6

Retrieving Data Using Subqueries

ORACLE

Objectives

After completing this lesson, you should be able to do the following:

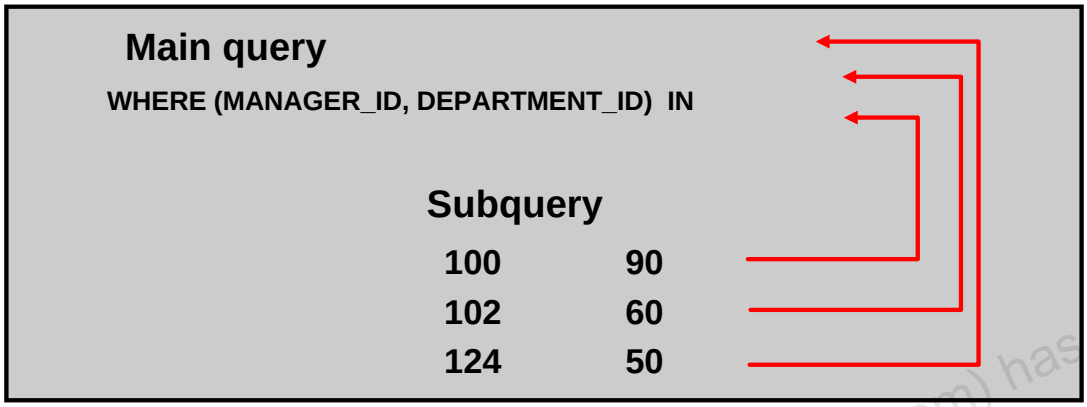
- Write a multiple-column subquery
- Use scalar subqueries in SQL
- Solve problems with correlated subqueries
- Update and delete rows using correlated subqueries
- Use the `EXISTS` and `NOT EXISTS` operators
- Use the `WITH` clause

ORACLE

Objectives

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.

Multiple-Column Subqueries



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

Multiple-Column Subqueries

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.

Column Comparisons

Multiple-column comparisons involving subqueries can be:

- Nonpairwise comparisons
- Pairwise comparisons

ORACLE

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 employees
WHERE manager_id IN (SELECT manager_id
                     FROM employees
                     WHERE first_name = ‘Daniel’)
AND department_id IN (SELECT department_id
                      FROM employees
                      WHERE first_name = ‘Daniel’);
```

There is only one “Daniel” in the `EMPLOYEES` 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 `manger_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 employees with the first name of “John.”

```
SELECT  employee_id, manager_id, department_id
FROM    employees
WHERE  (manager_id, department_id) IN
                      (SELECT manager_id, department_id
                       FROM employees
                       WHERE first_name = 'John')
AND first_name <> 'John';
```

ORACLE

Pairwise Comparison Subquery

The example in the slide compares the combination of 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 FIRST_NAME of “John.” First, the subquery to retrieve the MANAGER_ID and DEPARTMENT_ID values for the employees with the FIRST_NAME of “John” is executed. This subquery returns the following:

	MANAGER_ID	DEPARTMENT_ID
1	108	100
2	123	50
3	100	80

These values are compared with the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table. If the combination matches, the row is displayed. In the output, the records of the employees with the FIRST_NAME of “John” will not be displayed. The following is the output of the query in the slide:

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	137	123	50
2	138	123	50
3	140	123	50

...

Nonpairwise Comparison Subquery

Display the details of the employees who are managed by the same manager as the employees with the first name of “John” and work in the same department as the employees with the first name of “John.”

```
SELECT  employee_id, manager_id, department_id
FROM    employees
WHERE   manager_id IN
                (SELECT manager_id
                 FROM employees
                 WHERE first_name = 'John')
AND department_id IN
                (SELECT department_id
                 FROM employees
                 WHERE first_name = 'John')
AND first_name <> 'John';
```

ORACLE

Nonpairwise Comparison Subquery

The example shows a nonpairwise comparison of the columns. First, the subquery to retrieve the MANAGER_ID values for the employees with the FIRST_NAME of “John” is executed. Similarly, the second subquery to retrieve the DEPARTMENT_ID values for the employees with the FIRST_NAME of “John” is executed. The retrieved values of the MANAGER_ID and DEPARTMENT_ID columns are compared with the MANAGER_ID and DEPARTMENT_ID columns 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 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 DEPARTMENT_ID retrieved by the second subquery, the record is displayed. The following is the output of the query in the slide:

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	120	100	50
2	121	100	50
3	122	100	50
4	123	100	50

...

This query retrieves five rows more than the pairwise comparison (those with the combination of manager_id=100 and department_id=50, although no employee named “John” has such a combination).

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:
  - Condition and expression part of DECODE and CASE
  - All clauses of SELECT except GROUP BY

ORACLE

Scalar Subqueries in SQL

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:

- As default values for columns and hash expressions for clusters
- In the RETURNING clause of data manipulation language (DML) statements
- As the basis of a function-based index
- In GROUP BY clauses, CHECK constraints, WHEN conditions
- 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 ORDER BY clause:

```
SELECT   employee_id, last_name
FROM     employees e
ORDER BY (SELECT department_name
          FROM departments d
          WHERE e.department_id = d.department_id);
```

ORACLE

Scalar Subqueries: Examples

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 or not.

The following is the result of the first example in the slide:

	EMPLOYEE_ID	LAST_NAME	LOCATION
1	100	King	USA
2	101	Kochhar	USA
3	102	De Haan	USA
4	103	Hunold	USA
5	104	Ernst	USA
6	105	Austin	USA
7	106	Pataballa	USA

...

107	206	Gietz	USA

Scalar Subqueries: Examples (continued)

The second example in the slide demonstrates that scalar subqueries can be used in the `ORDER BY` clause. The example orders the output based on the `DEPARTMENT_NAME` by matching the `DEPARTMENT_ID` from the `EMPLOYEES` table with the `DEPARTMENT_ID` from the `DEPARTMENTS` table. This comparison is done in a scalar subquery in the `ORDER BY` clause. The following is the result of the second example:

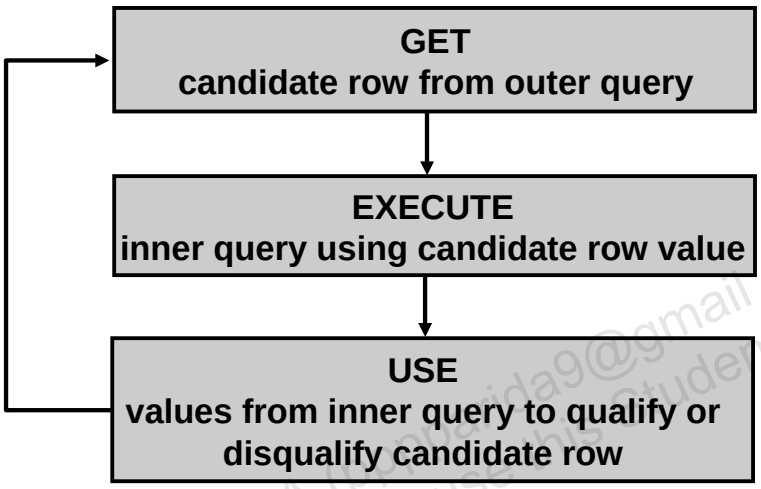
	EMPLOYEE_ID	LAST_NAME
1	205	Higgins
2	206	Gietz
3	200	Whalen
4	100	King
5	101	Kochhar
6	102	De Haan
7	109	Faviet
8	108	Greenberg
9	112	Urman
10	111	Sciarra
11	110	Chen
12	113	Popp
13	203	Mavris

96	134	Rogers
97	135	Gee
98	136	Philtanker
99	137	Ladwig
100	138	Stiles
101	139	Seo
102	140	Patel
103	141	Rajs
104	142	Davies
105	143	Matos
106	181	Fleaur
107	178	Grant

The second example uses a correlated subquery. In a correlated subquery, the subquery references a column from a table referred to in the parent statement. Correlated subqueries are explained later in this lesson.

Correlated Subqueries

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



Correlated Subqueries

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


ORACLE

Correlated Subqueries (continued)

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

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

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

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

Using Correlated Subqueries

The example in the slide determines 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.

Using Correlated Subqueries

Display details of those employees who have changed jobs at least twice.

```
SELECT e.employee_id, last_name,e.job_id
FROM   employees e
WHERE  2 <= (SELECT COUNT(*)
             FROM   job_history
             WHERE  employee_id = e.employee_id);
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID
1	101	Kochhar	AD_VP
2	176	Taylor	SA_REP
3	200	Whalen	AD_ASST

ORACLE

Using Correlated Subqueries (continued)

The example in the slide displays the details of those employees who have changed jobs at least twice. 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.EMPLOYEE_ID`.)
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(*)` group function is evaluated based on the value of the `E.EMPLOYEE_ID` 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 the `EMPLOYEE_ID` from the table in the outer query.

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

ORACLE

EXISTS Operator

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.

Find Employees Who Have At Least One Person Reporting to Them

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees outer
WHERE  EXISTS ( SELECT 'X'
                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
...				
18	205	Higgins	AC_MGR	110

Using the EXISTS Operator

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 'X'
                  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

...

15	260	Recruiting
16	270	Payroll

ORACLE

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);
0 rows selected.
```

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.

Correlated UPDATE

Use a correlated subquery to update rows in one table based on rows from another table.

```
UPDATE table1 alias1
SET    column = (SELECT expression
                 FROM   table2 alias2
                 WHERE  alias1.column =
                        alias2.column);
```


ORACLE

Correlated UPDATE

In the case of the UPDATE statement, you can use a correlated subquery to update rows in one table based on rows from another table.

Using Correlated UPDATE

- Denormalize the EMPL6 table by adding a column to store the department name.
- Populate the table by using a correlated update.

```
ALTER TABLE empl6
ADD(department_name VARCHAR2(25));
```

```
UPDATE empl6 e
SET    department_name =
              (SELECT department_name
               FROM   departments d
               WHERE  e.department_id = d.department_id);
```


ORACLE

Correlated UPDATE (continued)

The example in the slide denormalizes the EMPL6 table by adding a column to store the department name and then populates the table by using a correlated update.

Following is another example for a correlated update.

Problem Statement

The REWARDS table has a list of employees who have exceeded expectations in their performance. Use a correlated subquery to update rows in the EMPL6 table based on rows from the REWARDS table:

```
UPDATE empl6
SET    salary = (SELECT empl6.salary + rewards.pay_raise
                 FROM   rewards
                 WHERE  employee_id  =
                        empl6.employee_id
                 AND  payraise_date =
                      (SELECT MAX(payraise_date)
                      FROM   rewards
                      WHERE  employee_id = empl6.employee_id))
WHERE  empl6.employee_id
IN     (SELECT employee_id FROM rewards);
```

Correlated UPDATE (continued)

This example uses the REWARDS table. The REWARDS table has the columns EMPLOYEE_ID, PAY_RAISE, and PAYRAISE_DATE. Every time an employee gets a pay raise, a record with the details of the employee ID, the amount of the pay raise, and the date of receipt of the pay raise is inserted into the REWARDS table. The REWARDS table can contain more than one record for an employee. The PAYRAISE _DATE column is used to identify the most recent pay raise received by an employee.

In the example, the SALARY column in the EMPL6 table is updated to reflect the latest pay raise received by the employee. This is done by adding the current salary of the employee with the corresponding pay raise from the REWARDS table.

Correlated DELETE

Use a correlated subquery to delete rows in one table based on rows from another table.

```
DELETE FROM table1 alias1
WHERE  column operator
              (SELECT expression
               FROM   table2 alias2
               WHERE  alias1.column = alias2.column);
```


ORACLE

Correlated DELETE

In the case of a DELETE statement, you can use a correlated subquery to delete only those rows that also exist in another table. If you decide that you will maintain only the last four job history records in the JOB_HISTORY table, then when an employee transfers to a fifth job, you delete the oldest JOB_HISTORY row by looking up the JOB_HISTORY table for the MIN(START_DATE) for the employee. The following code illustrates how the preceding operation can be performed using a correlated DELETE:

```
DELETE FROM emp_history JH
WHERE employee_id =
      (SELECT employee_id
       FROM employees E
       WHERE JH.employee_id = E.employee_id
       AND START_DATE =
             (SELECT MIN(start_date)
              FROM job_history JH
              WHERE JH.employee_id = E.employee_id)
              AND 5 >  (SELECT COUNT(*)
                        FROM job_history JH
                        WHERE JH.employee_id = E.employee_id
                        GROUP BY EMPLOYEE_ID
                        HAVING COUNT(*) >= 4));
```

Using Correlated DELETE

Use a correlated subquery to delete only those rows from the EMPL6 table that also exist in the EMP_HISTORY table.

```
DELETE FROM empl6 E
WHERE employee_id =
           (SELECT employee_id
            FROM   emp_history
            WHERE  employee_id = E.employee_id);
```


ORACLE

Correlated DELETE (continued)

Example

Two tables are used in this example. They are:

- The EMPL6 table, which provides details of all the current employees
- The EMP_HISTORY table, which provides details of previous employees

EMP_HISTORY contains data regarding previous employees, so it would be erroneous if the same employee's record existed in both the EMPL6 and EMP_HISTORY tables. You can delete such erroneous records by using the correlated subquery shown in the slide.

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 improves performance.

ORACLE

WITH Clause

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

Using the WITH clause, write a query to display the department name and total salaries for those departments whose total salary is greater than the average salary across departments.

ORACLE

WITH Clause: Example

The problem in the slide would require the following intermediate calculations:

1. Calculate the total salary for every department, and store the result using a WITH clause.
2. Calculate the average salary across departments, and store the result using a WITH clause.
3. Compare the total salary calculated in the first step with the average salary calculated in the second step. If the total salary for a particular department is greater than the average salary across departments, then display the department name and the total salary for that department.

The solution for this problem is provided on the next page.

WITH Clause: Example

```
WITH
dept_costs  AS (
   SELECT d.department_name, SUM(e.salary) AS dept_total
   FROM   employees e JOIN departments d
   ON     e.department_id = d.department_id
   GROUP BY d.department_name),
avg_cost    AS (
   SELECT SUM(dept_total)/COUNT(*) AS dept_avg
   FROM   dept_costs)
SELECT *
FROM   dept_costs
WHERE  dept_total >
       (SELECT dept_avg
        FROM avg_cost)
ORDER BY department_name;
```

ORACLE

WITH Clause: Example (continued)

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 names DEPT_COSTS and AVG_COST and then uses them in the body of the main query. Internally, the WITH clause is resolved either as an in-line 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.

The output generated by the SQL code in the slide is as follows:

	DEPARTMENT_NAME	DEPT_TOTAL
1	Sales	304500
2	Shipping	156400

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.

Summary

In this lesson, you should have learned that:

- A multiple-column subquery returns more than one column
- Multiple-column comparisons can be pairwise or nonpairwise
- A multiple-column subquery can also be used in the FROM clause of a SELECT statement

ORACLE

Summary

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:

- 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

Summary

- Correlated subqueries are useful whenever a subquery must return a different result for each candidate row
- The EXISTS operator is a Boolean operator that tests the presence of a value
- Correlated subqueries can be used with SELECT, UPDATE, and DELETE statements
- You can use the WITH clause to use the same query block in a SELECT statement when it occurs more than once

Summary (continued)

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. Using the WITH clause, you can reuse the same query when it is costly to reevaluate 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

Practice 6: Overview

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

Practice 6

1. Write a query to display the last name, department number, and salary of any employee whose department number and salary both match the department number and salary of any employee who earns a commission.

	LAST_NAME	DEPARTMENT_ID	SALARY
1	Russell	80	14000
2	Partners	80	13500
3	Errazuriz	80	12000
...			
34	Livingston	80	8400

2. Display the last name, department name, and salary of any employee whose salary and commission match the salary and commission of any employee located in location ID 1700.

	LAST_NAME	DEPARTMENT_NAME	SALARY
1	King	Executive	24000
2	De Haan	Executive	17000
3	Kochhar	Executive	17000
4	Higgins	Accounting	12000
5	Greenberg	Finance	12000
...			
35	Whalen	Administration	4400
36	Gietz	Accounting	8300

3. Create a query to display the last name, hire date, and salary for all employees who have the same salary and commission as Kochhar.
Note: Do not display Kochhar in the result set.

	LAST_NAME	HIRE_DATE	SALARY
1	De Haan	13-JAN-1993	17000

4. Create a query to display the employees who earn a salary that is higher than the salary of all the sales managers (`JOB_ID = 'SA_MAN'`). Sort the results on salary from the highest to the lowest.

	LAST_NAME	JOB_ID	SALARY
1	King	AD_PRES	24000
2	De Haan	AD_VP	17000
3	Kochhar	AD_VP	17000

Practice 6 (continued)

5. Display the employee ID, last name, and department ID of those employees who live in cities whose name begins with *T*.

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	202	Fay	20
2	201	Hartstein	20

6. Write a query to find all employees who earn more than the average salary in their departments. Display the last name, salary, department ID, and the average salary for the department. Sort by average salary. Use aliases for the columns retrieved by the query as shown in the sample output.

	ENAME	SALARY	DEPTNO	DEPT_AVG
1	Chung	3800	50	3475.555555555555555555555555555555555556
2	Bell	4000	50	3475.555555555555555555555555555555555556
3	Sarchand	4200	50	3475.555555555555555555555555555555555556
4	Bull	4100	50	3475.555555555555555555555555555555555556
5	Vollman	6500	50	3475.555555555555555555555555555555555556
6	Ladwig	3600	50	3475.555555555555555555555555555555555556
7	Rajs	3500	50	3475.555555555555555555555555555555555556
8	Fripp	8200	50	3475.555555555555555555555555555555555556
9	Mourgos	5800	50	3475.555555555555555555555555555555555556
...				
34	Hall	9000	80	8955.882352941176470588235294117647058824
35	Hartstein	13000	20	9500
36	Higgins	12000	110	10150
37	King	24000	90	19333.333333333333333333333333333333333

7. Find all employees who are not supervisors.
   a. First do this using the `NOT EXISTS` operator.

	LAST_NAME
1	Ernst
2	Austin
3	Pataballa
4	Lorentz
5	Faviet
6	Chen
...	
88	Baer
89	Gietz

   b. Can this be done by using the `NOT IN` operator? How, or why not?

Practice 6 (continued)

8. Write a query to display the last names of the employees who earn less than the average salary in their departments.

	LAST_NAME
1	Chen
2	Sciarra
3	Urman
4	Popp
5	Khoo
6	Baida
7	Tobias
8	Himuro

...

9. Write a query to display the last names of the employees who have one or more coworkers in their departments with later hire dates but higher salaries.

	LAST_NAME
1	Pataballa
2	Austin
3	Faviet
4	Sciarra
5	Tobias
6	Bell
7	Sarchand
8	Rajs

...

Practice 6 (continued)

10. Write a query to display the employee ID, last name, and department name of all employees.
    Note: Use a scalar subquery to retrieve the department name in the SELECT statement.

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT
1	205	Higgins	Accounting
2	206	Gietz	Accounting
3	200	Whalen	Administration
4	100	King	Executive
5	101	Kochhar	Executive

...

102	140	Patel	Shipping
103	141	Rajs	Shipping
104	142	Davies	Shipping
105	143	Matos	Shipping
106	181	Fleaur	Shipping
107	178	Grant	(null)

11. Write a query to display the department names of those departments whose total salary cost is above one-eighth (1/8) the total salary cost of the whole company. Use the WITH clause to write this query. Name the query SUMMARY.

	DEPARTMENT_NAME	DEPT_TOTAL
1	Sales	304500
2	Shipping	156400