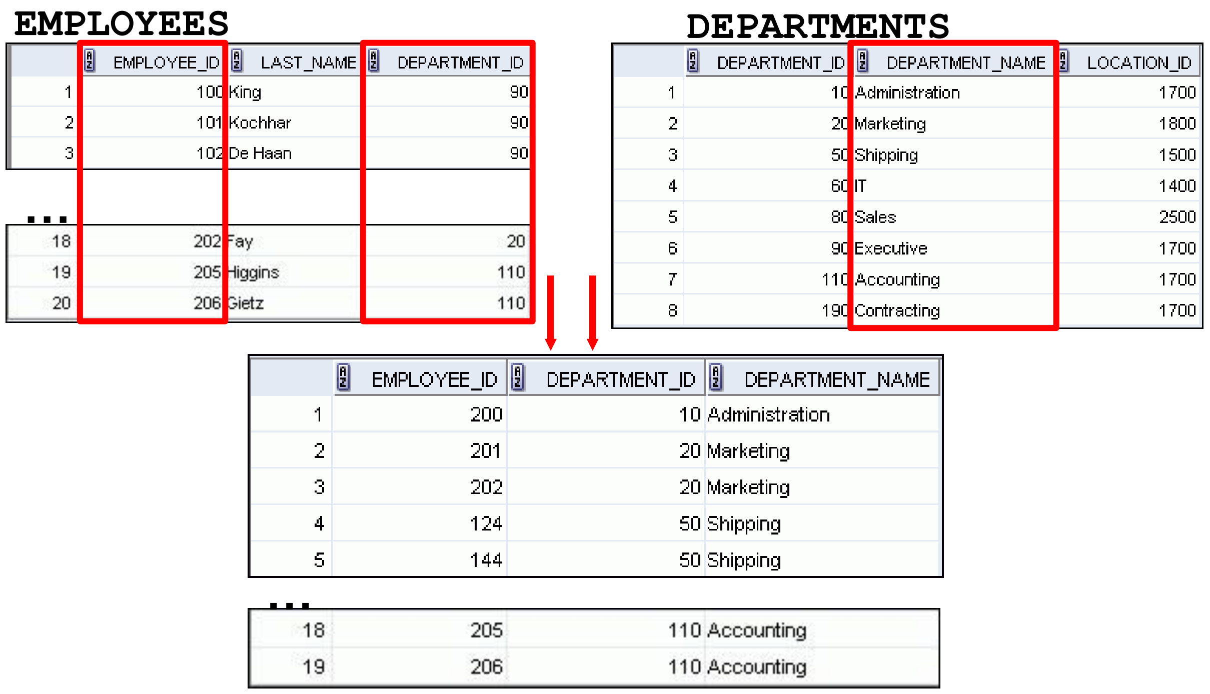
Lab # 6: Joins and Sub-Queries

Objective:

This lab familiarizes the student with the method to obtain data from more than one table and with the use of joins.

Obtaining Data from Multiple Tables



EMPLOYEES

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	100	King	90
2	101	Kochhar	90
3	102	De Haan	90
...			
18	202	Fay	20
19	205	Higgins	110
20	206	Gietz	110

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Administration	1700
2	20	Marketing	1800
3	50	Shipping	1500
4	60	IT	1400
5	80	Sales	2500
6	90	Executive	1700
7	110	Accounting	1700
8	190	Contracting	1700

	EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	200	10	Administration
2	201	20	Marketing
3	202	20	Marketing
4	124	50	Shipping
5	144	50	Shipping
...			
18	205	110	Accounting
19	206	110	Accounting

Sometimes you need to use data from more than one table. In the example in the slide, the report displays data from two separate tables:

- Employee IDs exist in the EMPLOYEES table.
- Department IDs exist in both the EMPLOYEES and DEPARTMENTS tables.
- Department names exist in the DEPARTMENTS table.

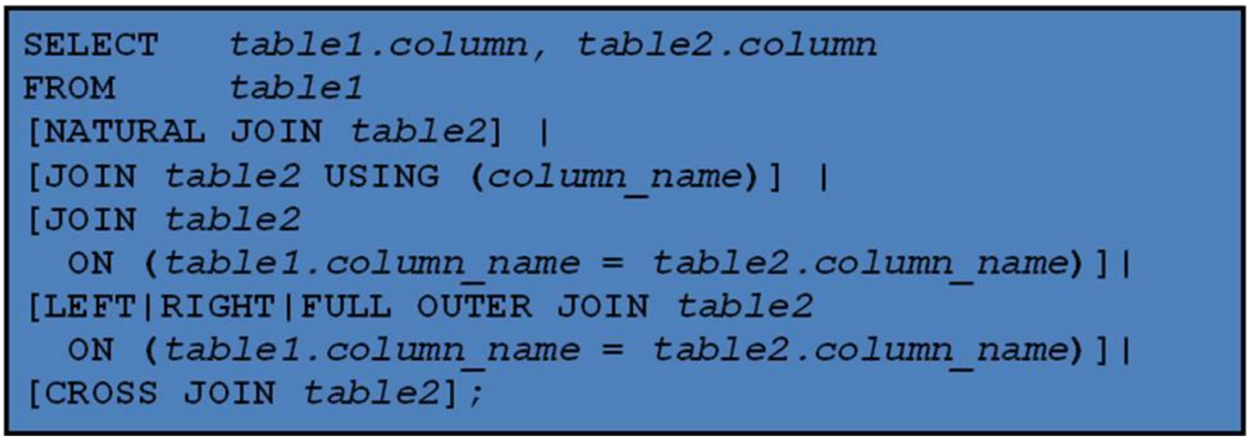
To produce the report, you need to link the EMPLOYEES and DEPARTMENTS tables, and access data from both of them.

Joins that are compliant with the SQL:1999 standard include the following:

- Natural joins:
  - NATURAL JOIN clause
  - USING clause
  - ON clause
- OUTER joins:
  - LEFT OUTER JOIN
  - RIGHT OUTER JOIN
  - FULL OUTER JOIN
- Cross joins

Joining Tables Using SQL:1999 Syntax

Use a join to query data from more than one table:



```
SELECT   table1.column, table2.column
FROM     table1
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
  ON (table1.column_name = table2.column_name)]|
[LEFT|RIGHT|FULL OUTER JOIN table2
  ON (table1.column_name = table2.column_name)]|
[CROSS JOIN table2];
```

In the syntax:

table1.column	denotes the table and the column from which data is retrieved
NATURAL JOIN	joins two tables based on the same column name
JOIN *table2* USING *column_name*	performs an equijoin based on the column name
JOIN *table2 ON table1.column_name* = *table2.column_name*	performs an equijoin based on the condition in the ON clause
LEFT/RIGHT/FULL OUTER	is used to perform OUTER joins
CROSS JOIN	returns a Cartesian product from the two tables

Qualifying Ambiguous Column Names

When joining two or more tables, you need to qualify the names of the columns with the table name to avoid ambiguity. Without the table prefixes, the DEPARTMENT_ID column in the

SELECT list could be from either the DEPARTMENTS table or the EMPLOYEES table. It is necessary to add the table prefix to execute your query. If there are no common column names between the two tables, there is no need to qualify the columns. However, using the table prefix improves performance, because you tell the Oracle server exactly where to find the columns.

However, qualifying column names with table names can be time consuming, particularly if the table names are lengthy. Instead, you can use *table aliases*. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help to keep SQL code smaller, therefore using less memory.

The table name is specified in full, followed by a space and then the table alias. For example, the EMPLOYEES table can be given an alias of e, and the DEPARTMENTS table an alias of d.

Guidelines

- Table aliases can be up to 30 characters in length, but shorter aliases are better than longer ones.
- If a table alias is used for a particular table name in the FROM clause, then that table alias must be substituted for the table name throughout the SELECT statement.
- Table aliases should be meaningful.
- The table alias is valid for only the current SELECT statement.

Creating Natural Joins

You can join tables automatically based on the columns in the two tables that have matching data types and names. You do this by using the NATURAL JOIN keywords. The join can happen on only those columns that have the same names and data types in both tables. If the columns have the same name but different data types, then the NATURAL JOIN syntax causes an error.

Retrieving Records with Natural Joins

In the following example, the LOCATIONS table is joined to the DEPARTMENT table by the LOCATION_ID column, which is the only column of the same name in both tables. If other common columns were present, the join would have used them all.

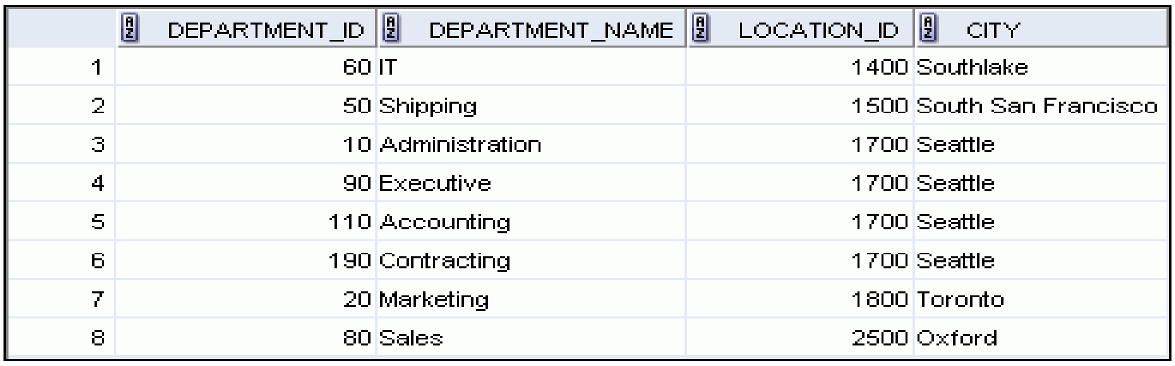
Example:

```
SELECT department_id, department_name,  location_id, city

FROM   departments

NATURAL JOIN locations ;
```

Output:



	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
1	60	IT	1400	Southlake
2	50	Shipping	1500	South San Francisco
3	10	Administration	1700	Seattle
4	90	Executive	1700	Seattle
5	110	Accounting	1700	Seattle
6	190	Contracting	1700	Seattle
7	20	Marketing	1800	Toronto
8	80	Sales	2500	Oxford

Natural Joins with a WHERE Clause

Additional restrictions on a natural join are implemented by using a WHERE clause. The following example limits the rows of output to those with a department ID equal to 20 or 50:

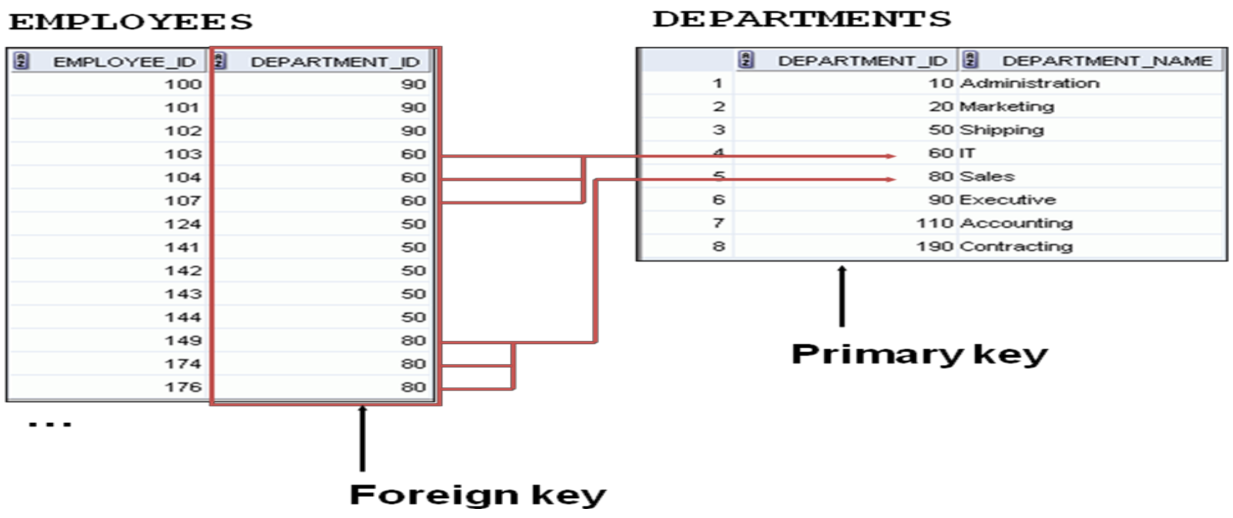
```
SELECT  department_id, department_name,
 location_id, city
FROM    departments
NATURAL JOIN locations
WHERE   department_id IN (20, 50);
```

Creating Joins with the USING Clause

Natural joins use all columns with matching names and data types to join the tables. The USING clause can be used to specify only those columns that should be used for an equijoin.

Joining Column Names

To determine an employee‘s department name, you compare the value in the DEPARTMENT_ID column in the EMPLOYEES table with the DEPARTMENT_ID values in the DEPARTMENTS table. The relationship between the EMPLOYEES and DEPARTMENTS tables is an *equijoin*; that is, values in the DEPARTMENT_ID column in both the tables must be equal. Frequently, this type of join involves primary and foreign key complements.



EMPLOYEE_ID	DEPARTMENT_ID
100	90
101	90
102	90
103	60
104	60
107	60
124	50
141	50
142	50
143	50
144	50
149	80
174	80
176	80

	DEPARTMENT_ID	DEPARTMENT_NAME
1	10	Administration
2	20	Marketing
3	50	Shipping
4	60	IT
5	80	Sales
6	90	Executive
7	110	Accounting
8	190	Contracting

Retrieving Records with the USING Clause

<u>Example:</u>

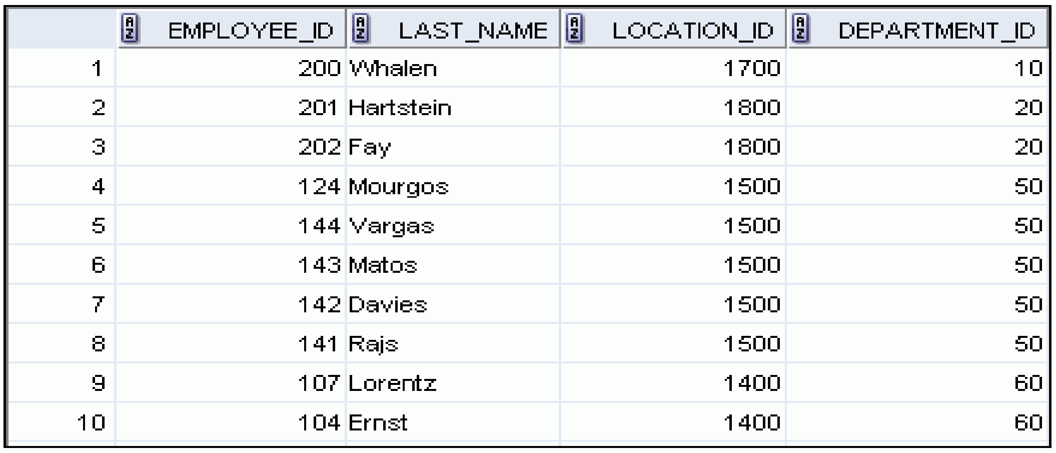
In the following example, the DEPARTMENT_ID columns in the EMPLOYEES and DEPARTMENTS tables are joined and thus the LOCATION_ID of the department where an employee works is shown.

SELECT employee_id, last_name, location_id, department_id

FROM employees JOIN departments

USING (department_id) ;

<u>Output:</u>



	EMPLOYEE_ID	LAST_NAME	LOCATION_ID	DEPARTMENT_ID
1	200	Whalen	1700	10
2	201	Hartstein	1800	20
3	202	Fay	1800	20
4	124	Mourgos	1500	50
5	144	Vargas	1500	50
6	143	Matos	1500	50
7	142	Davies	1500	50
8	141	Rajs	1500	50
9	107	Lorentz	1400	60
10	104	Ernst	1400	60

Using Table Aliases with the USING clause

When joining with the USING clause, you cannot qualify a column that is used in the USING clause itself. Furthermore, if that column is used anywhere in the SQL statement, you cannot alias it. For example, in the following example, you should not alias the location_id column in the WHERE clause because the column is used in the USING clause.

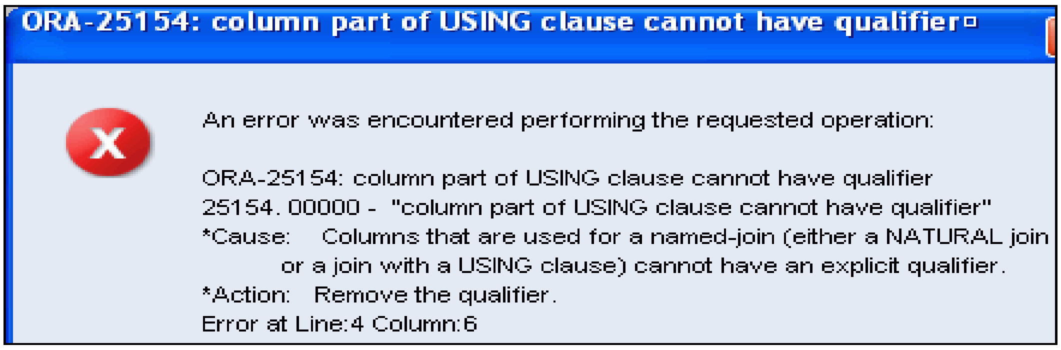
<u>Example</u>:

SELECT l.city, d.department_name

FROM   locations l JOIN departments d

USING (location_id)

WHERE d.location_id = 1400;



```
ORA-25154: column part of USING clause cannot have qualifier

An error was encountered performing the requested operation:

ORA-25154: column part of USING clause cannot have qualifier
25154. 00000 -  "column part of USING clause cannot have qualifier"
*Cause:    Columns that are used for a named-join (either a NATURAL join
           or a join with a USING clause) cannot have an explicit qualifier.
*Action:   Remove the qualifier.
Error at Line:4 Column:6
```

The columns that are referenced in the USING clause should not have a qualifier (table name or alias) anywhere in the SQL statement. For example, the following statement is valid:

SELECT l.city, d.department_name

FROM   locations l JOIN departments d USING (location_id)

WHERE location_id = 1400;

Because, other columns that are common in both the tables, but not used in the USING clause, must be prefixed with a table alias otherwise you get the –column ambiguously defined‖ error.

In the following statement, manager_id is present in both the employees and departments table and if manager_id is not prefixed with a table alias, it gives a –column ambiguously defined‖ error.

The following statement is valid:

SELECT first_name, d.department_name, d.manager_id

FROM   employees e JOIN departments d USING (department_id)

WHERE department_id = 50;

Creating Joins with the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- Use the ON clause to specify arbitrary conditions or specify columns to join.
- The join condition is separated from other search conditions.

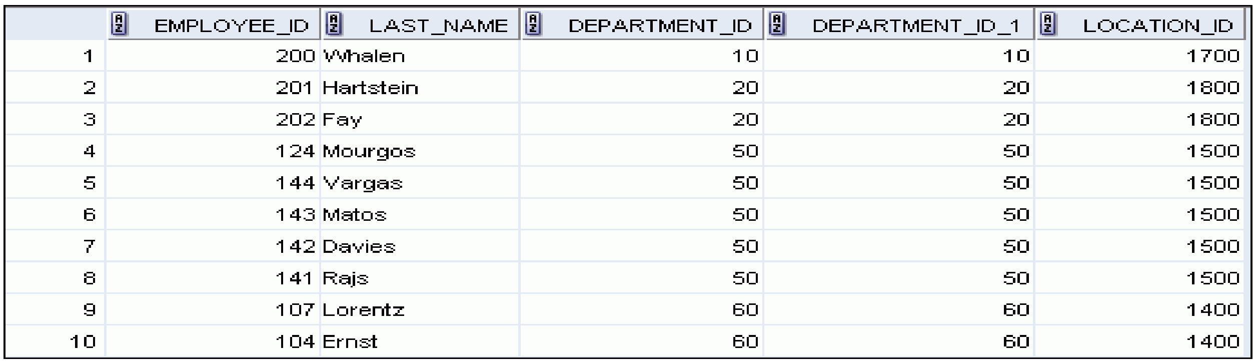
The ON clause makes code easy to understand.

Retrieving Records with the ON Clause

Example:

```
SELECT e.employee_id, e.last_name, e.department_id, d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id);
```

Output:



	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID_1	LOCATION_ID
1	200	Whalen	10	10	1700
2	201	Hartstein	20	20	1800
3	202	Fay	20	20	1800
4	124	Mourgos	50	50	1500
5	144	Vargas	50	50	1500
6	143	Matos	50	50	1500
7	142	Davies	50	50	1500
8	141	Rajs	50	50	1500
9	107	Lorentz	60	60	1400
10	104	Ernst	60	60	1400

In this example, the DEPARTMENT_ID columns in the EMPLOYEES and DEPARTMENTS table are joined using the ON clause. Wherever a department ID in the EMPLOYEES table equals a department ID in the DEPARTMENTS table, the row is returned. The table alias is necessary to qualify the matching column_names.

You can also use the ON clause to join columns that have different names. The parenthesis around the joined columns as in the slide example, (e.department_id = d.department_id) is optional. So, even ON e.department_id = d.department_id will work.

Creating Three-Way Joins with the ON Clause

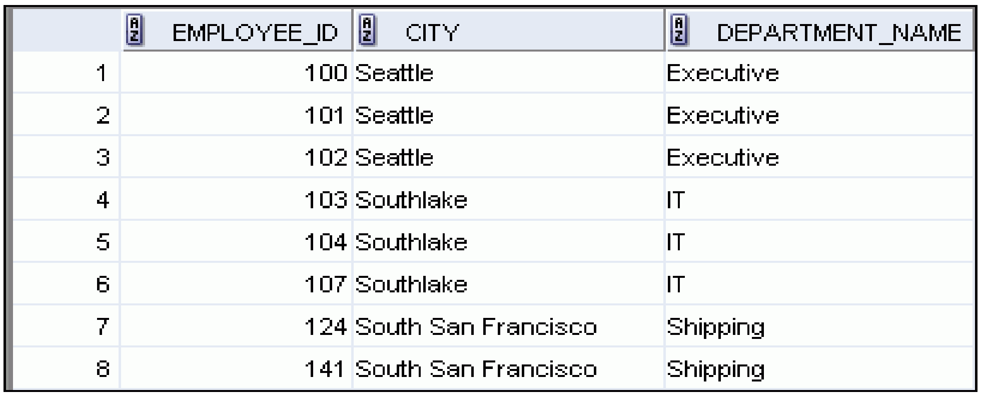
A three-way join is a join of three tables. Joins are performed from left to right. So, the first join to be performed is EMPLOYEES JOIN DEPARTMENTS. The first join condition can reference columns in EMPLOYEES and DEPARTMENTS but cannot reference columns in LOCATIONS. The second join condition can reference columns from all three tables.

Example:

The following example can also be accomplished with the USING clause:

```
SELECT employee_id, city, department_name
FROM   employees e
JOIN   departments d
ON     d.department_id = e.department_id
JOIN   locations l
ON     d.location_id = l.location_id;
```

Output:



	EMPLOYEE_ID	CITY	DEPARTMENT_NAME
1	100	Seattle	Executive
2	101	Seattle	Executive
3	102	Seattle	Executive
4	103	Southlake	IT
5	104	Southlake	IT
6	107	Southlake	IT
7	124	South San Francisco	Shipping
8	141	South San Francisco	Shipping

Example:

```
SELECT e.employee_id, l.city, d.department_name
FROM employees e
JOIN departments d
USING (department_id)
JOIN locations l
USING (location_id);
```

Applying Additional Conditions to a Join

Use the AND clause or the WHERE clause to apply additional conditions:

Example:

```
SELECT e.employee_id, e.last_name, e.department_id,  d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id)
```

```
AND    e.manager_id = 149 ;
```

Or

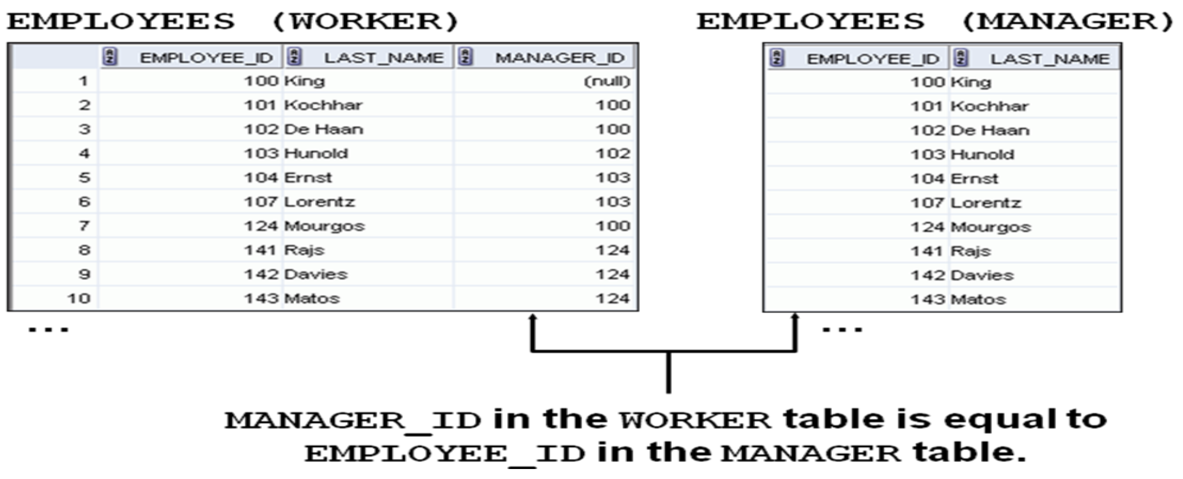
```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id)
WHERE  e.manager_id = 149 ;
```

Joining a Table to Itself

Sometimes you need to join a table to itself. To find the name of each employee‘s manager, you need to join the EMPLOYEES table to itself, or perform a self-join. For example, to find the name of Lorentz‘s manager, you need to:

- Find Lorentz in the EMPLOYEES table by looking at the LAST_NAME column
- Find the manager number for Lorentz by looking at the MANAGER_ID column. Lorentz‘s manager number is 103.
- Find the name of the manager with EMPLOYEE_ID 103 by looking at the LAST_NAME column. Hunold‘s employee number is 103, so Hunold is Lorentz‘s manager.

In this process, you look in the table twice. The first time you look in the table to find Lorentz in the LAST_NAME column and the MANAGER_ID value of 103. The second time you look in the EMPLOYEE_ID column to find 103 and the LAST_NAME column to find Hunold.


EMPLOYEES (WORKER)
EMPLOYEES (MANAGER)
...
...
MANAGER_ID in the WORKER table is equal to EMPLOYEE_ID in the MANAGER table.

	EMPLOYEE_ID	LAST_NAME	MANAGER_ID
1	100	King	(null)
2	101	Kochhar	100
3	102	De Haan	100
4	103	Hunold	102
5	104	Ernst	103
6	107	Lorentz	103
7	124	Mourgos	100
8	141	Rajs	124
9	142	Davies	124
10	143	Matos	124

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos
141	Rajs
142	Davies
143	Matos

Self-Joins Using the ON Clause

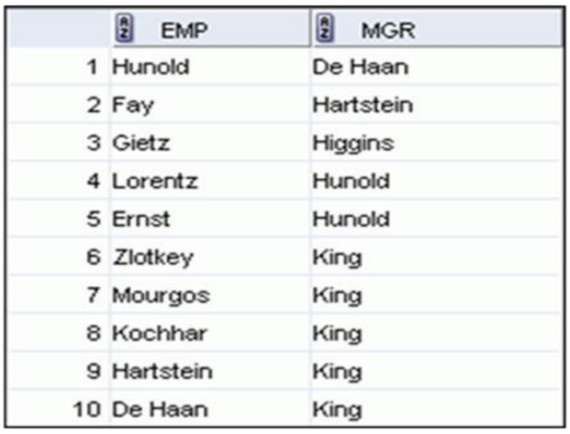
The ON clause can also be used to join columns that have different names, within the same table or in a different table.

The following example is a self-join of the EMPLOYEES table, based on the EMPLOYEE_ID and MANAGER_ID columns.

Example:

```
SELECT worker.last_name emp, manager.last_name mgr
FROM   employees worker JOIN employees manager
ON    (worker.manager_id = manager.employee_id);
```

Output:



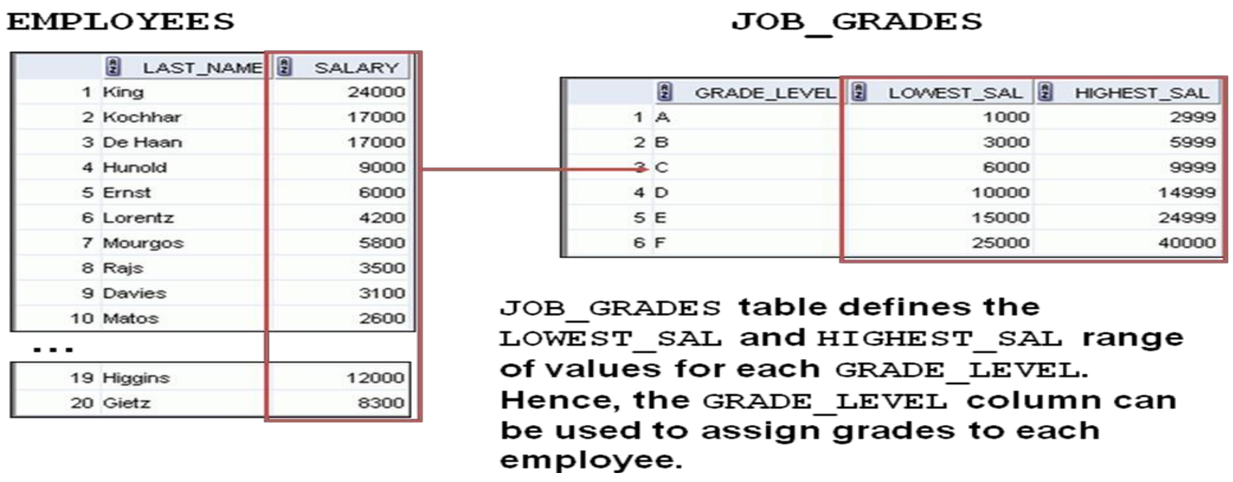
	EMP	MGR
1	Hunold	De Haan
2	Fay	Hartstein
3	Gietz	Higgins
4	Lorentz	Hunold
5	Ernst	Hunold
6	Zlotkey	King
7	Mourgos	King
8	Kochhar	King
9	Hartstein	King
10	De Haan	King

...

Nonequijoins

A nonequijoin is a join condition containing something other than an equality operator.

The relationship between the EMPLOYEES table and the JOB_GRADES table is an example of a nonequijoin. The SALARY column in the EMPLOYEES table ranges between the values in the LOWEST_SAL and HIGHEST_SAL columns of the JOB_GRADES table. Therefore, each employee can be graded based on their salary. The relationship is obtained using an operator other than the equality (=) operator.



EMPLOYEES

	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000
4	Hunold	9000
5	Ernst	6000
6	Lorentz	4200
7	Mourgos	5800
8	Rajs	3500
9	Davies	3100
10	Matos	2600
...		
19	Higgins	12000
20	Gietz	8300

JOB_GRADES

	GRADE_LEVEL	LOWEST_SAL	HIGHEST_SAL
1	A	1000	2999
2	B	3000	5999
3	C	6000	9999
4	D	10000	14999
5	E	15000	24999
6	F	25000	40000

JOB_GRADES table defines the LOWEST_SAL and HIGHEST_SAL range of values for each GRADE_LEVEL. Hence, the GRADE_LEVEL column can be used to assign grades to each employee.

Retrieving Records with Nonequijoins

The following example creates a nonequijoin to evaluate an employee‘s salary grade. The salary must be *between* any pair of the low and high salary ranges.

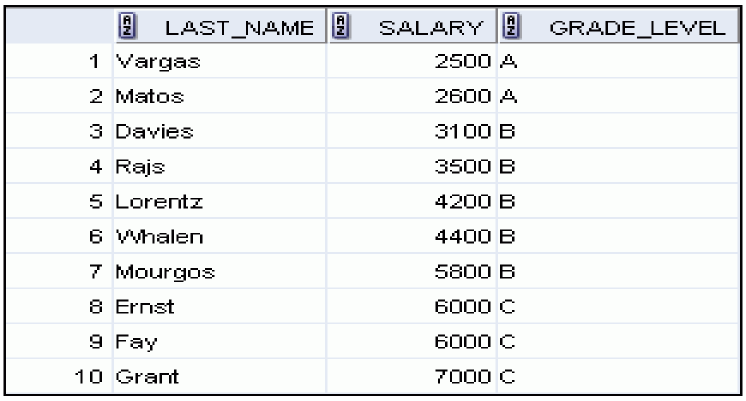
SELECT e.last_name, e.salary, j.grade_level

FROM employees e JOIN job_grades j

ON e.salary

BETWEEN j.lowest_sal AND j.highest_sal;

<u>Output:</u>



	LAST_NAME	SALARY	GRADE_LEVEL
1	Vargas	2500	A
2	Matos	2600	A
3	Davies	3100	B
4	Rajs	3500	B
5	Lorentz	4200	B
6	Whalen	4400	B
7	Mourgos	5800	B
8	Ernst	6000	C
9	Fay	6000	C
10	Grant	7000	C

It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:

- None of the rows in the JOB_GRADES table contain grades that overlap. That is, the salary value for an employee can lie only between the low salary and high salary values of one of the rows in the salary grade table.

- All of the employees‘ salaries lie within the limits provided by the job grade table. That is, no employee earns less than the lowest value contained in the LOWEST_SAL column or more than the highest value contained in the HIGHEST_SAL column.

INNER Versus OUTER Joins

Joining tables with the NATURAL JOIN, USING, or ON clauses results in an INNER join. Any unmatched rows are not displayed in the output. To return the unmatched rows, you can use an OUTER join. An OUTER join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other table satisfy the join condition.

There are three types of OUTER joins:

- LEFT OUTER
- RIGHT OUTER
- FULL OUTER

LEFT OUTER JOIN

Example:

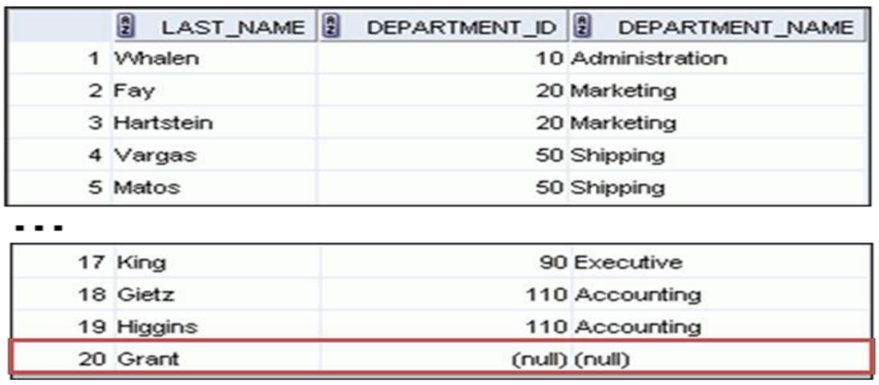
SELECT e.last_name, e.department_id, d.department_name

FROM employees e LEFT OUTER JOIN departments d

ON (e.department_id = d.department_id) ;

This query retrieves all the rows in the EMPLOYEES table, which is the left table, even if there is no match in the DEPARTMENTS table.

Output:



	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Fay	20	Marketing
3	Hartstein	20	Marketing
4	Vargas	50	Shipping
5	Matos	50	Shipping

...

17	King	90	Executive
18	Gietz	110	Accounting
19	Higgins	110	Accounting
20	Grant	(null)	(null)

RIGHT OUTER JOIN

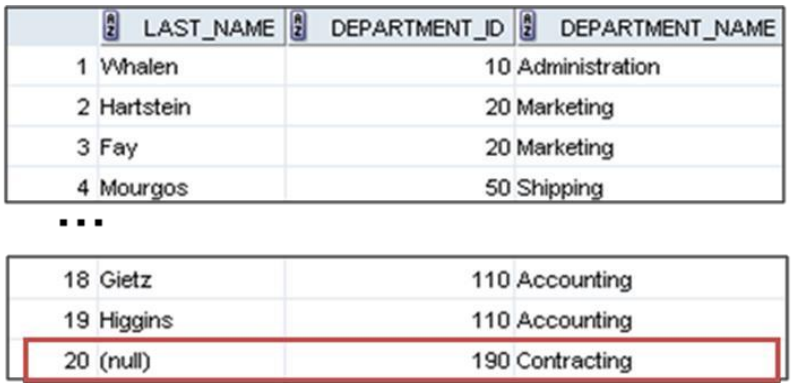
SELECT e.last_name, d.department_id, d.department_name

FROM employees e RIGHT OUTER JOIN departments d

ON (e.department_id = d.department_id) ;

This query retrieves all the rows in the DEPARTMENTS table, which is the table at the right, even if there is no match in the EMPLOYEES table.

<u>Output:</u>



	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Hartstein	20	Marketing
3	Fay	20	Marketing
4	Mourgos	50	Shipping
...			
18	Gietz	110	Accounting
19	Higgins	110	Accounting
20	(null)	190	Contracting

FULL OUTER JOIN

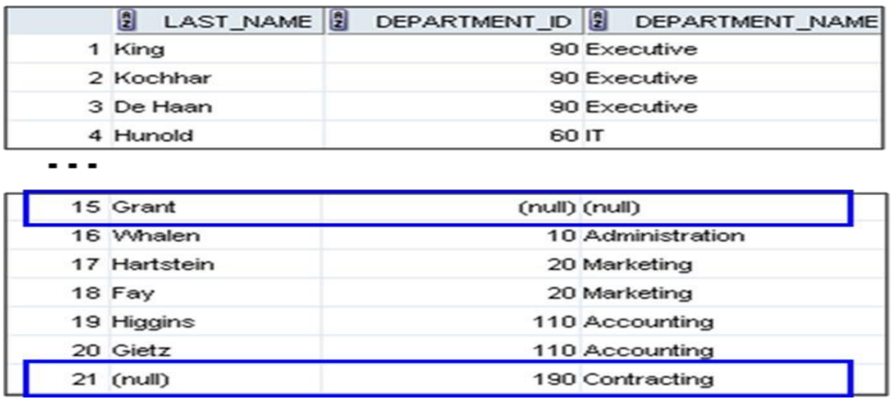
SELECT e.last_name, d.department_id, d.department_name

FROM   employees e FULL OUTER JOIN departments d

ON   (e.department_id = d.department_id) ;

This query retrieves all rows in the EMPLOYEES table, even if there is no match in the DEPARTMENTS table. It also retrieves all rows in the DEPARTMENTS table, even if there is no match in the EMPLOYEES table.

<u>Output:</u>



	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	King	90	Executive
2	Kochhar	90	Executive
3	De Haan	90	Executive
4	Hunold	60	IT
...			
15	Grant	(null)	(null)
16	Whalen	10	Administration
17	Hartstein	20	Marketing
18	Fay	20	Marketing
19	Higgins	110	Accounting
20	Gietz	110	Accounting
21	(null)	190	Contracting

Cartesian Products

- A Cartesian product is formed when:
  - A join condition is omitted
  - A join condition is invalid
  - All rows in the first table are joined to all rows in the second table

- To avoid a Cartesian product, always include a valid join condition.

When a join condition is invalid or omitted completely, the result is a *Cartesian product,* in which all combinations of rows are displayed. All rows in the first table are joined to all rows in the second table.

A Cartesian product tends to generate a large number of rows and the result is rarely useful. You should, therefore, always include a valid join condition unless you have a specific need to combine all rows from all tables.

However, Cartesian products are useful for some tests when you need to generate a large number of rows to simulate a reasonable amount of data.

Creating Cross Joins

- The CROSS JOIN clause produces the cross-product of two tables.
- This is also called a Cartesian product between the two tables.

Example:

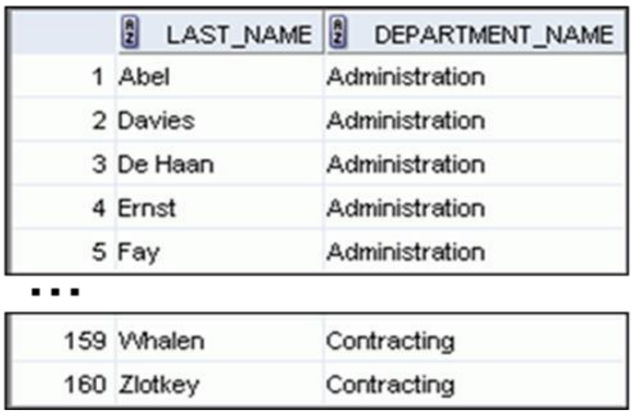
The following example produces a Cartesian product of the EMPLOYEES and DEPARTMENTS tables.

SELECT last_name, department_name

FROM   employees

CROSS JOIN departments ;

Output:

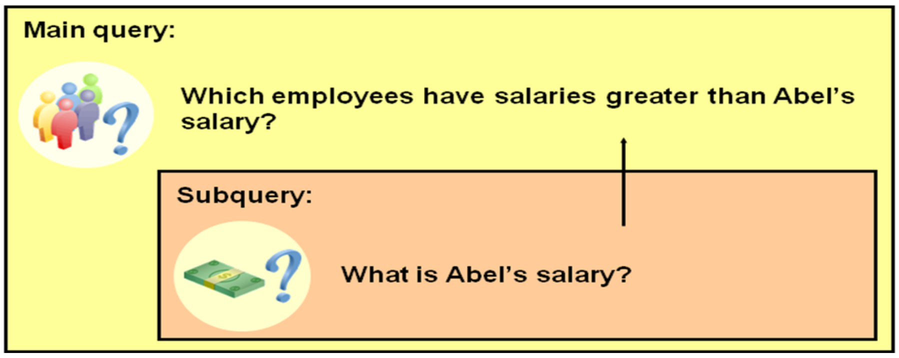


```
      LAST_NAME   DEPARTMENT_NAME
  1   Abel        Administration
  2   Davies      Administration
  3   De Haan     Administration
  4   Ernst       Administration
  5   Fay         Administration
...
159   Whalen      Contracting
160   Zlotkey     Contracting
```

Using a Subquery to Solve a Problem

Suppose you want to write a query to find out who earns a salary greater than Abel‘s salary. To solve this problem, you need *two* queries: one to find how much Abel earns, and a second query to find who earns more than that amount.

You can solve this problem by combining the two queries, placing one query *inside* the other query.



The inner query (or *subquery*) returns a value that is used by the outer query (or *main query*). Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the search value in the second query.

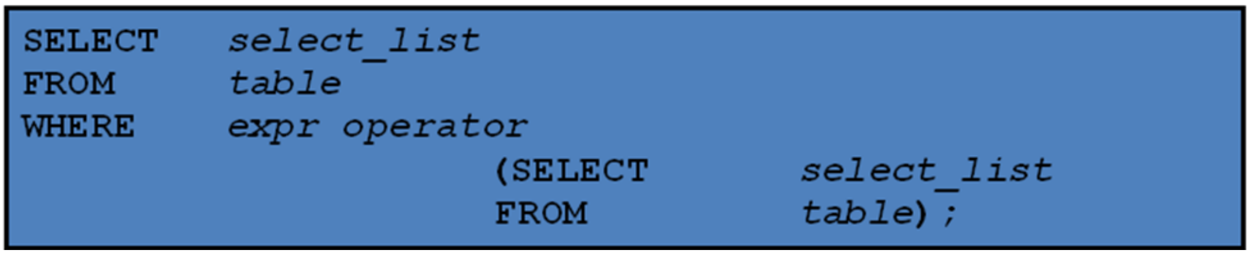
Subquery Syntax

A subquery is a SELECT statement that is embedded in the clause of another SELECT statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

You can place the subquery in a number of SQL clauses, including the following:

- WHERE clause
- HAVING clause
- FROM clause

Syntax:



```
SELECT    select_list
FROM      table
WHERE     expr operator
                        (SELECT      select_list
                         FROM        table);
```

In the syntax:

operator includes a comparison condition such as >, =, or IN

Note: Comparison conditions fall into two classes: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators (IN, ANY, ALL).

The subquery is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement. The subquery generally executes first, and its output is used to complete the query condition for the main (or outer) query.

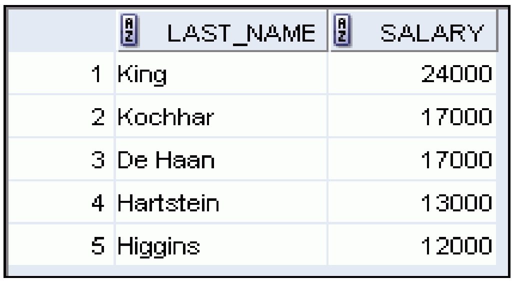
Using a Subquery

Example:

```
SELECT last_name, salary
FROM   employees
WHERE  salary >
         (SELECT salary
          FROM   employees
          WHERE  last_name = 'Abel');
```

In the example, the inner query determines the salary of employee Abel that is 11000. The outer query takes the result of the inner query and uses this result to display all the employees who earn more than employee Abel.

Output:



	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000
4	Hartstein	13000
5	Higgins	12000

Guidelines for Using Subqueries

- A subquery must be enclosed in parentheses.
- Place the subquery on the right side of the comparison condition for readability. However, the subquery can appear on either side of the comparison operator.
- Two classes of comparison conditions are used in subqueries: single-row operators and multiple-row operators.

Types of Subqueries

- **Single-row subqueries:** Queries that return only one row from the inner SELECT statement
- **Multiple-row subqueries:** Queries that return more than one row from the inner SELECT statement

Single-Row Subqueries

A single-row subquery is one that returns one row from the inner SELECT statement. This type of subquery uses a single-row operator.

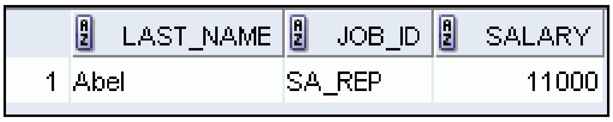
Executing Single-Row Subqueries

A SELECT statement can be considered as a query block.

Example 1:

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id = (SELECT
            job_id FROM
            employees
             WHERE  last_name = 'Taylor')
AND    salary >
            (SELECT salary
             FROM   employees
             WHERE  last_name = 'Taylor');
```

Output:



	LAST_NAME	JOB_ID	SALARY
1	Abel	SA_REP	11000

The above example displays employees who do the same job as –Taylor,‖ but earn more salary than him.

The example consists of three query blocks: the outer query and two inner queries. The inner query blocks are executed first, producing the query results SA_REP and 8600, respectively. The outer query block is then processed and uses the values that were returned by the inner queries to complete its search conditions.

Both inner queries return single values (SA_REP and 8600, respectively), so this SQL statement is called a single-row subquery.

Note: The outer and inner queries can get data from different tables.

Example 2:

Display the employees whose job ID is the same as that of employee 141:

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id =
              (SELECT job_id
               FROM   employees
               WHERE  employee_id = 141);
```

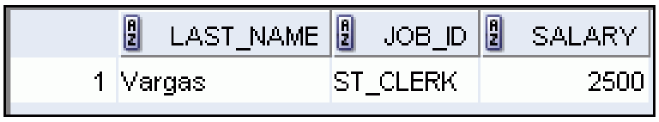
Using Group Functions in a Subquery

You can display data from a main query by using a group function in a subquery to return a single row. The subquery is in parentheses and is placed after the comparison condition.

Example:

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  salary =
               (SELECT MIN(salary)
                FROM   employees);
```

Output:



	LAST_NAME	JOB_ID	SALARY
1	Vargas	ST_CLERK	2500

This example displays the employee last name, job ID, and salary of all employees whose salary is equal to the minimum salary. The MIN group function returns a single value (2500) to the outer query.

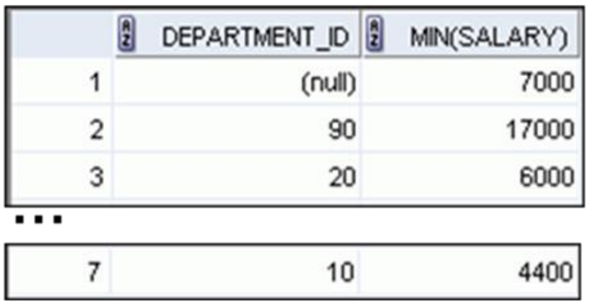
The HAVING Clause with Subqueries

You can use subqueries not only in the WHERE clause, but also in the HAVING clause. The SQL statement in the following example displays all the departments that have a minimum salary greater than that of department 50.

Example:

```
SELECT   department_id, MIN(salary)
FROM     employees
GROUP BY department_id
HAVING   MIN(salary) >
                  (SELECT MIN(salary)
                   FROM   employees
                   WHERE  department_id = 50);
```

Output*:*



	DEPARTMENT_ID	MIN(SALARY)
1	(null)	7000
2	90	17000
3	20	6000
...		
7	10	4400

Example:

Find the job with the lowest average salary.

```
SELECT   job_id, AVG(salary)
FROM     employees
GROUP BY job_id
HAVING   AVG(salary) = (SELECT   MIN(AVG(salary))
                    FROM     employees
                    GROUP BY job_id);
```

Using Single-row operator with multiple-row subquery

A common error with subqueries occurs when more than one row is returned for a single-row subquery.

In the SQL statement in the slide, the subquery contains a GROUP BY clause, which implies that the subquery will return multiple rows, one for each group that it finds. In this case, the results of the subquery are 4400, 6000, 2500, 4200, 7000, 17000, and 8300.

The outer query takes those results and uses them in its WHERE clause. The WHERE clause contains an equal (=) operator, a single-row comparison operator that expects only one value. The = operator cannot accept more than one value from the subquery and, therefore, generates the error.

Example:

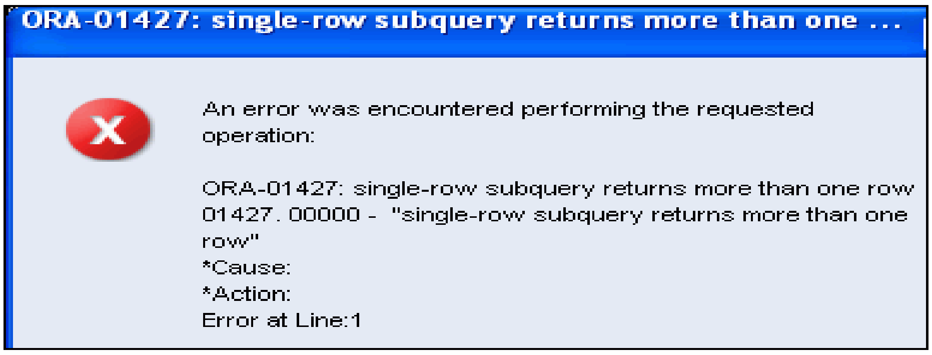
```
SELECT employee_id, last_name

FROM   employees

WHERE  salary =

            (SELECT   MIN(salary)

             FROM     employees;
```



```
ORA-01427: single-row subquery returns more than one ...

An error was encountered performing the requested
operation:

ORA-01427: single-row subquery returns more than one row
01427. 00000 -  "single-row subquery returns more than one
row"
*Cause:
*Action:
Error at Line:1
```

To correct this error, change the = operator to IN.

No Rows Returned by the Inner Query

A common problem with subqueries occurs when no rows are returned by the inner query.

Example:

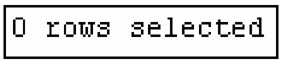
In the SQL statement in following example, the subquery contains a WHERE clause. Presumably, the intention is to find the employee whose name is Haas. The statement is correct, but selects no rows when executed.

```
SELECT last_name, job_id

FROM   employees

WHERE  job_id =
            (SELECT job_id
             FROM   employees
             WHERE  last_name = 'Haas');
```

Output:

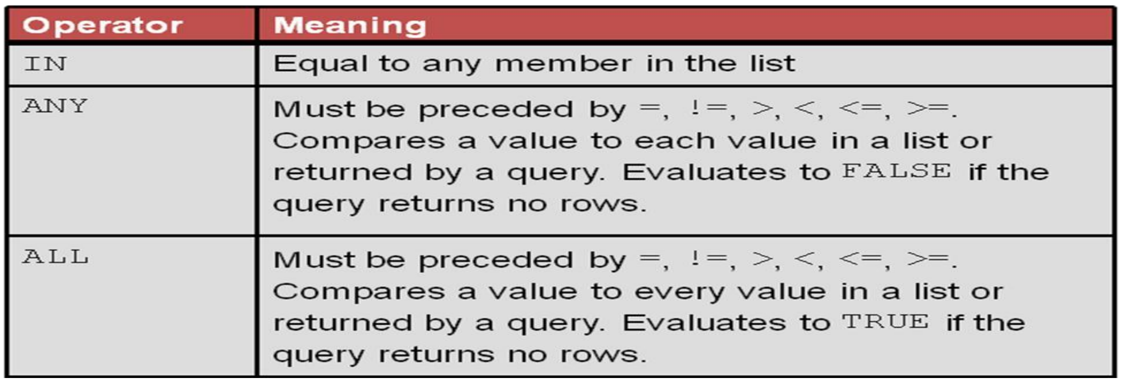


```
0 rows selected
```

Because, there is no employee having name Haas. So the subquery returns no rows. The outer query takes the results of the subquery (null) and uses these results in its WHERE clause. The outer query finds no employee with a job ID equal to null, and so returns no rows. If a job existed with a value of null, the row is not returned because comparison of two null values yields a null; therefore, the WHERE condition is not true.

Multiple-Row Subqueries

Subqueries that return more than one row are called multiple-row subqueries. You use a multiple-row operator, instead of a single-row operator, with a multiple-row subquery. The multiple-row operator expects one or more values:



Operator	Meaning
IN	Equal to any member in the list
ANY	Must be preceded by =, !=, >, <, <=, >=. Compares a value to each value in a list or returned by a query. Evaluates to FALSE if the query returns no rows.
ALL	Must be preceded by =, !=, >, <, <=, >=. Compares a value to every value in a list or returned by a query. Evaluates to TRUE if the query returns no rows.

Example:

```
 SELECT last_name, salary, department_id
   FROM   employees
   WHERE  salary IN (SELECT   MIN(salary)
                     FROM     employees
                     GROUP BY department_id);
```

Example:

Find the employees who earn the same salary as the minimum salary for each department.

The inner query is executed first, producing a query result. The main query block is then processed and uses the values that were returned by the inner query to complete its search condition. In fact, the main query appears to the Oracle server as follows:

```
SELECT last_name, salary, department_id
FROM   employees
WHERE  salary IN (2500, 4200, 4400, 6000, 7000, 8300, 8600, 17000);
```

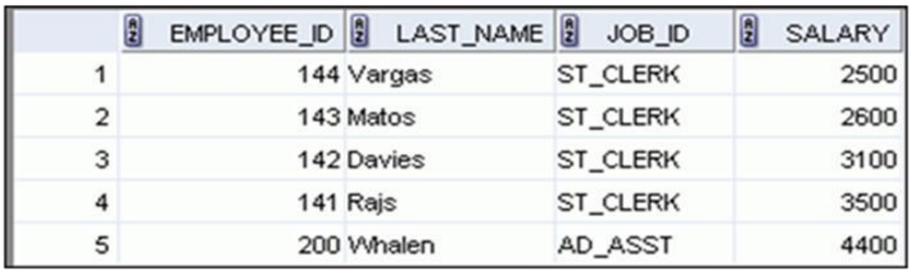
Using the ANY Operator in Multiple-Row Subqueries

The ANY operator (and its synonym, the SOME operator) compares a value to *each* value returned by a subquery. The following example displays employees who are not IT programmers and whose salary is less than that of any IT programmer. The maximum salary that a programmer earns is $9,000.

Example:

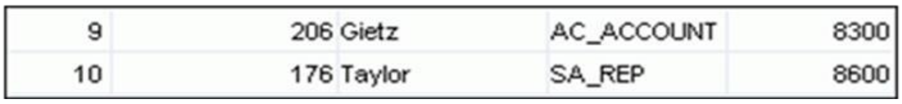
```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary < ANY (SELECT
             salary FROM
             employees
              WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

Output:



	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	144	Vargas	ST_CLERK	2500
2	143	Matos	ST_CLERK	2600
3	142	Davies	ST_CLERK	3100
4	141	Rajs	ST_CLERK	3500
5	200	Whalen	AD_ASST	4400

▪ ▪ ▪



9	206	Gietz	AC_ACCOUNT	8300
10	176	Taylor	SA_REP	8600

<ANY means less than the maximum. >ANY means more than the minimum. =ANY is equivalent to IN.

Using the ALL Operator in Multiple-Row Subqueries

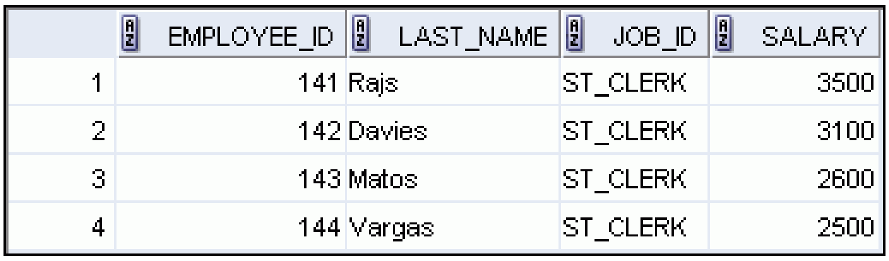
The ALL operator compares a value to *every* value returned by a subquery.

Example:

This example displays employees whose salary is less than the salary of all employees with a job ID of IT_PROG and whose job is not IT_PROG.

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary < ALL (SELECT
           salary FROM
           employees
            WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

Output:



	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	141	Rajs	ST_CLERK	3500
2	142	Davies	ST_CLERK	3100
3	143	Matos	ST_CLERK	2600
4	144	Vargas	ST_CLERK	2500

>ALL means more than the maximum and <ALL means less than the minimum.

The NOT operator can be used with IN, ANY, and ALL operators.

Null Values in a Subquery

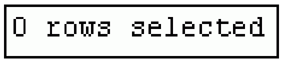
Example:

The SQL statement in the following attempts to display all the employees who do not have any subordinates. Logically, this SQL statement should have returned 12 rows. However, the SQL statement does not return any rows. One of the values returned by the inner query is a null value, and, therefore, the entire query returns no rows.

```
SELECT emp.last_name
FROM   employees emp
WHERE  emp.employee_id NOT IN
```

```
                    (SELECT mgr.manager_id
                     FROM   employees mgr);
```

Output:



```
0 rows selected
```

The reason is that all conditions that compare a null value result in a null. So whenever null values are likely to be part of the results set of a subquery, do not use the NOT IN operator. The NOT IN operator is equivalent to <> ALL.

Notice that the null value as part of the results set of a subquery is not a problem if you use the IN operator. The IN operator is equivalent to =ANY. For example, to display the employees who have subordinates, use the following SQL statement:

```
SELECT
emp.last_name FROM
employees emp
WHERE
emp.employee_id  IN
            (SELECT mgr.manager_id
             FROM   employees mgr);
```

Lab Task:

1. The HR department wants to determine the names of all the employees who were hired after Davies. Create a query to display the name and hire date of any employee hired after employee Davies.
2. The HR department needs to find the names and hire dates of all the employees who were hired before their managers, along with their managers‘ names and hire dates.
3. The HR department needs a report of all employees. Write a query to display the last name, department number, and department name for all the employees.
4. The HR department needs a report of employees in Toronto. Display the last name, job, department number, and the department name for all employees who work in Toronto.
5. Create a report to display employees‘ last name and employee number along with their manager‘s last name and manager number. Label the columns Employee, Emp#, Manager, and Mgr#, respectively.
6. Create a report for the HR department that displays employee last names, department numbers, and all the employees who work in the same department as a given employee. Give each column an appropriate label.
7. The HR department needs a report on job grades and salaries. To familiarize yourself with the JOB_GRADES table, first show the structure of the JOB_GRADES table. Then create a query that displays the name, job, department name, salary, and grade for all employees
8. The HR department needs a query that prompts the user for an employee last name. The query then displays the last name and hire date of any employee in the same department as the employee whose name they supply (excluding that employee). For example, if the user enters Zlotkey, find all employees who work with Zlotkey (excluding Zlotkey).
9. Write a query that displays the employee number and last name of all employees who work in a department with any employee whose last name contains the letter “u”.
10. The HR department needs a report that displays the last name, department number, and job ID of all employees whose department location ID is 1700.