Оконные функции

Что позволяет делать оконная функция?

Оконные (аналитические) функции позволяют делать агрегацию в запросе не накладывая ограничения, которые есть при обычной группировке. Такой подход позволяет сильно сократить запрос.

Помимо этого аналитические функции позволяют использовать в вычислениях соседние строки, что дает возможность решать более сложные аналитические задачи.

Давайте рассмотрим пример решения задачи с помощью обычной группировки и с использованием аналитических функций.

Задача

Сформируйте поле **delta** которое содержит разницу между максимальной зарплатой в отделе и зарплатой сотрудника.

```
/*
Без аналитических функций
*/
select
    t1.FIRST_NAME,
    t1.LAST_NAME,
    t2.max_salary - t1.salary as delta
from hr.employees t1
inner join (
    select
        department_id,
        max(salary) as max_salary
    from hr.employees
    group by department_id
) t2
on t1.department_id = t2.department_id;




/*
С аналитической функцией
*/

select
    FIRST_NAME,
    LAST_NAME,
```

```
    max(salary) over(partition by department_id) - salary as delta
from hr.employees;
```

Как можно видеть, синтаксис во втором варианте значительно короче.

Давайте разберем строку с аналитической функцией

max(salary) over(partition by department_id)

over() - функция, позволяющая указать параметры расчета (окна).

partition by - поле группировки

Помимо указанных параметров функция **over** может принимать параметр **order by** который работает не совсем интуитивно понятно.

Разберем следующий запрос.

```
select
    CUSTOMER_ID,
    ORDER_DATE,
    sum(ORDER_TOTAL) over(partition by CUSTOMER_ID
                          order by ORDER_DATE) as sum_orders
from oe.orders;
```

Данный запрос позволит определить сумму текущего заказа и предыдущих (предыдущий определяется по порядку значений в поле **ORDER_DATE**)

Задание

У нас есть таблица проводок по счетам и мы хотим получить баланс после совершения каждой проводки.

Скрипт для создания таблицы источника

```
create table tBalance(
 id    number,
 account varchar2(20),
 value  number
```

```
);

insert into tBalance values (1,'01',100);
insert into tBalance values (2,'01',200);
insert into tBalance values (3,'01',-100);
insert into tBalance values (4,'01',200);
insert into tBalance values (5,'01',100);
insert into tBalance values (6,'01',-100);
insert into tBalance values (7,'01',100);

insert into tBalance values (8, '02',10);
insert into tBalance values (9, '02',20);
insert into tBalance values (10,'02',-10);
insert into tBalance values (11,'02',-20);
insert into tBalance values (12,'02',10);
insert into tBalance values (13,'02',-10);
insert into tBalance values (14,'02',10);
```

Решение

```
select
  t.*,
  sum(t.value) OVER (PARTITION BY t.account order by t.id) as total
from tBalance t
```

row_number() vs rank() vs dense_rank()

Данная тройка оконных функций позволяет нумеровать строки, однако делает это по разному.

Синткасис

```
select
    LOCATION_ID,
    row_number() over(order by LOCATION_ID) as row_number_column,
    rank() over(order by LOCATION_ID) as rank_column,
    dense_rank() over(order by LOCATION_ID) as dense_rank_column
from hr.departments;
```

Разбор

row_number()

1 value1
2 value1
3 value1
4 value1
5 value2
6 value2
7 value2
8 value2
9 value2

Данная функция просто нумерует записи никаким образом не опираясь на значения поля

rank()

1 value1
1 value1
1 value1
1 value1
5 value2
5 value2
5 value2
5 value2
5 value2

Данная функция нумерует значения группами (по совпадающим значениям). При этом учитывает кол-во прошлых элементов (продолжает нумерование не с 2, а с 5).

dense_rank()

1 value1
1 value1
1 value1
1 value1
2 value2
2 value2
2 value2
2 value2
2 value2

Данная функция нумерует значения группами (по совпадающим значениям). При этом в отличие от **rank** не учитывает кол-во прошлых элементов.

Для более глубокого понимания смысла данных функций давайте разберем пример.

Задача

Определите id департамента, который стоит на втором месте (в списке по убыванию) по кол-ву сотрудников.

```
select
    department_id
from (
    select
        row_number() over(order by count(*) desc) as num,
        department_id
    from hr.employees
    group by department_id
)t2
where num = 2
```

Отлично!

З**адача**

Однако как решить подобную задачу, если нам понадобится определить третье место в списке, а там расположились сразу два департамента?

Для решения этой задачи мы можем использовать **dense_rank.**

```
create view v_mp_1 as
    select
        department_id,
        count(*) as emp_count
    from hr.employees
    group by department_id;


select
    t1.DEPARTMENT_ID
from (
    select
```

```
        dense_rank() over(order by emp_count desc) as num,
        DEPARTMENT_ID
    from v_tmp_1
) t1
where num = 3;
```

Функции lag и lead

Как говорилось ранее оконные функции позволяют в расчетах использовать соседние строки. Функции lag и lead позволяют получить прошлую и последующую запись соответственно.

Давайте разберем пример.

Задача.

Определить разницу зарплаты сотрудника по отношению к зарплате предыдущего и последующего сотрудника по размеру зарплаты.

```
select
    employee_id,
    lag(salary) over(order by salary) as lag_salary,
    salary,
    lead(salary) over(order by salary) as lead_salary
from hr.employees;
```

Помимо основного аргумента функции lag и lead принимают два дополнительных.

1- поле рассчета

2- шаг сдвига (по умолчанию 1)

3- значение. применяемое в случае отсутствия следующего/предидущего

Ограничение окна

Еще одним достаточно мощным инструментом в оконных функциях являет **ROWS BETWEEN.** Он позволяет указать грани окна расчета.

Давайте рассмотрим его применение на практике.

Задача

Необходимо рассчитать поле coef, которое хранит среднее значение среди текущего значение заказа, предыдущего и следующего у клиента (ранжировать по дате).

```
select
    t1.*,
    avg(ORDER_TOTAL) over(partition by customer_id
                          order by ORDER_DATE
                                rows between 1 PRECEDING
                                         and 1 following) as coef
from oe.orders t1;
```