2013

SQL FOR MODELING BY EXAMPLE

Document describes very powerful but very rare used SQL syntax – MODEL.

Environment description

- OS - Oracle Linux Server release 6.3 x64
- Database – Oracle Database 11.2.0.3 EE with sample schemas

Article details

In this article, I will describe the most powerful SQL syntax in Oracle Database. SQL For Modeling was first introduced in Oracle 10g but for some reason it is very unpopular. I've been an Oracle trainer for 8 years now and this topic is covered by ca 1% official trainings that I've seen. And this high score is often a result of my interference in course agenda. This is very odd, because – as you will see – this syntax can be very useful.

Quoting after Oracle Documentation:

> "The MODEL clause brings a new level of power and flexibility to SQL calculations. With the MODEL clause, you can create a multidimensional array from query results and then apply formulas (called rules) to this array to calculate new values. The rules can range from basic arithmetic to simultaneous equations using recursion. For some applications, the MODEL clause can replace PC-based spreadsheets. Models in SQL leverage Oracle Database's strengths in scalability, manageability, collaboration, and security. The core query engine can work with unlimited quantities of data. By defining and executing models within the database, users avoid transferring large data sets to and from separate modeling environments. Models can be shared easily across workgroups, ensuring that calculations are consistent for all applications. Just as models can be shared, access can also be controlled precisely with Oracle's security features. With its rich functionality, the MODEL clause can enhance all types of applications."

If you would like to read more about theory, please refer to documentation:

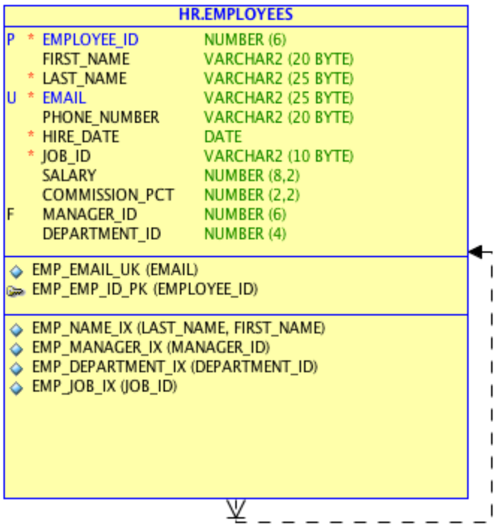
http://docs.oracle.com/cd/B28359_01/server.111/b28313/sqlmodel.htm

In this article I will focus on examples, assuming that you have strong SQL knowledge – including window functions. And at least basic knowledge of programming in any structural language.

Ok, enough of talking – let's see some action!

CASE 1 – Finding salary in EMPLOYEES tree

The HR.EMPLOYEES table represents people, which were hired in some company.



Let's assume that I want to create a report, that will show me: FIRST_NAME, LAST_NAME, EMPLOYEE_ID, MANAGER_ID, SALARY and ***THE_LOWEST_SALARY_OF_EMPLOYEE_WHO_WORKS_DIRECTLY_FOR_A_MANAGAER***.

```sql
select employee_id, manager_id, first_name,
       last_name, salary
from employees
```

Query Result — All Rows Fetched: 107 in 0,018 seconds

	EMPLOYEE_ID	MANAGER_ID	FIRST_NAME	LAST_NAME	SALARY
1	100	(null)	Steven	King	24000
2	101	100	Neena	Kochhar	17000
3	102	100	Lex	De Haan	17000
4	103	102	Alexander	Hunold	9000
5	104	103	Bruce	Ernst	6000
6	105	103	David	Austin	4800
7	106	103	Valli	Pataballa	4800
8	107	103	Diana	Lorentz	4200
9	108	101	Nancy	Greenberg	12008
10	109	108	Daniel	Faviet	9000
11	110	108	John	Chen	8200
12	111	108	Ismael	Sciarra	7700
13	112	108	Jose Manuel	Urman	7800
14	113	108	Luis	Popp	6900
15	114	100	Den	Raphaely	11000
16	115	114	Alexander	Khoo	3100
17	116	114	Shelli	Baida	2900
18	117	114	Sigal	Tobias	2800
19	118	114	Guy	Himuro	2600
20	119	114	Karen	Colmenares	2500
21	120	100	Matthew	Weiss	8000
22	121	100	Adam	Fripp	8200
23	122	100	Payam	Kaufling	7900
24	123	100	Shanta	Vollman	6500
25	124	100	Kevin	Mourgos	5800

For example – for Alexander Hunold (employee_id=103), the lowest salary from all employees that are working directly for him is 4200. There are a lot of potential solutions for this problem, for example:

```
select employee_id, manager_id, first_name,
       last_name, salary,
       (select min(salary)
        from employees e2
        where e2.manager_id=e1.employee_id) as min_sal
from employees e1;
```

OR

```
select e.employee_id, e.manager_id, e.first_name,
       e.last_name, e.salary,
       min(m.salary) min_sal
from employees e, employees m
where e.employee_id=m.manager_id(+)
group by e.employee_id, e.manager_id, e.first_name,
       e.last_name, e.salary
```

Both queries are ok, but both have the same issue – accessing the same table twice! Such queries are ineffective for large scaling sets of data and can produce excessive IO operations on TEMP tablespace (*direct path read temp* and *direct path write temp* for multipass operations).

Before writing SQL MODEL syntax, let's look on this set of data in multidimensional view ☺

This is SALARY array, indexed by EMPLOYEE_ID

employee_id	salary
100	24000
101	17000
102	17000
103	9000
104	6000
105	4800

I could get a value of SALARY for EMPLOYEE_ID=100 with this syntax: salary[100]

(NOTE: If SALARY would be unique, this could be also EMPLOYEE_ID array, indexed by SALARY – do not get trapped in a trap of stereotype thinking)

I could get minimal value for SALARY with this syntax: min(salary)[any]

It can be a little bit confusing, because more natural syntax would be: min(salary[any]) but don't be worry – you'll get used to it ☺

To solve my example, I need another value for indexing my array – MANAGER_ID.

(NOTE: In model syntax, array is called MEASURE and INDEX is called DIMENSION – just like in CUBE. I will be using those names in further part of my article)

employee_id	Manager_id	salary
100		24000
101	100	17000
102	100	17000
103	102	9000
104	103	6000
105	103	4800

OK. Now I have two dimensions and one measure. So the third row could be signed like this: salary[102,100]. Of course I don't need the MANAGER_ID dimension for gaining uniqueness in my array – I need it to find some specific values – in this case, the minimum value of SALARY for every direct employee of each manager (for people who are not managers, the value will be NULL).

- For the first row: *min(salary)[any,100]*
- For the second row: *min(salary)[any,101]*
- For the third row: *min(salary)[any,102]*
- OK, I think You already know what I mean ☺…

So, I could say, that loop spins by EMPLOYEE_ID and I use the counter of the loop, to get appropriate values. I'm using the EMPLOYEE_ID dimension on the position of MANAGER_ID (this is almost a JOIN☺).

In general I could write it like this: *min(salary)[any,cv(employee_id)],* where CV stands for: "Current Value".

employee_id	Manager_id	salary	Min_sal
100		24000	*min(salary)[any,cv(employee_id)]*
101	100	17000	*min(salary)[any,cv(employee_id)]*
102	100	17000	*min(salary)[any,cv(employee_id)]*
103	102	9000	*min(salary)[any,cv(employee_id)]*
104	103	6000	*min(salary)[any,cv(employee_id)]*
105	103	4800	*min(salary)[any,cv(employee_id)]*

Now I have two dimensions (EMPLOYEE_ID and MANAGER_ID) and two measures (SALARY and newly defined MIN_SAL). Let's write our SQL based on above information ☺

```
select employee_id, manager_id, salary, min_sal
from employees
model
dimension by (employee_id, manager_id)
measures (salary, 0 as min_sal)
rules
(
  min_sal[any,any]=min(salary)[any,cv(employee_id)]
);
```

OK, maybe the syntax is not the most intuitive in the world, but if you look closer – it's logical and easy. After the MODEL keyword, we defined dimensions and measures – because in EMPLOYEES table there is no MIN_SAL column, I'm creating it by defining new measure allocated with 0.

Very important thing is, that after SELECT keyword you don't specify table columns but measures or dimensions, used in MODEL. So if you try to use the FIRST_NAME column, you would get error like below:

ORA-32614: illegal MODEL SELECT expression

So, if we want to display additional columns in our query, we have to use them somewhere in model syntax – the easiest way is to put them as measures and never use them in RULES section. The final query:

```
select first_name, last_name, salary, min_sal
from employees
model
dimension by (employee_id, manager_id)
measures (salary, 0 as min_sal, first_name, last_name)
rules
(
  min_sal[any,any]=min(salary)[any,cv(employee_id)]
);
```

Notice, that I don't have to display every dimension or measure that I used in model.

CASE 2 – Finding people employed in the year with greatest number of hirings.

We could write this example, for example like this:

```
with v_emps_year as
(
  select first_name, last_name, to_char(hire_date,'YYYY') as h_year,
         count(1) over (partition by to_char(hire_date,'YYYY')) as cnt
  from employees
), v_emps_rank as
  (
    select y.*, dense_rank() over (order by cnt desc) as rnk
    from v_emps_year y
  )
select *
from v_emps_rank
where rnk=1;
```

Now let's resolve this problem with modeling – I can see here two measures (**CNT** for counting the number of employees hired in specific year and **RNK** for dense_rank) and two dimensions (**EMPLOYEE_ID** will provide uniqueness in my array and **H_YEAR** will provide me desired information about current year value, which I need to make calculations).

employee_id	to_char(hire_date,'YYYY') as H_YEAR	0 as CNT	0 as RNK
100	2003	*count(cnt)[any,cv()]*	*dense_rank() over (order by cnt desc)*
101	2005	*count(cnt)[any,cv()]*	*dense_rank() over (order by cnt desc)*
102	2001	*count(cnt)[any,cv()]*	*dense_rank() over (order by cnt desc)*
103	2006	*count(cnt)[any,cv()]*	*dense_rank() over (order by cnt desc)*
104	2007	*count(cnt)[any,cv()]*	*dense_rank() over (order by cnt desc)*
105	2005	*count(cnt)[any,cv()]*	*dense_rank() over (order by cnt desc)*

(NOTE: My measures are allocated at the beginning with value "0" – that's why all rows in my set, will have this value, before rules will apply. And that's why I can count my CNT measure – I count occurrence of values "0". In Above example I used syntax "*CV()*" – earlier I used this function with argument name – here I used it as positional)

The final query could look like this:

```
with v_emps_year as
(
 select first_name, last_name, h_year, cnt, rnk
 from employees
 model
 dimension by (employee_id, to_char(hire_date,'YYYY') as h_year)
 measures (0 as cnt, 0 as rnk, first_name, last_name)
 rules
 (
  cnt[any,any]=count(cnt)[any,cv()],
  rnk[any,any]=dense_rank() over (order by cnt desc)
 )
)
select *
from v_emps_year
where rnk=1;
```

Please notice, that I used window function in RULES syntax to find create ranking – in DENSE_RANK() I'm using CNT measure as normal column. It gives me possibility to combine analytical functions with other calculated values without using unnecessary subqueries.

CASE 3 – Aggregating rows

One of the most common problems is aggregating rows to columns – the opposite to SPLIT (which is also quite a big problem☺) Suppose we want to create a report that shows DEPARTMENT_NAME and ids of employees, hired in that department, separated with '#'. In 11g database we can use LISTAGG function to resolve this problem. The query that is using this function could look like this:

```
select department_name,
       listagg(employee_id,'#') within group (order by employee_id)
from employees e, departments d
where e.department_id=d.department_id
group by department_name;
```

In 10g database we didn't have this function, so solving this problem was much harder. Often CONNECT BY functionality was used to produce desired result. For example:

```
with v_emps as
(
  select department_name, employee_id,
         row_number() over (partition by department_name
                           order by employee_id) as rn
  from employees e, departments d
  where e.department_id=d.department_id
)
select department_name,
       sys_connect_by_path(employee_id,'#')
from v_emps
where connect_by_isleaf=1
start with rn=1
connect by rn=prior rn+1
and department_name=prior department_name;
```

Unfortunately this syntax can cause a lot of memory usage in PGA buffer.

Now, let's try to use model syntax – I would like to execute rules for each department separately – to achieve this I will use PARTITION BY syntax. My only dimension will be artificial ID based on ROW_NUMBER() function – thanks to this I will be able to access previous and next elements in my measures, which are: EMPLOYEE_ID, LEAF (for finding last value after concatenation), EMPS (the product of the concatenation) – first value of each partition is the same as the first value of EMPLOYEE_ID measure, each next element is concatenation of previous EMPS value, the '#' character and current EMPLOYEE_ID value. As the final step we will find last concatenated record of each partition by using DENSE_RANK() function. The final query looks like this:

```
with v_emp_concat as
(
select department_name, employee_id, emps, leaf
from employees e, departments d
where e.department_id=d.department_id
model
partition by (department_name)
dimension by (row_number() over (partition by department_name
                                 order by employee_id) as i)
measures (employee_id, 0 as leaf,
          cast(null as varchar2(4000)) as emps)
rules
(
  emps[any] order by i=case when cv(i)=1 then to_char(employee_id[1])
                           else emps[cv()-1] || '#' || employee_id[cv()]
                      end,
  leaf[any]=dense_rank() over (order by i desc)
)
)
select department_name, emps
from v_emp_concat
where leaf=1;
```

Notice, the "***order by i***" syntax – this is "cure" for the following error:

ORA-32637: Self cyclic rule in sequential order MODEL

CASE 4 – Split a string into rows

The next case is very popular – let's assume that we have such situation – in application we can use a checkbox to select some elements, which ids are concatenated into a string like this: "10,20,50,40" – based on this string, application executes query to calculate, for example, the average salary in departments, represented by those ids. The most obvious solution is to execute a dynamic query with concatenated "IN" clause – unfortunately this approach causes a lot of hard parsing.

To resolve our problem, using MODEL syntax we will use the fact, that assigning new value to measure which address (dimension) doesn't exists in the array, will create a new element. Regular expressions will be also very helpful. Have you ever seen a FOR loop in SQL? ☺

1 as i [for i from 1 to regexp_count(ids[1],'[0-9]+') increment 1]	'10,20,50,40' as IDS	0 as ID *regexp_substr(ids[1],'[0-9]+',1,cv(i))*
1	'10,20,50,40'	*10*
2	'10,20,50,40'	*20*
3	'10,20,50,40'	*50*
4	'10,20,50,40'	*40*

The final query:

```
select id
from dual
model
dimension by (1 as i)
measures ('10,20,50,40' as ids,0 as id)
rules
(
  id[for i from 1 to regexp_count(ids[1],'[0-9]+') increment 1]=
        regexp_substr(ids[1],'[0-9]+',1,cv(i))
);
```

To calculate our average among departments we could use this query like this:

```
with v_ids as
(
select id
from dual
model
dimension by (1 as i)
measures ('10,20,50,40' as ids,0 as id)
rules
(
  id[for i from 1 to regexp_count(ids[1],'[0-9]+') increment 1]=
        regexp_substr(ids[1],'[0-9]+',1,cv(i))
```

```
)
)
select avg(salary)
from employees e, v_ids i
where e.department_id=i.id;
```

Of course this “WITH” clause (CTE) to every SQL query we want to execute, would be very uncomfortable. That’s why I suggest a using simple pipeline function (this is not the subject of our divagations but what the hell!):

```
create type o_numbers as object (
  x number);
/

create type t_o_numbers as table of o_numbers;
/

create or replace function split_numbers(p_str varchar2)
                           return t_o_numbers pipelined
is
begin
  for i in 1..regexp_count(p_str,'[0-9]+') loop
    pipe row (o_numbers(regexp_substr(p_str,'[0-9]+',1,i)));
  end loop;
end;
/
```

So in the above example of calculating average I could use this function like this:

```
select avg(salary)
from employees e,table(split_numbers('10,20,50,40')) ids
where e.department_id=ids.x;
```

CASE 5 – Generating subaccounts

In an insurance company we have had the following problem: there was a table with columns, representing ID of main accounts and the number of subaccounts to generate. Of course rule for generating subaccount ID was quite complicate but for training purposes let’s assume that it was: MAIN_ACCOUNT_ID.NEXT_SUBACCOUNT.

We have to generate sample data for this case:

```
create table accounts
(
  account_id varchar2(30),
  number_of_subaccounts number
);

insert into accounts
values ('A',2);

insert into accounts
values ('B',1);
```

```
insert into accounts
values ('C',5);

insert into accounts
values ('D',3);

commit;
```

OK, now all we have to do is to answer a simple question – which column is the measure and which is the dimension? Well, I see only two measures here and no dimension. Additionally, because I want to generate separate subaccounts for each account I'd like to execute rules separately for each unique account – so ACCOUNT_ID should be used in PARTITION BY and in MEASURES clause.

When I issue a PARTITION BY clause on my rowset I'll have four one-element arrays – so main account id and number of accounts will be at the first position in the array. This fact gives me a simple way to manipulate a FOR loop. So the final question is – where is my dimension? How to index an array? Well, the simplest way is to generate an artificial dimension like in "CASE 3 – Aggregating rows", but this time I don't have to use ROW_NUMBER() function because, as I mentioned before, at the beginning I'm having arrays with just one element. Let's see the solution:

```
select account_id, number_of_subaccounts, subaccount_id
from accounts
model
partition by (account_id)
dimension by (1 as i)
measures (account_id as aid, number_of_subaccounts,
          cast(null as varchar2(30)) as subaccount_id)
rules
(
  subaccount_id[for i from 2 to number_of_subaccounts[1]+1 increment 1]=
          aid[1] || '.' || (cv(i)-1)
)
order by account_id, subaccount_id nulls first;
```

(NOTE: If you are using the same column more than once in MODEL syntax, you have to use an alias. Notice, that thanks to the fact, that at the beginning I had only one element per each array I can find the number of subaccounts (to determine the number of loops) very easily – I have used the same feature to determine the main account id for generating subaccounts.)

A few words about performance

OK maybe this syntax is useful and interesting, but what about performance? How can I use it to scale up my queries? To answer that question we have to use a little bit bigger tables than in previous examples – fortunately we have SH schema ☺

Let's look closer on this query:

```
select sum(s.amount_sold) as sum_sold,  t.calendar_quarter_desc,
                                        p.prod_category,
                                        c.cust_first_name,
                                        c.cust_last_name
from sales s, products p, times t, customers c
where s.prod_id=p.prod_id
and   s.time_id=t.time_id
and   s.cust_id=c.cust_id
group by t.calendar_quarter_desc,
         p.prod_category,
         c.cust_first_name,
         c.cust_last_name;
```

Explain Plan × Query Result ×

SQL | Fetched 500 rows in 3,673 seconds

	SUM_SOLD	CALENDAR_QUARTER_DESC	PROD_CATEGORY	CUST_FIRST_NAME	CUST_LAST_NAME
1	782,65	2001-01	Peripherals and Accessories	Gabriel	Whitehead
2	2060,95	2001-01	Peripherals and Accessories	Reginald	Levi
3	4299,29	2001-01	Peripherals and Accessories	Madeline	Conard
4	664,04	2001-01	Peripherals and Accessories	Frederick	Gilmore
5	629,58	2001-01	Peripherals and Accessories	Deloris	Eaton
6	1439,22	2001-01	Peripherals and Accessories	Madge	Gimes
7	15238,56	2001-01	Peripherals and Accessories	Una	Linden
8	9642,91	2001-01	Peripherals and Accessories	Urban	Ogletree
9	2537,51	2001-01	Peripherals and Accessories	Lolita	Katz
10	29,67	2001-01	Peripherals and Accessories	Idola	Tavener
11	121,71	2001-01	Peripherals and Accessories	Morel	Gregory
12	2353,32	2001-01	Software/Other	Gladys	Roberts
13	108,99	2001-01	Software/Other	Rosamond	Colven
14	929,94	2001-01	Software/Other	Bette	Stock
15	687,48	1998-04	Electronics	Orilla	Riffken
16	45,35	1998-04	Electronics	Ronald	Geiss

Based on the results of this query I can tell, that Gabriel Whitehead spent on 'Peripherals and Accessories' 782,65$ in the first quarter of 2001 year. Let's extend this analyze – I want to know what is average amount spent on the same product category in the same quarter, by people who spent more money than the examined customer. To achieve this I will issue the following query:

```
with v_sold as
(
select sum(s.amount_sold) as sum_sold,  t.calendar_quarter_desc,
                                        p.prod_category,
                                        c.cust_first_name,
                                        c.cust_last_name
from sales s, products p, times t, customers c
where s.prod_id=p.prod_id
and   s.time_id=t.time_id
and   s.cust_id=c.cust_id
group by t.calendar_quarter_desc,
         p.prod_category,
         c.cust_first_name,
         c.cust_last_name
)
select s.*,
           (select avg(sum_sold)
```

```
              from v_sold s2
              where s2.sum_sold>s.sum_sold
              and   s2.prod_category=s.prod_category
              and   s2.calendar_quarter_desc=s.calendar_quarter_desc) as avg_better
from v_sold s;
```

Explain Plan × | Query Result ×

SQL | Fetched 500 rows in 7,378 seconds

	SUM_SOLD	CALENDAR_QUARTER_DESC	PROD_CATEGORY	CUST_FIRST_NAME	CUST_LAST_NAME	AVG_BETTER
1	1293,74	1998-01	Photo	Finlay	Hurst	3232,74866425992779783393501
2	5572,56	1998-01	Photo	Iris	Litefoote	7726,5216666666666666666666
3	2804,86	1998-01	Photo	Teresa	Maine	4389,2583333333333333333333
4	2809,83	1998-01	Photo	Oriena	Baley	4421,54789115646258503401360
5	1267,64	1998-01	Photo	Bertha	Kuehler	3198,24936170212765957446808
6	1232,99	1998-01	Photo	Ada	Nenninger	3069,39198675496688741721854
7	3285,97	1998-01	Peripherals and Accessories	Yola	Mulligan	5108,89563636363636363636363
8	2386,67	1998-01	Peripherals and Accessories	Rachel	Lowers	4168,74232142857142857142857
9	2507,08	1998-01	Peripherals and Accessories	Guido	Resnick	4294,40118773946360153256704

Now I can see that Finlay Hurst spent on 'Photo' category 1293,74$ in the first quarter of 1998 year and average sale from people who spent more on that category in the same quarter is about 3232,75$.

OK to spice up things a little let's say that I want to find out how many customers have the SUM_SOLD/AVG_BETTER ratio lower then 0,0005.

I will use two queries – without modeling:

```
with v_sold as
(
select sum(s.amount_sold) as sum_sold,  t.calendar_quarter_desc,
                                        p.prod_category,
                                        c.cust_first_name,
                                        c.cust_last_name
from sales s, products p, times t, customers c
where s.prod_id=p.prod_id
and   s.time_id=t.time_id
and   s.cust_id=c.cust_id
group by t.calendar_quarter_desc,
         p.prod_category,
         c.cust_first_name,
         c.cust_last_name
), v_avg_better as (
select s.*,
           (select avg(sum_sold)
            from v_sold s2
            where s2.sum_sold>s.sum_sold
            and   s2.prod_category=s.prod_category
            and   s2.calendar_quarter_desc=s.calendar_quarter_desc) as avg_better
from v_sold s)
select /*+ gather_plan_statistics */ *
from v_avg_better
where sum_sold/avg_better<=0.0005
```

And with the modeling:

```
with v_sold as
(
select sum(s.amount_sold) as sum_sold,  t.calendar_quarter_desc,
                                        p.prod_category,
                                        c.cust_first_name,
```

```
                                                                       c.cust_last_name
from sales s, products p, times t, customers c
where s.prod_id=p.prod_id
and   s.time_id=t.time_id
and   s.cust_id=c.cust_id
group by t.calendar_quarter_desc,
         p.prod_category,
         c.cust_first_name,
         c.cust_last_name
), v_model as (
select sum_sold,calendar_quarter_desc,prod_category,cust_first_name,cust_last_name,
       avg_better,sum_sold/avg_better as ratio
from v_sold
model unique single reference
dimension by (sum_sold,calendar_quarter_desc,prod_category)
measures (0 as avg_better, sum_sold as ss,
          cust_first_name,
          cust_last_name)
rules
(
  avg_better[any,any,any]=avg(ss)[sum_sold>cv(),cv(),cv()]
)
)
select /*+ gather_plan_statistics */ *
from v_model
where ratio<=0.0005
```

(NOTE: The *gather_plan_statistics* hint allows for the collection of extra metrics during the execution of the query – thanks to it we can display more accurate explain plan for the query. Each query will be run after flushing the buffer cache and the shared pool and with the 10046 event enabled.)

```
SQL> set timing on
SQL> set pagesize 100
SQL> set linesize 250
SQL> alter session set tracefile_identifier='q_nomodel';

Session altered.

Elapsed: 00:00:00.01
SQL> alter session set events '10046 trace name context forever, level 12';

Session altered.

Elapsed: 00:00:00.00
SQL> get nomodel_nop
with v_sold as
(
select sum(s.amount_sold) as sum_sold,  t.calendar_quarter_desc,
                                        p.prod_category,
                                        c.cust_first_name,
                                        c.cust_last_name
from sales s, products p, times t, customers c
where s.prod_id=p.prod_id
and   s.time_id=t.time_id
and   s.cust_id=c.cust_id
group by t.calendar_quarter_desc,
         p.prod_category,
         c.cust_first_name,
         c.cust_last_name
), v_avg_better as (
select s.*,
           (select avg(sum_sold)
            from v_sold s2
            where s2.sum_sold>s.sum_sold
            and   s2.prod_category=s.prod_category
            and   s2.calendar_quarter_desc=s.calendar_quarter_desc) as avg_better
from v_sold s)
select /*+ gather_plan_statistics */ *
from v_avg_better
where sum_sold/avg_better<=0.0005
SQL> /

no rows selected

Elapsed: 00:40:42.22
```

As you can see, the first query execution was longer than 40 minutes. What about second query – with modeling?

```
SQL> set timing on
SQL> set pagesize 100
SQL> set linesize 250
SQL> alter session set tracefile_identifier='q_model';

Session altered.

Elapsed: 00:00:00.00
SQL> alter session set events '10046 trace name context forever, level 12';

Session altered.

Elapsed: 00:00:00.06
SQL> get model_nop
  with v_sold as
  (
  select sum(s.amount_sold) as sum_sold,  t.calendar_quarter_desc,
                                          p.prod_category,
                                          c.cust_first_name,
                                          c.cust_last_name
  from sales s, products p, times t, customers c
  where s.prod_id=p.prod_id
  and   s.time_id=t.time_id
  and   s.cust_id=c.cust_id
  group by t.calendar_quarter_desc,
           p.prod_category,
           c.cust_first_name,
           c.cust_last_name
  ), v_model as (
  select sum_sold,calendar_quarter_desc,prod_category,cust_first_name,cust_last_name,
         avg_better,sum_sold/avg_better as ratio
  from v_sold
  model unique single reference
  dimension by (sum_sold,calendar_quarter_desc,prod_category)
  measures (0 as avg_better, sum_sold as ss,
            cust_first_name,
            cust_last_name)
  rules
  (
    avg_better[any,any,any]=avg(ss)[sum_sold>cv(),cv(),cv()]
  )
  )
  select /*+ gather_plan_statistics */ *
  from v_model
  where ratio<=0.0005
SQL> /

no rows selected

Elapsed: 00:08:42.28
```

Only 8 minutes and 42 seconds!

Let's compare explain plans from our execution – thanks to *gather_plan_statistics* hint we can use DBMS_XPLAN.DISPLAY_CURSOR function with 'ALLSTATS LAST' parameter, which will show us much more details than regular explain plan.

Quoting after Oracle documentation:

> ALLSTATS - A shortcut for 'IOSTATS MEMSTATS'
>
> IOSTATS - assuming that basic plan statistics are collected when SQL statements are executed (either by using the gather_plan_statistics hint or by setting the parameter statistics_level to ALL), this format shows IO statistics for ALL (or only for the LAST as shown below) executions of the cursor.
>
> MEMSTATS - Assuming that PGA memory management is enabled (that is, pga_aggregate_target parameter is set to a non 0 value), this format allows to display memory management statistics (for example, execution mode of the operator, how much memory was used, number of bytes spilled to disk, and so on). These statistics only apply to memory intensive operations like hash-joins, sort or some bitmap operators.
>
> LAST - By default, plan statistics are shown for all executions of the cursor. The keyword LAST can be specified to see only the statistics for the last execution.

Explain plan for the query without modeling:

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------
SQL_ID  6k6akna567d6j, child number 0
-------------------------------------
with v_sold as ( select sum(s.amount_sold) as sum_sold,
t.calendar_quarter_desc,
p.prod_category,
c.cust_first_name,
c.cust_last_name from sales s, products p, times t, customers c where
s.prod_id=p.prod_id and   s.time_id=t.time_id and   s.cust_id=c.cust_id
group by t.calendar_quarter_desc,          p.prod_category,
c.cust_first_name,          c.cust_last_name ), v_avg_better as (
select s.*,          (select avg(sum_sold)          from v_sold s2
          where s2.sum_sold>s.sum_sold          and
s2.prod_category=s.prod_category          and
s2.calendar_quarter_desc=s.calendar_quarter_desc) as avg_better from
v_sold s) select /*+ gather_plan_statistics */ * from v_avg_better
where sum_sold/avg_better<=0.0005

Plan hash value: 3581748979

----------------------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                   | Name                         | Starts | E-Rows | A-Rows |   A-Time    | Buffers | Reads  | Writes |  OMem |  1Mem | Used-Mem |
----------------------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                              |      1 |        |      0 |00:40:41.48 |     44M|     44M|    556 |       |       |          |
|   1 |  SORT AGGREGATE             |                              |  80227 |      1 |  80227 |00:40:36.80 |     44M|     44M|      0 |       |       |          |
|*  2 |   VIEW                      |                              |  80227 |   918K|     50M|00:53:09.49 |     44M|     44M|      0 |       |       |          |
|   3 |    TABLE ACCESS FULL        | SYS_TEMP_0FD9D662B_211DFD    |  80227 |   918K|   6714M|14:01:04.29 |     44M|     44M|      0 |       |       |          |
|   4 |  TEMP TABLE TRANSFORMATION  |                              |      1 |        |      0 |00:40:41.48 |     44M|     44M|    556 |       |       |          |
|   5 |   LOAD AS SELECT            |                              |      1 |        |      0 |00:00:02.14 |   3722 |   3158 |    556 |   529K|   529K|  529K (0)|
|   6 |    HASH GROUP BY            |                              |      1 |   918K|  83691 |00:00:02.61 |   3154 |   3156 |      0 |  8321K|  2082K|   17M (0)|
|*  7 |     HASH JOIN               |                              |      1 |   918K|   918K|00:01:29.58 |   3154 |   3156 |      0 |  1049K|  1049K| 1346K (0)|
|   8 |      TABLE ACCESS FULL      | TIMES                        |      1 |  1826 |   1826 |00:00:00.01 |     55 |     54 |      0 |       |       |          |
|*  9 |      HASH JOIN              |                              |      1 |   918K|   918K|00:00:57.86 |   3099 |   3102 |      0 |   915K|   915K| 1244K (0)|
|  10 |       VIEW                  | index$_join$_002             |      1 |     72 |     72 |00:00:00.01 |      6 |     12 |      0 |       |       |          |
|* 11 |        HASH JOIN            |                              |      1 |        |     72 |00:00:00.01 |      6 |     12 |      0 |  1096K|  1096K| 1581K (0)|
|  12 |         INDEX FAST FULL SCAN| PRODUCTS_PK                  |      1 |     72 |     72 |00:00:00.01 |      3 |      6 |      0 |       |       |          |
|  13 |         INDEX FAST FULL SCAN| PRODUCTS_PROD_CAT_IX         |      1 |     72 |     72 |00:00:00.01 |      3 |      6 |      0 |       |       |          |
|* 14 |       HASH JOIN             |                              |      1 |   918K|   918K|00:00:37.63 |   3093 |   3090 |      0 |  3151K|  1363K| 4707K (0)|
|  15 |        TABLE ACCESS FULL    | CUSTOMERS                    |      1 |  55500 |  55500 |00:00:01.38 |   1458 |   1455 |      0 |       |       |          |
|  16 |        PARTITION RANGE ALL  |                              |      1 |   918K|   918K|00:00:16.41 |   1635 |   1635 |      0 |       |       |          |
|  17 |         TABLE ACCESS FULL   | SALES                        |     28 |   918K|   918K|00:00:05.41 |   1635 |   1635 |      0 |       |       |          |
|* 18 |   VIEW                      |                              |      1 |   918K|      0 |00:40:39.34 |     44M|     44M|      0 |       |       |          |
|  19 |    VIEW                     |                              |      1 |   918K|  83691 |00:00:01.44 |    561 |    557 |      0 |       |       |          |
|  20 |     TABLE ACCESS FULL       | SYS_TEMP_0FD9D662B_211DFD    |      1 |   918K|  83691 |00:00:00.47 |    561 |    557 |      0 |       |       |          |
----------------------------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(("S2"."SUM_SOLD">:B1 AND "S2"."PROD_CATEGORY"=:B2 AND "S2"."CALENDAR_QUARTER_DESC"=:B3))
   7 - access("S"."TIME_ID"="T"."TIME_ID")
   9 - access("S"."PROD_ID"="P"."PROD_ID")
  11 - access(ROWID=ROWID)
  14 - access("S"."CUST_ID"="C"."CUST_ID")
```

Explain plan for the query with modeling:

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------
SQL_ID  dkps6afxk00wx, child number 0
-------------------------------------
with v_sold as ( select sum(s.amount_sold) as sum_sold,
t.calendar_quarter_desc,
p.prod_category,
c.cust_first_name,
c.cust_last_name from sales s, products p, times t, customers c where
s.prod_id=p.prod_id and   s.time_id=t.time_id and   s.cust_id=c.cust_id
group by t.calendar_quarter_desc,          p.prod_category,
c.cust_first_name,          c.cust_last_name ), v_model as ( select
sum_sold,calendar_quarter_desc,prod_category,cust_first_name,cust_last_n
ame,        avg_better,sum_sold/avg_better as ratio from v_sold model
unique single reference dimension by
(sum_sold,calendar_quarter_desc,prod_category) measures (0 as
avg_better, sum_sold as ss,          cust_first_name,
cust_last_name) rules (   avg_better[any,any,any]=avg(ss)[sum_sold>cv(),
cv(),cv()] ) ) select /*+ gather_plan_statistics */ * from v_model
where ratio<=0.0005

Plan hash value: 2907411999

----------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name                 | Starts | E-Rows | A-Rows |   A-Time    | Buffers | Reads  |  OMem |  1Mem | Used-Mem |
----------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                      |      1 |        |      0 |00:08:06.58 |    3154 |   3156 |       |       |          |
|*  1 |  VIEW                          |                      |      1 |   918K|      0 |00:08:06.58 |    3154 |   3156 |       |       |          |
|   2 |   SQL MODEL ORDERED            |                      |      1 |   918K|  83691 |00:00:33.71 |    3154 |   3156 |  9622K|  2041K|   12M (0)|
|   3 |    HASH GROUP BY               |                      |      1 |   918K|  83691 |00:00:03.26 |    3154 |   3156 |  8321K|  2082K|   17M (0)|
|*  4 |     HASH JOIN                  |                      |      1 |   918K|   918K|00:01:29.50 |    3154 |   3156 |  1049K|  1049K| 1339K (0)|
|   5 |      PART JOIN FILTER CREATE   | :BF0000              |      1 |  1826 |   1826 |00:00:00.04 |      55 |     54 |       |       |          |
|   6 |       TABLE ACCESS FULL        | TIMES                |      1 |  1826 |   1826 |00:00:00.02 |      55 |     54 |       |       |          |
|*  7 |      HASH JOIN                 |                      |      1 |   918K|   918K|00:00:58.25 |    3099 |   3102 |   915K|   915K| 1284K (0)|
|   8 |       VIEW                     | index$_join$_002     |      1 |     72 |     72 |00:00:00.01 |       6 |     12 |       |       |          |
|*  9 |        HASH JOIN               |                      |      1 |        |     72 |00:00:00.01 |       6 |     12 |  1096K|  1096K| 1581K (0)|
|  10 |         INDEX FAST FULL SCAN   | PRODUCTS_PK          |      1 |     72 |     72 |00:00:00.01 |       3 |      6 |       |       |          |
|  11 |         INDEX FAST FULL SCAN   | PRODUCTS_PROD_CAT_IX |      1 |     72 |     72 |00:00:00.01 |       3 |      6 |       |       |          |
|* 12 |       HASH JOIN                |                      |      1 |   918K|   918K|00:00:37.04 |    3093 |   3090 |  3151K|  1363K| 4705K (0)|
|  13 |        TABLE ACCESS FULL       | CUSTOMERS            |      1 |  55500 |  55500 |00:00:01.51 |    1458 |   1455 |       |       |          |
|  14 |        PARTITION RANGE JOIN-FILTER|                   |      1 |   918K|   918K|00:00:16.22 |    1635 |   1635 |       |       |          |
|  15 |         TABLE ACCESS FULL      | SALES                |     20 |   918K|   918K|00:00:05.61 |    1635 |   1635 |       |       |          |
----------------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("RATIO"<=.0005)
   4 - access("S"."TIME_ID"="T"."TIME_ID")
   7 - access("S"."PROD_ID"="P"."PROD_ID")
   9 - access(ROWID=ROWID)
  12 - access("S"."CUST_ID"="C"."CUST_ID")
```

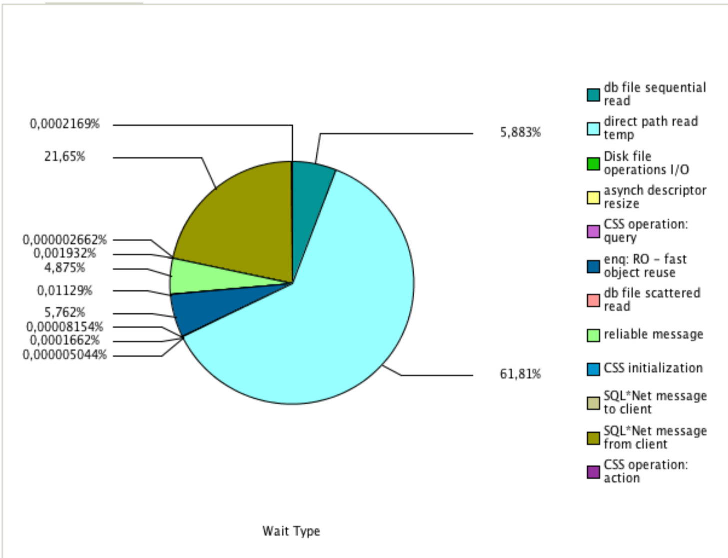
As we can see, the first query caused an excessive temporary space usage. This has happened because I have used correlated subquery to access the same CTE (Common Table Expression – the WITH V_SOLD clause) twice – the Cost Based Optimizer transformed CTE into internal temporary table. We can find the appropriate DDL in the trace file:

```
CREATE GLOBAL TEMPORARY TABLE "SYS"."SYS_TEMP_0FD9D662B_211DFD" ("C0" NUMBER,
  "C1" CHARACTER(7),"C2" VARCHAR2(50),"C3" VARCHAR2(20),"C4" VARCHAR2(40) )
  IN_MEMORY_METADATA CURSOR_SPECIFIC_SEGMENT STORAGE (OBJNO 4254950955 )
  NOPARALLEL
```

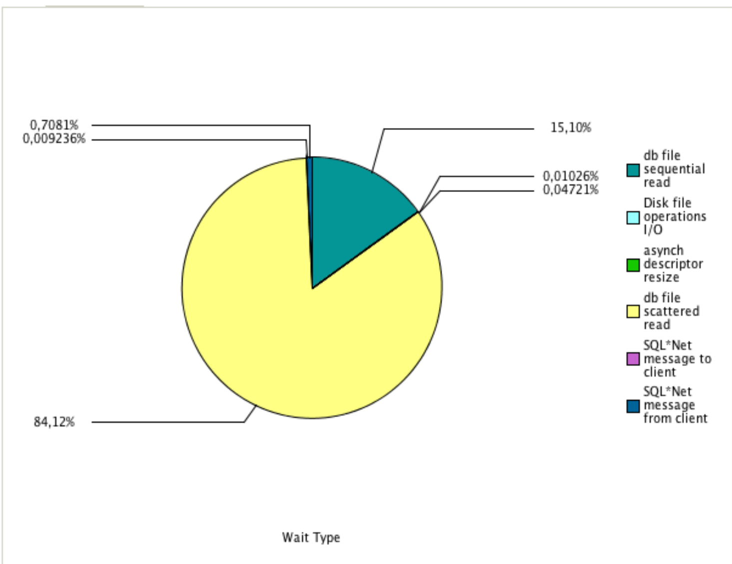
In the second query I have used the MODEL syntax to avoid accessing the same rowset twice – in this example the "V_SOLD" CTE.

Here you can see histograms of wait events for both queries (charts where produced by excellent tool – Trace Analyzer – developed by Dominic Giles).

No modeling:



Modeling:



Now let's see execution times of this two queries without the additional trace and statistics overhead but with the PARALLEL hint – each query will be executed two times: first time with empty buffers and again just after the previous execution.

Query with correlated subquery:

```
SQL> set timing on
SQL> get nomodel
with v_sold as
(
select /*+ parallel(s 4) */ sum(s.amount_sold) as sum_sold,  t.calendar_quarter_desc,
                                        p.prod_category,
                                        c.cust_first_name,
                                        c.cust_last_name
from sales s, products p, times t, customers c
where s.prod_id=p.prod_id
and   s.time_id=t.time_id
and   s.cust_id=c.cust_id
group by t.calendar_quarter_desc,
         p.prod_category,
         c.cust_first_name,
         c.cust_last_name
), v_avg_better as (
select s.*,
           (select avg(sum_sold)
            from v_sold s2
            where s2.sum_sold>s.sum_sold
            and   s2.prod_category=s.prod_category
            and   s2.calendar_quarter_desc=s.calendar_quarter_desc) as avg_better
from v_sold s)
select *
from v_avg_better
where sum_sold/avg_better<=0.0005
SQL> /

no rows selected

Elapsed: 00:03:29.87
SQL> /

no rows selected

Elapsed: 00:03:24.13
SQL>
```

Query with modeling:

```
SQL> set timing on
SQL> get model
with v_sold as
(
select /*+ parallel(s 4) */ sum(s.amount_sold) as sum_sold,  t.calendar_quarter_desc,
                                        p.prod_category,
                                        c.cust_first_name,
                                        c.cust_last_name
from sales s, products p, times t, customers c
where s.prod_id=p.prod_id
and   s.time_id=t.time_id
and   s.cust_id=c.cust_id
group by t.calendar_quarter_desc,
         p.prod_category,
         c.cust_first_name,
         c.cust_last_name
), v_model as (
select sum_sold,calendar_quarter_desc,prod_category,cust_first_name,cust_last_name,
       avg_better,sum_sold/avg_better as ratio
from v_sold
model unique single reference
dimension by (sum_sold,calendar_quarter_desc,prod_category)
measures (0 as avg_better, sum_sold as ss,
          cust_first_name,
          cust_last_name)
rules
(
  avg_better[any,any,any]=avg(ss)[sum_sold>cv(),cv(),cv()]
)
)
select *
from v_model
where ratio<=0.0005
SQL> /

no rows selected

Elapsed: 00:00:15.73
SQL> /

no rows selected

Elapsed: 00:00:15.06
```

Fascinating – isn't it? Goodbye and Happy Modeling! ☺