Table of Contents

1. Aggregate () over ()

Suppose you want to display each person's salary and how much their salary is over the average salary for all employees.

We'll start by looking just at department 20. We have four employees. The sum of their salaries is 81000 ; the average salary (rounded to an integer) is 20250 . We could do this with a subquery in the Select.

```
variable dpt number
exec :dpt := 20;
```

Demo 01: Using a subquery and the avg function

```
select emp_id, salary
, Round(salary -
        (select Avg(salary)
         from adv_emp
         where dept_id = :dpt),2)  as Over_under_avg
from adv_emp
where dept_id = :dpt
;
```

```
EMP_ID     SALARY OVER_UNDER_AVG
------ ---------- --------------
302         14000          -6250
303         27000           6750
312         28000           7750
315         12000          -8250
```

Demo 02: You could also use a CTE and a Cross join. Be sure you understand why a cross join will work here

```
with avgSal as (
   select Avg(salary * 1.0)  as AvgDept100
   from adv_emp
   where dept_id = :dpt
   )
select emp_id, salary
, Round(salary - AvgDept100,2) as Over_under_avg
from adv_emp
cross join avgSal
where dept_id = :dpt;
```

Demo 03: Using avg() Over()

We can also do this with an Avg() Over function to get the same result. This is a much simpler syntax than the subquery. If we just do the *avg(salary) over()* for the second column, you can see that the average is calculated over dept_id 20 for each row.

So we do the subtraction to get the offset of each employee's salary compared to the average.

```
   select emp_id, salary
   , Round(salary - ( Avg(salary) Over() ) ,2 ) as Over_under_avg
   from  adv_emp
   where  dept_id = :dpt;
```

In this case the Over() clause has no argument; this is referred to as a Null Over clause and it means that the function applies to the entire dataset- since the function is calculated after the Where clause, this means to the rows for dept_id 20 only. In the subquery version we had to do the filter in both the subquery and the parent query to get the correct result.

1.1. Partition

We might want to look at all the employees and check their over_under_avg based on their dept_id only. This means we want to group the employees by dept_id and calculate the average for each group separately. That is a partition. This query will give that result. I have sorted by Dept ID and salary to make the output easier to read.

Demo 04: Using avg() Over() with a partition

```
   select dept_id, salary
   , Round(salary - ( Avg(salary) Over( Partition by dept_id) ), 0 )
              as Over_under_avg
   , emp_id
   from  adv_emp
   order by dept_id, emp_id
   ;
```

```
   DEPT_ID     SALARY OVER_UNDER_AVG EMP_I
---------- ---------- -------------- -----
        10      15000         -11111 301
        10      27000            889 305
        10      30000           3889 309
        10      25000          -1111 310
        10      28000           1889 311
        10      30000           3889 319
        10      25000          -1111 320
        10      30000           3889 321
        10      25000          -1111 322
        15      25000              0 323
        20      14000          -6250 302
        20      27000           6750 303
        20      28000           7750 312
        20      12000          -8250 315
        30      28000           7111 304
        30      28000           7111 306
        30      13500          -7389 307
        30      15000          -5889 308
        30      11000          -9889 313
        30      30000           9111 314
        30      26000           5111 316
        30      25000           4111 317
        30      11500          -9389 318

23 rows selected.
```

Demo 05: Who earned more than the average salary for their department?

```
with Aggs as (
  select emp_id, dept_id, salary
  , Avg(salary) Over( Partition by dept_id) as dept_avg
  from  adv_emp
  )
select emp_id, dept_id, salary
from aggs
where salary > dept_avg
order by dept_id, emp_id;
```

```
EMP_I    DEPT_ID     SALARY
----- ---------- ----------
305           10      27000
309           10      30000
311           10      28000
319           10      30000
321           10      30000
303           20      27000
312           20      28000
304           30      28000
306           30      28000
314           30      30000
316           30      26000
317           30      25000
12 rows selected.
```

Demo 06: Calculating Percent of total: Now we want to know what each employee's salary is as a percent of the total salary for that department.

```
select dept_id, emp_id, salary
, round(salary  /(sum(salary) over (partition by dept_id) ) * 100, 2)
           as percent_dept_salary
from  adv_emp
order by dept_id, salary;
```

```
   DEPT_ID EMP_I     SALARY PERCENT_DEPT_SALARY
---------- ----- ---------- -------------------
        10 301        15000                6.38
        10 320        25000               10.64
        10 310        25000               10.64
        10 322        25000               10.64
        10 305        27000               11.49
        10 311        28000               11.91
        10 319        30000               12.77
        10 321        30000               12.77
        10 309        30000               12.77
        15 323        25000                 100
        20 315        12000               14.81
        20 302        14000               17.28
        20 303        27000               33.33
        20 312        28000               34.57
        30 313        11000                5.85
        30 318        11500                6.12
        30 307        13500                7.18
        30 308        15000                7.98
        30 317        25000                13.3
        30 316        26000               13.83
        30 306        28000               14.89
        30 304        28000               14.89
        30 314        30000               15.96

23 rows selected.
```

Start by looking at the results for dept 15. There is one employee, with a salary of 25000. This row reports as 100% of the department salary total. Then look at the results for dept 20. There are four employees. The total salary for dept 210 is 28000. Employee 315 has a salary of 12000 which is about 15% of the department total salary.

You could get the same result with the **`ratio_to_report`** function:

```
ratio_to_report ( salary ) over ( partition by dept_id )
```

Demo 07: Using Ratio_to_Report

```
select dept_id, emp_id, salary
, round(ratio_to_report ( salary ) over ( partition by dept_id ) * 100,2)
  as percent_dept_salary
from  adv_emp
order by dept_id, salary;
```

1.2. ListAgg

This is another aggregate function you can use with the analytical techniques. ListAgg returns a concatenated set of values.

Demo 08: Simple aggregate of the names with a semicolon followed by a space as the delimiter, We get a single row returned with the employees last names.

```
select listagg(name_last, '; ')
       within group (order by name_last) as  "Employee List"
from adv_emp;
```

```
Employee List
--------------------------------------------------------------------------------
Battaglia; Beiderbecke; Brubeck; Cohen; Coltrane; Davis; Ellington; Evans; Green; Hancock;
Jarrett; Mobley; Monk; Montgomery; Quebec; Redman; Rollins; Shorter; Tatum; Turrentine;
Wabich; Wabich; Wasliewski
```

Demo 09: We can order the values by salary in descending order

```
select listagg(name_last, '; ')
        within group (order by salary desc) as  "Employee List"
from adv_emp;
```

```
Employee List
--------------------------------------------------------------------------------
Beiderbecke; Redman; Rollins; Turrentine; Brubeck; Cohen; Ellington; Mobley; Coltrane;
Quebec; Monk; Jarrett; Montgomery; Wabich; Wabich; Wasliewski; Evans; Green; Hancock;
Tatum; Battaglia; Shorter; Davis
```

Demo 10: We can add a regular Group by clause and listagg returns aggregates for each group

```
select dept_id as  "Dept",
  listagg(name_last, '; ')
     within group (order by year_hired) as  "Employees by Year Hired"
from adv_emp
group by dept_id;
```

```
Dept Employees by Year Hired
---- -----------------------------------------------------------
  10 Green; Brubeck; Coltrane; Jarrett; Wabich; Wabich; Beiderbecke; Redman; Rollins
  15 Montgomery
  20 Ellington; Hancock; Battaglia; Quebec
  30 Cohen; Mobley; Davis; Tatum; Evans; Monk; Turrentine; Shorter; Wasliewski  10 King

4 rows selected.
```

You can also put the dept_id into a partition by clause in the function. Also try this without the Distinct keyword.

```
   Select Distinct dept_id As "Dept"
   , listagg(name_last, '; ')
      WITHIN GROUP
      (ORDER BY year_hired)
      OVER (PARTITION BY dept_id) as  "Employees by Year Hired"
   from adv_emp;
```

```
Dept Employees by Year Hired
---- ----------------------------------------------------------------------------------
  15 Montgomery
  30 Cohen; Mobley; Davis; Tatum; Turrentine; Evans; Monk; Wasliewski; Shorter
  20 Hancock; Ellington; Battaglia; Quebec
  10 Green; Coltrane; Brubeck; Jarrett; Wabich; Wabich; Redman; Rollins; Beiderbecke

4 rows selected.
```

2. Running Totals

We can calculate the total of the salaries and get one value for the rows in the table.

### Demo 11:	Simple aggregate functions over the table

```
      select sum(salary), count(*)
      from  adv_emp;
```

```
SUM(SALARY)   COUNT(*)
----------- ----------
     529000         23
```

Suppose we want a running total instead; we want the first row to show the total of the first salary; the second row to show the total of row 1to row 2; the third row to show the total of row 1to row 3, etc.

The following query does that; it sums the salary over a range consisting of all of the rows from the start (**unbounded preceding**) to the **current** row. We want the rows in the running total to be in emp_id order.

### Demo 12:	Running total over the table using Range Between

```
   select emp_id,  salary
   , sum(salary) over(
            order by emp_id
            range between unbounded preceding and current row)
            as run_tot
   from adv_emp
   order by emp_id
   ;
```

```
EMP_ID SALARY    RUN_TOT
------ ------ ----------
301     15000      15000
302     14000      29000
303     27000      56000
304     28000      84000
305     27000     111000
```

```
306     28000     139000
307     13500     152500
308     15000     167500
309     30000     197500
310     25000     222500
311     28000     250500
312     28000     278500
313     11000     289500
314     30000     319500
315     12000     331500
316     26000     357500
317     25000     382500
318     11500     394000
319     30000     424000
320     25000     449000
321     30000     479000
322     25000     504000
323     25000     529000

23 rows selected.
```

Be certain to have the two Order By clauses to sort on the same columns for the output to be understandable.

3. Windowing clause

By adding **`Partition BY department_id`**, we have told SQL to do a running total for each department, restarting the total when the department changes. We are ordering by dept_id, salary.

This is an example of a windowing frame- a set of rows creating a windows partition. The windowing frame is a moving frame. In this example the windowing frame consists of all rows from the last dept_id change (the partition by argument) up to and including the current row.

Demo 13: Running total by department, adding a partition clause.

```
select dept_id
,  emp_id
,  salary
,  sum(salary) over(
            PARTITION BY  DEPT_ID
            order by dept_id, salary
            RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
            as run_tot
from  adv_emp
order by dept_id, salary
;
```

I artificially added blank lines for each dept id break

```
   DEPT_ID EMP_ID SALARY    RUN_TOT
---------- ------ ------ ----------
        10 301     15000      15000
        10 320     25000      90000
        10 310     25000      90000
        10 322     25000      90000
        10 305     27000     117000
        10 311     28000     145000
        10 319     30000     235000
        10 321     30000     235000
        10 309     30000     235000

        15 323     25000      25000
        20 315     12000      12000
        20 302     14000      26000
        20 303     27000      53000
        20 312     28000      81000

        30 313     11000      11000
        30 318     11500      22500
        30 307     13500      36000
        30 308     15000      51000
        30 317     25000      76000
        30 316     26000     102000
        30 306     28000     158000
        30 304     28000     158000
        30 314     30000     188000
```

3.1. The Range phrase

The Range phrase is called the Windowing-clause and it defaults to range between unbounded preceding and current row.

Demo 14: This query uses the default and gives us the same output as above.

```
select dept_id, emp_id,  salary
, sum(salary) over(
         PARTITION BY DEPT_ID
         order by dept_id, salary
         )
           as run_tot
from  adv_emp
order by dept_id, salary;
```

Demo 15: Using the default range between unbounded preceding and current row and ordering by employee id

```
select emp_id,  salary
, sum(salary) over(
           order by emp_id
           --  range between unbounded preceding and current row
          )
           as run_tot
from adv_emp
order by emp_id;
```

```
EMP_I     SALARY    RUN_TOT
----- ---------- ----------
301        15000      15000
302        14000      29000
303        27000      56000
304        28000      84000
305        27000     111000
306        28000     139000
307        13500     152500
308        15000     167500
309        30000     197500
310        25000     222500
311        28000     250500
312        28000     278500
313        11000     289500
314        30000     319500
315        12000     331500
316        26000     357500
```

```
317         25000      382500
318         11500      394000
319         30000      424000
320         25000      449000
321         30000      479000
322         25000      504000
323         25000      529000
```

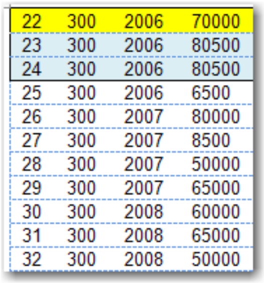
3.2. Moving Windowing Clause

The windowing-clause allows you to set a moving window of rows over which the function should be applied.

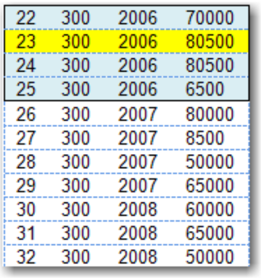
Suppose we want the current row and two rows before and after the current row. In the picture below the current row is highlighted in yellow and the window includes the blue and yellow highlighted rows

22	300	2006	70000
23	300	2006	80500
24	300	2006	80500
25	300	2006	6500
26	300	2007	80000
27	300	2007	8500
28	300	2007	50000
29	300	2007	65000
30	300	2008	60000
31	300	2008	65000
32	300	2008	50000
33	300	2008	80000

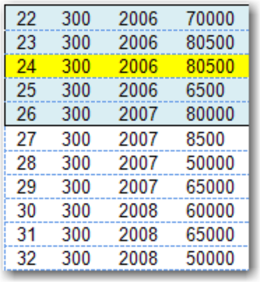
This shows the moving window frame. Note that at the start or the end of the dataset, the window will include less than 5 rows.



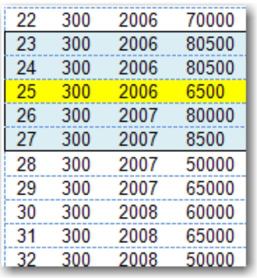
22	300	2006	70000
23	300	2006	80500
24	300	2006	80500
25	300	2006	6500
26	300	2007	80000
27	300	2007	8500
28	300	2007	50000
29	300	2007	65000
30	300	2008	60000
31	300	2008	65000
32	300	2008	50000



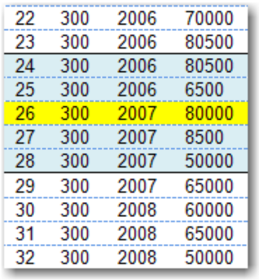
22	300	2006	70000
23	300	2006	80500
24	300	2006	80500
25	300	2006	6500
26	300	2007	80000
27	300	2007	8500
28	300	2007	50000
29	300	2007	65000
30	300	2008	60000
31	300	2008	65000
32	300	2008	50000



22	300	2006	70000
23	300	2006	80500
24	300	2006	80500
25	300	2006	6500
26	300	2007	80000
27	300	2007	8500
28	300	2007	50000
29	300	2007	65000
30	300	2008	60000
31	300	2008	65000
32	300	2008	50000



22	300	2006	70000
23	300	2006	80500
24	300	2006	80500
25	300	2006	6500
26	300	2007	80000
27	300	2007	8500
28	300	2007	50000
29	300	2007	65000
30	300	2008	60000
31	300	2008	65000
32	300	2008	50000

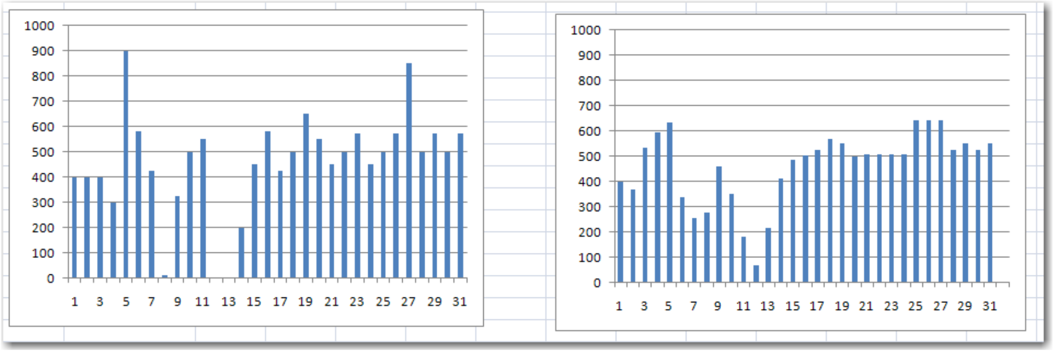


22	300	2006	70000
23	300	2006	80500
24	300	2006	80500
25	300	2006	6500
26	300	2007	80000
27	300	2007	8500
28	300	2007	50000
29	300	2007	65000
30	300	2008	60000
31	300	2008	65000
32	300	2008	50000

We can use several ways to express the range that compose a window. We can use a physical number of rows to be included or a term such as unbounded preceding. The windowing clause is often used when we have a series of values and we want to find a moving average. A moving average can be used with a time dependent series of data and we want to smooth out the data by looking at a three day average. That way if we have a few days that are somewhat out of the normal range they will be blended into the moving average. An example will help with this.

Suppose you were tracking sales over a period of several days. The sales values are apt to fluctuate each day. We could produce the graph of the sales shown on the left which shows each day's sales and we see some pretty big changes. We could also graph a three day average as shown on the right which smoothes out those one day changes. We still see variation in data but a single value does not show up as being that significant.

This is not a case of which graph is correct- but rather what do we want to see.



The test table is adv_sales with the first column being a number from 1-20 representing the day of the time period and the second column will be the sales for that day. The inserts are provided on the SQL included with these notes(demo 01). We can refer to this as a three-day average if we have a row, in the proper order, for each day form the first to the last - with no gaps or duplicates.

```
Create table adv_sales (
  sales_day number(2) primary key
, sales number (5) check (sales >= 0)
);
```

The window we will use include the current row and the 2 preceding rows.

Demo 16: Here the window is the current row and the 2 preceding rows. The Column statement is an SQL*Plus command to format the third column; it does not work in SQL Developer.

```
Column Three_day_avg format "9999.99"

select sales_day
, sales
, Avg(sales) OVER (
            ORDER  BY sales_day
            ROWS BETWEEN
            2 PRECEDING  AND CURRENT ROW
            ) AS three_day_avg
from  adv_sales
order by sales_day;
```

```
 SALES_DAY    SALES THREE_DAY_AVG
---------- -------- -------------
25-APR-15       400        400.00
26-APR-15       400        400.00
27-APR-15       400        400.00
28-APR-15       300        366.67
29-APR-15       900        533.33
30-APR-15       580        593.33
01-MAY-15       425        635.00
02-MAY-15        10        338.33
03-MAY-15       325        253.33
04-MAY-15       500        278.33
05-MAY-15       550        458.33
06-MAY-15         0        350.00
07-MAY-15         0        183.33
08-MAY-15       200         66.67
09-MAY-15       450        216.67
10-MAY-15       580        410.00
11-MAY-15       425        485.00
12-MAY-15       475        493.33
13-MAY-15       375        425.00
14-MAY-15       500        450.00
15-MAY-15       650        508.33
```

```
16-MAY-15       550        566.67
17-MAY-15       450        550.00
18-MAY-15       500        500.00
19-MAY-15       575        508.33
20-MAY-15       450        508.33
21-MAY-15       500        508.33
22-MAY-15       575        508.33
23-MAY-15       850        641.67
24-MAY-15       500        641.67
25-MAY-15       575        641.67
26-MAY-15       500        525.00
27-MAY-15       575        550.00
28-MAY-15       500        525.00
29-MAY-15       575        550.00
30-MAY-15       575        550.00
31-MAY-15       575        575.00
01-JUN-15       425        525.00
02-JUN-15       500        500.00
03-JUN-15       455        460.00
04-JUN-15         0        318.33
05-JUN-15         0        151.67
06-JUN-15         0          0.00
07-JUN-15         0          0.00
08-JUN-15       900        300.00
09-JUN-15       450        450.00
10-JUN-15       780        710.00
11-JUN-15       475        568.33
12-JUN-15       875        710.00
13-JUN-15       375        575.00
14-JUN-15       800        683.33
```

We can see that the last column has less variation since each value (except for the end points depending on row preceding or rows following) represents the average of thee data points. The low number and the high number do affect the three_Day_avg, but their effect is not as much as in the sales column.

The first two rows are not three row averages, since we do not yet have three rows of sale. What Oracle does is treat the missing rows as Nulls. We would need to consider if this is relevant to the purpose for which we are running the query.

It may be that our company sells more merchandise at the start or the end of the month. In that case, the three-day-average could be misleading. If we were storing data about weather and the data stored the high temperature for each day, we might not think that that is influenced by the start or end of the month. Queries like this are generally done with large amounts of data and often the end points can be ignored or removed.

However you always need to check these assumptions. Maybe some factory in the area does larger runs on the last two days of the month and this could influence the local temperature due to atmospheric conditions. Assumptions in statistics can hide the data you are trying to find.

Demo 17: Here the window extends over four rows, the current, two days preceding and one day following. It also limits the display to some of the rows in the table.

```
Column Four_day_avg format "9999.99"

select sales_day
, sales
, round(avg(sales) over (
           order by sales_day
           rows between 2 preceding and 1 following ), 2) as four_day_avg
from adv_sales
```

```
where extract (month from sales_day) = 5
and   extract (year from sales_day) = 2015
order   by sales_day;
```

```
SALES_DAY      SALES FOUR_DAY_AVG
--------- ---------- ------------
01-MAY-15        425       217.50
02-MAY-15         10       253.33
03-MAY-15        325       315.00
04-MAY-15        500       346.25
05-MAY-15        550       343.75
06-MAY-15          0       262.50
07-MAY-15          0       187.50
08-MAY-15        200       162.50
09-MAY-15        450       307.50
10-MAY-15        580       413.75
11-MAY-15        425       482.50
12-MAY-15        475       463.75
13-MAY-15        375       443.75
14-MAY-15        500       500.00
15-MAY-15        650       518.75
16-MAY-15        550       537.50
17-MAY-15        450       537.50
18-MAY-15        500       518.75
19-MAY-15        575       493.75
20-MAY-15        450       506.25
21-MAY-15        500       525.00
22-MAY-15        575       593.75
23-MAY-15        850       606.25
24-MAY-15        500       625.00
25-MAY-15        575       606.25
26-MAY-15        500       537.50
27-MAY-15        575       537.50
28-MAY-15        500       537.50
29-MAY-15        575       556.25
30-MAY-15        575       556.25
31-MAY-15        575       575.00

31 rows selected.
```

3.3. Logical Windowing

The previous windowing clauses have used a certain number of rows. We can also have Oracle calculate which rows fit into our window groups. This is often done with date values where we might want the average of the previous week's sales. In that case the range would be

```
Range between Interval '7' day preceding and current row
```

This syntax requires a datetime field (year, month, day, hour, minute, second)

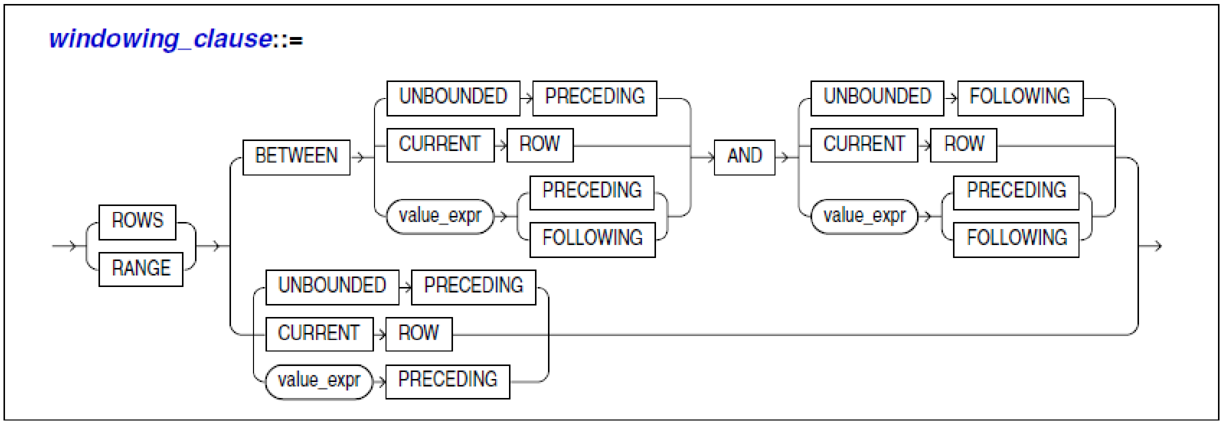
Demo 18: We want a total of the quantity ordered for this month and the previous month. This uses the oe tables

```
with CTE as (
  select order_date, sum(quantity_ordered) As AMount
  from oe_orderheaders
  join oe_orderdetails using (order_id)
  where extract(year from order_date)= 2016
  group by order_date
  )
  select order_date, amount
  , sum(amount) over (
     order by order_date range between
```

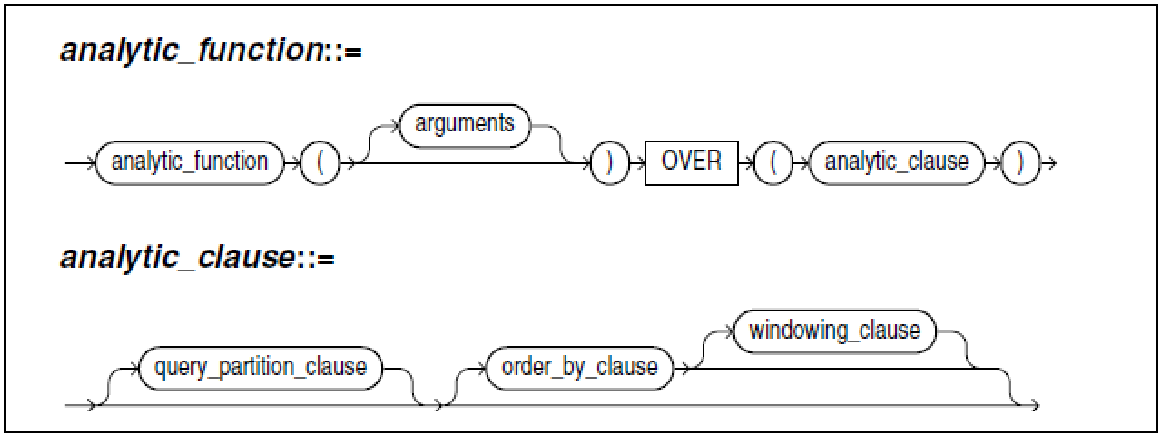
```
                       interval '1' month preceding and current row
     ) as MonthSum
  from CTE
  ;
ORDER_DATE   AMOUNT MONTHSUM
---------- -------- --------
02-JAN-16        10       10
03-JAN-16        21       31
04-JAN-16         1       32
05-JAN-16        29       61
07-JAN-16         1       62
11-JAN-16         1       63
12-JAN-16         4       67
15-JAN-16        10       77
23-JAN-16         5       82
26-JAN-16         3       85
27-JAN-16         7       92
31-JAN-16        24      116
01-FEB-16        16      132
02-FEB-16         2      134
03-FEB-16         6      130
12-FEB-16         7       84
01-MAR-16        53       84
05-MAR-16        54      114
07-MAR-16        20      134
08-MAR-16        14      148
09-MAR-16         5      153
04-APR-16        12      105
05-APR-16         4      109
06-APR-16         1       56
07-APR-16         3       59
08-APR-16         4       43
01-MAY-16        86      110
09-MAY-16         4       90
12-MAY-16        15      105

 29 rows selected
```

Again the first few rows of data may not be as valuable. since we do not have a full month of prior data. This is more practical and meaningful if we have a lot of data.

The windowing clause can be quite flexible; this is the model for that clause from the Oracle documentation. Windows can be based on a number of rows on a logical interval (often based on time).



The analytic functions can occur only in the Select list or the Order By clause. Other parts of the query (join, Where, Group by, Having) are carried out before the analytic functions.


analytic_function::=
analytic_function
(
arguments
)
OVER
(
analytic_clause
)
analytic_clause::=
query_partition_clause
order_by_clause
windowing_clause