1

Using SET Operators

Schedule:	Timing	Topic
	20 minutes	Lecture
	20 minutes	Practice
	40 minutes	Total

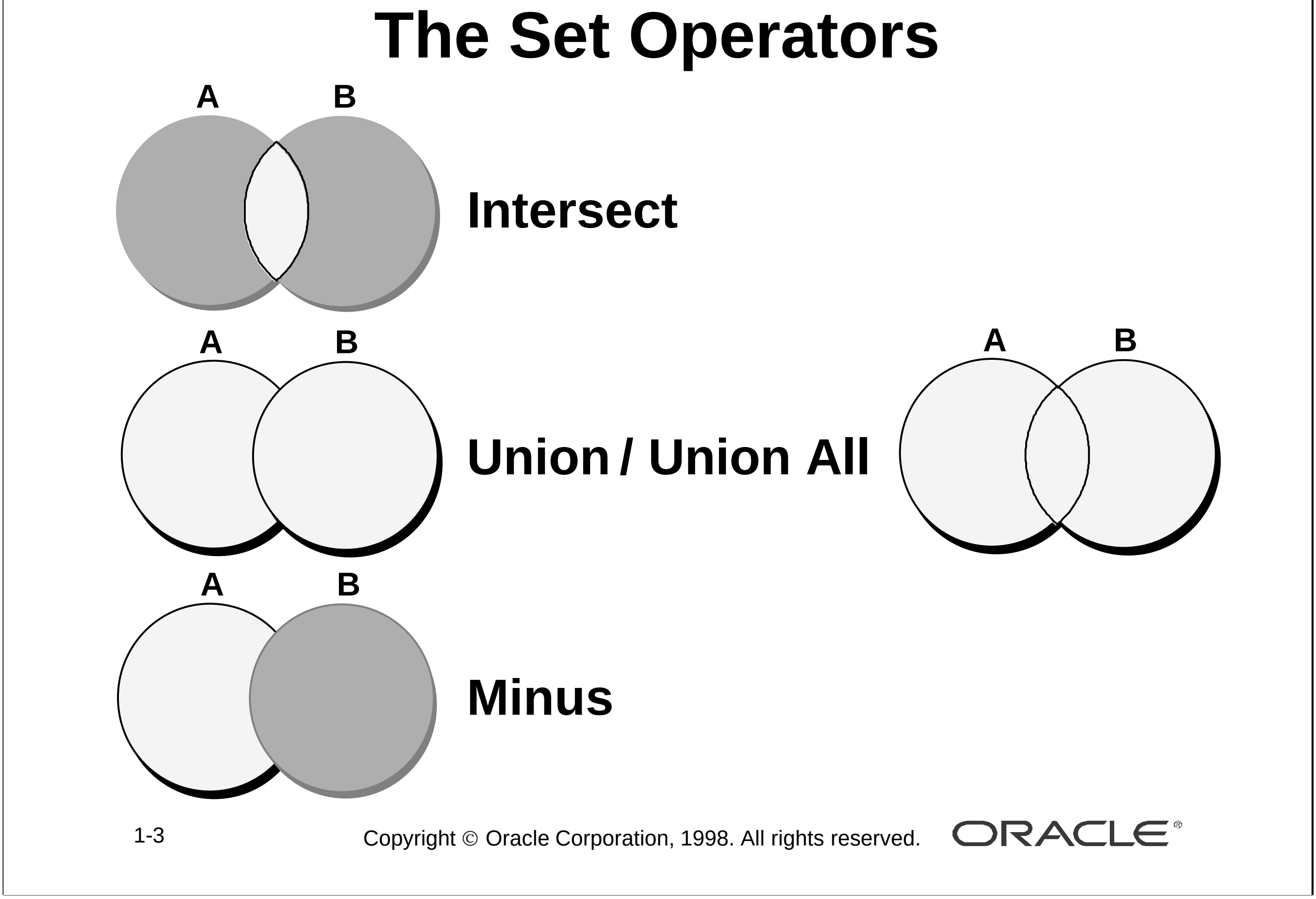
Objectives

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

- Describe the SET operators
- Use a SET operator to combine multiple queries into a single query
- Control the order of rows returned

Lesson Aim

In this lesson, you will learn how to write queries using SET operators.



The Set Operators

Intersect

Union / Union All

Minus

The Set Operator

SET operators combine the results of two or more component queries into one result. Queries containing set operators are called compound queries.

Operator	Returns
INTERSECT	All distinct rows selected by both queries. INTERSECT combines two queries and returns only those rows from the first select statement that are identical to at least one row from the second select statement.
UNION	All rows selected by either query.
UNION ALL	All rows selected by either query, including all duplicates.
MINUS	All distinct rows selected by the first select statement that are not produced in the second select statement.

All SET operators have equal precedence. If a SQL statement contains multiple SET operators, the database evaluates them from left (top) to right (bottom) if no parentheses explicitly specifies another order. To comply with emerging SQL standards, a future version of the database will give the INTERSECT operator greater precedence than the other SET operators, so you should use parentheses to explicitly specify the order of evaluation in queries that use the INTERSECT operator with other SET operators.

Class Management Note

To demonstrate the SET operators presented in this lesson, run the script *emphis.sql* to create the table EMP_HISTORY. In the slide above, the light color in the diagram represents the query result.

Tables Used in this Lesson

EMP

```
   EMPNO ENAME      JOB              MGR HIREDATE        SAL       COMM
DEPTNO
--------- ---------- --------- --------- --------- --------- --------- --------
-
     7839 KING       PRESIDENT           17-NOV-81      5000
10
     7698 BLAKE      MANAGER        7839 01-MAY-81      2850
30
     7782 CLARK      MANAGER        7839 09-JUN-81      1500
10
     7566 JONES      MANAGER        7839 02-APR-81      2975
20
     7654 MARTIN     SALESMAN
30
     7499 ALLEN      SALESMAN
30
     7844 TURNER     SALESMAN
30
     7900 JAMES      CLERK
30
     7521 WARD       SALESMAN
30
     7902 FORD       ANALYST
20
     7369 SMITH      CLERK          7902 17-DEC-80       800
20
```

EMP_HISTORY

```
    EMPID NAME                 TITLE     DATE_OUT
DEPTID
--------- -------------------- --------- --------- --------
-
     6087 SPENCER              OPERATOR  27-NOV-81
20
     6185 VANDYKE              MANAGER   17-JAN-81
10
     6235 BALFORD              CLERK     22-FEB-80
20
     7788 SCOTT                ANALYST   05-MAY-81
20
     7001 JEWELL               ANALYST   10-JUN-81
30
```

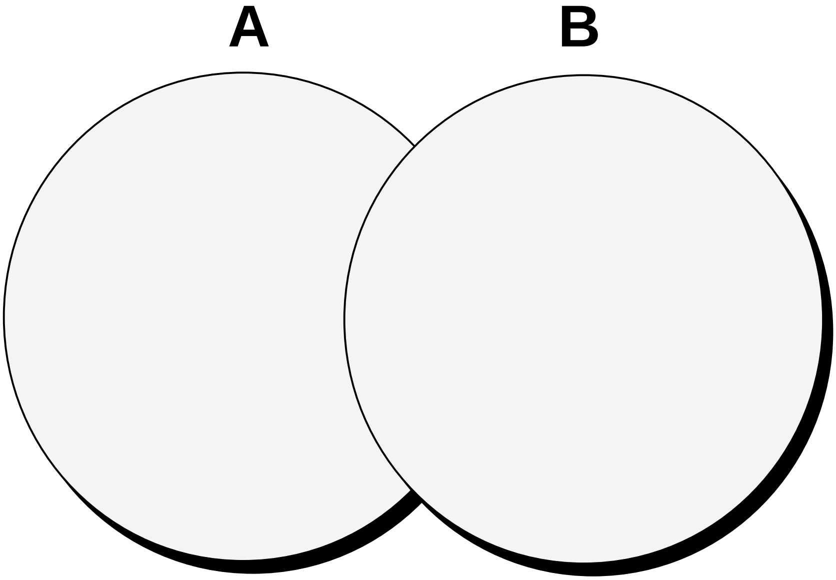
Tables Used in this Lesson

Two tables are used in this course. They are:

- The EMP table, which gives details of all the employees
- The EMP_HISTORY which gives details of previous employees

The script required to create the EMP_HISTORY table is given in the practice at the end of this lesson.

UNION



The UNION Set Operator

UNION is the combination of two tables. Use the UNION operator to return all rows from multiple queries and eliminate any duplicate rows.

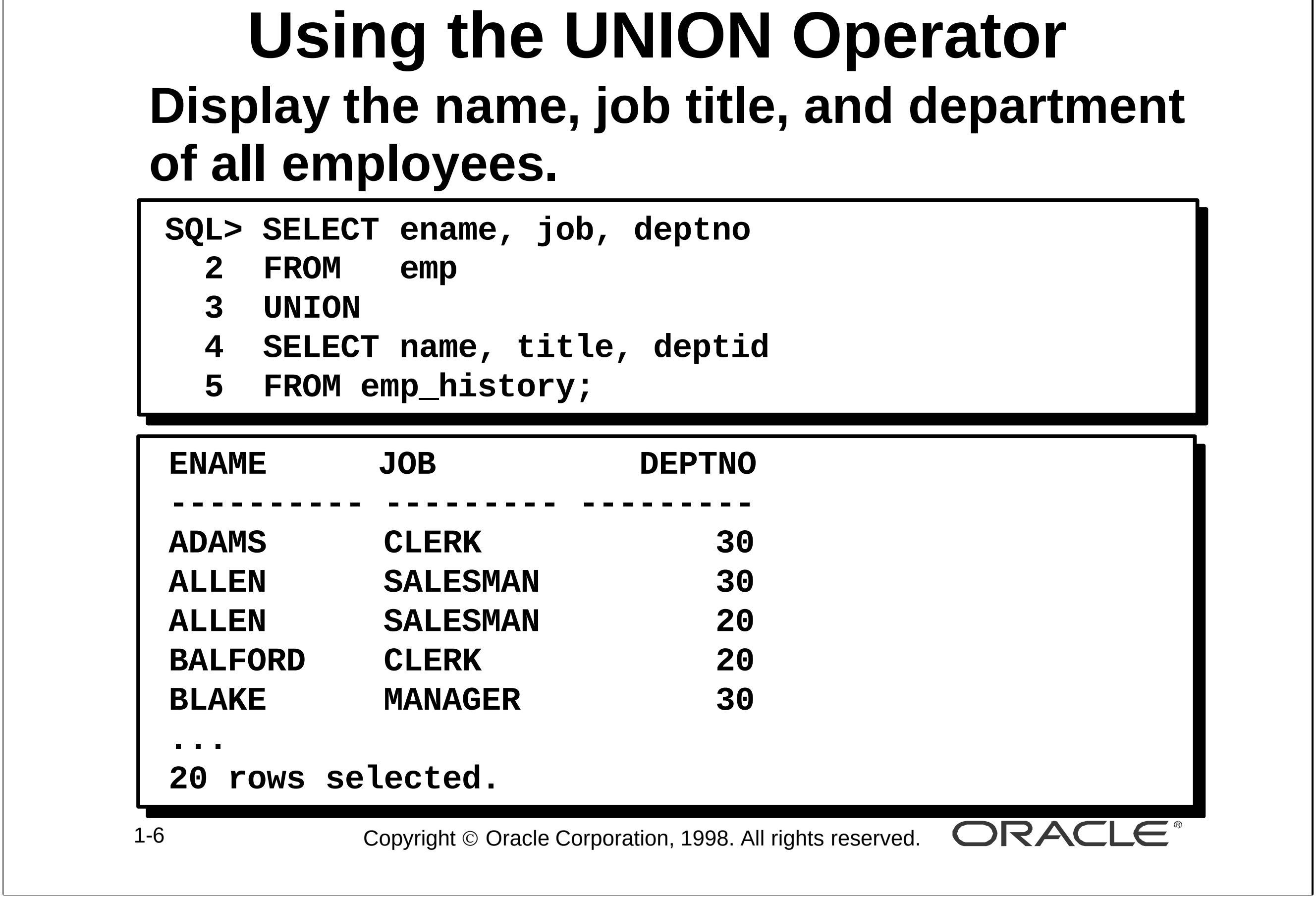
Guidelines

- The number of columns and the datatypes of those columns must be identical between the two select statements. The names of the columns need not be identical.
- UNION operates over all of the columns being selected. For example, if the query on the next page were rewritten to select the employee name and job only, then ALLEN would appear in the results only once.
- NULL columns are ignored during duplicate checking. For example, if ALLEN had a NULL value in the DEPTNO column (if DEPTNO were not a NOT NULL column) in the first select statement rather than a value of 30 as in the second select statement, ALLEN would have appeared in the results set only once.
- The IN operator has a higher precedence than the UNION operator.
- Queries that use UNION in the where clause must have the same number and type of columns in their select list.
- The output is sorted in ascending order by default.

Class Management Note

Demo: *l1union1.sql*

Purpose: To illustrate the UNION set operator.



Using the UNION Operator

Display the name, job title, and department of all employees.

```
SQL> SELECT ename, job, deptno
  2  FROM   emp
  3  UNION
  4  SELECT name, title, deptid
  5  FROM emp_history;
```

```
ENAME      JOB          DEPTNO
---------- --------- ---------
ADAMS      CLERK            30
ALLEN      SALESMAN         30
ALLEN      SALESMAN         20
BALFORD    CLERK            20
BLAKE      MANAGER          30
...
20 rows selected.
```

The UNION Set Operator (continued)

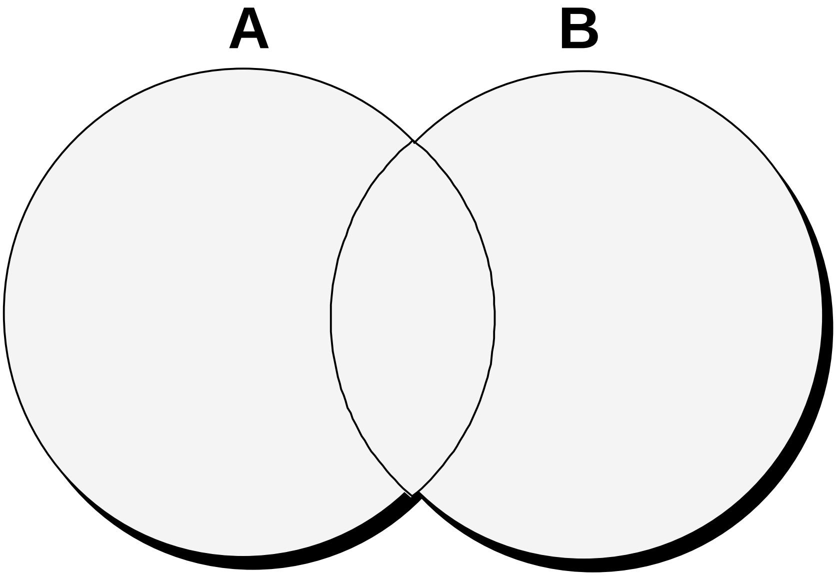
In the slide above 20 rows were selected. Although the combination of the two tables totals more than 20 records only 20 were returned. This is because the UNION operator eliminates any duplicate records.

The EMP and EMP_HISTORY tables have several columns in common. For example ENAME and NAME, JOB and TITLE, and EMPNO and EMPID. But what if you wanted the query to display the employee name, job title, and salary using the UNION operator? Knowing that the salary does not exist in both tables? The following statement matches the ENAME and NAME columns, the JOB and TITLE columns, and adds a literal of 0 to the EMP_HISTORY select statement to match the numeric SAL column in the EMP select statement.

```
SELECT ename, job, sal FROM   emp
UNION
SELECT name, title, 0  FROM   emp_history;
```

```
ENAME      JOB             SAL
---------- --------- ---------
ADAMS      CLERK          1100
ALLEN      SALESMAN          0
ALLEN      SALESMAN       1600
BALFORD    CLERK             0
...
```

UNION ALL



The UNION ALL Operator

Use the UNION ALL operator to return all rows from multiple queries.

Guidelines

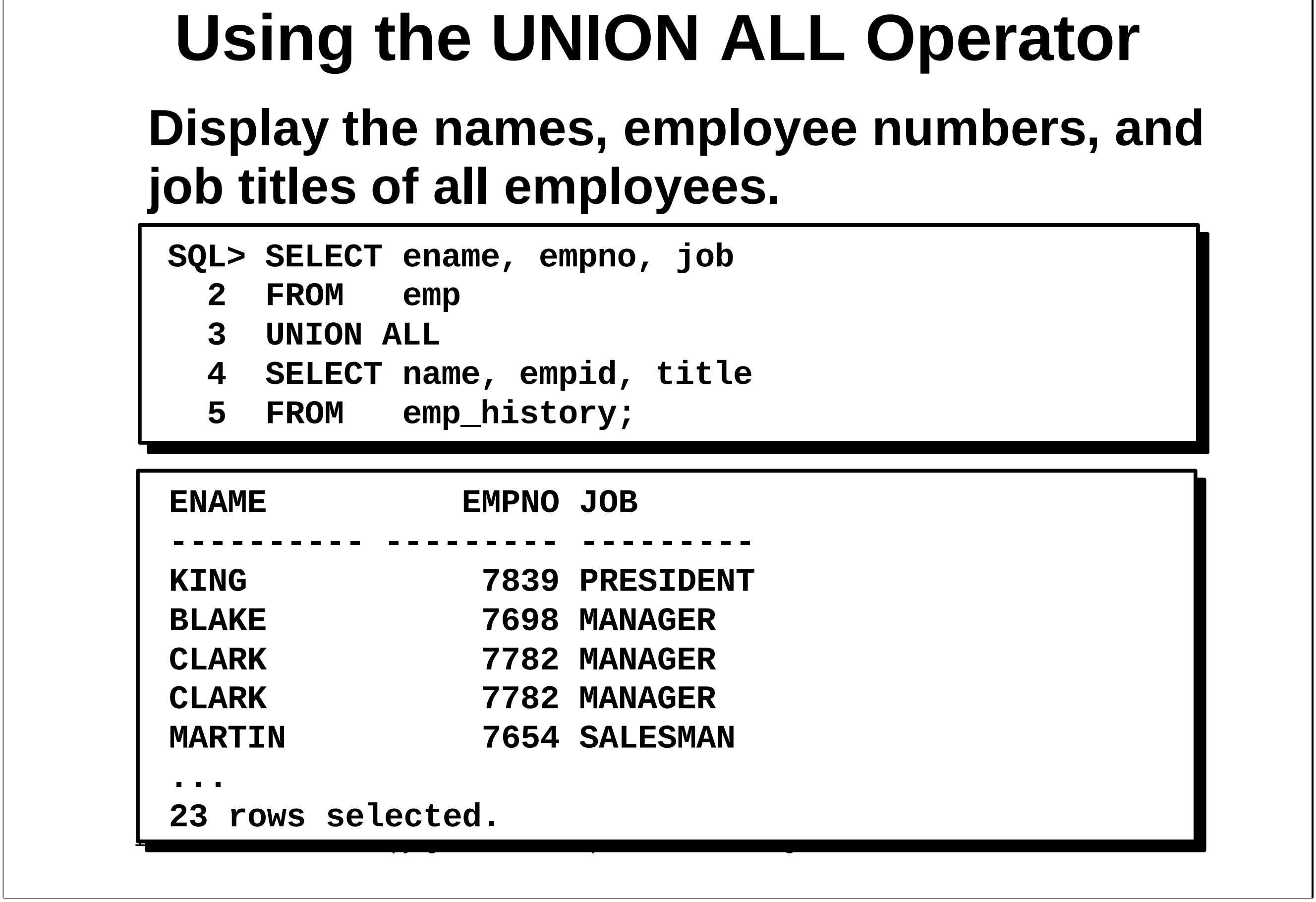
- Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.
- The DISTINCT keyword cannot be used.

Note: With the exception of the above, the guidelines for UNION and UNION ALL are the same.

Class Management Note

Demo: *l1union2.sql*

Purpose: To illustrate the UNION ALL set operator.



Using the UNION ALL Operator

Display the names, employee numbers, and job titles of all employees.

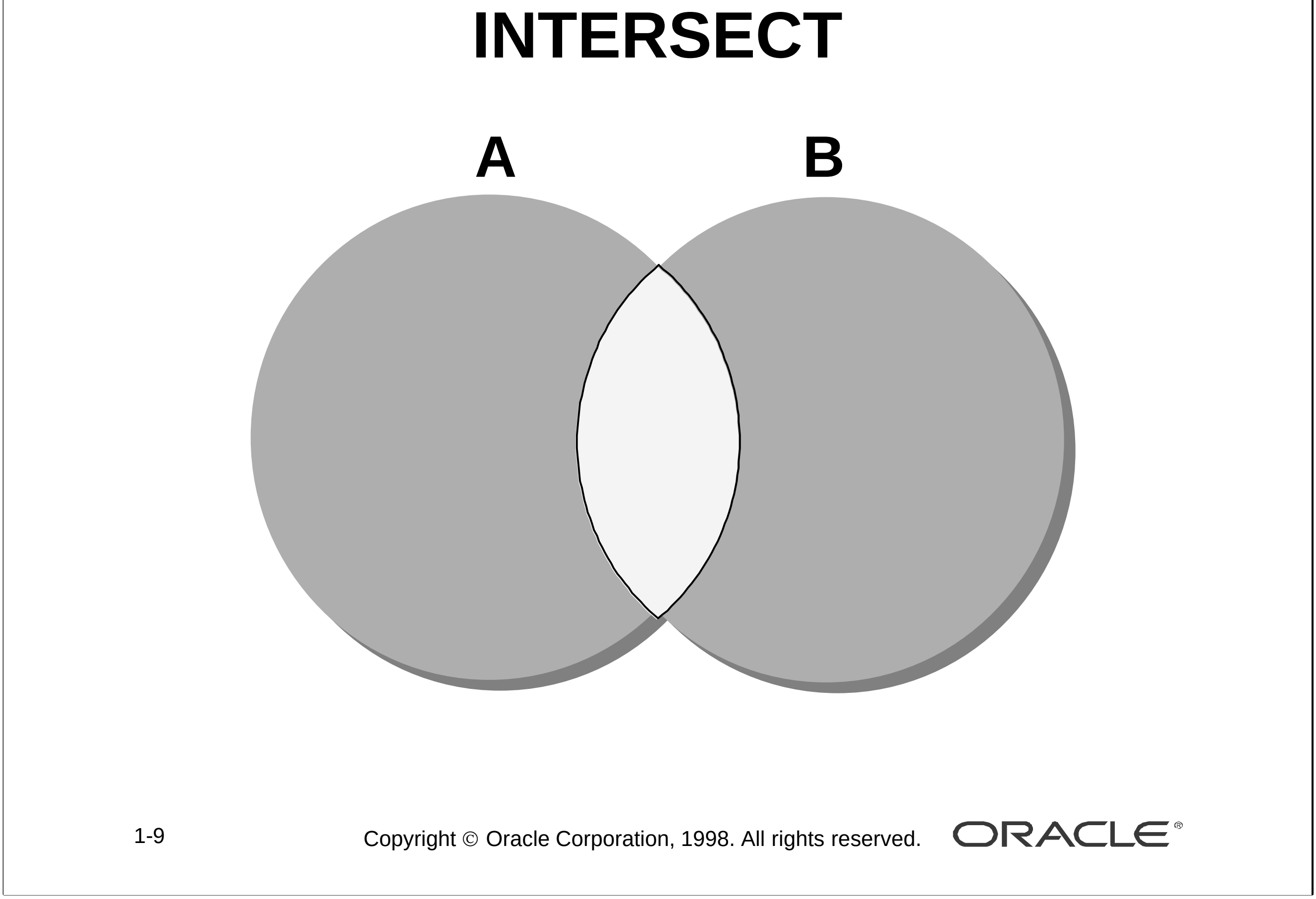
```
SQL> SELECT ename, empno, job
  2  FROM   emp
  3  UNION ALL
  4  SELECT name, empid, title
  5  FROM   emp_history;
```

```
ENAME          EMPNO JOB
---------- --------- ---------
KING            7839 PRESIDENT
BLAKE           7698 MANAGER
CLARK           7782 MANAGER
CLARK           7782 MANAGER
MARTIN          7654 SALESMAN
...
23 rows selected.
```

The UNION ALL Operator (continued)

In the slide, 23 rows were selected. The combination of the two tables totals 23 records. This is because the UNION ALL operator does not eliminate duplicate records. The slide example results contains three sets of duplicate:

```
ENAME              EMPNO JOB
---------- --------- ---------
KING                7839 PRESIDENT
BLAKE               7698 MANAGER
CLARK               7782 MANAGER
CLARK               7782 MANAGER
MARTIN              7654 SALESMAN
ALLEN               7499 SALESMAN
TURNER              7844 SALESMAN
JAMES               7900 CLERK
SMITH               7369 CLERK
SCOTT               7788 ANALYST
ADAMS               7876 CLERK
MILLER              7934 CLERK
BALFORD             6235 CLERK
SCOTT               7788 ANALYST
JEWELL              7001 ANALYST
ALLEN               7499 SALESMAN
BRIGGS              7225 PAY CLERK
...
23 rows selected.
```



INTERSECT

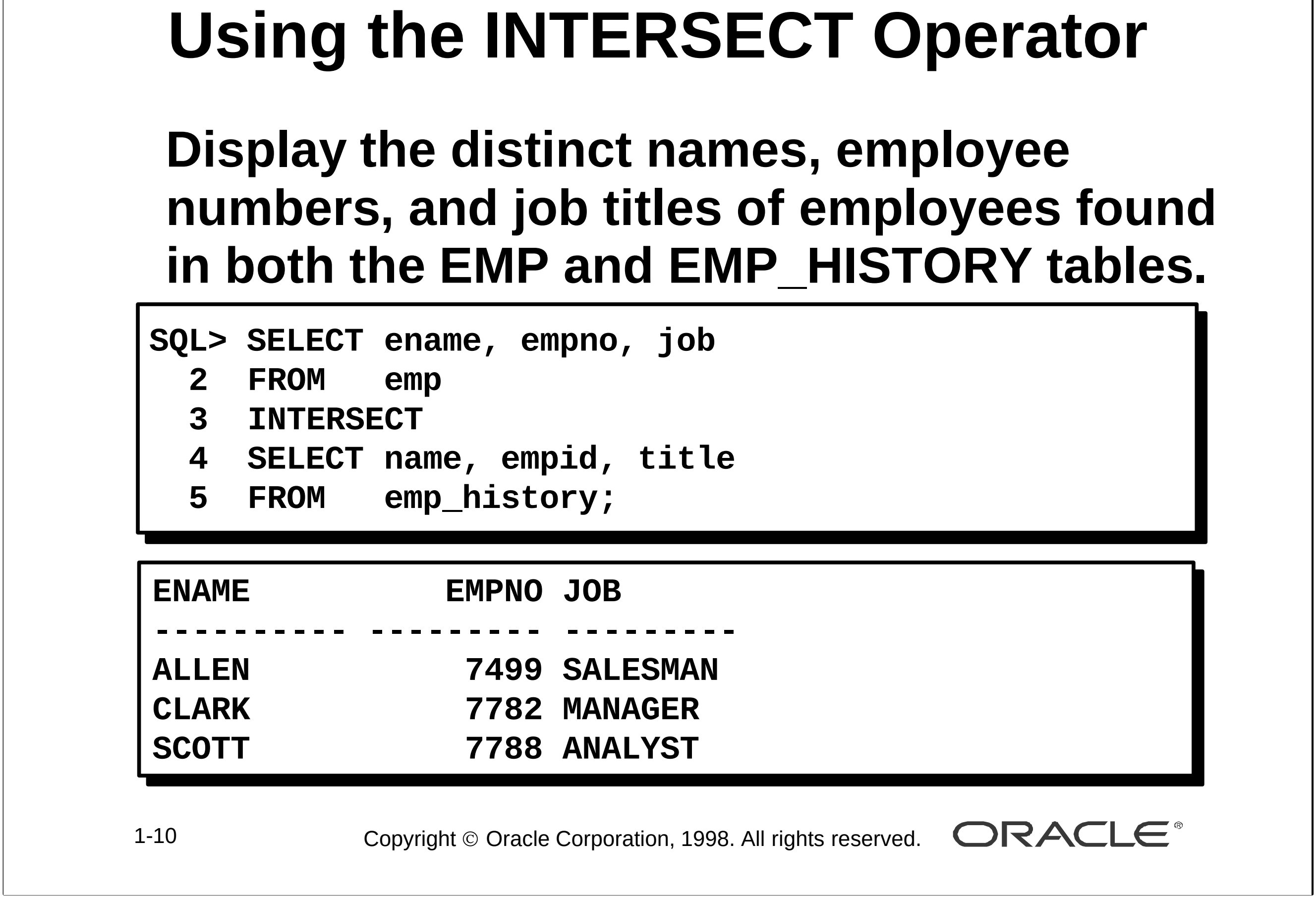
The INTERSECT Operator

Use the INTERSECT operator to return all common rows in both queries.

- The number of columns and the datatypes of those columns must be identical between the two select statements. The names of the columns need not be identical.
- Reversing the order of the INTERSECTed tables will not alter the result.
- INTERSECT, like UNION, ignores NULL columns.
- Queries that use INTERSECT in the where clause must have the same number and type of columns in their select list.

Class Management Note

Demo: *l1inters.sql*
Purpose: To illustrate the INTERSECT set operator.



Using the INTERSECT Operator

Display the distinct names, employee numbers, and job titles of employees found in both the EMP and EMP_HISTORY tables.

```
SQL> SELECT ename, empno, job
  2  FROM   emp
  3  INTERSECT
  4  SELECT name, empid, title
  5  FROM   emp_history;
```

```
ENAME          EMPNO JOB
---------- --------- ---------
ALLEN           7499 SALESMAN
CLARK           7782 MANAGER
SCOTT           7788 ANALYST
```

The INTERSECT Operator (continued)

In the example on this slide, the query returns only the records that have the same values in the selected columns in both tables.

What will be the results if we add the DEPTNO column to the EMP SELECT statement and add the DEPTID column to the EMP_HISTORY SELECT statement and run this query? The results may be different because of the introduction of another column whose values may or may not be duplicates.

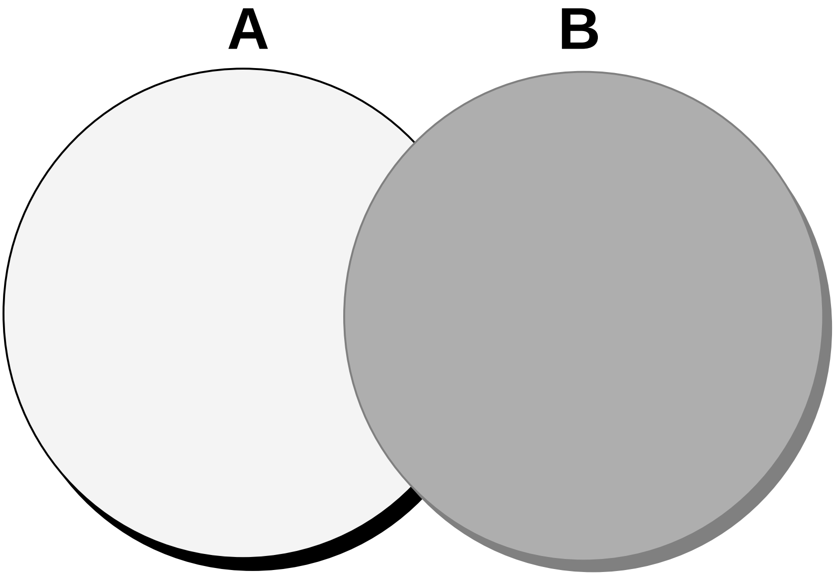
Example

```
SQL> SELECT ename, empno, job, deptno
  2  FROM   emp
  3  INTERSECT
  4  SELECT name, empid, title, deptid
  5  FROM   emp_history;
```

```
ENAME          EMPNO JOB          DEPTNO
---------- --------- --------- ---------
CLARK           7782 MANAGER          10
SCOTT           7788 ANALYST          20
```

The employee ALLEN is no longer part of the results because the EMP.DEPTNO value is different from the EMP_HISTORY.DEPTID value.

MINUS



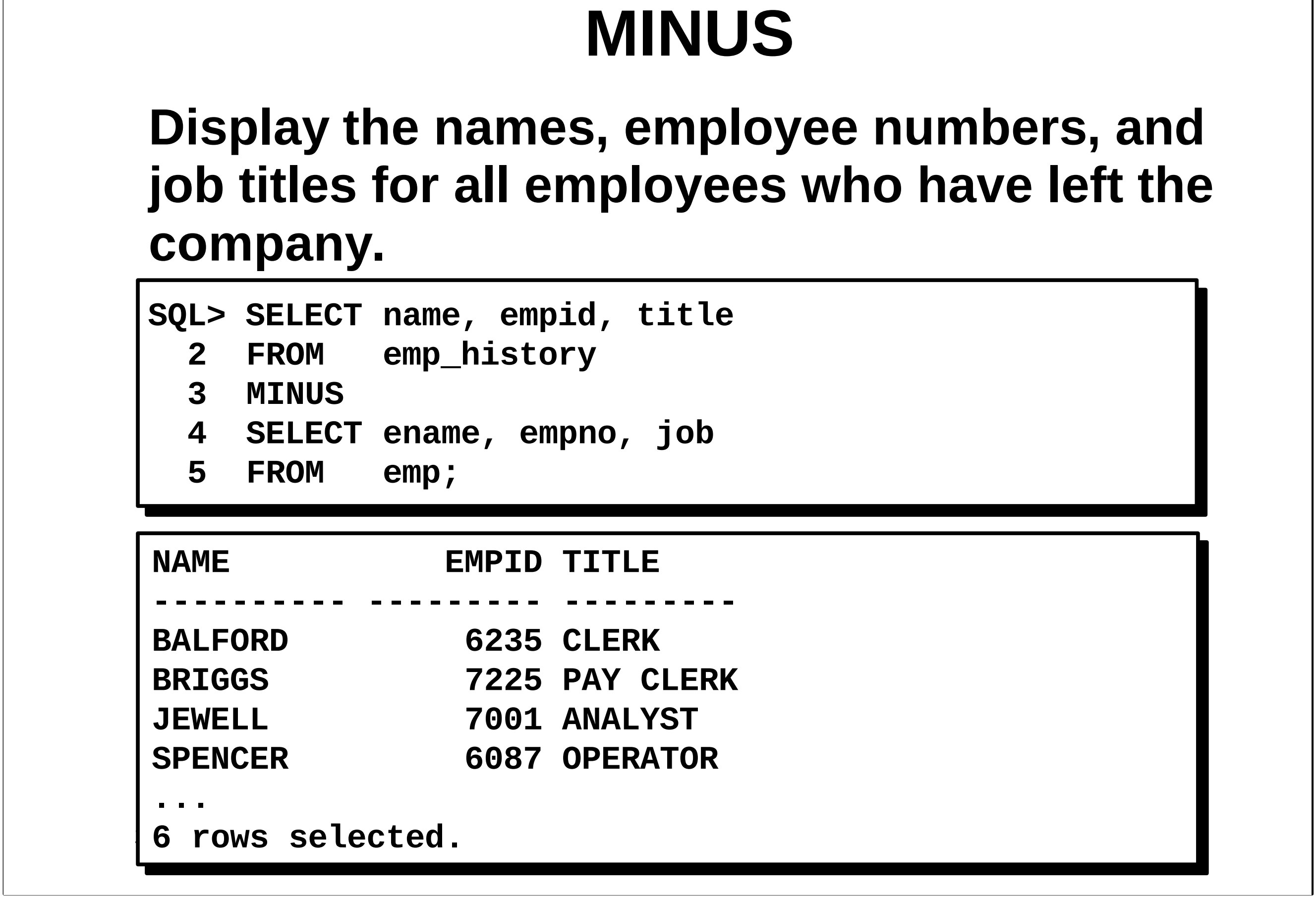
The MINUS Operator

Use the MINUS operator to return rows returned by the first query but not the second query (the first select statement MINUS the second select statement).

- The number of columns and the datatypes of those columns must be identical between the two select statements. The names of the columns need not be identical.
- All of the columns in the where clause must be in the select clause for the MINUS operator query to work.
- Queries that use MINUS in the where clause must have the same number and type of columns in their select list.

Class Management Note

Demo: *l1minus.sql*
Purpose: To illustrate the MINUS set operator.



MINUS

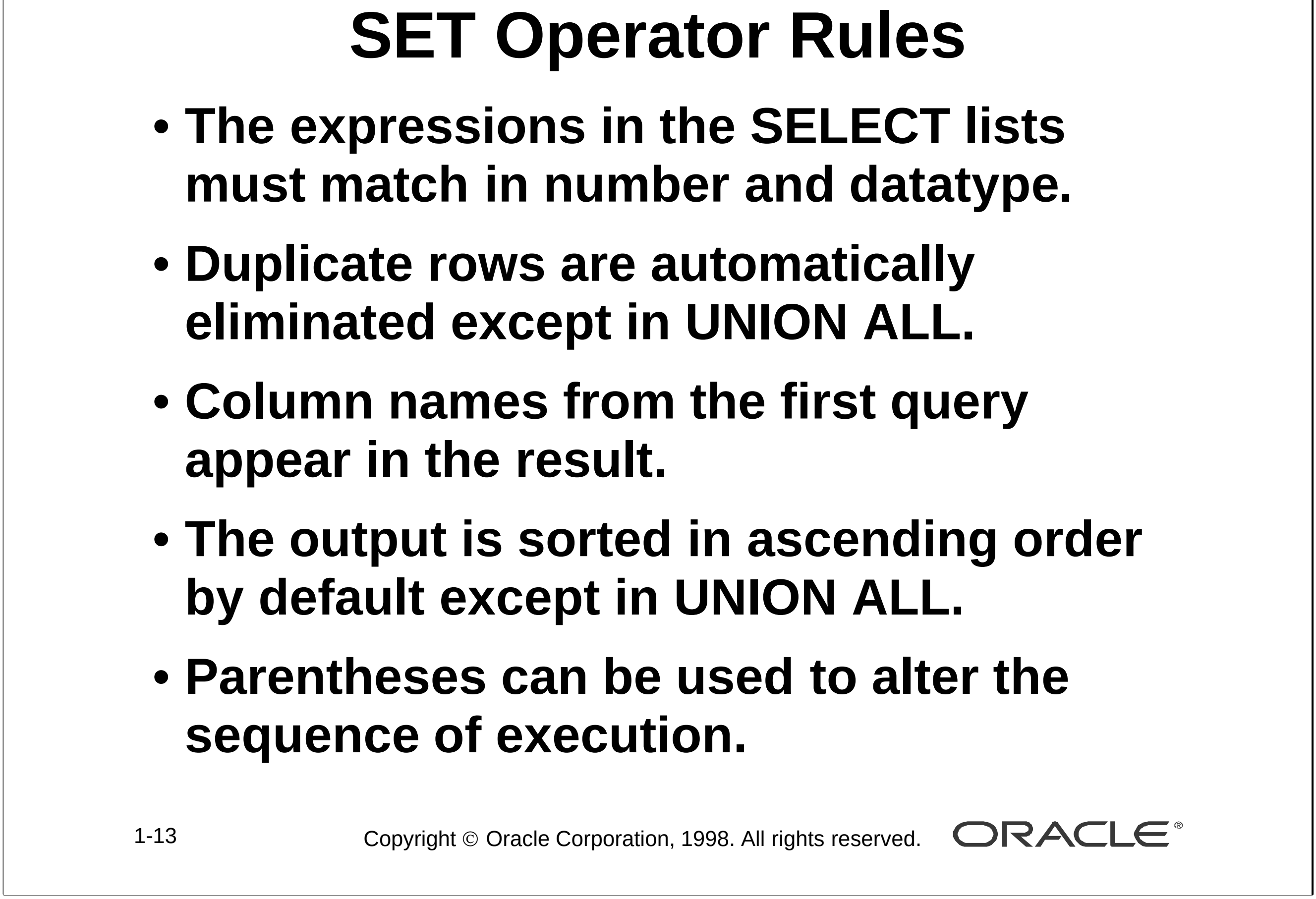
Display the names, employee numbers, and job titles for all employees who have left the company.

```
SQL> SELECT name, empid, title
  2  FROM   emp_history
  3  MINUS
  4  SELECT ename, empno, job
  5  FROM   emp;
```

```
NAME             EMPID TITLE
---------- --------- ---------
BALFORD           6235 CLERK
BRIGGS            7225 PAY CLERK
JEWELL            7001 ANALYST
SPENCER           6087 OPERATOR
...
6 rows selected.
```

The MINUS Operator (continued)

In the example on the slide, the employee names and job titles in the EMP table are subtracted from those in the EMP_HISTORY table. The result set displays the employees remaining after the subtraction; they are rows that exist in the EMP_HISTORY table that do not exist in the EMP table.



SET Operator Rules

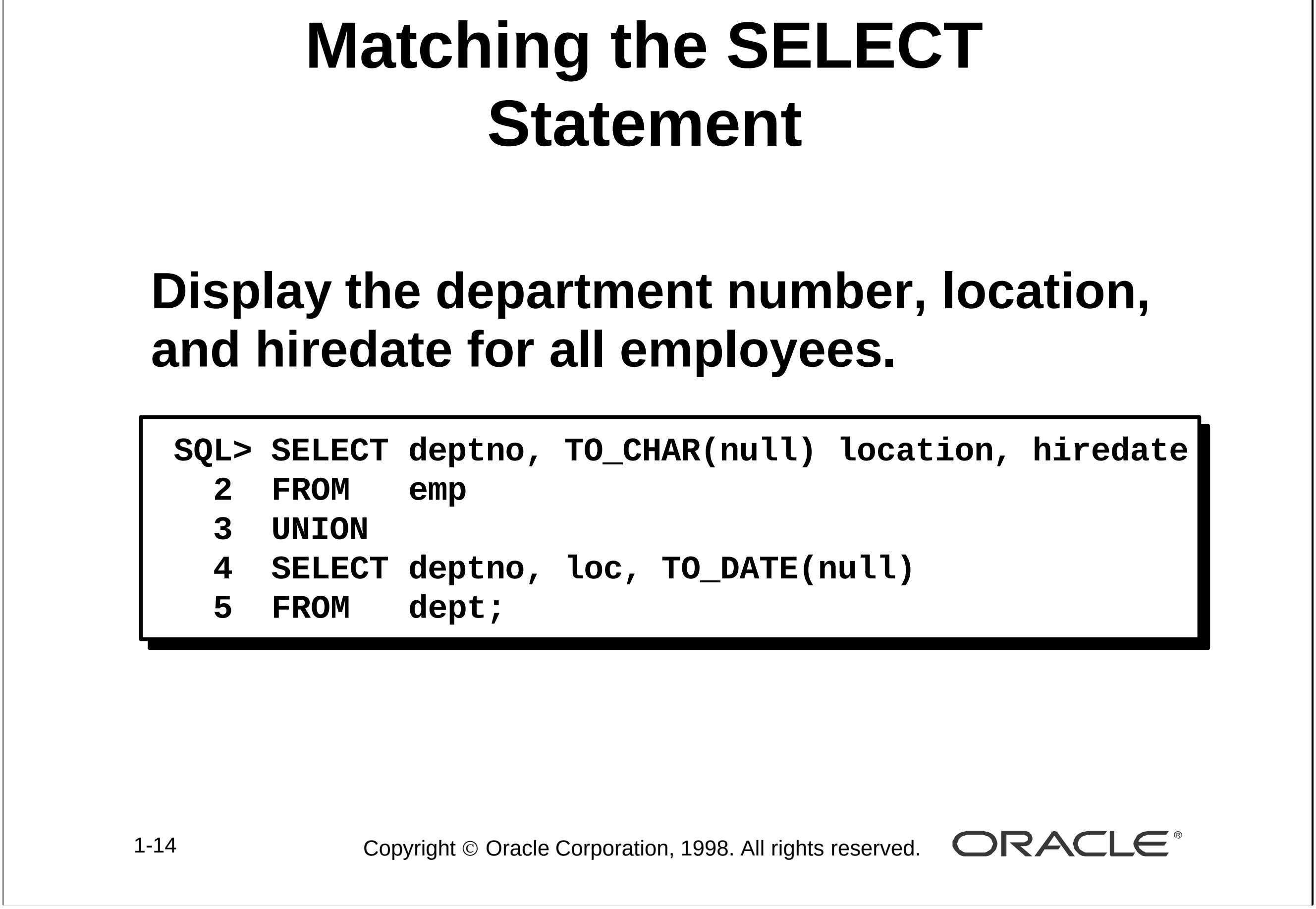
- The expressions in the SELECT lists must match in number and datatype.
- Duplicate rows are automatically eliminated except in UNION ALL.
- Column names from the first query appear in the result.
- The output is sorted in ascending order by default except in UNION ALL.
- Parentheses can be used to alter the sequence of execution.

ORACLE®

SET Operator Rules

- If both queries select values of the CHAR datatype, the returned values have a CHAR datatype.
- If either or both of the queries select values of datatype VARCHAR2, the returned values have a datatype of VARCHAR2.
- The ORDER BY clause:
  - Can appear only at the very end of the statement
  - Will accept the column name, an alias, or the positional notation
- The column name or alias if used in an ORDER BY clause must be from the first SELECT list.
- Set operators can be used in subqueries.
- The SELECT statements are executed left (top) to right (bottom).
- You can alter the operator precedence by using parentheses.
- Queries that use UNION, INTERSECT, and MINUS set operators in their WHERE clause must have the same number and type of columns in their SELECT list. For example:

```
SQL> SELECT ename, deptno
  2  FROM   emp
  3  WHERE  (ename, deptno) IN (SELECT ename, deptno
  4                              FROM   emp)
  5                             (SELECT name, deptid
  6                              FROM   emp_history);
```



Matching the SELECT Statement

Display the department number, location, and hiredate for all employees.

```
SQL> SELECT deptno, TO_CHAR(null) location, hiredate
  2  FROM   emp
  3  UNION
  4  SELECT deptno, loc, TO_DATE(null)
  5  FROM   dept;
```

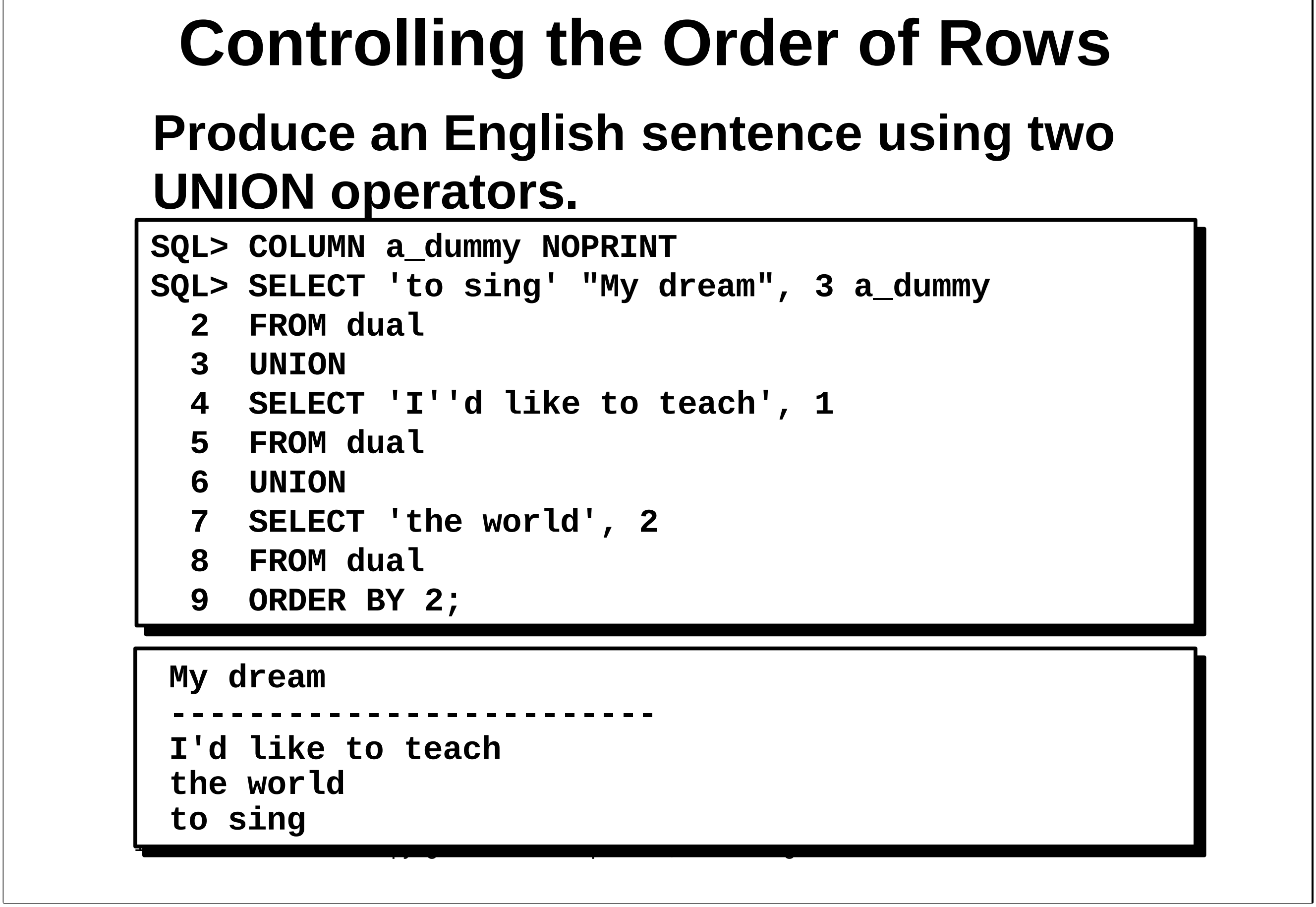
Matching the SELECT Lists

As the expressions in the SELECT lists of compound queries must match in number and datatype, you can use dummy columns and the datatype conversion functions to comply with this rule. In the slide above the name *location* is given as the dummy column heading.

```
   DEPTNO LOCATION      HIREDATE
--------- ------------- ---------
       10 NEW YORK
       10               09-JUN-81
       10               17-NOV-81
       10               23-JAN-82
       10
       20 DALLAS
       20               17-DEC-80
...
       30               03-DEC-81
       40 BOSTON
19 rows selected.
```

Class Management Note

Demo: *l1union3.sql* uses conversion functions. dummy.sql uses dummy columns (run, add ORDER BY 2, 1 and rerun, remove the REM commands, and rerun).



Controlling the Order of Rows

Produce an English sentence using two UNION operators.

```
SQL> COLUMN a_dummy NOPRINT
SQL> SELECT 'to sing' "My dream", 3 a_dummy
  2  FROM dual
  3  UNION
  4  SELECT 'I''d like to teach', 1
  5  FROM dual
  6  UNION
  7  SELECT 'the world', 2
  8  FROM dual
  9  ORDER BY 2;
```

```
 My dream
 -------------------------
 I'd like to teach
 the world
 to sing
```

Controlling the Order of Rows

The output is sorted in ascending order by default. You can use the ORDER BY clause to change this around.

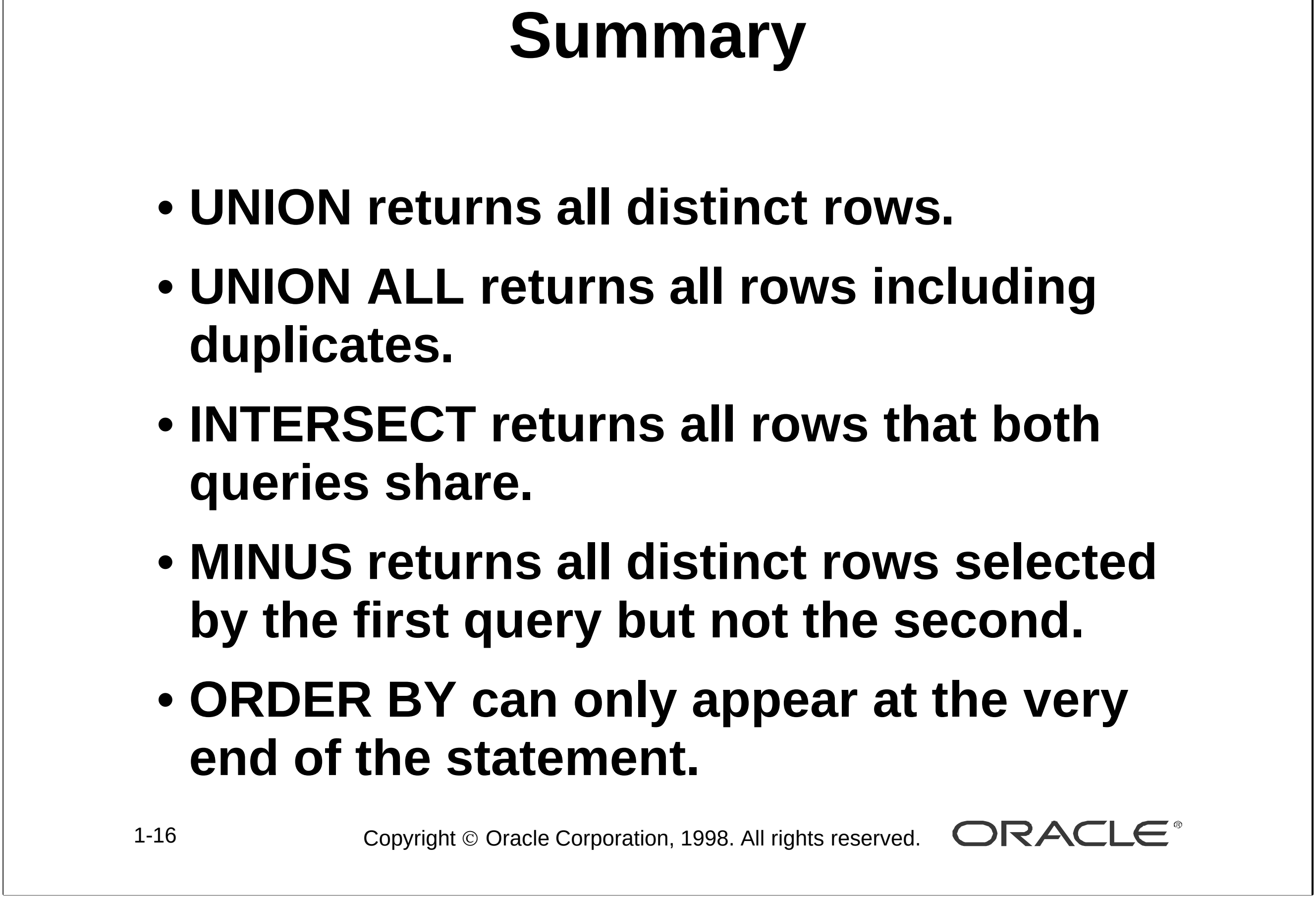
Using ORDER BY to Order Rows

ORDER BY can be used only once in a compound query. If used, the ORDER BY clause must be placed at the end of the query. The ORDER BY clause accepts the column name, an alias, or the positional notation.

Note: The ORDER BY clause, when used in a compound query with the UNION (used more than once) set operator, can only use positions, rather than explicit expressions.

Class Management Note

Demo: *l1setord.sql*
Purpose: To illustrate the ordering of rows with a set operator.



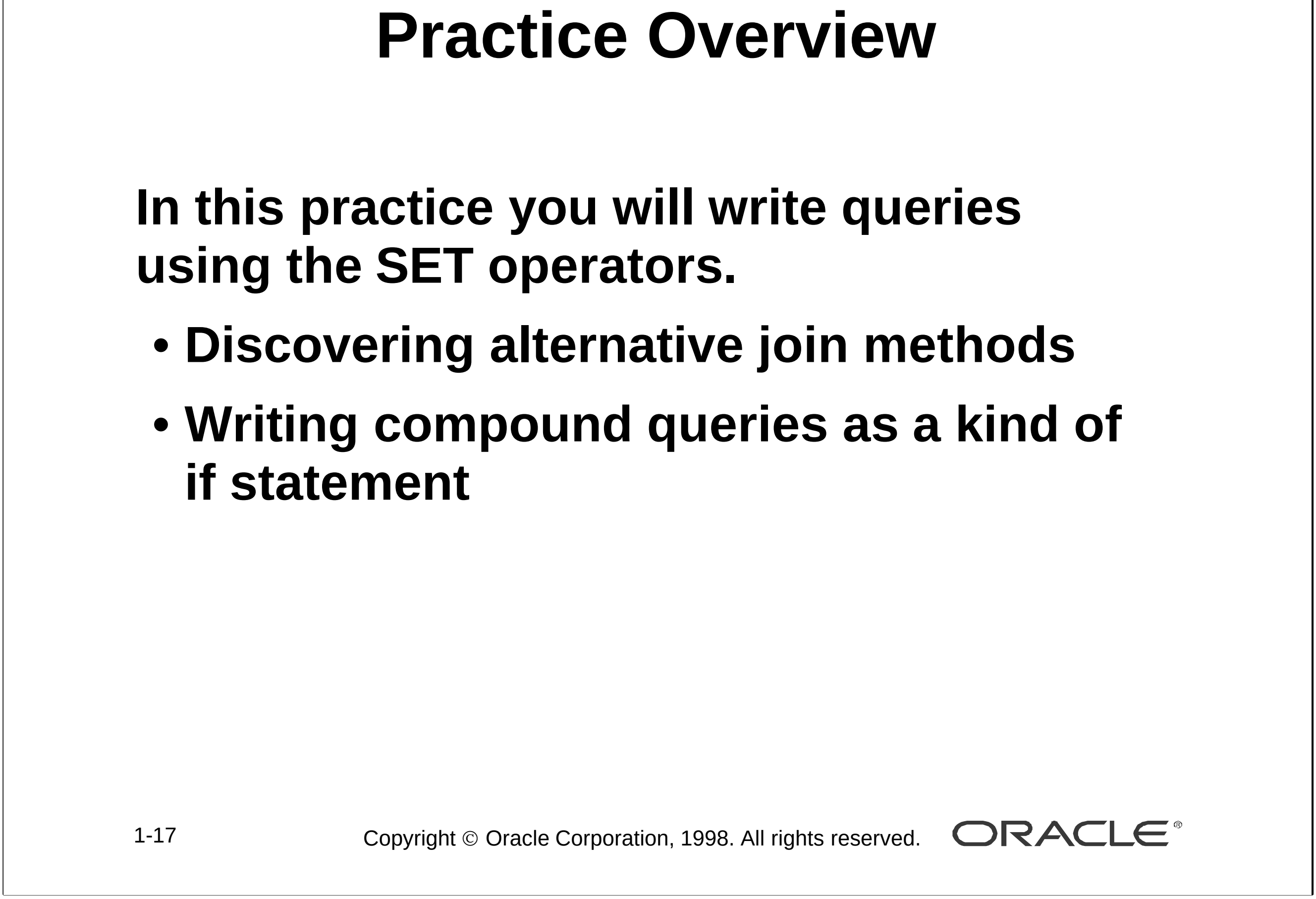
Summary

- UNION returns all distinct rows.
- UNION ALL returns all rows including duplicates.
- INTERSECT returns all rows that both queries share.
- MINUS returns all distinct rows selected by the first query but not the second.
- ORDER BY can only appear at the very end of the statement.

Summary

Remember to use the ORDER BY clause only at the very end of the compound statement.

Make sure that the corresponding expressions in the SELECT lists match in number and datatype.



Practice Overview

In this practice you will write queries using the SET operators.

- **Discovering alternative join methods**
- **Writing compound queries as a kind of if statement**

Note: To create the table EMP_HISTORY, run the script *emphis.sql*.

Practice 1

1. Display the department that has no employees.

```
   DEPTNO DNAME
--------- --------------
       40 OPERATIONS
```

2. Find the job that was filled in the last half of 1981 and the same job that was filled during the same period in 1982.

```
JOB
---------
ANALYST
```

3. Write a compound query to produce a list of products showing discount percentages, product id, and old and new actual price. Products under $10 are reduced by 10%, products between $10 and $30 are reduced by 15%, products over $30 are reduced by 20%, and products over $40 are not reduced at all.

```
DISCOUNT    PRODID  STDPRICE  ACTPRICE
-------- --------- --------- ---------
10% off     100870       2.4      2.16
10% off     100870       2.8      2.52
10% off     100871       4.8      4.32
10% off     100871       5.6      5.04
10% off     102130       3.4      3.06
10% off     200376       2.4      2.16
10% off     200380         4       3.6
15% off     100860        30      25.5
15% off     101860        24      20.4
15% off     101863      12.5    10.625
20% off     100860        32      25.6
20% off     100860        35        28
20% off     100861        39      31.2
no disc     100861        42        42
no disc     100861        45        45
no disc     100890        54        54
no disc     100890        58        58
```

Practice 1 (continued)

4. Produce a list of jobs for departments 10, 30, and 20 in that order. Display job and department number.

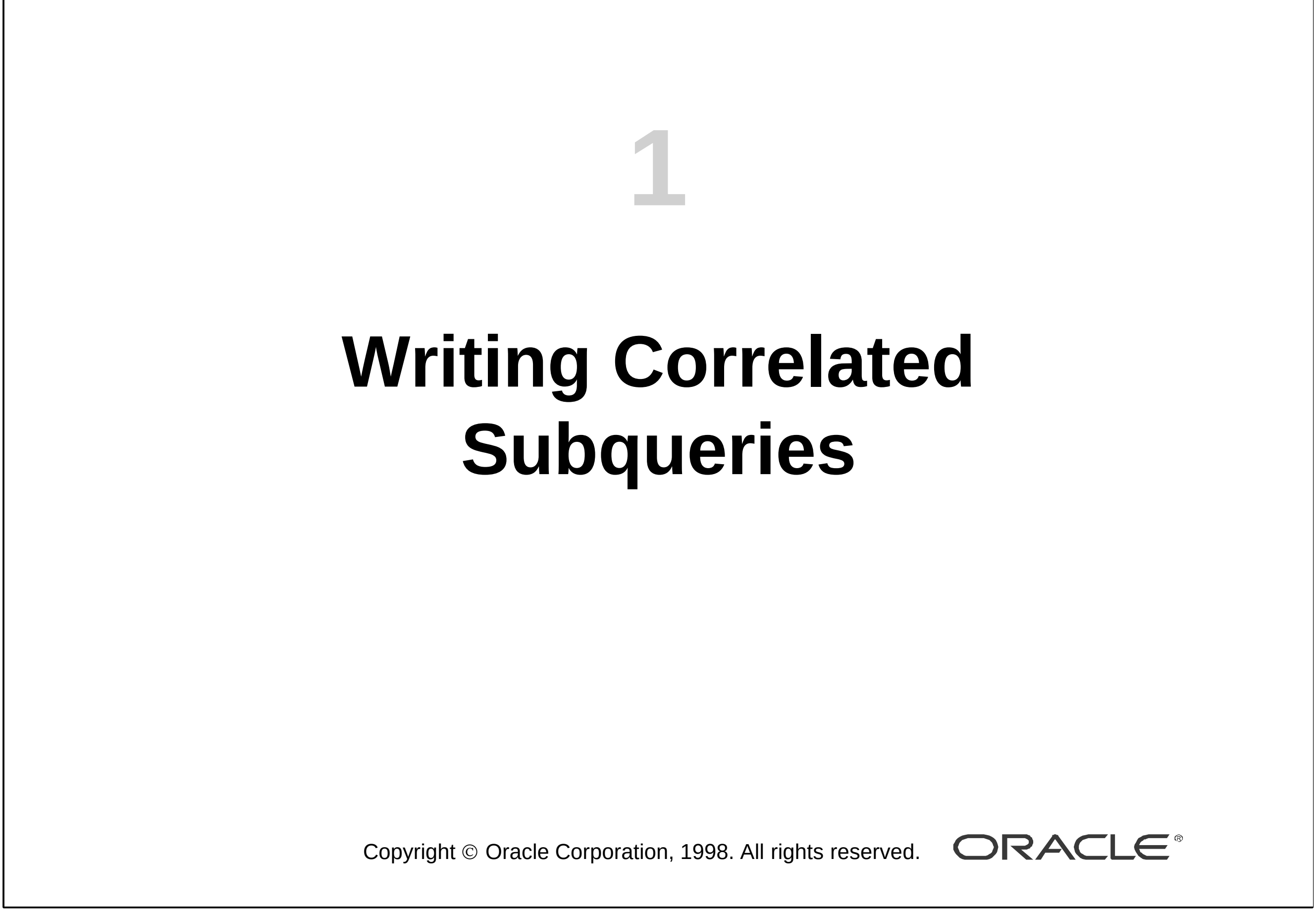
```
JOB          DEPTNO
--------- --------
CLERK            10
MANAGER          10
PRESIDENT        10
CLERK            30
MANAGER          30
SALESMAN         30
ANALYST          20
CLERK            20
MANAGER          20
```

5. List the department number for departments without the job title ANALYST .

```
DEPTNO
------
    10
    30
    40
```

6. List all job titles in department 10 and 20 that do not occur in both departments.

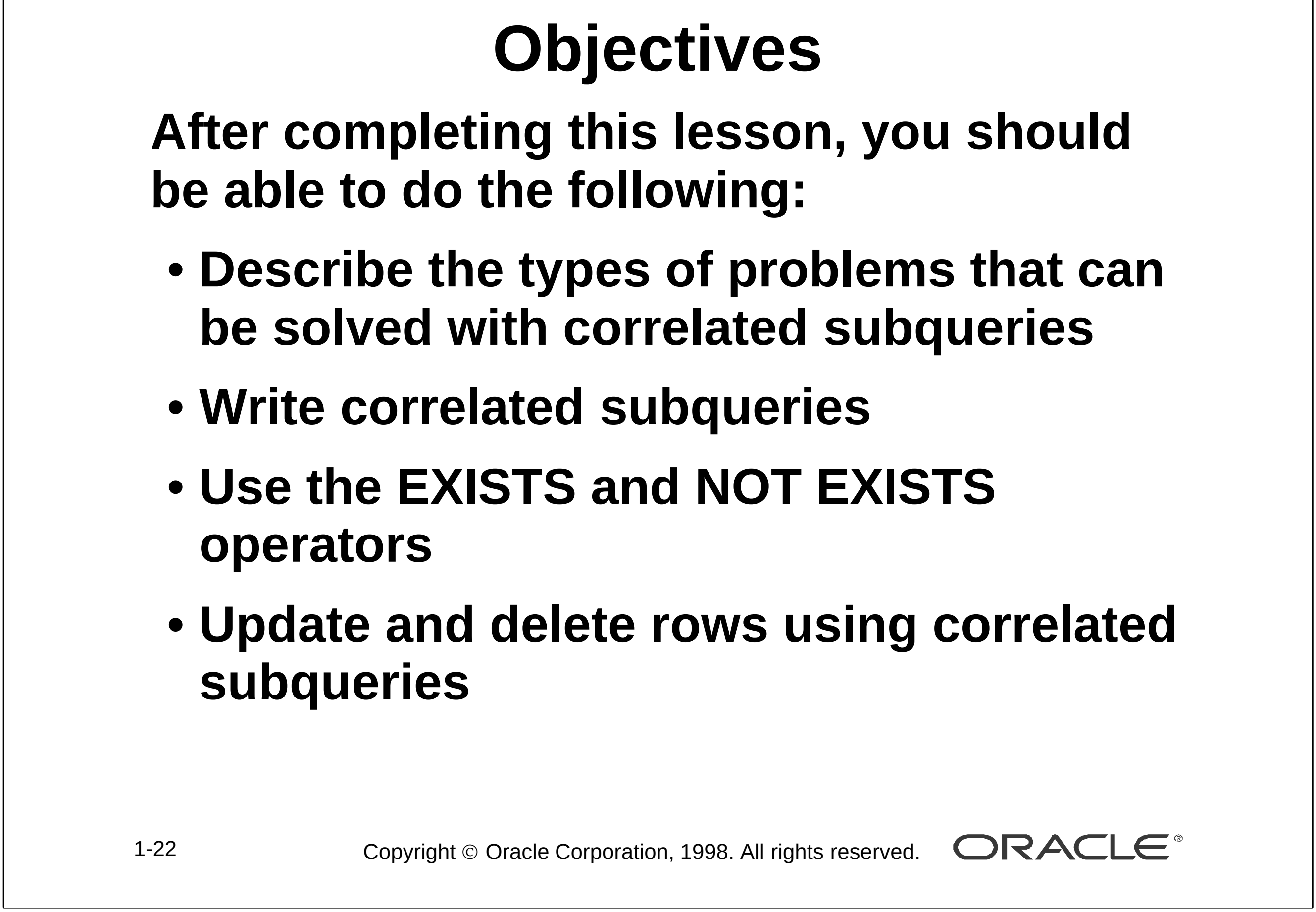
```
JOB
--------
ANALYST
PRESIDENT
```



1

Writing Correlated Subqueries

Schedule:	Timing	Topic
	35 minutes	Lecture
	25 minutes	Practice
	60 minutes	Total



Objectives

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

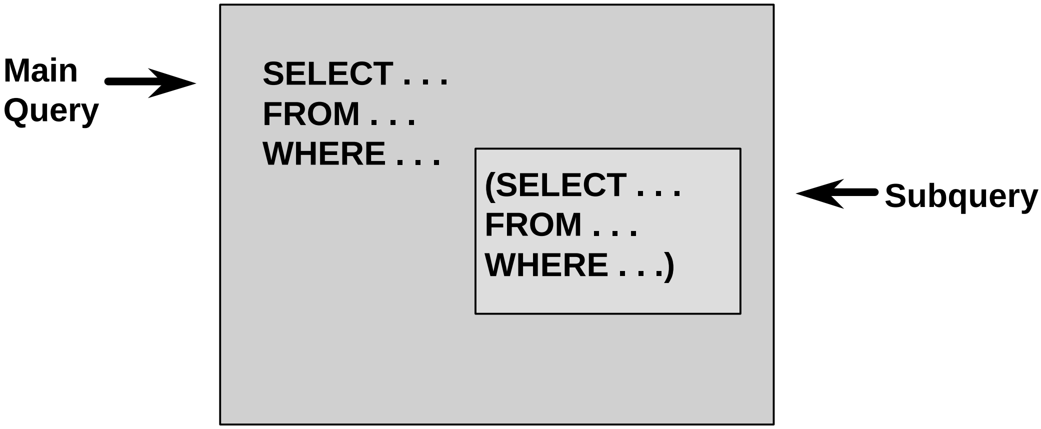
- **Describe the types of problems that can be solved with correlated subqueries**
- **Write correlated subqueries**
- **Use the EXISTS and NOT EXISTS operators**
- **Update and delete rows using correlated subqueries**

Lesson Aim

In this lesson you will learn how to solve problems using correlated subqueries.

What Is a Subquery?

A subquery is a SELECT statement embedded in a clause of another SQL statement.



What Is a Subquery?

A *subquer*y is a SELECT statement that is embedded in the clause of another SQL statement.

The subquery (inner query) returns a value that is used by the main query (outer 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.

Subqueries can be used for the following purposes:

- To provide values for conditions in WHERE, HAVING, and START WITH clauses of SELECT, UPDATE, and DELETE statements
- To define the set of rows to be inserted into the target table of an INSERT or CREATE TABLE statement
- To define the set of rows to be included in a view or snapshot in a CREATE VIEW or CREATE SNAPSHOT statement
- To define one or more values to be assigned to existing rows in an UPDATE statement
- To define a table to be operated on by a containing query. You do this by placing the subquery in the FROM clause. This can be done in INSERT, UPDATE, and DELETE statements as well.

Note: A subquery is evaluated once for the entire parent statement.

Subqueries

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

- **The subquery (inner query) executes once before the main query.**
- **The result of the subquery is used by the main query (outer query).**

Subqueries

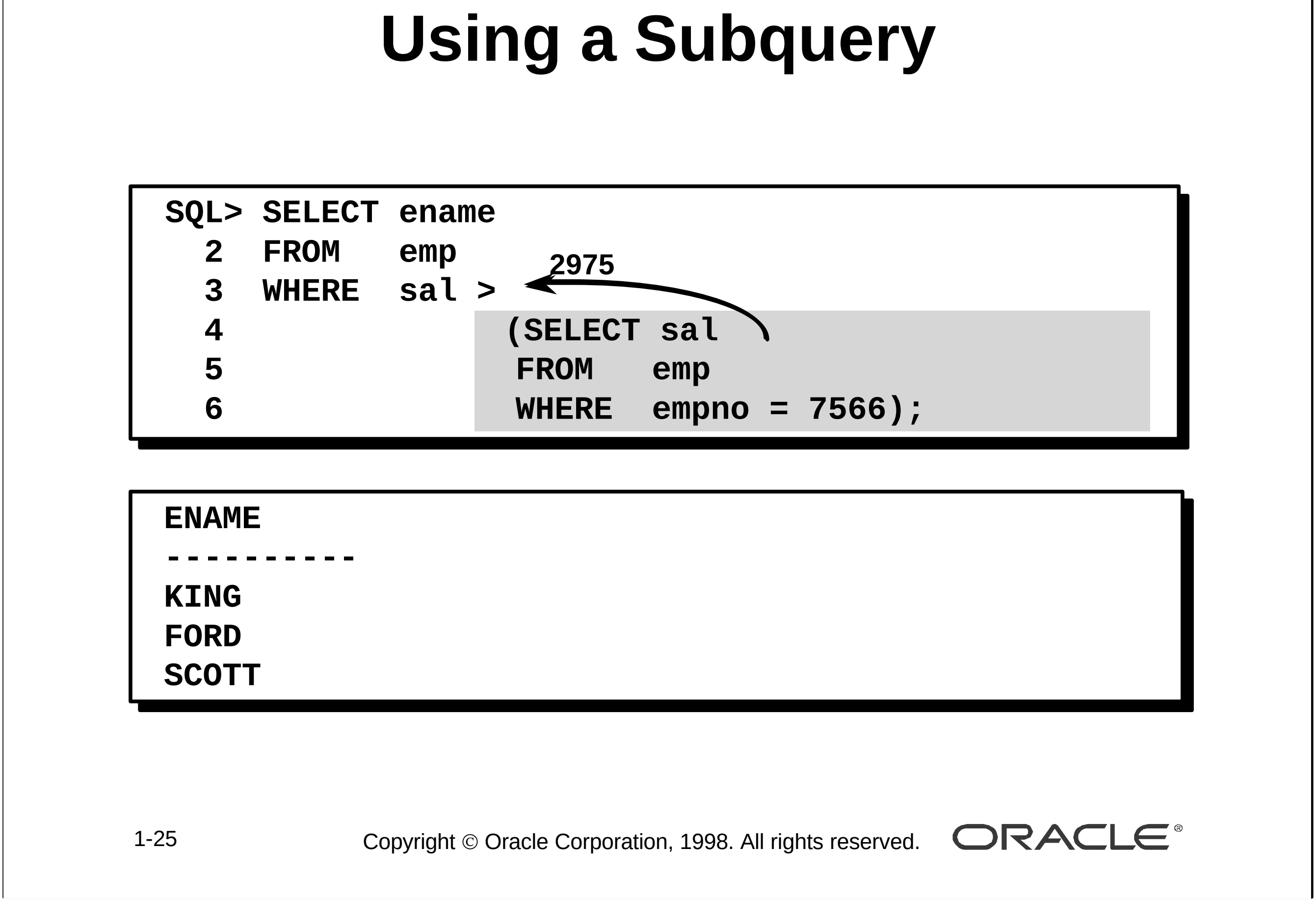
You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

In the syntax:

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

Note: Comparison operators 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.



Using a Subquery

```
SQL> SELECT ename
  2  FROM   emp
  3  WHERE  sal >
  4                (SELECT sal
  5                 FROM   emp
  6                 WHERE  empno = 7566);
```

```
ENAME
----------
KING
FORD
SCOTT
```

Using a Subquery

In the slide, the inner query returns the salary of employee 7566. The outer query takes the result of the inner query and uses this result to display the names of all the employees who earn more than this amount.

Subqueries are very useful for writing SQL statements that need values based on an unknown conditional value.

Examples

Create a duplicate of the DEPT table.

```
SQL> CREATE TABLE dept_copy(deptno, dname, loc)
  2  AS SELECT deptno, dname, loc
  3  FROM dept;
```

Display all employees who make less than the average salary in the company.

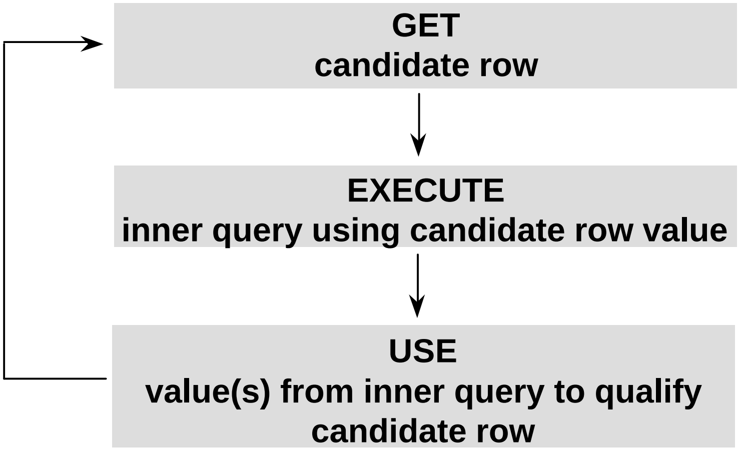
```
SQL> SELECT ename, job, sal
  2  FROM   emp
  3  WHERE  sal < (SELECT AVG(sal)
  4                FROM   emp);
```

Class Management Note

Execute the subquery (inner query) on its own first to show the value that the subquery returns. Then execute the outer query using the result returned by the inner query. Finally, execute the entire query (containing the subquery) and show that the result is the same.

Correlated Subqueries

Used to affect row-by-row processing, each subquery is executed once for every row of the outer query.



GET
candidate row

EXECUTE
inner query using candidate row value

USE
value(s) from inner query to qualify candidate row

Correlated Subqueries

A correlated subquery is a nested subquery that is evaluated once for each row processed by the main query, and that on execution uses a value from a column in the outer query.

Nested Subqueries Versus Correlated Subqueries

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

Correlated Subquery Execution

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

Although this discussion focuses on correlated subqueries in SELECT statements, it also applies to correlated UPDATE and DELETE statements.

Correlated Subqueries

```
SELECT outer1, outer2, ...
FROM   table1 alias1
WHERE  outer1 operator
                     (SELECT inner1
                     FROM    table2 alias2
                     WHERE   alias1.outer2 =
                             alias2.inner1);
```

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

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. In other words, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

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

Using Correlated Subqueries

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

```
SQL> SELECT empno, sal, deptno
  2  FROM   emp outer
  3  WHERE  sal > (SELECT AVG(sal)
  4                 FROM   emp inner
  5                 WHERE  outer.deptno = inner.deptno);
```

Each time the outer query is processed the inner query is evaluated.

```
   EMPNO       SAL    DEPTNO
-------- --------- ---------
    7839      5000        10
    7698      2850        30
    7566      2975        20
 ...
 6 rows selected.
```

Using Correlated Subqueries

In the example, we determine which employees earn more than the average salaries for their departments. In this case, the correlated subquery specifically computes the average salary for each department.

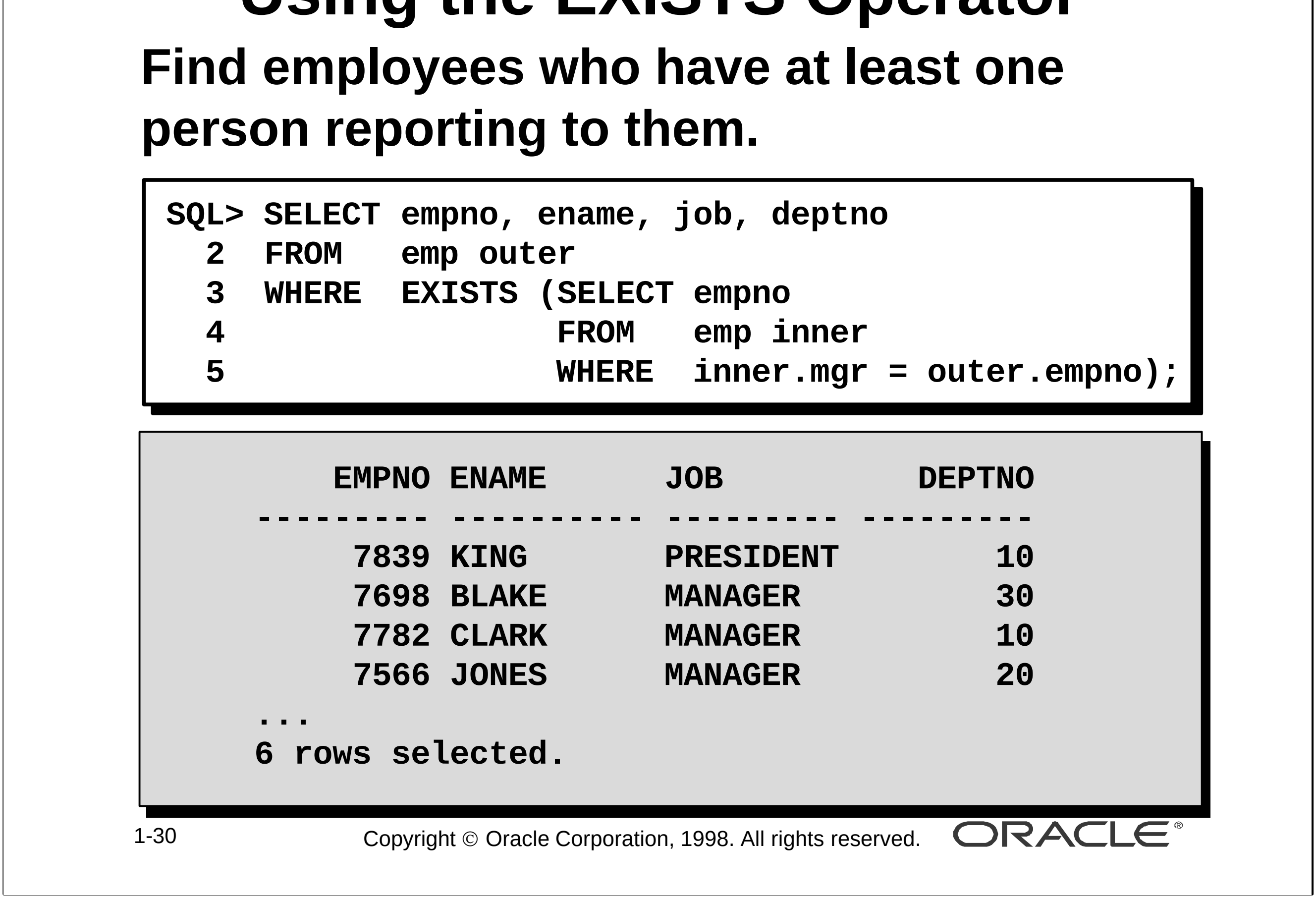
Because the outer query and inner query both use the EMP table in the FROM clause, an alias is given to EMP in each separate SELECT statement for clarity. Not only does the alias make the entire SELECT statement more readable, without the alias the query would not work properly because the inner statement would not be able to distinguish the inner table column from the outer table column.

Using the EXISTS Operator

- 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.

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. It tests whether a value is there. If the value exists, it returns TRUE; if the value does not exist, it returns FALSE. Similarly, NOT EXISTS ensures that a value is not there.



Using the EXISTS Operator

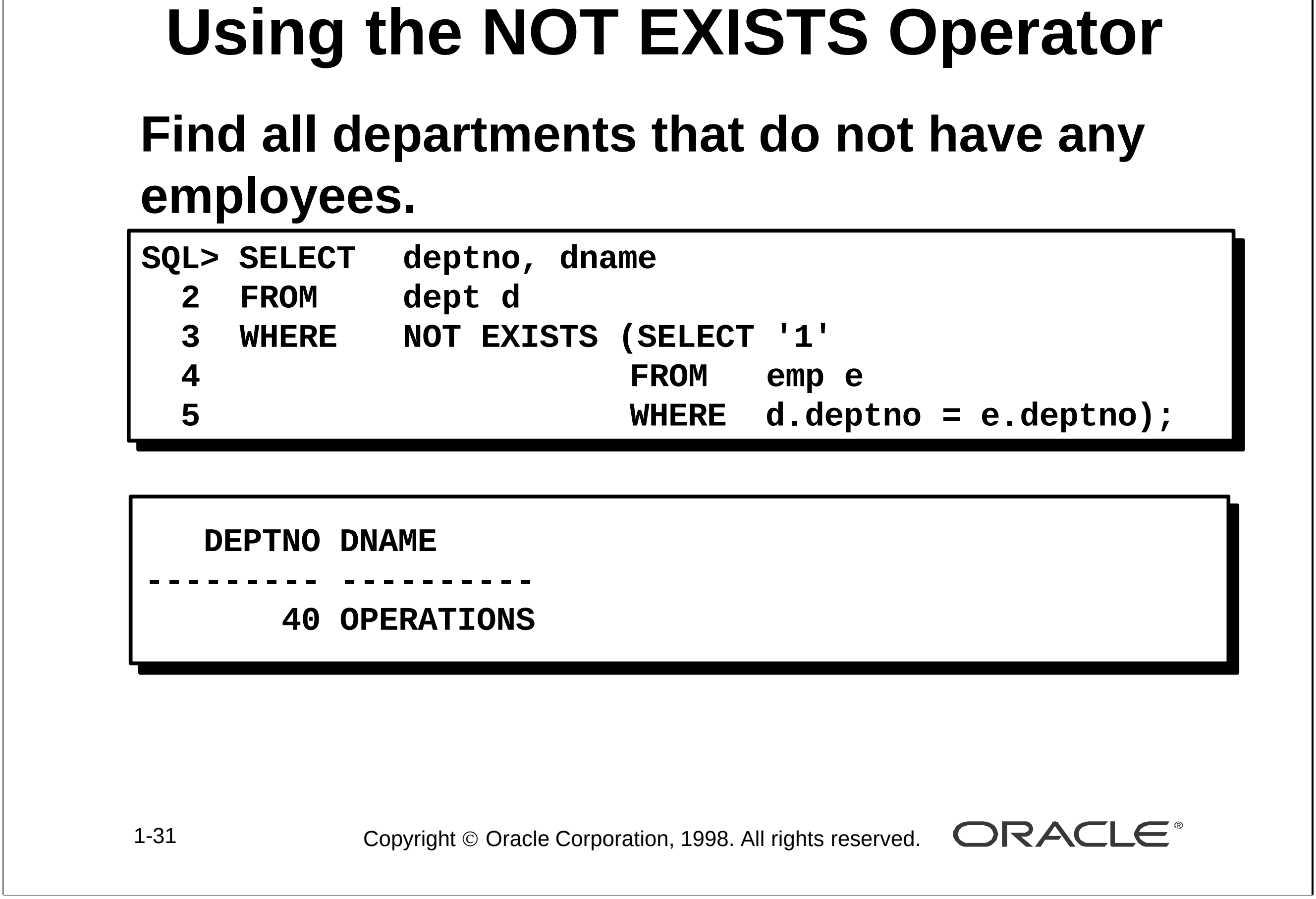
Find employees who have at least one person reporting to them.

```
SQL> SELECT empno, ename, job, deptno
  2  FROM   emp outer
  3  WHERE  EXISTS (SELECT empno
  4                 FROM   emp inner
  5                 WHERE  inner.mgr = outer.empno);
```

```
    EMPNO ENAME      JOB          DEPTNO
--------- ---------- --------- ---------
     7839 KING       PRESIDENT        10
     7698 BLAKE      MANAGER          30
     7782 CLARK      MANAGER          10
     7566 JONES      MANAGER          20
...
6 rows selected.
```

Using the EXISTS Operator

The EXISTS operator ensures that the search in the inner query will not continue when at least one match is found for the manager and employee numbers.



Using the NOT EXISTS Operator

Find all departments that do not have any employees.

```
SQL> SELECT   deptno, dname
  2  FROM     dept d
  3  WHERE    NOT EXISTS (SELECT '1'
  4                        FROM   emp e
  5                        WHERE  d.deptno = e.deptno);
```

```
   DEPTNO DNAME
--------- ----------
       40 OPERATIONS
```

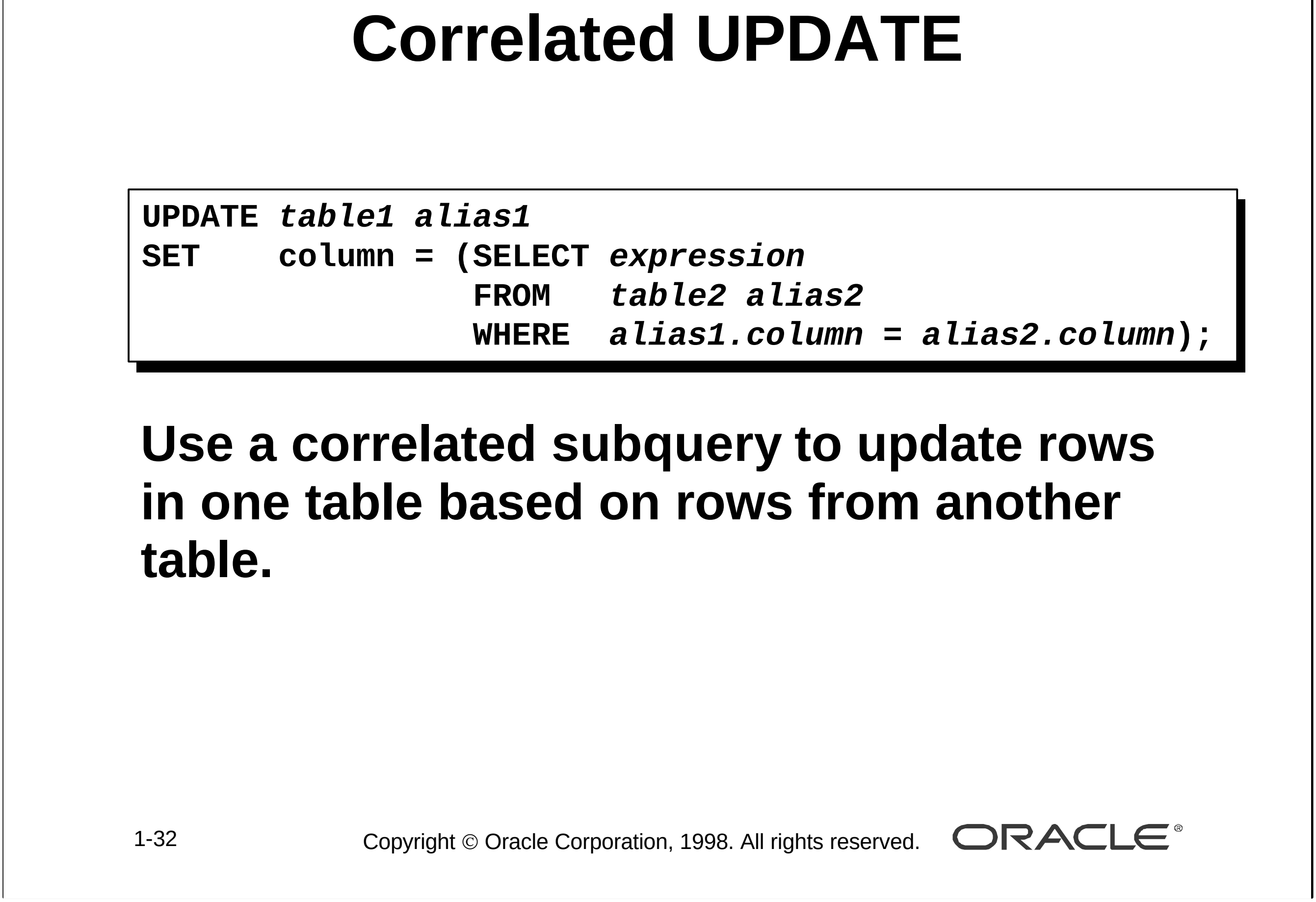
Using the EXISTS Operator (continued)

Note that the inner SELECT does not need to return a specific value, so a literal can be selected. From a performance standpoint, it is faster to select a constant than a column.

Alternative Solution

```
SQL> SELECT deptno, dname
  2  FROM   dept
  3  WHERE  deptno NOT IN (SELECT deptno
  4                        FROM   emp);
```

As shown in the previous example, a NOT IN construct can be used as an alternative for a NOT EXISTS. However, caution should be used. NOT IN evaluates to FALSE if any member of the set is NULL. In that case your query will not return any rows.



Correlated UPDATE

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

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

Using Correlated Subqueries

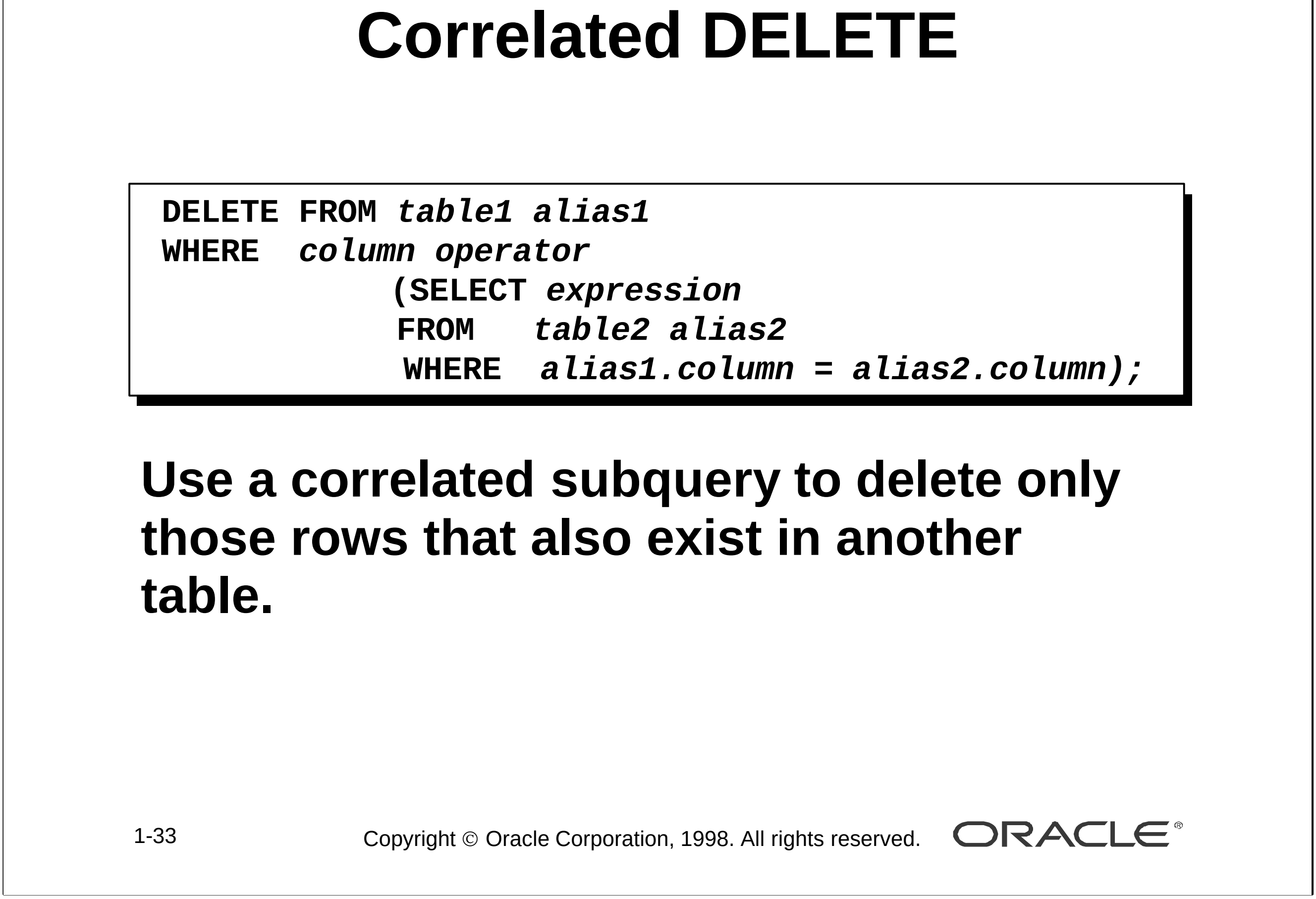
In the case of the UPDATE statement, you can use a correlated subquery to update rows in one table based on rows from another table.

Example

Denormalize the EMP table by adding a column to store the department name. Then populate the table using a correlated update.

```
SQL> ALTER TABLE emp
  2  ADD(dname VARCHAR2(14));
```

```
SQL> UPDATE emp e
  2  SET    dname = (SELECT dname
  3                  FROM   dept d
  4                  WHERE  e.deptno = d.deptno);
```



Correlated DELETE

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

Use a correlated subquery to delete only those rows that also exist in another table.

Using Correlated Subqueries (continued)

In the case of a DELETE statement, you can use a correlated subquery to delete only those rows that also exist in another table.

Example

Write a query to find all records with duplicate employee numbers and delete the duplicate rows from the EMP table.

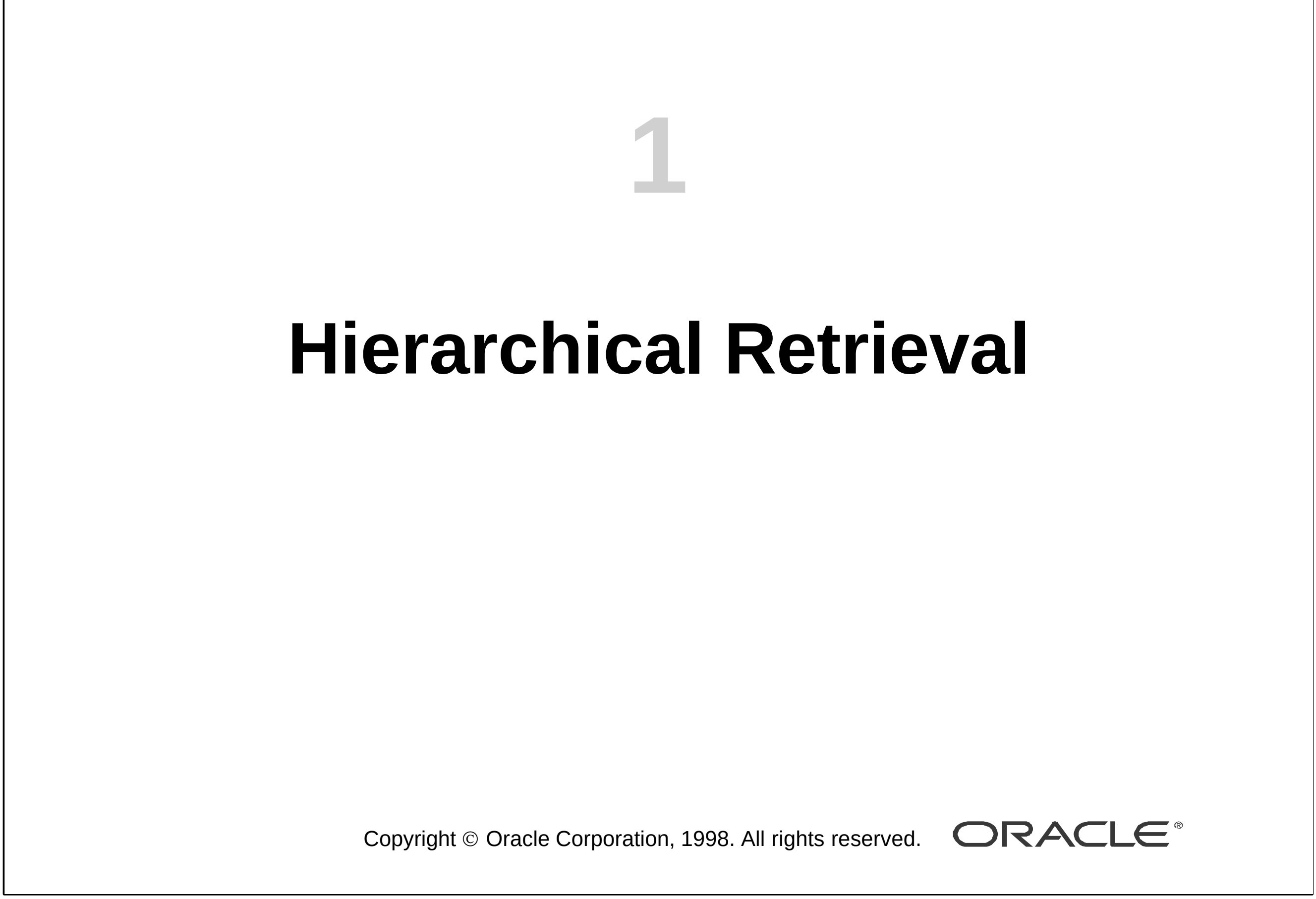
```
SQL> SELECT ename
  2  FROM   emp outer
  3  WHERE  ROWID > (SELECT MIN(ROWID)
  4                  FROM   emp inner
  5                  WHERE  outer.empno = inner.empno)
  6  FOR UPDATE;
```

```
SQL> DELETE FROM emp outer
  2  WHERE  ROWID > (SELECT MIN(ROWID)
  3                  FROM   emp inner
  4                  WHERE  outer.empno = inner.empno);
```

Note: The FOR UPDATE clause locks the rows selected by the query. Other users cannot lock or update the selected rows until you end your transaction.

Summary

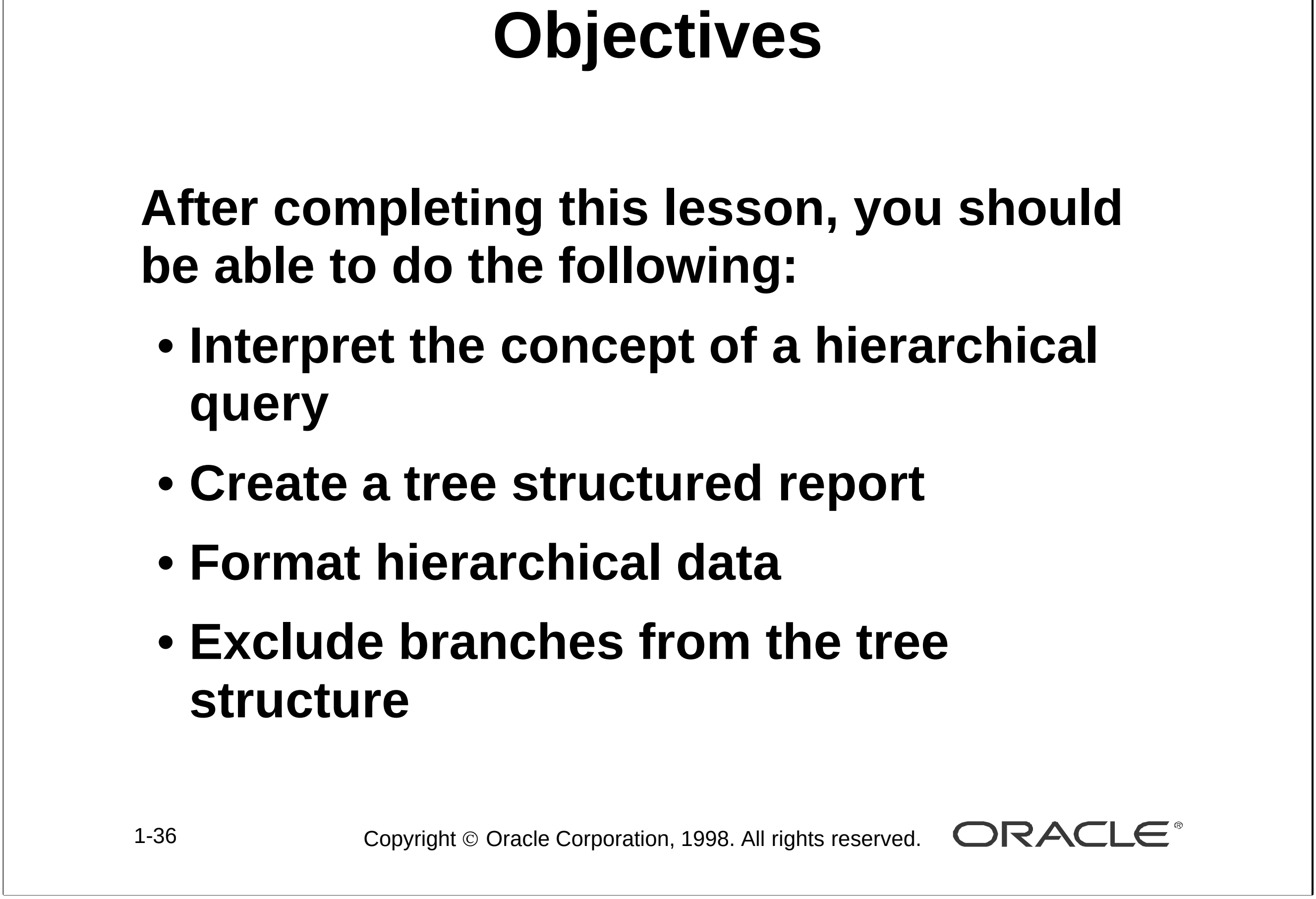
- Correlated subqueries are useful whenever a subquery must return a different result for each candidate row.
- The EXISTS operator is a Boolean operator, testing the presence of a value.
- Correlated subqueries can be used with SELECT, UPDATE, and DELETE statements.



1

Hierarchical Retrieval

Schedule:	Timing	Topic
	30 minutes	Lecture
	20 minutes	Practice
	50 minutes	Total



Objectives

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

- Interpret the concept of a hierarchical query
- Create a tree structured report
- Format hierarchical data
- Exclude branches from the tree structure

Lesson Aim

In this lesson, you will learn how to use hierarchical queries to create tree structured reports.

When a Hierarchical Query Is Possible

```
    EMPNO ENAME      JOB             MGR
--------- ---------- --------- ---------
     7839 KING       PRESIDENT
     7698 BLAKE      MANAGER        7839
     7782 CLARK      MANAGER        7839
     7566 JONES      MANAGER        7839
     7654 MARTIN     SALESMAN       7698
     7499 ALLEN      SALESMAN       7698
     7844 TURNER     SALESMAN       7698
     7900 JAMES      CLERK          7698
     7521 WARD       SALESMAN       7698
     7902 FORD       ANALYST        7566
     7369 SMITH      CLERK          7902
     7788 SCOTT      ANALYST        7566
     7876 ADAMS      CLERK          7788
     7934 MILLER     CLERK          7782
```

Overview

Hierarchical queries are a facility that enable you to retrieve data based on a natural hierarchical relationship between rows in a table.

A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, there is a process called *tree walking* that enables the hierarchy to be constructed. A hierarchical query is a method of reporting, in order, the branches of a tree.

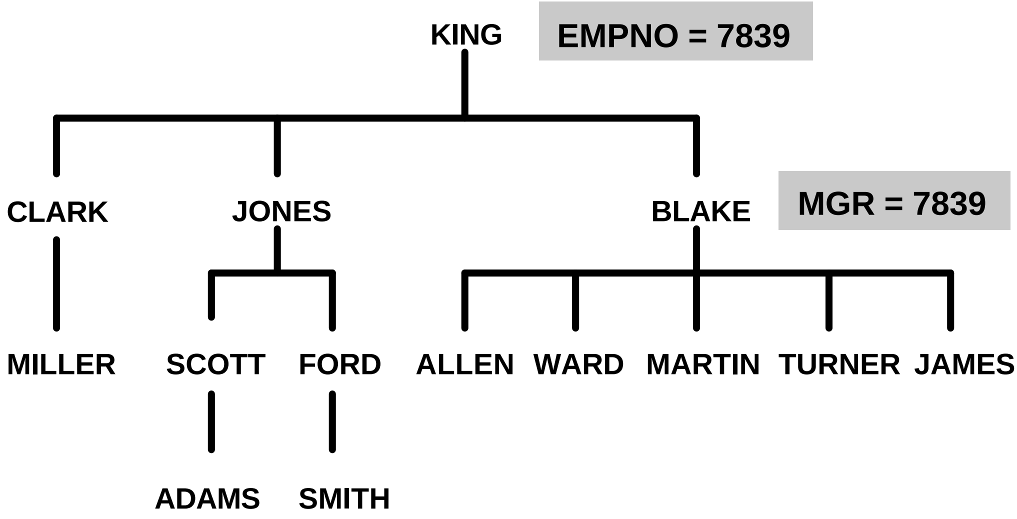
Imagine a family tree with the eldest members of the family found close to the base or trunk of the tree and the youngest members representing branches of the tree, which have branches, which also have branches, and so on.

A hierarchical query is possible when a relationship exists between rows in a table. For example, in the slide, you see that an employee with the job title of MANAGER reports directly to the president of the company. We know this because the MGR column contains the employee number of 7839, which belongs to the president.

Class Management Note

Hierarchical trees are found quite often: human genealogy (family trees), livestock (for breeding purposes), corporate management (management hierarchies), manufacturing (product assembly), evolution (species development), and scientific research to name a few.

Natural Tree Structure



Natural Tree Structure

The EMP table has a tree structure representing the management reporting line. The hierarchy can be created by looking at the relationship between equivalent values in the columns EMPNO and MGR. This relationship has been exploited by joining the table to itself. An employee’s MGR number is the EMPNO of his or her manager.

The parent-child relationship of a tree structure allows you to control:

- The *direction* that the hierarchy is walked
- The *starting point* inside the hierarchy

Note: The slide displays an inverted tree structure of the management hierarchy of the employees in the EMP table.

Hierarchical Queries

```
SELECT[LEVEL], column, expr...
FROM table
[WHERE condition(s)]
[START WITH condition(s)]
[CONNECT BY PRIOR condition(s)];
```

where *condition*:

```
expr comparison_operator expr
```

Keywords and Clauses

Hierarchical queries can be identified by the presence of the CONNECT BY and START WITH clauses.

In the syntax:

SELECT is the standard SELECT clause, with the LEVEL pseudocolumn.

LEVEL is a pseudocolumn. LEVEL returns 1 for a root node, when set to 2 it equals a child, of a root, and so on. LEVEL counts how far down a hierarchical tree you have traveled.

FROM *table* specifies the table, view, or snapshot containing the columns. You can select from only one table.

WHERE restricts the rows returned by the query without affecting other rows of
the hierarchy.

START WITH specifies the root rows of the hierarchy (where to start). This clause is required for a true hierarchical query.

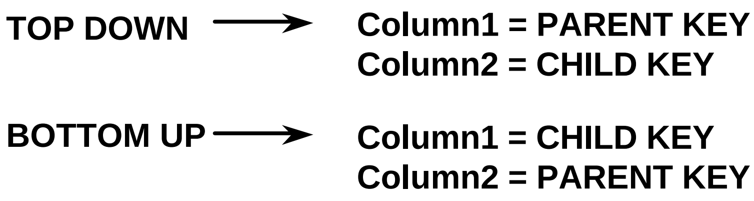
condition is a comparison with expressions.

CONNECT BY PRIOR specifies the columns where the relationship between parent and child rows exist. This clause is required for a hierarchical query.

The SELECT statement cannot contain a join or query from a view that contains a join.

Walking the Tree

DIRECTION



TOP DOWN → Column1 = PARENT KEY
Column2 = CHILD KEY

BOTTOM UP → Column1 = CHILD KEY
Column2 = PARENT KEY

```
CONNECT BY PRIOR column1 = column2
```

Walk top down using the EMP table.

```
... CONNECT BY PRIOR empno = mgr
```

Walking the Tree

The direction of the query, whether it is from parent to child or from child to parent is determined by the CONNECT BY PRIOR column placement. The PRIOR operator refers to the parent row. To find the children of a parent row the Oracle Server evaluates the PRIOR expression for the parent row and the other expression for each row in the table. Rows for which the condition is true are the children of the parent. The Oracle Server always selects children by evaluating the CONNECT BY condition with respect to a current parent row.

Examples

Walk top down using the EMP table. Define a hierarchical relationship in which the EMPNO value of the parent row is equal to the MGR value of the child row.

```
... CONNECT BY PRIOR empno = mgr
```

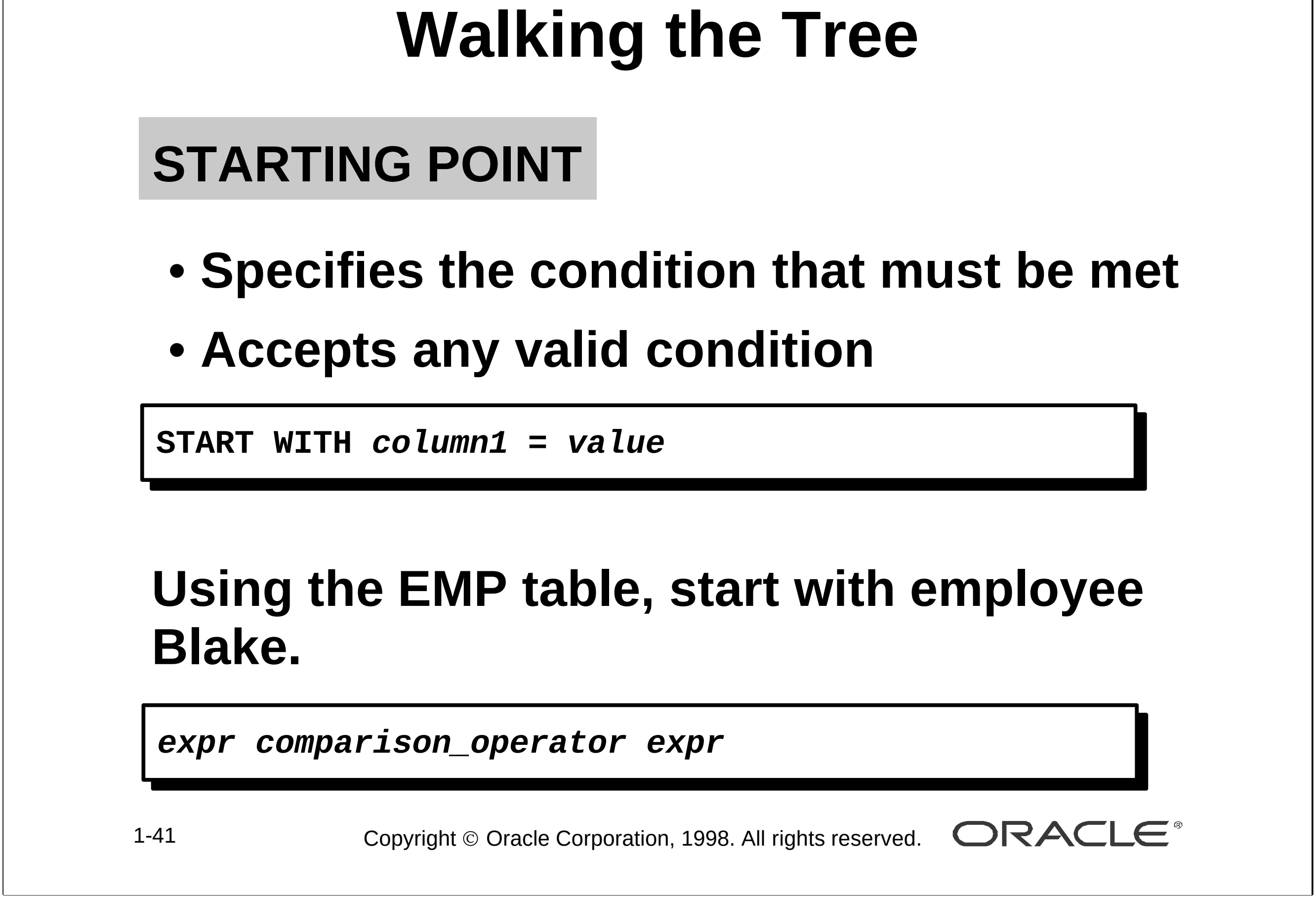
Walking bottom up using the EMP table.

```
... CONNECT BY PRIOR mgr = empno
```

The PRIOR operator does not necessarily need to be coded immediately following the CONNECT BY. Hence the CONNECT BY PRIOR clause below gives the same result as the one in the example above.

```
... CONNECT BY empno = PRIOR mgr
```

Note: The CONNECT BY clause cannot contain a subquery.



Walking the Tree

STARTING POINT

- Specifies the condition that must be met
- Accepts any valid condition

```
START WITH column1 = value
```

Using the EMP table, start with employee Blake.

```
expr comparison_operator expr
```

Walking the Tree (continued)

The row or rows to be used as the root of the tree is determined by the START WITH clause. The START WITH clause can be used in conjunction with any valid condition.

Examples

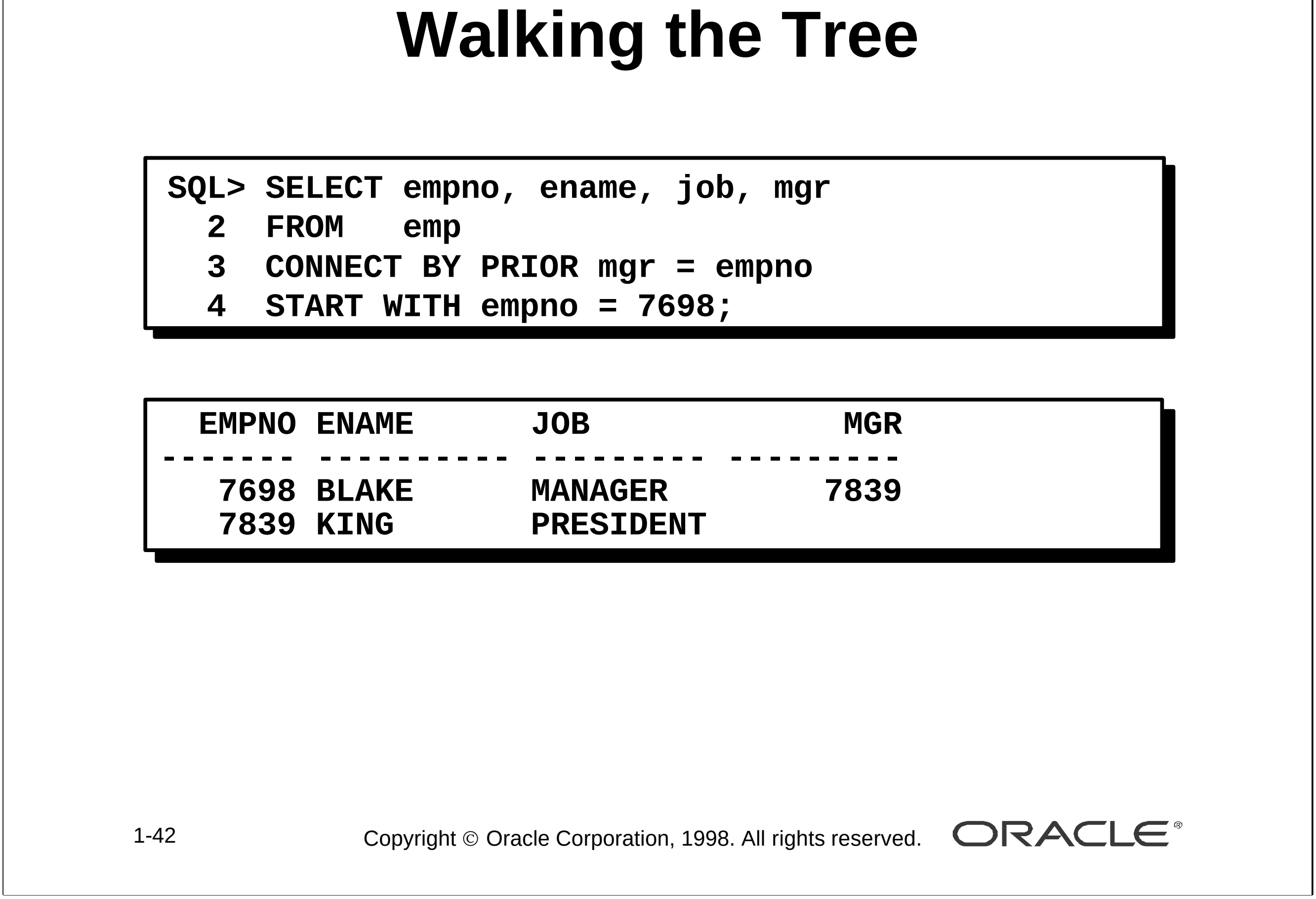
Using the EMP table, start with KING, the president of the company.

```
... START WITH mgr IS NULL
```

Using the EMP table, start with employee SMITH. A START WITH condition can contain a subquery.

```
... START WITH empno = (SELECT empno
                        FROM   emp
                        WHERE  ename = 'SMITH')
```

If the START WITH clause is omitted, the tree walk is started with all of the rows in the table as root rows. If a WHERE clause is used the walk is started with all the rows that satisfy the WHERE condition. This no longer reflects a true hierarchy.



Walking the Tree

```
SQL> SELECT empno, ename, job, mgr
  2  FROM   emp
  3  CONNECT BY PRIOR mgr = empno
  4  START WITH empno = 7698;
```

```
  EMPNO ENAME      JOB              MGR
------- ---------- --------- ---------
   7698 BLAKE      MANAGER        7839
   7839 KING       PRESIDENT
```

Walking Bottom Up

Walking from the bottom up the slide example displays a list of managers starting with the employee with EMPNO 7698.

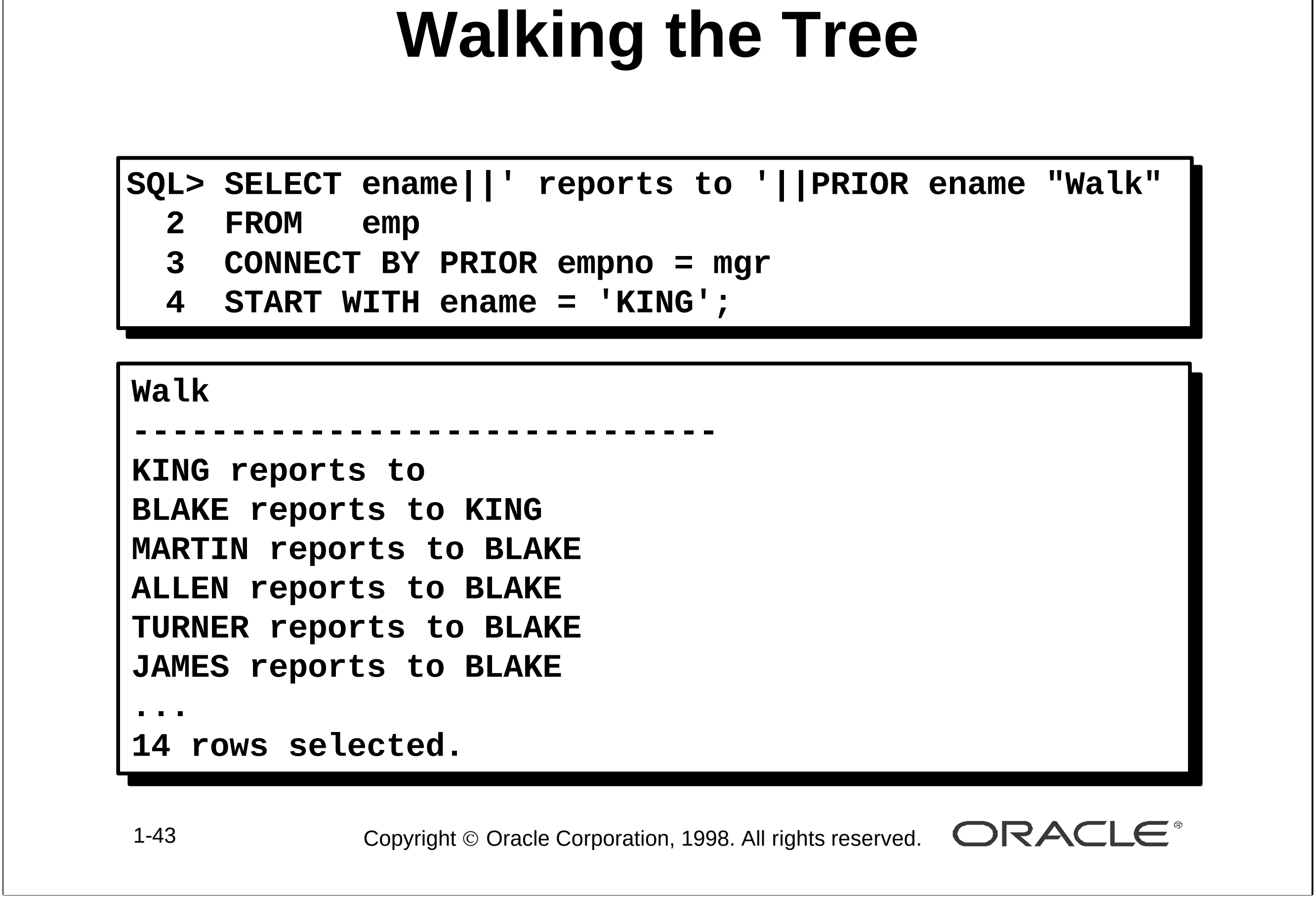
Note: An expression may represent more than a single column.

Example

In this example EMPNO values are evaluated for the parent row and MGR, SAL, and COMM values are evaluated for the child rows. The PRIOR operator applies only to the EMPNO value.

```
... CONNECT BY PRIOR empno = mgr AND sal > comm
```

To qualify as a child row, a row must have a MGR value equal to the EMPNO value of the parent row and it must have a SAL value greater than its COMM value.



Walking the Tree

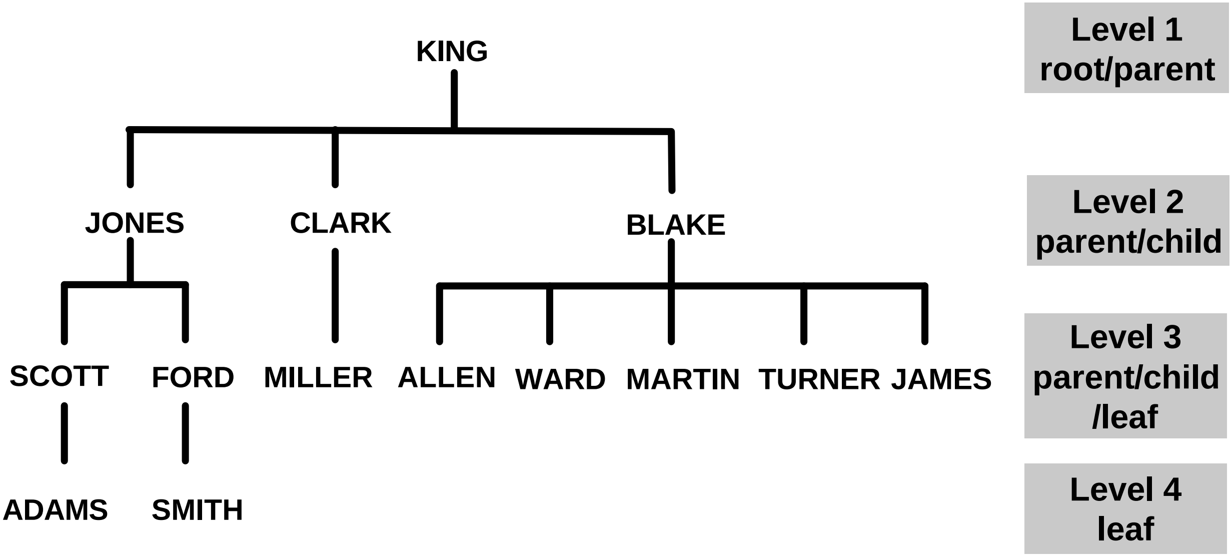
```
SQL> SELECT ename||' reports to '||PRIOR ename "Walk"
  2  FROM   emp
  3  CONNECT BY PRIOR empno = mgr
  4  START WITH ename = 'KING';
```

```
Walk
------------------------------
KING reports to
BLAKE reports to KING
MARTIN reports to BLAKE
ALLEN reports to BLAKE
TURNER reports to BLAKE
JAMES reports to BLAKE
...
14 rows selected.
```

Walking Top Down

Walking from the top down, display the names of the employees and their manager. Use employee “King” as the starting point. Print only one column.

Ranking Rows with the LEVEL Pseudocolumn


KING

JONES CLARK BLAKE

SCOTT FORD MILLER ALLEN WARD MARTIN TURNER JAMES

ADAMS SMITH

Level 1 root/parent

Level 2 parent/child

Level 3 parent/child /leaf

Level 4 leaf

Ranking the Rows Returned

You can explicitly show the rank or level of a row in the hierarchy by using the LEVEL pseudocolumn. This will make your report more readable. The forks where one or more branches split away from a larger branch are called nodes, and the very end of a branch is called a leaf, or leaf node. The diagram above shows the nodes of the inverted tree with their LEVEL values. For example, employee Ford is a parent and a child, while employee Allen is a child and a leaf.

The LEVEL Pseudocolumn

Value	For . . .
1	A root node
2	A child of a root node
3	A child of a child, and so on

Note: A *root node* is the highest node within an inverted tree. A *child node* is any non-root node. A parent node is any node that has children. A leaf node is any node without children.

The number of levels returned by a hierarchical query may be limited by available user memory.

Class Management Note

In the slide King is the root or parent. Clark, Jones, Blake, Scott, and Ford are children and also parents. Miller, Allen, Ward, Martin, Turner, James, Adams, and Smith are children and leaves.

Formatting Hierarchical Reports Using LEVEL and LPAD

Create a report displaying company management levels beginning with the highest level and indenting each of the following levels to the lowest level.

```
SQL> COLUMN org_chart FORMAT A15
SQL> SELECT LPAD(' ', 3 * LEVEL-3)||ename org_chart,
  2  LEVEL, empno, mgr, deptno
  3  FROM   emp
  4  CONNECT BY PRIOR empno = mgr
  5  START WITH mgr is null;
```

Formatting Hierarchical Reports Using LEVEL

The nodes in a tree are assigned level numbers from the root. Use the LPAD function in conjunction with the pseudocolumn LEVEL to display a hierarchical report as an indented tree.

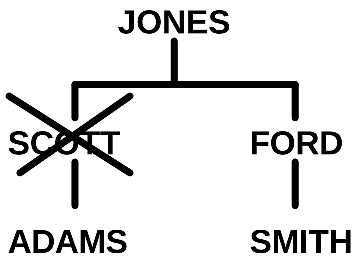
```
ORG_CHART                LEVEL      EMPNO        MGR     DEPTNO
--------------- --------- --------- --------- ---------
KING                    1      7839                  10
   JONES                2      7566      7839        20
      SCOTT             3      7788      7566        20
         ADAMS          4      7876      7788        20
      FORD              3      7902      7566        20
         SMITH          4      7369      7902        20
   BLAKE                2      7698      7839        30
      ALLEN             3      7499      7698        30
      WARD              3      7521      7698        30
      MARTIN            3      7654      7698        30
      TURNER            3      7844      7698        30
      JAMES             3      7900      7698        30
   CLARK                2      7782      7839        10
       MILLER           3      7934      7782        10
14 ROWS SELECTED.
```

LPAD(' ', 3*LEVEL-3) defines the display format. The “3 * LEVEL-3” is the length, and because the set (' ') is not defined, the default, also a space, is used. In other words, this tells SQL to take this string of one space and left pad it to the number of spaces determined by “3 * LEVEL-3.” Basically, each level will be indented by three spaces three levels deep.

Pruning Branches

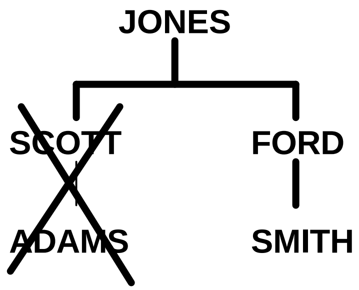
Use the WHERE clause to eliminate an individual node.

WHERE ename != 'SCOTT'



Use the CONNECT BY clause to eliminate a branch.

CONNECT BY PRIOR empno = mgr AND ename != 'SCOTT'



Pruning Branches

You can use the WHERE and CONNECT BY clauses to prune the tree, that is, to control which nodes or rows are displayed. The predicate you use acts as a Boolean condition.

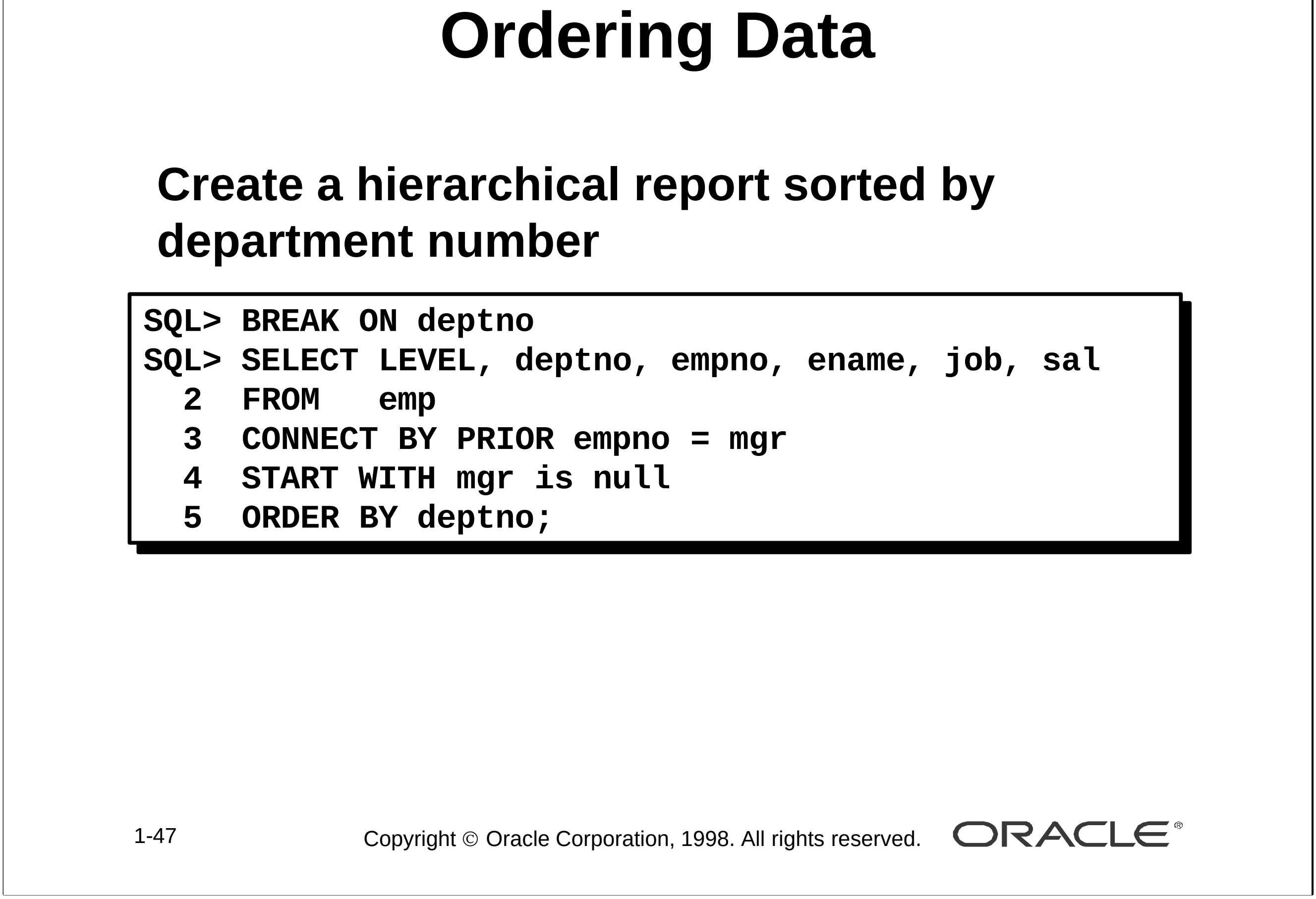
Examples

Starting at the root, walk top down and eliminate employee SCOTT in the result, but process the child rows.

```
SQL> SELECT  deptno, empno, ename, job, sal
  2  FROM    emp
  3  WHERE   ename  != 'SCOTT'
  4  CONNECT BY PRIOR empno = mgr
  5  START WITH mgr IS NULL;
```

Starting at the root, walk top down and eliminate employee SCOTT and all child rows.

```
SQL> SELECT  deptno, empno, ename, job, sal
  2  FROM    emp
  3  CONNECT BY PRIOR empno = mgr
  4  AND ename != 'SCOTT'
  5  START WITH mgr IS NULL;
```



Ordering Data

Create a hierarchical report sorted by department number

```
SQL> BREAK ON deptno
SQL> SELECT LEVEL, deptno, empno, ename, job, sal
  2  FROM   emp
  3  CONNECT BY PRIOR empno = mgr
  4  START WITH mgr is null
  5  ORDER BY deptno;
```

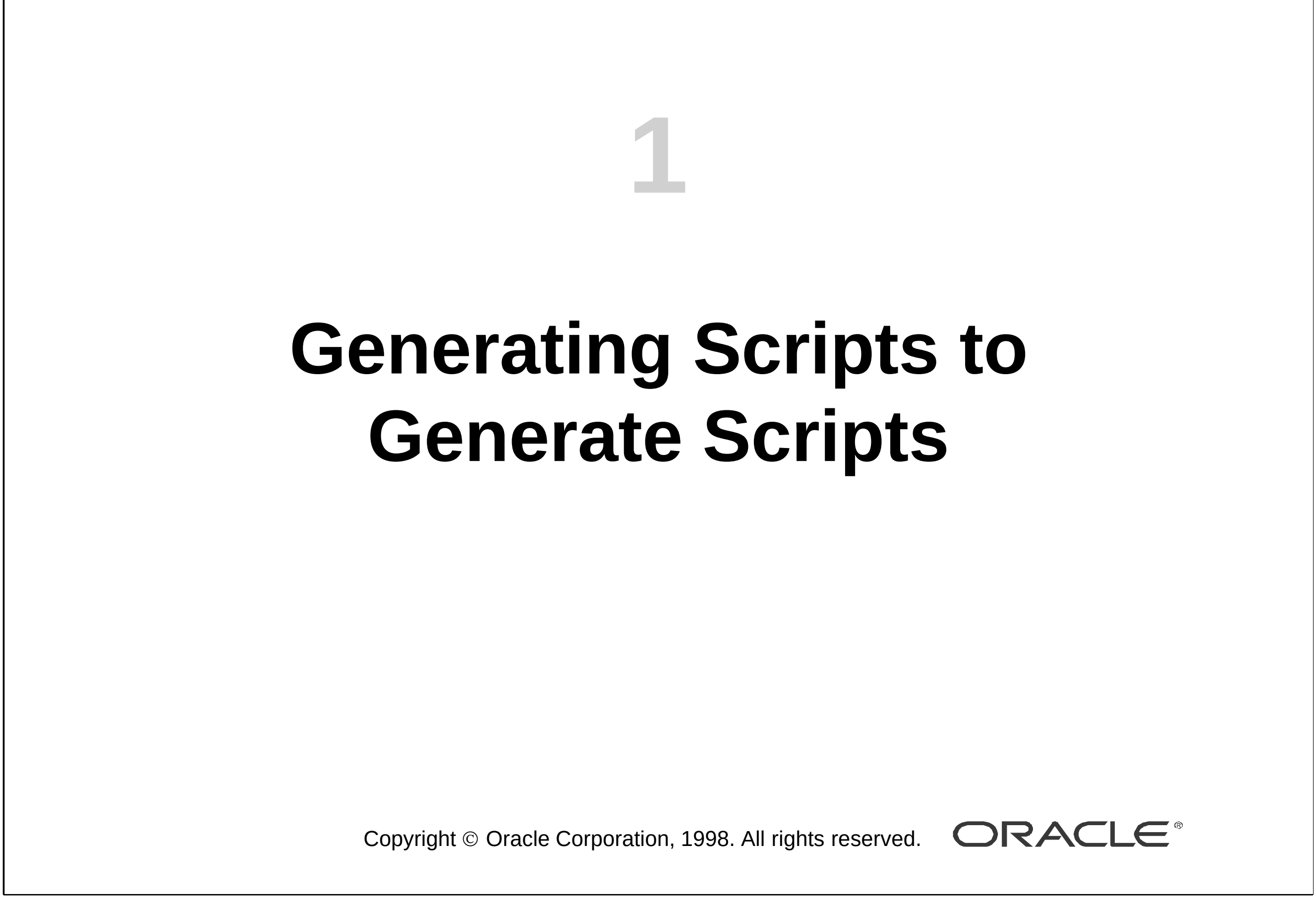
Ordering Data

It is recommended that you do not use the ORDER BY clause when creating hierarchical query reports, because the implicit natural ordering of the tree may be destroyed. The only exception is the use of the LEVEL pseudocolumn in the ORDER BY clause.

```
   LEVEL    DEPTNO    EMPNO ENAME      JOB
-------- --------- -------- ---------- ----------
       1        10     7839 KING       PRESIDENT
       2               7782 CLARK      MANAGER
       3               7934 MILLER     CLERK
       2        20     7566 JONES      MANAGER
       3               7788 SCOTT      ANALYST
       4               7876 ADAMS      CLERK
       3               7902 FORD       ANALYST
       4               7369 SMITH      CLERK
       2        30     7698 BLAKE      MANAGER
       3               7499 ALLEN      SALESMAN
       3               7521 WARD       SALESMAN
       3               7654 MARTIN     SALESMAN
       3               7844 TURNER     SALESMAN
       3               7900 JAMES      CLERK

14 rows selected.
```

Summary

- You can use hierarchical queries to view a hierarchical relationship between rows in a table.
- You control the direction and starting point.
- Pruning can eliminate nodes or branches.



1

Generating Scripts to Generate Scripts

Schedule:	Timing	Topic
	45 minutes	Lecture
	45 minutes	Practice
	90 minutes	Total

Objectives

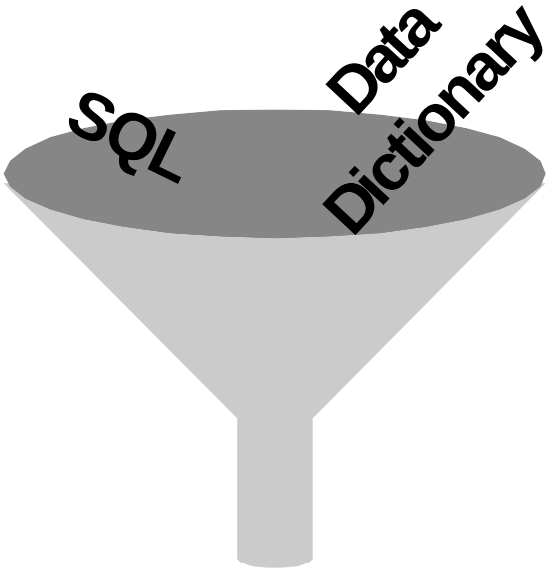
After completing this lesson, you should be able to do the following:

- **Describe the types of problems that are solved by using SQL to generate SQL**
- **Write a script that generates a script of drop table statements**
- **Write a script that generates a script of insert into statements**

Lesson Aim

In this lesson, you will learn how to write a SQL script that generates a SQL script.

Using SQL to Generate SQL



SQL

Data Dictionary

SQL Script

Using SQL to Generate SQL

SQL can be a powerful tool to generate other SQL statements. In most cases this involves writing a script file. You can use SQL from SQL to:

- Avoid repetitive coding
- Get help from the data dictionary
- Drop or re-create database objects
- Generate dynamic predicates that contain run-time parameters

Creating a Basic Script

```
SQL> SELECT 'DROP TABLE ' || object_name || ';'
  2  FROM    user_objects
  3  WHERE   object_type = 'TABLE';
```

```
DROP TABLE EMP;
DROP TABLE DEPT;
DROP TABLE SALGRADE;
. . .
```

A Basic Script

The example in the slide produces a report with DROP TABLE statements for every table you own. The next step is to enhance the report to automate the process.

Class Management Note

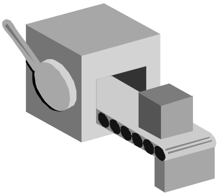
Demo: *script1.sql*.

Purpose: To explain the example on the slide.

Controlling the Environment

```
SET ECHO OFF
SET FEEDBACK OFF
SET PAGESIZE 0

SPOOL droptab.sql

  SQL STATEMENT
SPOOL OFF


SET ECHO ON
SET FEEDBACK ON
SET PAGESIZE 24
```

Set system variables to appropriate values



Set system variables back to default value

Controlling the Environment

In order to execute the SQL statements that are generated, you must capture them in a spool file that can then be started. You must also plan to tidy up the output generated and make sure that you suppress things like headings, feedback messages, ttitles, and so on. You can accomplish all of this by using SQL*Plus commands.

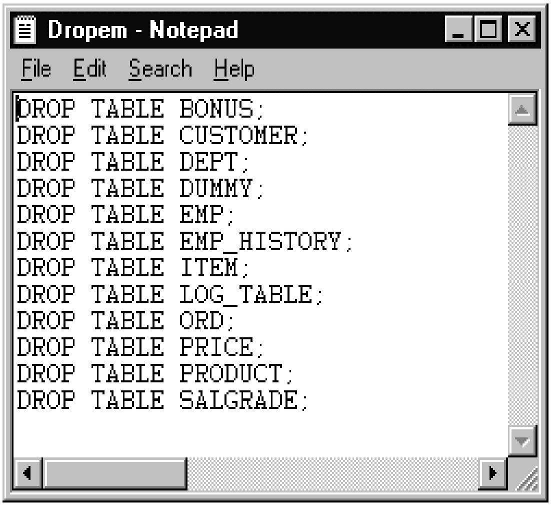
System Variable	Description
TERMOUT	Controls the display of output generated by commands executed from a command file
PAGESIZE	Controls the number of lines in each page (Set PAGESIZE to 0 to suppress headings, page breaks, titles, and so on.)
FEEDBACK	Controls the display of the number of records returned
ECHO	Controls whether the START command lists each statement in a command file as the statement is executed

The Complete Picture

```
SET ECHO OFF
SET FEEDBACK OFF
SET PAGESIZE 0

SPOOL dropem.sql

SELECT 'DROP TABLE ' || object_name || ';'
FROM    user_objects
WHERE   object_type = 'TABLE';

SPOOL OFF



SET ECHO ON
SET FEEDBACK ON
SET PAGESIZE 24
```

The Complete Picture

The spool file *dropem.sql* contains the following data. This file can now be started from the SQL prompt.



```
DROP TABLE BONUS;
DROP TABLE CUSTOMER;
DROP TABLE DEPT;
DROP TABLE DUMMY;
DROP TABLE EMP;
DROP TABLE EMP_HISTORY;
DROP TABLE ITEM;
DROP TABLE LOG_TABLE;
DROP TABLE ORD;
DROP TABLE PRICE;
DROP TABLE PRODUCT;
DROP TABLE SALGRADE;
```

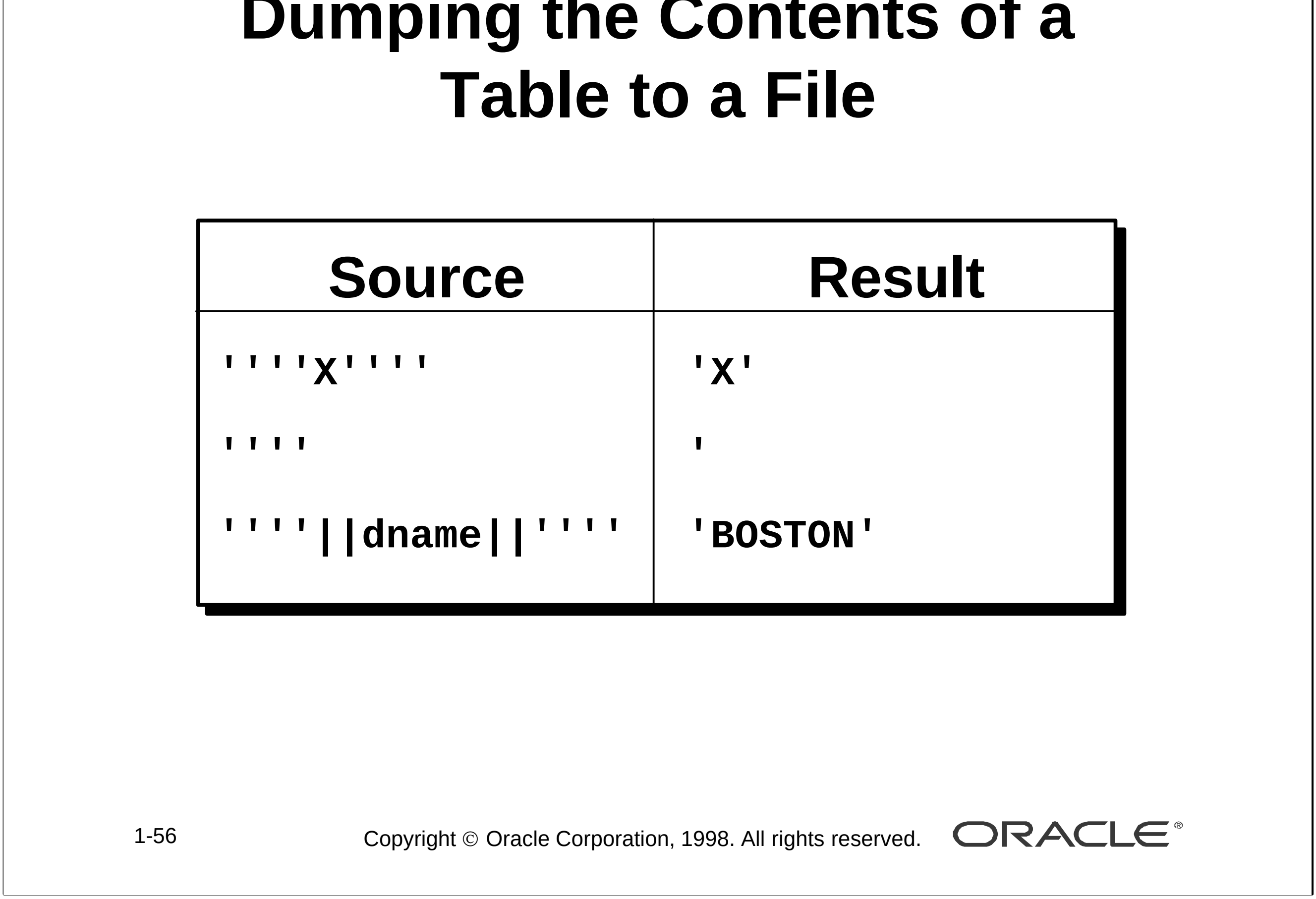
Dumping the Contents of a Table to a File

```
SET HEADING OFF ECHO OFF FEEDBACK OFF
SET PAGESIZE 0

SPOOL data.sql

SELECT 'INSERT INTO dept VALUES ('||
       deptno||','||
       ''''||dname||''''||','||
       ''''||loc||''''||');'
FROM   dept;

SPOOL OFF

SET HEADING ON ECHO OFF FEEDBACK ON
SET PAGESIZE 24
```

Dumping Table Contents to a File

Although the data already exists in a table, sometimes it is useful to have the values for the rows of a table in a text file in the format of an INSERT INTO VALUES statement.

The example in the slide produces INSERT statements for the DEPT table, captured in the file *data.sql.*



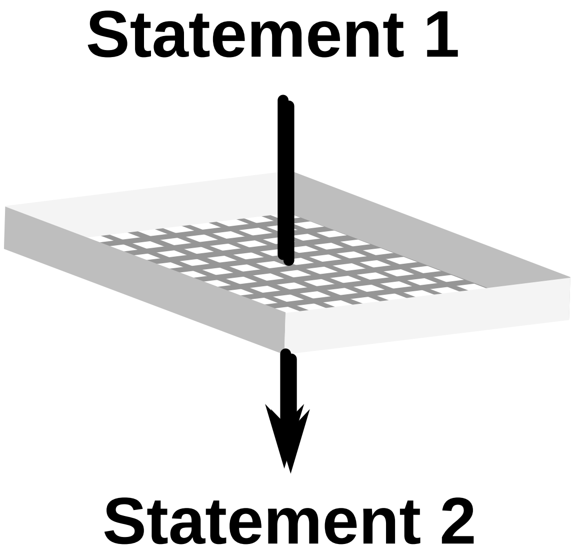
Dumping the Contents of a Table to a File

<table>
<tr><th>Source</th><th>Result</th></tr>
<tr><td>''''X''''</td><td>'X'</td></tr>
<tr><td>''''</td><td>'</td></tr>
<tr><td>''''||dname||''''</td><td>'BOSTON'</td></tr>
</table>

Dumping Table Contents to a File (continued)

You may have noticed the large number of single quotes in the slide on the previous page. A set of four single quotes produces one single quote in the final statement. Also remember that character values must be surrounded by quotes.

Generating a Dynamic Predicate



Statement 1

Statement 2

ORACLE®

Generating a Dynamic Predicate

Generate a report for a specific department and/or for starting on a specific date.

```
SQL> COLUMN my_col NEW_VALUE dyn_where_clause
SQL> SELECT DECODE('&deptno',
  2        null, DECODE('&hiredate',null,' ',
  3                'where hiredate = '''||'&hiredate'||''''),
  4            DECODE('&hiredate',
  5                  null,'where deptno = '||'&deptno',
  6                  'where deptno = '||'&deptno'||
  7                  ' and hiredate = '''||'&hiredate'||'''')
  8                 )  my_col
  9   FROM dual;
```

```
SQL> SELECT ename FROM emp &dyn_where_clause;
```

Note: You can use the NEW_VALUE clause of the COLUMN command to specify a variable to hold a COLUMN value. More about NEW_VALUE is covered in a subsequent lesson.

Class Management Note

Demo: *dyn.sql*.

Purpose: To explain the above example.

Summary

- You can select virtually anything.
- Script files often use the data dictionary.
- You use the SPOOL command to capture output in a file.