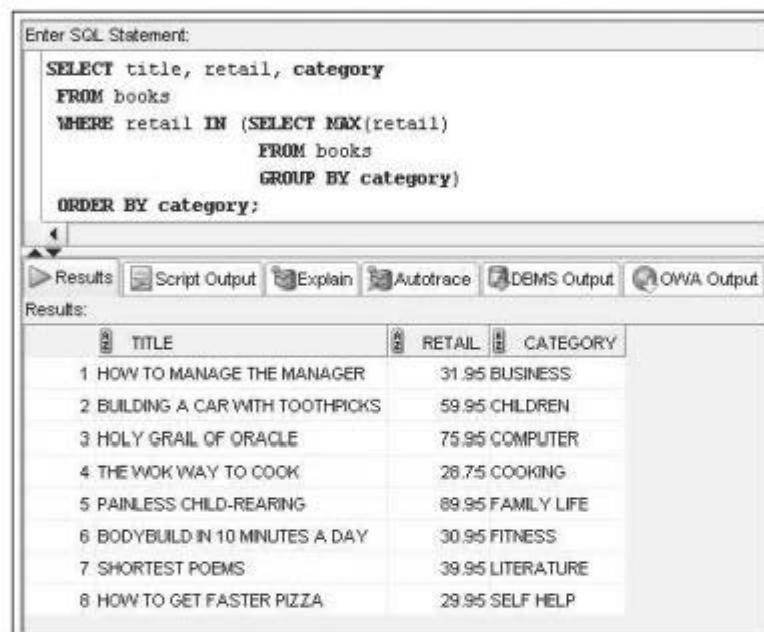
MULTIPLE-ROW SUBQUERIES

Multiple-row subqueries are nested queries that can return more than one row of results to the parent query. Multiple-row subqueries are used most commonly in WHERE and HAVING clauses. The main rule to keep in mind when working with multiple-row

subqueries is that you *must* use multiple-row operators. If a single-row operator is used with a subquery that returns more than one row of results, Oracle 12c returns an error message, and the SELECT statement fails. Valid multiple-row operators include IN, ALL, and ANY, discussed in the following sections.

The IN Operator

Of the three multiple-row operators, the IN operator is used most often. Figure 12-10 shows a multiple-row subquery with this operator. This query identifies books with a retail value matching the highest retail value for any book category.



The screenshot shows an Oracle SQL Developer interface. At the top, a text area labeled 'Enter SQL Statement:' contains the following SQL query:

```
SELECT title, retail, category
FROM books
WHERE retail IN (SELECT MAX(retail)
                 FROM books
                 GROUP BY category)
ORDER BY category;
```

Below the text area is a toolbar with buttons for 'Results', 'Script Output', 'Explain', 'Autotrace', 'DBMS Output', and 'OWA Output'. The 'Results' button is selected, and below it, a table of results is displayed. The table has three columns: 'TITLE', 'RETAIL', and 'CATEGORY'. It contains eight rows of data, each representing a book.

	TITLE	RETAIL	CATEGORY
1	HOW TO MANAGE THE MANAGER	31.95	BUSINESS
2	BUILDING A CAR WITH TOOTHPICKS	59.95	CHILDREN
3	HOLY GRAIL OF ORACLE	75.95	COMPUTER
4	THE WOK WAY TO COOK	28.75	COOKING
5	PAINLESS CHILD-REARING	89.95	FAMILY LIFE
6	BODYBUILD IN 10 MINUTES A DAY	30.95	FITNESS
7	SHORTEST POEMS	39.95	LITERATURE
8	HOW TO GET FASTER PIZZA	29.95	SELF HELP

FIGURE 12-10 Multiple-row subquery with the IN operator

The IN operator in this subquery indicates that the records the outer query processes must match one of the values the subquery returns. (In other words, it creates an OR condition.) The order of execution in Figure 12-10 is as follows:

1. The subquery determines the price of the most expensive book in each category.
2. The maximum retail price in each category is passed to the WHERE clause of the outer query (a list of values).
3. The outer query compares each book's price to the prices generated by the subquery.
4. If a book's retail price matches one of the prices returned by the subquery, the book's title, retail price, and category are displayed in the output.

The ALL and ANY Operators

The ALL and ANY operators can be combined with other comparison operators to treat a subquery's results as a set of values instead of single values. Table 12-2 summarizes the use of the ALL and ANY operators with other comparison operators.

TABLE 12-2 ALL and ANY Operator Combinations

Operator	Description
>ALL	More than the highest value returned by the subquery
<ALL	Less than the lowest value returned by the subquery
<ANY	Less than the highest value returned by the subquery
>ANY	More than the lowest value returned by the subquery
=ANY	Equal to any value returned by the subquery (same as IN)

461

The ALL operator is fairly straightforward:

- If the ALL operator is combined with the “greater than” symbol (>), the outer query searches for all records with a value higher than the highest value returned by the subquery (in other words, more than ALL the values returned).
- If the ALL operator is combined with the “less than” symbol (<), the outer query searches for all records with a value lower than the lowest value returned by the subquery (in other words, less than ALL the values returned).

To examine the impact of using the ALL comparison operator, look at the query in Figure 12-11, which will be used later as a subquery. It returns the retail prices for two books in the Cooking category. The lowest value returned is \$19.95, and the highest value is \$28.75.



The screenshot shows a SQL interface with the following components:

- Enter SQL Statement:**

```
SELECT retail
FROM books
WHERE category = 'COOKING';
```
- Buttons:** Results, Script Output, Explain, Autotrace, DBMS Output.
- Results:** A table with two columns: an index and RETAIL.

	RETAIL
1	19.95
2	28.75

FIGURE 12-11 Retail price of books in the Cooking category

Suppose you want to know the titles of all books having a retail price greater than the most expensive book in the Cooking category. One approach is using the MAX function in

a subquery to find the highest retail price. Another approach is using the `> ALL` operator, as shown in Figure 12-12.

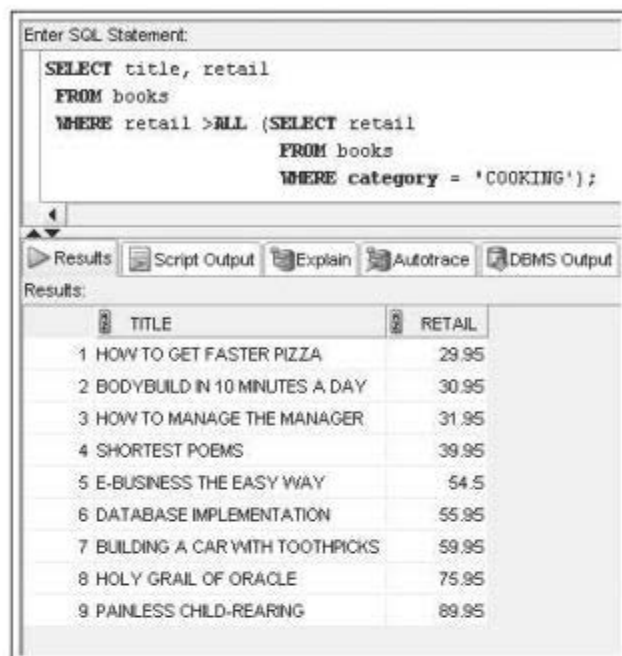


FIGURE 12-12 Using the `> ALL` operator

The Oracle 12c strategy for processing the `SELECT` statement in Figure 12-12 is as follows:

- The subquery passes the retail prices of the two books in the Cooking category (\$19.95 and \$28.75) to the outer query.
- Because the `> ALL` operator is used in the outer query, Oracle 12c is instructed to list all books with a retail price higher than the largest value returned by the subquery (\$28.75).

In this example, nine books have a higher price than the most expensive book in the Cooking category.

TIP

You could get the same results as in Figure 12-12 by using the `MAX` function in the subquery. In this case, a single value for the highest priced book in the Cooking category is returned from the subquery, and a multiple-row operator isn't required.

Similarly, the `< ALL` operator is used to determine records with a value less than the lowest value returned by a subquery. Therefore, if you need to find books priced lower than the least expensive book in the Cooking category, first formulate a subquery that identifies

books in the Cooking category. Then you can compare the retail price of books in the BOOKS table against the values returned by a subquery by using the < ALL operator.

As in the previous query, the subquery in Figure 12-13 first finds the two books in the Cooking category (shown in Figure 12-11). The retail prices of these books (\$19.95 and \$28.75) are then passed to the outer query. Because \$19.95 is the lowest retail price of all books in the Cooking category, only books with a retail price lower than \$19.95 are displayed in the output. In this case, Oracle 12c found only one book with a retail price lower than the least expensive book in the Cooking category: *Big Bear and Little Dove*.



FIGURE 12-13 Using the < ALL operator

By contrast, the < ANY operator is used to find records with a value less than the highest value returned by a subquery. To determine which books cost less than the most expensive book in the Cooking category, evaluate the subquery's results by using the < ANY operator, as shown in Figure 12-14.

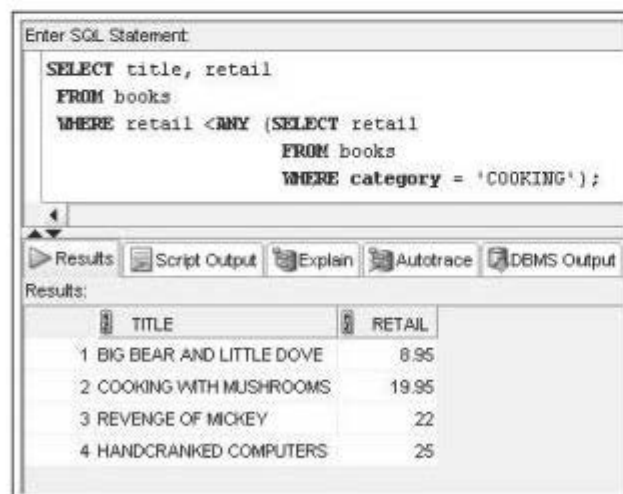


FIGURE 12-14 Using the < ANY operator

The outer query finds four books with a retail price lower than the most expensive book in the Cooking category. Notice, however, that the results also include *Cooking with Mushrooms*, a book in the Cooking category. Because the outer query compares the records to the highest value in the Cooking category, any other book in the Cooking category is also displayed in the query results. To eliminate any book in the Cooking category from appearing in the output, simply add the condition `AND category <> 'COOKING'` to the outer query's WHERE clause.

The `> ANY` operator is used to return records with a value greater than the lowest value returned by the subquery. In Figure 12-15, 12 records have a retail price greater than the lowest retail price returned by the subquery (\$19.95).

Enter SQL Statement	
<pre>SELECT title, retail FROM books WHERE retail >ANY (SELECT retail FROM books WHERE category = 'COOKING');</pre>	
Results Script Output Explain Autotrace DBMS Output	
Results:	
ID	TITLE
1	PAINLESS CHILD-REARING
2	HOLY GRAIL OF ORACLE
3	BUILDING A CAR WITH TOOTHPICKS
4	DATABASE IMPLEMENTATION
5	E-BUSINESS THE EASY WAY
6	SHORTEST POEMS
7	HOW TO MANAGE THE MANAGER
8	BODYBUILD IN 10 MINUTES A DAY
9	HOW TO GET FASTER PIZZA
10	THE WOK WAY TO COOK
11	HANDCRANKED COMPUTERS
12	REVENGE OF MICKEY
	RETAIL
	89.95
	75.95
	59.95
	55.95
	54.5
	39.95
	31.95
	30.95
	29.95
	28.75
	25
	22

FIGURE 12-15 Using the `> ANY` operator

The `= ANY` operator works the same way as the `IN` comparison operator. For example, the query in Figure 12-16 searches for the titles of books purchased by customers who also purchased the book with the ISBN 0401140733. Because this book could have appeared on more than one order, and the query is supposed to identify all these orders, the `= ANY` operator is used.

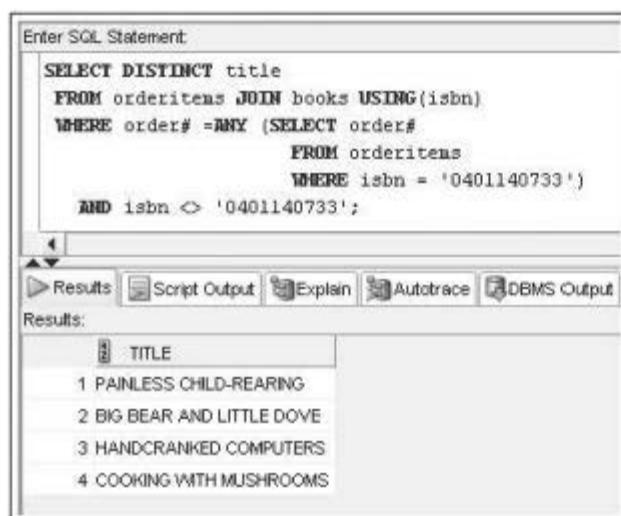


FIGURE 12-16 Using the =ANY operator

This query would have yielded the same results if the IN operator had been used instead of the =ANY operator. The DISTINCT keyword in the outer query's SELECT clause is included because, as mentioned, a title could have been ordered by more than one customer and would have multiple listings in the output.

TIP

If you don't get the same results as in Figure 12-16, make sure the closing parenthesis for the subquery is placed before the last line beginning with the AND keyword, which is part of the outer query.

Also, notice in Figure 12-16 that the columns needed to complete the outer query are in two different tables: ORDERITEMS and BOOKS. A join is required in the outer query to combine the rows of these two tables. Because the columns needed to perform the inner query are contained only in the ORDERITEMS table, no join is required in the subquery.

NOTE

Another operator, EXISTS, is available to handle multiple-row subqueries and is discussed later in "Correlated Subqueries."

Multiple-Row Subquery in a HAVING Clause

So far, you have seen multiple-row subqueries in a WHERE clause, but they can also be included in a HAVING clause. When the subquery's results are compared to grouped data in the outer query, the subquery *must* be nested in a HAVING clause in the outer query.

For example, you need to determine whether any customer's recently placed order has a total amount due greater than the total amount due for every order placed recently by customers in Florida. Getting this output requires determining the total amount due for each order placed by a Florida customer, and then comparing these totals with every order total. The order totals for Florida customers can be calculated in a subquery, but because one value is returned for each order, you need a multiple-row subquery. These totals need to be compared with the total amount due for each order, which requires the outer query to group all items in the ORDERITEMS table by the Order# column. Therefore, the outer query must use a HAVING clause because the comparison is based on grouped data.

As shown in Figure 12-17, the structure for using a multiple-row subquery in a HAVING clause is the same as using the subquery in a WHERE clause.

Enter SQL Statement:

```
SELECT order#, SUM(quantity*paideach)
FROM orderitems
HAVING SUM(quantity*paideach) > ALL (SELECT SUM(quantity*paideach)
                                     FROM customers JOIN orders USING (customer#)
                                     JOIN orderitems USING (order#)
                                     WHERE state = 'FL'
                                     GROUP BY order#)
GROUP BY order#;
```

Results: Script Output Explain Autotrace DBMS Output OWA Output

	ORDER#	SUM(QUANTITY*PAIDEACH)
1	1001	117.4
2	1002	111.9
3	1007	335.85
4	1004	170.9
5	1012	166.4

FIGURE 12-17 Multiple-row subquery in a HAVING clause

Single-row and multiple-row subqueries might look the same in terms of the subqueries themselves; however, a single-row subquery can return only *one* data value, whereas a multiple-row subquery can return *several* values. Therefore, if you execute a subquery that returns more than one data value and the comparison operator is intended to be used only with single-row subqueries, you get an error message and the query isn't executed, as shown in Figure 12-18.

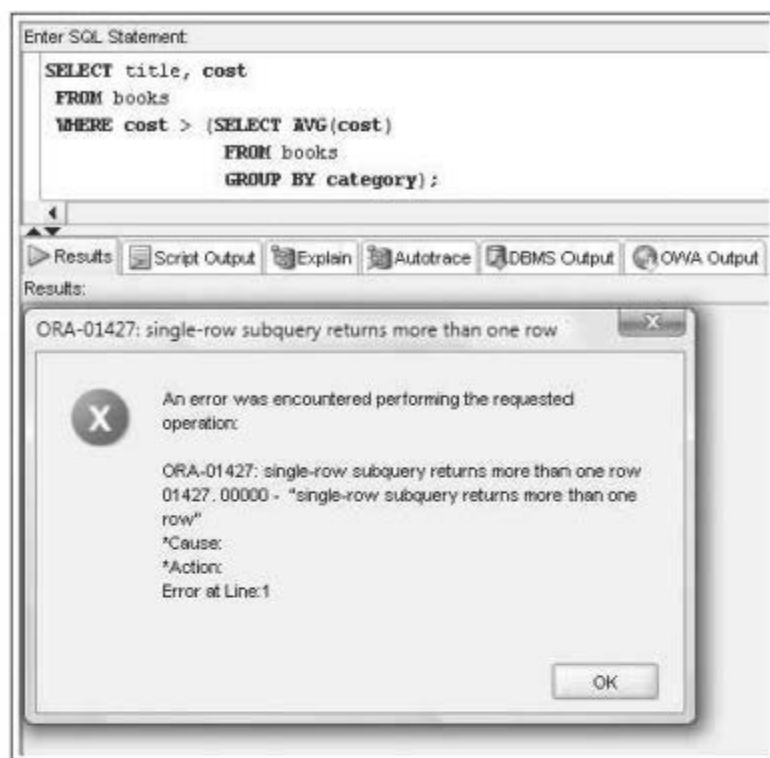


FIGURE 12-18 Flawed query: using a single-row operator for a multiple-row subquery

MULTIPLE-COLUMN SUBQUERIES

Now that you've examined multiple-row subqueries, this section explores multiple-column subqueries. A **multiple-column subquery** returns more than one column to the outer query and can be listed in the outer query's FROM, WHERE, or HAVING clause.

Multiple-Column Subquery in a FROM Clause

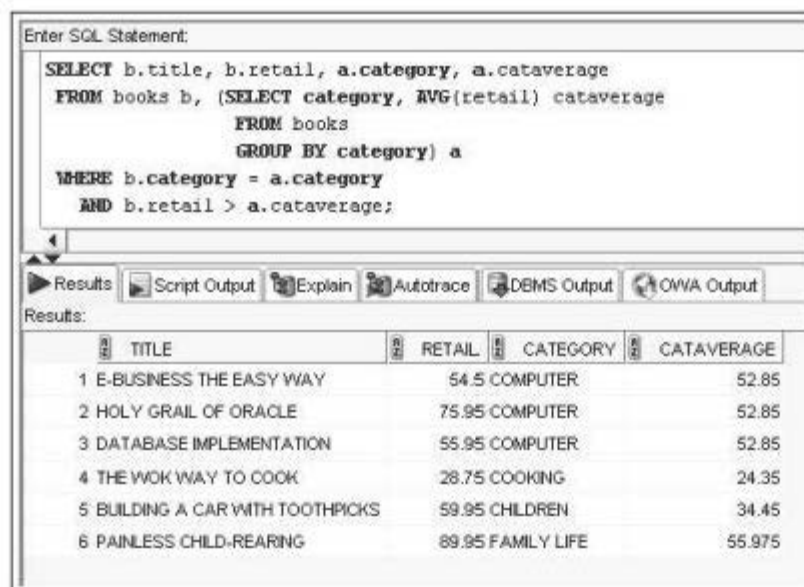
When a multiple-column subquery is used in the outer query's FROM clause, it creates a temporary table that can be referenced by other clauses of the outer query. This temporary table is more formally called an **inline view**. The subquery's results are treated like any other table in the FROM clause. If the temporary table contains grouped data, the grouped subsets are treated as separate rows of data in a table.

NOTE

Views are covered in Chapter 13.

Suppose you need a list of all books in the BOOKS table that have a higher-than-average selling price compared with other books in the same category. For each book, you need to display the title, retail price, category, and average selling price of books in that category. Because the average selling price is based on grouped data, this query presents a problem. How might you solve it?

In Figure 12-19, a multiple-column subquery is nested in the outer query's FROM clause. The subquery creates a temporary table, including a column for the category and a column for the category average. The subquery determines the categories in the BOOKS table and the average selling price of every book in each category.



Enter SQL Statement:

```
SELECT b.title, b.retail, a.category, a.cataverage
FROM books b, (SELECT category, AVG(retail) cataverage
FROM books
GROUP BY category) a
WHERE b.category = a.category
AND b.retail > a.cataverage;
```

Results: Script Output Explain Autotrace DBMS Output OWA Output

Results:

	TITLE	RETAIL	CATEGORY	CATAVERAGE
1	E-BUSINESS THE EASY WAY	54.5	COMPUTER	52.85
2	HOLY GRAIL OF ORACLE	75.95	COMPUTER	52.85
3	DATABASE IMPLEMENTATION	55.95	COMPUTER	52.85
4	THE WOK WAY TO COOK	28.75	COOKING	24.35
5	BUILDING A CAR WITH TOOTHPICKS	59.95	CHILDREN	34.45
6	PAINLESS CHILD-REARING	89.95	FAMILY LIFE	55.975

FIGURE 12-19 Multiple-column subquery in a FROM clause

However, how do you display the title of each book in the BOOKS table along with its retail price, its category, and the average price of all books in the same category? The BOOKS table contains the data for each book, and the subquery creates a temporary table that stores the grouped data. Notice in Figure 12-19 that the table alias “a” has been assigned to the subquery’s results, so the columns in the subquery (Category and Cataverage) can be referenced by other clauses in the outer SELECT statement. Essentially, this alias assigns a table name to the subquery’s results.

The query is referencing, or finding, data from two different tables, and one just happens to be created at runtime by the subquery. The tables have been joined by using the traditional approach—the outer query’s WHERE clause. The problem with the traditional approach is that both tables contain a column called Category, which creates an ambiguity problem if the Category column is referenced anywhere in the outer query. To avoid this problem, the Category column needs a column qualifier to identify which table contains the category data to be displayed. Therefore, table aliases are used in the SELECT and WHERE clauses to identify the table containing the column being referenced.

As shown in Figure 12-20, the query could have also been created by using an ANSI JOIN operation supported by Oracle 12c. Because both tables (BOOKS and the temporary table created by the subquery) contain a column named *Category*, the tables are linked in the FROM clause with a join using the *Category* field. Because column qualifiers aren't allowed with the JOIN statement, the temporary table created by the subquery isn't assigned a table alias.

Enter SQL Statement:				
<pre> SELECT title, retail, category, cataverage FROM books JOIN (SELECT category, AVG(retail) cataverage FROM books GROUP BY category) USING (category) WHERE retail > cataverage; </pre>				
<div> Results Script Output Explain Autotrace DBMS Output OWA Output </div>				
Results:				
	TITLE	RETAIL	CATEGORY	CATAVERAGE
1	E-BUSINESS THE EASY WAY	54.5	COMPUTER	52.85
2	HOLY GRAIL OF ORACLE	75.95	COMPUTER	52.85
3	DATABASE IMPLEMENTATION	55.95	COMPUTER	52.85
4	THE WOK WAY TO COOK	28.75	COOKING	24.35
5	BUILDING A CAR WITH TOOTHPIKES	59.95	CHILDREN	34.45
6	PAINLESS CHILD-REARING	89.95	FAMILY LIFE	55.975

FIGURE 12-20 Using a join with a multiple-column subquery in the FROM clause

Multiple-Column Subquery in a WHERE Clause

When a multiple-column subquery is included in the outer query's WHERE or HAVING clause, the outer query uses the IN operator to evaluate the subquery's results. The subquery's results consist of more than one column of results.

The syntax of the outer WHERE clause is **WHERE (columnname, columnname, . . .) IN subquery**. Keep these rules in mind:

- Because the WHERE clause contains more than one column name, the column list must be enclosed in parentheses.
- Column names listed in the WHERE clause must be in the same order as they're listed in the subquery's SELECT clause.

TIP

Double-check that the column list in the outer query's WHERE clause is enclosed in parentheses and is in the same order as the column list in the subquery's SELECT clause.

In Figure 12-10 shown earlier, the subquery returned the price of the most expensive book in each category, and the outer query generated a list of the title, retail price, and category of books matching the retail price returned by the subquery. The overall result of the outer query was to display the title, retail price, and category for the most expensive book in each category. However, the query results could be misleading because a book retail value that matches a MAX value of any category is included in the output. So books in the output might be matches of the category MAX value of a different category than their own.

To create a query that specifically lists the most expensive books in each category, a multiple-column subquery is more suitable. Look at the example in Figure 12-21. The subquery finds the highest retail value in each category and passes both the category names and retail prices to the outer query.

Enter SQL Statement:

```
SELECT title, retail, category
FROM books
WHERE (category, retail) IN (SELECT category, MAX(retail)
                           FROM books
                           GROUP BY category)
ORDER BY category;
```

Results Script Output Explain Autotrace DBMS Output OWA Output

Results:

	TITLE	RETAIL	CATEGORY
1	HOW TO MANAGE THE MANAGER	31.95	BUSINESS
2	BUILDING A CAR WITH TOOTHPICKS	59.95	CHILDREN
3	HOLY GRAIL OF ORACLE	75.95	COMPUTER
4	THE WOK WAY TO COOK	28.75	COOKING
5	PAINLESS CHILD-REARING	89.95	FAMILY LIFE
6	BODYBUILD IN 10 MINUTES A DAY	30.95	FITNESS
7	SHORTEST POEMS	39.95	LITERATURE
8	HOW TO GET FASTER PIZZA	29.95	SELF HELP

FIGURE 12-21 Multiple-column subquery in a WHERE clause

NOTE

Although a multiple-column subquery can be used in the outer query's HAVING clause, it's usually used only when analyzing extremely large sets of grouped numeric data. Generally, this method is discussed in more advanced courses focusing on quantitative methods.

NULL VALUES

As with everything else, NULL values present a challenge when using subqueries. Because a NULL value is the same as the absence of data, a NULL can't be returned to an outer query for comparison purposes; it's not equal to anything, not even another NULL. Therefore, if a NULL value is passed from a subquery, the results of the outer query are "no rows selected." Although the statement doesn't fail (it doesn't generate an Oracle 12c error message), you don't get the expected results, as you can see from the example in Figure 12-22.

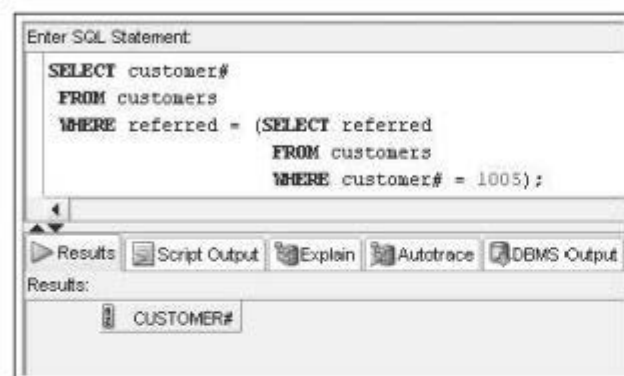


FIGURE 12-22 Flawed query: NULL results from a subquery

In Figure 12-22, the user is trying to determine whether the customer who referred customer 1005 has referred any other customers to JustLee Books. The problem is that no rows are listed as output from the outer query. Are no rows listed because the customer who referred customer 1005 hasn't referred any other customers or because customer 1005 wasn't referred to JustLee Books (in which case the Referred column is NULL)? If no one referred customer 1005, should the outer query's output be a list of all customers who weren't referred by other customers?

In this case, customer 1005 wasn't referred by any other customer; therefore, the Referred column is NULL. A NULL value is passed to the outer query, so no matches are found because the condition is `WHERE referred = NULL`. The `IS NULL` operator is required to identify NULL values in a conditional clause.

What if customer 1005 wasn't referred by another customer and you want a list of all customers who weren't referred by other customers? As always, it's the NVL function to the rescue.

NVL in Subqueries

If it's possible for a subquery to return a NULL value to the outer query for comparison, the NVL function should be used to substitute an actual value for the NULL. However, keep these two rules in mind:

- The substitution of the NULL value must occur for the NULL value in both the subquery and the outer query.
- The value substituted for the NULL value must be one that couldn't possibly exist anywhere else in that column.

Figure 12-23 uses the same premise as Figure 12-22 and shows an example of these two rules. The NVL function is included whenever the Referred column is referenced—in both the subquery and the outer query. In this example, a zero is substituted for a NULL value.

Because the value in the Referred column is actually a customer number in the CUSTOMERS table, and no customer has the customer number zero, substituting a zero for the NULL value doesn't accidentally make a NULL record equivalent to a non-NULL record.

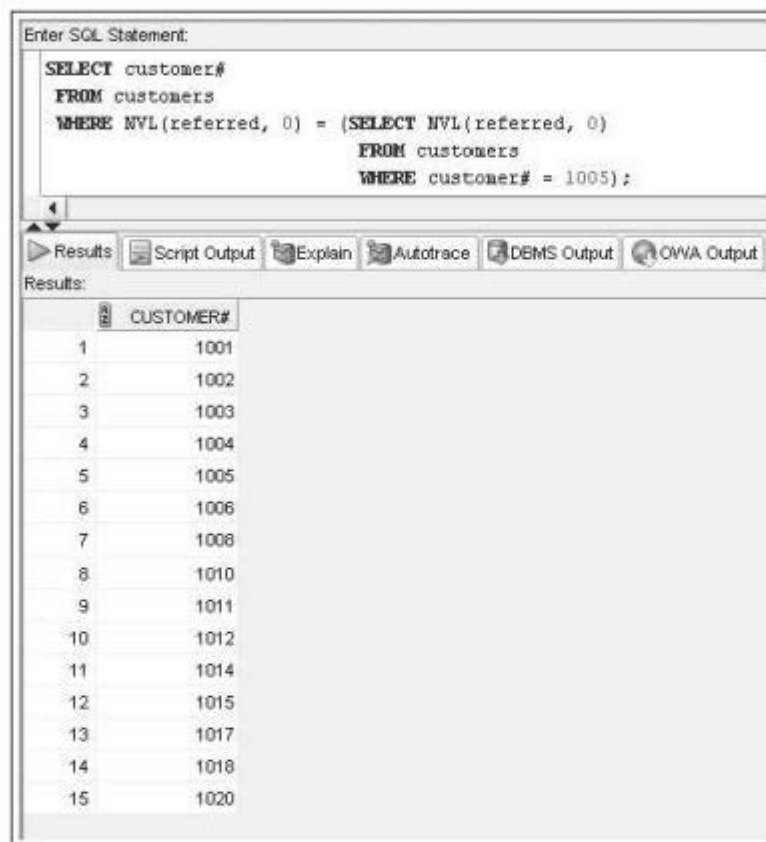


FIGURE 12-23 Using the NVL function to handle NULL values

When you substitute a value for a NULL, make sure no other record contains the substituted value. For example, use ZZZ for a customer name; in a date field, use a date that absolutely couldn't exist in the database.

IS NULL in Subqueries

Although passing a NULL value from a subquery to an outer query can be challenging, searches for NULL values are allowed in a subquery. As with regular queries, you can still search for NULL values with the IS NULL comparison operator.

For example, you need to find the title of all books that have been ordered but haven't shipped yet. The subquery in Figure 12-24 identifies the orders that haven't shipped—in other words, the ship date is NULL.

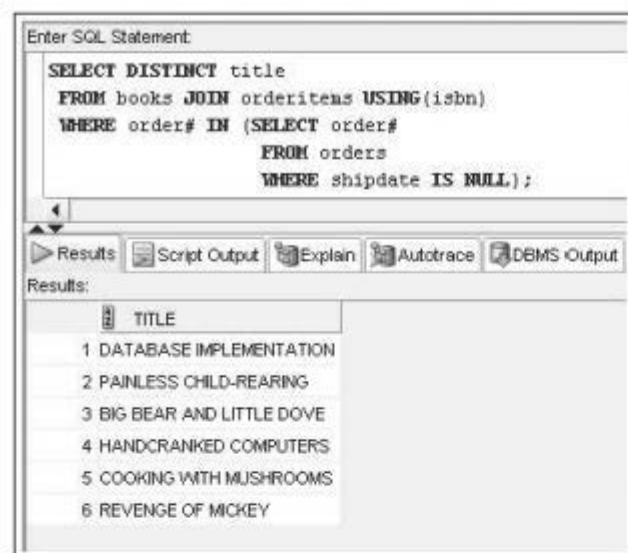


FIGURE 12-24 Using IS NULL in a subquery

As shown, the order number for each order is passed to the outer query, and the title for each book is displayed. The DISTINCT keyword is used to prevent duplicate titles from being listed. Although the subquery searches for records containing NULL values, it's the Order# column that's passed to the outer query. The Order# column is the primary key for the ORDERS table, and no NULL values can exist in this field. Therefore, there's no need to use the NVL function in this example.

CORRELATED SUBQUERIES

So far you have studied mostly **uncorrelated subqueries**: The subquery is executed first, its results are passed to the outer query, and then the outer query is executed. In a **correlated subquery**, Oracle 12c uses a different procedure to execute a query. A **correlated subquery** references one or more columns in the outer query, and the EXISTS operator is used to test whether the relationship or link is present.

Figure 12-25 shows an example of identifying books that have been ordered recently. Although this query is a multiple-row subquery, execution of the entire query requires processing each row in the BOOKS table to determine whether it also exists in the ORDERITEMS table.

Oracle 12c executes the outer query first, and when it encounters the outer query's WHERE clause, it's evaluated to determine whether that row is TRUE (whether it exists in the ORDERITEMS table). If it is TRUE, the book's title is displayed in the results. The outer query is executed again for the next book in the BOOKS table and compared to the ORDERITEMS table's contents, and so on, for each row of the BOOKS table. In other words, a correlated subquery is *processed, or executed, once for each row in the outer query*.

How does Oracle 12c distinguish between an uncorrelated and a correlated subquery? Simply speaking, if a subquery *references a column from the outer query*, it's a correlated

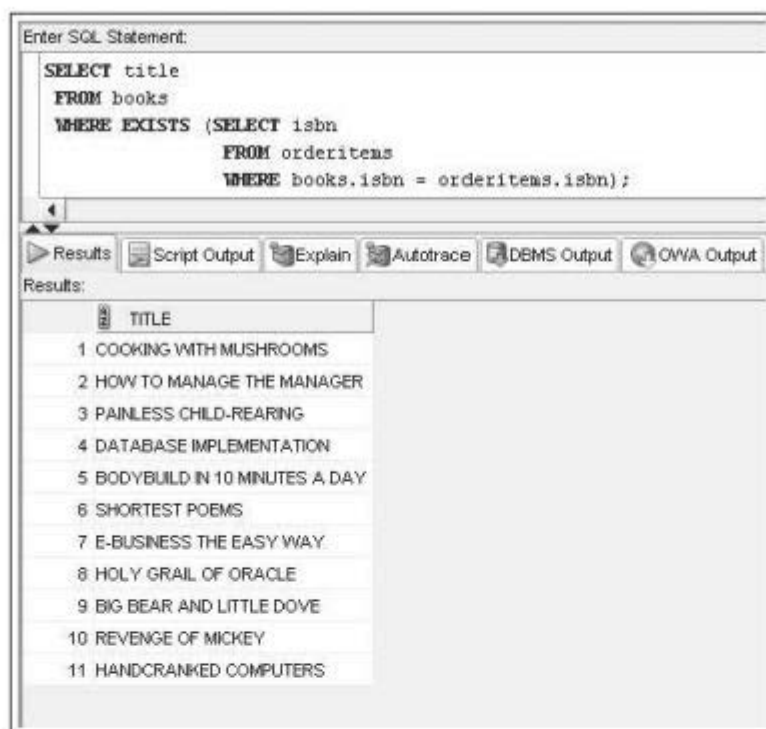


FIGURE 12-25 Correlated subquery

subquery. Notice that in the subquery in Figure 12-25, the WHERE clause specifies the ISBN column of the BOOKS table. Because the BOOKS table isn't included in the subquery's FROM clause, it's forced to use data processed by the outer query (the ISBN of books processed during that execution of the outer query). With an uncorrelated subquery, the subquery is executed first, and then the results are passed to the outer query. Because the subquery is used to identify every ISBN stored in the ORDERITEMS table, each ISBN listed in the table is returned to the outer query.

A join operation could also be used to tackle this query. Notice that in Figure 12-26, a join is used to identify the titles of books that have been ordered recently. Keep in mind that nonmatching rows are dropped from the results automatically in an equijoin.

Only partial results are included in Figure 12-26 because the output includes duplicates. Adding DISTINCT suppresses duplicates in the output so that it matches the results of the correlated subquery in Figure 12-25.

NOTE

You'll continue to discover that several techniques are available to solve query requests. Being familiar with different options is beneficial, as some techniques execute more efficiently in certain situations. Tuning is an advanced topic beyond this textbook's scope; however, the concept is introduced in Appendix E.

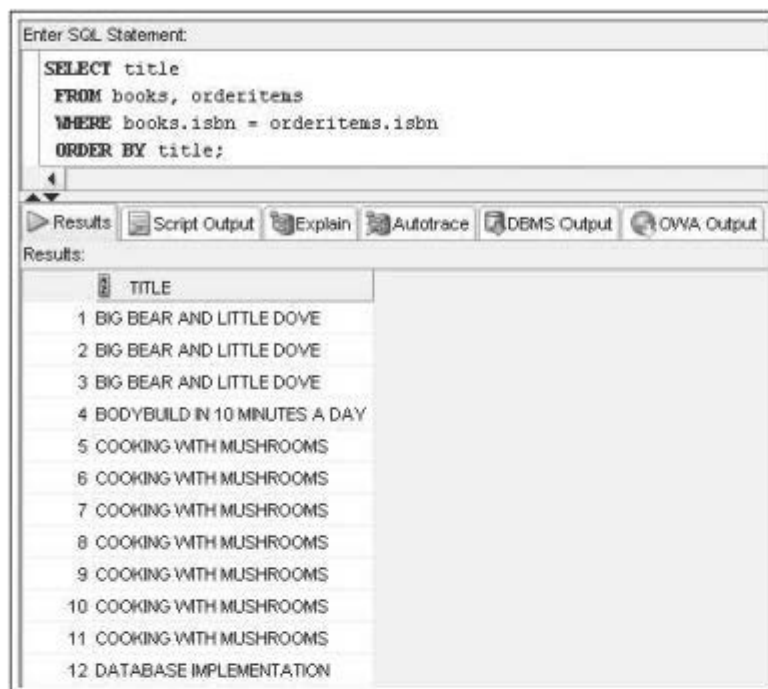


FIGURE 12-26 Using a join rather than a correlated subquery

NESTED SUBQUERIES

You can nest subqueries inside the FROM, WHERE, or HAVING clauses of other subqueries. In Oracle 12c, subqueries in a WHERE clause can be nested to a depth of 255 subqueries, and there's no depth limit when subqueries are nested in a FROM clause.

When nesting subqueries, you might want to use the following strategy:

- Determine exactly what you're trying to find—in other words, the goal of the query.
- Write the innermost subquery first.
- Next, look at the value you can pass to the outer query. If it isn't the value the outer query needs (for example, it references the wrong column), analyze how you need to convert the data to get the correct rows. If necessary, use another subquery between the outer query and the nested subquery. In some cases, you might need to create several layers of subqueries to link the value the innermost subquery returns to the value the outer query needs.

The most common reason for nesting subqueries is to create a chain of data. For example, you need to find the name of the customer who has ordered the most books from JustLee Books (not including multiple quantities of the same book) on *one* order. Figure 12-27 shows a query that returns these results.

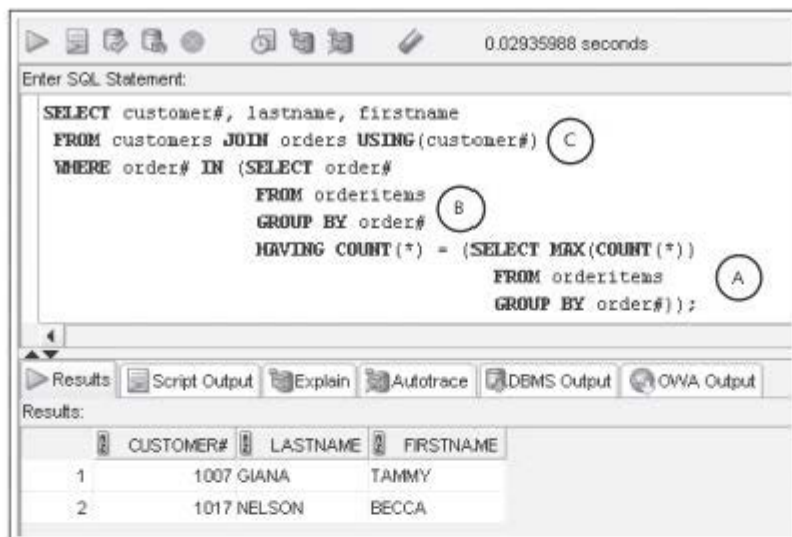


FIGURE 12-27 Nested subqueries

Here are the steps for creating the query in Figure 12-27:

1. The goal of the query is to count the number of items placed on each order and identify the order—or orders, in case of a tie—with the most items. The nested subquery identified by A in Figure 12-27 finds the highest count of books in any order.
2. The value of the highest count of items ordered is then passed to the outer subquery, B.
3. The outer subquery, B, is then used to identify which orders have the same number of items as the highest number of items that the innermost subquery, A, found.
4. After the order numbers have been identified, they are then passed to the outer query, C, which determines the customer number and name of the person who placed the orders. In this case, two customers tied for placing an order with the most items.

The statement uses the IN operator in the outer query's WHERE clause because the subquery, B, might return multiple rows.

TIP

Don't forget to include the extra set of parentheses for the nested group functions in the innermost subquery; if you do, you'll get an error message.

SUBQUERY FACTORING CLAUSE

The WITH clause is an alternative to using subqueries that offers potential improvements in statement readability and processing efficiency. This option is also referred to as a subquery factoring clause and allows a subquery to be defined at the beginning of a SELECT statement which may be referenced multiple times within the query if needed.

An example query to create a list of employees including the number of employees assigned to the same department, the manager's name and the number of employees in the manager's department will demonstrate how to use the WITH clause instead of subqueries. Review the code below and note how the same subquery is repeated in the FROM clause to address the counts for both the employee's and manager's departments.

```
SELECT e.lname Emp_Lastname,
       e.deptno e_dept,
       d1.dcount edept_count,
       m.lname manager_name,
       m.deptno mdept,
       d2.dcount mdept_count
FROM employees e,
     (SELECT deptno, COUNT(*) AS dcount
      FROM employees
      GROUP BY deptno) d1,
     employees m,
     (SELECT deptno, COUNT(*) AS dcount
      FROM employees
      GROUP BY deptno) d2
WHERE e.deptno = d1.deptno
AND e.mgr = m.empno
AND m.deptno = d2.deptno
AND e.mgr = '7839';
```

477

The results displayed in Figure 12-28 list the department and employee count information for the three employees who are assigned to manager 7839. The output includes the employee counts for the same department, as well as the manager's department, which are both retrieved by subqueries in the FROM clause. Again, note that the two subqueries are identical; however, the join conditions are controlling which department rows are being used in the COUNT operation.



	EMP_LASTNAME	E_DEPT	EDEPT_COUNT	MANAGER_NAME	MDEPT	MDEPT_COUNT
1	POTTS	20	2	KING	10	3
2	SMITH	20	2	KING	10	3
3	JONES	10	3	KING	10	3

FIGURE 12-28 Query results for department and employee counts

How is this same task accomplished using the WITH clause? The code below accomplishes the same query results while using the WITH clause to define the employee count by department operation. Notice that “dcount” is used as an alias to reference the COUNT operation results from the query defined in the WITH clause. The main query follows the WITH clause and references the dcount item throughout the statement to use the department count operation.

```
WITH dcount AS
  (SELECT deptno, COUNT(*) AS dcount
   FROM employees
   GROUP BY deptno)
SELECT e.lname Emp_Lastname,
       e.deptno e_dept,
       d1.dcount edept_count,
       m.lname manager_name,
       m.deptno mdept,
       d2.dcount mdept_count
FROM employees e,
     dcount d1,
     employees m,
     dcount d2
WHERE e.deptno = d1.deptno
AND e.mgr = m.empno
AND m.deptno = d2.deptno
AND e.mgr = '7839' ;
```

Since the COUNT operation in this example is being referenced multiple times to complete the query task, the WITH clause simplifies the statement in that the same subquery does not need to be defined multiple times within the statement. In addition, Oracle 12c may reuse the result set of the subquery in the WITH clause leading to improved processing performance. In situations where the same subquery is referenced multiple times in a query statement, the processing efficiency of the WITH clause should be evaluated.

NOTE

Oracle 12c introduced the ability to now include PL/SQL functions and procedures in the WITH clause.

DML ACTIONS USING SUBQUERIES

In Chapter 5, you discovered you can insert data from existing tables into another table by using a subquery in the INSERT statement. You can also perform UPDATE and DELETE statements by using subqueries. For example, employee Sue Stuart needs her bonus set to equal the average bonus of all employees. The SET clause needs a subquery to determine the current average bonus of all employees, as shown in Figure 12-29.

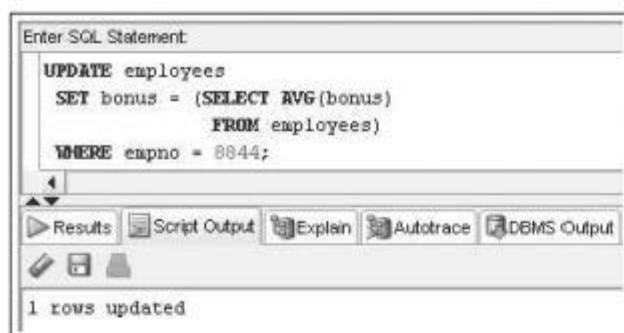


FIGURE 12-29 An UPDATE statement using a subquery

A subquery can also be used in the WHERE clause of a DELETE statement to determine which rows are deleted, based on the value the subquery returns. For example, the DEPARTMENT table contains a list of all departments initially established for JustLee Books; however, some departments might never have been used. JustLee management wants to eliminate any departments that currently have no employees. A subquery identifying all departments with employees can be used to accomplish this task with a DELETE statement, as shown in Figure 12-30. Notice that the WHERE clause uses the NOT IN operator to ensure that departments with employees aren't deleted.

479

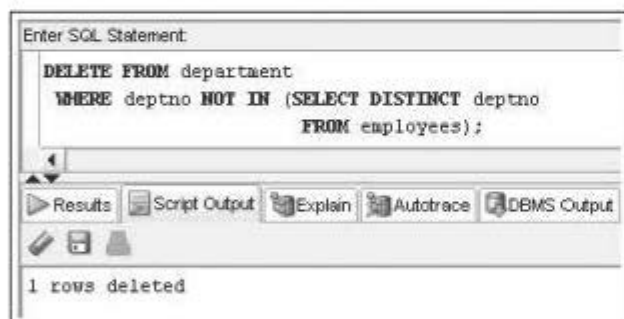


FIGURE 12-30 A DELETE statement using a subquery

MERGE STATEMENTS

With a **MERGE** statement, a series of DML actions can occur with a single SQL statement. The DML statements INSERT, UPDATE, and DELETE were covered in Chapter 5. However, conditionally updating one data source based on another wasn't covered. Now that you have an understanding of more complex SQL statements, this topic can be introduced.

In a data warehousing environment, often you need to conditionally update one table based on another table. For example, a BOOKS table might be used in the JustLee Books production system for recording orders. Any book price changes, category changes, and

new book additions are entered in this table. Another copy of the BOOKS table could be kept for querying and reporting. Many organizations don't want to slow down the production system, so copies of tables are maintained on separate servers to handle querying and reporting requests. In this situation, the tables used for reporting need periodic updating. The MERGE statement assists in this task, as it can compare two data sources or tables and determine which rows need updating and which need inserting.

Take a look at an example involving two BOOKS tables. A table named BOOKS_1 serves as the reporting table, and BOOKS_2 serves as the production table. In this case, the BOOKS_2 table is the input source, and BOOKS_1 is the target. If a book exists in both tables, an UPDATE is needed to capture any changes in retail price or category assignments. If a book is in the BOOKS_2 table but not in the BOOKS_1 table, an INSERT is needed to add the book to BOOKS_1. First, query both tables to review the existing data, as shown in Figure 12-31.

ISBN	TITLE	PUBDATE	RETAIL	CATEGORY
0043172113	DATABASE IMPLEMENTATION	04-JUN-05	55.95	COMPUTER
3437212490	COOKING WITH MUSHROOMS	28-FEB-06	19.95	COOKING
3957136468	HOLY GRAIL OF ORACLE	22-DEC-05	65.95	BUSINESS

3 rows selected

ISBN	TITLE	PUBDATE	RETAIL	CATEGORY
0043172113	DATABASE IMPLEMENTATION	04-JUN-05	55.95	COMPUTER
3437212490	COOKING WITH MUSHROOMS	28-FEB-06	29.95	COOKING
3957136468	HOLY GRAIL OF ORACLE	22-DEC-05	75.95	COMPUTER
1915762492	HAND-CHANGED COMPUTERS	21-JAN-05	25	COMPUTER
0299382519	THE WON WAY TO COOK	11-SEP-00	20.75	COOKING

5 rows selected

Updates

Inserts

FIGURE 12-31 Current contents of the BOOKS_1 and BOOKS_2 tables

The first three rows of the BOOKS_2 table, shown in Figure 12-31, are used to update the existing rows of the BOOKS_1 table because these three match based on ISBN. The last two rows of BOOKS_2 are added (using an INSERT) to the BOOKS_1 table, as these books don't currently exist in this table.

Figure 12-32 shows a MERGE statement that conditionally performs the UPDATES and INSERTS.

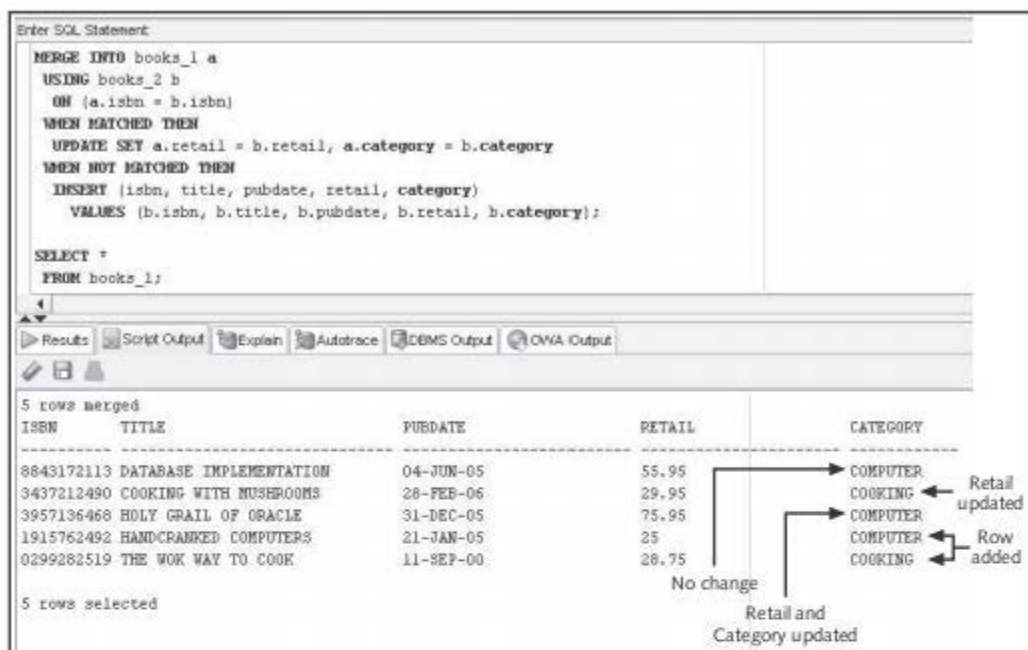


FIGURE 12-32 MERGE statement with UPDATE and INSERT

The following explains each part of this MERGE statement:

- **MERGE INTO books_1 a:** The BOOKS_1 table is to be changed, and a table alias of "a" is assigned to this table.
- **USING books_2 b:** The BOOKS_2 table provides the data to update or insert into BOOKS_1, and a table alias of "b" is assigned to this table.
- **ON (a.isbn = b.isbn):** The rows of the two tables are joined or matched based on ISBN.
- **WHEN MATCHED THEN:** If a row match based on ISBN is discovered, execute the UPDATE action in this clause. The UPDATE action instructs Oracle 12c to modify only two columns (Retail and Category).
- **WHEN NOT MATCHED THEN:** If no match is found based on the ISBN (a book exists in BOOKS_2 that isn't in BOOKS_1), perform the INSERT action in this clause.

NOTE

A MERGE statement containing an UPDATE and an INSERT clause is also called an UPSERT statement.

Including both WHEN MATCHED and WHEN NOT MATCHED isn't required. If only a particular DML operation is needed, you include only the corresponding clause.

Next, execute a ROLLBACK statement so that the BOOKS_1 data is set to the original three rows before performing the next example.

You can also include a WHERE condition in the matching clauses of a MERGE statement to conditionally perform the DML action based on a data value. Return to the previous example, but add a condition to update or insert only rows with the Computer category assigned in the BOOKS_2 table. Figure 12-33 shows WHERE clauses added to the previous MERGE statement.

The screenshot shows an SQL IDE window titled "Enter SQL Statement". The SQL code entered is:

```

MERGE INTO books_1 a
  USING books_2 b
    ON (a.isbn = b.isbn)
  WHEN MATCHED THEN
    UPDATE SET a.retail = b.retail, a.category = b.category
    WHERE b.category = 'COMPUTER'
  WHEN NOT MATCHED THEN
    INSERT (isbn, title, pubdate, retail, category)
    VALUES (b.isbn, b.title, b.pubdate, b.retail, b.category)
    WHERE b.category = 'COMPUTER';

SELECT *
FROM books_1;

```

Below the code, there are tabs for "Results", "Script Output", "Explain", "Autotrace", "DBMS Output", and "OWA Output". The "Results" tab is active, showing the output of the SQL statement:

3 rows merged

ISBN	TITLE	PUBDATE	RETAIL	CATEGORY
8843172113	DATABASE IMPLEMENTATION	04-JUN-05	55.95	COMPUTER
3437212490	COOKING WITH MUSHROOMS	28-FEB-06	19.95	COOKING
3957136468	HOLY GRAIL OF ORACLE	31-DEC-05	75.95	COMPUTER
1915762492	HANDCRANKED COMPUTERS	21-JAN-05	25	COMPUTER

4 rows selected

FIGURE 12-33 Using WHERE conditions in a MERGE statement

Recall that the BOOKS_2 table contains two rows with books in the Cooking category. These two rows are no longer processed because of the added WHERE conditions. Therefore, the retail price of the book *Cooking with Mushrooms* isn't updated, and the book *The Wok Way to Cook* isn't inserted. Also, the book *Holy Grail of Oracle* is originally assigned the category Business in the BOOKS_1 table and Computer in the BOOKS_2 table. The WHERE clause condition checks the category data in the BOOKS_2 table, which is Computer, so the MERGE statement updates this row in the BOOKS_1 table.

Execute another ROLLBACK statement so that the BOOKS_1 data is set to the original three rows before performing the next example.

When a match is found during a MERGE statement, a DELETE statement can also be conditionally processed. For example, assume the reporting table requires data only for

books with a retail price of at least \$50. Figure 12-34 shows a conditional DELETE action added to the WHEN MATCHED clause.

The screenshot shows a SQL IDE window titled "Enter SQL Statement:". The SQL code entered is:

```
MERGE INTO books_1 a
  USING books_2 b
    ON (a.isbn = b.isbn)
  WHEN MATCHED THEN
    UPDATE SET a.retail = b.retail, a.category = b.category
    DELETE WHERE (b.retail < 50);

SELECT *
FROM books_1;
```

Below the code editor, there is a toolbar with buttons for "Results", "Script Output", "Explain", "Autotrace", "DBMS Output", and "OWA Output". The "Results" button is selected, and the results are displayed in a table format:

3 rows merged				
ISBN	TITLE	PUBDATE	RETAIL	CATEGORY
8843172113	DATABASE IMPLEMENTATION	04-JUN-05	55.95	COMPUTER
3957136468	HOLY GRAIL OF ORACLE	31-DEC-05	75.95	COMPUTER

Below the table, it says "2 rows selected".

FIGURE 12-34 Conditional DELETE in a MERGE statement

The BOOKS_1 table contains only two rows instead of three because the DELETE action removed *Cooking with Mushrooms*. This row is deleted because the retail amount is \$29.95 in the BOOKS_2 table, which meets the DELETE condition of costing less than \$50. This MERGE statement processes only matched rows, so the other two rows in the BOOKS_2 table with retail amounts below \$50 aren't processed.

Chapter Summary

- A subquery is a complete query nested in the SELECT, FROM, HAVING, or WHERE clause of another query. The subquery must be enclosed in parentheses and have a SELECT and a FROM clause, at a minimum.
- Subqueries are completed first. The result of the subquery is used as input for the outer query.
- A single-row subquery can return a maximum of one value.
- Single-row operators include =, >, <, >=, <=, and <>.
- Multiple-row subqueries return more than one row of results.
- Operators that can be used with multiple-row subqueries include IN, ALL, ANY, and EXISTS.
- Multiple-column subqueries return more than one column to the outer query. The columns of data are passed to the outer query in the same order in which they're listed in the subquery's SELECT clause.
- NULL values returned by a multiple-row or multiple-column subquery aren't a problem if the IN or =ANY operator is used. The NVL function can be used to substitute a value for a NULL value when working with subqueries.
- Correlated subqueries reference a column contained in the outer query. When using correlated subqueries, the subquery is executed once for each row the outer query processes.
- The EXISTS operator is used to formulate a correlated subquery.
- Subqueries can be nested to a maximum depth of 255 subqueries in the outer query's WHERE clause. The depth is unlimited for subqueries nested in the outer query's FROM clause.
- With nested subqueries, the innermost subquery is executed first, then the next highest level subquery is executed, and so on, until the outermost query is reached.
- The WITH clause or subquery factoring clause may be used to define a subquery before the main query statement and then it allows this subquery to be referenced multiple times within the statement.
- DML actions can include subqueries to determine which rows are processed.
- A MERGE statement allows performing multiple DML actions conditionally while comparing data of two tables.

Chapter 12 Syntax Summary

The following table summarizes the syntax you have learned in this chapter. You can use the table as a study guide and reference.

Subquery Processing	Example
Correlated subquery: References a column in the outer query. Executes the subquery once for every row in the outer query.	<pre>SELECT title FROM books b WHERE b.isbn IN (SELECT isbn FROM orderitems o WHERE b.isbn = o.isbn);</pre>
Uncorrelated subquery: Executes the subquery first and passes the value to the outer query.	<pre>SELECT title FROM books b, orderitems o WHERE books isbn IN (SELECT isbn FROM orderitems) AND b.isbn = o.isbn;</pre>
Multiple-Row Comparison Operators	
Operator	Description
>ALL	More than the highest value returned by the subquery
<ALL	Less than the lowest value returned by the subquery
<ANY	Less than the highest value returned by the subquery
>ANY	More than the lowest value returned by the subquery
=ANY	Equal to any value returned by the subquery (same as IN)
[NOT] EXISTS	Row must match a value in the subquery
DML Action with a MERGE Statement	
	Example
Conditionally performs a series of DML actions	<pre>MERGE INTO books_1 a USING books_2 b ON (a.isbn = b.isbn) WHEN MATCHED THEN UPDATE SET a.retail = b.retail, a.category = b.category WHEN NOT MATCHED THEN INSERT (isbn, title, pubdate, retail, category) VALUES (b.isbn, b.title, b.pubdate, b.retail, b.category);</pre>

Review Questions

1. What's the difference between a single-row subquery and a multiple-row subquery?
2. What comparison operators are required for multiple-row subqueries?
3. What happens if a single-row subquery returns more than one row of results?
4. Which SQL clause(s) can't be used in a subquery in the WHERE or HAVING clauses?
5. If a subquery is used in the FROM clause of a query, how are the subquery's results referenced in other clauses of the query?
6. Why might a MERGE statement be used?
7. How can Oracle 12c determine whether clauses of a SELECT statement belong to an outer query or a subquery?
8. When should a subquery be nested in a HAVING clause?
9. What's the difference between correlated and uncorrelated subqueries?
10. What type of situation requires using a subquery?

486

Multiple Choice

To answer these questions, refer to the tables in the JustLee Books database.

1. Which query identifies customers living in the same state as the customer named Leila Smith?
 - a.

```
SELECT customer# FROM customers
WHERE state = (SELECT state FROM customers
WHERE lastname = 'SMITH');
```
 - b.

```
SELECT customer# FROM customers
WHERE state = (SELECT state FROM customers
WHERE lastname = 'SMITH'
OR firstname = 'LEILA');
```
 - c.

```
SELECT customer# FROM customers
WHERE state = (SELECT state FROM customers
WHERE lastname = 'SMITH'
AND firstname = 'LEILA'
ORDER BY customer);
```
 - d.

```
SELECT customer# FROM customers
WHERE state = (SELECT state FROM customers
WHERE lastname = 'SMITH'
AND firstname = 'LEILA');
```