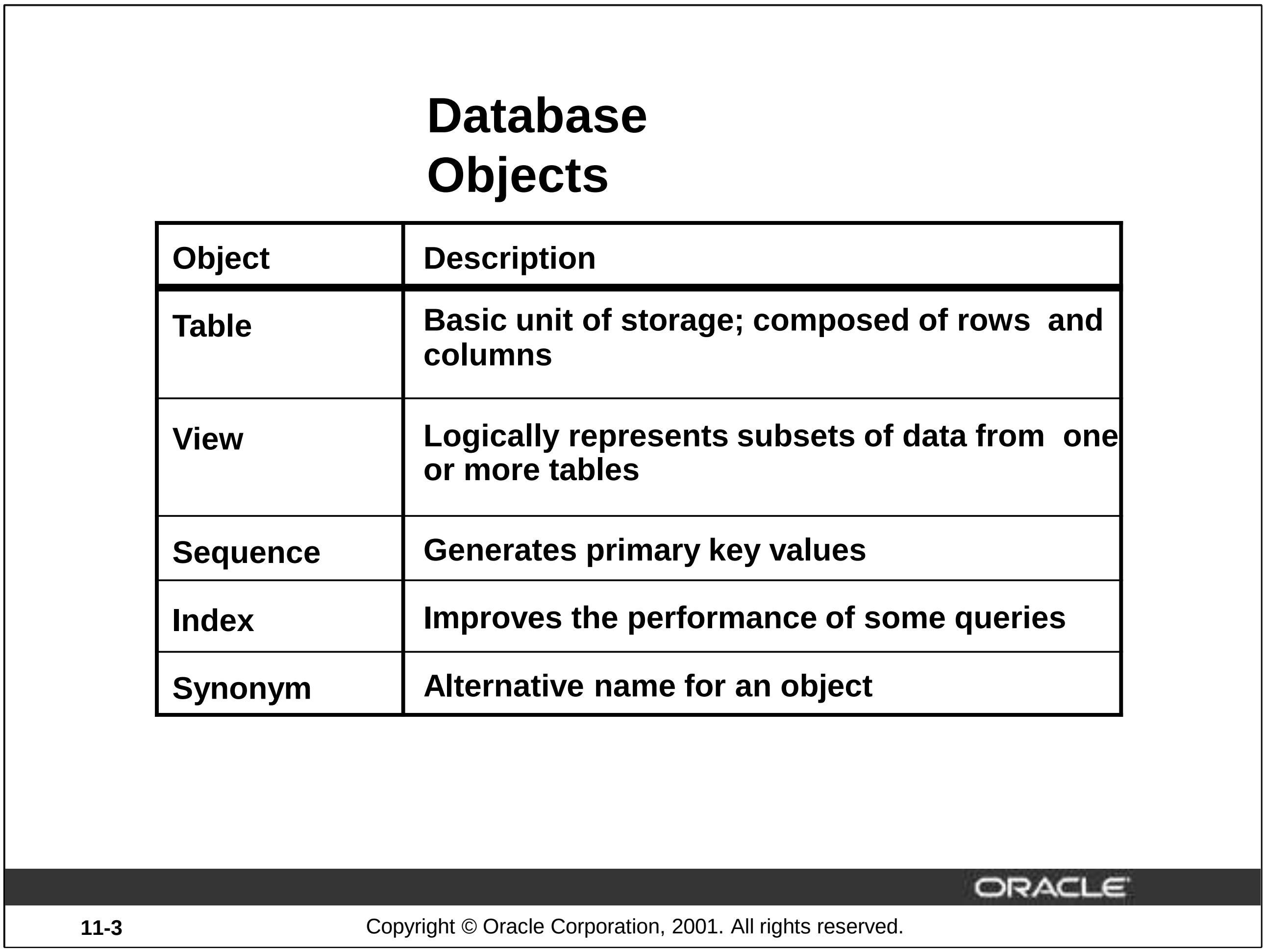
11

Creating Views

Objectives

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

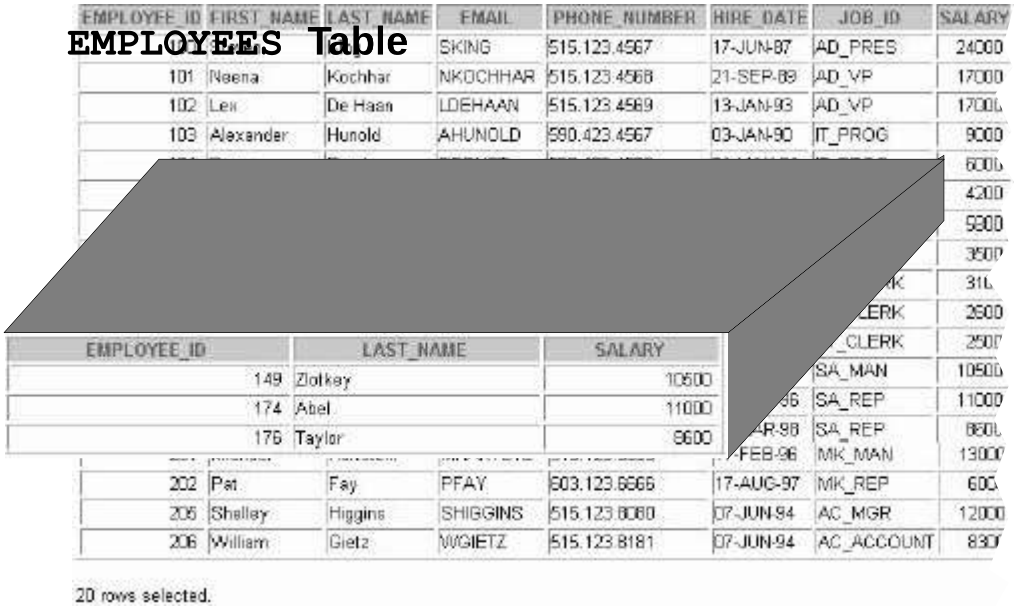
- Describe a view
- Create, alter the definition of, and drop a view
- Retrieve data through a view
- Insert, update, and delete data through a view
- Create and use an inline view
- Perform top-*n* analysis



Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Generates primary key values
Index	Improves the performance of some queries
Synonym	Alternative name for an object

What Is a View?

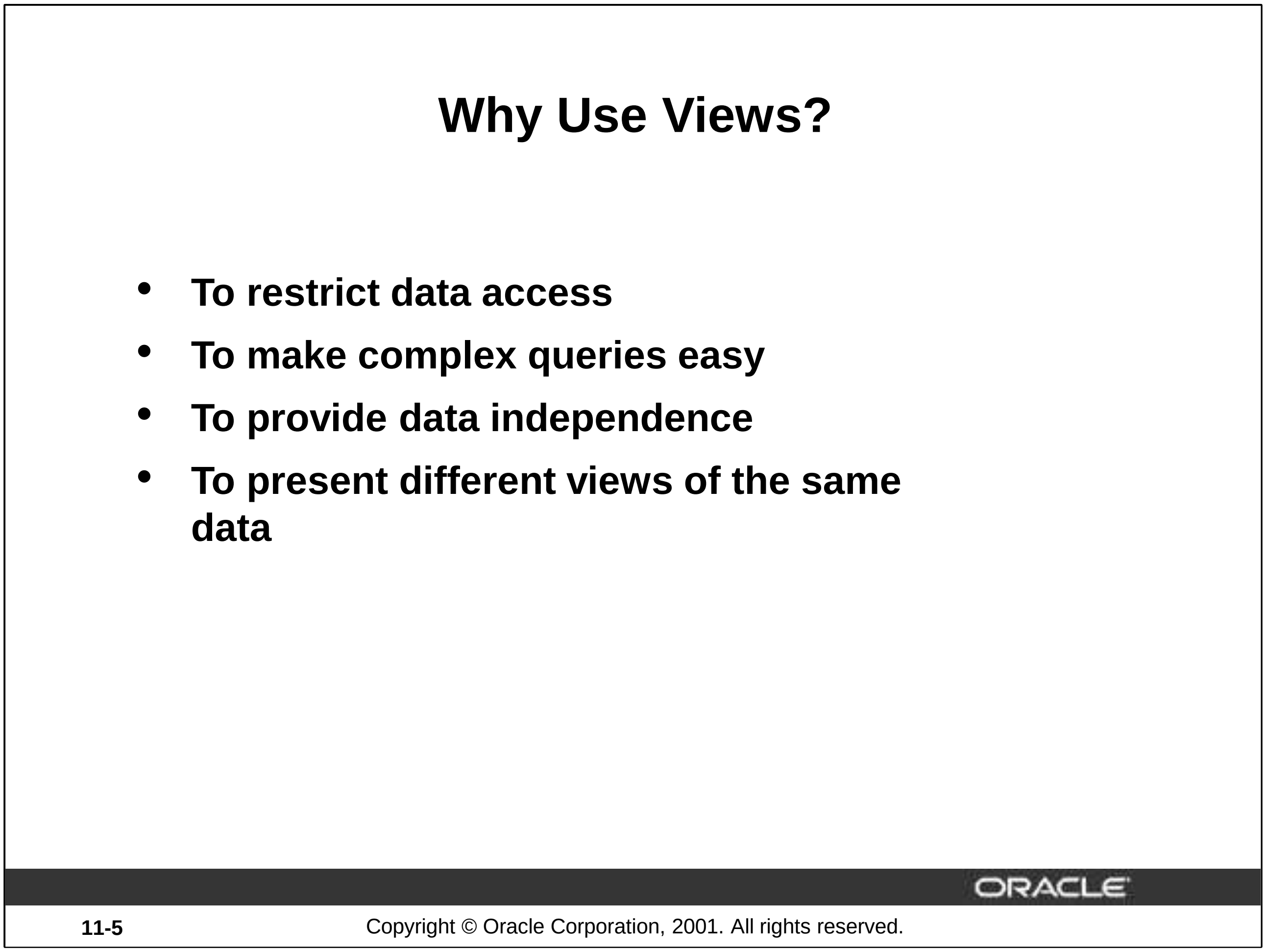


EMPLOYEES Table

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
			SKING	515.123.4567	17-JUN-87	AD_PRES	24000
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	1700
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000
							600
							4200
							5800
							350
						RK	31
						LERK	2600
						_CLERK	250
						SA_MAN	1050
					96	SA_REP	1100
					AR-98	SA_REP	860
					FEB-96	MK_MAN	1300
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	60
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	830

20 rows selected.

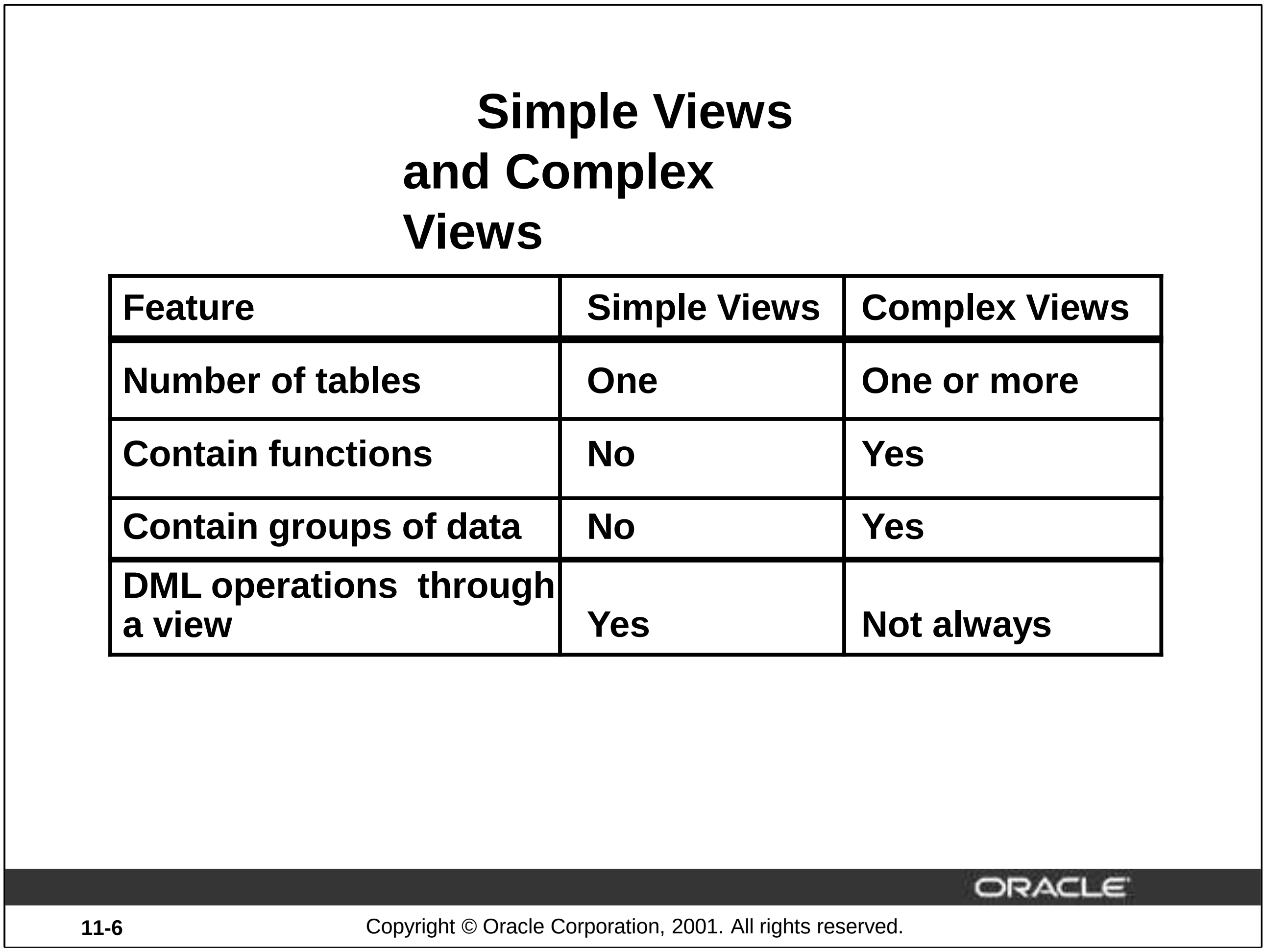
EMPLOYEE_ID	LAST_NAME	SALARY
149	Zlotkey	10500
174	Abel	11000
176	Taylor	8600



Why Use Views?

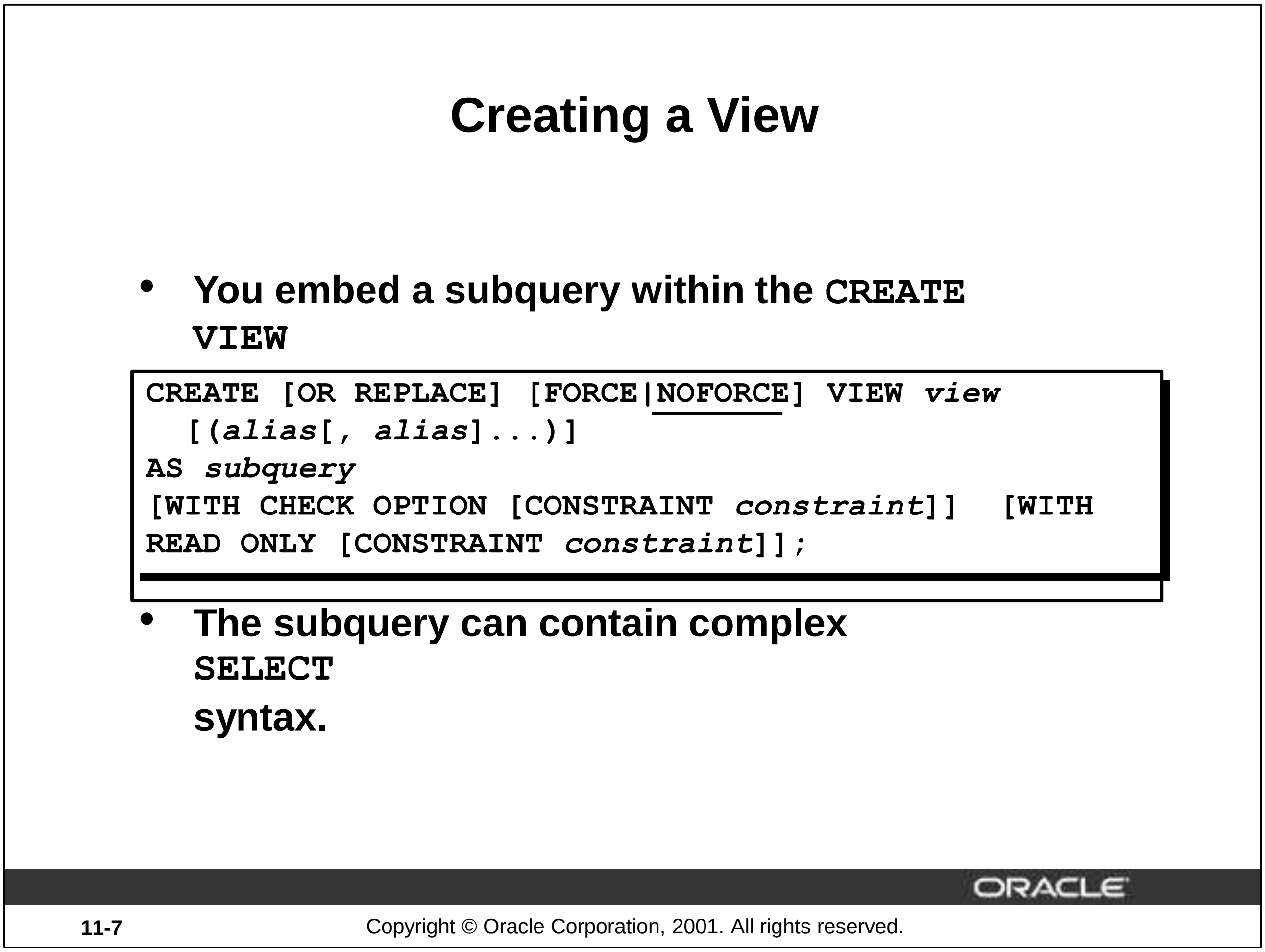
- To restrict data access
- To make complex queries easy
- To provide data independence
- To present different views of the same data

11-5



Simple Views and Complex Views

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operations through a view	Yes	Not always



Creating a View

- You embed a subquery within the `CREATE VIEW`

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
  [(alias[, alias]...)]
AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]  [WITH
READ ONLY [CONSTRAINT constraint]];
```

- The subquery can contain complex `SELECT` syntax.

Creating a View

- Create a view, **`EMPVU80`**, that contains details of employees in department 80.

```
CREATE VIEW  empvu80
AS SELECT    employee_id, last_name, salary  FROM
             employees
   WHERE     department_id = 80;  View created.
```

- Describe the structure of the view by using the
  *i*SQL*Plus **`DESCRIBE`** command.

```
DESCRIBE empvu80
```

Creating a View

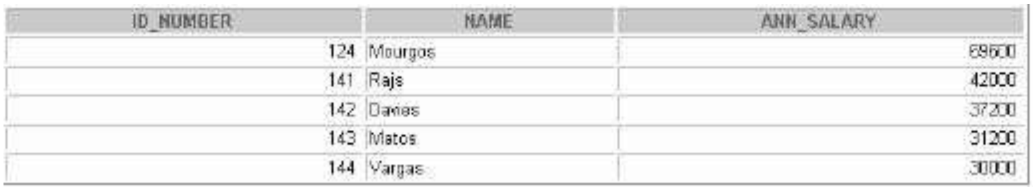
- **Create a view by using column aliases in the subquery.**

```
CREATE VIEW  salvu50
AS SELECT  employee_id ID_NUMBER, last_name NAME,
           salary*12 ANN_SALARY
FROM         employees
   WHERE     department_id = 50;  View created.
```

- **Select the columns from this view by the given alias names.**

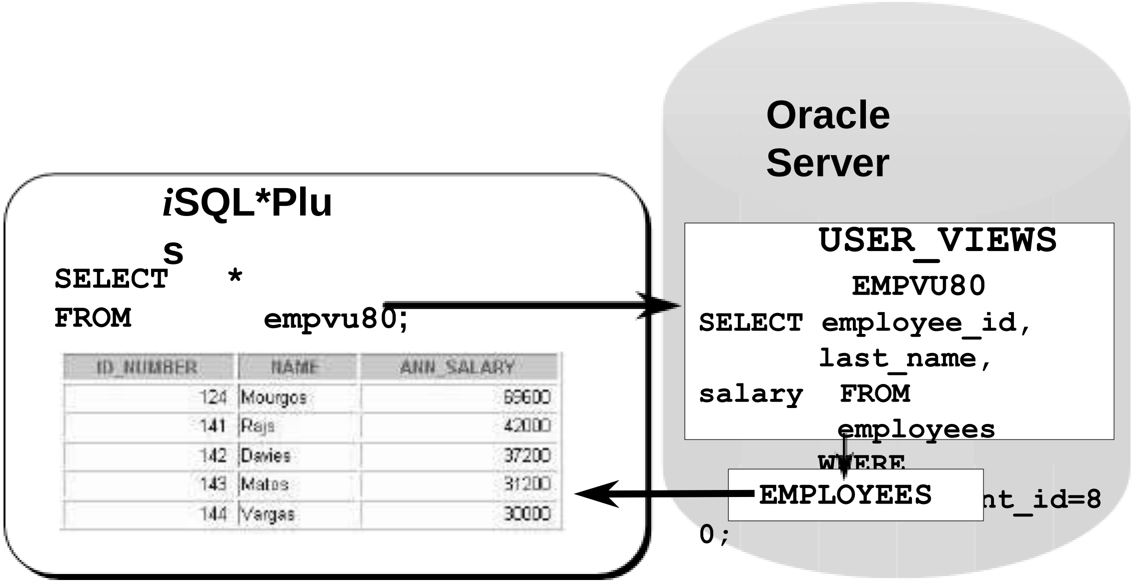
Retrieving Data from a View

```
SELECT *
FROM   salvu50;
```



ID_NUMBER	NAME	ANN_SALARY
124	Mourgos	69600
141	Rajs	42000
142	Davies	37200
143	Matos	31200
144	Vargas	30000

Querying a View



*i*SQL*Plus

```
SELECT   *
FROM        empvu80;
```

ID_NUMBER	NAME	ANN_SALARY
124	Mourgos	69600
141	Rajs	42000
142	Davies	37200
143	Matos	31200
144	Vargas	30000

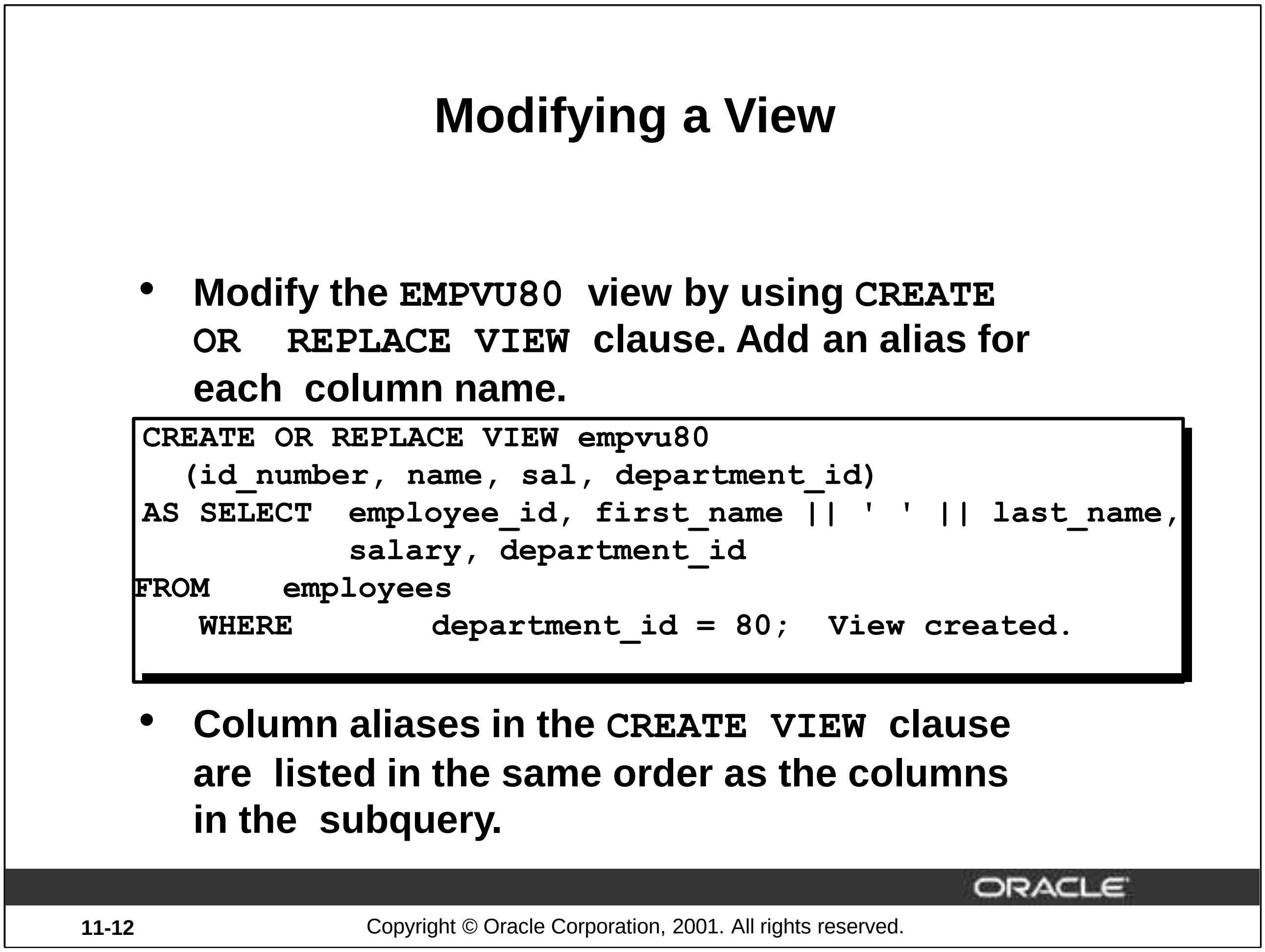
Oracle Server

USER_VIEWS

EMPVU80

```
SELECT employee_id,
       last_name,
salary  FROM
        employees
       WHERE
                 nt_id=8
0;
```

EMPLOYEES



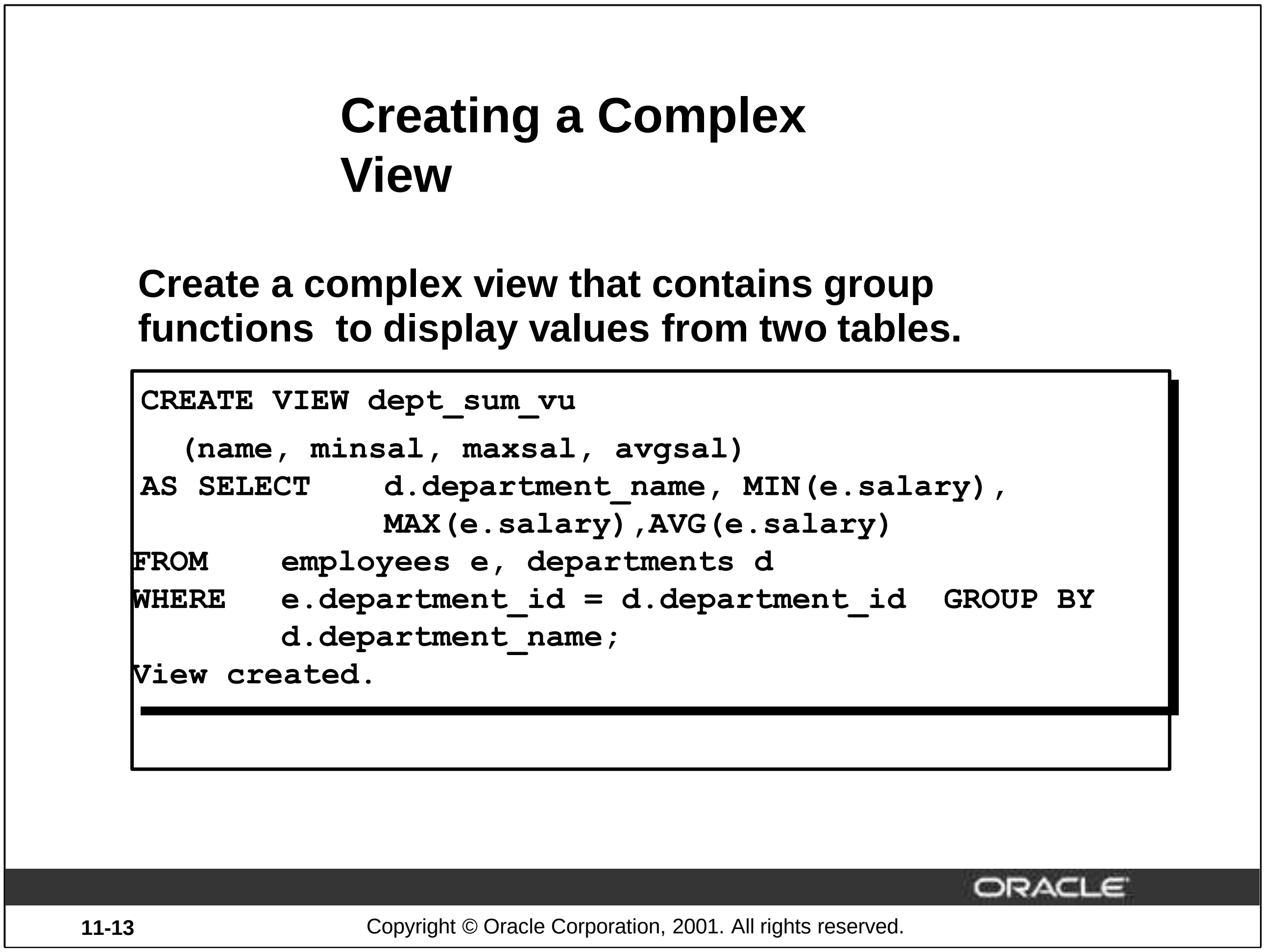
Modifying a View

- Modify the `EMPVU80` view by using `CREATE OR REPLACE VIEW` clause. Add an alias for each column name.

```
CREATE OR REPLACE VIEW empvu80
  (id_number, name, sal, department_id)
AS SELECT  employee_id, first_name || ' ' || last_name,
           salary, department_id
FROM    employees
   WHERE       department_id = 80;  View created.
```

- Column aliases in the `CREATE VIEW` clause are listed in the same order as the columns in the subquery.

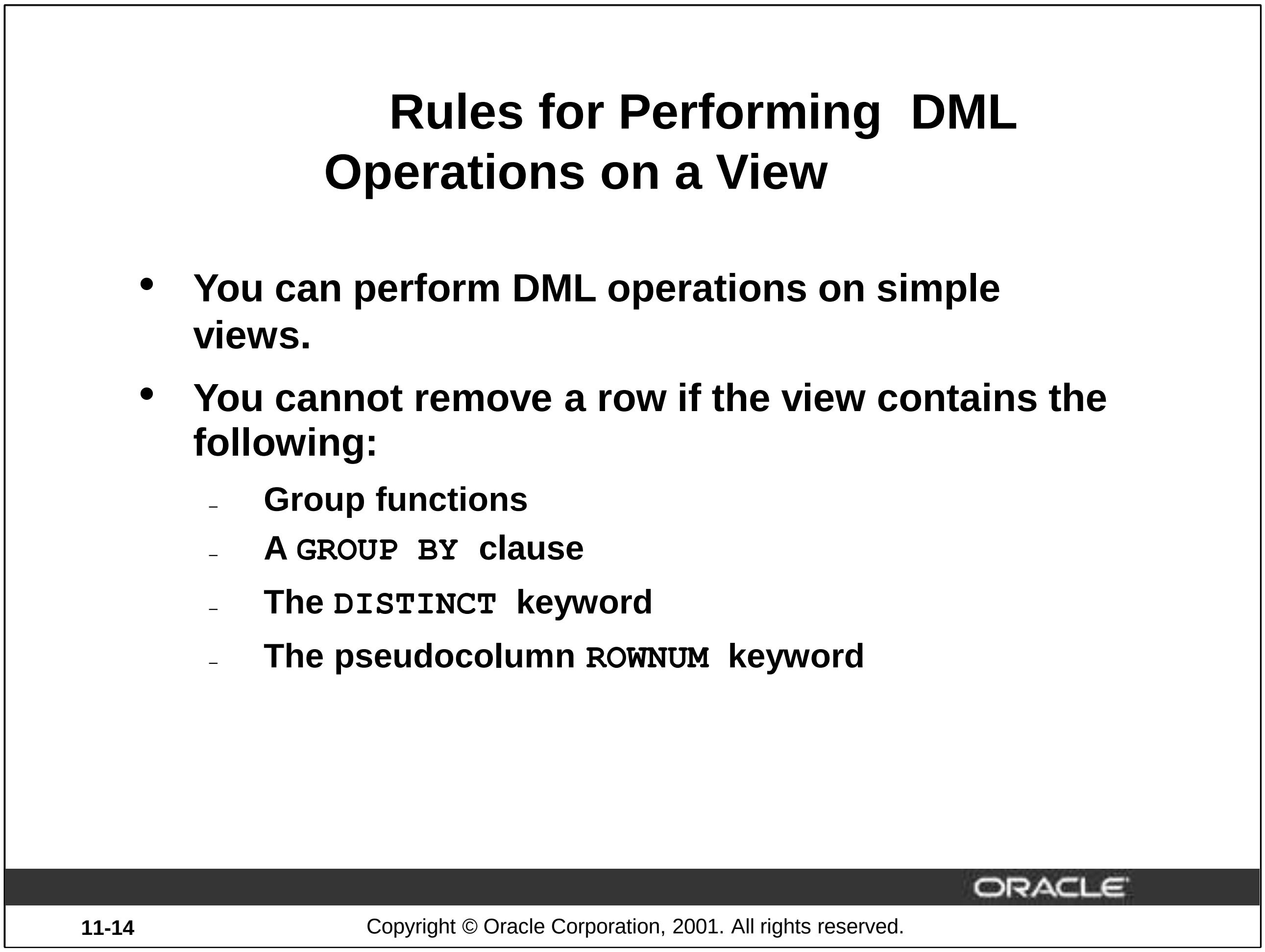
11-12



Creating a Complex View

Create a complex view that contains group functions to display values from two tables.

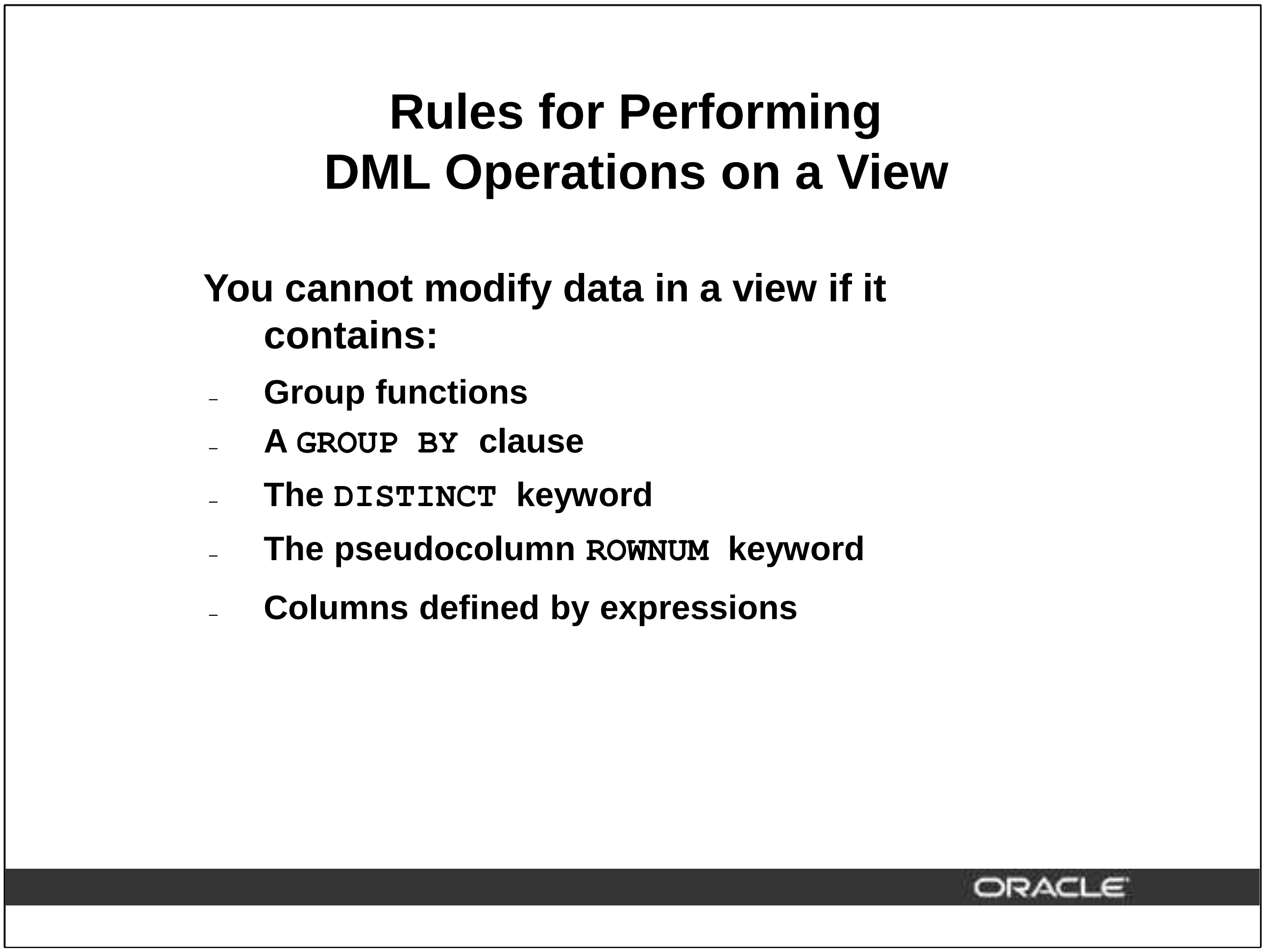
```
CREATE VIEW dept_sum_vu
  (name, minsal, maxsal, avgsal)
AS SELECT    d.department_name, MIN(e.salary),
             MAX(e.salary),AVG(e.salary)
FROM    employees e, departments d
WHERE   e.department_id = d.department_id  GROUP BY
        d.department_name;
View created.
```



Rules for Performing DML Operations on a View

- You can perform DML operations on simple views.
- You cannot remove a row if the view contains the following:
  - Group functions
  - A `GROUP BY` clause
  - The `DISTINCT` keyword
  - The pseudocolumn `ROWNUM` keyword

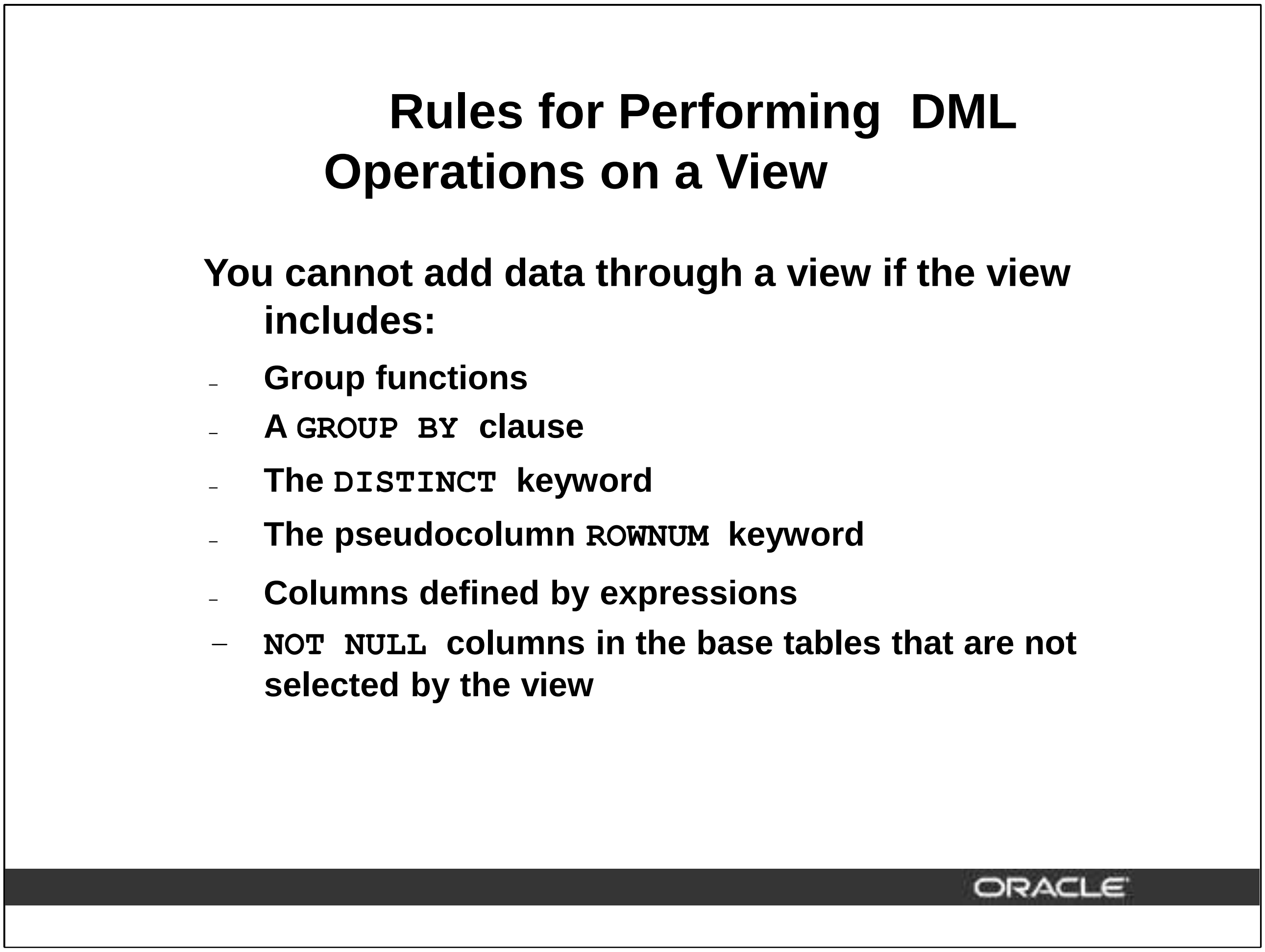
11-14



Rules for Performing DML Operations on a View

You cannot modify data in a view if it contains:

- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- The pseudocolumn `ROWNUM` keyword
- Columns defined by expressions



Rules for Performing DML Operations on a View

You cannot add data through a view if the view includes:

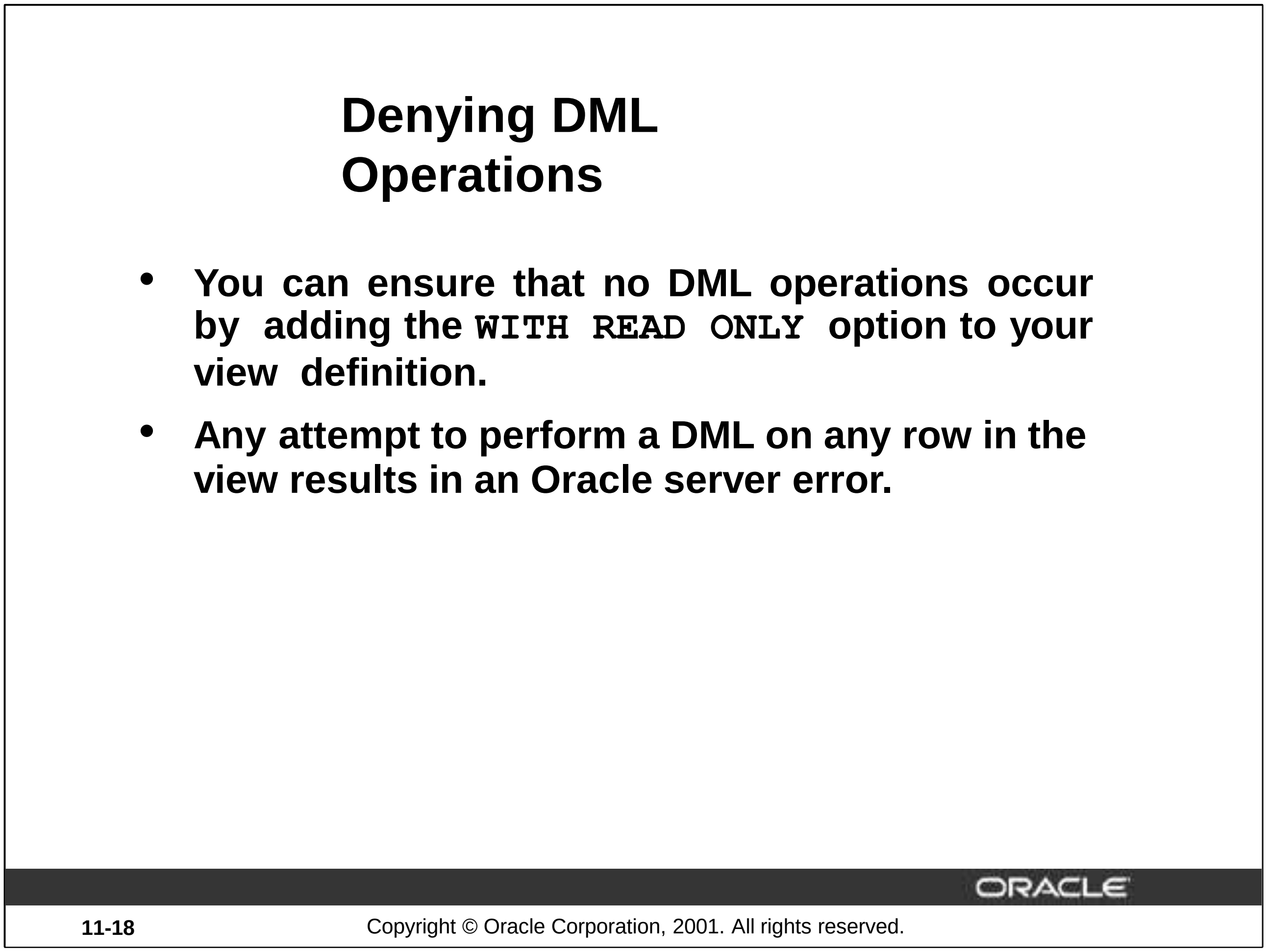
- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- The pseudocolumn `ROWNUM` keyword
- Columns defined by expressions
- `NOT NULL` columns in the base tables that are not selected by the view

Using the `WITH CHECK OPTION` Clause

- You can ensure that DML operations performed on the view stay within the domain of the view by using the `WITH CHECK OPTION` clause.

```
CREATE OR REPLACE VIEW empvu20  AS SELECT      *
   FROM     employees
   WHERE    department_id = 20
   WITH CHECK OPTION CONSTRAINT empvu20_ck;
View created.
```

- Any attempt to change the department number for any row in the view fails because it violates the `WITH CHECK OPTION` constraint.

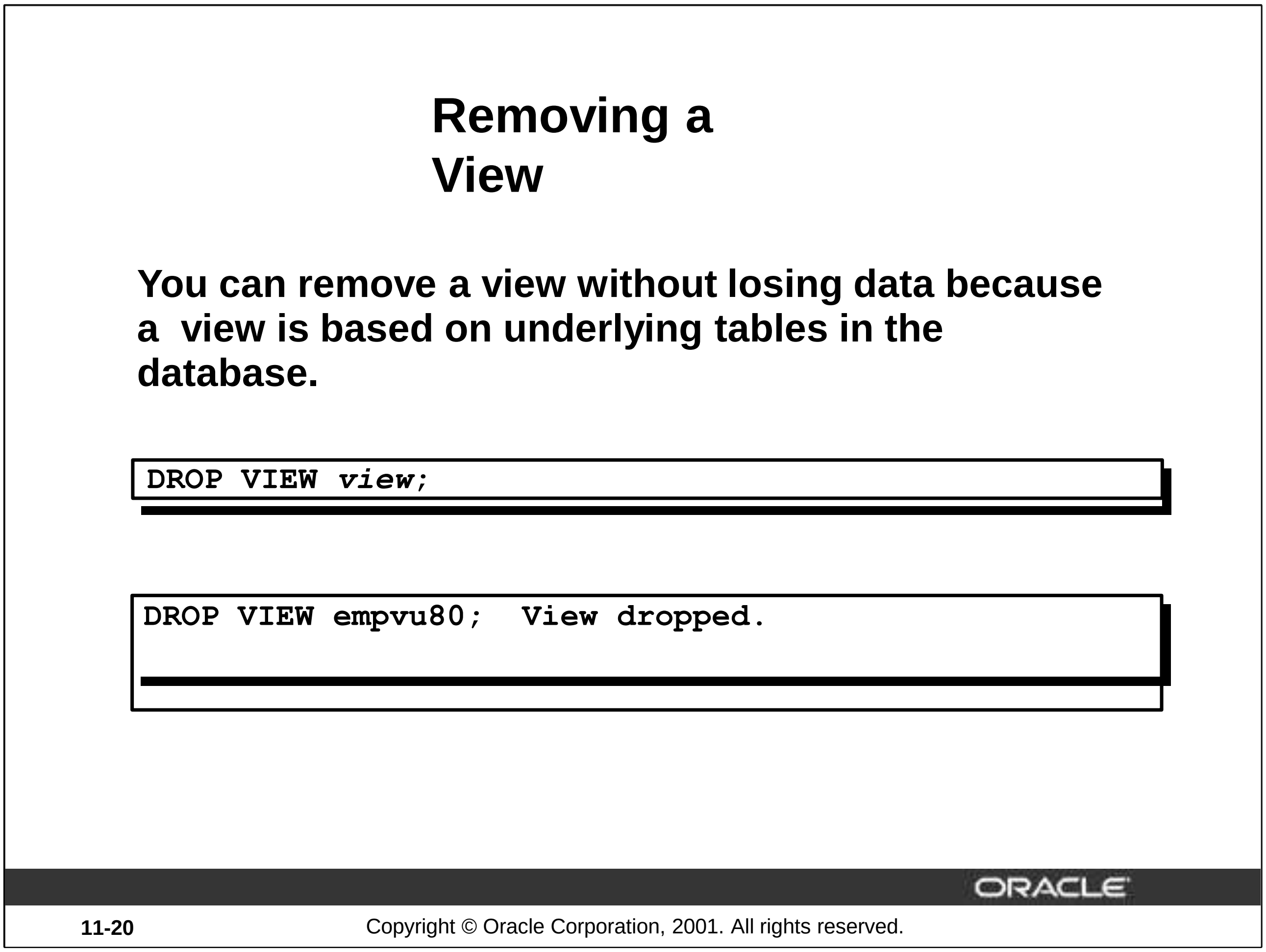


Denying DML Operations

- You can ensure that no DML operations occur by adding the `WITH READ ONLY` option to your view definition.
- Any attempt to perform a DML on any row in the view results in an Oracle server error.

Denying DML Operations

```
CREATE OR REPLACE VIEW empvu10
    (employee_number, employee_name, job_title)  AS
SELECT employee_id, last_name, job_id
       FROM    employees
       WHERE   department_id = 10  WITH READ ONLY;
    View created.
```

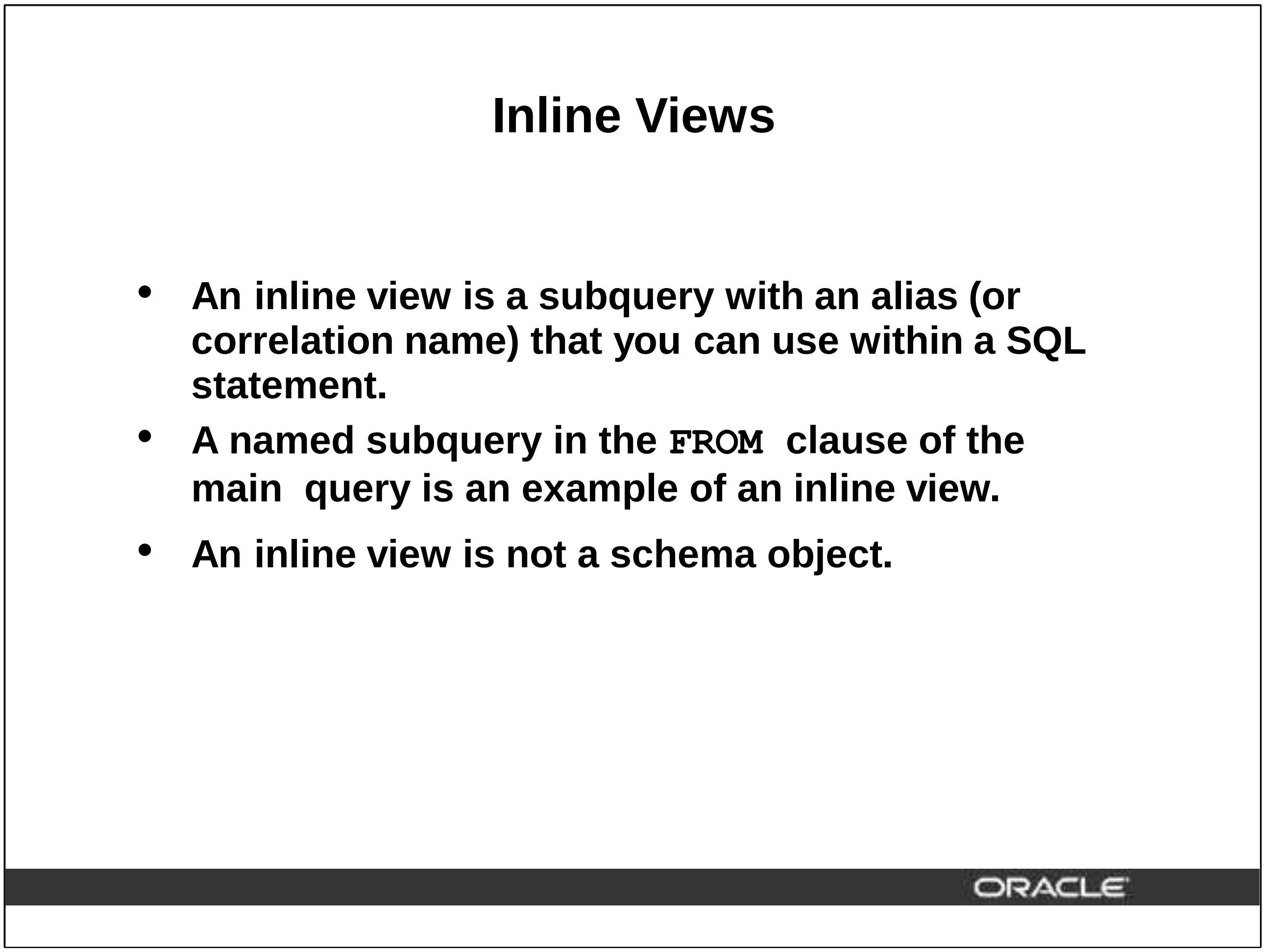


Removing a View

You can remove a view without losing data because a view is based on underlying tables in the database.

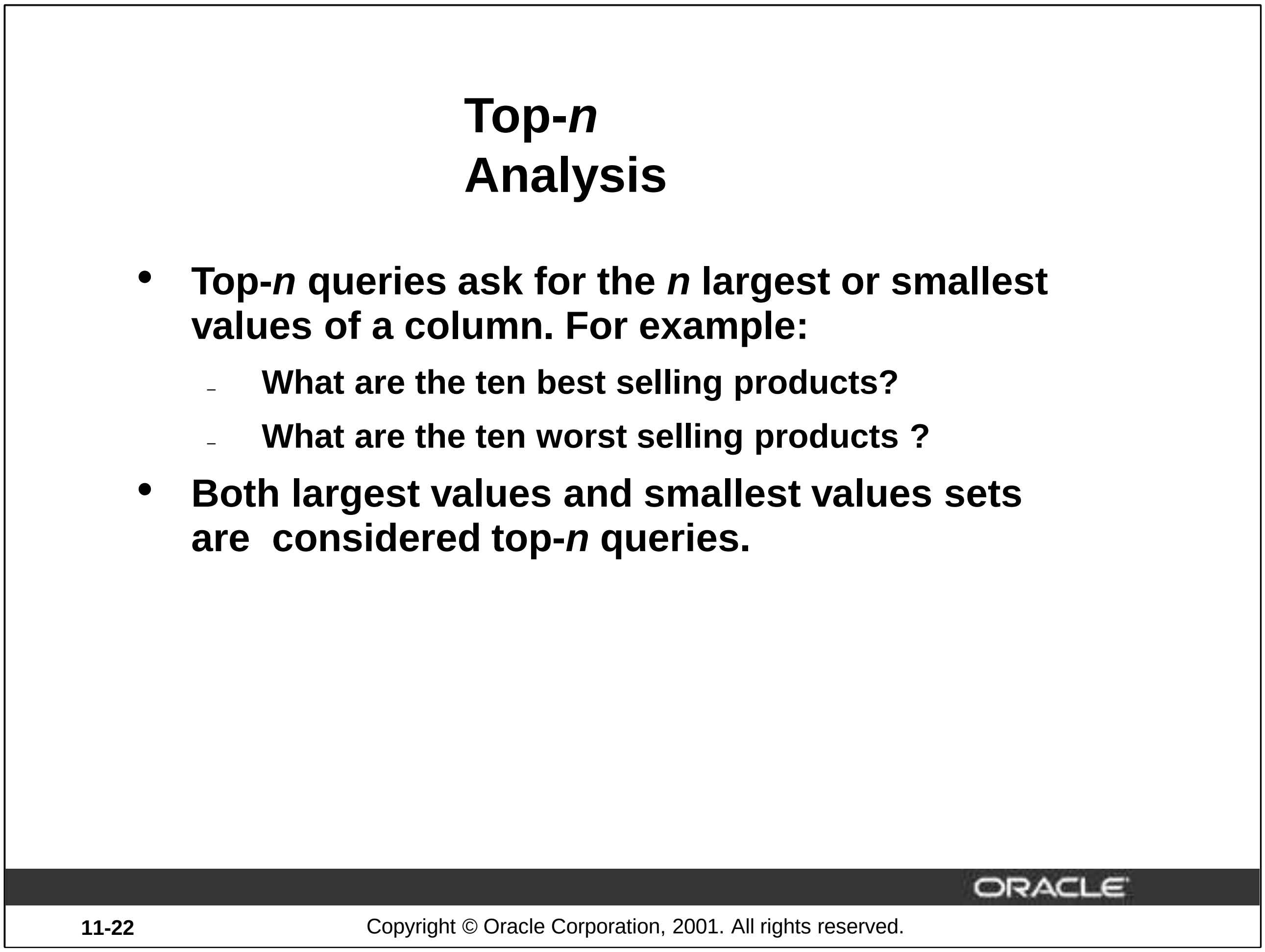
```
DROP VIEW view;
```

```
DROP VIEW empvu80;  View dropped.
```



Inline Views

- An inline view is a subquery with an alias (or correlation name) that you can use within a SQL statement.
- A named subquery in the `FROM` clause of the main query is an example of an inline view.
- An inline view is not a schema object.



Top-*n* Analysis

- **Top-*n* queries ask for the *n* largest or smallest values of a column. For example:**
  - **What are the ten best selling products?**
  - **What are the ten worst selling products ?**
- **Both largest values and smallest values sets are considered top-*n* queries.**

11-22

Performing Top-*n* Analysis

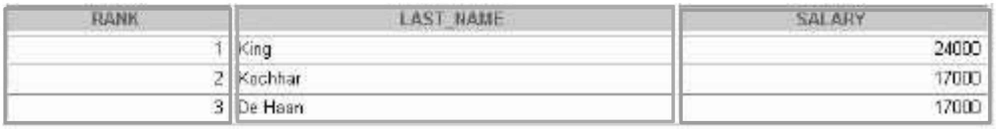
The high-level structure of a top-*n* analysis query is:

```
SELECT [column_list], ROWNUM
FROM   (SELECT [column_list]  FROM table
        ORDER  BY Top-N_column)  WHERE ROWNUM <=  N;
```

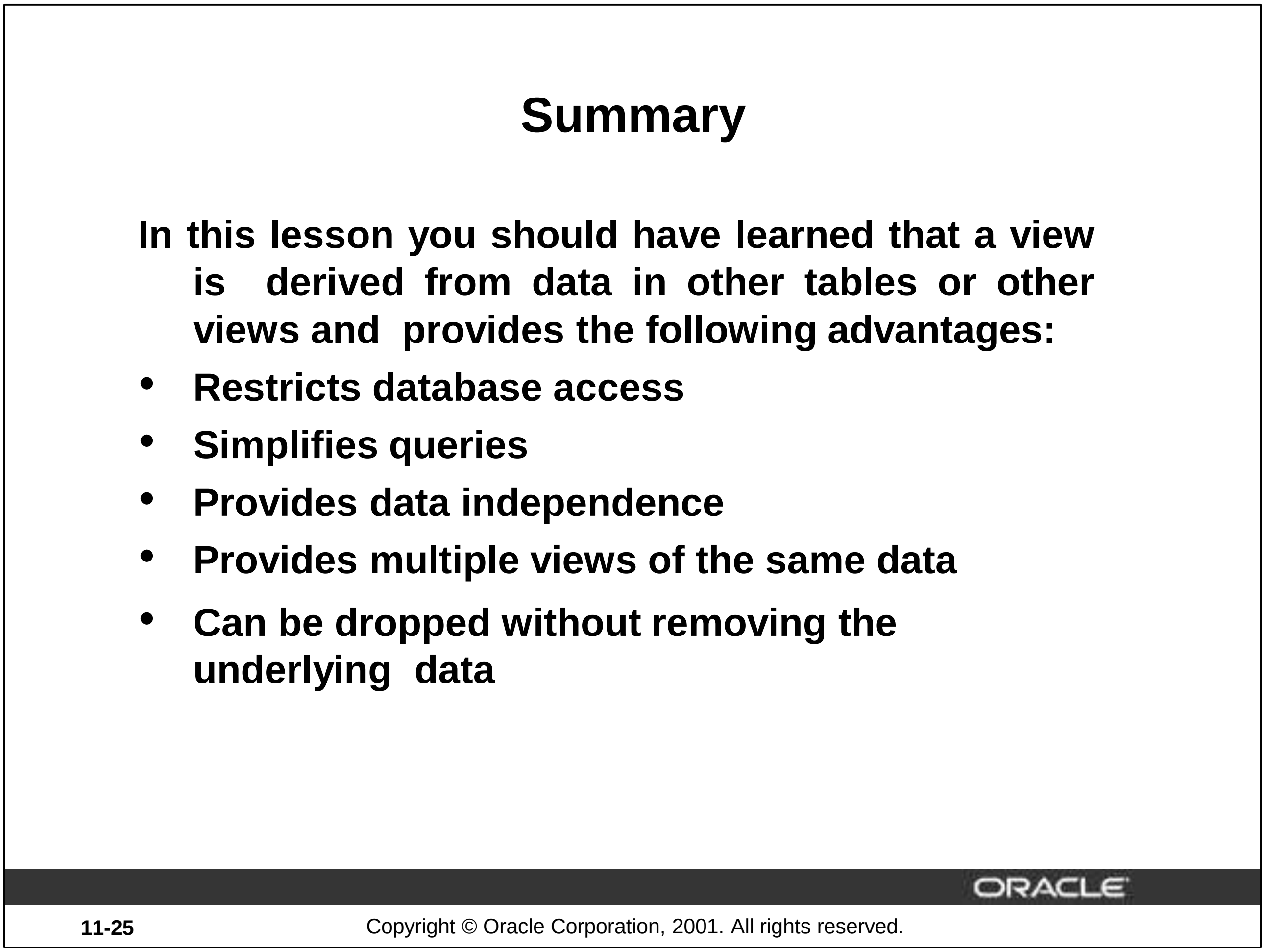
Example of Top-*n* Analysis

To display the top three earner names and salaries from the `EMPLOYEES` table.

```
SELECT ROWNUM as RANK, last_name, salary
FROM   (SELECT last_name,salary FROM employees
       ORDER BY salary DESC)
WHERE ROWNUM <= 3;
```



RANK	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000

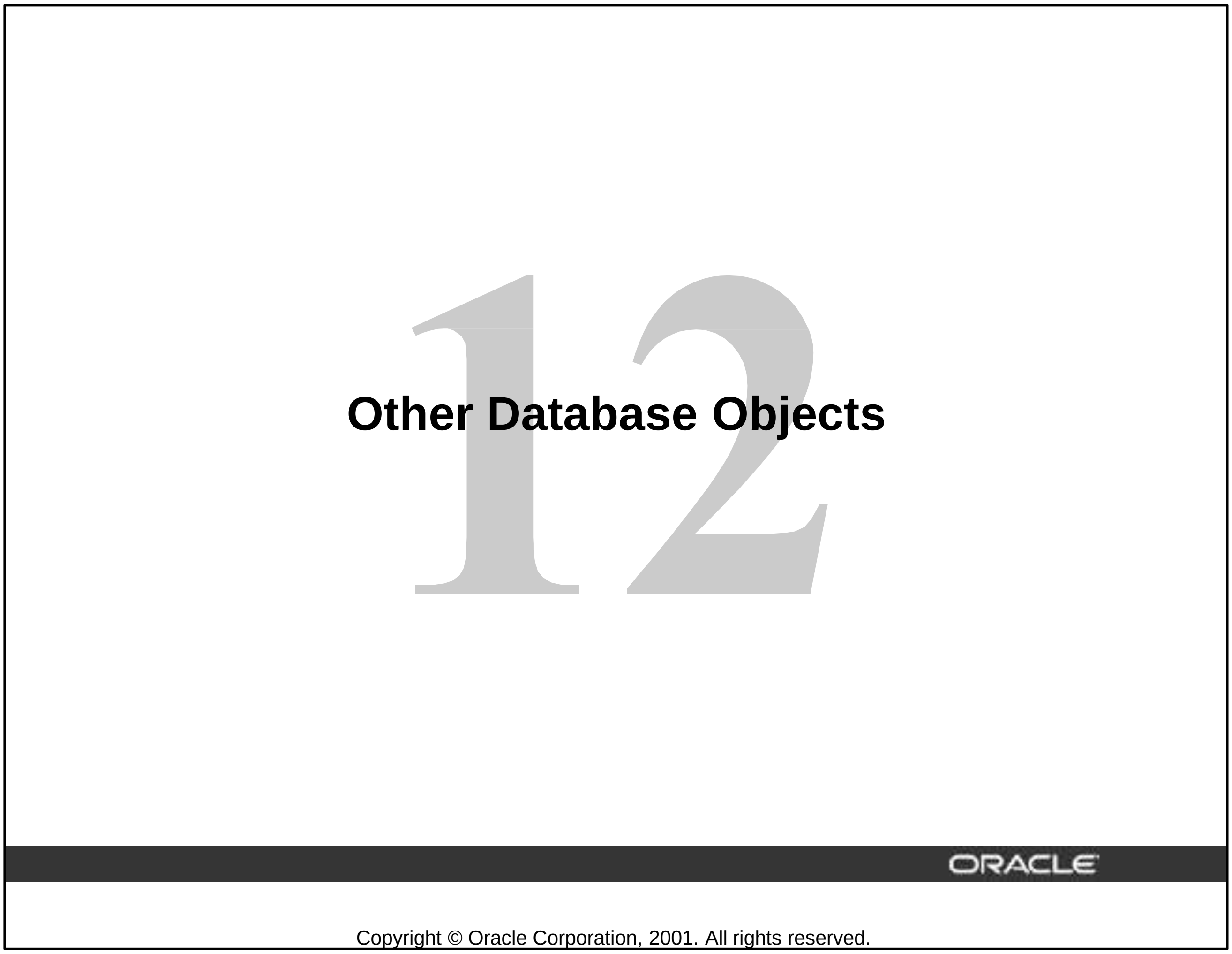


Summary

In this lesson you should have learned that a view is derived from data in other tables or other views and provides the following advantages:

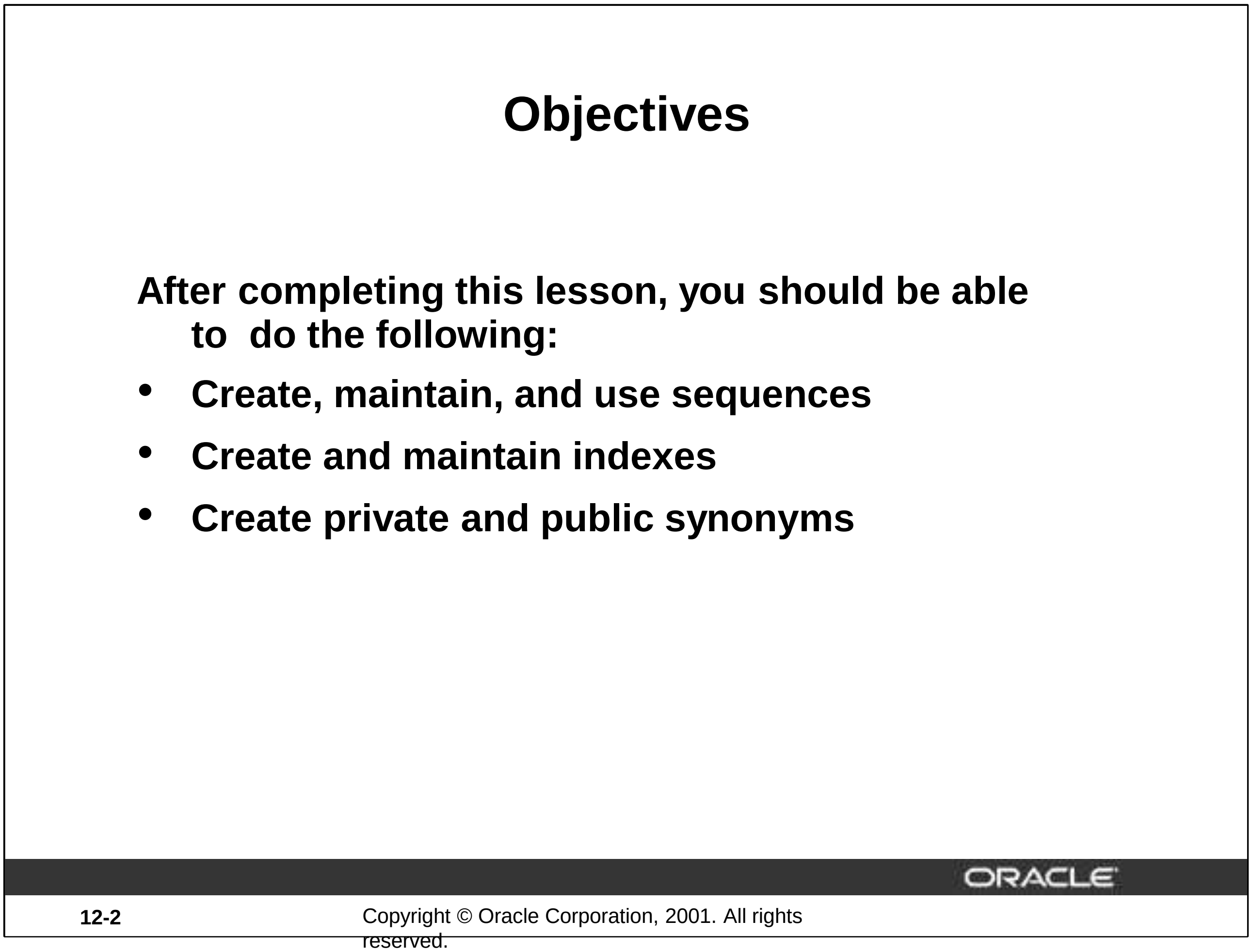
- Restricts database access
- Simplifies queries
- Provides data independence
- Provides multiple views of the same data
- Can be dropped without removing the underlying data

11-25



12

Other Database Objects

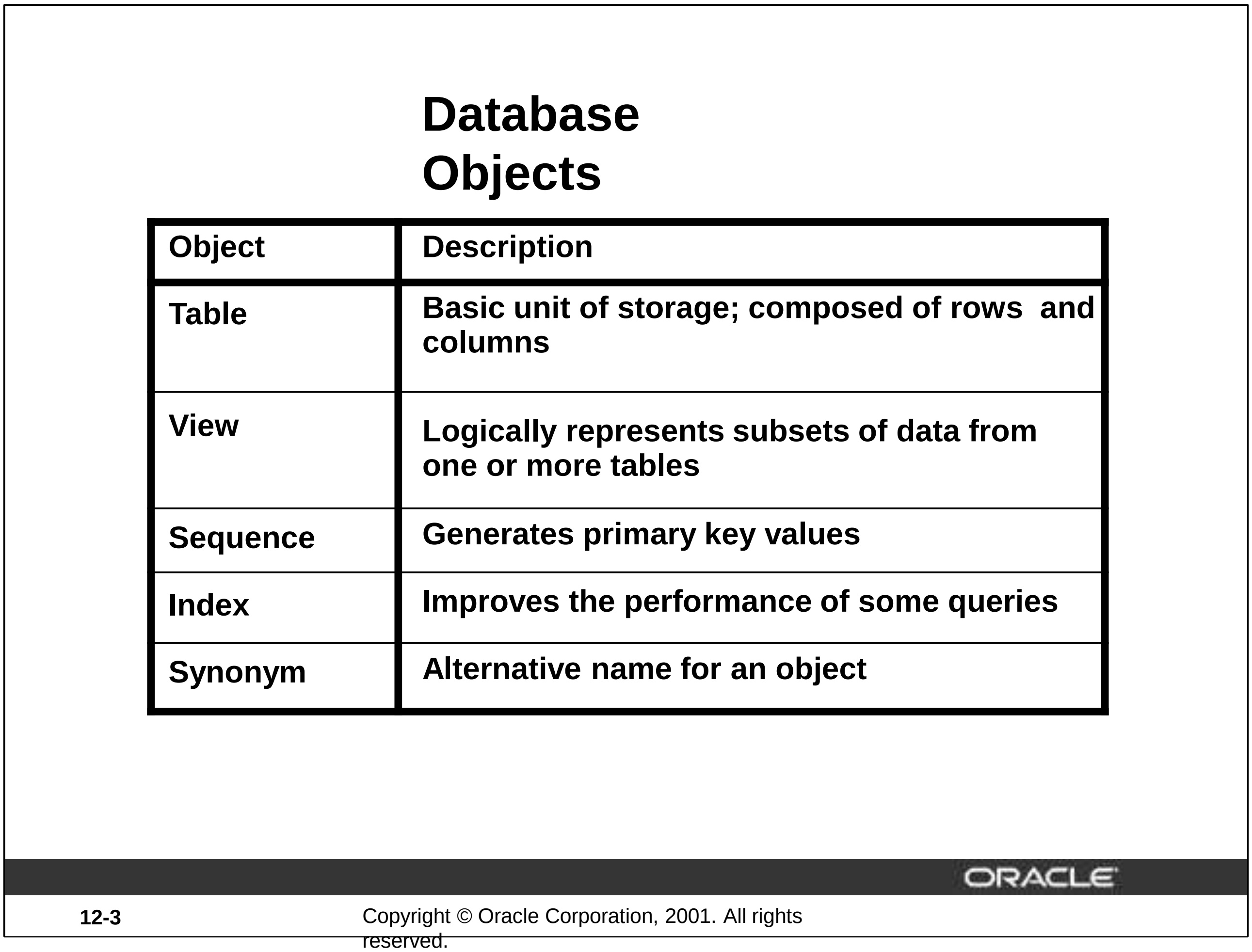


Objectives

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

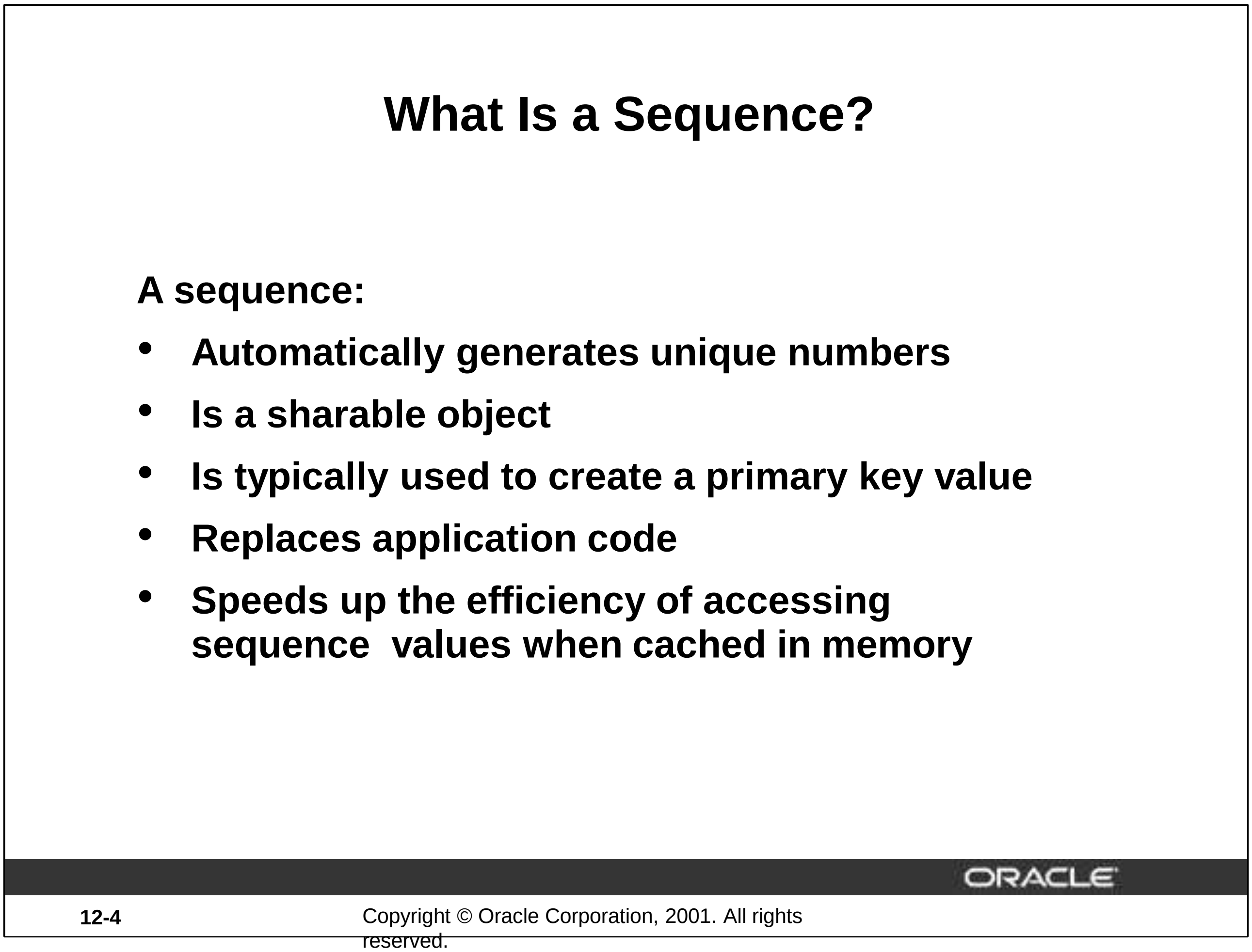
- Create, maintain, and use sequences
- Create and maintain indexes
- Create private and public synonyms

12-2



Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Generates primary key values
Index	Improves the performance of some queries
Synonym	Alternative name for an object

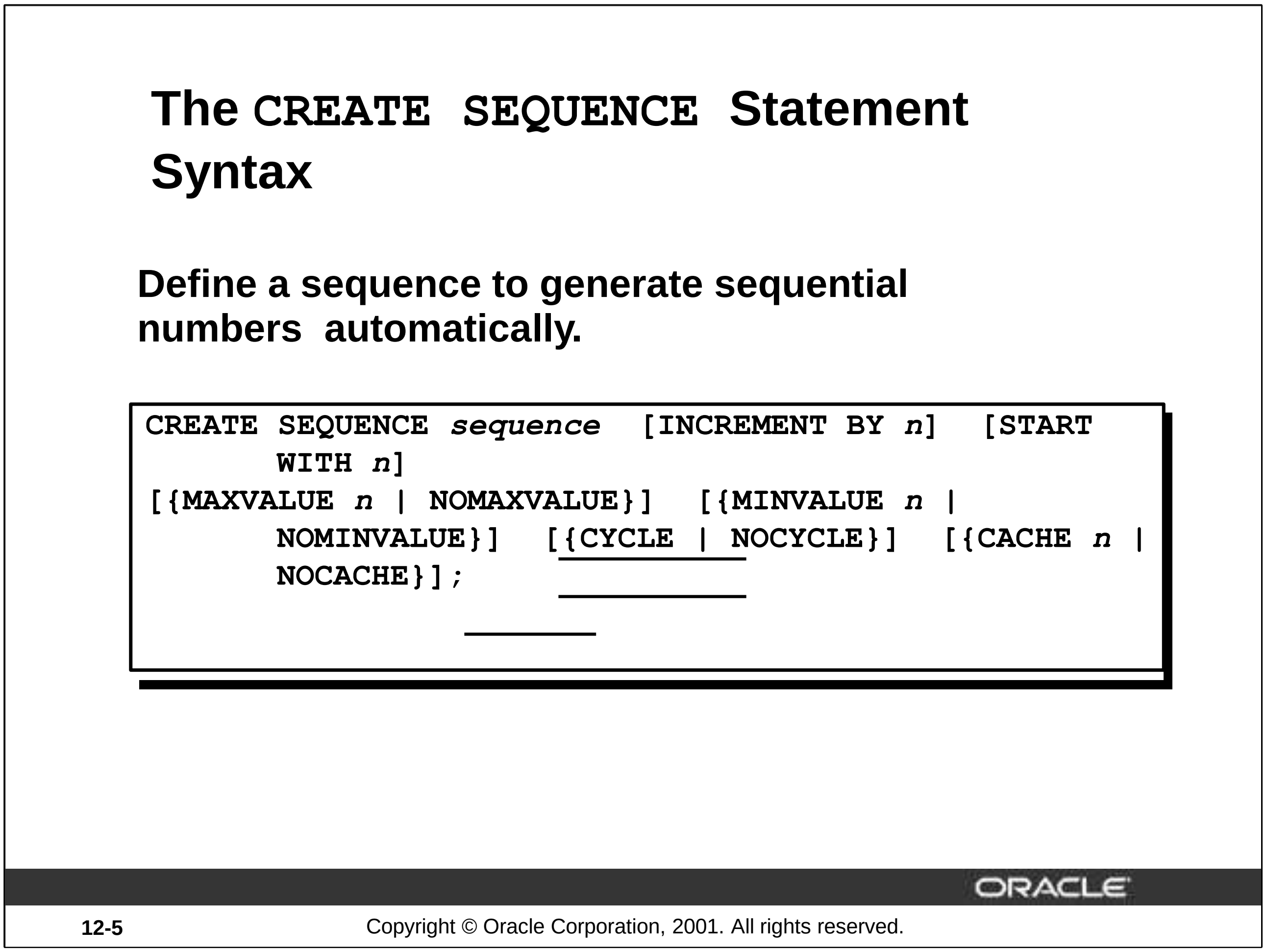


What Is a Sequence?

A sequence:

- Automatically generates unique numbers
- Is a sharable object
- Is typically used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory

12-4



The `CREATE SEQUENCE` Statement Syntax

Define a sequence to generate sequential numbers automatically.

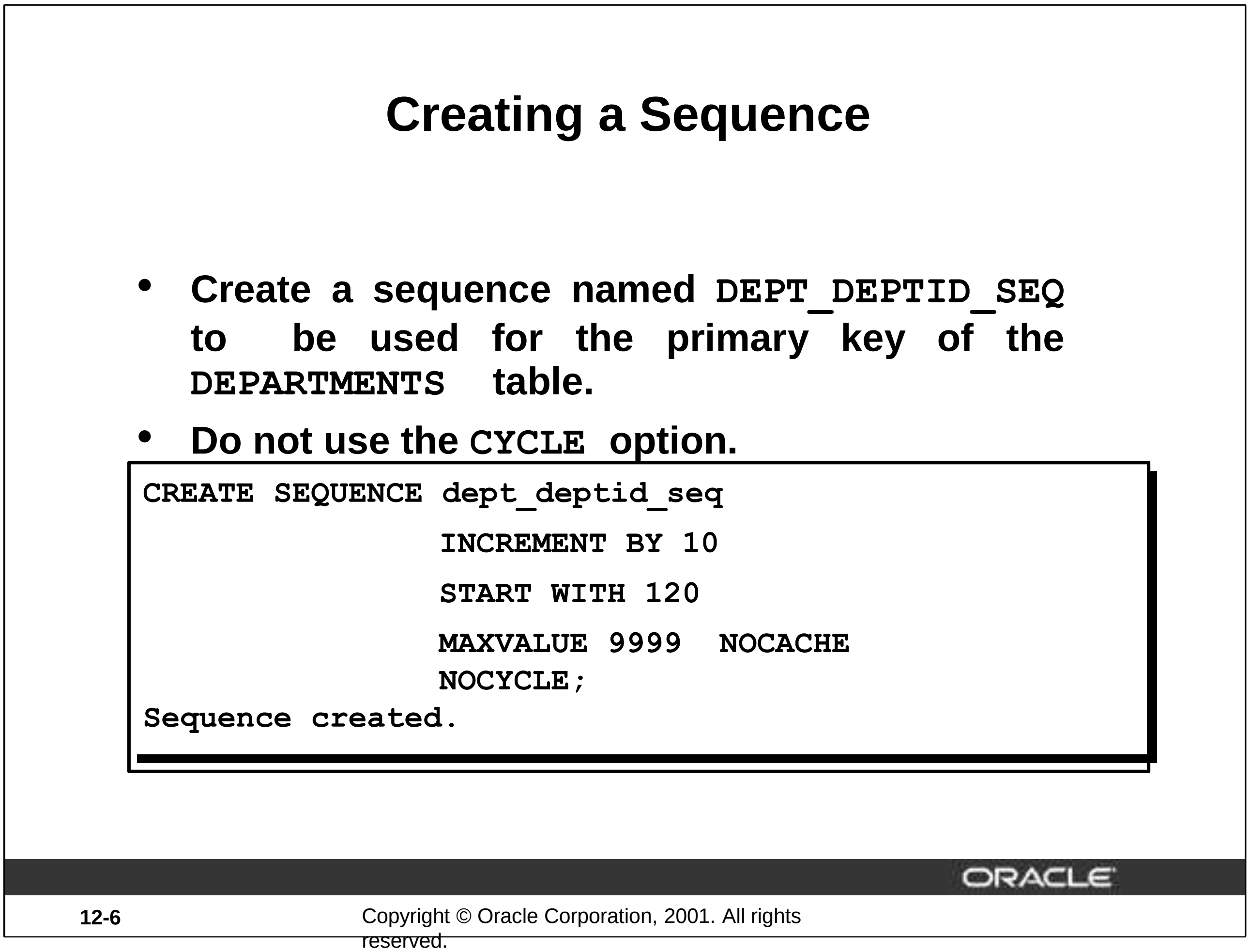
```
CREATE SEQUENCE sequence  [INCREMENT BY n]  [START
       WITH n]
[{MAXVALUE n | NOMAXVALUE}]  [{MINVALUE n |
       NOMINVALUE}]  [{CYCLE | NOCYCLE}]  [{CACHE n |
       NOCACHE}];
```

12-5

Creating a Sequence

Automatically generate sequential numbers by using the `CREATE SEQUENCE` statement. In the syntax:

`sequence`	is the name of the sequence generator
`INCREMENT BY` *`n`*	specifies the interval between sequence numbers where *`n`* is an integer (If this clause is omitted, the sequence increments by 1.)
`START WITH` *`n`*	specifies the first sequence number to be generated (If this cla use is omitted, the sequence starts with 1.)
`MAXVALUE` *`n`*	specifies the maximum value the sequence can generate
`NOMAXVALUE`	specifies a maximum value of 10^27 for an ascending sequence and –1 for a descending sequence (This is the default option.)
`MINVALUE` *`n`*	specifies the minimum sequence value
`NOMINVALUE`	specifies a minimum value of 1 for an ascending sequence and – (10^26) for a descending sequence (This is the default option.)
`CYCLE | NOCYCLE`	specifies whether the sequence continues to generate values afte r reaching its maximum or minimum value (`NOCYCLE` is the default option.)
`CACHE` *`n`* `| NOCACHE`	specifies how many values the Oracle Server preallocates and keep in memory (By default, the Oracle Server caches 20 values.)

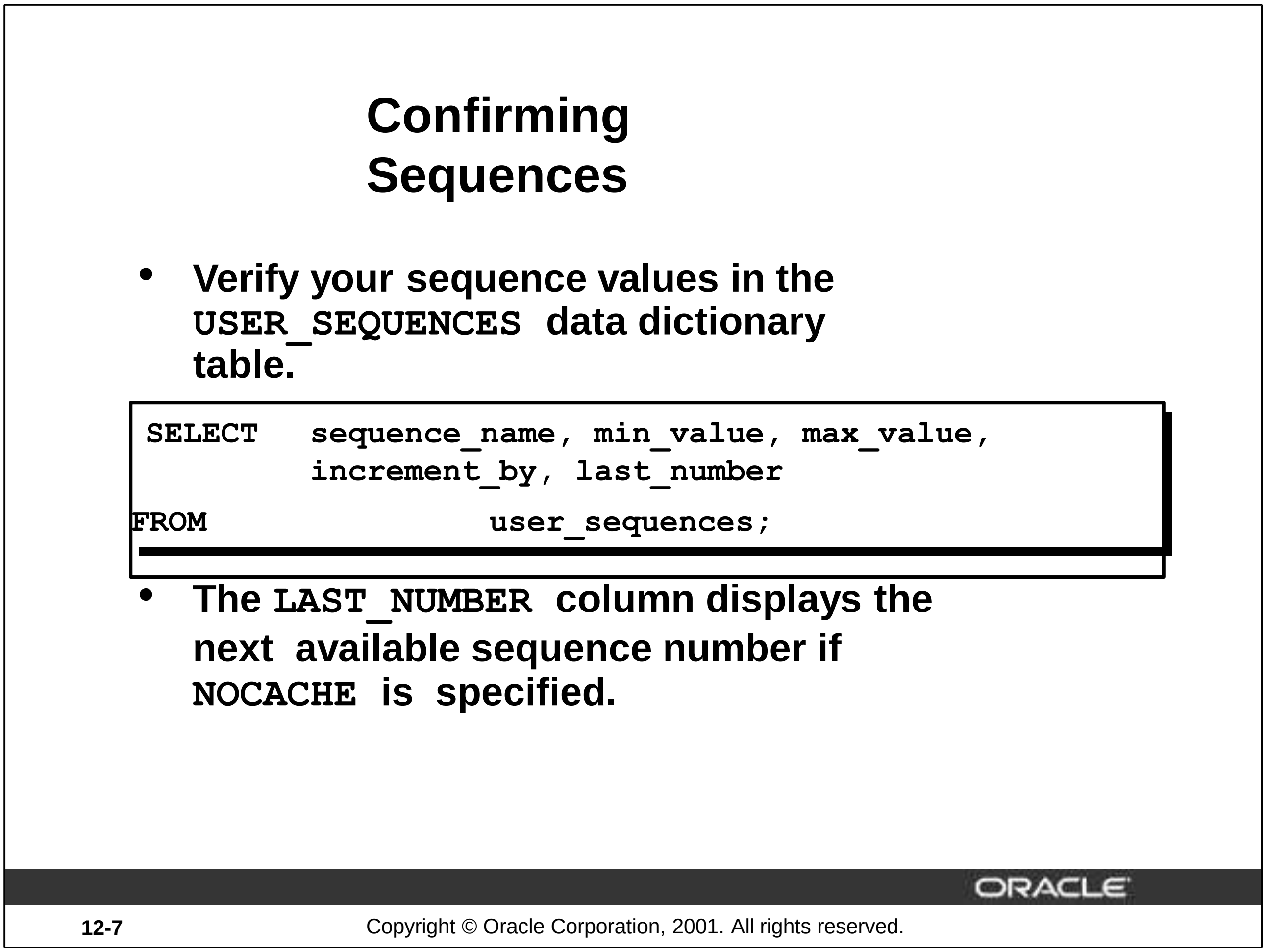


Creating a Sequence

- Create a sequence named `DEPT_DEPTID_SEQ` to be used for the primary key of the `DEPARTMENTS` table.
- Do not use the `CYCLE` option.

```
CREATE SEQUENCE dept_deptid_seq
                INCREMENT BY 10
                START WITH 120
                MAXVALUE 9999  NOCACHE
                NOCYCLE;
Sequence created.
```

12-6



Confirming Sequences

- Verify your sequence values in the `USER_SEQUENCES` data dictionary table.

```
SELECT   sequence_name, min_value, max_value,
         increment_by, last_number
FROM                   user_sequences;
```

- The `LAST_NUMBER` column displays the next available sequence number if `NOCACHE` is specified.

12-7

NEXTVAL and CURRVAL Pseudocolumns

- NEXTVAL returns the next available sequence value.

  It returns a unique value every time it is referenced, even for different users.
- CURRVAL obtains the current sequence value.
- NEXTVAL must be issued for that sequence before

  CURRVAL contains a value.

Using a Sequence

- **Insert a new department named “Support” in location ID 2500.**

```
INSERT INTO departments(department_id,
            department_name, location_id)  VALUES
       (dept_deptid_seq.NEXTVAL,
                        'Support', 2500);
            1 row created.
```

- **View the current value for the `DEPT_DEPTID_SEQ` sequence.**

```
SELECT   dept_deptid_seq.CURRVAL  FROMdual;
```

ORACLE

Using a Sequence

- **Caching sequence values in memory gives faster access to those values.**
- **Gaps in sequence values can occur when:**
  - **A rollback occurs**
  - **The system crashes**
  - **A sequence is used in another table**
- **If the sequence was created with `NOCACHE`, view the next available value, by querying the `USER_SEQUENCES` table.**

Modifying a Sequence

Change the increment value, maximum value, minimum value, cycle option, or cache option.

```
ALTER SEQUENCE dept_deptid_seq
               INCREMENT BY 20
               MAXVALUE 999999  NOCACHE  NOCYCLE;
Sequence altered.
```

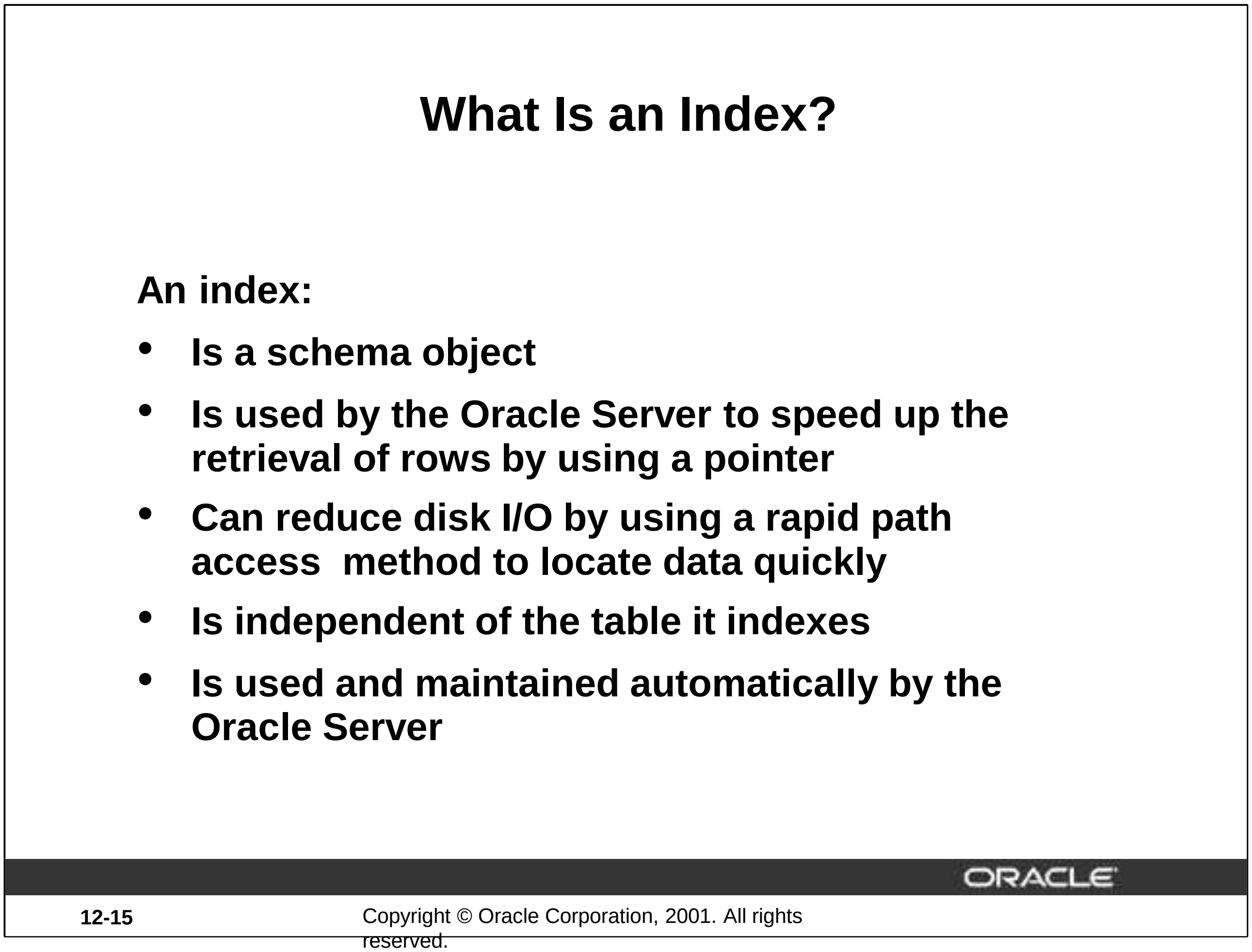
Guidelines for Modifying a Sequence

- You must be the owner or have the `ALTER` privilege for the sequence.
- Only future sequence numbers are affected.
- The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed.

Removing a Sequence

- Remove a sequence from the data dictionary by using the `DROP SEQUENCE` statement.
- Once removed, the sequence can no longer be referenced.

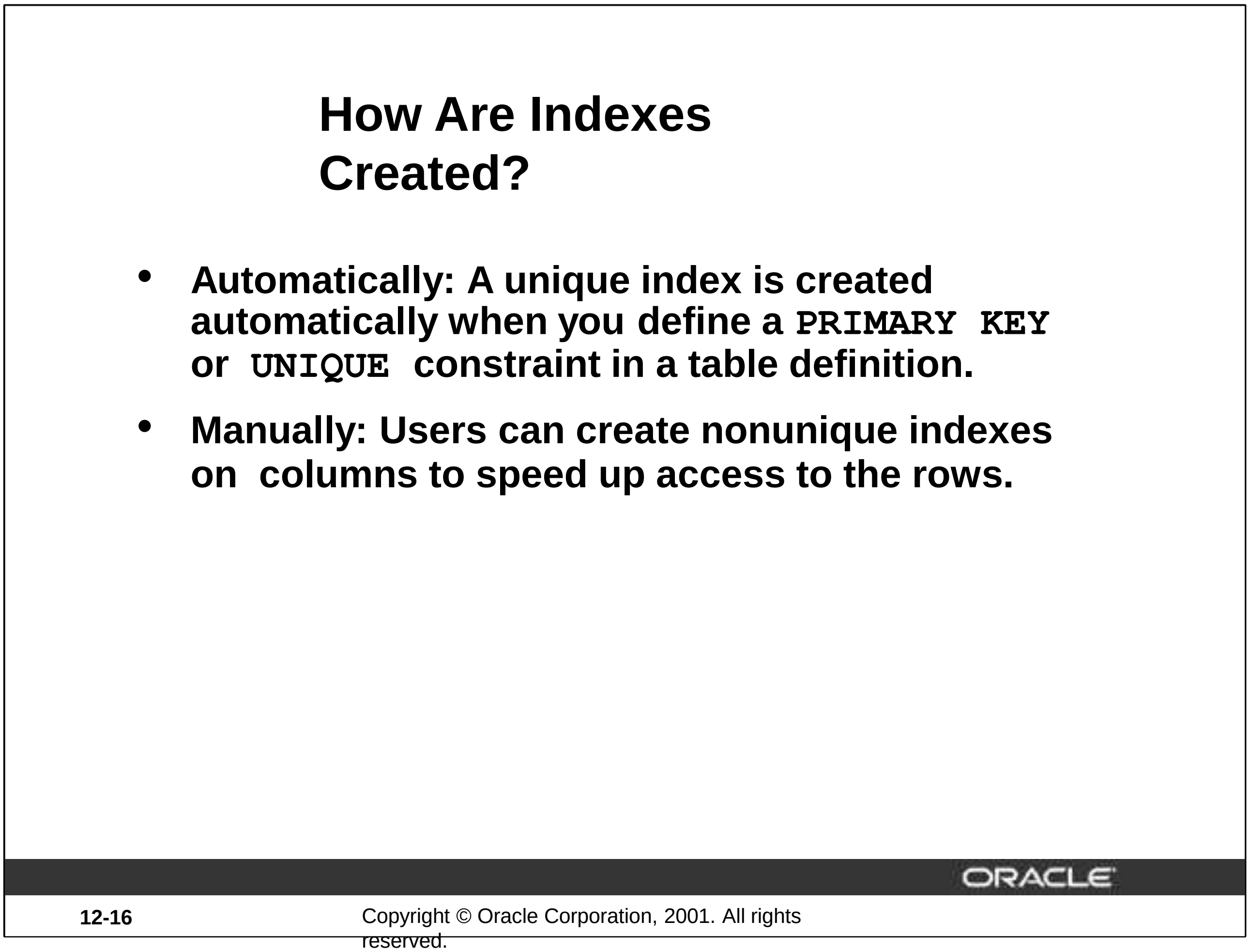
```
DROP SEQUENCE dept_deptid_seq;  Sequence dropped.
```



What Is an Index?

An index:

- Is a schema object
- Is used by the Oracle Server to speed up the retrieval of rows by using a pointer
- Can reduce disk I/O by using a rapid path access method to locate data quickly
- Is independent of the table it indexes
- Is used and maintained automatically by the Oracle Server



How Are Indexes Created?

- Automatically: A unique index is created automatically when you define a `PRIMARY KEY` or `UNIQUE` constraint in a table definition.
- Manually: Users can create nonunique indexes on columns to speed up access to the rows.

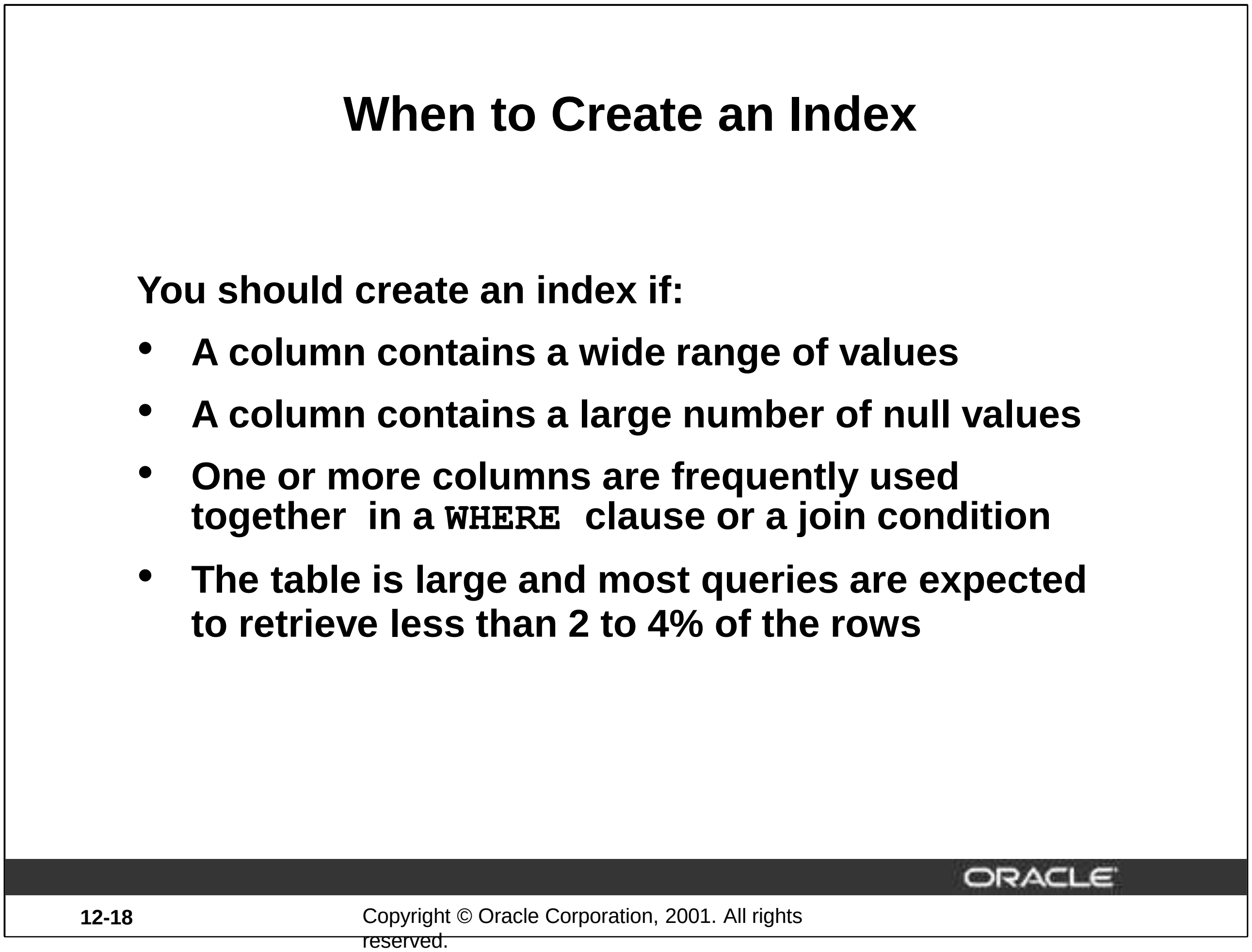
Creating an Index

- Create an index on one or more columns.

```
CREATE INDEX index
ON table (column[, column]...);
```

- Improve the speed of query access to the `LAST_NAME` column in the `EMPLOYEES` table.

```
CREATE INDEX emp_last_name_idx
ON     employees(last_name);  Index created.
```

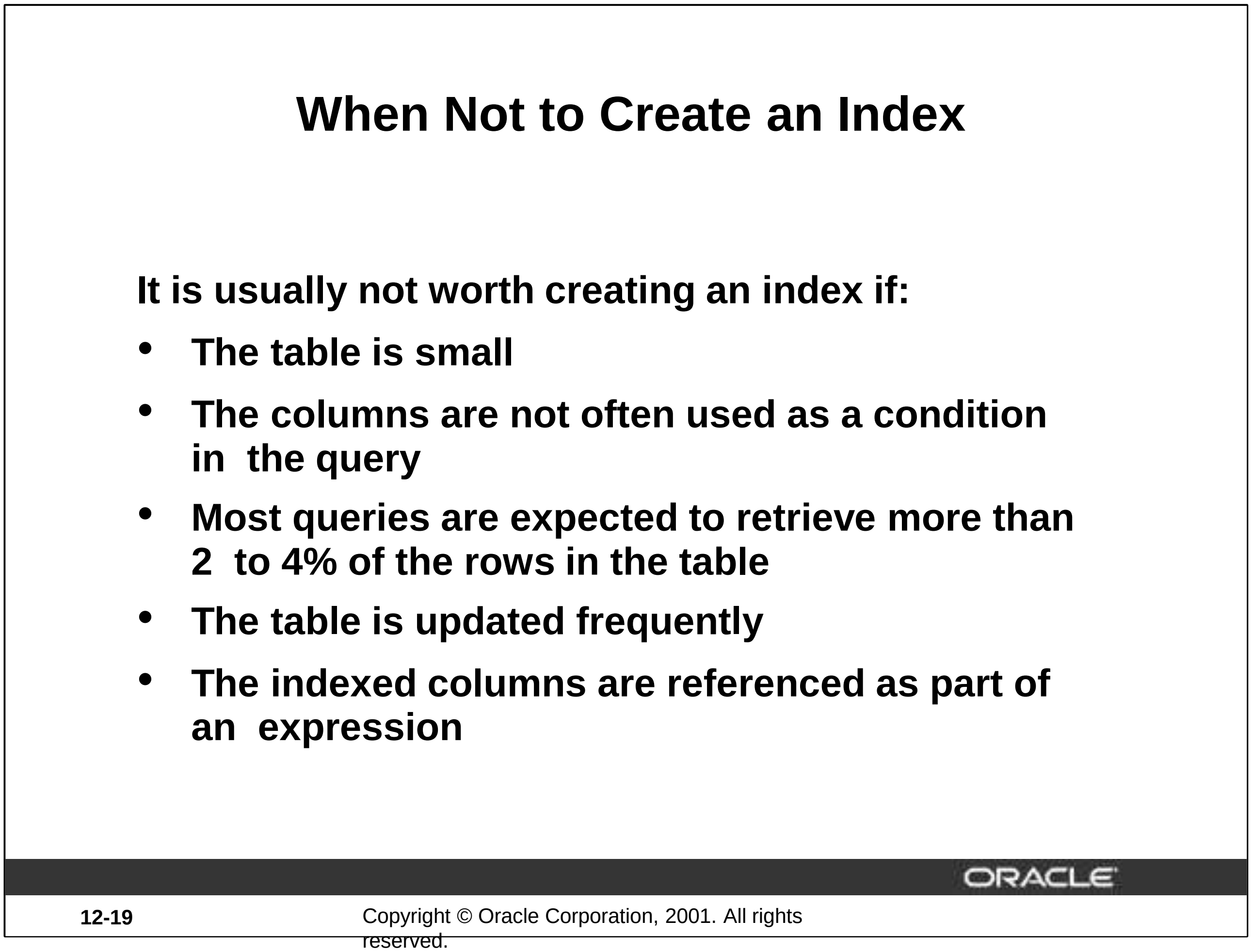


When to Create an Index

You should create an index if:

- A column contains a wide range of values
- A column contains a large number of null values
- One or more columns are frequently used together in a `WHERE` clause or a join condition
- The table is large and most queries are expected to retrieve less than 2 to 4% of the rows

12-18



When Not to Create an Index

It is usually not worth creating an index if:

- The table is small
- The columns are not often used as a condition in the query
- Most queries are expected to retrieve more than 2 to 4% of the rows in the table
- The table is updated frequently
- The indexed columns are referenced as part of an expression

Confirming Indexes

- **The `USER_INDEXES` data dictionary view contains the name of the index and its uniqueness.**
- **The `USER_IND_COLUMNS` view contains the index name, the table name, and the column name.**

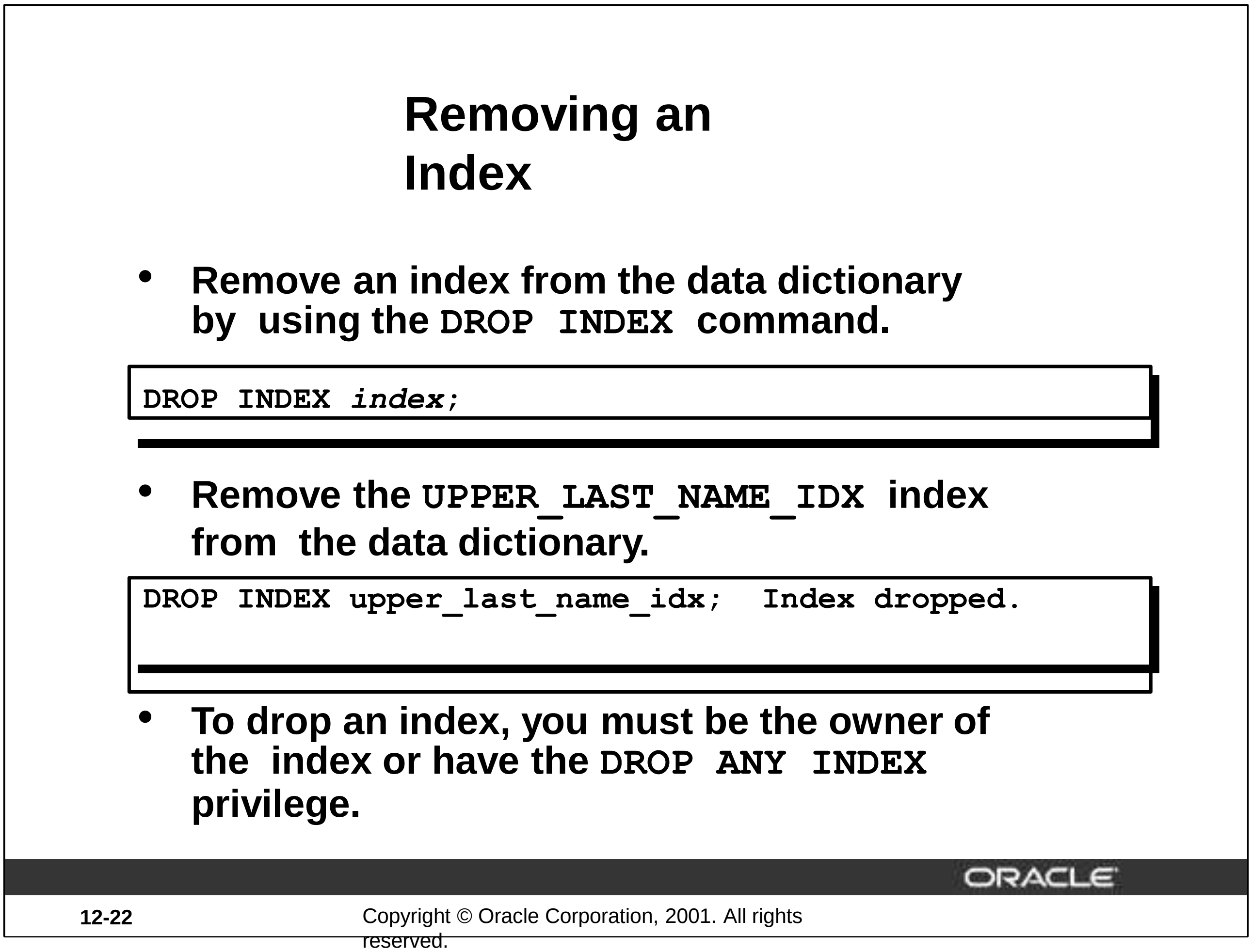
```
 SELECT  ic.index_name, ic.column_name,
         ic.column_position col_pos,ix.uniqueness
FROM                 user_indexes ix, user_ind_columns
 ic  WHERE           ic.index_name = ix.index_name
AND                  ic.table_name = 'EMPLOYEES';
```

Function-Based Indexes

- A function-based index is an index based on expressions.
- The index expression is built from table columns, constants, SQL functions, and user-defined functions.

```
CREATE INDEX upper_dept_name_idx
ON departments(UPPER(department_name));

Index created.  SELECT *
FROM   departments
WHERE  UPPER(department_name) = 'SALES';
```



Removing an Index

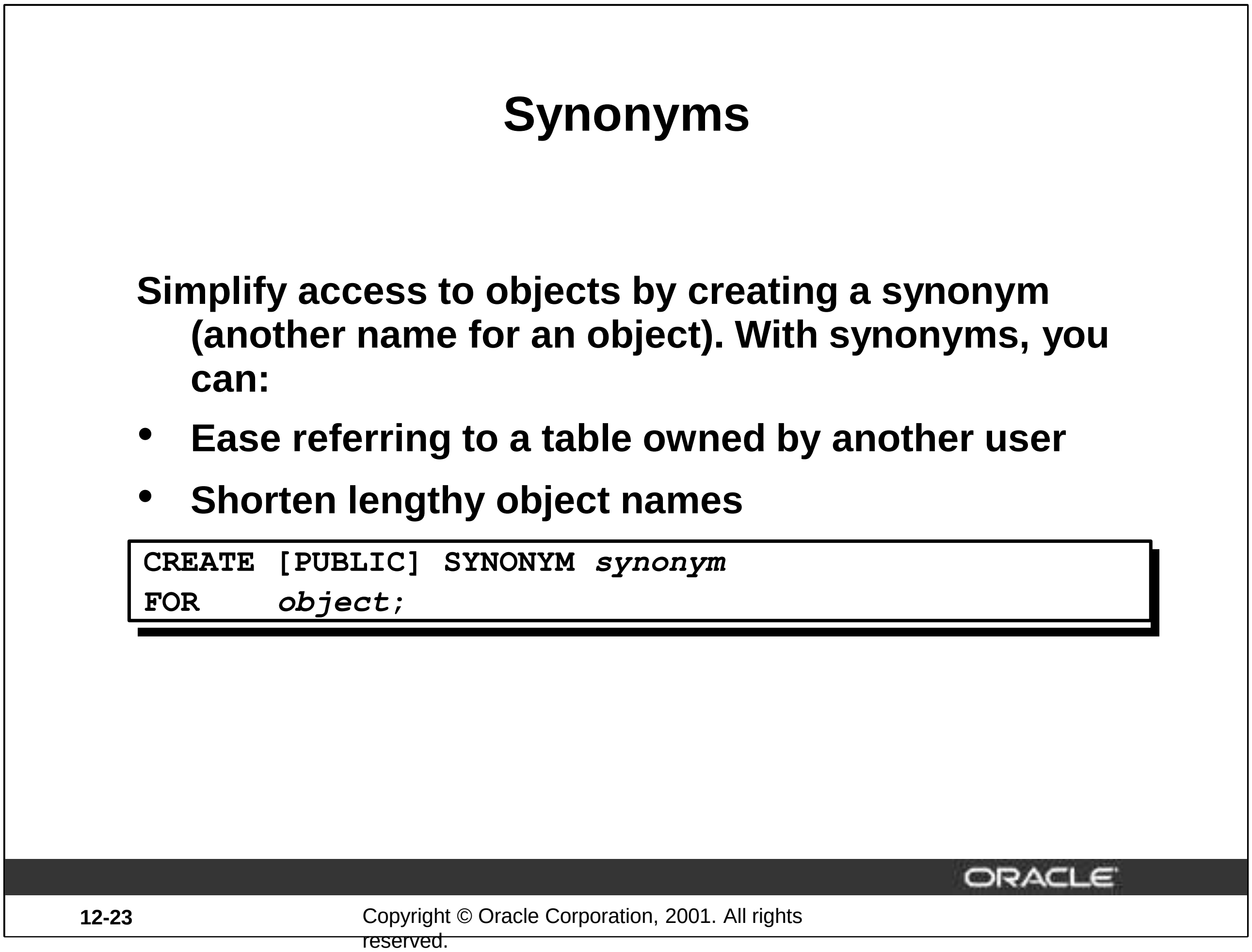
- Remove an index from the data dictionary by using the `DROP INDEX` command.

```
DROP INDEX index;
```

- Remove the `UPPER_LAST_NAME_IDX` index from the data dictionary.

```
DROP INDEX upper_last_name_idx;  Index dropped.
```

- To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.



Synonyms

Simplify access to objects by creating a synonym (another name for an object). With synonyms, you can:

- Ease referring to a table owned by another user
- Shorten lengthy object names

```
CREATE [PUBLIC] SYNONYM synonym
FOR    object;
```

12-23

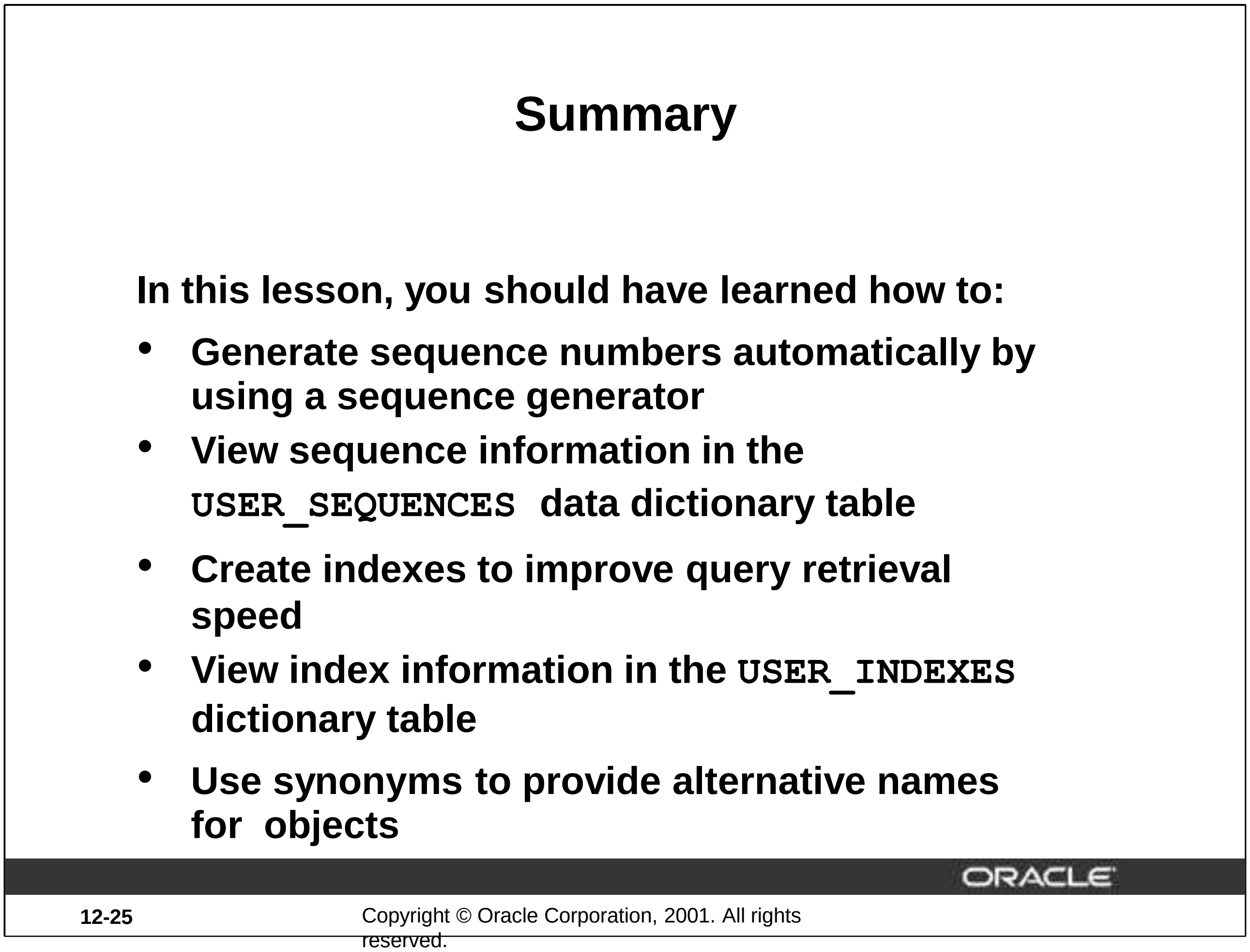
Creating and Removing Synonyms

- Create a shortened name for the DEPT_SUM_VU view.

```
CREATE SYNONYM   d_sum
FOR              dept_sum_vu;  Synonym Created.
```

- Drop a

```
DROP SYNONYM d_sum;
Synonym dropped.
```

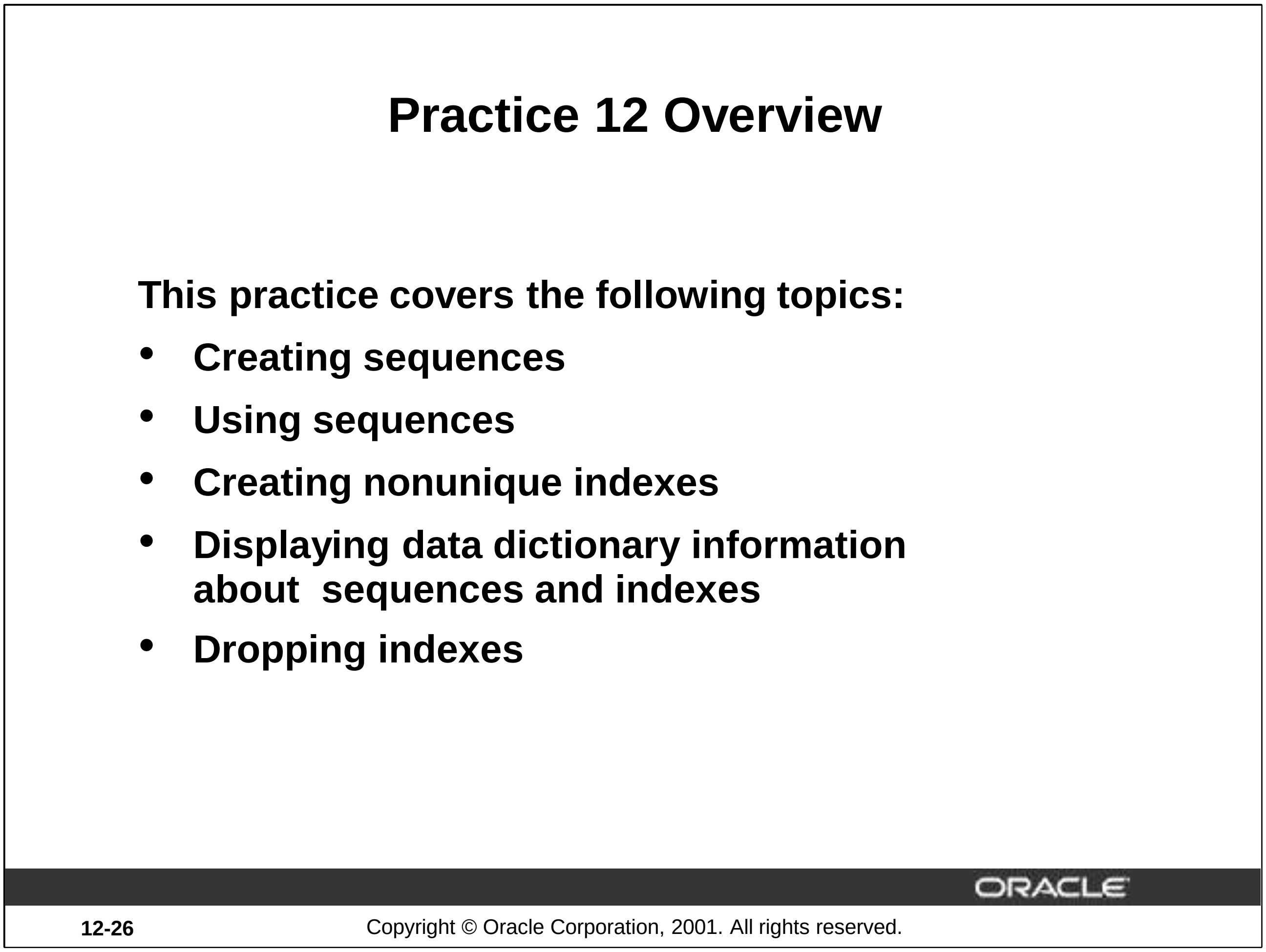


Summary

In this lesson, you should have learned how to:

- Generate sequence numbers automatically by using a sequence generator
- View sequence information in the `USER_SEQUENCES` data dictionary table
- Create indexes to improve query retrieval speed
- View index information in the `USER_INDEXES` dictionary table
- Use synonyms to provide alternative names for objects

12-25



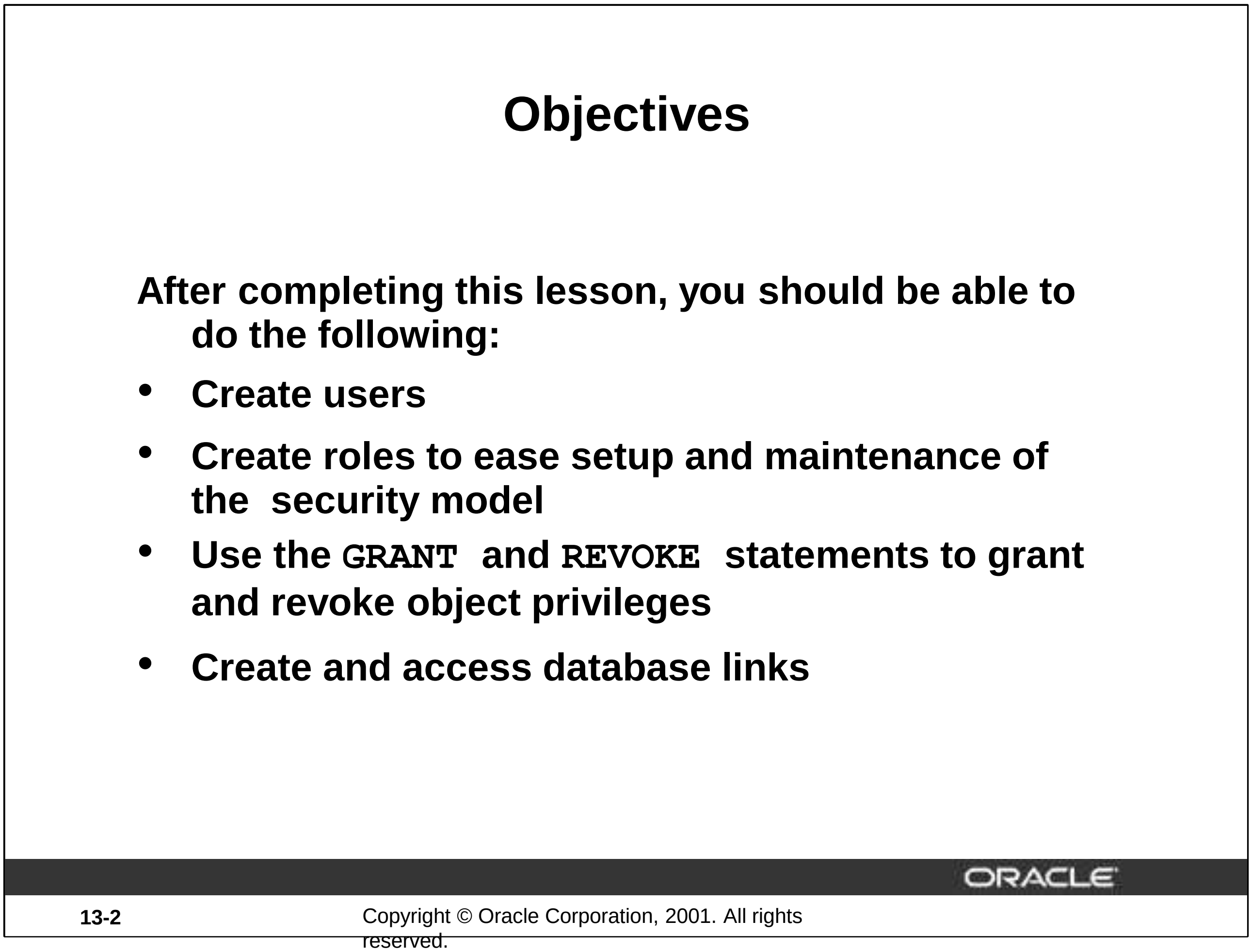
Practice 12 Overview

This practice covers the following topics:

- Creating sequences
- Using sequences
- Creating nonunique indexes
- Displaying data dictionary information about sequences and indexes
- Dropping indexes

13

Controlling User Access



Objectives

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

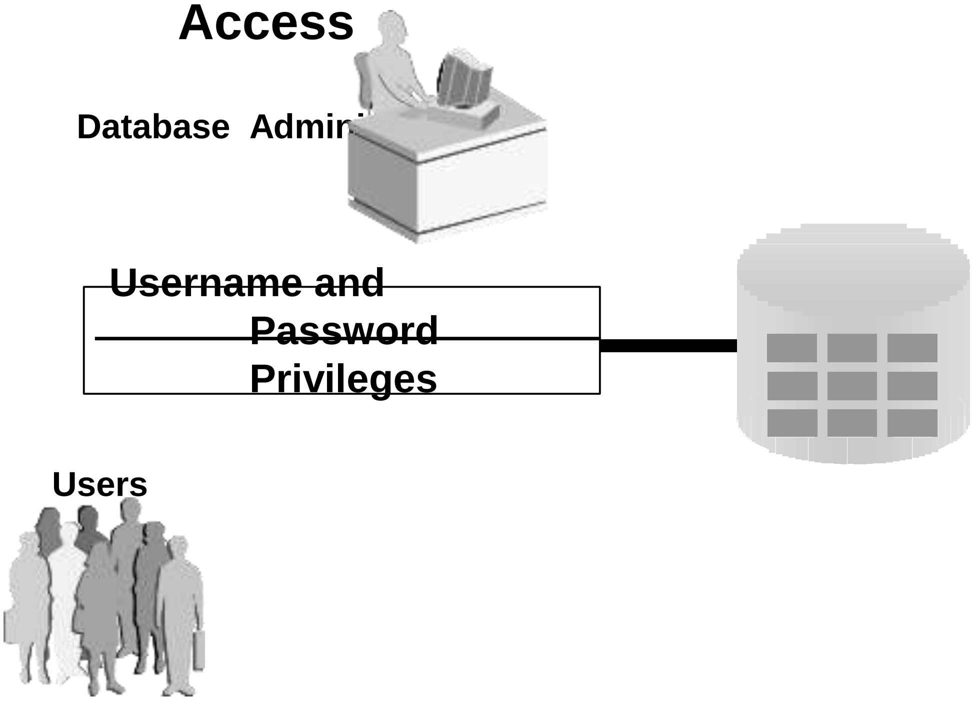
- Create users
- Create roles to ease setup and maintenance of the security model
- Use the `GRANT` and `REVOKE` statements to grant and revoke object privileges
- Create and access database links

13-2

Lesson Aim

In this lesson, you learn how to control database access to specific objects and add new users with different levels of access privileges.

Controlling User Access

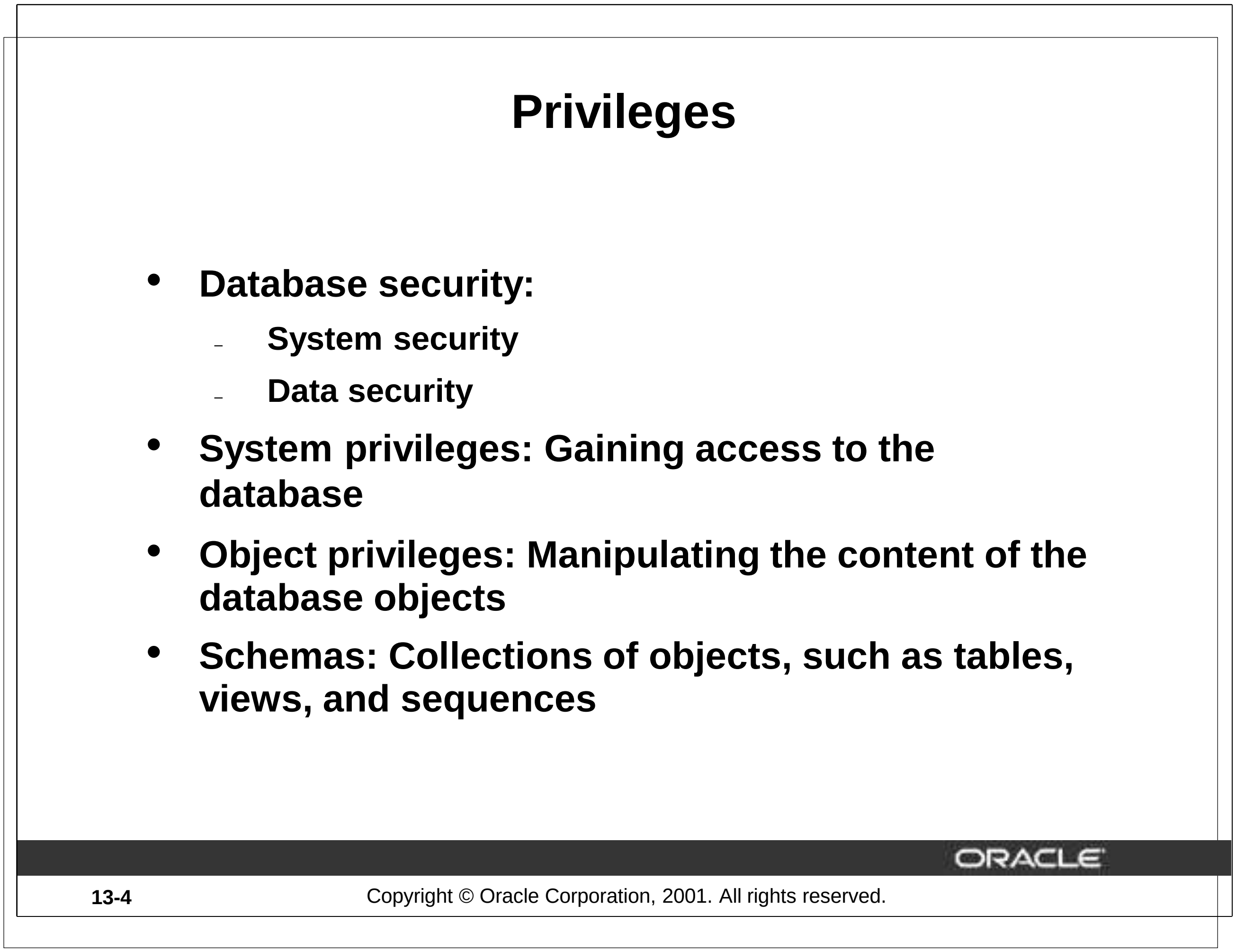


Database Admini

Username and Password

Privileges

Users



Privileges

- **Database security:**
  - **System security**
  - **Data security**
- **System privileges: Gaining access to the database**
- **Object privileges: Manipulating the content of the database objects**
- **Schemas: Collections of objects, such as tables, views, and sequences**

ORACLE

13-4

System Privileges

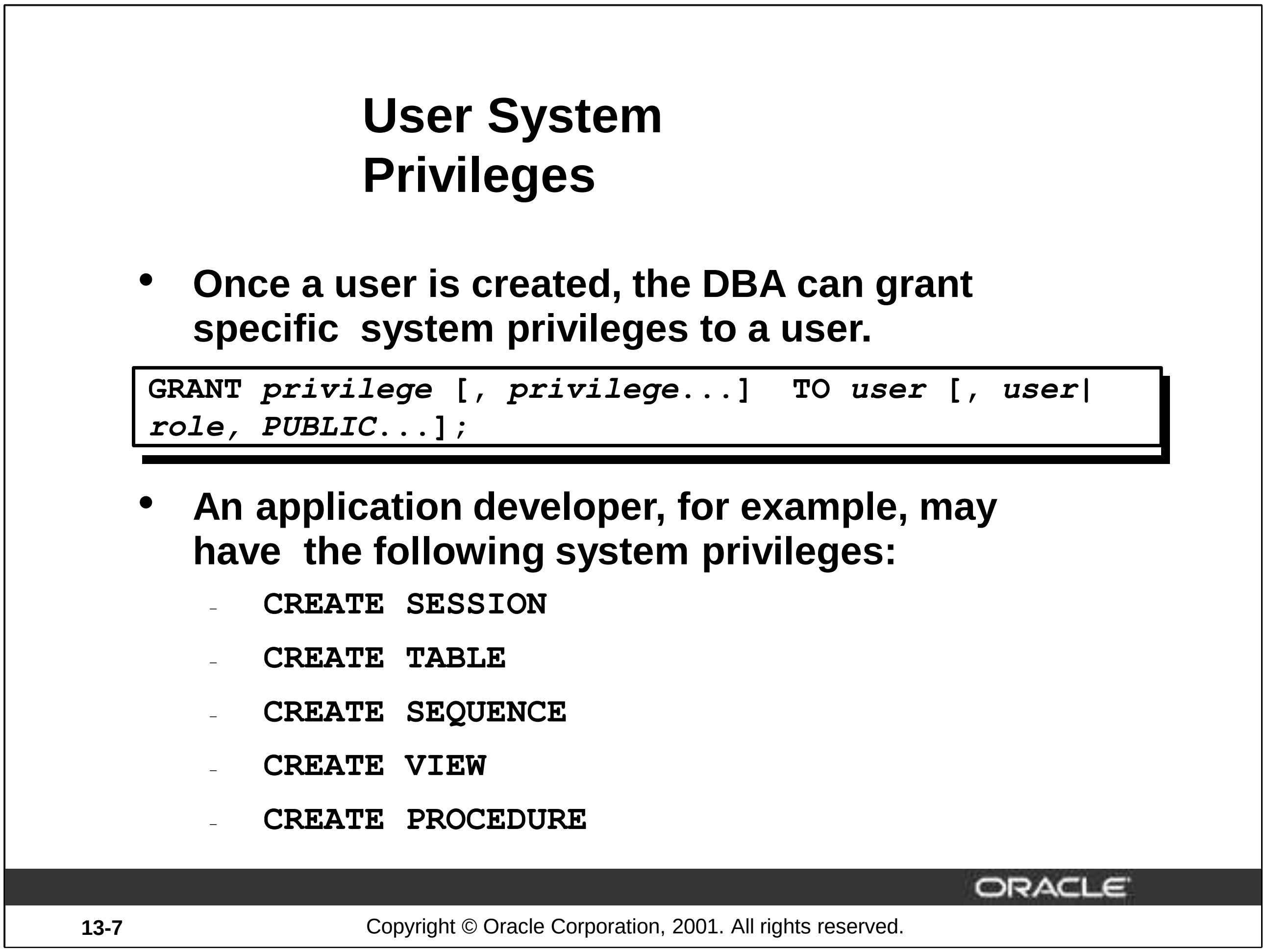
- More than 100 privileges are available.
- The database administrator has high-level system privileges for tasks such as:
  - Creating new users
  - Removing users
  - Removing tables
  - Backing up tables

Creating Users

The DBA creates users by using the `CREATE USER` statement.

```
CREATE USER user
IDENTIFIED BY  password;
```

```
CREATE USER   scott  IDENTIFIED BY             tiger;
User created.
```



User System Privileges

- **Once a user is created, the DBA can grant specific system privileges to a user.**

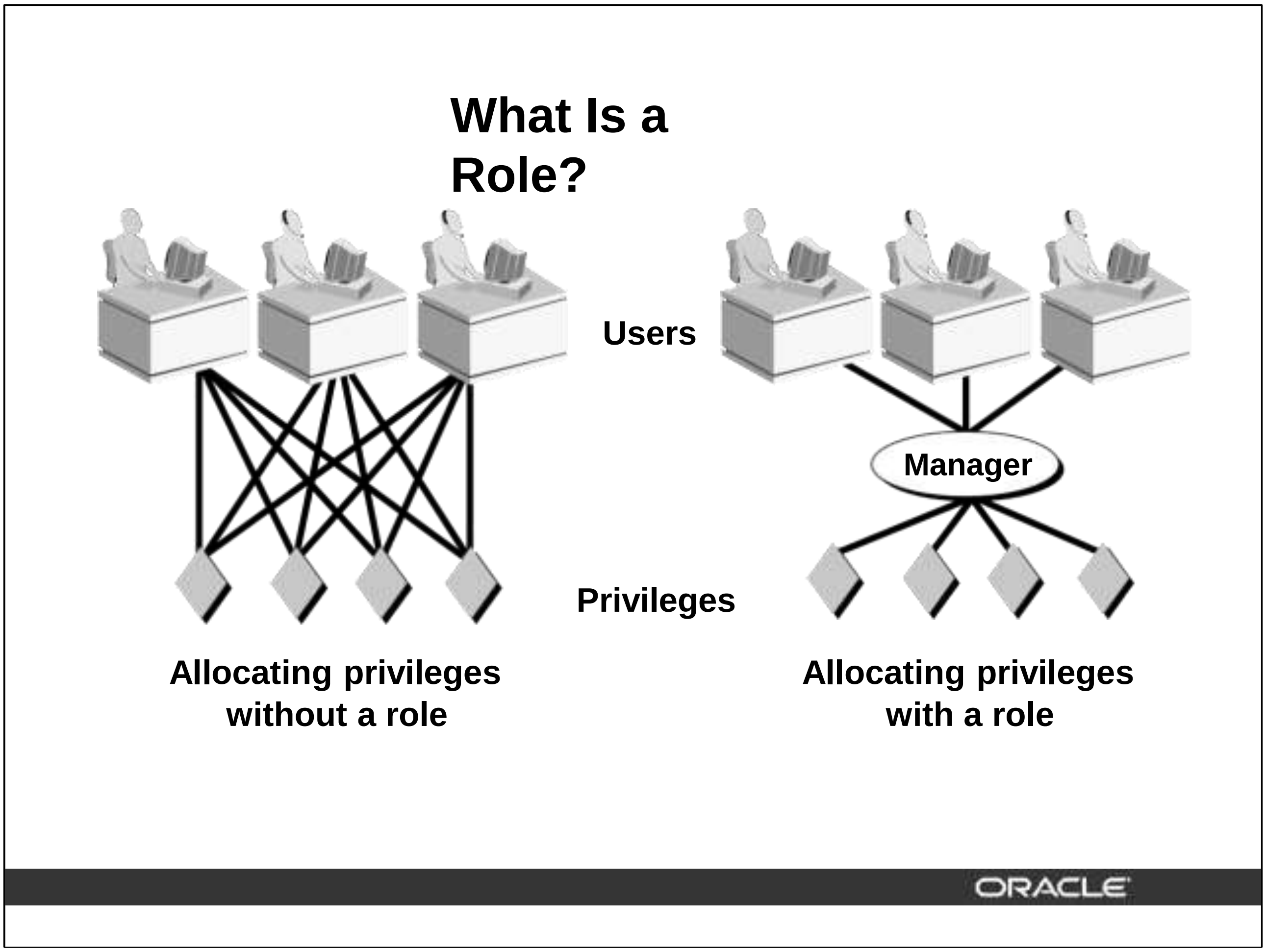
```
GRANT privilege [, privilege...]  TO user [, user|
role, PUBLIC...];
```

- **An application developer, for example, may have the following system privileges:**
  - `CREATE SESSION`
  - `CREATE TABLE`
  - `CREATE SEQUENCE`
  - `CREATE VIEW`
  - `CREATE PROCEDURE`

Granting System Privileges

The DBA can grant a user specific system privileges.

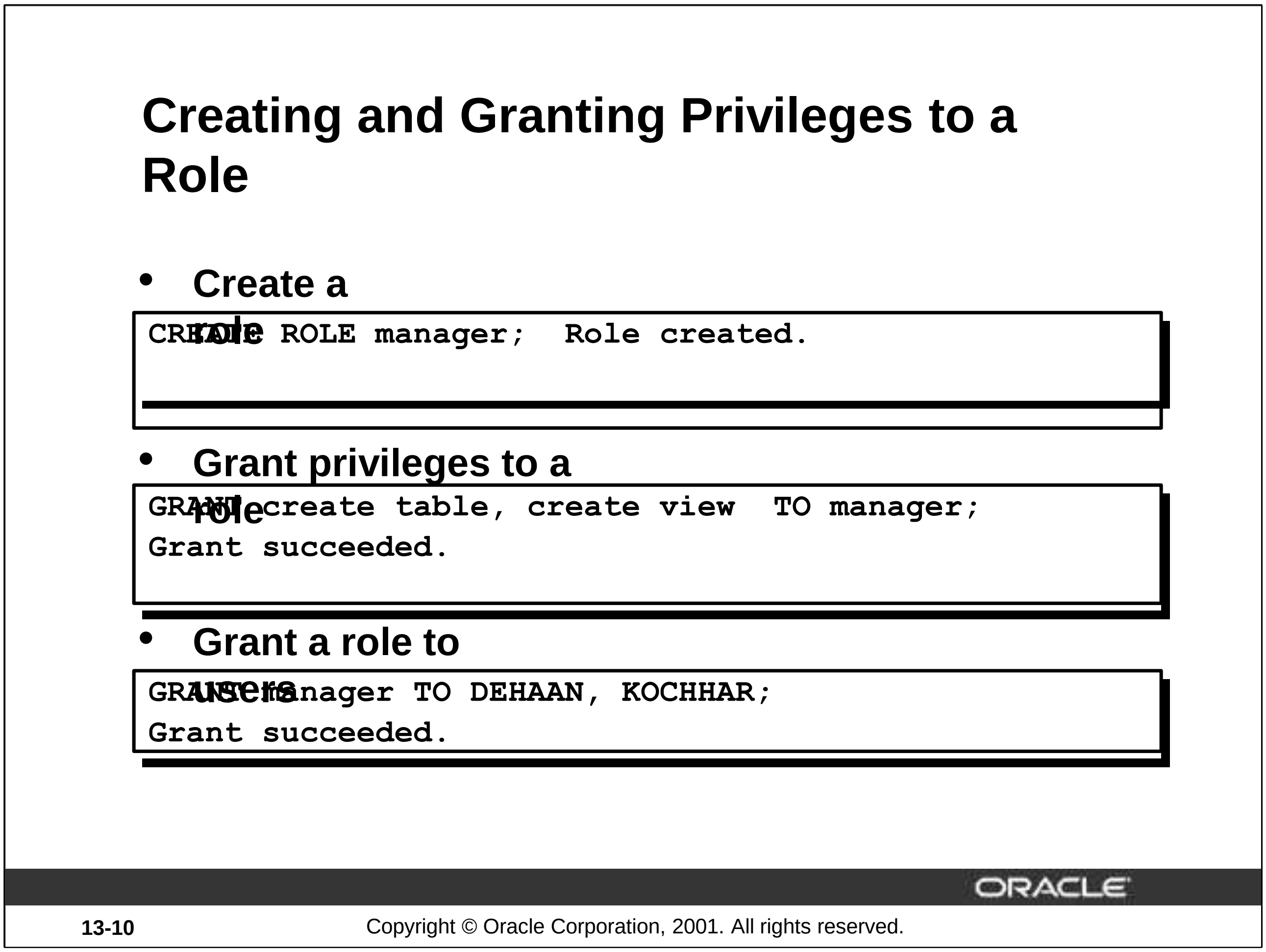
```
GRANT  create session, create table,  create sequence,
       create view
TO             scott;  Grant succeeded.
```



What Is a Role?

Allocating privileges without a role

Allocating privileges with a role



Creating and Granting Privileges to a Role

- Create a role

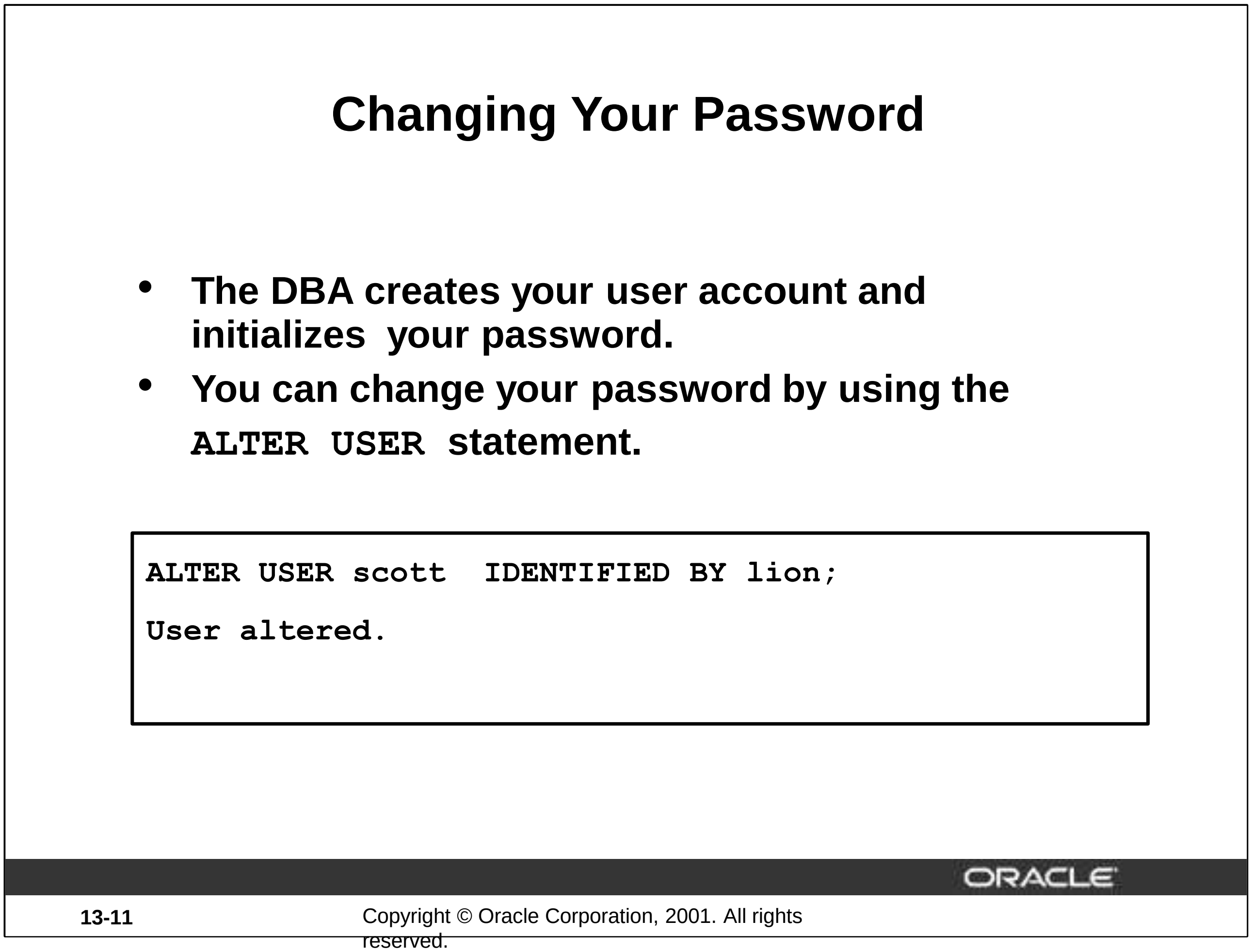
```
CREATE ROLE manager;  Role created.
```

- Grant privileges to a role

```
GRANT create table, create view  TO manager;
Grant succeeded.
```

- Grant a role to users

```
GRANT manager TO DEHAAN, KOCHHAR;
Grant succeeded.
```



Changing Your Password

- The DBA creates your user account and initializes your password.
- You can change your password by using the `ALTER USER` statement.

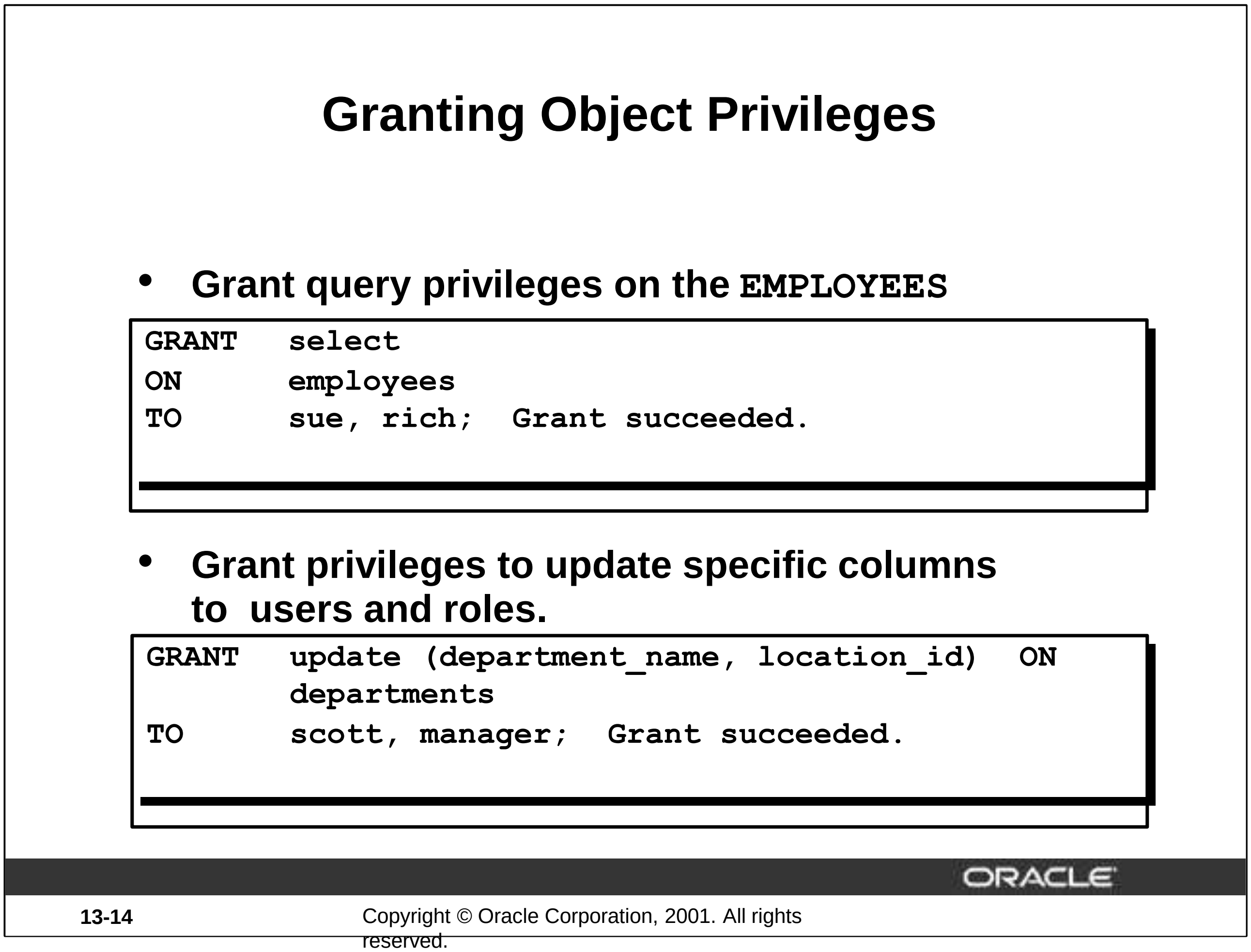
```
ALTER USER scott  IDENTIFIED BY lion;

User altered.
```

Object Privileges

- Object privileges vary from object to object.
- An owner has all the privileges on the object.
- An owner can give specific privileges on that owner’s object.

```
GRANT        object_priv
ON TO        [(columns)]
             object
[WITH GRANT  {user|role|PUBLIC}
OPTION];
```



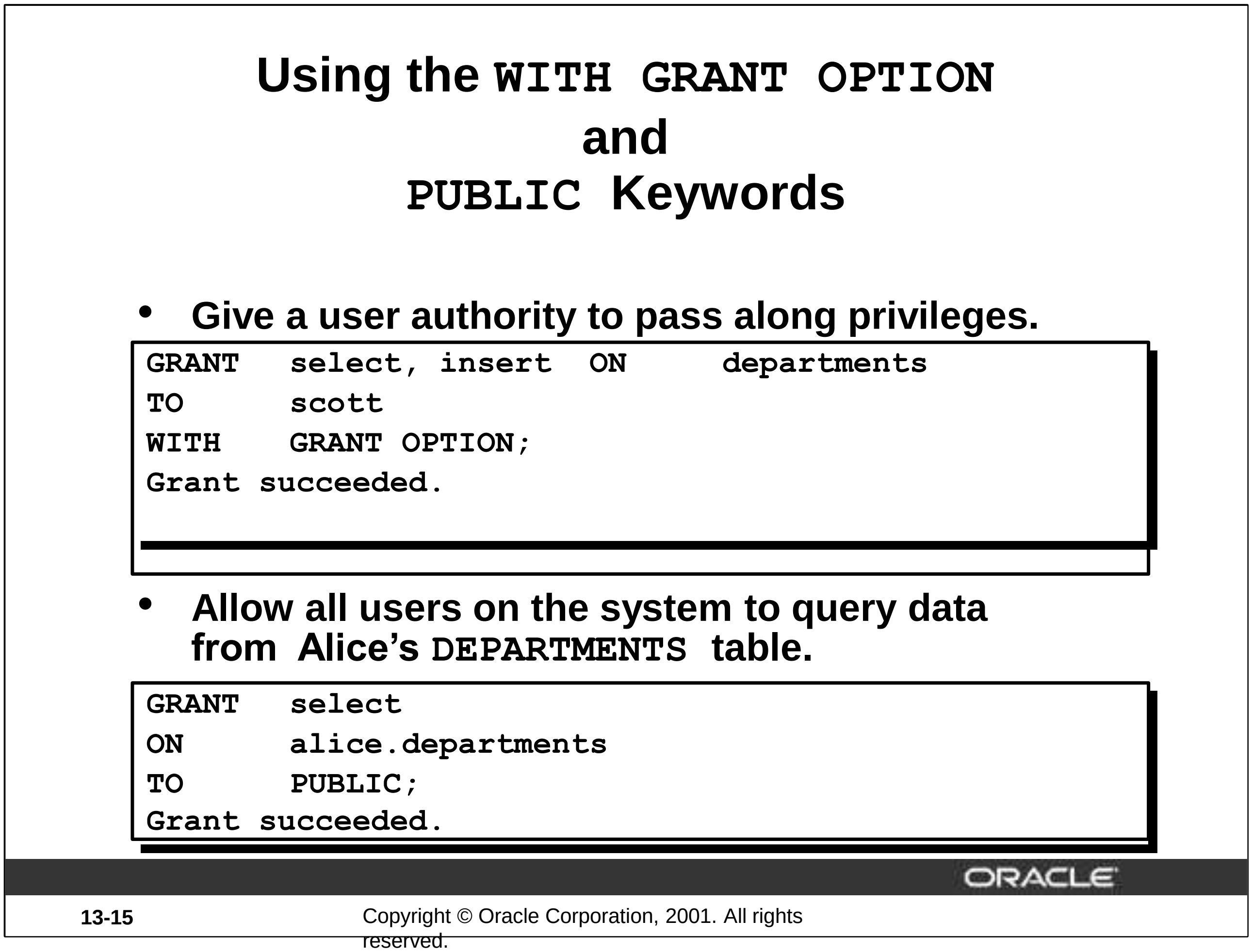
Granting Object Privileges

- **Grant query privileges on the `EMPLOYEES`**

```
GRANT   select
ON      employees
TO      sue, rich;  Grant succeeded.
```

- **Grant privileges to update specific columns to users and roles.**

```
GRANT   update (department_name, location_id)  ON
        departments
TO      scott, manager;  Grant succeeded.
```



Using the `WITH GRANT OPTION` and `PUBLIC` Keywords

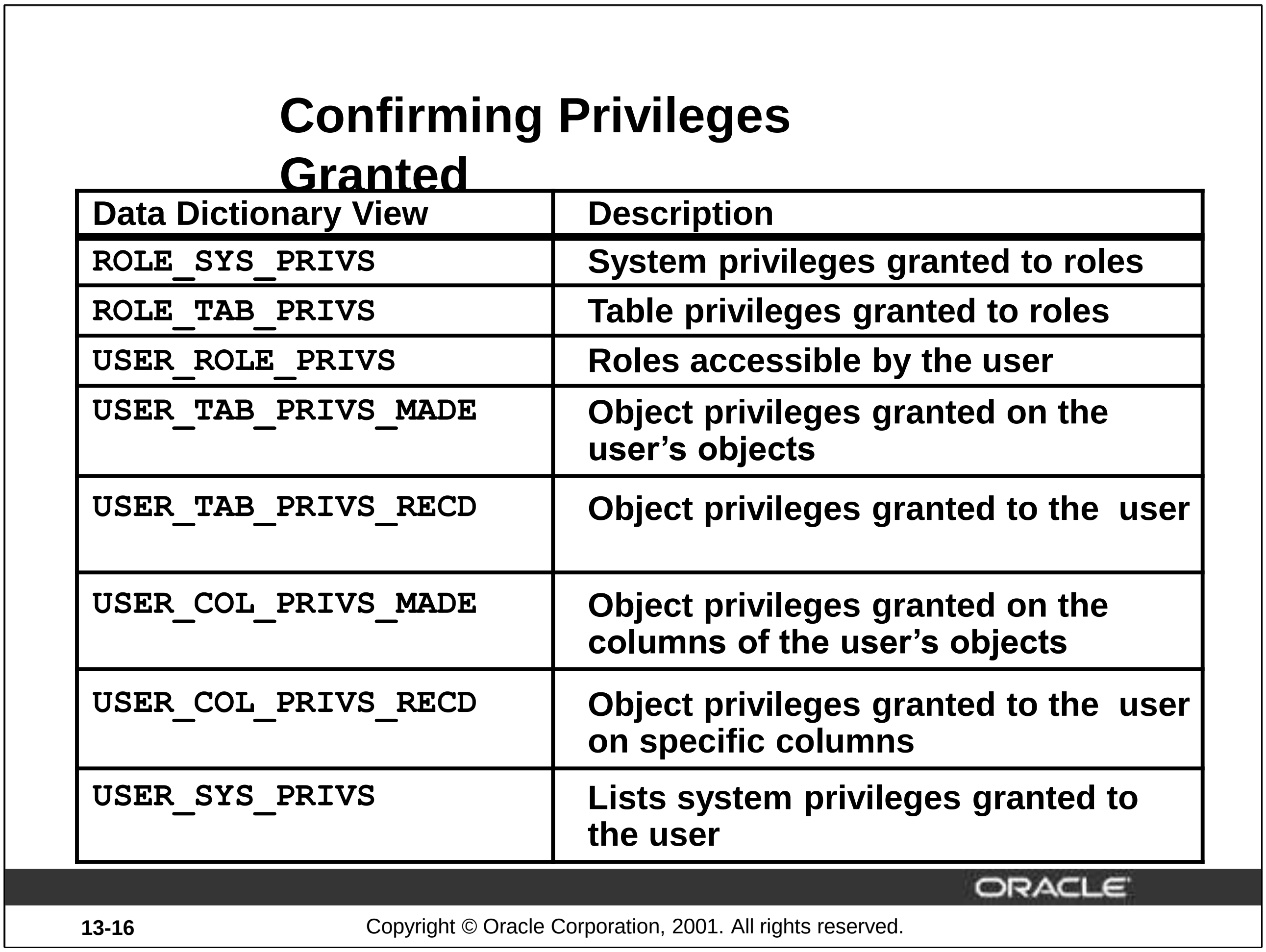
- Give a user authority to pass along privileges.

```
GRANT   select, insert  ON     departments
TO      scott
WITH    GRANT OPTION;
Grant succeeded.
```

- Allow all users on the system to query data from Alice's `DEPARTMENTS` table.

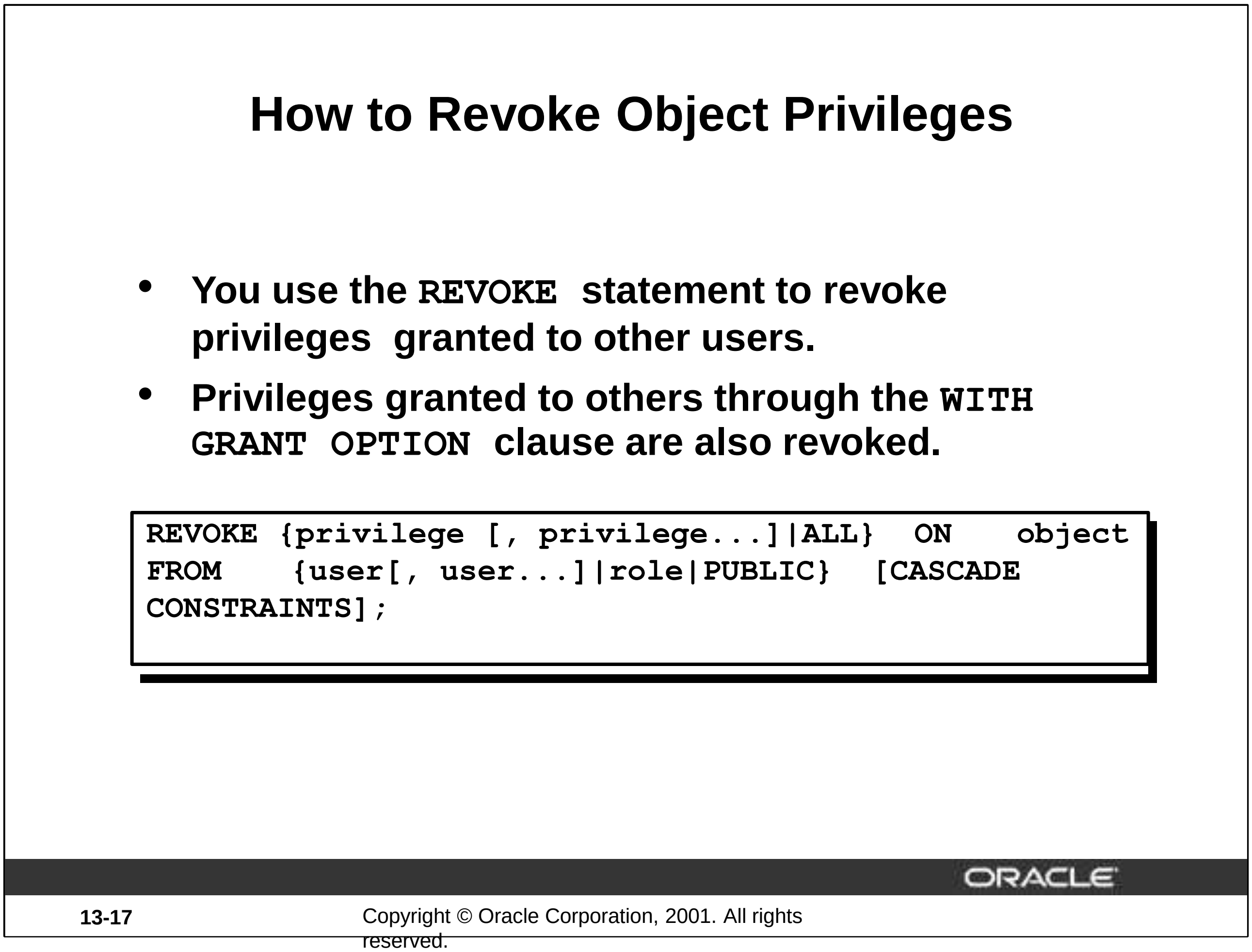
```
GRANT   select
ON      alice.departments
TO      PUBLIC;
Grant succeeded.
```

13-15



Confirming Privileges Granted

Data Dictionary View	Description
ROLE_SYS_PRIVS	System privileges granted to roles
ROLE_TAB_PRIVS	Table privileges granted to roles
USER_ROLE_PRIVS	Roles accessible by the user
USER_TAB_PRIVS_MADE	Object privileges granted on the user’s objects
USER_TAB_PRIVS_RECD	Object privileges granted to the user
USER_COL_PRIVS_MADE	Object privileges granted on the columns of the user’s objects
USER_COL_PRIVS_RECD	Object privileges granted to the user on specific columns
USER_SYS_PRIVS	Lists system privileges granted to the user



How to Revoke Object Privileges

- You use the `REVOKE` statement to revoke privileges granted to other users.
- Privileges granted to others through the `WITH GRANT OPTION` clause are also revoked.

```
REVOKE {privilege [, privilege...]|ALL}  ON   object
FROM    {user[, user...]|role|PUBLIC}  [CASCADE
CONSTRAINTS];
```

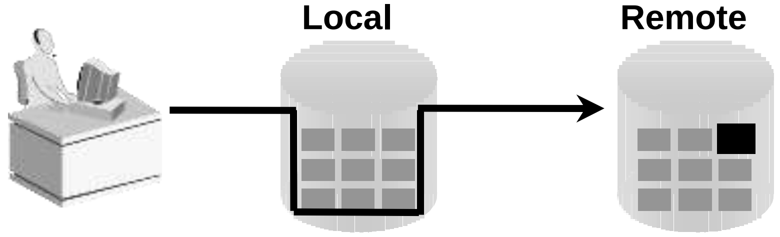
Revoking Object Privileges

As user Alice, revoke the `SELECT` and `INSERT` privileges given to user Scott on the `DEPARTMENTS` table.

```
REVOKE   select, insert  ON    departments
FROM     scott;  Revoke succeeded.
```

Database Links

A database link connection allows local users to access data on a remote database.



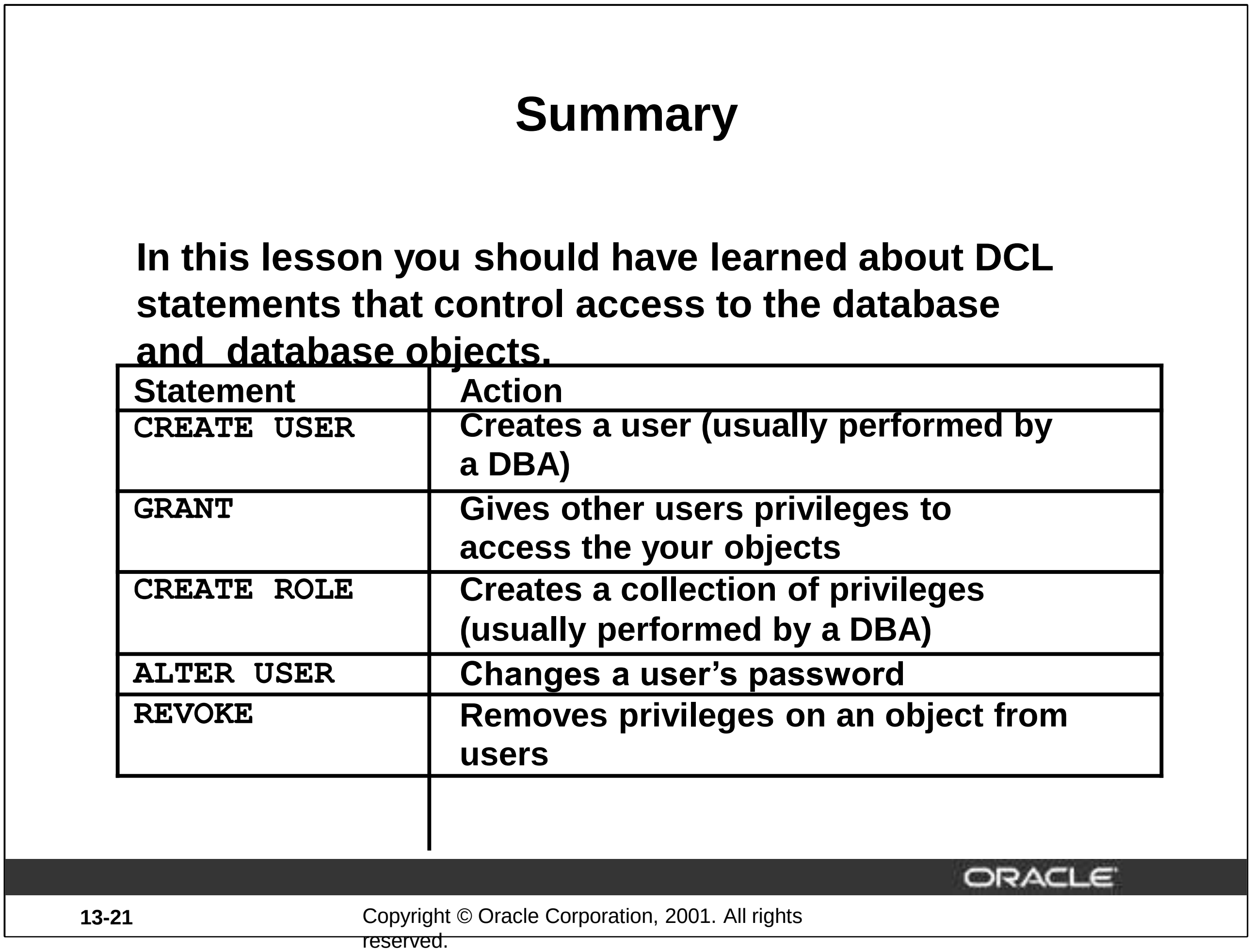
Local

Remote

fred.EMP Table

SELECT * FROM emp@HQ_ACME.COM;

HQ_ACME.COM Database



Summary

In this lesson you should have learned about DCL statements that control access to the database and database objects.

Statement	Action
`CREATE USER`	Creates a user (usually performed by a DBA)
`GRANT`	Gives other users privileges to access the your objects
`CREATE ROLE`	Creates a collection of privileges (usually performed by a DBA)
`ALTER USER`	Changes a user’s password
`REVOKE`	Removes privileges on an object from users

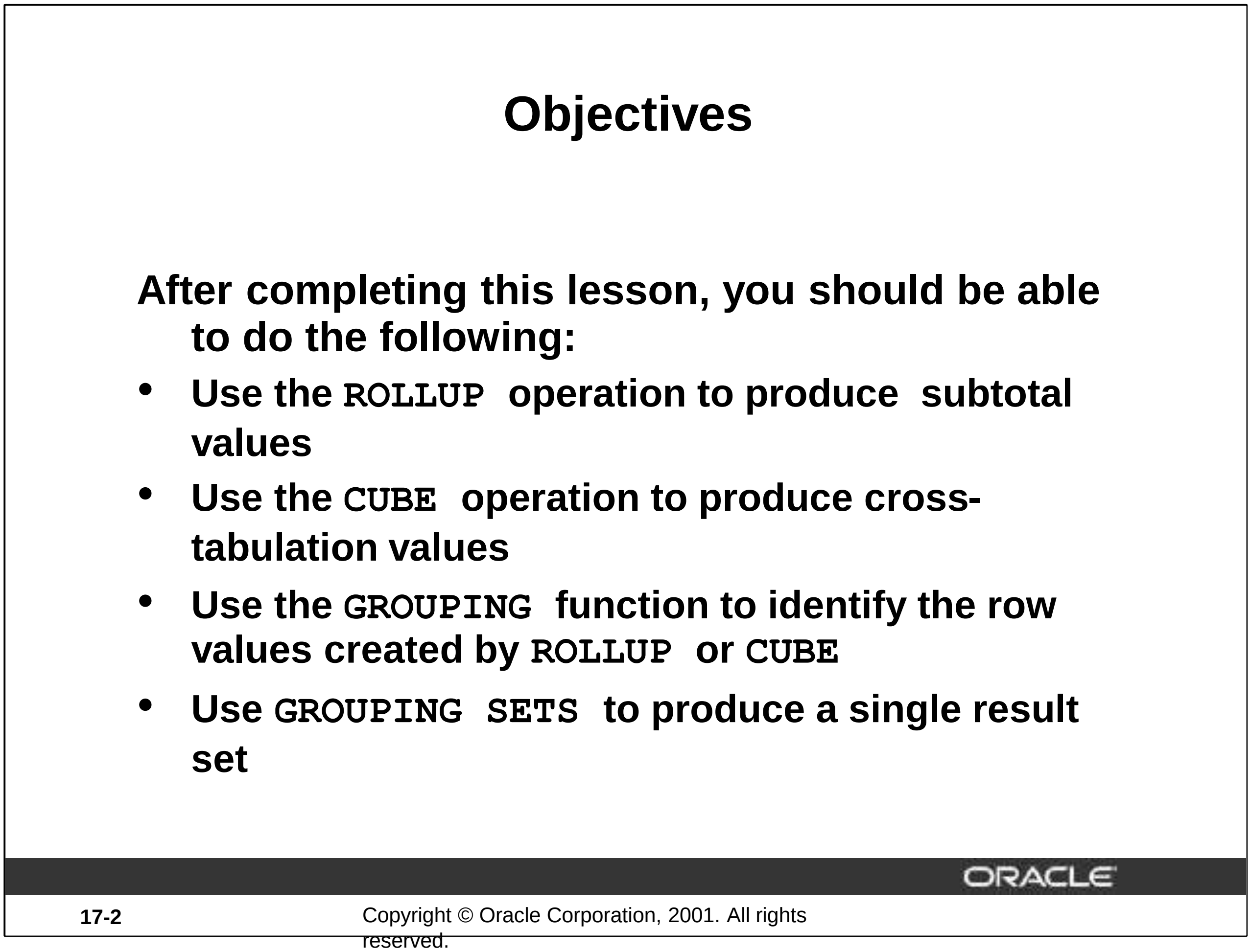
Summary

DBAs establish initial database security for users by assigning privileges to the users.

- The DBA creates users who must have a password. The DBA is also responsible for establishing the initial system privileges for a user.
- Once the user has created an object, the user can pass along any of the available object privileges to other users or to all users by using the `GRANT` statement.
- A DBA can create roles by using the `CREATE ROLE` statement to pass along a collection of system or object privileges to multiple users. Roles make granting and revoking privileges easier to maintain.
- Users can change their password by using the `ALTER USER` statement.
- You can remove privileges from users by using the `REVOKE` statement.
- With data dictionary views, users can view the privileges granted to them and those that are granted on their objects.
- With database links, you can access data on remote databases. Privileges cannot be granted on remote objects.

17

Enhancements to the `GROUP BY` Clause



Objectives

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

- Use the `ROLLUP` operation to produce subtotal values
- Use the `CUBE` operation to produce cross-tabulation values
- Use the `GROUPING` function to identify the row values created by `ROLLUP` or `CUBE`
- Use `GROUPING SETS` to produce a single result set

Review of Group Functions

Group functions operate on sets of rows to give one result per group.

```
SELECT        [column,] group_function(column). .
FROM          .  table
[WHERE        condition]  group_by_expression ]
[GROUP        column];
BY
[ORDER
BY
```

Example

```
:SELECT AVG(salary), STDDEV(salary),
        COUNT(commission_pct),MAX(hire_date)
FROM employees
WHERE job_id LIKE 'SA%';
```

Review of the GROUP BY Clause

Syntax:

```
SELECT          [column,] group_function(column). . .
FROM            table
[WHERE          condition]
[GROUP BY       group_by_expression ]
[ORDER BY       column];
```

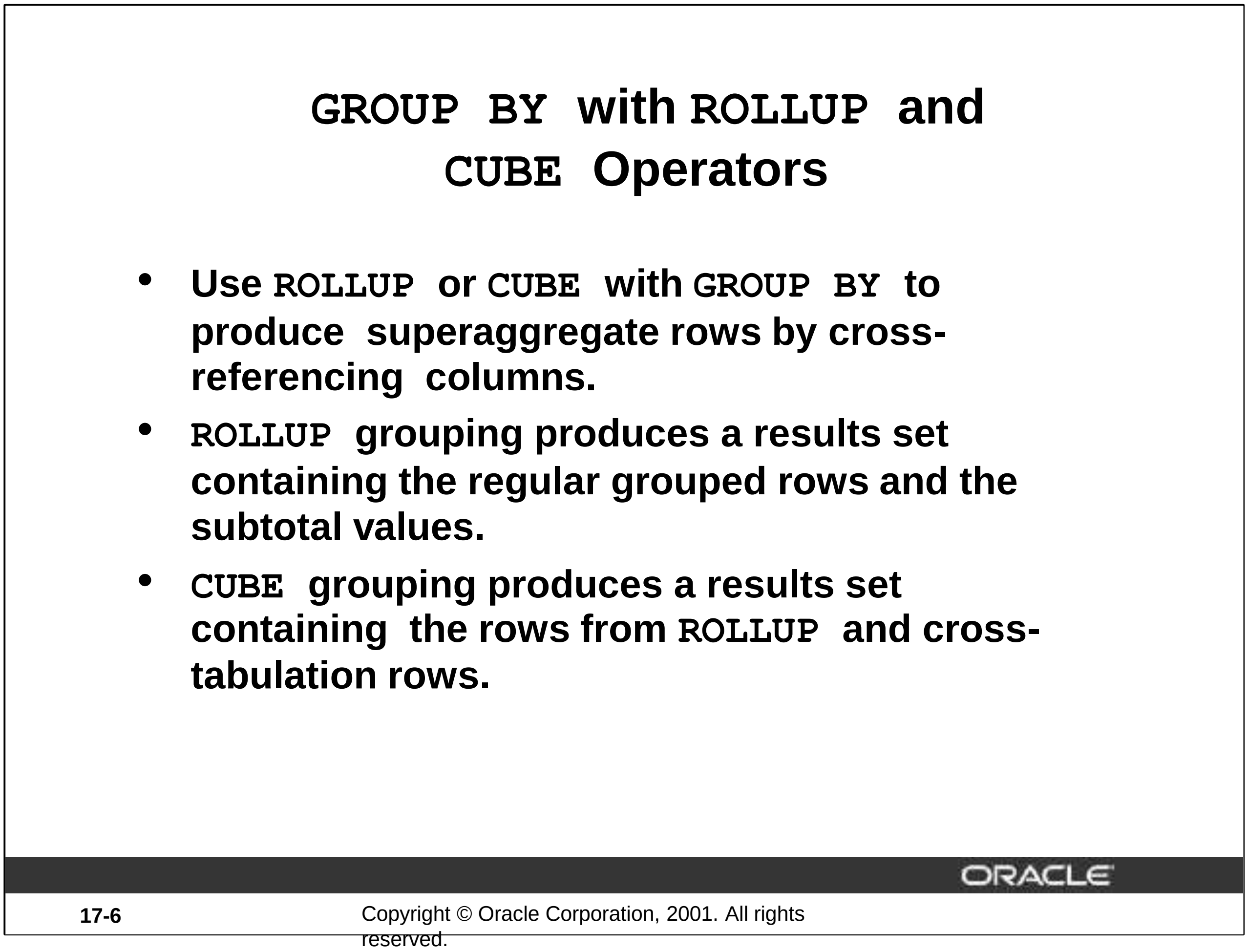
Example:

```
SELECT department_id, job_id, SUM(salary),
       COUNT(employee_id)
FROM   employees
GROUP BY department_id, job_id;
```

Review of the HAVING Clause

```
SELECT        [column,] group_function(column). .
FROM          .  table
[WHERE        condition]  group_by_expression ]
[GROUP        having_expression ];  column];
BY
[HAVING
[ORDER
BY
```

- Use the HAVING clause to specify which groups are to be displayed.
- You further restrict the groups on the basis of a limiting condition.



GROUP BY with ROLLUP and CUBE Operators

- Use ROLLUP or CUBE with GROUP BY to produce superaggregate rows by cross-referencing columns.
- ROLLUP grouping produces a results set containing the regular grouped rows and the subtotal values.
- CUBE grouping produces a results set containing the rows from ROLLUP and cross-tabulation rows.

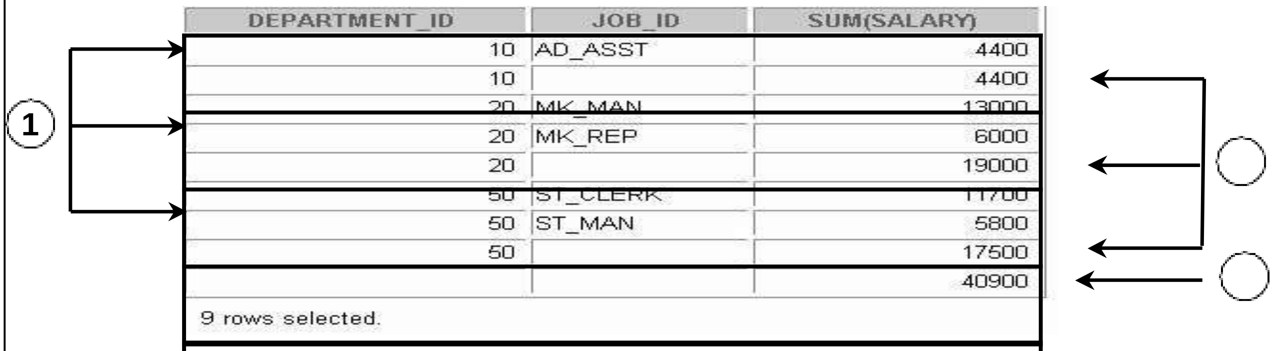
ROLLUP
Operator

```
SELECT        [column,] group_function(column). .
FROM          .  table
[WHERE        condition]
[GROUP        [ROLLUP] group_by_expression ]
BY            having_expression ];
[HAVING       column];
[ORDER
BY
```

- ROLLUP is an extension to the GROUP BY clause.
- Use the ROLLUP operation to produce cumulative aggregates such as subtotals.

`ROLLUP` Operator Example

```
SELECT    department_id, job_id, SUM(salary)
FROM      employees
WHERE department_id < 60
GROUP BY ROLLUP(department_id, job_id);
```



```
DEPARTMENT_ID   JOB_ID     SUM(SALARY)
           10   AD_ASST           4400
           10                     4400
           20   MK_MAN           13000
           20   MK_REP            6000
           20                    19000
           50   ST_CLERK         11700
           50   ST_MAN            5800
           50                    17500
                                 40900

9 rows selected.
```

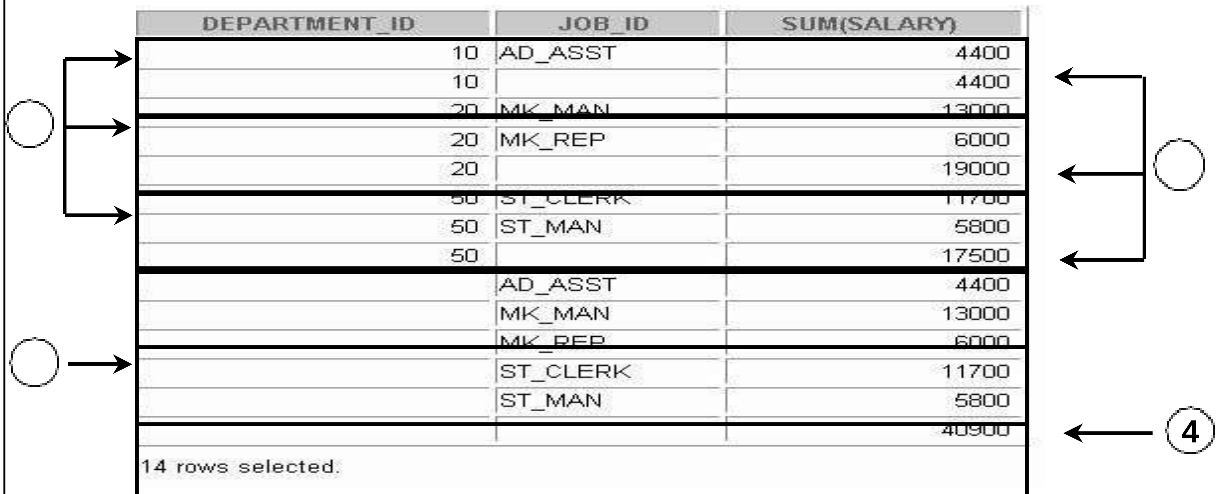
CUBE Operator

```
SELECT        [column,] group_function(column). .
FROM          .  table
[WHERE        condition]
[GROUP        [CUBE] group_by_expression ]
BY            having_expression ];  column];
[HAVING
[ORDER
BY
```

- `CUBE` is an extension to the `GROUP BY` clause.
- You can use the `CUBE` operator to produce cross- tabulation values with a single `SELECT` statement.

`CUBE` Operator: Example

```
SELECT    department_id, job_id, SUM(salary)
FROM      employees
WHERE     department_id < 60
GROUP BY CUBE (department_id, job_id);
```



DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
10		4400
20	MK_MAN	13000
20	MK_REP	6000
20		19000
50	ST_CLERK	11700
50	ST_MAN	5800
50		17500
	AD_ASST	4400
	MK_MAN	13000
	MK_REP	6000
	ST_CLERK	11700
	ST_MAN	5800
		40900

14 rows selected.

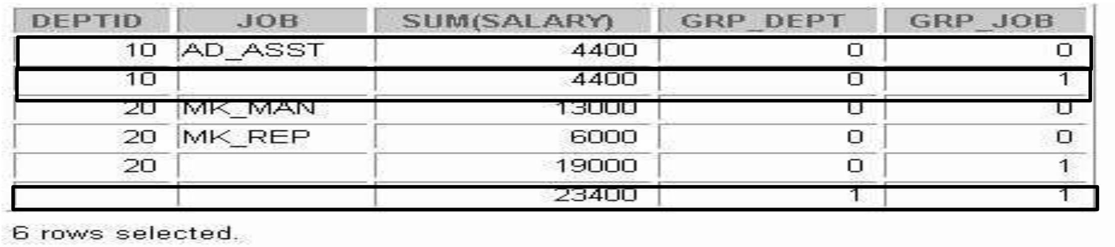
GROUPING Function

```
SELECT   [column,] group_function(column) . .,
FROM     GROUPING(expr)  table
[WHERE   condition]
[GROUP   BY  [ROLLUP][CUBE] group_by_expression
[HAVING  ]having_expression
[ORDER       ];  column];
BY
```

- The GROUPING function can be used with either the CUBE or ROLLUP operator.
- Using it, you can find the groups forming the subtotal in a row.
- Using it, you can differentiate stored NULL values from NULL values created by ROLLUP or CUBE.
- It returns 0 or 1.

GROUPING Function: Example

```
SELECT   department_id DEPTID, job_id JOB, SUM(salary),
GROUPING(department_id) GRP_DEPT,GROUPING(job_id)
GRP_JOB  FROM   employees
WHERE department_id < 50
GROUP BY ROLLUP(department_id, job_id);
```



DEPTID	JOB	SUM(SALARY)	GRP_DEPT	GRP_JOB
10	AD_ASST	4400	0	0
10		4400	0	1
20	MK_MAN	13000	0	0
20	MK_REP	6000	0	0
20		19000	0	1
		23400	1	1

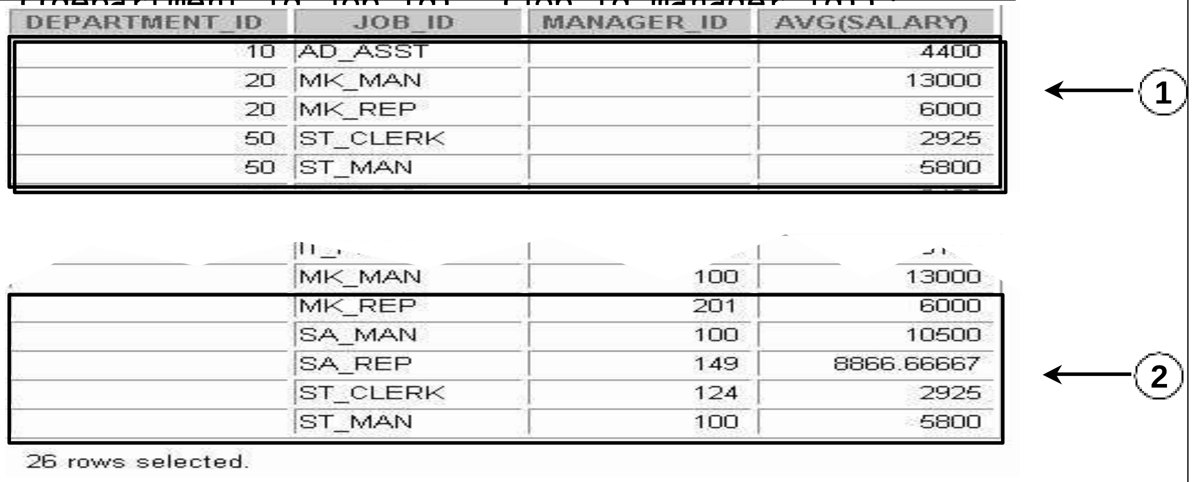
6 rows selected.

GROUPING SETS

- GROUPING SETS are a further extension of the GROUP BY clause.
- You can use GROUPING SETS to define multiple groupings in the same query.
- The Oracle Server computes all groupings specified in the GROUPING SETS clause and combines the results of individual groupings with a UNION ALL operation.
- Grouping set efficiency:
  - Only one pass over the base table is required.
  - There is no need to write complex UNION statements.
  - The more elements the GROUPING SETS have, the higher the performance benefit is.

GROUPING SETS:

```
SELECT department_id, job_id, manager_id,
avg(salary)
FROM   employees
GROUP BY GROUPING SETS
((department_id,job_id), (job_id,manager_id));
```



DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
10	AD_ASST		4400
20	MK_MAN		13000
20	MK_REP		6000
50	ST_CLERK		2925
50	ST_MAN		5800
	MK_MAN	100	13000
	MK_REP	201	6000
	SA_MAN	100	10500
	SA_REP	149	8866.66667
	ST_CLERK	124	2925
	ST_MAN	100	5800

26 rows selected.

Composite Columns

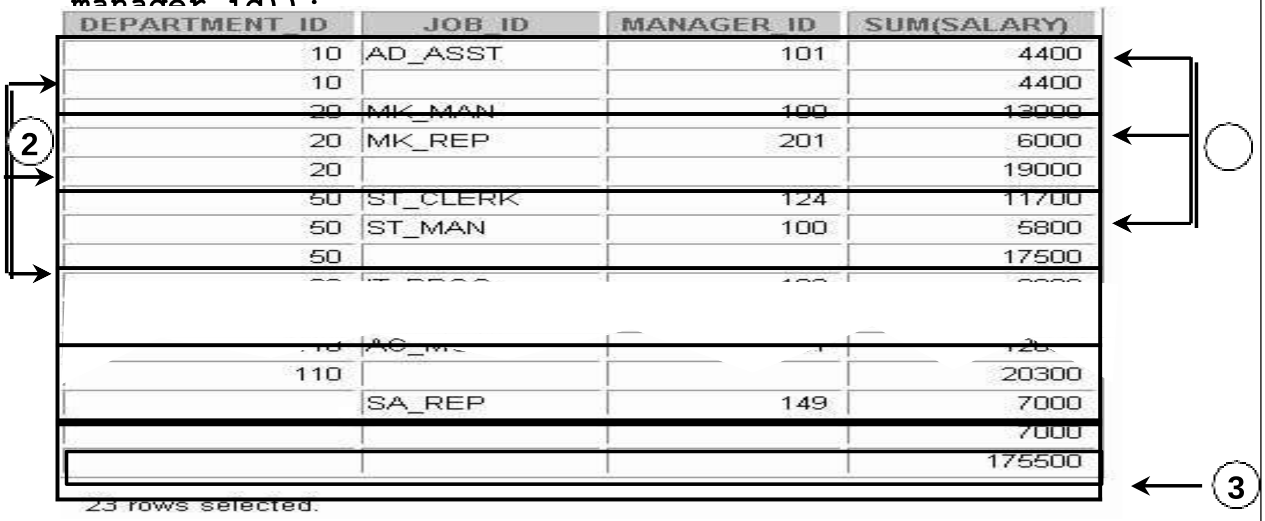
- A composite column is a collection of columns that are treated as a unit.

  `ROLLUP (a, (b,c) , d)`

- To specify composite columns, in the `GROUP BY` clause you group columns within parentheses so that the Oracle server treats them as a unit while computing `ROLLUP` or `CUBE` operations.
- When used with `ROLLUP` or `CUBE`, composite columns would mean skipping aggregation across certain levels.

Composite Columns: Example

```
SELECT   department_id, job_id, manager_id,
         SUM(salary)
FROM     employees
GROUP BY ROLLUP( department_id,(job_id, manager_id));
```



DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
10	AD_ASST	101	4400
10			4400
20	MK_MAN	100	13000
20	MK_REP	201	6000
20			19000
50	ST_CLERK	124	11700
50	ST_MAN	100	5800
50			17500
[illegible]	IT_PROG	[illegible]	[illegible]
[illegible]	AC_M[illegible]	[illegible]	[illegible]
110			20300
	SA_REP	149	7000
			7000
			175500

23 rows selected.

Summary

In this lesson, you should have learned how to:

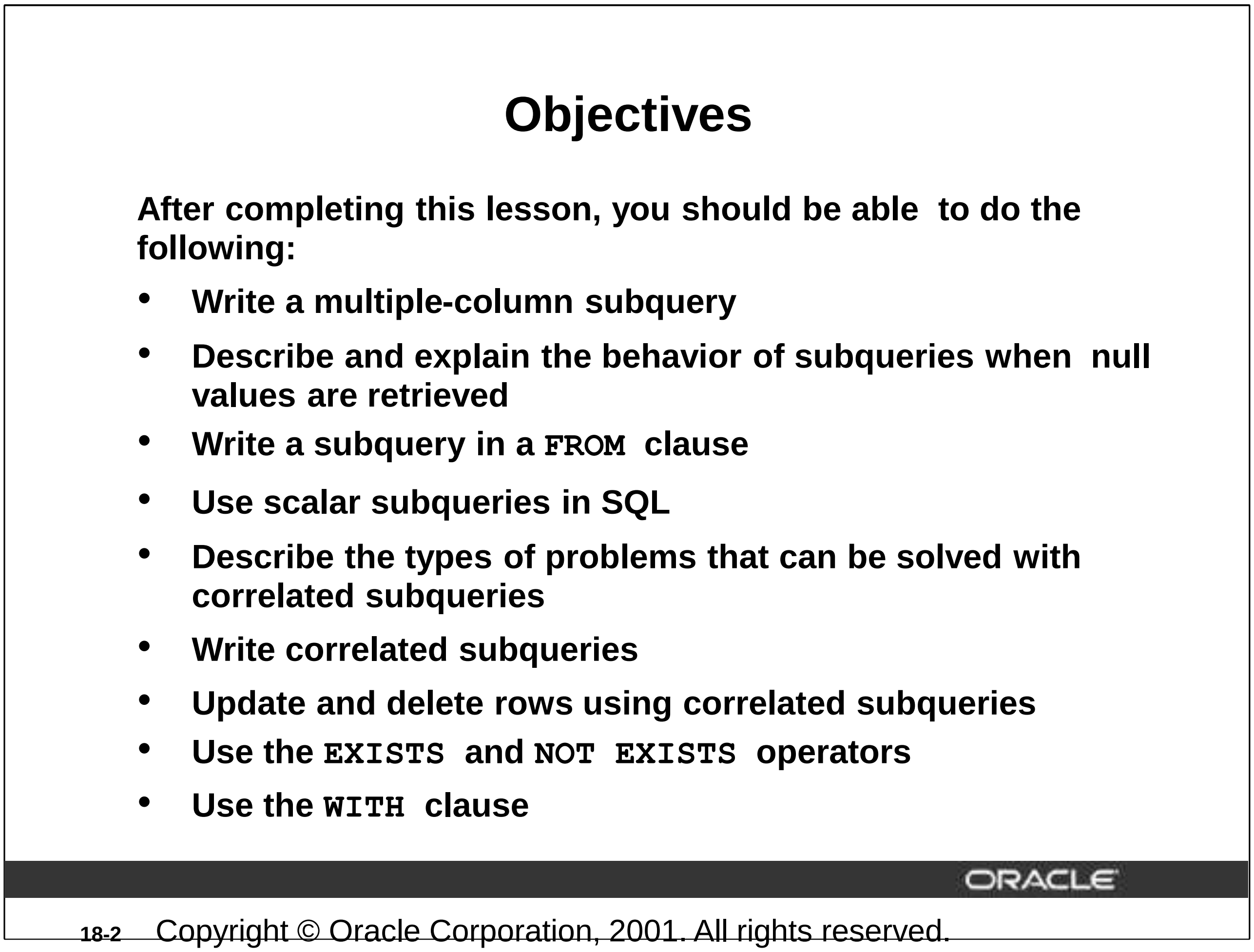
- **Use the `ROLLUP` operation to produce subtotal values**
- **Use the `CUBE` operation to produce cross-tabulation values**
- **Use the `GROUPING` function to identify the row values created by `ROLLUP` or `CUBE`**
- **Use the `GROUPING SETS` syntax to define multiple groupings in the same query.**
- **Use the `GROUP BY` clause, to combine expressions in various ways:**
  - **Composite columns**
  - **Concatenated grouping sets**

Summary

- `ROLLUP` and `CUBE` are extensions of the `GROUP BY` clause.
- `ROLLUP` is used to display subtotal and grand total values.
- `CUBE` is used to display cross-tabulation values.
- The `GROUPING` function helps you determine whether a row is an aggregate produced by a `CUBE` or `ROLLUP` operator.
- With the `GROUPING SETS` syntax, you can define multiple groupings in the same query. `GROUP BY` computes all the groupings specified and combines them with `UNION ALL`.
- Within the `GROUP BY` clause, you can combine expressions in various ways:
  - To specify composite columns, you group columns within parentheses so that the Oracle Server treats them as a unit while computing `ROLLUP` or `CUBE` operations.
  - To specify concatenated grouping sets, you separate multiple grouping sets, `ROLLUP`, and `CUBE` operations with commas so that the Oracle Server combines them into a single `GROUP BY` clause. The result is a cross-product of groupings from each grouping set.

18

Advanced Subqueries



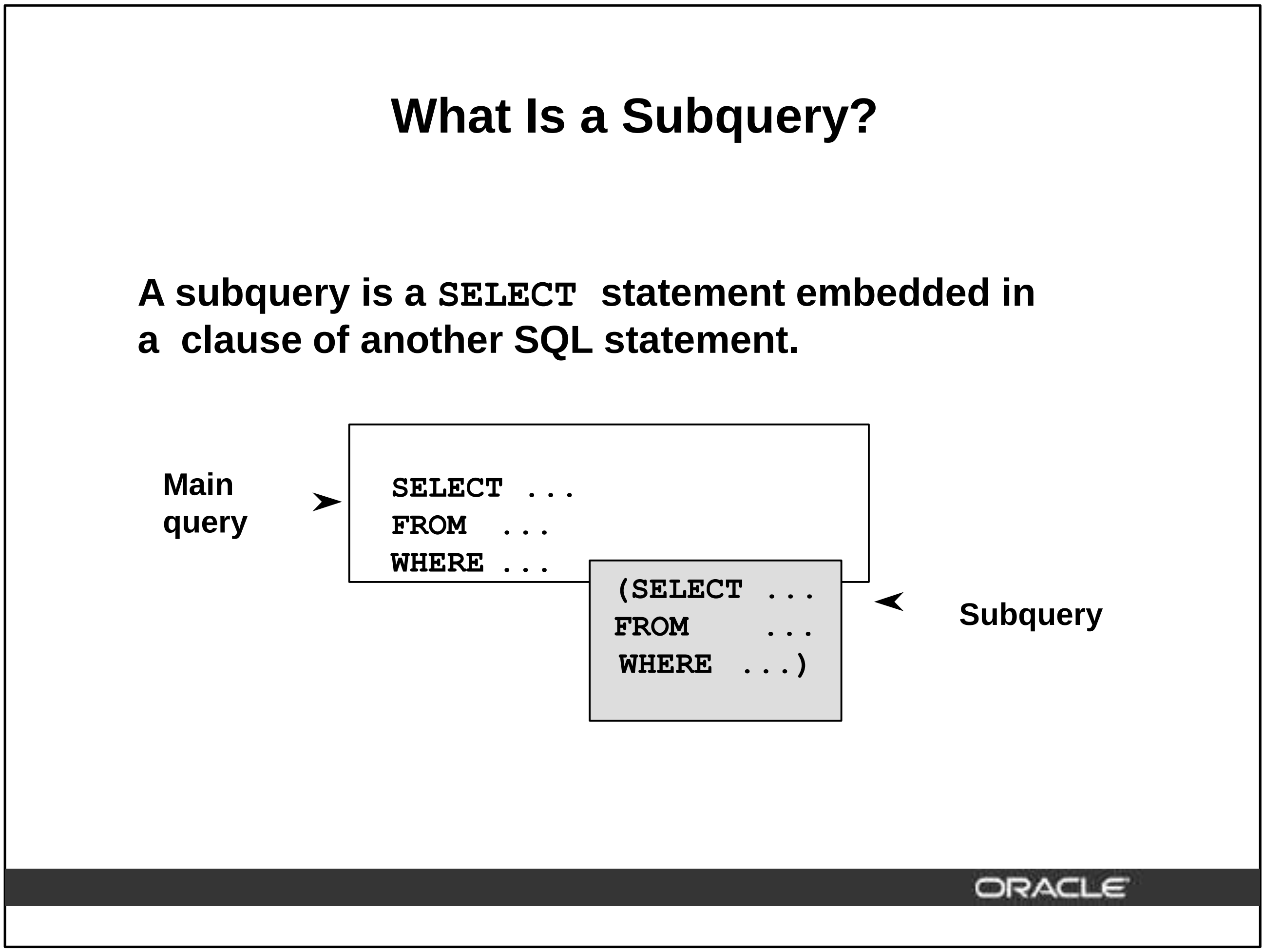
Objectives

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

- Write a multiple-column subquery
- Describe and explain the behavior of subqueries when null values are retrieved
- Write a subquery in a `FROM` clause
- Use scalar subqueries in SQL
- Describe the types of problems that can be solved with correlated subqueries
- Write correlated subqueries
- Update and delete rows using correlated subqueries
- Use the `EXISTS` and `NOT EXISTS` operators
- Use the `WITH` clause

Lesson Aim

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



What Is a Subquery?

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

```
SELECT ...
FROM  ...
WHERE ...
```

```
(SELECT ...
FROM     ...
 WHERE  ...)
```

Subqueries

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

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

Scalar Subqueries: Examples

Scalar Subqueries in `CASE` Expressions

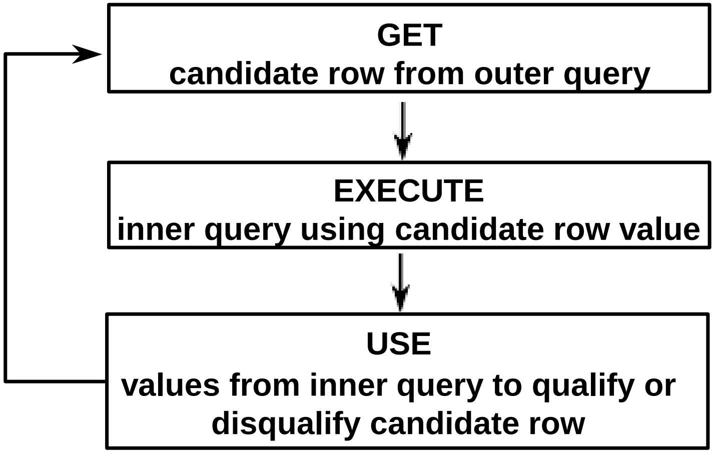
```
SELECT employee_id, last_name,
       (CASE
 WHEN department_id =                      20
             (SELECT department_id FROM departments
              WHERE location_id = 1800)
        THEN 'Canada' ELSE 'USA' END) location
FROM employees;
```

Scalar Subqueries in `ORDER BY` Clause

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

Correlated Subqueries

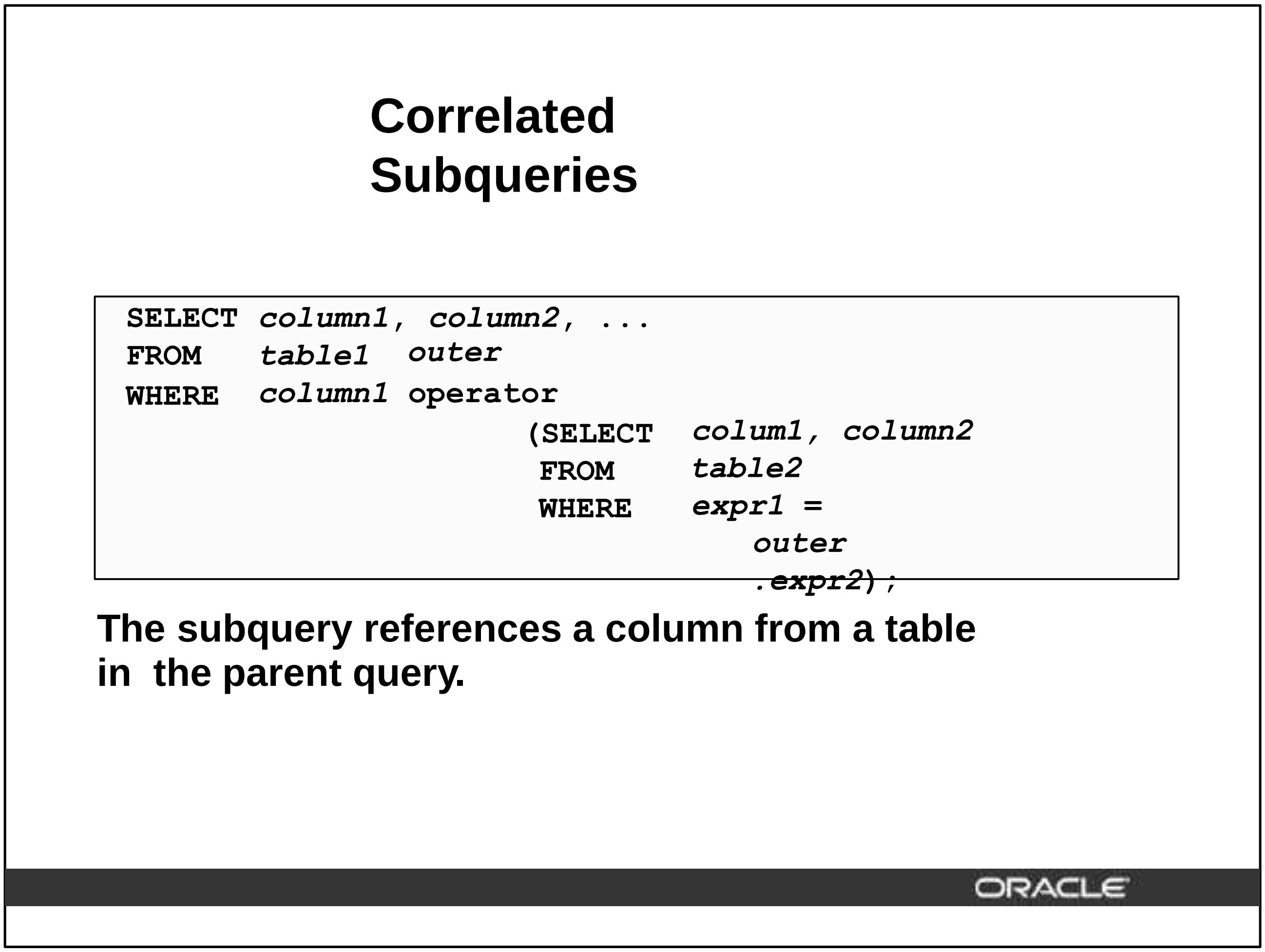
Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.



GET
candidate row from outer query

EXECUTE
inner query using candidate row value

USE
values from inner query to qualify or disqualify candidate row



Correlated Subqueries

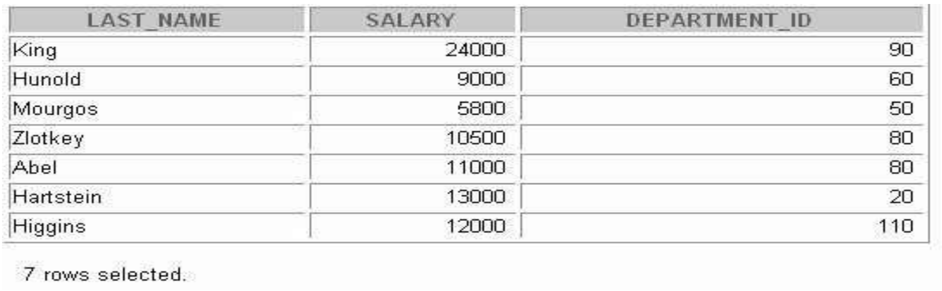
```
SELECT column1, column2, ...
FROM   table1  outer
WHERE  column1 operator
                         (SELECT  colum1, column2
                          FROM    table2
                          WHERE   expr1 =
                                      outer
                                      .expr2);
```

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

Using Correlated Subqueries

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

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



LAST_NAME	SALARY	DEPARTMENT_ID
King	24000	90
Hunold	9000	60
Mourgos	5800	50
Zlotkey	10500	80
Abel	11000	80
Hartstein	13000	20
Higgins	12000	110

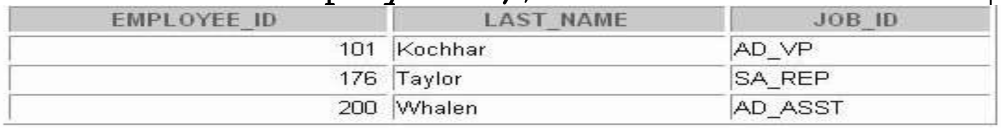
7 rows selected.

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

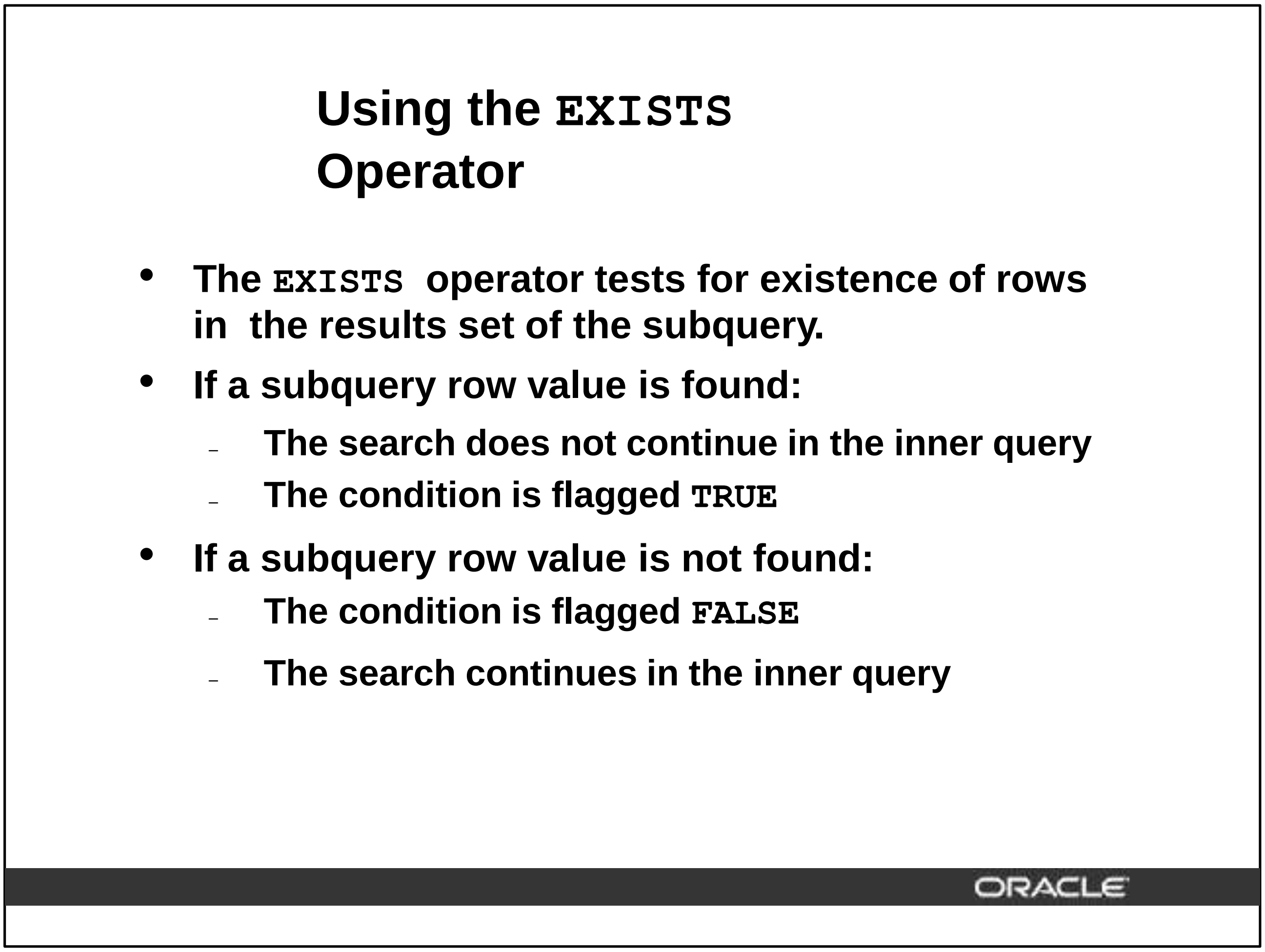
Using Correlated Subqueries

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

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



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



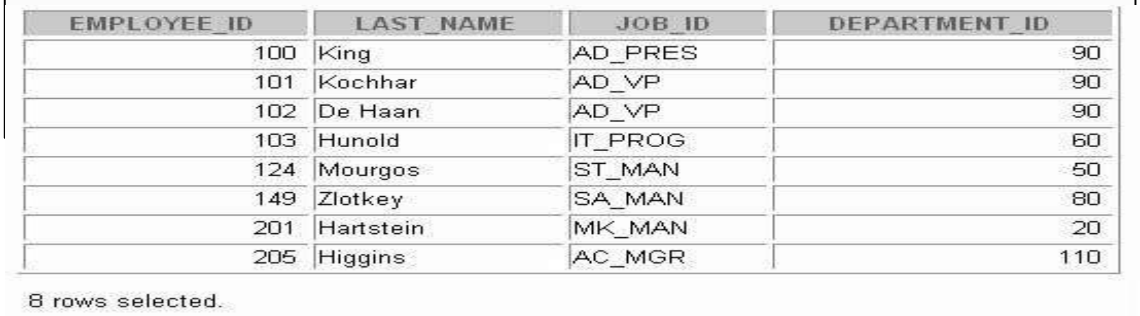
Using the `EXISTS` Operator

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

Using the EXISTS Operator

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

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



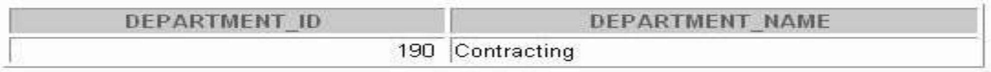
```
EMPLOYEE_ID  LAST_NAME   JOB_ID    DEPARTMENT_ID
        100  King        AD_PRES              90
        101  Kochhar     AD_VP                90
        102  De Haan     AD_VP                90
        103  Hunold      IT_PROG              60
        124  Mourgos     ST_MAN               50
        149  Zlotkey     SA_MAN               80
        201  Hartstein   MK_MAN               20
        205  Higgins     AC_MGR              110

8 rows selected.
```

Using the `NOT EXISTS` Operator

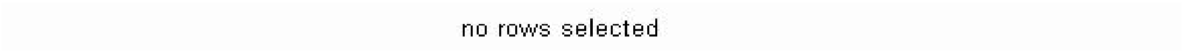
Find all departments that do not have any employees.

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

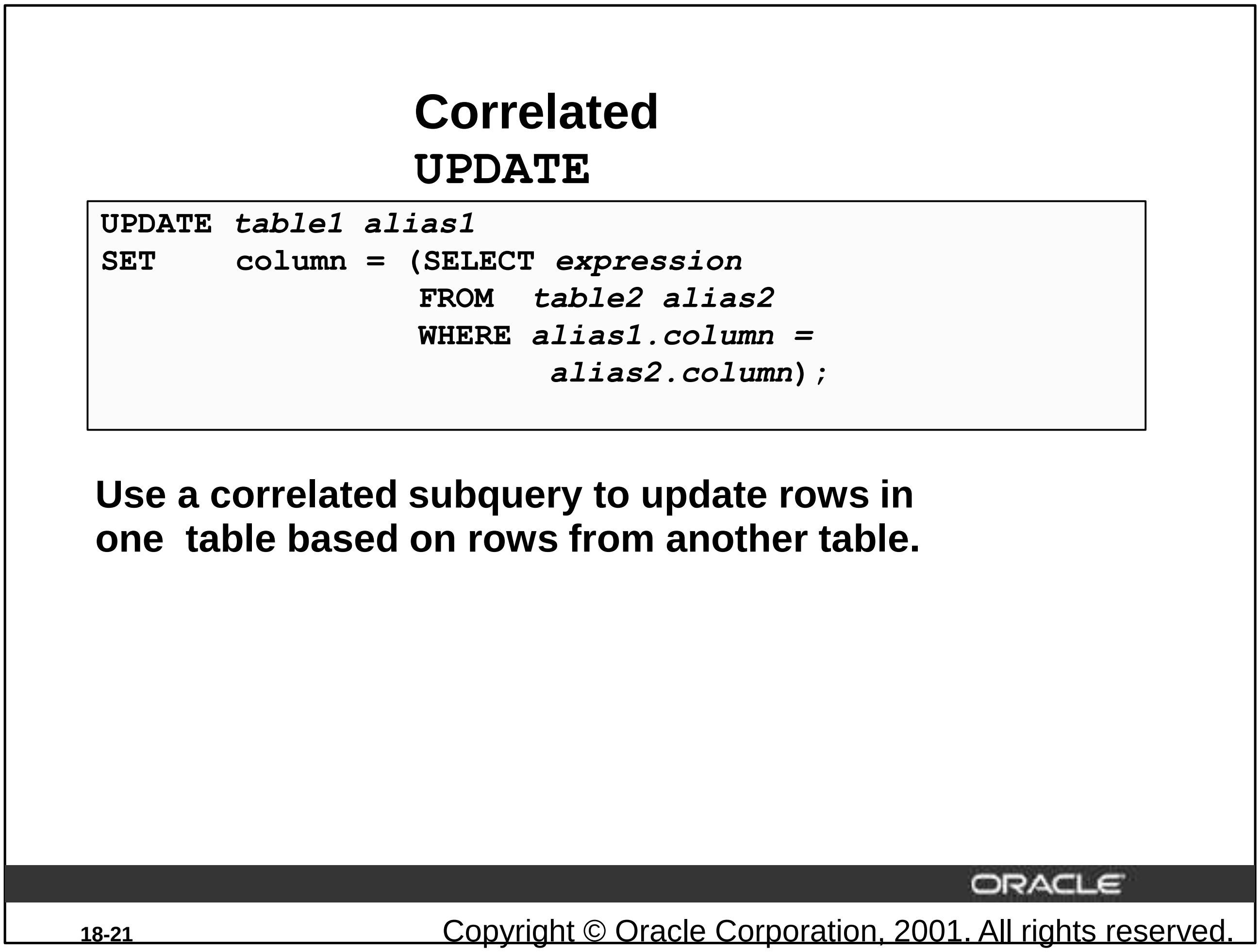


DEPARTMENT_ID	DEPARTMENT_NAME
190	Contracting

18-20



```
no rows selected
```



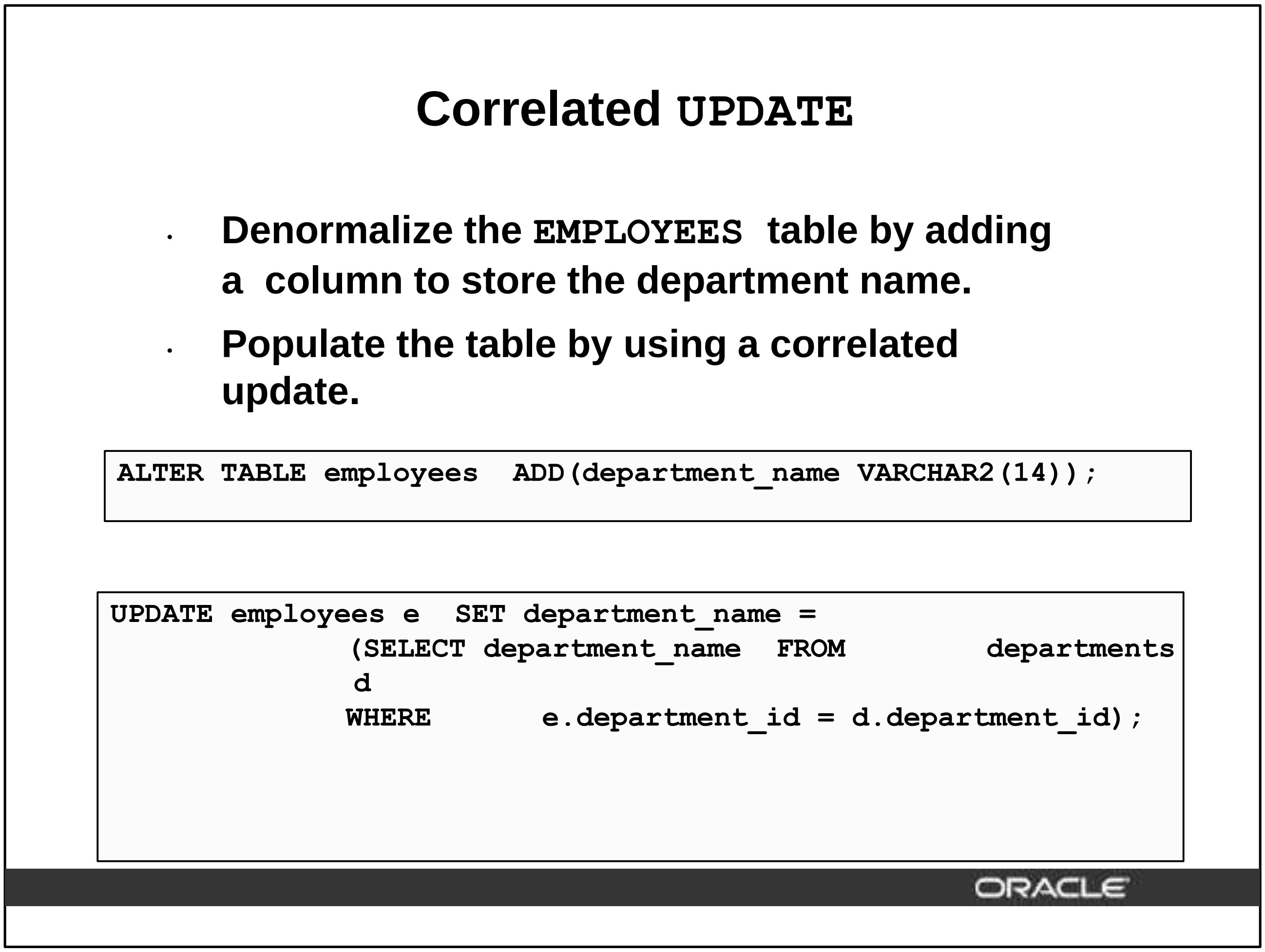
Correlated UPDATE

```
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.

Correlated `UPDATE`

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

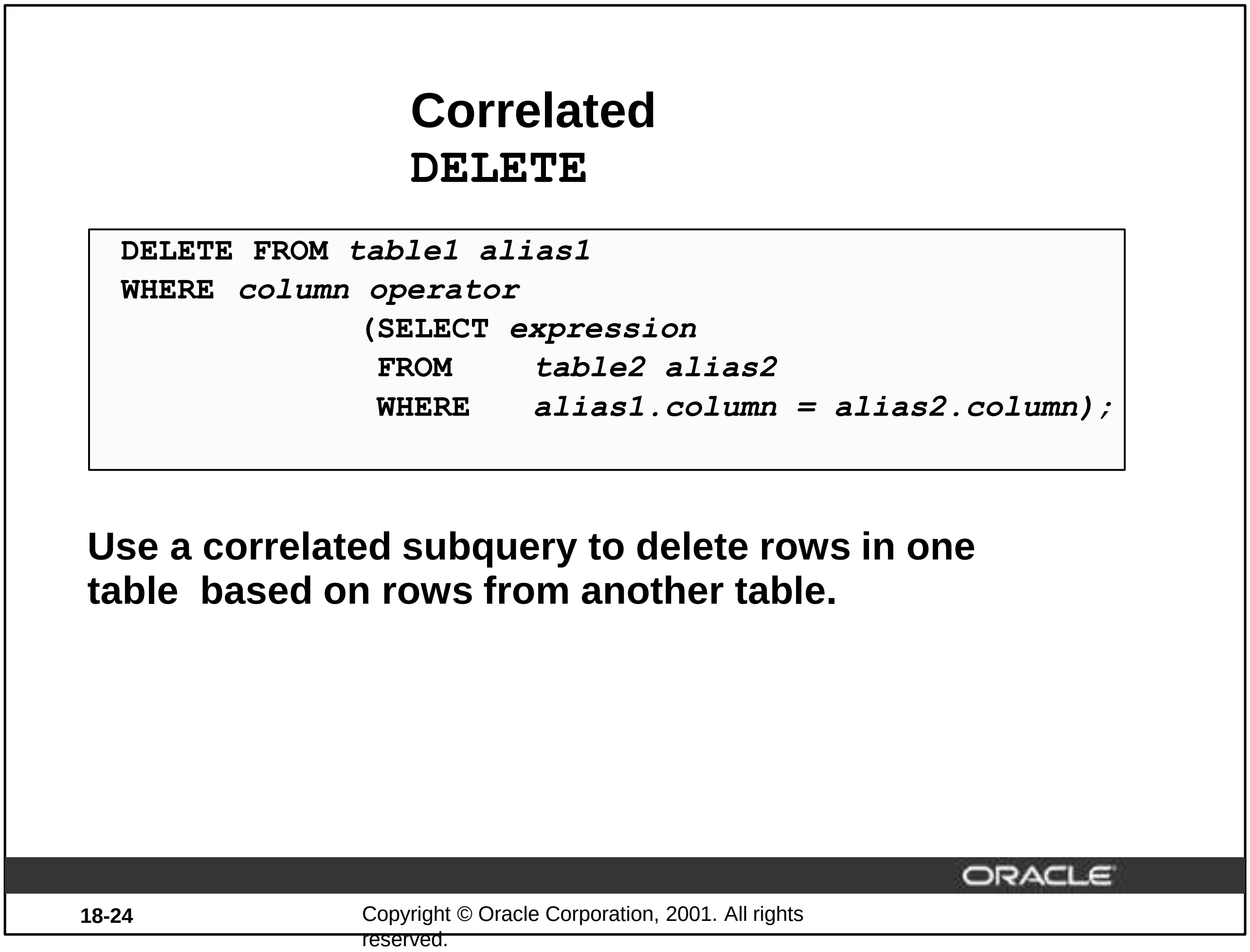


Correlated `UPDATE`

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

```
ALTER TABLE employees  ADD(department_name VARCHAR2(14));
```

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



Correlated `DELETE`

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

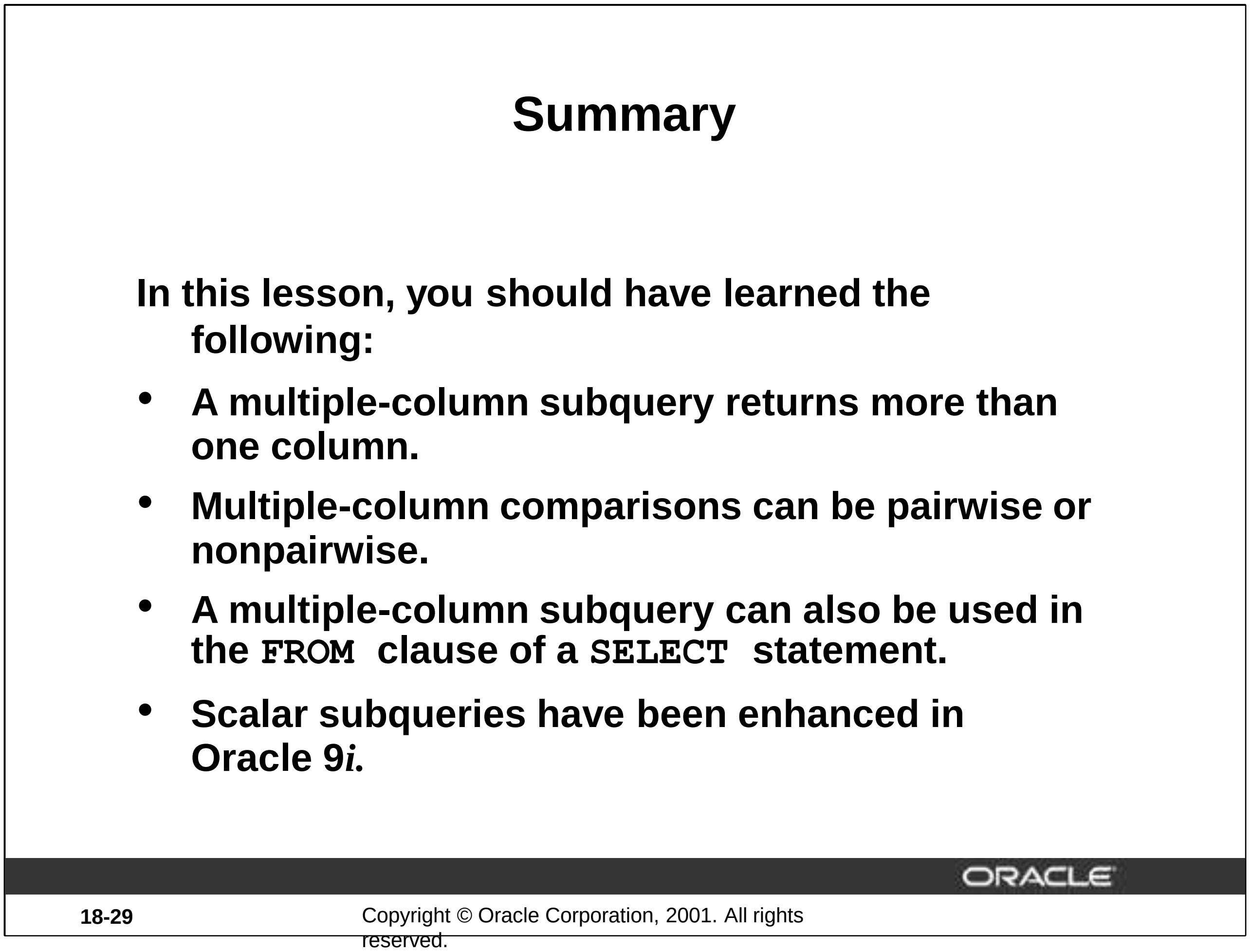
Use a correlated subquery to delete rows in one table based on rows from another table.

18-24

Correlated `DELETE`

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

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



Summary

In this lesson, you should have learned the following:

- A multiple-column subquery returns more than one column.
- Multiple-column comparisons can be pairwise or nonpairwise.
- A multiple-column subquery can also be used in the `FROM` clause of a `SELECT` statement.
- Scalar subqueries have been enhanced in Oracle 9*i*.

18-29

Summary

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

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

Oracle 9*i* enhances the the uses of scalar subqueries. Scalar subqueries can now be used in:

- Condition and expression part of `DECODE` and `CASE`
- All clauses of `SELECT` except `GROUP BY`
- `SET` clause and `WHERE` clause of `UPDATE` statement