

Database Programming

Using Rollup and Cube Operations, and Grouping Sets

Objectives

This lesson covers the following objectives:

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

Purpose

Let's develop the problem that you were presented with in the last lesson just a little further. To find the average height of all students, you use this query:

```
SELECT AVG(height) FROM students;
```

If you want to know the average height of the students based on their years in school, you could write a number of different SQL statements like this:

```
SELECT AVG(height) FROM students WHERE year_in_school = 10;  
SELECT AVG(height) FROM students WHERE year_in_school = 11;  
SELECT AVG(height) FROM students WHERE year_in_school = 12;
```

Purpose (cont.)

Or you could simplify the problem by writing just one statement containing the GROUP BY and HAVING clauses.

What if, once you have selected your groups and computed your aggregates across these groups, you also wanted subtotals per group and a grand total of all the rows selected.

Purpose (cont.)

You could import the results into a spreadsheet application, get out your calculator, or compute the totals manually on paper using arithmetic. But better still, you could use some of the extensions to the GROUP BY clause specifically created for this purpose: ROLLUP, CUBE, and GROUPING SETS.

Using these extensions requires less work on your part and they are all highly efficient to use, from the point of view of the database.

ROLLUP

In GROUP BY queries, you are quite often required to produce subtotals and totals, and the ROLLUP operation can do that for you.

ROLLUP (cont.)

The action of ROLLUP is straightforward: it creates subtotals that roll up from the most detailed level to a grand total. ROLLUP uses an ordered list of grouping columns in its argument list.

- First, it calculates the standard aggregate values specified in the GROUP BY clause.
- Next, it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns.
- Finally, it creates a grand total.

ROLLUP Result Table

In the result table below, the rows highlighted in red are generated by the ROLLUP operation:

```
SELECT department_id, job_id, SUM(salary)
FROM   employees
WHERE  department_id < 50
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
		23400

← Subtotal for dept_id 10

← Subtotal for dept_id 20

← Grand Total for report

ROLLUP Result Formula

The number of columns or expressions that appear in the ROLLUP argument list determines the number of groupings. The formula is (number of columns) + 1 where number of columns is the number of columns listed in the ROLLUP argument list.

ROLLUP Result Formula (cont.)

In the example query below, two columns are listed in the ROLLUP argument list and, therefore, you will see three values generated automatically.

```
SELECT department_id, job_id, SUM(salary)
FROM   employees
WHERE  department_id < 50
GROUP BY ROLLUP (department_id, job_id)
```

Without ROLLUP

If you use GROUP BY without ROLLUP for the same query, what would the results look like?

```
SELECT department_id, job_id, SUM(salary)
FROM   employees
WHERE  department_id < 50
GROUP BY (department_id, job_id)
```

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000

You would have to execute multiple queries to get the subtotals produced by ROLLUP.

CUBE

CUBE, like ROLLUP, is an extension to the GROUP BY clause. It produces cross-tabulation reports.

It can be applied to all aggregate functions including AVG, SUM, MIN, MAX, and COUNT.

Columns listed in the GROUP BY clause are cross-referenced to create a superset of groups. The aggregate functions specified in the SELECT list are applied to this group to create summary values for the additional super-aggregate rows.

CUBE (cont.)

Every possible combination of rows is aggregated by CUBE. If you have n columns in the GROUP BY clause, there will be 2^n possible super-aggregate combinations. Mathematically these combinations form an n -dimensional cube, which is how the operator got its name.

CUBE (cont.)

CUBE is typically most suitable in queries that use columns from separate tables rather than separate columns from a single table.

Imagine, for example, a user querying the Sales table for a company like AMAZON.COM. A commonly requested cross-tabulation report might include subtotals for all possible combinations of sales across a Month, Region, and Product.

CUBE (cont.)

An analysis of all possible subtotal combinations is commonplace. In contrast, a cross-tabulation showing all possible combinations of year, month, and day would have several values of limited interest because there is a natural hierarchy in the time table.

Subtotals such as profit by day of month summed across year would be unnecessary in most analyses. Relatively few users need to ask "What were the total sales for the 16th of each month across the year?"

CUBE (cont.)

In the following statement, the rows in red are generated by the CUBE operation:

```
SELECT department_id, job_id, SUM(salary)
FROM   employees
WHERE  department_id < 50
GROUP BY CUBE (department_id, job_id)
```

DEPARTMENT_ID	JOB_ID	SUM(SALARY)	
-	-	23400	← Total for report
-	MK_MAN	13000	← Subtotal for MK_MAN
-	MK_REP	6000	← Subtotal for MK_REP
-	AD_ASST	4400	← Subtotal for AD_ASST
10	-	4400	← Subtotal for dept 10
10	AD_ASST	4400	
20	-	19000	← Subtotal for dept 20
20	MK_MAN	13000	
20	MK_REP	6000	

GROUPING SETS

GROUPING SETS is another extension to the GROUP BY clause. It is used to specify multiple groupings of data. It gives you the functionality of having multiple GROUP BY clauses in the same SELECT statement, which is not allowed in the normal syntax.

GROUPING SETS (cont.)

If you want to see data from the EMPLOYEES table grouped by (department_id, job_id, manager_id), but also grouped by (department_id, manager_id) and also grouped by (job_id, manager_id), then you would normally have to write three different select statements with the only difference between them being the GROUP BY clauses.

GROUPING SETS (cont.)

For the database, this means retrieving the same data three different times, and that can be quite a big overhead. Imagine if your company had 3,000,000 employees. Then you are asking the database to retrieve 9 million rows instead of just 3 million rows – quite a big difference.

So GROUPING SETS are much more efficient when writing complex reports.

GROUPING SETS (cont.)

In the following statement, the rows highlighted in color are generated by the GROUPING SETS operation:

```
SELECT department_id, job_id, manager_id, SUM(salary)
FROM employees
WHERE department_id < 50
GROUP BY GROUPING SETS
((job_id, manager_id), (department_id, job_id), (department_id,
manager_id));
```

GROUPING SETS (cont.)

In the following statement, the rows highlighted in color are generated by the GROUPING SETS operation:

DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
-	MK_MAN	100	13000
-	MK_MAN	201	6000
-	AD_ASST	101	4400
10	AD_ASST	-	4400
20	MK_MAN	-	13000
20	MK_REP	-	6000
10	-	101	19000
20	-	100	13000
20	-	201	6000

GROUPING Functions

When you use ROLLUP or CUBE to create reports with subtotals, you quite often also have to be able to tell which rows in the output are actual rows returned from the database and which rows are computed subtotal rows resulting from the ROLLUP or CUBE operations.

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
-	-	23400
-	MK_MAN	13000
-	MK_REP	6000
-	AD_ASST	4400
10	-	4400
10	AD_ASST	4400
20	-	19000
20	MK_MAN	13000
20	MK_REP	6000

GROUPING Functions (cont.)

If you look at the report on the right, how will you be able to differentiate between the actual database rows and the calculated rows?

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
-	-	23400
-	MK_MAN	13000
-	MK_REP	6000
-	AD_ASST	4400
10	-	4400
10	AD_ASST	4400
20	-	19000
20	MK_MAN	13000
20	MK_REP	6000

GROUPING Functions (cont.)

You may also need to find the exact level of aggregation for a given subtotal. You often need to use subtotals in calculations such as percent-of-totals, so you need an easy way to determine which rows are the subtotals.

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
-	-	23400
-	MK_MAN	13000
-	MK_REP	6000
-	AD_ASST	4400
10	-	4400
10	AD_ASST	4400
20	-	19000
20	MK_MAN	13000
20	MK_REP	6000

GROUPING Functions (cont.)

You might also be presented with a problem when you are trying to tell the difference between a stored NULL value returned by the query and NULL values created by a ROLLUP or CUBE. How can you differentiate between the two?

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
-	-	23400
-	MK_MAN	13000
-	MK_REP	6000
-	AD_ASST	4400
10	-	4400
10	AD_ASST	4400
20	-	19000
20	MK_MAN	13000
20	MK_REP	6000

GROUPING Functions (cont.)

The GROUPING function handles these problems.

Using a single column from the query as its argument, the GROUPING function will return a 1 for an aggregated (computed) row and a 0 for a non-aggregated (returned) row.

The syntax for the GROUPING is simply GROUPING (column_name). It is used only in the SELECT clause and it takes only a single column expression as the argument.

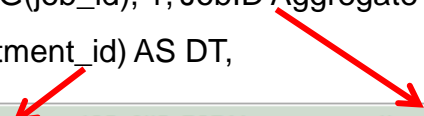
GROUPING Functions (cont.)

```
SELECT department_id, job_id, SUM(salary),  
        GROUPING(department_id) Dept_sub_total,  
        DECODE(GROUPING(department_id),  
                1, 'Dept Aggregate row',  
department_id) AS DT,  
        GROUPING(job_id) job_sub_total,  
        DECODE(GROUPING(job_id),  
                1, 'JobID Aggregate  
row', job_id) AS JI  
FROM    employees  
WHERE department_id < 50  
GROUP BY CUBE (department_id, job_id);
```

GROUPING Functions (cont.)

The output below is from the query on the previous slide, and it shows the use of the GROUPING function along with a DECODE to translate the 0 and 1 returned by GROUPING into either a text string or the actual table data:

DECODE(GROUPING(job_id), 1, 'JobID Aggregate row', job_id) AS JI
 DECODE(GROUPING(department_id), 1, 'Dept Aggregate row', department_id) AS DT,



DEPARTMENT_ID	JOB_ID	SUM(SALARY)	DEPT_SUB_TOTAL	DT	JOB_SUB_TOTAL	JI
-	-	23400	1	Dept Aggregate row	1	JobID Aggregate row
-	MK_MAN	13000	1	Dept Aggregate row	0	MK_MAN
-	MK_REP	6000	1	Dept Aggregate row	0	MK_REP
-	AD_ASST	4400	1	Dept Aggregate row	0	AD_ASST
10	-	4400	0	10	1	JobID Aggregate row
10	AD_ASST	4400	0	10	0	AD_ASST
20	-	19000	0	20	1	JobID Aggregate row
20	MK_MAN	13000	0	20	0	MK_MAN
20	MK_REP	6000	0	20	0	MK_REP

GROUPING Functions (cont.)

Quite often queries like these are used as a basis for reporting these numbers in graphical reporting tools, and all sorts of charts are created from the data to present the results in an easy to understand format to the end user.

To save the end-user from having to do the summaries and calculations manually, make the database do the work for you. But accurately interpreting the data is just as important as generating it.

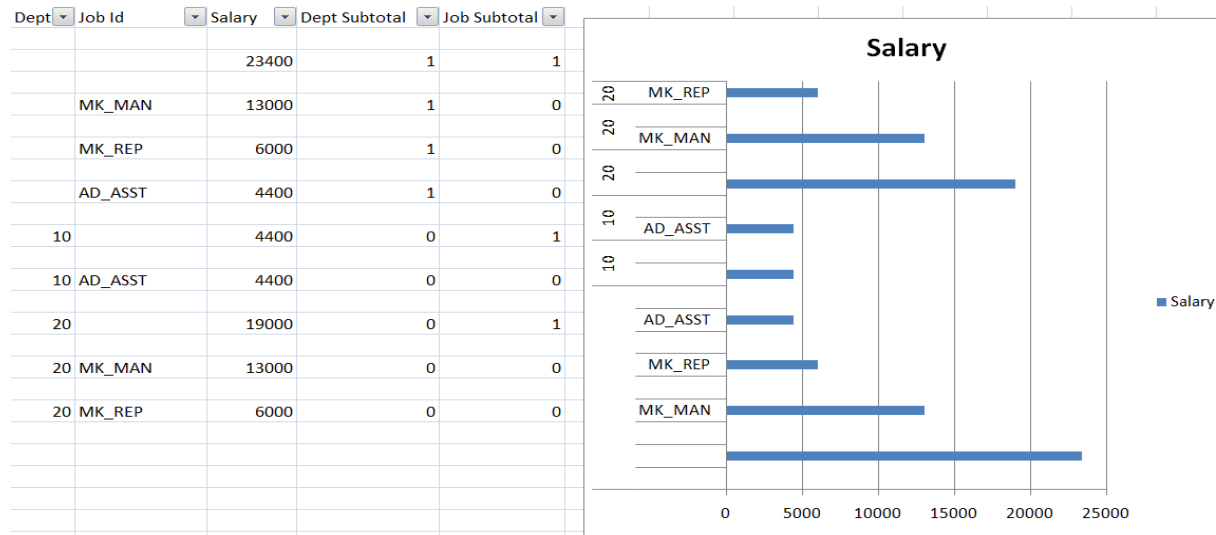
GROUPING Functions (cont.)

For example, the total `sum(salary)` for employees working in `department_id < 50` is only 23400, not 93600, which is the total of all the rows returned by the query. So you need to distinguish the real rows from the aggregated rows, and the 0 and 1 make that job really easy.

DEPARTMENT_ID	JOB_ID	SUM(SALARY)	DEPT_SUB_TOTAL	DT	JOB_SUB_TOTAL	Jl
-	-	23400	1	Dept Aggregate row	1	JobID Aggregate row
-	MK_MAN	13000	1	Dept Aggregate row	0	MK_MAN
-	MK_REP	6000	1	Dept Aggregate row	0	MK_REP
-	AD_ASST	4400	1	Dept Aggregate row	0	AD_ASST
10	-	4400	0	10	1	JobID Aggregate row
10	AD_ASST	4400	0	10	0	AD_ASST
20	-	19000	0	20	1	JobID Aggregate row
20	MK_MAN	13000	0	20	0	MK_MAN
20	MK_REP	6000	0	20	0	MK_REP

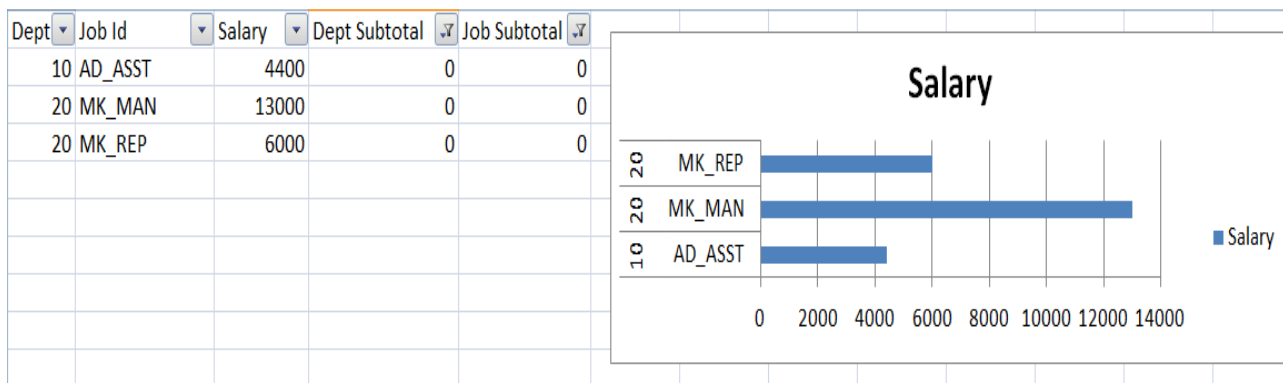
GROUPING Functions (cont.)

If the result of the query on the previous slide were to be imported into a spreadsheet and graphed without any filters applied, it would result in the following graph, which is inaccurate because it includes the same numbers multiple times.



GROUPING Functions (cont.)

Here is the same query displayed with filters applied to only include rows with a 0 in the `GROUPING(department_id)` or `GROUPING(job_id)` columns. In this report, no number is included more than once. With the 0 and 1 from the `GROUPING` function present, it is very easy to filter out the subtotals.



Terminology

Key terms used in this lesson included:

- CUBE
- GROUPING FUNCTION
- GROUPING SETS
- ROLLUP

Summary

In this lesson, you should have learned how to:

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