ALL ABOUT GROUPING

Rollups, Cubes, Grouping Sets and their inner workings

Rob van Wijk
CIBER Nederland
rob.van.wijk@ciber.nl

December, 2009
Version 1.0

Whenever you need to aggregate your data to give your users useful summary information, you need aggregate functions and SQL's group by clause. It is part of every basic SQL course, and as a database administrator, data warehouse developer or application developer, you're probably very familiar with this clause already. Oracle extended the group by clause in version 8 with rollups and cubes, and in version 9 with grouping sets, giving lots of extra possibilities. This article explores these extra possibilities, explaining how they work, both in terms of functionality, as well as in terms of how they are executed internally.

1 GROUP BY CLAUSE

The purpose of the group by clause is to group an incoming set of rows to one row per distinct value inside the set of expressions in the group by clause. In a basic SELECT … FROM … WHERE … GROUP BY query, the incoming set is the set that is formed after joining the tables and applying the predicates specified in the where clause. If the incoming set contains N rows (N ≥ 1), then the result set always contains M rows, where M is an integer between and including 1 and N. An example using the well known EMP table:

```
SQL> select sum(sal)
  2    from emp
  3   group by deptno
  4  /

  SUM(SAL)
----------
      9400
     10875
      8750

3 rows selected.
```

The incoming set is the table EMP, containing 14 rows (N=14). The group by clause contains the column deptno, meaning a row is created per distinct value of deptno. Column deptno contains three different values, 10, 20 and 30, so the result sets contains three rows (M=3).

At the extreme ends of 1 ≤ M ≤ N, we have M=N and M=1. If every row in the incoming set is a group on its own, then M equals N. This is the case when the group by clause contains a set of expressions that is unique. For example, table EMP's primary key is the empno column. When you group by empno, you get a result set containing 14 rows, so here M equals N:

```
SQL> select sum(sal)
  2    from emp
  3   group by empno
  4  /

  SUM(SAL)
----------
       800
      1600
      1250
      2975
      1250
      2850
      2450
      3000
      5000
      1500
      1100
       950
      3000
      1300

14 rows selected.
```

The other extreme end is when every row in the incoming set is placed in the same group, leading to a result set of only 1 row. For example, if you group by a constant value, or a column containing only one distinct value:

```
SQL> select sum(sal)
  2    from emp
  3   group by 1
  4  /

                                SUM(SAL)
----------------------------------------
                                   29025

1 row selected.
```

Or when you omit the group by clause altogether:

```
SQL> select sum(sal)
  2    from emp
  3  /

  SUM(SAL)
----------
     29025

1 row selected.
```

Which is shorthand for grouping by the empty set, which is denoted as ():

```
SQL> select sum(sal)
  2    from emp
  3   group by ()
  4  /

  SUM(SAL)
----------
     29025

1 row selected.
```

In all three cases above, M equals 1.

2 GROUPING SETS

Grouping sets were introduced in version 9, after rollups and cubes were already introduced in version 8. In Oracle's documentation, the order in which they are explained is: rollup, cube and then grouping sets. However, it is my experience that rollups and cubes are harder to understand without completely understanding the concept of grouping sets. By reversing the order here, and starting with explaining grouping sets, I hope to make the subject easier to understand.

First of all, what is a grouping set? A single grouping set is exactly like a regular group by clause that you already know. A regular group by clause contains a set of expressions, and the expressions are separated by commas. In a formula:

GROUP BY <set of expressions>
==
GROUP BY GROUPING SETS (<set of expressions>)

So when you have written a regular group by query in the past, you have been using a single grouping set already, and you could also have used the grouping sets notation. The point of the more verbose grouping sets notation is that it allows you to specify more than one grouping set, where the regular group by can only handle one grouping set. Or, in other words, with grouping sets, you can apply several regular group by clauses to the same incoming set. All grouping sets are working on the same incoming set and all individual result sets are put together with a union all operator. In a not-so-exact formula, but illustrating the point:

... GROUP BY GROUPING SETS (<set of expressions>$_1$, ... , <set of expressions>$_X$) ...
==
... GROUP BY <set of expressions>$_1$ ...
UNION ALL
...
UNION ALL
... GROUP BY < set of expressions>$_X$ ...

The rest of the query containing the grouping sets clause - marked with the triple dots before and after the grouping sets clause - is repeated in each union all part. The select clause is the only part that can be different, since not all non-aggregated expressions in the select clause necessarily appear in the grouping set. In this case, the expression in the select clause in the union all part should read as "null". An example:

```
SQL> select empno
  2       , deptno
  3       , sum(sal)
  4    from emp
  5   group by grouping sets ( (empno,deptno), deptno, () )
  6   order by deptno
  7       , empno
  8  /

     EMPNO     DEPTNO   SUM(SAL)
---------- ---------- ----------
      7782         10       2450
      7839         10       5000
      7934         10       1300
                   10       8750
```

```
      7369         20        800
      7566         20       2975
      7788         20       3000
      7876         20       1100
      7902         20       3000
                   20      10875
      7499         30       1600
      7521         30       1250
      7654         30       1250
      7698         30       2850
      7844         30       1500
      7900         30        950
                   30       9400
                           29025

18 rows selected.
```

As you can see by the output, this query contains three grouping sets: The first one selects the 14 original rows from EMP, the second one produces three rows containing department level aggregates where empno is null and deptno is not null. And the third grouping set, the empty set, produces the grand total where both empno as well as deptno are null. According to the earlier mentioned formula above, this query can be rewritten to:

```
SQL> select empno
  2       , deptno
  3       , sum(sal)
  4    from emp
  5   group by empno
  6       , deptno
  7   union all
  8  select empno
  9       , deptno
 10       , sum(sal)
 11    from emp
 12   group by deptno
 13   union all
 14  select empno
 15       , deptno
 16       , sum(sal)
 17    from emp
 18   group by ()
 19   order by deptno
 20       , empno
 21  /
select empno
       *
ERROR at line 8:
ORA-00979: not a GROUP BY expression
```

But this doesn't work, because the select clause for union all parts 2 and 3 now contains non-aggregated expressions that do not appear in the group by clause. So the equivalent query is one where the non-appearing non-aggregated expressions in the select list are replaced by null:

```
SQL> select empno
  2       , deptno
  3       , sum(sal)
  4    from emp
  5   group by empno
  6       , deptno
  7   union all
  8  select null empno
  9       , deptno
 10       , sum(sal)
 11    from emp
 12   group by deptno
 13   union all
 14  select null empno
 15       , null deptno
 16       , sum(sal)
```

```
 17    from emp
 18   group by ()
 19   order by deptno
 20       , empno
 21  /

     EMPNO     DEPTNO   SUM(SAL)
---------- ---------- ----------
      7782         10       2450
      7839         10       5000
      7934         10       1300
                   10       8750
      7369         20        800
      7566         20       2975
      7788         20       3000
      7876         20       1100
      7902         20       3000
                   20      10875
      7499         30       1600
      7521         30       1250
      7654         30       1250
      7698         30       2850
      7844         30       1500
      7900         30        950
                   30       9400
                           29025

18 rows selected.
```

You might have asked yourself, why the first grouping set is (empno,deptno), and not just (empno). The latter is certainly not wrong. It's just that by specifying deptno as well, the deptno column can appear in the select list, which leads to a nicer output.

With the grouping sets notation, you can specify very precisely which grouping sets you want your query to return. No additional functionality is needed per se.

3 ROLLUP

So a rollup doesn't provide extra functionality over grouping sets: every query with a rollup can be expressed with grouping sets as well. A rollup "just" provides a shorter alternative for a type of aggregation query that is very common in management reports. In such a report you typically report data at some aggregation level, and then provide subtotals at higher aggregation levels, probably resulting in the grand total. A rollup can be expressed in grouping sets notation like this:

$$GROUP\ BY\ ROLLUP\ (\ set_1\ ,\ ...,\ set_X\ )$$
$$\equiv$$
$$GROUP\ BY\ GROUPING\ SETS$$
$$(\ (set_1 \cup ... \cup set_X),\ (set_1 \cup ... \cup set_{X-1}),\ ...,\ set_1\ ,\ ()\ )$$

So you start out with a grouping sets containing all sets from the rollup, then the next grouping sets is one level higher, where you aggregate over the last set. This is repeated until you reach the empty grouping set, or in other words, the grand total.

From the formula you can deduce that a rollup with N sets, always leads to N+1 grouping sets. In the example below, the rollup contains 3 sets, so it leads to 4 grouping sets:

```
SQL> select deptno
  2       , job
  3       , mgr
  4       , empno
```

```
  5       , sum(sal)
  6    from emp
  7   group by rollup ( (deptno), (job,mgr), (empno) )
  8   order by deptno
  9       , job
 10       , mgr
 11       , empno
 12  /

    DEPTNO JOB              MGR       EMPNO   SUM(SAL)
---------- --------- ---------- ---------- ----------
        10 CLERK           7782       7934       1300
        10 CLERK           7782                  1300
        10 MANAGER         7839       7782       2450
        10 MANAGER         7839                  2450
        10 PRESIDENT                  7839       5000
        10 PRESIDENT                             5000
        10                                       8750
        20 ANALYST         7566       7788       3000
        20 ANALYST         7566       7902       3000
        20 ANALYST         7566                  6000
        20 CLERK           7788       7876       1100
        20 CLERK           7788                  1100
        20 CLERK           7902       7369        800
        20 CLERK           7902                   800
        20 MANAGER         7839       7566       2975
        20 MANAGER         7839                  2975
        20                                      10875
        30 CLERK           7698       7900        950
        30 CLERK           7698                   950
        30 MANAGER         7839       7698       2850
        30 MANAGER         7839                  2850
        30 SALESMAN        7698       7499       1600
        30 SALESMAN        7698       7521       1250
        30 SALESMAN        7698       7654       1250
        30 SALESMAN        7698       7844       1500
        30 SALESMAN        7698                  5600
        30                                       9400
                                                29025

28 rows selected.
```

Applying the formula above with set_1 = (deptno), set_2 = (job,mgr) and set_3 = (empno) leads to: grouping sets ((deptno,job,mgr,empno), (deptno,job,mgr), (deptno), ()). The next query is the same as the previous, only written using grouping sets notation instead of rollup. As you can see, the syntax is more verbose, but the semantics are the same.

```
SQL> select deptno
  2       , job
  3       , mgr
  4       , empno
  5       , sum(sal)
  6    from emp
  7   group by grouping sets
  8         ( (deptno,job,mgr,empno)
  9         , (deptno,job,mgr)
 10         , (deptno)
 11         , ()
 12         )
 13   order by deptno
 14       , job
 15       , mgr
 16       , empno
 17  /

    DEPTNO JOB              MGR       EMPNO   SUM(SAL)
---------- --------- ---------- ---------- ----------
        10 CLERK           7782       7934       1300
        10 CLERK           7782                  1300
        10 MANAGER         7839       7782       2450
        10 MANAGER         7839                  2450
```

```
        10 PRESIDENT                    7839       5000
        10 PRESIDENT                               5000
        10                                         8750
        20 ANALYST         7566         7788       3000
        20 ANALYST         7566         7902       3000
        20 ANALYST         7566                    6000
        20 CLERK           7788         7876       1100
        20 CLERK           7788                    1100
        20 CLERK           7902         7369        800
        20 CLERK           7902                    800
        20 MANAGER         7839         7566       2975
        20 MANAGER         7839                    2975
        20                                        10875
        30 CLERK           7698         7900        950
        30 CLERK           7698                    950
        30 MANAGER         7839         7698       2850
        30 MANAGER         7839                    2850
        30 SALESMAN        7698         7499       1600
        30 SALESMAN        7698         7521       1250
        30 SALESMAN        7698         7654       1250
        30 SALESMAN        7698         7844       1500
        30 SALESMAN        7698                    5600
        30                                         9400
                                                  29025

28 rows selected.
```

4 CUBE

A cube doesn't provide extra functionality over grouping sets either: also every query with a cube can be expressed with grouping sets. A cube provides a shorter alternative for queries typically used in online analytical processing. If you want to provide the user with aggregated results along every possible dimension, for example in a materialized view, then a cube comes in handy. A cube is hard to express in grouping sets notation in generic terms, but hopefully this one works:

$$GROUP\ BY\ CUBE\ (\ set_1, \ldots, set_X\ )$$
$$\equiv$$
$$GROUP\ BY\ GROUPING\ SETS$$
$$(\ all\ possible\ combinations\ between\ (\ set_1 \cup \ldots \cup set_X\ )\ and\ ()\ )$$

An example to make it a little clearer: "group by cube (set_1, set_2)" is the same as "group by grouping sets (($set_1 \cup set_2$), set_1 , set_2 , ())". All possible combinations of N sets in the cube, always lead to 2^N grouping sets.

In the next example, you see a cube with 3 sets leading to 8 (= 2^3) grouping sets:

```
SQL> select deptno
  2       , job
  3       , mgr
  4       , empno
  5       , sum(sal)
  6    from emp
  7   group by cube ( (deptno), (job,mgr), (empno) )
  8   order by deptno
  9       , job
 10       , mgr
 11       , empno
 12  /

    DEPTNO JOB              MGR      EMPNO   SUM(SAL)
---------- --------- ---------- ---------- ----------
        10 CLERK            7782       7934       1300
```

```
        10 CLERK           7782                    1300
           ...<rows 3-76 deleted>...
                                        7934       1300
                                                  29025

78 rows selected.
```

Transforming the cube ((deptno), (job,mgr), (empno)) to grouping sets notation using the "all possible combinations" formula leads to grouping sets ((deptno,job,mgr,empno), (deptno,job,mgr), (deptno,empno), (job,mgr,empno), (deptno), (job,mgr), (empno), ()). So the next query is semantically the same as the previous one:

```
SQL> select deptno
  2       , job
  3       , mgr
  4       , empno
  5       , sum(sal)
  6    from emp
  7   group by grouping sets
  8         ( (deptno,job,mgr,empno)
  9         , (deptno,job,mgr)
 10         , (deptno,empno)
 11         , (job,mgr,empno)
 12         , (deptno)
 13         , (job,mgr)
 14         , (empno)
 15         , ()
 16         )
 17   order by deptno
 18       , job
 19       , mgr
 20       , empno
 21  /

    DEPTNO JOB              MGR       EMPNO   SUM(SAL)
---------- --------- ---------- ---------- ----------
        10 CLERK           7782       7934       1300
        10 CLERK           7782                  1300
           ...<rows 3-76 deleted>...
                                      7934       1300
                                                29025

78 rows selected.
```

5 CALCULATING WITH GROUPING SETS

In a group by clause, you can combine a regular group by clause with one or more of the extensions. For example: "group by deptno, rollup(empno), cube(job,mgr), grouping sets (ename)" is perfectly valid syntax. But what does it mean? And how many grouping sets does this yield?

It's nothing frightening, really. We know by now that everything can be rewritten to grouping sets notation. So an equivalent group by clause, is this one: "group by grouping sets(deptno), grouping sets(empno,()), grouping sets((job,mgr),job,mgr,()), grouping sets(ename)". Now everything is expressed in the same way. And then we can calculate. We do that by applying a Cartesian product. Every grouping set in one part is unioned with every grouping set in the other part. In a formula:

$$GROUP\ BY\ GROUPING\ SETS\ (\ set_1\ ,\ ...,\ set_X\ )\ ,\ GROUPING\ SETS\ (\ set_{X+1}\ ,\ ...,\ set_{X+Y}\ ) \equiv GROUP\ BY\ GROUPING\ SETS$$

$$(set_1 \cup set_{X+1}, \ldots, set_1 \cup set_{X+Y}$$
$$, \ldots$$
$$, set_X \cup set_{X+1}, \ldots, set_X \cup set_{X+Y})$$

In the first mentioned example in this paragraph, this formula can be applied three times. First let's apply it to the first and second grouping sets occurrences:

```
GROUP BY GROUPING SETS (deptno)
, GROUPING SETS (empno,())
, GROUPING SETS ((job,mgr),job,mgr,())
, GROUPING SETS (ename)
≡
GROUP BY GROUPING SETS ((deptno,empno),deptno)
, GROUPING SETS ((job,mgr),job,mgr,())
, GROUPING SETS (ename)
```

Note that (deptno,()) equals (deptno). Then apply the formula again on the new grouping sets occurrence and the original third one:

```
GROUP BY GROUPING SETS ((deptno,empno),deptno)
, GROUPING SETS ((job,mgr),job,mgr,())
, GROUPING SETS (ename)
≡
GROUP BY GROUPING SETS
((deptno,empno,job,mgr),(deptno,empno,job),(deptno,empno,mgr),(deptno,empno)
,(deptno,job,mgr),(deptno,job),(deptno,mgr),deptno)
, GROUPING SETS (ename)
```

And apply it a last time:

```
GROUP BY GROUPING SETS
((deptno,empno,job,mgr),(deptno,empno,job),(deptno,empno,mgr),(deptno,empno)
,(deptno,job,mgr),(deptno,job),(deptno,mgr),deptno)
, GROUPING SETS (ename)
≡
GROUP BY GROUPING SETS ((deptno,empno,job,mgr,ename),(deptno,empno,job,ename)
,(deptno,empno,mgr,ename),(deptno,empno,ename)
,(deptno,job,mgr,ename),(deptno,job,ename),(deptno,mgr,ename),(deptno,ename))
```

And the end result is 8 grouping sets (1 * 2 * 4 * 1). You may also want to calculate the other way round. In that case you'll want to factor the common elements out of each grouping set. Each grouping set contains the columns deptno and ename. So you can rewrite this to:

```
GROUP BY deptno
, ename
, GROUPING SETS ((empno,job,mgr),(empno,job)
,(empno,mgr),empno,( job,mgr),job,mgr,())
```

And you may recognize the last one as a cube. So a final rewrite leads to:

```
GROUP BY deptno, ename, CUBE(empno,job,mgr)
```

I'm not saying the last one is the best. I know some people prefer to use grouping sets notation all the time. You have to agree that the last one is quite compact though.

Another general rule you might have spotted from above is the next one, where the rollup contains just one set:

$$GROUP\ BY\ ROLLUP\ (set_1),\ CUBE\ (set_2,\ \ldots,\ set_X)$$
$$\equiv$$
$$GROUP\ BY\ GROUPING\ SETS\ (set_1, ())$$
$$,\ GROUPING\ SETS\ (all\ possible\ combinations\ between\ (set_2 \cup \ldots \cup set_X)\ and\ ())$$
$$\equiv$$
$$GROUP\ BY\ GROUPING\ SETS\ (all\ possible\ combinations\ between\ (set_1 \cup \ldots \cup set_X)\ and\ (set_1))$$
$$,GROUPING\ SETS\ (all\ possible\ combinations\ between\ (set_2 \cup \ldots \cup set_X)\ and\ ())$$
$$\equiv$$
$$GROUP\ BY\ GROUPING\ SETS\ (all\ possible\ combinations\ between\ (set_1 \cup \ldots \cup set_X)\ and\ ())$$
$$\equiv$$
$$GROUP\ BY\ CUBE\ (set_1,\ \ldots,\ set_X)$$

6 SUPPORTING FUNCTIONS

Oracle has three supporting functions for the group by clause: GROUPING, GROUPING_ID and GROUP_ID. That sounds similar enough to have to look each one up each time you need them. The supporting functions all return integers to help you determine inside which grouping set you are. Since a column value appears as null when aggregating over the column, you might think you can use IS NULL or IS NOT NULL. This doesn't work however, when the column itself can contain null values. The supporting functions help you to determine whether a null value is from the column itself or because it was aggregated over.

The GROUPING function takes one expression from the group by clause as an argument and returns a 1 if the value is null because it represents the set of all values. It returns 0 if it represents a regular row. An example:

```
SQL> select mgr
  2       , sum(sal)
  3       , grouping(mgr)
  4    from emp
  5   group by rollup(mgr)
  6  /

       MGR   SUM(SAL) GROUPING(MGR)
---------- ---------- -------------
      7566       6000             0
      7698       6550             0
      7782       1300             0
      7788       1100             0
      7839       8275             0
      7902        800             0
                 5000             0
                29025             1

8 rows selected.
```

The last two rows in this example have their mgr value set to null, but only the last row, with sum(sal) 29025, represents the set of all values.

If you have a large set of expressions in your group by clause, then you could get a large Boolean expression to determine the grouping set you are in, something like "grouping($expr_1$) = 0 and grouping($expr_2$) = 1 and grouping($expr_3$) = 0 and grouping($expr_4$) = 1". This can become cumbersome and that's when the GROUPING_ID function comes in handy. The function

accepts one or more expressions as its arguments. It applies the GROUPING function to each of its arguments and turns them into a string of zeros and ones and returns that as a number. So the above expression can also be written as "grouping_id($expr_1$,$expr_2$,$expr_3$,$expr_4$) = 5", where 5 is "0101".

And then there is the rarely used GROUP_ID function. This one is only useful when you specified duplicate grouping sets. If you don't specify duplicate grouping sets, the GROUP_ID function always returns 0. For the first duplicate, it returns a 1, for the next a 2, and so on. An example:

```
SQL> select deptno
  2       , sum(sal)
  3       , group_id()
  4    from emp
  5   group by grouping sets ( deptno, deptno, (), (), () )
  6   order by deptno
  7  /

    DEPTNO   SUM(SAL) GROUP_ID()
---------- ---------- ----------
        10       8750          1
        10       8750          0
        20      10875          0
        20      10875          1
        30       9400          1
        30       9400          0
                29025          1
                29025          0
                29025          2

9 rows selected.
```

7 HOW DOES ORACLE IMPLEMENT THE GROUP BY EXTENSIONS?

Up until Oracle 10g release 2, most grouping operations were performed by the SORT GROUP BY operation. There are several other operations that can be chosen by the optimizer as well, but they are far less common. A good overview of all operations can be found on the website of Julian Dyke[1]. Oracle RDBMS version 10.2.0.1 introduced the HASH GROUP BY operation, which is a more efficient way to perform groupings in most cases. Oracle doesn't document how exactly these operations are performed. However, by their name we can rather safely assume what each operation does. A SORT GROUP BY will sort the incoming set, traverse the ordered set and compare each row with its predecessor; if it encounters the same value, then the row is placed in the same group and if a different value is encountered, then a new group is created. A HASH GROUP BY will build an in-memory hash table to hold the group rows. Each row from the incoming set will be hashed and placed into the hash table. When the hash table doesn't fit into memory, it will spill to disk. This is not only true for a HASH GROUP BY: both HASH GROUP BY and SORT GROUP BY need a SQL workarea in the PGA-memory of the session. When the workarea becomes too small, the operation is divided into smaller pieces. Some pieces are processed in memory, while the remainder is written to temporary disk storage for later processing. If a query does not contain an order by clause, I have not been able to let the optimizer come up with a plan using the SORT GROUP BY operation in a version >= 10.2.0.1.

Now, let's have a look at the operations needed for rollups, cubes and grouping sets. Please note that all queries below were executed in version 11.1.0.7, and checked in 10.2.0.1 and 11.2.0.1 to be the same. First, let's have a look at how a rollup is calculated:

```
SQL> select /*+ gather_plan_statistics */
  2         deptno
  3       , empno
```

```
  4       , sum(sal)
  5    from emp
  6   group by rollup(deptno,empno)
  7  /

    DEPTNO      EMPNO   SUM(SAL)
---------- ---------- ----------
        10       7782       2450
    ... <rows 2-17 deleted> ...
                           29025

18 rows selected.

SQL> select * from table(dbms_xplan.display_cursor(null,null,'iostats last'))
  2  /

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------
SQL_ID  2d0yuq4g2z951, child number 0
-------------------------------------
select /*+ gather_plan_statistics */        deptno      , empno      ,
sum(sal)   from emp  group by rollup(deptno,empno)

Plan hash value: 52302870

-----------------------------------------------------------------------------------------
| Id  | Operation            | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |      1 |        |     18 |00:00:00.01 |       7 |
|   1 |  SORT GROUP BY ROLLUP|      |      1 |     14 |     18 |00:00:00.01 |       7 |
|   2 |   TABLE ACCESS FULL  | EMP  |      1 |     14 |     14 |00:00:00.01 |       7 |
-----------------------------------------------------------------------------------------


15 rows selected.
```

A rollup always performs either a SORT GROUP BY ROLLUP or a SORT GROUP BY ROLLUP NOSORT operation. Possibly, this is worded too strong, but I have not witnessed other operations so far.

The SORT GROUP BY ROLLUP operation uses the SORT GROUP BY operation for the lowest level grouping set: the one containing all expressions of the rollup. The output rows for each subsequent grouping set can always be calculated using the output rows of the previous grouping set as their incoming set. And this previous set of output rows is already sorted as well. Therefore, a SORT GROUP BY ROLLUP can be thought of as recursively implementing the SORT GROUP BY operation, sorting only at the first recursion. In the above example the entire EMP table is read using a full table scan. These 14 rows are the incoming set for processing the first grouping set (deptno,empno). The SORT GROUP BY operation is applied, leading to 14 group rows, sorted. After that, grouping set (deptno) gets processed by using the 14 output rows of grouping set (deptno,empno) as its incoming set. Again, a SORT GROUP BY is performed, without actually having to sort. So it really is a SORT GROUP BY NOSORT operation leading to 3 rows. And finally the grandtotal, grouping set (), is calculated by using the 3 output rows of grouping set (deptno) and again performing a SORT GROUP BY NOSORT.

If the incoming set is already ordered, for example because the rows are retrieved by an INDEX FULL SCAN, then you'll see a SORT GROUP BY NOSORT ROLLUP. This one behaves exactly the same as a SORT GROUP BY ROLLUP, but it doesn't use a SORT GROUP BY for the first grouping set, but a SORT GROUP BY NOSORT. You can see that happening when you create an index on (deptno,empno) and run the same query as above. The plan then looks like this:

```
-----------------------------------------------------------------------------------------
| Id  | Operation                   | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |      |      1 |        |     18 |00:00:00.01 |       4 |
|   1 |  SORT GROUP BY NOSORT ROLLUP|      |      1 |     14 |     18 |00:00:00.01 |       4 |
```

```
|   2 |   TABLE ACCESS BY INDEX ROWID| EMP  |      1 |     14 |     14 |00:00:00.01 |       4 |
|   3 |    INDEX FULL SCAN           | I1   |      1 |     14 |     14 |00:00:00.01 |       2 |
-------------------------------------------------------------------------------------------------
```

The number of performed sort operations was checked by performing this query right before and after the queries above:

```
SQL> select sn.name
  2       , ms.value
  3    from v$mystat   ms
  4       , v$statname sn
  5   where ms.statistic# = sn.statistic#
  6     and sn.name like '%sort%'
  7  /
```

The SORT GROUP BY ROLLUP performs 1 memory sort and sorts 14 rows. The SORT GROUP BY ROLLUP NOSORT performs 0 sorts.

A cube is always calculated by performing three operations: a SORT GROUP BY, GENERATE CUBE and another SORT GROUP BY. Again, this may be worded too strong, but I have not seen other plans yet. An example of the calculation of a cube:

```
SQL> select /*+ gather_plan_statistics */
  2         deptno
  3       , job
  4       , sum(sal)
  5    from emp
  6   group by cube(deptno,job)
  7  /

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
                          29025
   ... <rows 2-17 deleted> ...
        30 SALESMAN         5600

18 rows selected.

SQL> select * from table(dbms_xplan.display_cursor(null,null,'iostats last'))
  2  /

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
SQL_ID  d706ztmnyxft3, child number 0
-------------------------------------
select /*+ gather_plan_statistics */        deptno      , job      ,
sum(sal)   from emp  group by cube(deptno,job)

Plan hash value: 3627207636

-------------------------------------------------------------------------------------------
| Id  | Operation            | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |      1 |        |     18 |00:00:00.01 |       7 |
|   1 |  SORT GROUP BY       |      |      1 |     11 |     18 |00:00:00.01 |       7 |
|   2 |   GENERATE CUBE      |      |      1 |     11 |     36 |00:00:00.01 |       7 |
|   3 |    SORT GROUP BY     |      |      1 |     11 |      9 |00:00:00.01 |       7 |
|   4 |     TABLE ACCESS FULL| EMP  |      1 |     14 |     14 |00:00:00.01 |       7 |
-------------------------------------------------------------------------------------------


17 rows selected.
```

For a cube, it is not possible to use the same operations as used in a rollup. The difference is, there might be more than one grouping set that needs the same incoming set. For example, both grouping sets (deptno) and (job) would need the incoming set of (deptno,job). To calculate a cube, we need the incoming set duplicated as many times as there are grouping sets.

This is what the GENERATE CUBE operation does: duplicate the intermediate result sets by the number of grouping sets, which equals $2^{\#sets\ in\ cube}$. To do this as efficient as possible, the number of rows in the incoming set is reduced by first applying a SORT GROUP BY operation on the lowest level grouping set, the one containing all sets in the cube. After the GENERATE CUBE step, a SORT GROUP BY operation is performed for each of the dimensions of the cube. This one puzzled me somewhat: it clearly says that 1 SORT GROUP BY was performed, where I would have expected the number of "Starts" to be $2^{\#sets\ in\ cube}$ and the operation to be SORT GROUP BY NOSORT. The statistics in v$mystat using the same query as mentioned earlier, reveals that 2 memory sorts are performed and that 50 rows were sorted. This indicates that the first sort from step 3 sorted 14 rows and that the second sort was indeed only one SORT GROUP BY operation that sorted the 36 rows (14 + 36 = 50). So the last SORT GROUP BY operation is able to compute all grouping sets in one pass. This must mean that the GENERATE CUBE does some more than just duplicating the intermediate set; it nullifies the group by columns according to the cube. Because of this nullifying, a single SORT GROUP BY operation afterwards is able to produce the result for all grouping sets.

In the example you can see the story above when looking at the "A-rows" column. The first step is a full table scan of EMP, leading to 14 rows. Those 14 rows are then grouped by (deptno,job). There are 9 different combinations of deptno and job, so this leads to 9 rows. The GENERATE CUBE operation expands this set 4 times to four sets of 9 rows, a total of 36 rows. It does so by transforming this set:

DEPTNO	JOB	SUM(SAL)
20	CLERK	1900
30	SALESMAN	5600
20	MANAGER	2975
30	CLERK	950
10	PRESIDENT	5000
30	MANAGER	2850
10	CLERK	1300
10	MANAGER	2450
20	ANALYST	6000

to this set:

DEPTNO	JOB	SUM(SAL)
20	CLERK	1900
30	SALESMAN	5600
20	MANAGER	2975
30	CLERK	950
10	PRESIDENT	5000
30	MANAGER	2850
10	CLERK	1300
10	MANAGER	2450
20	ANALYST	6000
20		1900
30		5600
20		2975
30		950
10		5000
30		2850
10		1300
10		2450
20		6000
	CLERK	1900
	SALESMAN	5600
	MANAGER	2975
	CLERK	950
	PRESIDENT	5000
	MANAGER	2850
	CLERK	1300
	MANAGER	2450
	ANALYST	6000

```
                                1900
                                5600
                                2975
                                950
                                5000
                                2850
                                1300
                                2450
                                6000
```

The last step is applying a SORT GROUP BY operation to the 36 rows. The set is grouped by (deptno,job), leading to 18 rows, which are exactly the 18 group rows corresponding to the four grouping sets (deptno,job), (deptno), (job) and (). Expressed in SQL, this cube is calculated like this:

```
with emp_after_first_sortgroupby as
( select deptno
       , job
       , sum(sal) sumsal
    from emp
   group by deptno
       , job
)
, emp_after_generate_cube as
( select deptno,job,sumsal from emp_after_first_sortgroupby
  union all
  select deptno,null,sumsal from emp_after_first_sortgroupby
  union all
  select null,job,sumsal from emp_after_first_sortgroupby
  union all
  select null,null, sumsal from emp_after_first_sortgroupby
)
select deptno
     , job
     , sum(sumsal)
  from emp_after_generate_cube
 group by deptno
     , job
```

Both rollup and cube are special in the sense that all output rows can be obtained by scanning the incoming set only once to calculate the lowest level grouping set. And then all other grouping sets are subsets of that first grouping set. However, in general, with grouping sets, this is not the case. You can have two grouping sets (deptno) and (job), without having a grouping set (deptno,job). In such a scenario, you would need to scan the incoming set more than once. Oracle solves this problem by introducing two temporary tables, an input table and an output table. The incoming set is loaded into the input table and when computing the grouping sets, the output rows are stored into the output table. The latter is used for returning the result set. An example:

```
SQL> select /*+ gather_plan_statistics */
  2         deptno
  3       , job
  4       , mgr
  5       , sum(sal)
  6    from emp
  7   group by grouping sets(deptno,job,mgr)
  8  /

    DEPTNO JOB              MGR   SUM(SAL)
---------- --------- ---------- ----------
                           7839       8275
   ... <rows 2-14 deleted> ...
        10                            8750

15 rows selected.

SQL> select * from table(dbms_xplan.display_cursor(null,null,'iostats last'))
```

```
  2  /

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------
SQL_ID  cqn9d7t3atmbd, child number 0
-------------------------------------
select /*+ gather_plan_statistics */        deptno      , job      ,
mgr      , sum(sal)    from emp  group by grouping sets(deptno,job,mgr)

Plan hash value: 3776576756

-------------------------------------------------------------------------------------------------
| Id  | Operation                  | Name                       | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                            |      1 |        |     15 |
|   1 |  TEMP TABLE TRANSFORMATION |                            |      1 |        |     15 |
|   2 |   LOAD AS SELECT           |                            |      1 |        |      1 |
|   3 |    TABLE ACCESS FULL       | EMP                        |      1 |     14 |     14 |
|   4 |   LOAD AS SELECT           |                            |      1 |        |      1 |
|   5 |    HASH GROUP BY           |                            |      1 |      1 |      7 |
|   6 |     TABLE ACCESS FULL      | SYS_TEMP_0FD9D6603_B63617  |      1 |      1 |     14 |
|   7 |   LOAD AS SELECT           |                            |      1 |        |      1 |
|   8 |    HASH GROUP BY           |                            |      1 |      1 |      5 |
|   9 |     TABLE ACCESS FULL      | SYS_TEMP_0FD9D6603_B63617  |      1 |      1 |     14 |
|  10 |   LOAD AS SELECT           |                            |      1 |        |      1 |
|  11 |    HASH GROUP BY           |                            |      1 |      1 |      3 |
|  12 |     TABLE ACCESS FULL      | SYS_TEMP_0FD9D6603_B63617  |      1 |      1 |     14 |
|  13 |   VIEW                     |                            |      1 |      1 |     15 |
|  14 |    TABLE ACCESS FULL       | SYS_TEMP_0FD9D6604_B63617  |      1 |      1 |     15 |
-------------------------------------------------------------------------------------------------

27 rows selected.
```

Note that I removed the columns A-time, Buffers, Reads and Writes from the plan output for better readability. The plan starts with step 3, full scanning table EMP. The rows from EMP are copied into the input table SYS_TEMP_0FD9D6603_B63617 (step 2) using direct path. This table is used as a starting point for calculating the three grouping sets (deptno), (job) and (mgr) and is therefore scanned in full, three times. The result of each grouping set is copied into the output table SYS_TEMP_0FD9D6604_B63617. At first glance, the name of the output table looks exactly the same as the input table. Note however the "4_" instead of the "3_". Steps 4, 7 and 10 represent the copying of the results of the grouping sets into the output table. At the end the output table is full scanned and returned via a VIEW and TEMP TABLE TRANSFORMATION operation. The way the output table is filled differs per query, but there grouping sets queries always use temporary input and output tables.

But do grouping sets queries also use temporary input and output tables, even when the grouping sets query can be transformed to a rollup or cube? Let's start with a grouping sets query that can be transformed to a rollup:

```
SQL> select /*+ gather_plan_statistics */
  2         deptno
  3       , job
  4       , sum(sal)
  5    from emp
  6   group by grouping sets( (deptno,job), deptno, () )
  7  /

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        10                 8750
        20 CLERK           1900
        20 ANALYST         6000
        20 MANAGER         2975
        20                10875
```

```
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600
        30                 9400
                          29025

13 rows selected.

SQL> select * from table(dbms_xplan.display_cursor(null,null,'iostats last'))
  2  /

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------

SQL_ID  32utu83m0fv8n, child number 0
-------------------------------------
select /*+ gather_plan_statistics */        deptno      , job      ,
sum(sal)   from emp  group by grouping sets( (deptno,job), deptno,  () )

Plan hash value: 52302870

-----------------------------------------------------------------------------------------
| Id  | Operation            | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |      1 |        |     13 |00:00:00.01 |       7 |
|   1 |  SORT GROUP BY ROLLUP|      |      1 |     11 |     13 |00:00:00.01 |       7 |
|   2 |   TABLE ACCESS FULL  | EMP  |      1 |     14 |     14 |00:00:00.01 |       7 |
-----------------------------------------------------------------------------------------


15 rows selected.
```

This query can be rewritten using a "group by rollup (deptno, job)". The cost based optimizer is smart enough to see that the grouping sets notation is really a rollup in disguise. It has therefore rewritten the query in the optimization phase to a rollup, as you can see by the SORT GROUP BY ROLLUP operation in step 1. Now a similar test for a cube expressed in grouping sets notation:

```
SQL> select /*+ gather_plan_statistics */
  2         deptno
  3       , job
  4       , sum(sal)
  5    from emp
  6   group by grouping sets( (deptno,job), deptno, job, () )
  7  /

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        10 CLERK           1300
   ...<rows 2-17 deleted>...
                          29025

18 rows selected.

SQL> select * from table(dbms_xplan.display_cursor(null,null,'iostats last'))
  2  /

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
SQL_ID  9wptn033f092m, child number 0
-------------------------------------
select /*+ gather_plan_statistics */        deptno      , job      ,
sum(sal)   from emp  group by grouping sets( (deptno,job), deptno, job,
() )

Plan hash value: 1142402200

-----------------------------------------------------------------------------------------
| Id  | Operation                  | Name                       | Starts | E-Rows | A-Rows |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                            |      1 |        |     18 |
```

```
|   1 |  TEMP TABLE TRANSFORMATION |                            |      1 |        |     18 |
|   2 |   MULTI-TABLE INSERT       |                            |      1 |        |      2 |
|   3 |    SORT GROUP BY ROLLUP    |                            |      1 |     11 |     14 |
|   4 |     TABLE ACCESS FULL      | EMP                        |      1 |     14 |     14 |
|   5 |   LOAD AS SELECT           |                            |      1 |        |      1 |
|   6 |    SORT GROUP BY ROLLUP    |                            |      1 |      1 |      4 |
|   7 |     TABLE ACCESS FULL      | SYS_TEMP_0FD9D6605_B69701  |      1 |      1 |      9 |
|   8 |   VIEW                     |                            |      1 |      2 |     18 |
|   9 |    VIEW                    |                            |      1 |      2 |     18 |
|  10 |     UNION-ALL              |                            |      1 |        |     18 |
|  11 |      TABLE ACCESS FULL     | SYS_TEMP_0FD9D6605_B69701  |      1 |      1 |      9 |
|  12 |      TABLE ACCESS FULL     | SYS_TEMP_0FD9D6606_B69701  |      1 |      1 |      9 |
-------------------------------------------------------------------------------------------

26 rows selected.
```

Note again that I removed the columns A-time, Buffers, Reads and Writes from the plan output for better readability. Here, we humans immediately see that the group by clause equals "group by cube(deptno,job)", but the cost based optimizer does not. We do see an interesting variation of how grouping sets can be implemented. The temporary input and output table are still present, but the way they are populated differs. The plan starts with a full table scan of the EMP table. The 14 rows go through a SORT GROUP BY ROLLUP operation which leads to 14 rows. This must mean that the SORT GROUP BY ROLLUP has computed the grouping sets (deptno,job) and (job). Those grouping sets result in 9 and 5 group rows, which equal 14 rows in total. The MULTI-TABLE INSERT at step 2 inserts those 14 rows into two tables. The 9 rows from grouping set (deptno,job) are inserted into the input table SYS_TEMP_0FD9D6605_B69701, and the 5 rows from grouping set (job) are inserted into the output table SYS_TEMP_pFD9D6606_B69701. Then at step 7, the input table containing 9 rows from grouping set (deptno,job) is read and is used for a SORT GROUP BY ROLLUP of grouping sets (deptno) and (). The 4 resulting rows are also inserted into the output table at step 5. In step 10, the contents of both the input and the output table are union-all'd together to construct the final result set of 18 rows.

For small queries, meaning the ones that should execute in less than a few seconds, the overhead of setting up temporary input and output tables and populating them, might be too much performance wise. Whether you need grouping sets notation depends on your functional requirements. However, if you can spot a cube in your grouping sets, you'll want to rewrite it manually to a cube.

8 REFERENCES

1) http://www.juliandyke.com/Optimisation/Operations/Operations.html