6

Retrieving Data by Using Subqueries

ORACLE

Objectives

After completing this lesson, you should be able to:

- Write a multiple-column subquery
- Use scalar subqueries in SQL
- Solve problems with correlated subqueries
- Use the `EXISTS` and `NOT EXISTS` operators
- Use the `WITH` clause

In this lesson, you learn how to write multiple-column subqueries and subqueries in the `FROM` clause of a `SELECT` statement. You also learn how to solve problems by using scalar, correlated subqueries and the `WITH` clause.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Retrieving Data by Using a Subquery as a Source

```
SELECT department_name, city
FROM   departments
NATURAL JOIN (SELECT l.location_id, l.city, l.country_id
              FROM   locations l
              JOIN   countries c
              ON(l.country_id = c.country_id)
              JOIN regions
              USING(region_id)
              WHERE region_name = 'Europe');
```

	DEPARTMENT_NAME	CITY
1	Human Resources	London
2	Sales	Oxford
3	Public Relations	Munich

You can use a subquery in the FROM clause of a SELECT statement, which is very similar to how views are used. A subquery in the FROM clause of a SELECT statement is also called an *inline* view. A subquery in the FROM clause of a SELECT statement defines a data source for that particular SELECT statement, and only that SELECT statement. As with a database view, the SELECT statement in the subquery can be as simple or as complex as you like.

When a database view is created, the associated SELECT statement is stored in the data dictionary. In situations where you do not have the necessary privileges to create database views, or when you would like to test the suitability of a SELECT statement to become a view, you can use an inline view.

With inline views, you can have all the code needed to support the query in one place. This means that you can avoid the complexity of creating a separate database view. The example in the slide shows how to use an inline view to display the department name and the city in Europe. The subquery in the FROM clause fetches the location ID, city name, and the country by joining three different tables. The output of the inner query is considered as a table for the outer query. The inner query is similar to that of a database view but does not have any physical name.

You can display the same output as in the example in the slide by performing the following two steps:

1. Create a database view:

```
CREATE OR REPLACE VIEW european_cities
AS
SELECT l.location_id, l.city, l.country_id
FROM   locations l
JOIN   countries c
ON(l.country_id = c.country_id)
JOIN regions USING(region_id)
WHERE region_name = 'Europe';
```

2. Join the `EUROPEAN_CITIES` view with the `DEPARTMENTS` table:

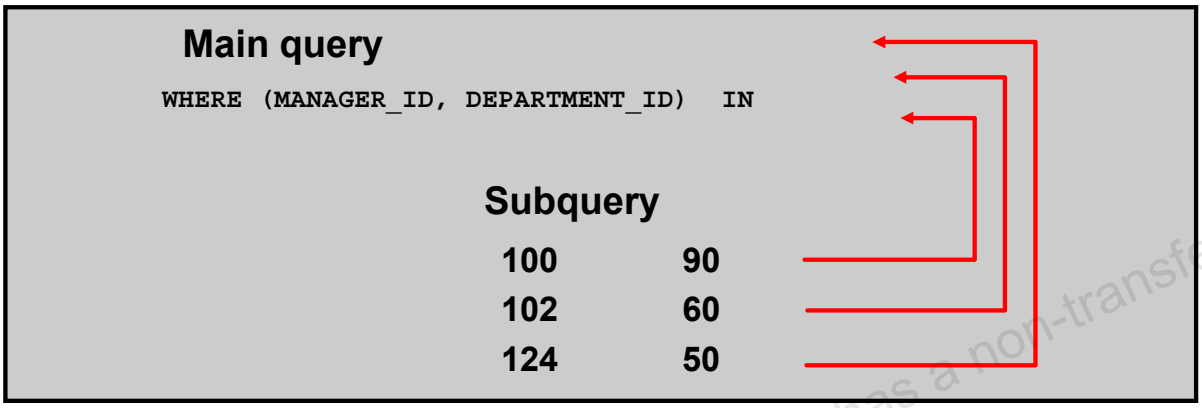
```
SELECT department_name, city
FROM   departments
NATURAL JOIN european_cities;
```

Note: You learned how to create database views in the lesson titled *Creating Views*.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Multiple-Column Subqueries



Each row of the main query is compared to values from a multiple-row and multiple-column subquery.

So far, you have written single-row subqueries and multiple-row subqueries where only one column is returned by the inner `SELECT` statement and this is used to evaluate the expression in the parent `SELECT` statement. If you want to compare two or more columns, you must write a compound `WHERE` clause using logical operators. Using multiple-column subqueries, you can combine duplicate `WHERE` conditions into a single `WHERE` clause.

Syntax

```
SELECT      column, column, ...
FROM  table
WHERE (column, column, ...) IN
              (SELECT column, column, ...
                           FROM   table
                           WHERE  condition);
```

The graphic in the slide illustrates that the values of `MANAGER_ID` and `DEPARTMENT_ID` from the main query are being compared with the `MANAGER_ID` and `DEPARTMENT_ID` values retrieved by the subquery. Because the number of columns that are being compared is more than one, the example qualifies as a multiple-column subquery.

Note: Before you run the examples in the next few slides, you need to create the `empl_demo` table and populate data into it by using the `lab_06_insert_empdata.sql` file.

Column Comparisons

Multiple-column comparisons involving subqueries can be:

- Pairwise comparisons
- Nonpairwise comparisons

Pairwise Versus Nonpairwise Comparisons

Multiple-column comparisons involving subqueries can be nonpairwise comparisons or pairwise comparisons. If you consider the example “Display the details of the employees who work in the same department, and have the same manager, as ‘Daniel’?,” you get the correct result with the following statement:

```
SELECT first_name, last_name, manager_id, department_id
FROM empl_demo
WHERE manager_id IN (SELECT manager_id
                     FROM empl_demo
                     WHERE first_name = 'Daniel')
AND department_id IN (SELECT department_id
                      FROM empl_demo
                      WHERE first_name = 'Daniel');
```

There is only one “Daniel” in the `EMPL_DEMO` table (Daniel Faviet, who is managed by employee 108 and works in department 100). However, if the subqueries return more than one row, the result might not be correct. For example, if you run the same query but substitute “John” for “Daniel,” you get an incorrect result. This is because the combination of `department_id` and `manager_id` is important. To get the correct result for this query, you need a pairwise comparison.

Pairwise Comparison Subquery

Display the details of the employees who are managed by the same manager and work in the same department as the employees with EMPLOYEE_ID 199 or 174.

```
SELECT employee_id, manager_id, department_id
FROM employees
WHERE (manager_id, department_id) IN
                              (SELECT manager_id, department_id
                               FROM employees
                               WHERE employee_id IN (174, 199))
AND employee_id NOT IN (174,199);
```

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	141	124	50
2	142	124	50
3	143	124	50
4	144	124	50

…

The example shows a pairwise comparison of the columns. It compares the values in the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table with the values in the MANAGER_ID column and the DEPARTMENT_ID column for the employees with the EMPLOYEE_ID 199 or 174.

First, the subquery to retrieve the MANAGER_ID and DEPARTMENT_ID values for the employees with the EMPLOYEE_ID 199 or 174 is executed. These values are compared with the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table. If the values match, the row is displayed. In the output, the records of the employees with the EMPLOYEE_ID 199 or 174 will not be displayed. The output of the query in the slide follows.

Nonpairwise Comparison Subquery

Display the details of the employees who are managed by the same manager as the employees with EMPLOYEE_ID 174 or 141 and work in the same department as the employees with EMPLOYEE_ID 174 or 141.

```
SELECT employee_id, manager_id, department_id
FROM employees
WHERE manager_id IN
                    (SELECT manager_id
                     FROM employees
                     WHERE employee_id IN (174,141))
AND department_id IN
                    (SELECT department_id
                     FROM employees
                     WHERE employee_id IN (174,141))
AND employee_id NOT IN(174,141);
```

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	142	124	50
2	143	124	50

...

ORACLE

The example shows a nonpairwise comparison of the columns. It displays the EMPLOYEE_ID, MANAGER_ID, and DEPARTMENT_ID of any employee whose manager ID matches any of the manager IDs of employees whose employee IDs are either 174 or 141 and DEPARTMENT_ID match any of the department IDs of employees whose employee IDs are either 174 or 141.

First, the subquery to retrieve the MANAGER_ID values for the employees with the EMPLOYEE_ID 174 or 141 is executed. Similarly, the second subquery to retrieve the DEPARTMENT_ID values for the employees with the EMPLOYEE_ID 174 or 141 is executed. The retrieved values of the MANAGER_ID and DEPARTMENT_ID columns are compared with the MANAGER_ID and DEPARTMENT_ID column for each row in the EMPLOYEES table. If the MANAGER_ID column of the row in the EMPLOYEES table matches with any of the values of the MANAGER_ID retrieved by the inner subquery and if the DEPARTMENT_ID column of the row in the EMPLOYEES table matches with any of the values of the DEPARTMENT_ID retrieved by the second subquery, the record is displayed.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Scalar Subquery Expressions

- A scalar subquery expression is a subquery that returns exactly one column value from one row.
- Scalar subqueries can be used in:
  - The condition and expression part of DECODE and CASE
  - All clauses of SELECT except GROUP BY
  - The SET clause and WHERE clause of an UPDATE statement

A subquery that returns exactly one column value from one row is also referred to as a scalar subquery. Multiple-column subqueries that are written to compare two or more columns, using a compound WHERE clause and logical operators, do not qualify as scalar subqueries.

The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, the value of the scalar subquery expression is NULL. If the subquery returns more than one row, the Oracle Server returns an error. The Oracle Server has always supported the usage of a scalar subquery in a SELECT statement. You can use scalar subqueries in:

- The condition and expression part of DECODE and CASE
- All clauses of SELECT except GROUP BY
- The SET clause and WHERE clause of an UPDATE statement

However, scalar subqueries are not valid expressions in the following places:

- In the RETURNING clause of data manipulation language (DML) statements
- As the basis of a function-based index
- In GROUP BY clauses, CHECK constraints
- In CONNECT BY clauses
- In statements that are unrelated to queries, such as CREATE PROFILE

Scalar Subqueries: Examples

- Scalar subqueries in `CASE` expressions:

```
SELECT employee_id, last_name,
       (CASE
        WHEN department_id =
                (SELECT department_id
                 FROM departments
                 WHERE location_id = 1800)
         THEN 'Canada' ELSE 'USA' END) location
FROM   employees;
```

20

- Scalar subqueries in the `SELECT` statement:

```
select department_id, department_name,
      (select count(*)
       from   employees e
       where  e.department_id = d.department_id) as emp_count
from   departments d;
```


The first example in the slide demonstrates that scalar subqueries can be used in `CASE` expressions. The inner query returns the value 20, which is the department `ID` of the department whose location `ID` is 1800. The `CASE` expression in the outer query uses the result of the inner query to display the employee `ID`, last names, and a value of Canada or USA, depending on whether the department `ID` of the record retrieved by the outer query is 20.

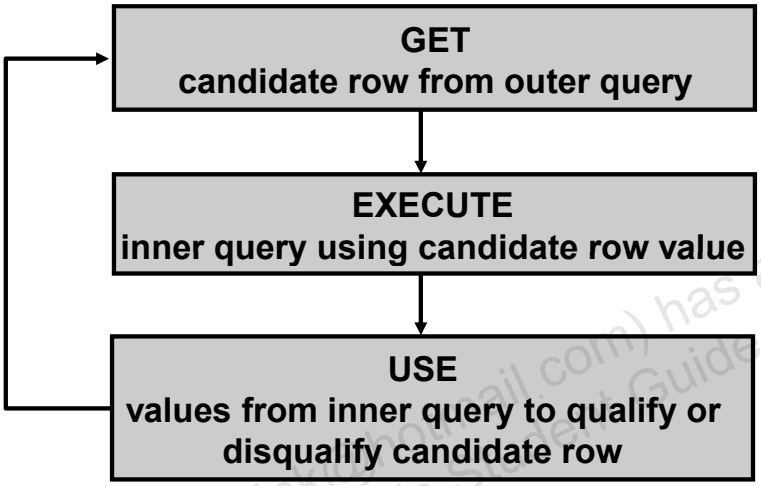
The second example in the slide demonstrates that scalar subqueries can be used in `SELECT` statements 13

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Correlated Subqueries

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.



The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a `SELECT`, `UPDATE`, or `DELETE` statement.

Nested Subqueries Versus Correlated Subqueries

With a normal nested subquery, the inner `SELECT` query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. That is, the inner query is driven by the outer query.

Nested Subquery Execution

- The inner query executes first and finds a value.
- The outer query executes once, using the value from the inner query.

Correlated Subquery Execution

- Get a candidate row (fetched by the outer query).
- Execute the inner query by using the value of the candidate row.
- Use the values resulting from the inner query to qualify or disqualify the candidate.
- Repeat until no candidate row remains.

Correlated Subqueries

The subquery references a column from a table in the parent query.

```
SELECT column1, column2, ...
FROM   table1   Outer_table
WHERE  column1 operator
                      (SELECT  column1, column2
                       FROM    table2
                       WHERE   expr1 =
                             Outer_table.expr2);
```


A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. That is, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

The Oracle Server performs a correlated subquery when the subquery references a column from a table in the parent query.

Note: You can use the `ANY` and `ALL` operators in a correlated subquery.

Using Correlated Subqueries: Example 1

Find all employees who earn more than the average salary in their department.

```
SELECT last_name, salary, department_id
FROM   employees outer_table
WHERE  salary >
              (SELECT AVG(salary)
               FROM   employees inner_table
               WHERE inner_table.department_id =
               outer_table.department_id);
```

	LAST_NAME	SALARY	DEPARTMENT_ID
1	King	24000	90
2	Hunold	9000	60
3	Ernst	6000	60
4	Greenberg	12008	100
5	Faviet	9000	100
6	Raphaely	11000	30

....

Each time a row from the outer query is processed, the inner query is evaluated.

ORACLE

The example in the slide finds which employees earn more than the average salary of their department. In this case, the correlated subquery specifically computes the average salary for each department.

Because both the outer query and inner query use the `EMPLOYEES` table in the `FROM` clause, an alias is given to `EMPLOYEES` in the outer `SELECT` statement for clarity. The alias makes the entire `SELECT` statement more readable. Without the alias, the query would not work properly because the inner statement would not be able to distinguish the inner table column from the outer table column.

The correlated subquery performs the following steps for each row of the `EMPLOYEES` table:

1. The `department_id` of the row is determined.
2. The `department_id` is then used to evaluate the parent query.
3. If the salary in that row is greater than the average salary of the departments of that row, then the row is returned.

The subquery is evaluated once for each row of the `EMPLOYEES` table.

Using Correlated Subqueries: Example 2

Display details of highest earning employee in each department.

```
SELECT department_id, employee_id, salary
FROM EMPLOYEES e
WHERE 1 =
      (SELECT  COUNT(DISTINCT salary)
       FROM EMPLOYEES
       WHERE e.department_id = department_id
       AND e.salary <= salary)
```

	DEPARTMENT_ID	EMPLOYEE_ID	SALARY
1	90	100	24000
2	60	103	9000
3	100	108	12008
4	30	114	11000

....

The example in the slide displays the details of highest earning employees in each department. The Oracle Server evaluates a correlated subquery as follows:

1. Select a row from the table specified in the outer query. This will be the current candidate row.
2. Store the value of the column referenced in the subquery from this candidate row. (In the example in the slide, the column referenced in the subquery is `e.salary`.)
3. Perform the subquery with its condition referencing the value from the outer query's candidate row. (In the example in the slide, the `COUNT(DISTINCT salary)` group function is evaluated based on the value of the `E.SALARY` column obtained in step 2.)
4. Evaluate the `WHERE` clause of the outer query on the basis of results of the subquery performed in step 3. This determines whether the candidate row is selected for output. (In the example, the number of times an employee has changed jobs, evaluated by the subquery, is compared with 2 in the `WHERE` clause of the outer query. If the condition is satisfied, that employee record is displayed.)
5. Repeat the procedure for the next candidate row of the table, and so on, until all the rows in the table have been processed.

The correlation is established by using an element from the outer query in the subquery. In this example, you compare `EMPLOYEE_ID` from the table in the subquery with `EMPLOYEE_ID` from the table in the outer query.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Using the EXISTS Operator

- The EXISTS operator tests for existence of rows in the results set of the subquery.
- If a subquery row value is found:
  - The search does not continue in the inner query
  - The condition is flagged TRUE
- If a subquery row value is not found:
  - The condition is flagged FALSE
  - The search continues in the inner query

With nesting SELECT statements, all logical operators are valid. In addition, you can use the EXISTS operator. This operator is frequently used with correlated subqueries to test whether a value retrieved by the outer query exists in the results set of the values retrieved by the inner query. If the subquery returns at least one row, the operator returns TRUE. If the value does not exist, it returns FALSE. Accordingly, NOT EXISTS tests whether a value retrieved by the outer query is not a part of the results set of the values retrieved by the inner query.

Using the EXISTS Operator

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees outer
WHERE  EXISTS ( SELECT NULL
                FROM   employees
                WHERE  manager_id =
                       outer.employee_id);
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	100	King	AD_PRES	90
2	101	Kochhar	AD_VP	90
3	102	De Haan	AD_VP	90
4	103	Hunold	IT_PROG	60
5	108	Greenberg	FI_MGR	100
6	114	Raphaely	PU_MAN	30
7	120	Weiss	ST_MAN	50
8	121	Fripp	ST_MAN	50

...

The EXISTS operator ensures that the search in the inner query does not continue when at least one match is found for the manager and employee number by the condition:

```
WHERE manager_id = outer.employee_id
```

Note that the inner SELECT query does not need to return a specific value, so a constant can be selected.

Find All Departments That Do Not Have Any Employees

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT NULL
                  FROM   employees
                  WHERE  department_id = d.department_id);
```

	DEPARTMENT_ID	DEPARTMENT_NAME
1	120	Treasury
2	130	Corporate Tax
3	140	Control And Credit
4	150	Shareholder Services
5	160	Benefits
6	170	Manufacturing
7	180	Construction
8	190	Contracting
9	200	Operations
10	210	IT Support

...

All Rows Fetched: 16

Using the NOT EXISTS Operator

Alternative Solution

A NOT IN construct can be used as an alternative for a NOT EXISTS operator, as shown in the following example:

```
SELECT department_id, department_name
FROM   departments
WHERE  department_id NOT IN (SELECT department_id
                             FROM   employees);
```

However, NOT IN evaluates to FALSE if any member of the set is a NULL value. Therefore, your query will not return any rows even if there are rows in the departments table that satisfy the WHERE condition.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

WITH Clause

- Using the WITH clause, you can use the same query block in a SELECT statement when it occurs more than once within a complex query.
- The WITH clause retrieves the results of a query block and stores it in the user's temporary tablespace.
- The WITH clause may improve performance.

Using the WITH clause, you can define a query block before using it in a query. The WITH clause (formally known as subquery_factoring_clause) enables you to reuse the same query block in a SELECT statement when it occurs more than once within a complex query. This is particularly useful when a query has many references to the same query block and there are joins and aggregations.

Using the WITH clause, you can reuse the same query when it is costly to evaluate the query block and it occurs more than once within a complex query. Using the WITH clause, the Oracle Server retrieves the results of a query block and stores it in the user's temporary tablespace. This can improve performance.

WITH Clause Benefits

- Makes the query easy to read
- Evaluates a clause only once, even if it appears multiple times in the query
- In most cases, may improve performance for large queries

WITH Clause: Example

```
WITH CNT_DEPT AS
(
SELECT department_id,
 COUNT(1) NUM_EMP
FROM EMPLOYEES
GROUP BY department_id
)
SELECT employee_id,
 SALARY/NUM_EMP
FROM EMPLOYEES E
JOIN CNT_DEPT C
ON (e.department_id = c.department_id);
```

	EMPLOYEE_ID	SALARY/NUM_EMP
1	100	8000
2	103	1800
3	108	2001.333333333333333333333333333333333333
4	110	1366.666666666666666666666666666666666667
5	114	1833.333333333333333333333333333333333333
6	137	80
7	139	60

...

The SQL code in the slide is an example of a situation in which you can improve performance and write SQL more simply by using the WITH clause. The query creates the query name as CNT_DEPT then uses them in the body of the main query. Here, you perform a math operation by dividing the salary of employee with the total number of employees in each department. Internally, the WITH clause is resolved either as an inline view or a temporary table. The optimizer chooses the appropriate resolution depending on the cost or benefit of temporarily storing the results of the WITH clause.

WITH Clause Usage Notes

- It is used only with SELECT statements.
- A query name is visible to all WITH element query blocks (including their subquery blocks) defined after it and the main query block itself (including its subquery blocks).
- When the query name is the same as an existing table name, the parser searches from the inside out, and the query block name takes precedence over the table name.
- The WITH clause can hold more than one query. Each query is then separated by a comma.

Recursive WITH Clause

The Recursive WITH clause:

- Enables formulation of recursive queries
- Creates a query with a name, called the Recursive WITH element name
- Contains two types of query block members: an anchor and a recursive
- Is ANSI-compatible

The WITH clause has been extended to enable formulation of recursive queries.

Recursive WITH defines a recursive query with a name, the Recursive WITH element name. The Recursive WITH element definition must contain at least two query blocks: an anchor member and a recursive member. There can be multiple anchor members, but there can be only a single recursive member. The anchor member must appear before the recursive member, and it cannot reference *query_name*. The anchor member can be composed of one or more query blocks combined by the set operators for example , UNION ALL, UNION, INTERSECT or MINUS. The recursive member must follow the anchor member and must reference *query_name* exactly once. You must combine the recursive member with the anchor member using the UNION ALL set operator.

Recursive WITH clause complies with the American National Standards Institute (ANSI) standard.

Recursive WITH can be used to query hierarchical data such as organization charts.

Recursive WITH Clause: Example

FLIGHTS Table

	SOURCE	DESTIN	FLIGHT_TIME
1	San Jose	Los Angeles	1.3
2	New York	Boston	1.1
3	Los Angeles	New York	5.8

1

```
WITH Reachable_From (Source, Destin, TotalFlightTime) AS
(
    SELECT Source, Destin, Flight_time
    FROM Flights
  UNION ALL
    SELECT incoming.Source, outgoing.Destin,
           incoming.TotalFlightTime+outgoing.Flight_time
    FROM Reachable_From incoming, Flights outgoing
    WHERE incoming.Destin = outgoing.Source
)
SELECT Source, Destin, TotalFlightTime
FROM Reachable_From;
```

2

3

	SOURCE	DESTIN	TOTALFLIGHTTIME
1	San Jose	Los Angeles	1.3
2	New York	Boston	1.1
3	Los Angeles	New York	5.8
4	Los Angeles	Boston	6.9
5	San Jose	New York	7.1
6	San Jose	Boston	8.2

ORACLE

Example 1 in the slide displays records from a FLIGHTS table describing flights between two cities.

Using the query in example 2, you query the FLIGHTS table to display the total flight time between any source and destination. The WITH clause in the query, which is named Reachable From, has a UNION ALL query with two branches. The first branch is the *anchor* branch, which selects all the rows from the Flights table. The second branch is the recursive branch. It joins the contents of Reachable From to the Flights table to find other cities that can be reached, and adds these to the content of Reachable From. The operation will finish when no more rows are found by the recursive branch.

Example 3 displays the result of the query that selects everything from the WITH clause element Reachable From.

For details, see:

- *Oracle Database SQL Language Reference 12c Release 1.0*
- *Oracle Database Data Warehousing Guide 12c Release 1.0*

Quiz

With a correlated subquery, the inner `SELECT` statement drives the outer `SELECT` statement.

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- Write a multiple-column subquery
- Use scalar subqueries in SQL
- Solve problems with correlated subqueries
- Use the `EXISTS` and `NOT EXISTS` operators
- Use the `WITH` clause

You can use multiple-column subqueries to combine multiple `WHERE` conditions in a single `WHERE` clause. Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons.

You can use a subquery to define a table to be operated on by a containing query.

Scalar subqueries can be used in:

- The condition and expression part of `DECODE` and `CASE`
- All clauses of `SELECT` except `GROUP BY`
- A `SET` clause and `WHERE` clause of the `UPDATE` statement

The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a `SELECT` statement. Using the `WITH` clause, you can reuse the same query when it is costly to re-evaluate the query block and it occurs more than once within a complex query.

Practice 6: Overview

This practice covers the following topics:

- Creating multiple-column subqueries
- Writing correlated subqueries
- Using the `EXISTS` operator
- Using scalar subqueries
- Using the `WITH` clause

ORACLE

In this practice, you write multiple-column subqueries, and correlated and scalar subqueries. You also solve problems by writing the `WITH` clause.