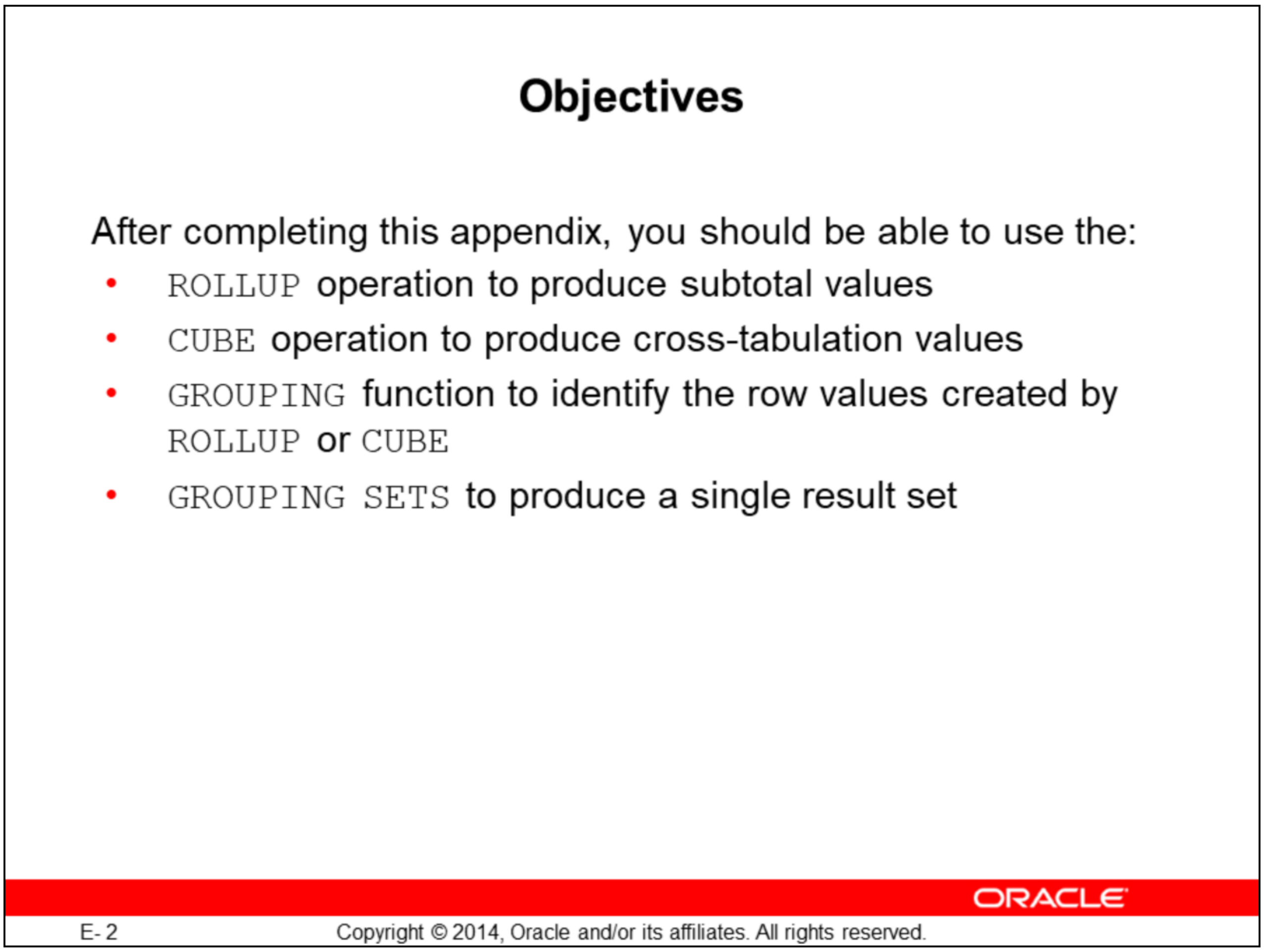
E

Generating Reports by Grouping Related Data

ORACLE



Objectives

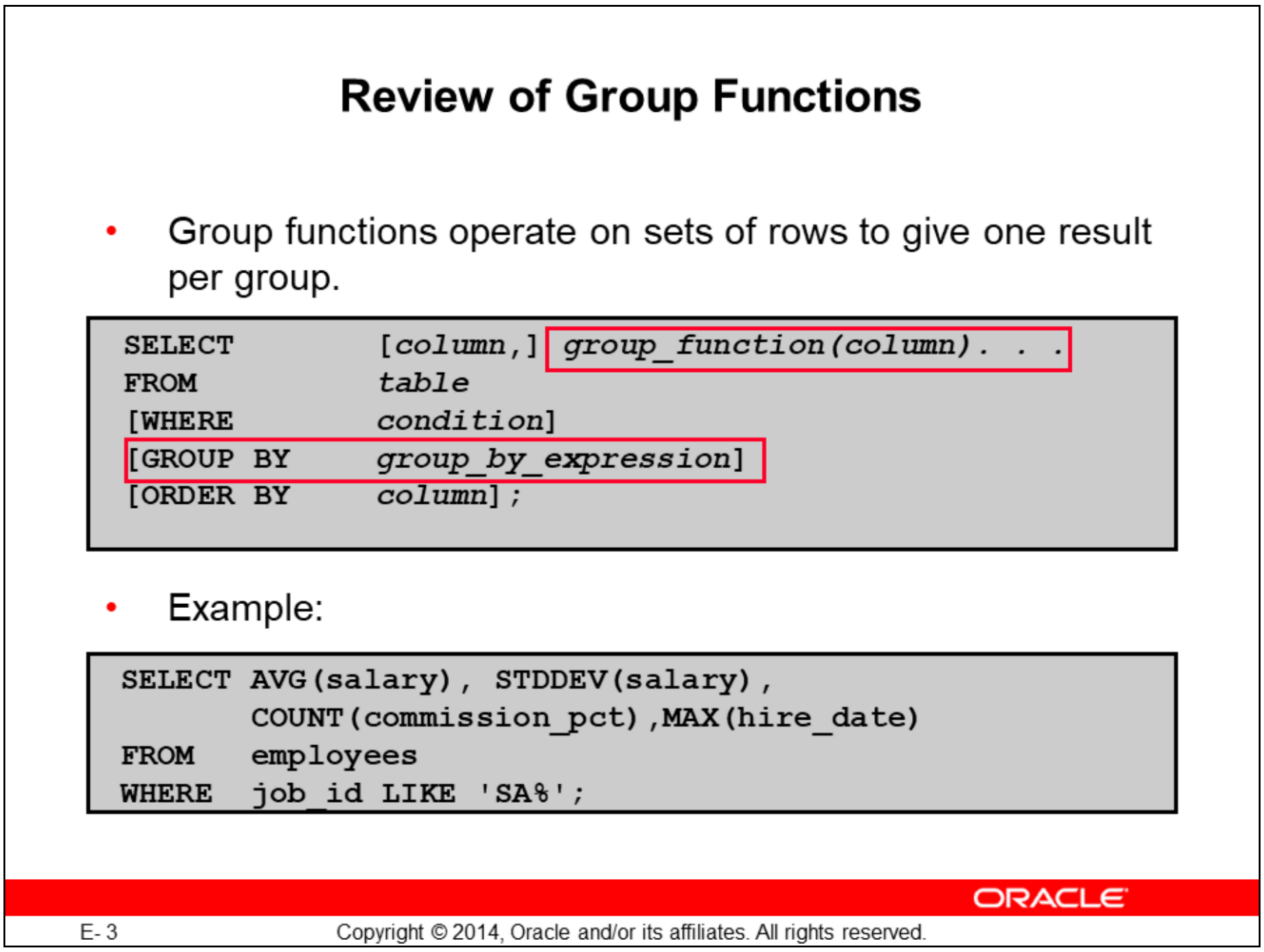
After completing this appendix, you should be able to use the:

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

E- 2

In this appendix, you learn how to:

- Group data to obtain subtotal values by using the `ROLLUP` operator
- Group data to obtain cross-tabulation values by using the `CUBE` operator
- Use the `GROUPING` function to identify the level of aggregation in the result set produced by a `ROLLUP` or `CUBE` operator
- Use `GROUPING SETS` to produce a single result set that is equivalent to a `UNION ALL`



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      group_by_expression]
[ORDER BY      column];
```

- Example:

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

E- 3

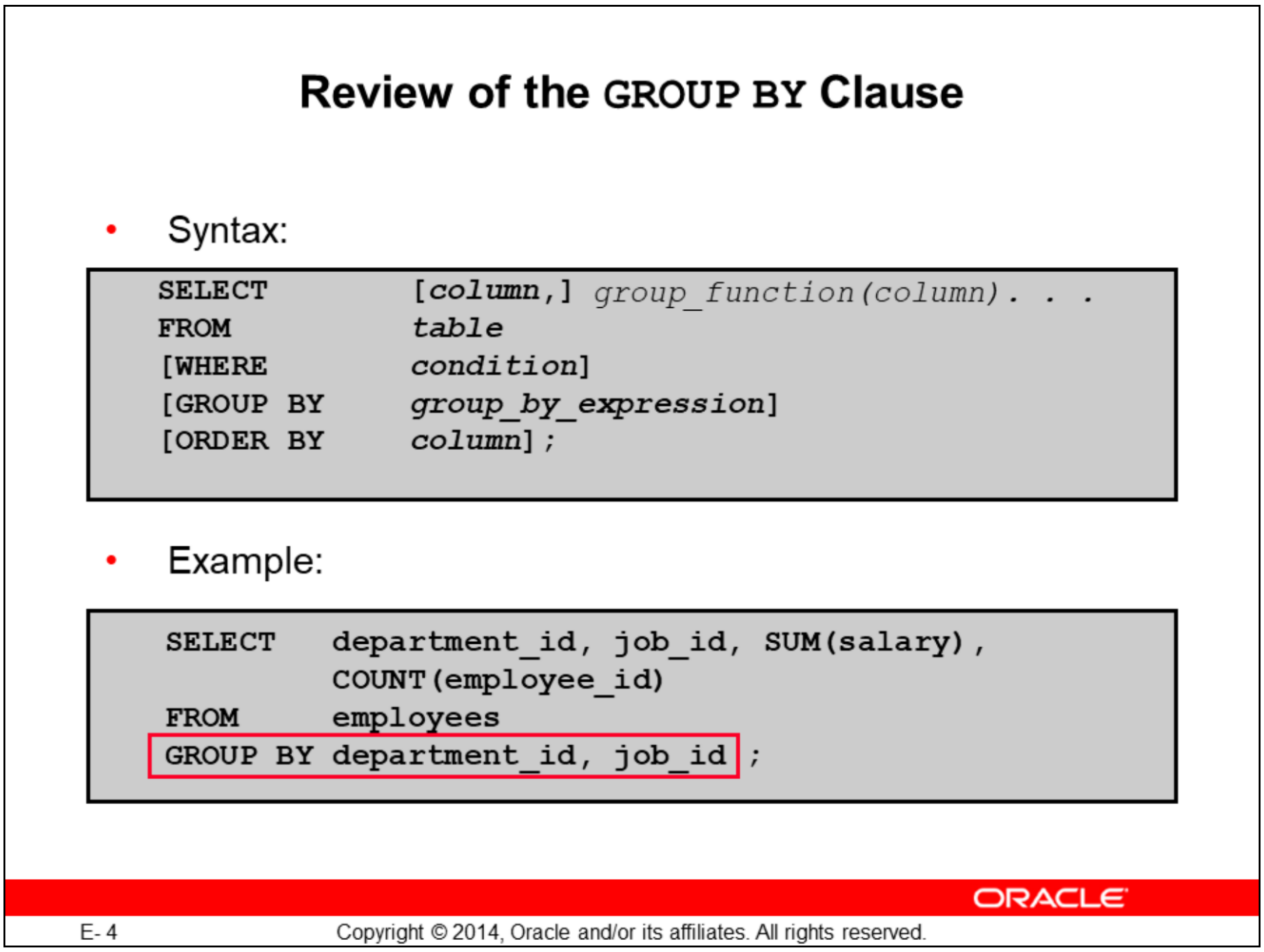
You can use the GROUP BY clause to divide the rows in a table into groups. You can then use group functions to return summary information for each group. Group functions can appear in select lists and in ORDER BY and HAVING clauses. The Oracle server applies the group functions to each group of rows and returns a single result row for each group.

Types of group functions: Each of the group functions—AVG, SUM, MAX, MIN, COUNT, STDDEV, and VARIANCE—accepts one argument. The AVG, SUM, STDDEV, and VARIANCE functions operate only on numeric values. MAX and MIN can operate on numeric, character, or date data values. COUNT returns the number of non-NULL rows for the given expression. The example in the slide calculates the average salary, standard deviation on the salary, number of employees earning a commission, and the maximum hire date for those employees whose JOB_ID begins with SA.

Guidelines for Using Group Functions

- The data types for the arguments can be CHAR, VARCHAR2, NUMBER, or DATE.
- All group functions except COUNT(*) ignore null values. To substitute a value for null values, use the NVL function. COUNT returns either a number or zero.
- The Oracle server implicitly sorts the result set in ascending order of the grouping columns specified, when you use a GROUP BY clause. To override this default ordering,

you can use `DESC` in an `ORDER BY` clause.



Review of the GROUP BY Clause

- Syntax:

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

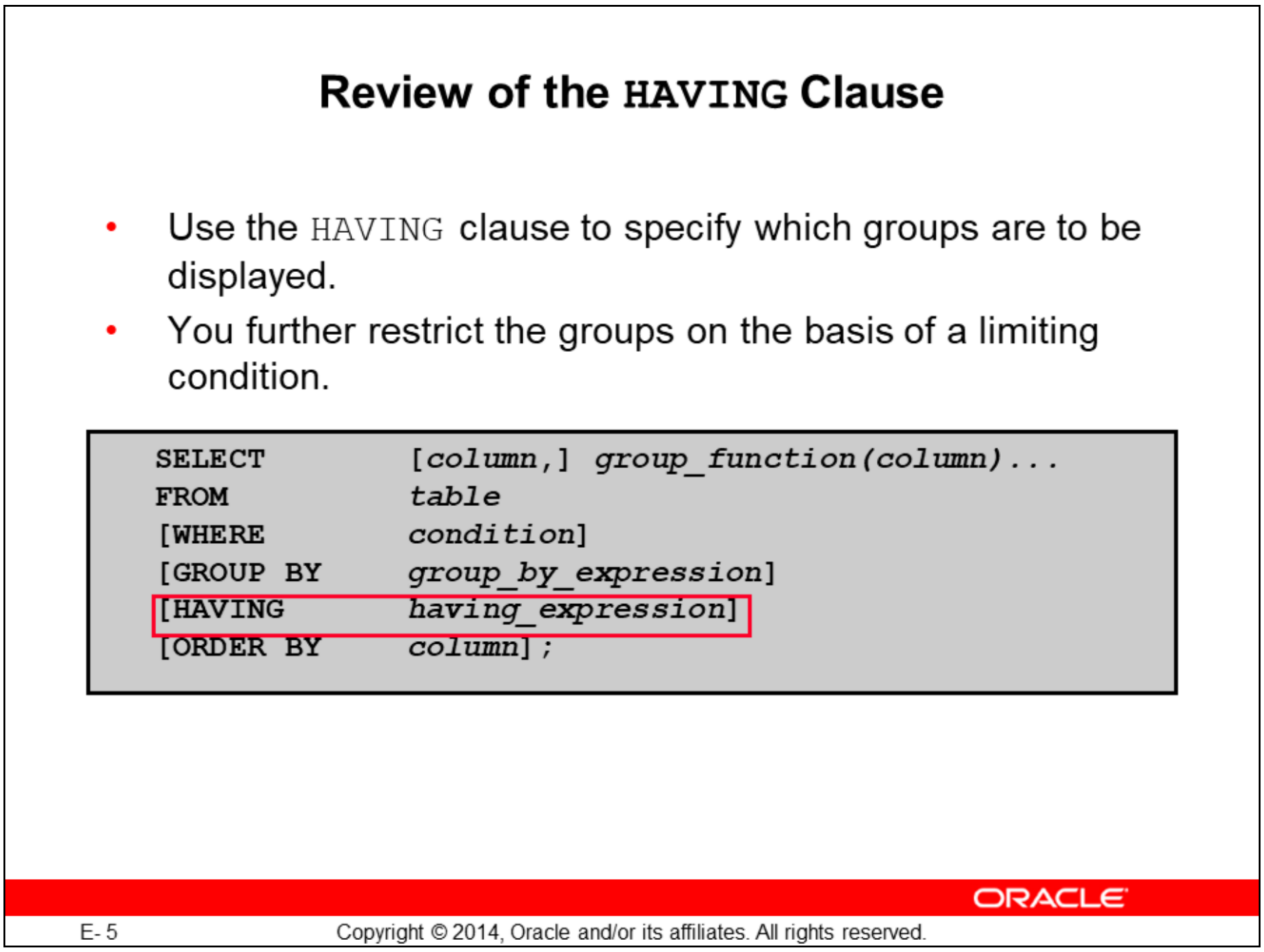
- Example:

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

ORACLE

The example illustrated in the slide is evaluated by the Oracle server as follows:

- The `SELECT` clause specifies that the following columns be retrieved:
  - Department ID and job ID columns from the `EMPLOYEES` table
  - The sum of all the salaries and the number of employees in each group that you have specified in the `GROUP BY` clause
- The `GROUP BY` clause specifies how the rows should be grouped in the table. The total salary and the number of employees are calculated for each job ID within each department. The rows are grouped by department ID and then grouped by job within each department.



Review of the HAVING Clause

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

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

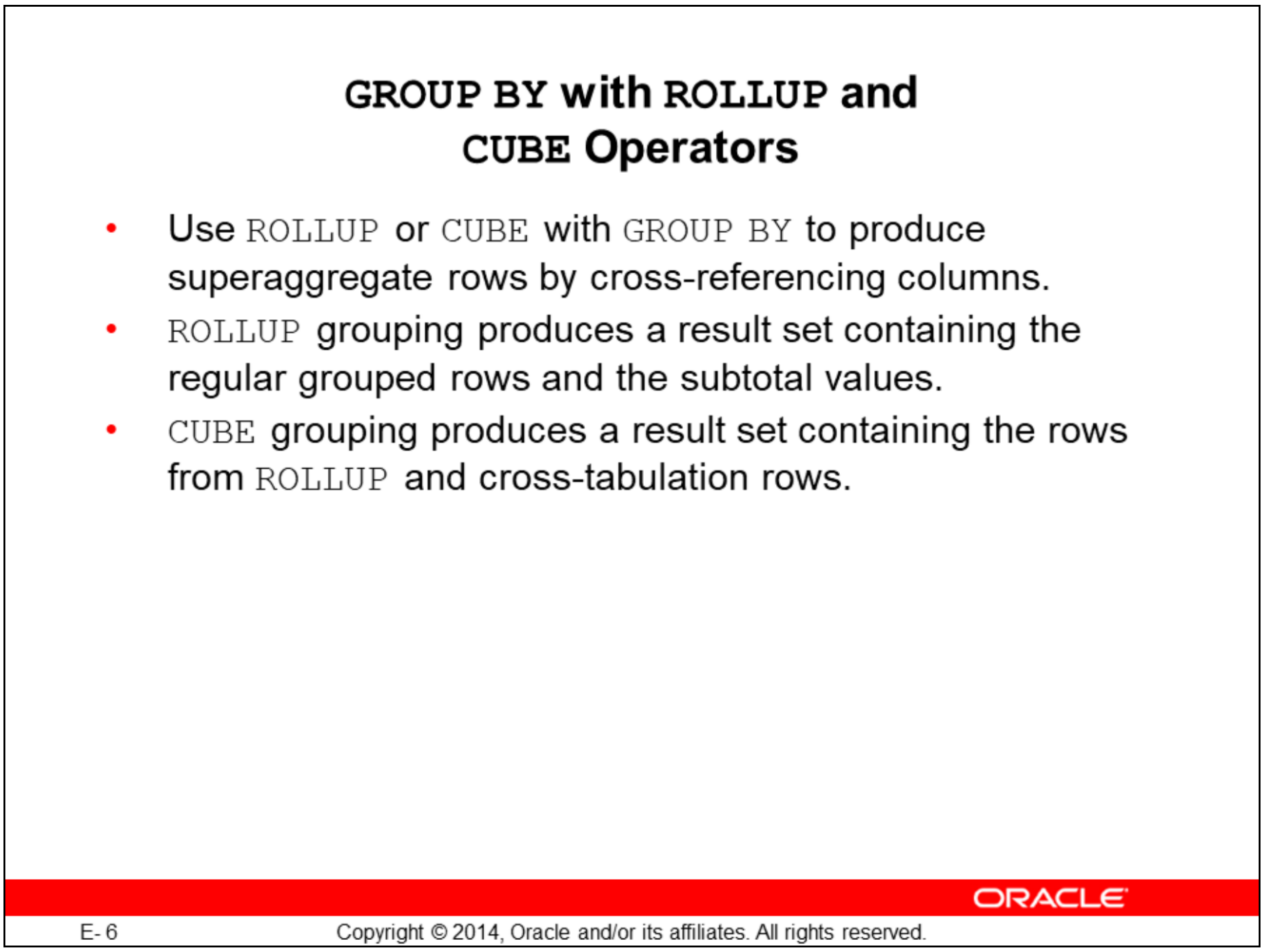
E- 5

HAVING Clause

Groups are formed and group functions are calculated before the `HAVING` clause is applied to the groups. The `HAVING` clause can precede the `GROUP BY` clause, but it is recommended that you place the `GROUP BY` clause first, because the `GROUP BY` clause is more logical than the `HAVING` clause.

The Oracle server performs the following steps when you use the `HAVING` clause:

1. It groups rows.
2. It applies the group functions to the groups and displays the groups that match the criteria in the `HAVING` clause.



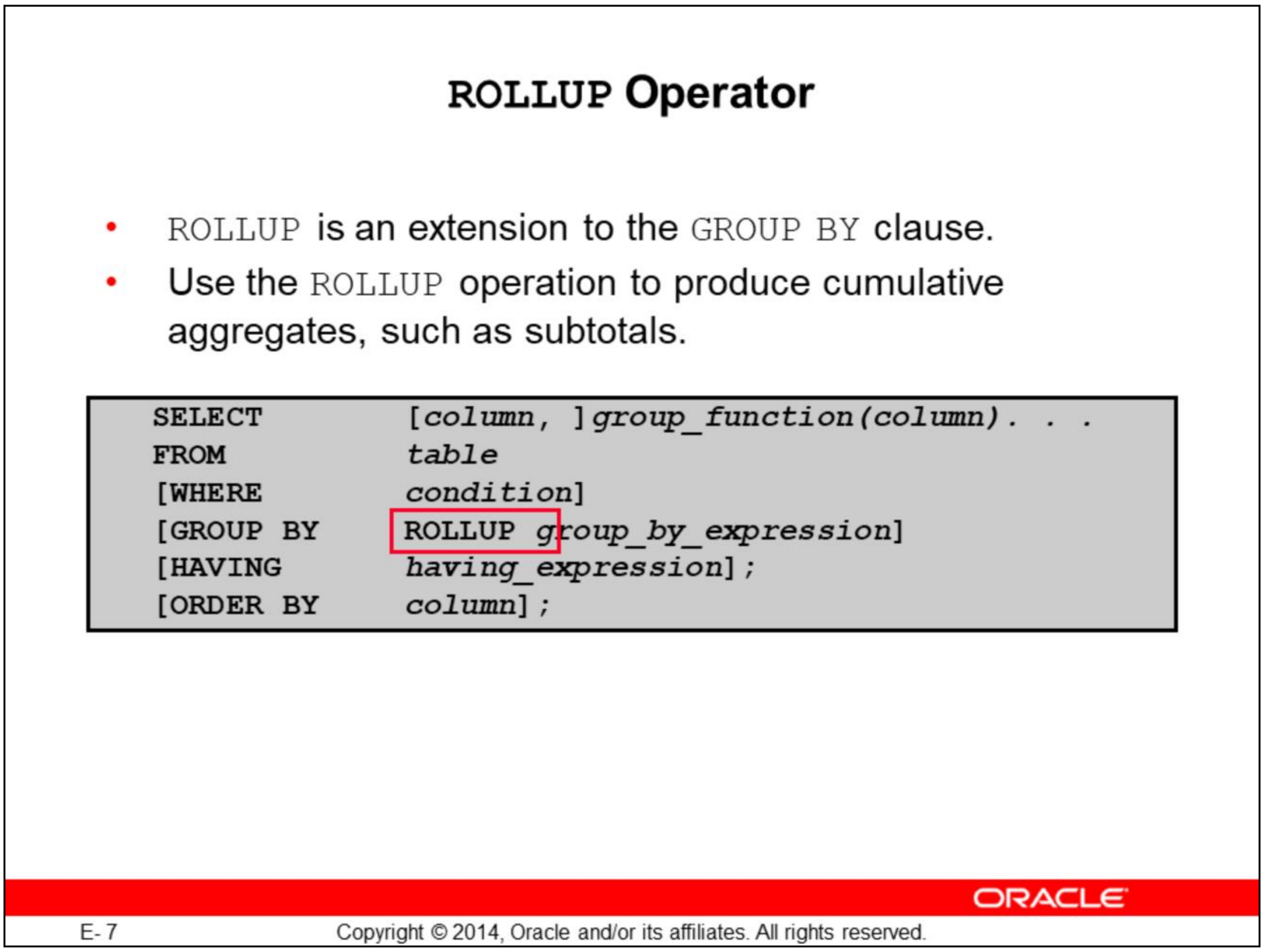
GROUP BY with ROLLUP and CUBE Operators

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

E- 6

You specify ROLLUP and CUBE operators in the GROUP BY clause of a query. ROLLUP grouping produces a result set containing the regular grouped rows and subtotal rows. The ROLLUP operator also calculates a grand total. The CUBE operation in the GROUP BY clause groups the selected rows based on the values of all possible combinations of expressions in the specification and returns a single row of summary information for each group. You can use the CUBE operator to produce cross-tabulation rows.

Note: When working with ROLLUP and CUBE, make sure that the columns following the GROUP BY clause have meaningful, real-life relationships with each other; otherwise, the operators return irrelevant information.



ROLLUP Operator

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

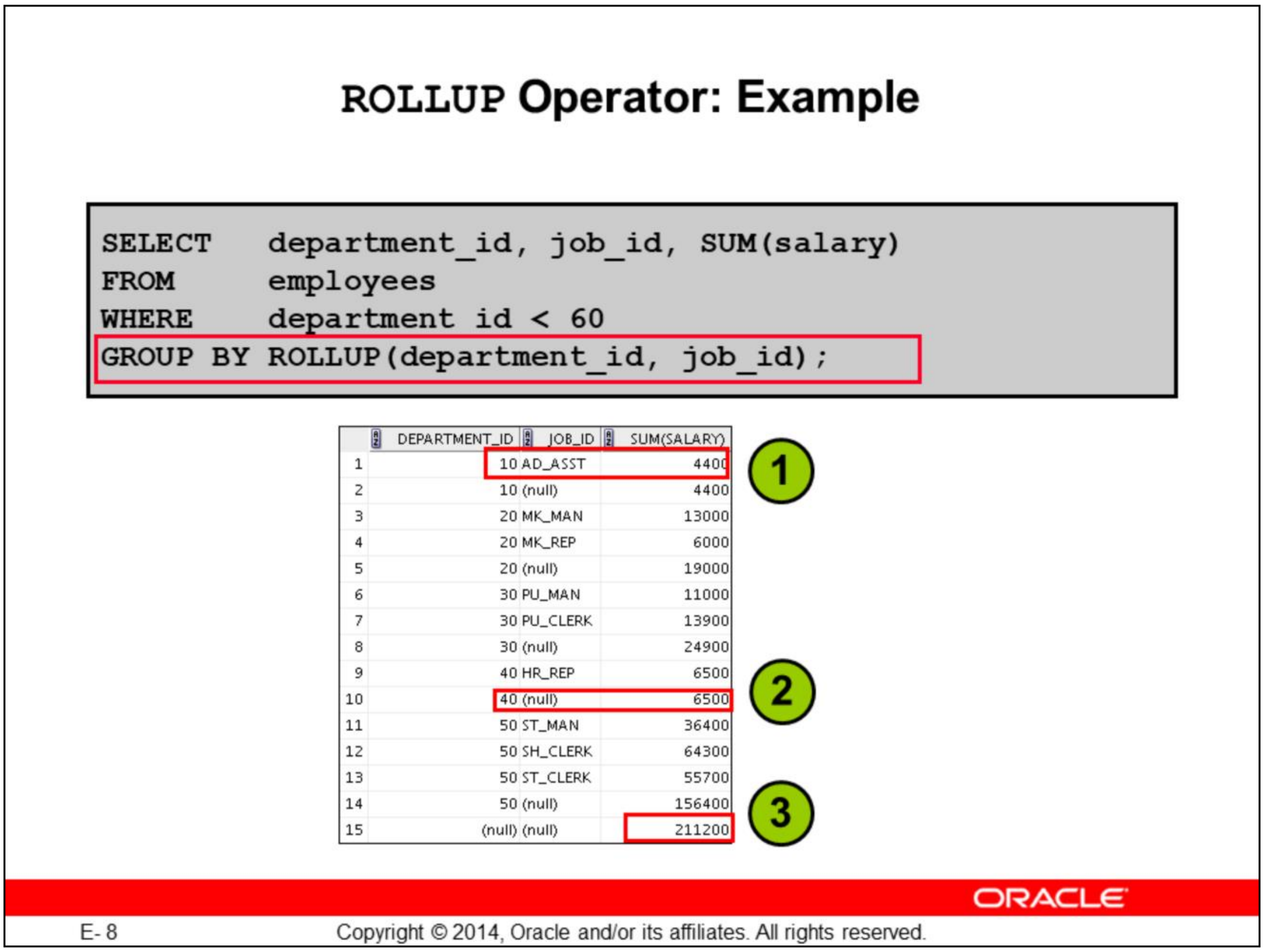
```
SELECT          [column, ]group_function(column). . .
FROM            table
[WHERE          condition]
[GROUP BY       ROLLUP group_by_expression]
[HAVING         having_expression];
[ORDER BY       column];
```

The ROLLUP operator delivers aggregates and superaggregates for expressions within a GROUP BY statement. The ROLLUP operator can be used by report writers to extract statistics and summary information from result sets. The cumulative aggregates can be used in reports, charts, and graphs.

The ROLLUP operator creates groupings by moving in one direction, from right to left, along the list of columns specified in the GROUP BY clause. It then applies the aggregate function to these groupings.

Note

- To produce subtotals in *n* dimensions (that is, *n* columns in the GROUP BY clause) without a ROLLUP operator, *n*+1 SELECT statements must be linked with UNION ALL. This makes the query execution inefficient because each of the SELECT statements causes table access. The ROLLUP operator gathers its results with just one table access. The ROLLUP operator is useful when there are many columns involved in producing the subtotals.
- Subtotals and totals are produced with ROLLUP. CUBE produces totals as well but effectively rolls up in each possible direction, producing cross-tabular data.



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)
1	10	AD_ASST	4400
2	10	(null)	4400
3	20	MK_MAN	13000
4	20	MK_REP	6000
5	20	(null)	19000
6	30	PU_MAN	11000
7	30	PU_CLERK	13900
8	30	(null)	24900
9	40	HR_REP	6500
10	40	(null)	6500
11	50	ST_MAN	36400
12	50	SH_CLERK	64300
13	50	ST_CLERK	55700
14	50	(null)	156400
15	(null)	(null)	211200

ORACLE

In the example in the slide:

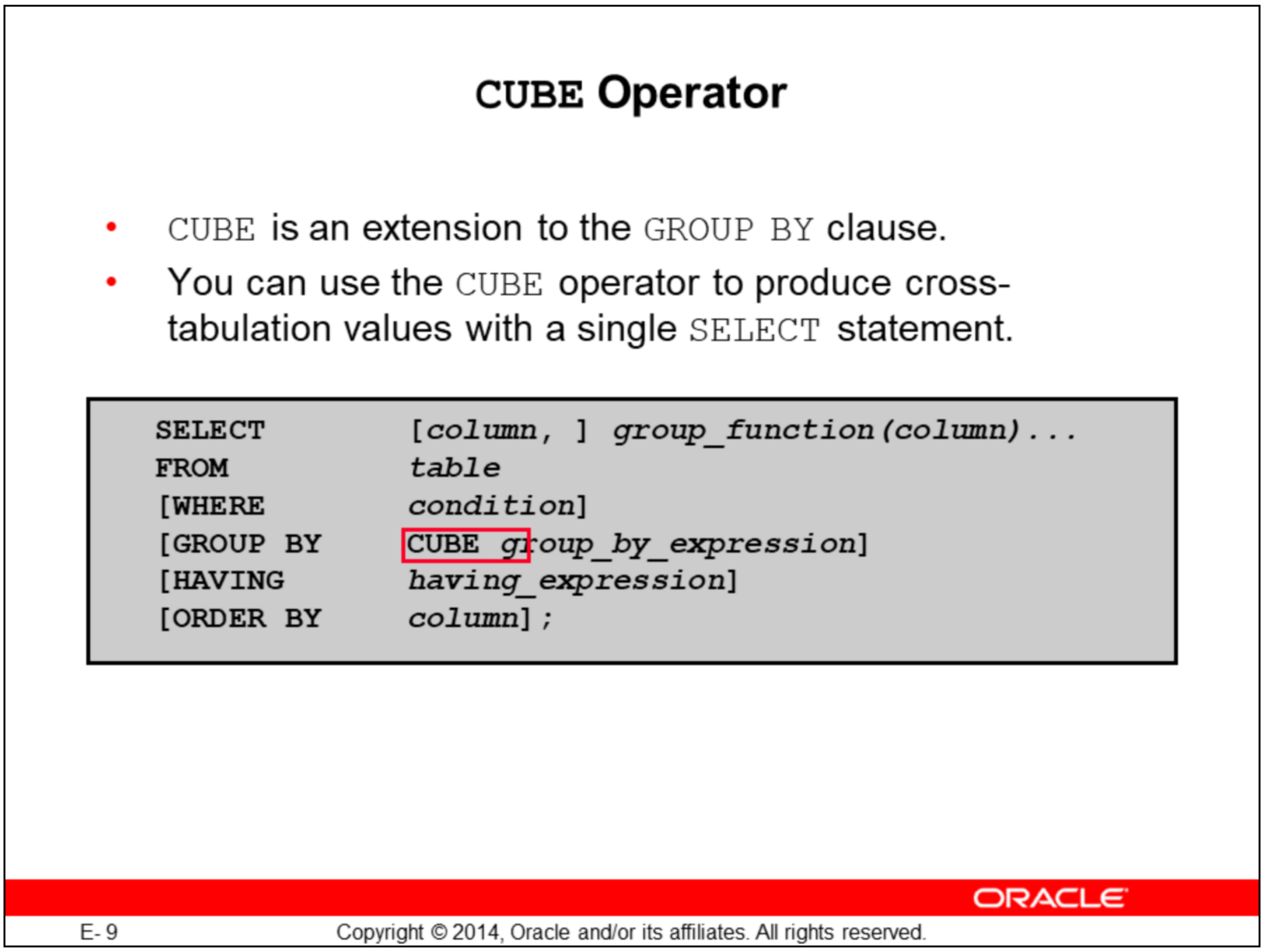
- Total salaries for every job ID within a department for those departments whose department ID is less than 60 are displayed by the GROUP BY clause
- The ROLLUP operator displays:
  - The total salary for each department whose department ID is less than 60
  - The total salary for all departments whose department ID is less than 60, irrespective of the job IDs

In this example, 1 indicates a group totaled by both DEPARTMENT_ID and JOB_ID, 2 indicates a group totaled only by DEPARTMENT_ID, and 3 indicates the grand total.

The ROLLUP operator creates subtotals that roll up from the most detailed level to a grand total, following the grouping list specified in the GROUP BY clause. First, it calculates the standard aggregate values for the groups specified in the GROUP BY clause (in the example, the sum of salaries grouped on each job within a department). Then it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. (In the example, the sum of salaries for each department is calculated, followed by the sum of salaries for all departments.)

- Given *n* expressions in the ROLLUP operator of the GROUP BY clause, the operation results in *n* + 1 (in this case, 2 + 1 = 3) groupings.

- Rows based on the values of the first *n* expressions are called rows or regular rows, and the others are called superaggregate rows.



CUBE Operator

- 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.

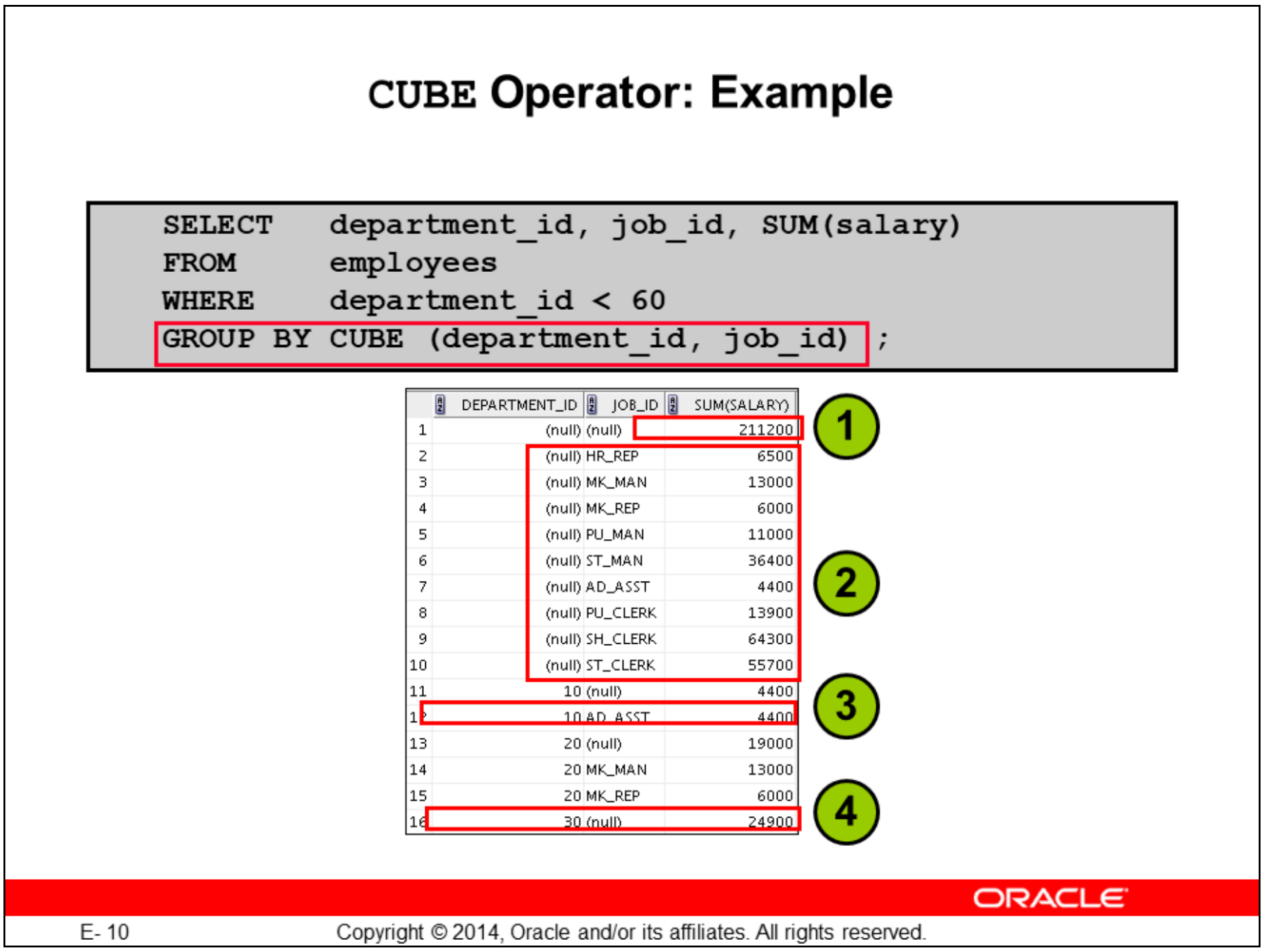
```
SELECT        [column, ] group_function(column)...
FROM          table
[WHERE        condition]
[GROUP BY     CUBE group_by_expression]
[HAVING       having_expression]
[ORDER BY     column];
```

The CUBE operator is an additional switch in the GROUP BY clause in a SELECT statement. The CUBE operator can be applied to all aggregate functions, including AVG, SUM, MAX, MIN, and COUNT. It is used to produce result sets that are typically used for cross-tabular reports. ROLLUP produces only a fraction of possible subtotal combinations, whereas CUBE produces subtotals for all possible combinations of groupings specified in the GROUP BY clause, and a grand total.

The CUBE operator is used with an aggregate function to generate additional rows in a result set. Columns included in the GROUP BY clause are cross-referenced to produce a superset of groups. The aggregate function specified in the select list is applied to these groups to produce summary values for the additional superaggregate rows. The number of extra groups in the result set is determined by the number of columns included in the GROUP BY clause.

In fact, every possible combination of the columns or expressions in the GROUP BY clause is used to produce superaggregates. If you have *n* columns or expressions in the GROUP BY clause, there will be 2^n possible superaggregate combinations. Mathematically, these combinations form an *n*-dimensional cube, which is how the operator got its name.

By using application or programming tools, these superaggregate values can then be fed into charts and graphs that convey results and relationships visually and effectively.



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)
1	(null)	(null)	211200
2	(null)	HR_REP	6500
3	(null)	MK_MAN	13000
4	(null)	MK_REP	6000
5	(null)	PU_MAN	11000
6	(null)	ST_MAN	36400
7	(null)	AD_ASST	4400
8	(null)	PU_CLERK	13900
9	(null)	SH_CLERK	64300
10	(null)	ST_CLERK	55700
11	10	(null)	4400
12	10	AD_ASST	4400
13	20	(null)	19000
14	20	MK_MAN	13000
15	20	MK_REP	6000
16	30	(null)	24900

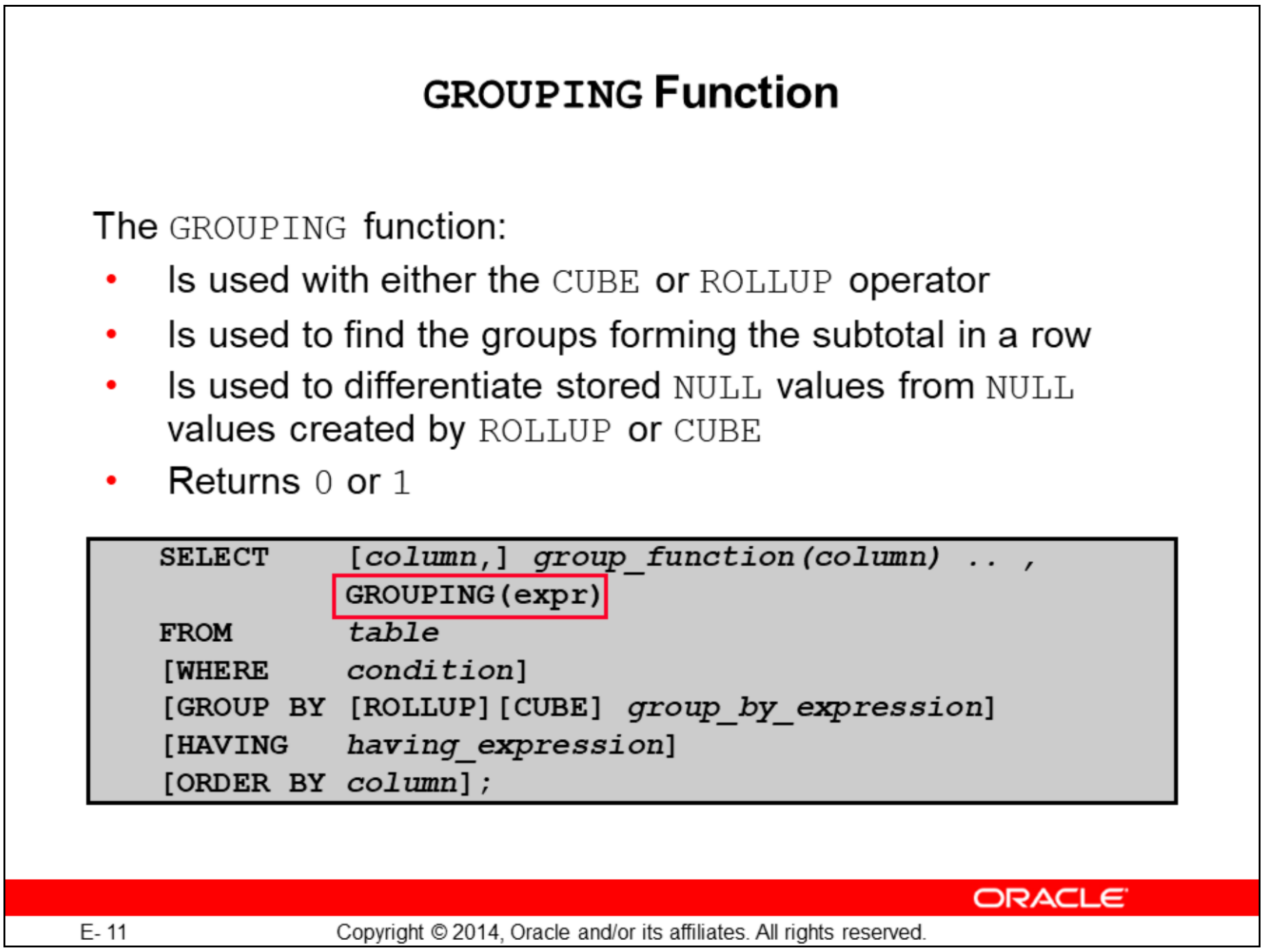
The output of the SELECT statement in the example can be interpreted as follows:

- The total salary for every job within a department (for those departments whose department ID is less than 60)
- The total salary for each department whose department ID is less than 60
- The total salary for each job irrespective of the department
- The total salary for those departments whose department ID is less than 60, irrespective of the job titles

In this example, 1 indicates the grand total, 2 indicates the rows totaled by JOB_ID alone, 3 indicates some of the rows totaled by DEPARTMENT_ID and JOB_ID, and 4 indicates some of the rows totaled by DEPARTMENT_ID alone.

The CUBE operator has also performed the ROLLUP operation to display the subtotals for those departments whose department ID is less than 60 and the total salary for those departments whose department ID is less than 60, irrespective of the job titles. Further, the CUBE operator displays the total salary for every job irrespective of the department.

Note: Similar to the ROLLUP operator, producing subtotals in *n* dimensions (that is, *n* columns in the GROUP BY clause) without a CUBE operator requires that 2^n SELECT statements be linked with UNION ALL. Thus, a report with three dimensions requires $2^3 = 8$ SELECT statements to be linked with `UNION ALL`.



GROUPING Function

The GROUPING function:

- Is used with either the CUBE or ROLLUP operator
- Is used to find the groups forming the subtotal in a row
- Is used to differentiate stored NULL values from NULL values created by ROLLUP or CUBE
- Returns 0 or 1

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

The GROUPING function can be used with either the CUBE or ROLLUP operator to help you understand how a summary value has been obtained.

The GROUPING function uses a single column as its argument. The *expr* in the GROUPING function must match one of the expressions in the GROUP BY clause. The function returns a value of 0 or 1.

The values returned by the GROUPING function are useful to:

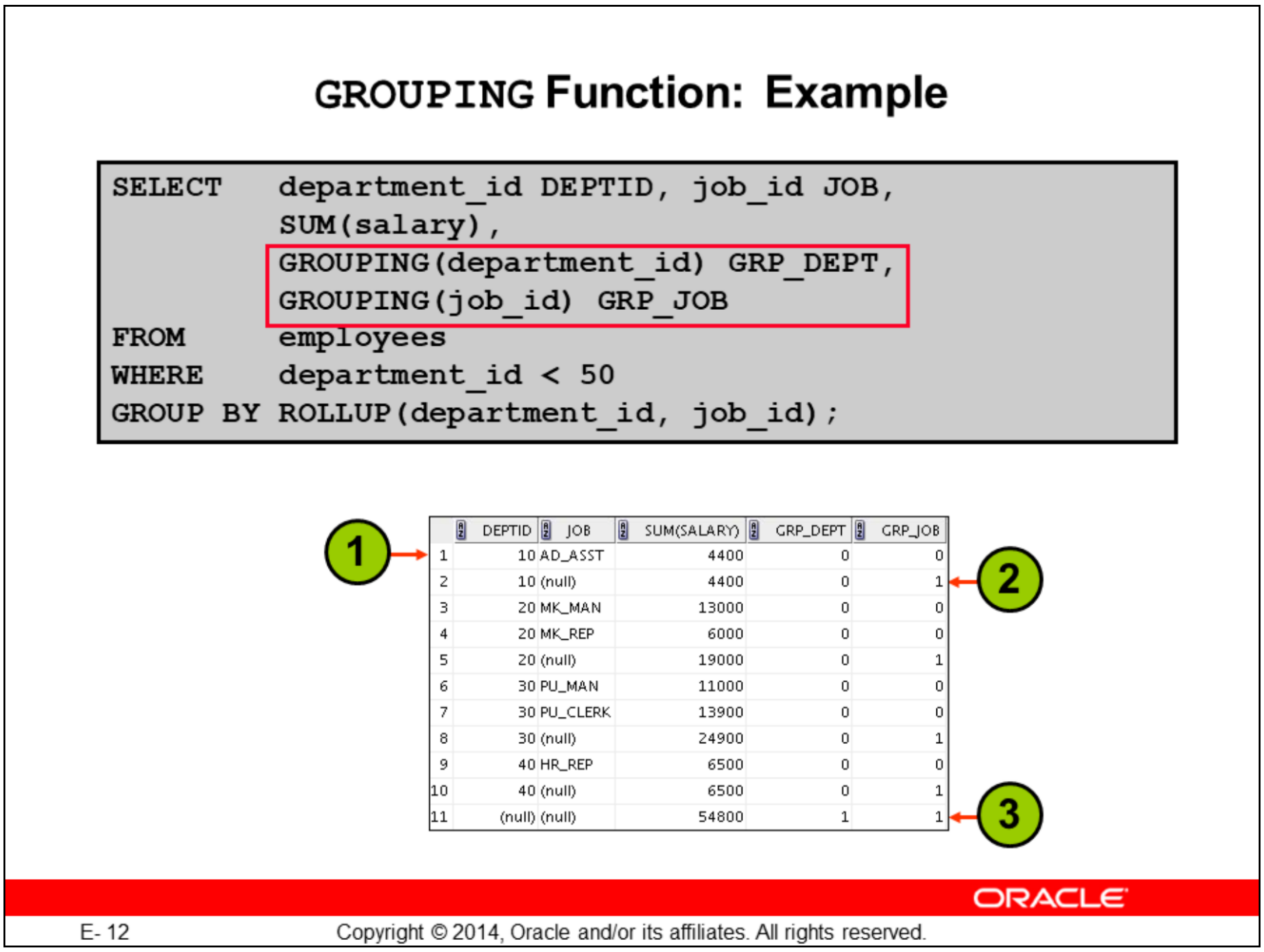
- Determine the level of aggregation of a given subtotal (that is, the group or groups on which the subtotal is based)
- Identify whether a NULL value in the expression column of a row of the result set indicates:
  - A NULL value from the base table (stored NULL value)
  - A NULL value created by ROLLUP or CUBE (as a result of a group function on that expression)

A value of 0 returned by the GROUPING function based on an expression indicates one of the following:

- The expression has been used to calculate the aggregate value.
- The NULL value in the expression column is a stored NULL value.

A value of 1 returned by the GROUPING function based on an expression indicates one of the following:

- The expression has not been used to calculate the aggregate value.
- The `NULL` value in the expression column is created by `ROLLUP` or `CUBE` as a result of grouping.



GROUPING Function: Example

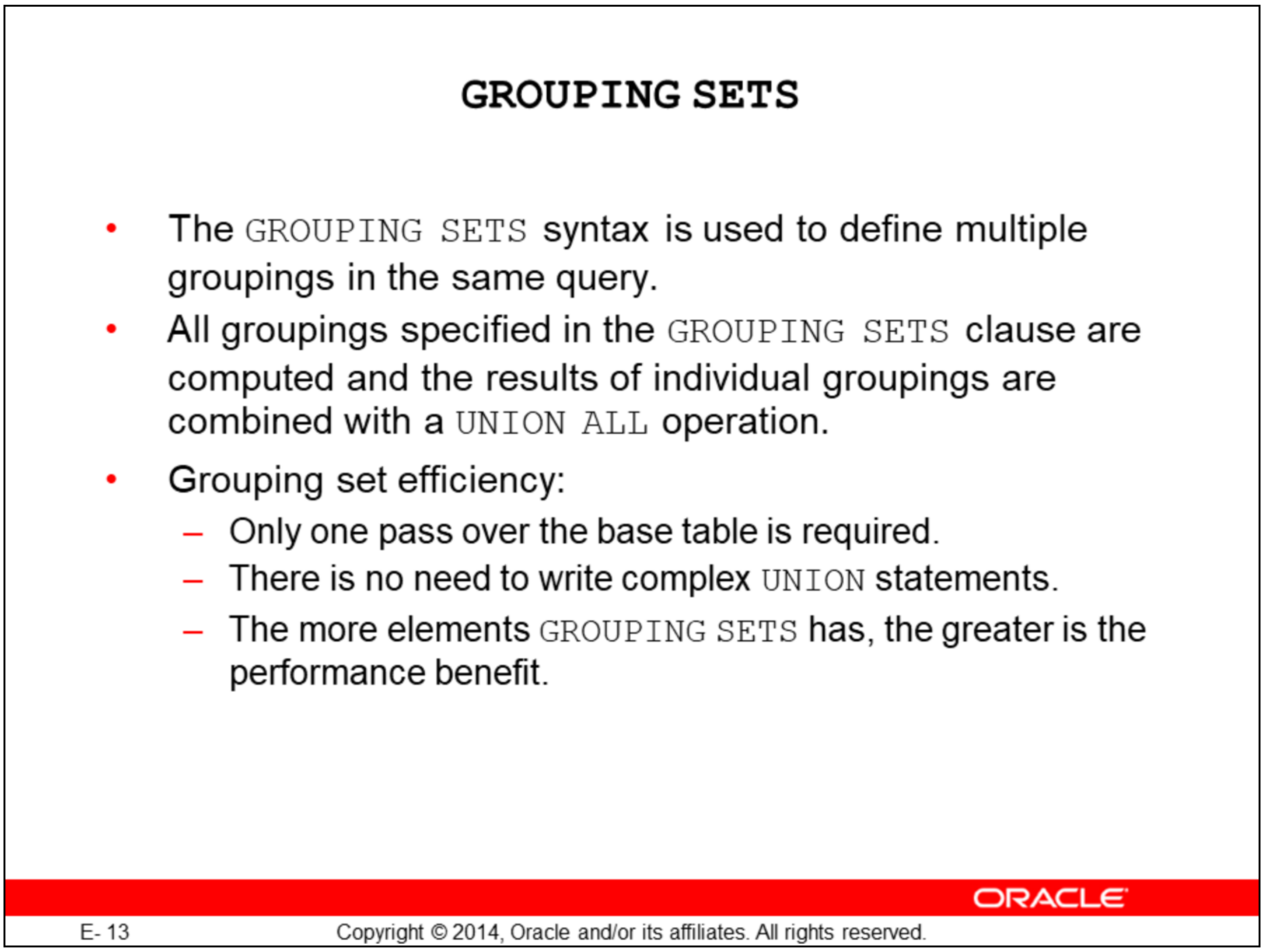
```
SELECT    department_id DEPTID, job_id JOB,
          SUM(salary),
          GROUPING(department_id) GRP_DEPT,
          GROUPING(job_id) GRP_JOB
FROM      employees
WHERE     department_id < 50
GROUP BY ROLLUP(department_id, job_id);
```

	DEPTID	JOB	SUM(SALARY)	GRP_DEPT	GRP_JOB
1	10	AD_ASST	4400	0	0
2	10	(null)	4400	0	1
3	20	MK_MAN	13000	0	0
4	20	MK_REP	6000	0	0
5	20	(null)	19000	0	1
6	30	PU_MAN	11000	0	0
7	30	PU_CLERK	13900	0	0
8	30	(null)	24900	0	1
9	40	HR_REP	6500	0	0
10	40	(null)	6500	0	1
11	(null)	(null)	54800	1	1

In the example in the slide, consider the summary value 4400 in the first row (labeled 1). This summary value is the total salary for the job ID of `AD_ASST` within department 10. To calculate this summary value, both the `DEPARTMENT_ID` and `JOB_ID` columns have been taken into account. Thus, a value of `0` is returned for both the `GROUPING(department_id)` and `GROUPING(job_id)` expressions.

Consider the summary value 4400 in the second row (labeled 2). This value is the total salary for department 10 and has been calculated by taking into account the `DEPARTMENT_ID` column; thus, a value of `0` has been returned by `GROUPING(department_id)`. Because the `JOB_ID` column has not been taken into account to calculate this value, a value of `1` has been returned for `GROUPING(job_id)`. You can observe similar output in the fifth row.

In the last row, consider the summary value 54800 (labeled 3). This is the total salary for those departments whose department ID is less than 50 and all job titles. To calculate this summary value, neither of the `DEPARTMENT_ID` and `JOB_ID` columns have been taken into account. Thus, a value of `1` is returned for both the `GROUPING(department_id)` and `GROUPING(job_id)` expressions.



GROUPING SETS

- The GROUPING SETS syntax is used to define multiple groupings in the same query.
- All groupings specified in the GROUPING SETS clause are computed and the results of individual groupings are combined 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 GROUPING SETS has, the greater is the performance benefit.

E- 13

GROUPING SETS is a further extension of the GROUP BY clause that you can use to specify multiple groupings of data. Doing so facilitates efficient aggregation and, therefore, facilitates analysis of data across multiple dimensions.

A single SELECT statement can now be written using GROUPING SETS to specify various groupings (which can also include ROLLUP or CUBE operators), rather than multiple SELECT statements combined by UNION ALL operators. For example:

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

This statement calculates aggregates over three groupings:

```
(department_id, job_id, manager_id), (department_id,
  manager_id)and (job_id, manager_id)
```

Without this feature, multiple queries combined together with UNION ALL are required to obtain the output of the preceding SELECT statement. A multiquery approach is inefficient because it requires multiple scans of the same data.

Compare the previous example with the following alternative:

```
SELECT department_id, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY CUBE(department_id, job_id, manager_id);
```

This statement computes all the 8 (2 *2 *2) groupings, though only the `(department_id, job_id, manager_id)`, `(department_id, manager_id)`, and `(job_id, manager_id)` groups are of interest to you.

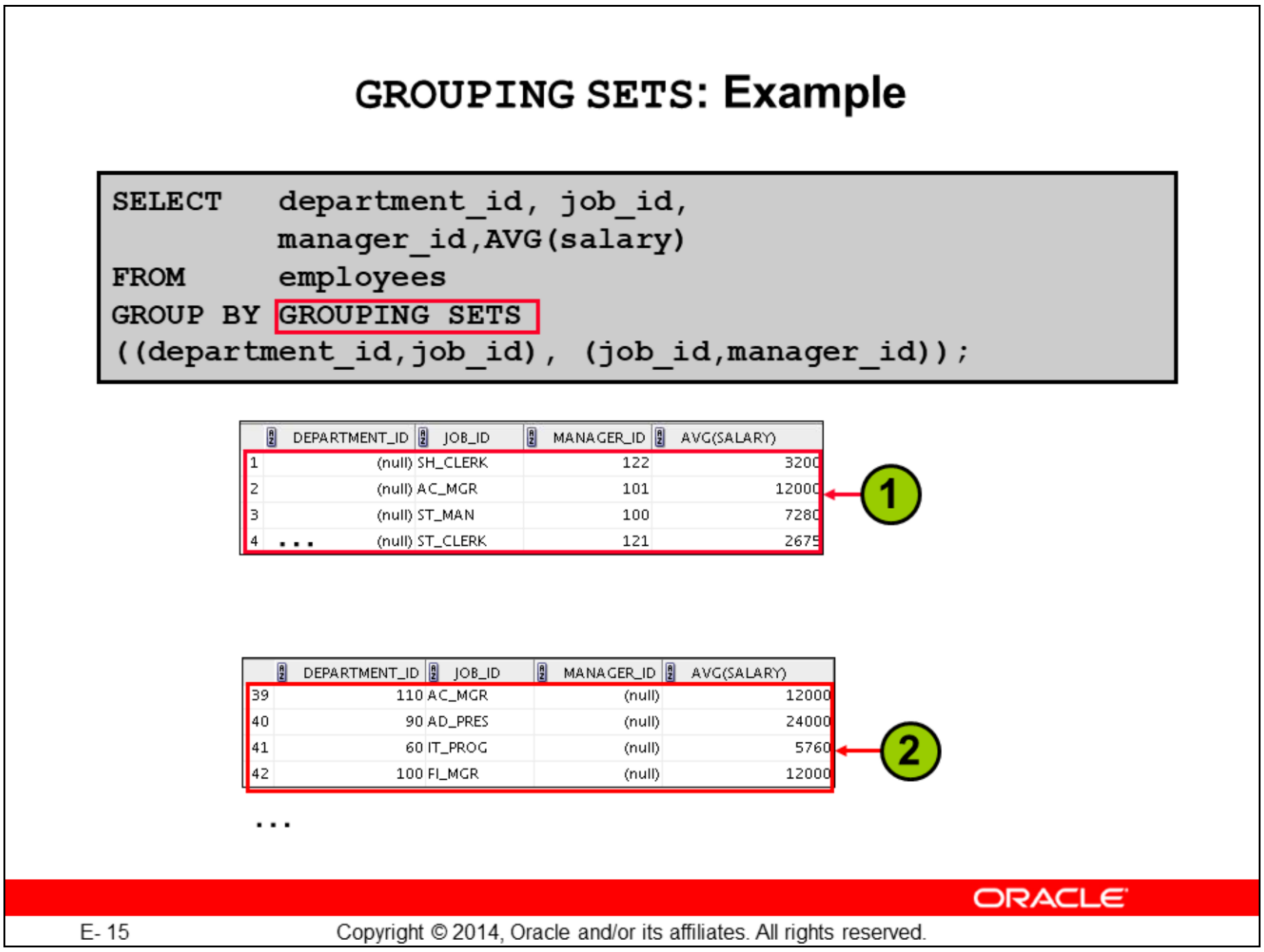
Another alternative is the following statement:

```
SELECT department_id, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY department_id, job_id, manager_id
UNION ALL
SELECT department_id, NULL, manager_id, AVG(salary)
FROM employees
GROUP BY department_id, manager_id
UNION ALL
SELECT NULL, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY job_id, manager_id;
```

This statement requires three scans of the base table, which makes it inefficient.

`CUBE` and `ROLLUP` can be thought of as grouping sets with very specific semantics and results. The following equivalencies show this fact:

`CUBE(a, b, c)` is equivalent to	`GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())`
`ROLLUP(a, b,c)` is equivalent to	`GROUPING SETS ((a, b, c), (a, b),(a), ())`



GROUPING SETS: Example

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

1

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
1	(null)	SH_CLERK	122	3200
2	(null)	AC_MGR	101	12000
3	(null)	ST_MAN	100	7280
4 ...	(null)	ST_CLERK	121	2675

2

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
39	110	AC_MGR	(null)	12000
40	90	AD_PRES	(null)	24000
41	60	IT_PROG	(null)	5760
42	100	FI_MGR	(null)	12000

...

ORACLE

E- 15

The query in the slide calculates aggregates over two groupings. The table is divided into the following groups:

- Department ID, Job ID
- Job ID, Manager ID

The average salaries for each of these groups are calculated. The result set displays the average salary for each of the two groups.

In the output, the group marked as 1 can be interpreted as the following:

- The average salary of all employees with the `SH_CLERK` job ID under manager 122 is 3,200.
- The average salary of all employees with the `AC_MGR` job ID under manager 101 is 12,000, and so on.

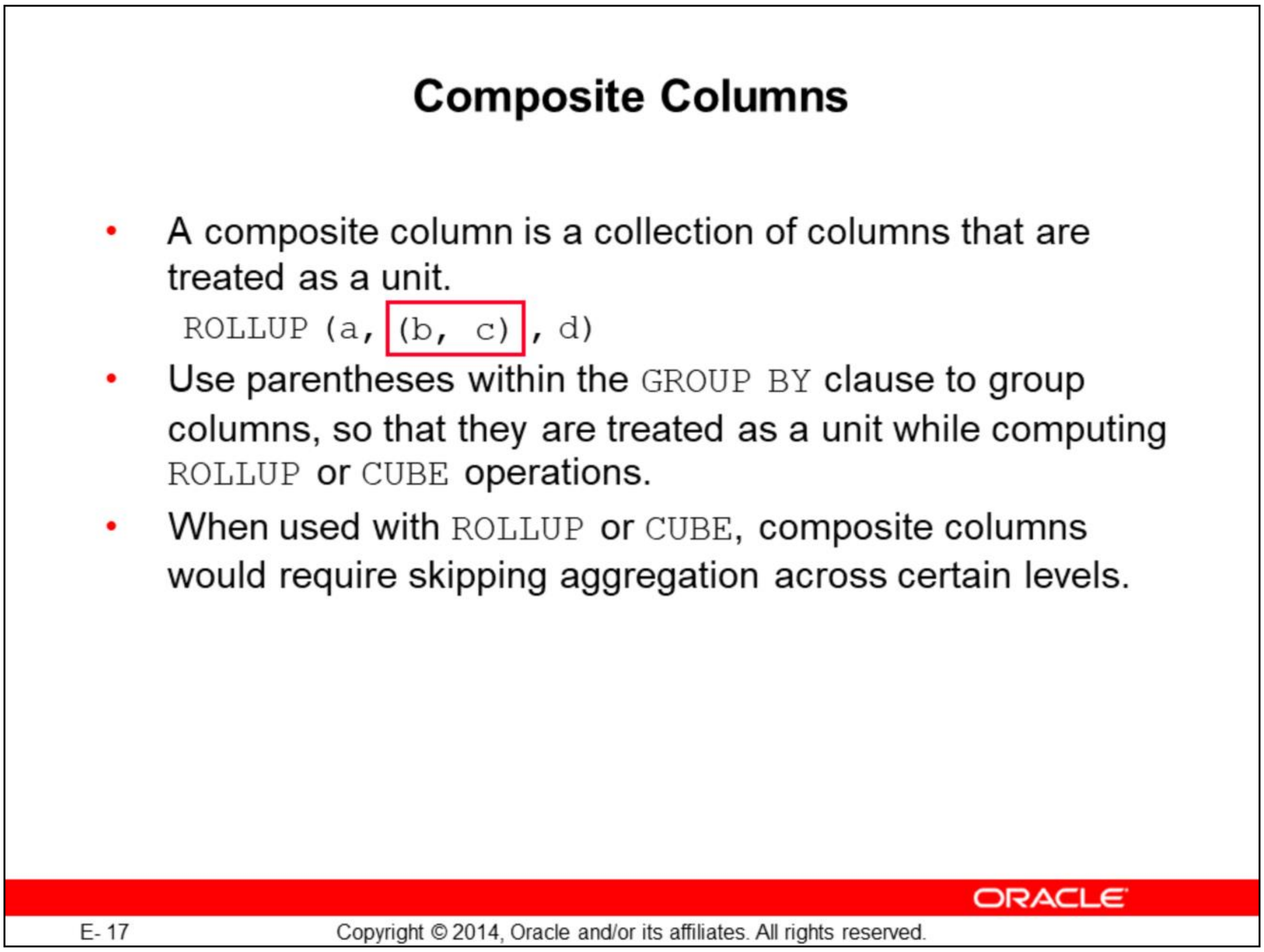
The group marked as 2 in the output is interpreted as the following:

- The average salary of all employees with the `AC_MGR` job ID in department 110 is 12,000.
- The average salary of all employees with the `AD_PRES` job ID in department 90 is 24,000, and so on.

The example in the slide can also be written as:

```
SELECT department_id, job_id, NULL as manager_id,
       AVG(salary) as AVGSAL
FROM employees
GROUP BY department_id, job_id
UNION ALL
SELECT NULL, job_id, manager_id, avg(salary) as AVGSAL
FROM employees
GROUP BY job_id, manager_id;
```

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query would need two scans of the base table, `EMPLOYEES`. This could be very inefficient. Therefore, the usage of the `GROUPING SETS` statement is recommended.



Composite Columns

- A composite column is a collection of columns that are treated as a unit.
  ```
  ROLLUP (a, (b, c), d)
  ```
- Use parentheses within the `GROUP BY` clause to group columns, so that they are treated as a unit while computing `ROLLUP` or `CUBE` operations.
- When used with `ROLLUP` or `CUBE`, composite columns would require skipping aggregation across certain levels.

E- 17

A composite column is a collection of columns that are treated as a unit during the computation of groupings. You specify the columns in parentheses as in the following statement: `ROLLUP (a, (b, c), d)`

Here, `(b, c)` forms a composite column and is treated as a unit. In general, composite columns are useful in `ROLLUP`, `CUBE`, and `GROUPING SETS`. For example, in `CUBE` or `ROLLUP`, composite columns would require skipping aggregation across certain levels.

That is, `GROUP BY ROLLUP(a, (b, c) )` is equivalent to:

```
GROUP BY a, b, c UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

Here, `(b, c)` is treated as a unit and `ROLLUP` is not applied across `(b, c)`. It is as though you have an alias—for example, `z` as an alias for `(b, c)`, and the `GROUP BY` expression reduces to: `GROUP BY ROLLUP(a, z)`.

Note: `GROUP BY( )` is typically a `SELECT` statement with `NULL` values for the columns `a` and `b` and only the aggregate function. It is generally used for generating grand totals.

```
SELECT   NULL, NULL, aggregate_col
FROM     <table_name>
```

```
GROUP BY ( );
```

Compare this with the normal `ROLLUP` as in:

```
GROUP BY ROLLUP(a, b, c)
```

This would be:

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

Similarly:

```
GROUP BY CUBE((a, b), c)
```

This would be equivalent to:

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY c UNION ALL
GROUP By ()
```

The following table shows the `GROUPING SETS` specification and the equivalent `GROUP BY` specification.

`GROUPING SETS` Statements	Equivalent `GROUP BY` Statements

`GROUP BY GROUPING SETS(a, b, c)`	`GROUP BY a UNION ALL`
	`GROUP BY b UNION ALL`
	`GROUP BY c`

`GROUP BY GROUPING SETS(a, b,(b, c))`
(The `GROUPING SETS` expression has a composite column.)
	`GROUP BY a UNION ALL`
	`GROUP BY b UNION ALL`
	`GROUP BY b, c`

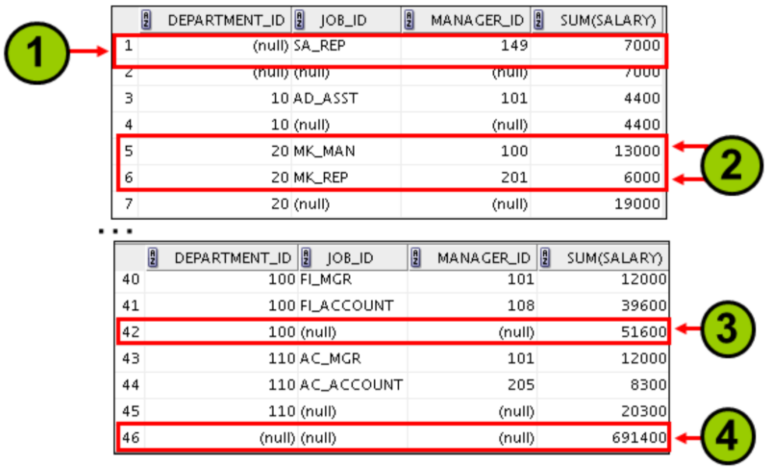
`GROUP BY GROUPING SETS((a, b, c))`	`GROUP BY a, b, c`

`GROUP BY GROUPING SETS(a, (b), ())`	`GROUP BY a UNION ALL`
	`GROUP BY b UNION ALL`
	`GROUP BY ()`

`GROUP BY GROUPING SETS`
`(a,ROLLUP(b, c))`
(The GROUPING SETS expression has a composite column.)
	`GROUP BY a UNION ALL`
	`GROUP BY ROLLUP(b, c)`

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)
1	(null)	SA_REP	149	7000
2	(null)	(null)	(null)	7000
3	10	AD_ASST	101	4400
4	10	(null)	(null)	4400
5	20	MK_MAN	100	13000
6	20	MK_REP	201	6000
7	20	(null)	(null)	19000

. . .

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
40	100	FI_MGR	101	12000
41	100	FI_ACCOUNT	108	39600
42	100	(null)	(null)	51600
43	110	AC_MGR	101	12000
44	110	AC_ACCOUNT	205	8300
45	110	(null)	(null)	20300
46	(null)	(null)	(null)	691400

ORACLE

Consider the example:

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

This query results in the Oracle server computing the following groupings:

- `(job_id, manager_id)`
- `(department_id, job_id, manager_id)`
- `(department_id)`
- Grand total

If you are interested only in specific groups, you cannot limit the calculation to those groupings without using composite columns. With composite columns, this is possible by treating `JOB_ID` and `MANAGER_ID` columns as a single unit while rolling up. Columns enclosed in parentheses are treated as a unit while computing `ROLLUP` and `CUBE`. This is illustrated in the example in the slide. By enclosing the `JOB_ID` and `MANAGER_ID` columns in parentheses, you indicate to the Oracle server to treat `JOB_ID` and `MANAGER_ID` as a single unit—that is, a composite column.

The example in the slide computes the following groupings:

- `(department_id, job_id, manager_id)`
- `(department_id)`
- `( )`

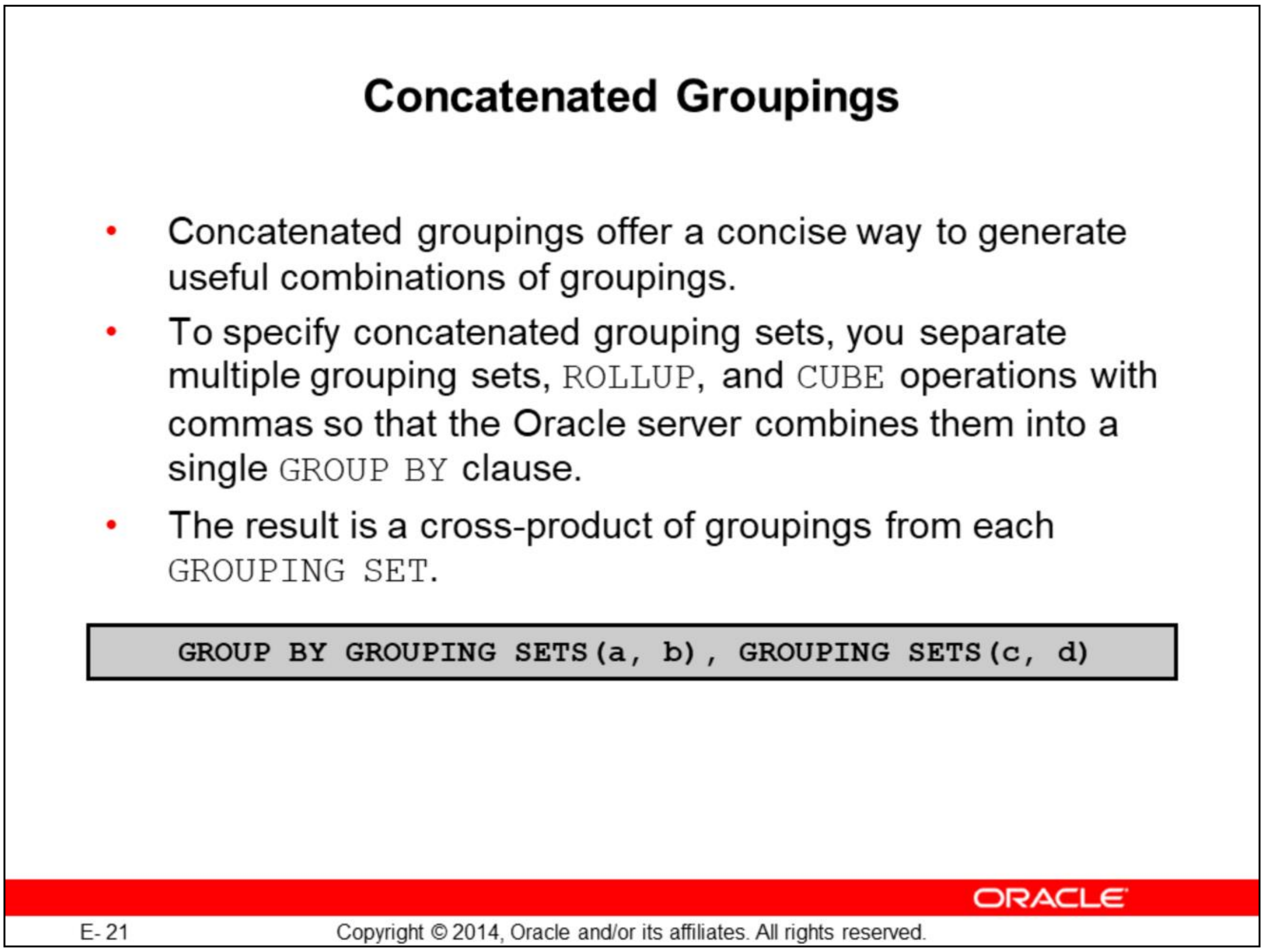
The example in the slide displays the following:

- Total salary for every job and manager (labeled 1)
- Total salary for every department, job, and manager (labeled 2)
- Total salary for every department (labeled 3)
- Grand total (labeled 4)

The example in the slide can also be written as:

```
SELECT      department_id, job_id, manager_id, SUM(salary)
FROM        employees
GROUP       BY department_id,job_id, manager_id
UNION       ALL
SELECT      department_id, TO_CHAR(NULL),TO_NUMBER(NULL),
      SUM(salary)
FROM        employees
GROUP BY    department_id
UNION ALL
SELECT  TO_NUMBER(NULL), TO_CHAR(NULL),TO_NUMBER(NULL), SUM(salary)
FROM    employees
GROUP BY ();
```

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query would need three scans of the base table, `EMPLOYEES`. This could be very inefficient. Therefore, the use of composite columns is recommended.



Concatenated Groupings

- Concatenated groupings offer a concise way to generate useful combinations of groupings.
- 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`.

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

E- 21 Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Concatenated groupings offer a concise way to generate useful combinations of groupings. The concatenated groupings are specified by listing multiple grouping sets, `CUBE`s, and `ROLLUP`s, and separating them with commas. The following is an example of concatenated grouping sets:

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

This SQL example defines the following groupings:

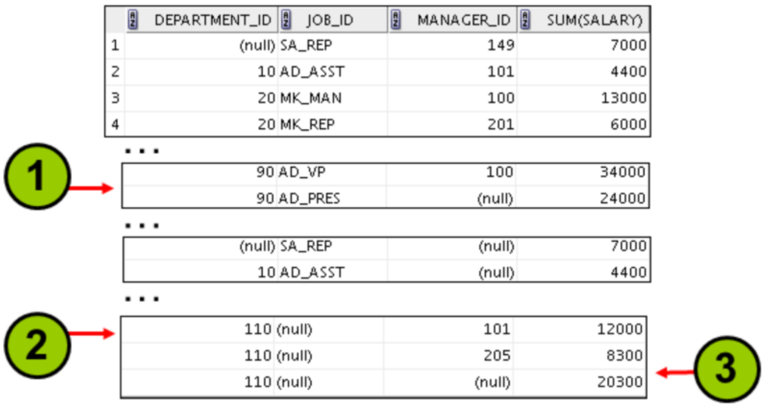
```
(a, c), (a, d), (b, c), (b, d)
```

Concatenation of grouping sets is very helpful for these reasons:

- **Ease of query development:** You need not manually enumerate all groupings.
- **Use by applications:** SQL generated by online analytical processing (OLAP) applications often involves concatenation of grouping sets, with each `GROUPING SET` defining groupings needed for a dimension.

Concatenated Groupings: Example

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



	DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
1	(null)	SA_REP	149	7000
2	10	AD_ASST	101	4400
3	20	MK_MAN	100	13000
4	20	MK_REP	201	6000
...				
	90	AD_VP	100	34000
	90	AD_PRES	(null)	24000
...				
	(null)	SA_REP	(null)	7000
	10	AD_ASST	(null)	4400
...				
	110	(null)	101	12000
	110	(null)	205	8300
	110	(null)	(null)	20300

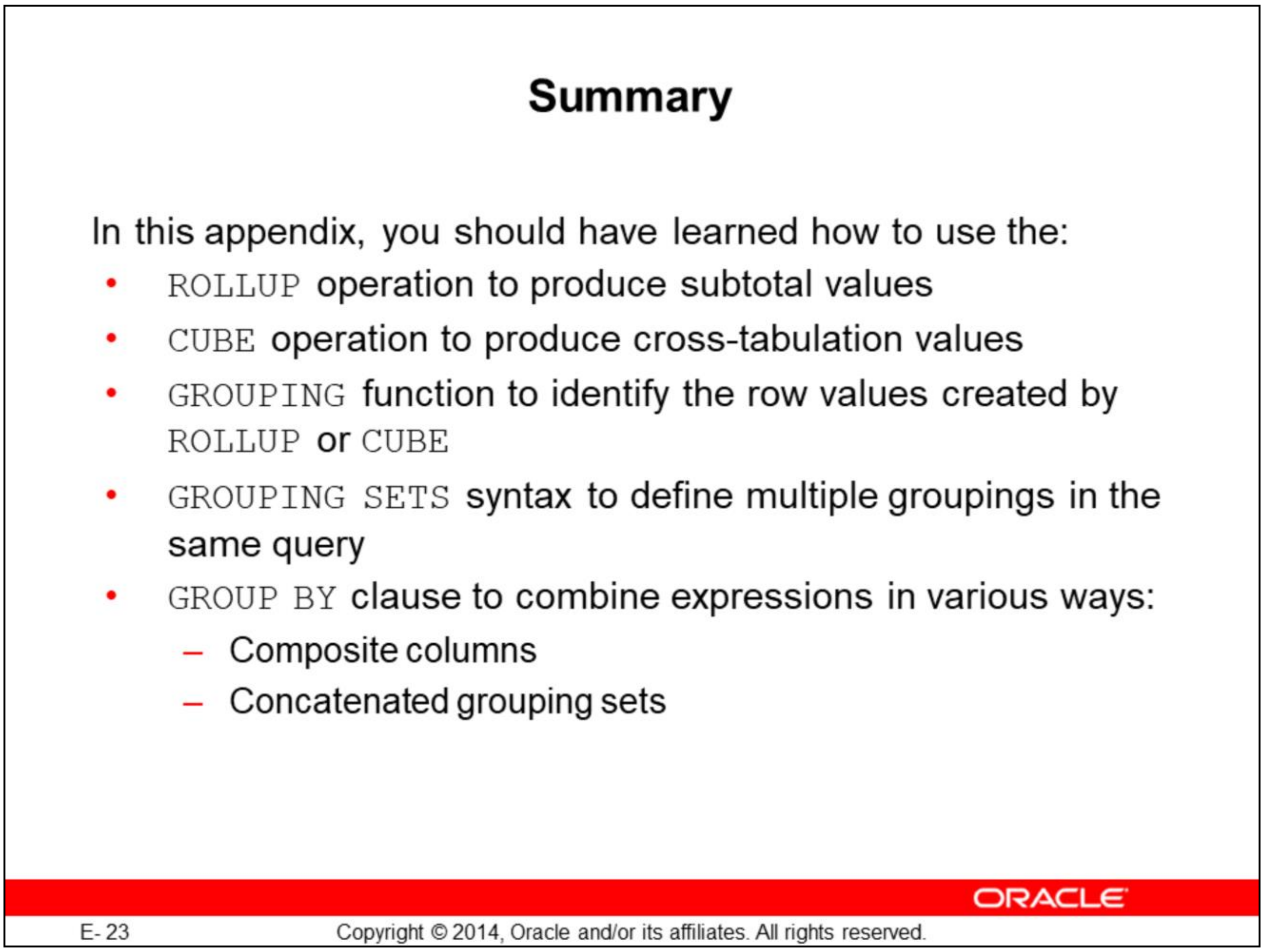
The example in the slide results in the following groupings:

- `(department_id,job_id,) (1)`
- `(department_id,manager_id)` (2)
- `(department_id)` (3)

The total salary for each of these groups is calculated.

The following is another example of a concatenated grouping.

```
SELECT department_id, job_id, manager_id, SUM(salary) totsal
FROM employees
WHERE department_id<60
GROUP BY GROUPING SETS(department_id),
GROUPING SETS (job_id, manager_id);
```



Summary

In this appendix, you should have learned how to use the:

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

E- 23

- 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 enables you to 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.