17

Enhancements to the GROUP BY Clause

Schedule:	Timing	Topic
	45 minutes	Lecture
	30 minutes	Practice
	75 minutes	Total

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**

ORACLE

Lesson Aim

In this lesson you learn how to:

- Group data for obtaining the following:
  - Subtotal values by using the `ROLLUP` operator
  - Cross-tabulation values by using the `CUBE` operator
- Use the `GROUPING` function to identify the level of aggregation in the results set produced by a `ROLLUP` or `CUBE` operator.
- Use `GROUPING SETS` to produce a single result set that is equivalent to a `UNION ALL` approach.

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%';
```

ORACLE

Group Functions

You can use the GROUP BY clause to divide the rows in a table into groups. You can then use the 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 accept one argument. The functions AVG, SUM, STDDEV, and VARIANCE operate only on numeric values. MAX and MIN can operate on numeric, character, or date data values. COUNT returns the number of nonnull 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 results 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.

Instructor Note

You can skip this slide if the students are already familiar with these concepts.

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

Review of GROUP BY Clause

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

- The SELECT clause specifies that the following columns are to 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.

DEPARTMENT_ID	JOB_ID	SUM(SALARY)	COUNT(EMPLOYEE_ID)
10	AD_ASST	4400	1
20	MK_MAN	13000	1
20	MK_REP	6000	1
50	ST_CLERK	11700	4
...			
110	AC_ACCOUNT	8300	1
110	AC_MGR	12000	1
	SA_REP	7000	1

13 rows selected.

Review of the HAVING Clause

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

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

ORACLE

The 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 it is more logical.

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

1. Groups rows
2. Applies the group functions to the groups and displays the groups that match the criteria in the HAVING clause

```
SELECT department_id, AVG(salary)
FROM   employees
GROUP  BY  department_id
HAVING AVG(salary) >9500;
```

The

DEPARTMENT_ID	AVG(SALARY)
80	10033.3333
90	19333.3333
110	10150

salary is greater than $9,500.

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.

ORACLE

GROUP BY with the ROLLUP and CUBE Operators

You specify ROLLUP and CUBE operators in the GROUP BY clause of a query. ROLLUP grouping produces a results set containing the regular grouped rows and subtotal rows. 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.

The ROLLUP and CUBE operators are available only in Oracle8*i* and later releases.

ROLLUP Operator

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

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

The ROLLUP Operator

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 results 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.

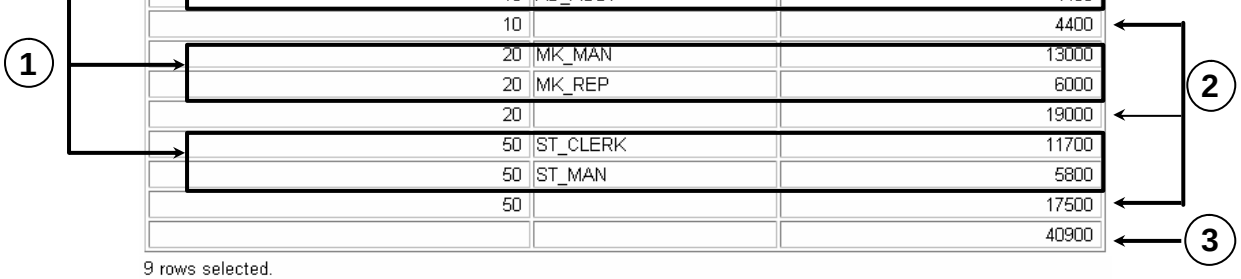
Note: To produce subtotals in *n* dimensions (that is, *n* columns in the GROUP BY clause) without a ROLLUP operator, *n*+1 SELECT statements must be linked with UNION ALL. This makes the query execution inefficient, because each of the SELECT statements causes table access. The ROLLUP operator gathers its results with just one table access. The ROLLUP operator is useful if there are many columns involved in producing the 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.



Example of a ROLLUP Operator

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 (labeled 1)
- The ROLLUP operator displays:
  - Total salary for those departments whose department ID is less than 60 (labeled 2)
  - Total salary for all departments whose department ID is less than 60, irrespective of the job IDs (labeled 3)
- All rows indicated as 1 are regular rows and all rows indicated as 2 and 3 are superaggregate rows.

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 preceding 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 $n + 1 = 2 + 1 = 3$ 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

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

- **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.**

ORACLE

The CUBE Operator

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 results sets that are typically used for cross-tabular reports. While ROLLUP produces only a fraction of possible subtotal combinations, 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 results 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 results 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)	
10	AD_ASST	4400	
10		4400	2
20	MK_MAN	13000	
20	MK_REP	6000	
20		19000	2
50	ST_CLERK	11700	
50	ST_MAN	5800	
50		17500	2
	AD_ASST	4400	3
	MK_MAN	13000	3
	MK_REP	6000	3
	ST_CLERK	11700	3
	ST_MAN	5800	3
		40900	4

1

14 rows selected.

Example of a CUBE Operator

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) is displayed by the GROUP BY clause (labeled 1)
- The total salary for those departments whose department ID is less than 60 (labeled 2)
- The total salary for every job irrespective of the department (labeled 3)
- Total salary for those departments whose department ID is less than 60, irrespective of the job titles (labeled 4)

In the preceding example, all rows indicated as 1 are regular rows, all rows indicated as 2 and 4 are superaggregate rows, and all rows indicated as 3 are cross-tabulation values.

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. Additionally, 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 2^n SELECT statements to 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

```
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.
- Using the GROUPING function, you can find the groups forming the subtotal in a row.
- Using the GROUPING function, you can differentiate stored NULL values from NULL values created by ROLLUP or CUBE.
- The GROUPING function returns 0 or 1.

ORACLE

The GROUPING Function

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/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 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.

1 → row 1 (10 | AD_ASST | 4400 | 0 | 0)
2 → row 2 (10 | | 4400 | 0 | 1)
3 → row 5 (20 | | 19000 | 0 | 1)

ORACLE

Example of a GROUPING Function

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 columns DEPARTMENT_ID and JOB_ID have been taken into account. Thus a value of 0 is returned for both the expressions GROUPING(department_id) and GROUPING(job_id).

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 column DEPARTMENT_ID; thus a value of 0 has been returned by GROUPING(department_id). Because the column JOB_ID 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 23400 (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 columns DEPARTMENT_ID and JOB_ID have been taken into account. Thus a value of 1 is returned for both the expressions GROUPING(department_id) and GROUPING(job_id).

Instructor Note

Explain that if the same example is run with the CUBE operator, it returns a results set that has 1 for GROUPING(department_id) and 0 for GROUPING(job_id) in the cross-tabulation rows, because the subtotal values are the result of grouping on job irrespective of department number.

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 greater the performance benefit.**

ORACLE

GROUPING SETS

GROUPING SETS are a further extension of the GROUP BY clause that let you specify multiple groupings of data. Doing so facilitates efficient aggregation and hence facilitates analysis of data across multiple dimensions.

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

```
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 enhancement in Oracle9*i*, multiple queries combined together with UNION ALL are required to get the output of the preceding SELECT statement. A multiquery approach is inefficient, for it requires multiple scans of the same data.

Compare the preceding statement with this alternative:

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

The preceding statement computes all the 8 (2 *2 *2) groupings, though only the groups `(department_id, job_id, manager_id)`, `(department_id, manager_id)` and `(job_id, manager_id)`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, making it inefficient.

`CUBE` and `ROLLUP` can be thought of as grouping sets with very specific semantics. 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));
```

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

1

2

26 rows selected.

GROUPING SETS: Example

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 results set displays average salary for each of the two groups.

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

- The average salary of all employees with the job ID AD_ASST in the department 10 is 4400.
- The average salary of all employees with the job ID MK_MAN in the department 20 is 13000.
- The average salary of all employees with the job ID MK_REP in the department 20 is 6000.
- The average salary of all employees with the job ID ST_CLERK in the department 50 is 2925 and so on.

- The average salary of all employees with the job ID MK_REP, who report to the manager with the manager ID 201, is 6000.
- The average salary of all employees with the job ID SA_MAN, who report to the manager with the manager ID 100, is 10500, 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. Hence 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)`
- **To specify composite columns, use the `GROUP BY` clause to 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.**

ORACLE

Composite Columns

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)` form a composite column and are 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 mean 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)` are treated as a unit and rollup will not be applied across `(b, c)`. It is as if you have an alias, for example `z`, 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. This is generally used for generating the grand totals.

```
SELECT   NULL, NULL, aggregate_col
FROM     <table_name>
GROUP BY ( );
```

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

which 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)
```

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 grouping sets specification and 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`

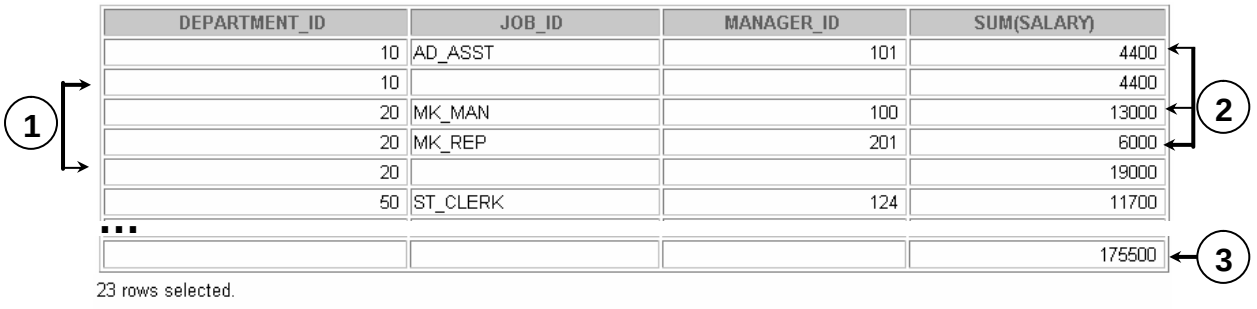
`GROUP BY GROUPING SETS((a, b, c))`	`GROUP BY a, b, c`

`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)
10	AD_ASST	101	4400
10			4400
20	MK_MAN	100	13000
20	MK_REP	201	6000
20			19000
50	ST_CLERK	124	11700
...			
			175500

23 rows selected.

Composite Columns: Example

Consider the example:

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

The preceding query results in the Oracle Server computing the following groupings:

1. (department_id, job_id, manager_id)
2. (department_id, job_id)
3. (department_id)
4. ( )

If you are just interested in grouping of lines (1), (3), and (4) in the preceding example, 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 on the slide. By enclosing JOB_ID and MANAGER_ID columns in parenthesis, we indicate to the Oracle Server to treat JOB_ID and MANAGER_ID as a single unit, as 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 department (labeled 1)
- Total salary for every department, job ID, and manager (labeled 2)
- Grand total (labeled 3)

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. Hence, 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)
```

ORACLE

Concatenated Columns

Concatenated groupings offer a concise way to generate useful combinations of groupings. The concatenated groupings are specified simply by listing multiple grouping sets, cubes, and rollups, and separating them with commas. Here is an example of concatenated grouping sets:

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

The preceding SQL 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 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)
10	AD_ASST	101	4400
20	MK_MAN	100	13000
...			
10		101	4400
20		100	13000
...			
10	AD_ASST		4400
10			4400
...			
	SA_REP		7000
			7000

49 rows selected.

1
2
3
4

ORACLE

Concatenated Groupings Example

The example in the slide results in the following groupings:

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

The total salary for each of these groups is calculated.

The example in the slide displays the following:

- Total salary for every department, job ID, manager
- Total salary for every department, manager ID
- Total salary for every department, job ID
- Total salary for every department

For easier understanding, the details for the department 10 are highlighted in the output.

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**

ORACLE

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.

Practice 17 Overview

This practice covers the following topics:

- **Using the `ROLLUP` operator**
- **Using the `CUBE` operator**
- **Using the `GROUPING` function**
- **Using `GROUPING SETS`**

Practice 17 Overview

In this practice, you use the `ROLLUP` and `CUBE` operators as extensions of the `GROUP BY` clause. You will also use `GROUPING SETS`.

Practice 2:

1. Write a query to display the following for those employees whose manager ID is less than 120:

- Manager ID
- Job ID and total salary for every job ID for employees who report to the same manager
- Total salary of those managers
- Total salary of those managers, irrespective of the job IDs

MANAGER_ID	JOB_ID	SUM(SALARY)
100	AD_VP	34000
100	MK_MAN	13000
100	SA_MAN	10500
100	ST_MAN	5800
100		63300
101	AC_MGR	12000
101	AD_ASST	4400
101		16400
102	IT_PROG	9000
102		9000
103	IT_PROG	10200
103		10200
		98900

13 rows selected.

Practice 17 (continued)

2. Observe the output from question 1. Write a query using the `GROUPING` function to determine whether the `NULL` values in the columns corresponding to the `GROUP BY` expressions are caused by the `ROLLUP` operation.

MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
100	AD_VP	34000	0	0
100	MK_MAN	13000	0	0
100	SA_MAN	10500	0	0
100	ST_MAN	5800	0	0
100		63300	0	1
101	AC_MGR	12000	0	0
101	AD_ASST	4400	0	0
101		16400	0	1
102	IT_PROG	9000	0	0
102		9000	0	1
103	IT_PROG	10200	0	0
103		10200	0	1
		98900	1	1

13 rows selected.

Practice 17 (continued)

3. Write a query to display the following for those employees whose manager ID is less than 120:
   - Manager ID
   - Job and total salaries for every job for employees who report to the same manager
   - Total salary of those managers
   - Cross-tabulation values to display the total salary for every job, irrespective of the manager
   - Total salary irrespective of all job titles

MANAGER_ID	JOB_ID	SUM(SALARY)
100	AD_VP	34000
100	MK_MAN	13000
100	SA_MAN	10500
100	ST_MAN	5800
100		63300
101	AC_MGR	12000
101	AD_ASST	4400
101		16400
102	IT_PROG	9000
102		9000
103	IT_PROG	10200
103		10200
	AC_MGR	12000
	AD_ASST	4400
MANAGER_ID	JOB_ID	SUM(SALARY)
	AD_VP	34000
	IT_PROG	19200
	MK_MAN	13000
	SA_MAN	10500
	ST_MAN	5800
		98900

20 rows selected.

Practice 17 (continued)

4. Observe the output from question 3. Write a query using the `GROUPING` function to determine whether the `NULL` values in the columns corresponding to the `GROUP BY` expressions are caused by the `CUBE` operation.

MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
100	AD_VP	34000	0	0
100	MK_MAN	13000	0	0
100	SA_MAN	10500	0	0
100	ST_MAN	5800	0	0
100		63300	0	1
101	AC_MGR	12000	0	0
101	AD_ASST	4400	0	0
101		16400	0	1
102	IT_PROG	9000	0	0
102		9000	0	1
103	IT_PROG	10200	0	0
103		10200	0	1
	AC_MGR	12000	1	0
	AD_ASST	4400	1	0
MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
	AD_VP	34000	1	0
	IT_PROG	19200	1	0
	MK_MAN	13000	1	0
	SA_MAN	10500	1	0
	ST_MAN	5800	1	0
		98900	1	1

20 rows selected.

Practice 17 (continued)

5. Using `GROUPING SETS`, write a query to display the following groupings :
   - `department_id, manager_id, job_id`
   - `department_id, job_id`
   - `manager_id, job_id`

The query should calculate the sum of the salaries for each of these groups.

DEPARTMENT_ID	MANAGER_ID	JOB_ID	SUM(SALARY)
10	101	AD_ASST	4400
20	100	MK_MAN	13000
20	201	MK_REP	6000
50	124	ST_CLERK	11700
50	100	ST_MAN	5800
60	102	IT_PROG	9000
60	103	IT_PROG	10200
80	100	SA_MAN	10500
80	149	SA_REP	19600
90		AD_PRES	24000
90	100	AD_VP	34000
110	205	AC_ACCOUNT	8300
110	101	AC_MGR	12000
	149	SA_REP	7000
...			
	100	MK_MAN	13000
	100	SA_MAN	10500
	100	ST_MAN	5800
	101	AC_MGR	12000
	101	AD_ASST	4400
	102	IT_PROG	9000
	103	IT_PROG	10200
	124	ST_CLERK	11700
	149	SA_REP	26600
	201	MK_REP	6000
	205	AC_ACCOUNT	8300
		AD_PRES	24000

40 rows selected.

Instructor Note

Analytical Functions

Oracle8*i*, release 2 (8.1.6) introduces a set of analytical group functions that provide the use of flexible and powerful calculation expressions. These analytical functions eliminate complex programming outside of standard SQL for calculations such as moving averages, rankings, and lead and lag comparisons.

In Oracle8*i*, release 2, each group defined with `GROUP BY` clause in a `SELECT` statement is called a *partition.* A query result set may have just one partition holding all the rows, a few large partitions, or many small partitions holding just a few rows each. Analytical functions are applied to each row in each partition.

RANK Function

The `RANK` function produces an ordered ranking of rows starting with a rank of one. Users specify an optional `PARTITION` clause and a required `ORDER BY` clause.The `PARTITION` keyword is used to define where the rank resets. The specific column that is ranked is determined by the `ORDER BY` clause. If no partition is specified, ranking is performed over the entire result set. `RANK` assigns a rank of 1 to the smallest value unless descending order is used.

In the following example, the query ranks managers for each department based on the total salary of all employees working under that manager.

```
SELECT  department_id deptno, job_id job, SUM(salary),
        RANK() OVER(PARTITION BY department_id ORDER BY
                SUM(salary) DESC)
        AS jobdep_rank,
        RANK() OVER(ORDER BY SUM(salary) DESC) AS sumsal_rank
FROM     employees
GROUP BY department_id, job_id;
```

```
    DEPTNO JOB        SUM(SALARY) JOBDEP_RANK SUMSAL_RANK
---------- ---------- ----------- ----------- -----------
        90 AD_VP            34000           1           1
        90 AD_PRES          24000           2           2
        80 SA_REP           19600           1           3
        60 IT_PROG          19200           1           4
        20 MK_MAN           13000           1           5
       110 AC_MGR           12000           1           6
        50 ST_CLERK         11700           1           7
        80 SA_MAN           10500           2           8
       110 AC_ACCOUNT        8300           2           9
           SA_REP             7000           1          10
        20 MK_REP            6000           2          11
        50 ST_MAN             5800           2          12
        10 AD_ASST           4400           1          13

13 rows selected.
```

Note: For ranking in groups provided by `CUBE` and `ROLLUP`, use `GROUPING()` flags in the `PARTITION BY` clause to trigger resetting.

Instructor Note (continued)

CUME_DIST Function

The cumulative distribution function computes the relative position of a value relative to the other values in its group (partition.) The CUME_DIST function defines the fraction of the rows, in the partition of a given row, that come before or are ties with the current value. It returns the results as a decimal value between zero and one, excluding zero and including one. The results of a CUME_DIST function are often called the percentile values. Default order is ascending, meaning that the lowest value in a partition gets the lowest CUME_DIST.

```
SELECT  department_id DEPTNO, job_id JOB, SUM(salary),
         CUME_DIST() OVER(PARTITION BY department_id ORDER BY
          SUM(salary) DESC)
          AS cume_dist_per_dep
FROM     employees
GROUP BY department_id, job_id
ORDER BY department_id, SUM(salary);
```

```
  DEPTNO JOB        SUM(SALARY) CUME_DIST_PER_DEP
---------- ---------- ----------- -----------------
        10 AD_ASST           4400                 1
        20 MK_REP            6000                 1
        20 MK_MAN           13000                .5
        50 ST_MAN            5800                 1
        50 ST_CLERK         11700                .5
        60 IT_PROG          19200                 1
        80 SA_MAN           10500                 1
        80 SA_REP           19600                .5
        90 AD_PRES          24000                 1
        90 AD_VP            34000                .5
       110 AC_ACCOUNT        8300                 1
       110 AC_MGR           12000                .5
           SA_REP            7000                 1

13 rows selected.
```

PERCENT_RANK Function:

This function returns the rank of a value relative to a group of values. It returns values in the range of zero to one. The formula used by this function is:

(rank of row in its partition - 1) / (number of rows in the partition - 1)