### Лабораторная работа № 6\_1 Оконные функции PostgreSQL

Оконная функция выполняет вычисления для набора строк, некоторым образом связанных с текущей строкой. Можно сравнить её с агрегатной функцией, но, в отличие от обычной агрегатной функции, при использовании оконной функции несколько строк не группируются в одну, а продолжают существовать отдельно. Внутри же, оконная функция, как и агрегатная, может обращаться не только к текущей строке результата запроса.

При обычном запросе все множество строк обрабатывается как бы единым «цельным куском», для которого считаются агрегаты. А при использовании оконных функций, запрос делится на части (окна) и уже для каждой из отдельных частей считаются свои агрегаты

Вот пример, показывающий, как сравнить зарплату каждого сотрудника со средней зарплатой его отдела:

SELECT deptno, empno, sal, avg(sal) OVER (PARTITION BY deptno)

FROM emp;

По empno (сумма не агрегируется)

SELECT deptno, empno, sal, avg(sal) OVER (PARTITION BY empno)

FROM emp;

(Пример без практического смысла – агрегирование по имени ename см. “SMITH”)

INSERT INTO emp VALUES (8369,'SMITH','CLERK',7902,to\_date('17-12-1980','dd-mm-yyyy'),10000,NULL,20);

SELECT ename, deptno, empno, sal, avg(sal) OVER (PARTITION BY ename)

FROM emp;

### (Фактически среднее вычисляет та же функция avg, которую мы знаем как агрегатную, но предложение OVER превращает её в оконную, так что она обрабатывает лишь заданный набор строк.)

Вызов оконной функции всегда содержит предложение OVER, следующее за названием и аргументами оконной функции. Это синтаксически отличает её от обычной или агрегатной функции. Предложение OVER определяет, как именно нужно разделить строки запроса для обработки оконной функцией. Предложение PARTITION BY, дополняющее OVER, указывает, что строки нужно разделить по группам или разделам, объединяя одинаковые значения выражений PARTITION BY. Оконная функция вычисляется по строкам, попадающим в один раздел с текущей строкой.

Вы можете также определять порядок, в котором строки будут обрабатываться оконными функциями, используя ORDER BY в OVER. (Порядок ORDER BY для окна может даже не совпадать с порядком, в котором выводятся строки.) функция rank выдаёт порядковый номер в разделе текущей строки для каждого уникального значения, по которому выполняет сортировку предложение ORDER BY.

Например:

SELECT deptno, empno, sal,

rank() OVER (PARTITION BY ename ORDER BY sal DESC)

FROM emp;

Есть ещё одно важное понятие, связанное с оконными функциями: для каждой строки существует набор строк в её разделе, называемый рамкой окна. По умолчанию, с указанием ORDER BY рамка состоит из всех строк от начала раздела до текущей строки и строк, равных текущей по значению выражения ORDER BY. Без ORDER BY рамка по умолчанию состоит из всех строк раздела. [[4]](https://postgrespro.ru/docs/postgrespro/9.5/tutorial-window#ftn.idp1363) Посмотрите на пример использования sum:

SELECT sal, sum(sal) OVER () FROM emp;

Так как в этом примере нет указания ORDER BY в предложении OVER, рамка окна содержит все строки раздела, а он, в свою очередь, без предложения PARTITION BY включает все строки таблицы; другими словами, сумма вычисляется по всей таблице и мы получаем один результат для каждой строки результата. Но если мы добавим ORDER BY, мы получим совсем другие результаты:

SELECT sal, sum(sal) OVER (ORDER BY sal) FROM emp;

Если вам нужно отфильтровать или сгруппировать строки после вычисления оконных функций, вы можете использовать вложенный запрос. Например:

SELECT deptno, empno, sal

FROM

(SELECT deptno, empno, sal,

rank() OVER (PARTITION BY deptno ORDER BY sal DESC, empno) AS pos

FROM emp

) AS ss

WHERE pos < 3;

Данный запрос покажет только те строки внутреннего запроса, у которых rank (порядковый номер) меньше 3.

Когда в запросе вычисляются несколько оконных функций для одинаково определённых окон, конечно можно написать для каждой из них отдельное предложение OVER, но при этом оно будет дублироваться, что неизбежно будет провоцировать ошибки. Поэтому лучше определение окна выделить в предложение WINDOW, а затем ссылаться на него в OVER. Например:

SELECT sum(sal) OVER w, avg(sal) OVER w

FROM emp

WINDOW w AS (PARTITION BY deptno ORDER BY sal DESC);

[Оконные функции (window functions)](https://www.postgresql.org/docs/current/tutorial-window.html) — это фича, позволяющая производить манипуляции между строками, возвращаемыми одним SQL-запросом. Похоже на GROUP BY, но, в отличие от него, строки не объединяются в одну. Есть задачи, в которых оконные функции исключительно удобны. Например, когда нужно показать некие значения (выручку, посещаемость) за месяц, и рядом с ними — насколько это больше или меньше по сравнению с прошлым месяцем.

Для начала, выведем числа от 1 до 3:

SELECT x FROM generate\_series(1,3) AS x;

Теперь перепишем запрос вот таким образом:

SELECT x, array\_agg(x) OVER ()  
   FROM generate\_series(1,3) AS x;

Каждый раз, когда вы видите синтаксис OVER (...), можете быть уверены — запрос использует оконные функции. В данном примере используется простой OVER () без указания чего-либо в скобочках. Поэтому функция array\_agg() видит все строки, возвращаемые запросом. Эти строки называются фреймом, а array\_agg() выступает в роли оконной функции.

Запрос имеет доступ только к тому, что было извлечено по WHERE условию:

SELECT x, array\_agg(x) OVER ()  
   FROM generate\_series(1,3) AS x  
   WHERE x > 1;  
Это правило действует всегда. Если нужно обратиться к чему-то, что не удовлетворяет WHERE-условию, необходимо использовать подзапросы.

Помимо array\_agg() можно использовать sum(), count(), и другие агрегаты:

SELECT x, count(x) OVER (), sum(x) OVER () FROM generate\_series(1,3) AS x;  
Кроме обычных агрегатов еще есть [специализированные оконные функции](https://www.postgresql.org/docs/current/functions-window.html). Некоторые из них будут рассмотрены далее.

OVER () с пустыми скобочками на самом деле эквивалентен:

SELECT x, array\_agg(x) OVER (  
     RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
   ) FROM generate\_series(1,3) AS x;

Чтобы несколько раз не писать OVER (...), можно воспользоваться таким синтаксисом:

SELECT x, count(x) OVER w, sum(x) OVER w  
   FROM generate\_series(1,3) AS x  
   WINDOW w AS (RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW);

В рассмотренных примерах фрейм всегда содержал все возвращаемые строки. Это можно исправить:

SELECT x, array\_agg(x) OVER w  
   FROM generate\_series(1,3) AS x  
   WINDOW w AS (  
     ORDER BY x  
     ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
   );  
  
 SELECT x, array\_agg(x) OVER w  
    FROM generate\_series(1,3) AS x  
    WINDOW w AS (  
      ORDER BY x  
      ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING  
    );

Часть про ORDER BY x не является обязательной. Ведь нам известно, что generate\_series() возвращает числа в порядке возрастания. Но в общем случае, при SELECT‘е из таблицы, таких гарантий нет.

Внимательные читатели могли заметить, что вместо синтаксиса RANGE здесь мы перешли на ROWS. Это сделано не случайно. Дело в том, что вызов оконных функций может происходить в разных режимах. Они так и называются, RANGE mode и ROWS mode. Существует также GROUPS mode. RANGE mode немного запутанный. В частности, он переопределяет смысл CURRENT ROW в зависимости от того, какая часть остального синтаксиса используется в запросе. Сделано так, по всей видимости, чтобы используемый по умолчанию RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW работал по принципу наименьшего удивления, и эту часть запроса можно было опускать. Читателям, желающим основательно во всем этом разобраться, стоит [обратиться к официальной документации](https://www.postgresql.org/docs/current/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS). Здесь же мы тактично обойдем стороной этот вопрос.

Рассмотрим еще примеры:

 SELECT x, array\_agg(x) OVER w  
   FROM generate\_series(1,3) AS x  
   WINDOW w AS (  
     ORDER BY x  
     ROWS BETWEEN CURRENT ROW AND CURRENT ROW  
   );  
  
SELECT x, array\_agg(x) OVER w  
    FROM generate\_series(1,3) AS x  
    WINDOW w AS (  
      ORDER BY x  
      ROWS BETWEEN 1 PRECEDING AND CURRENT ROW  
    );  
  
  SELECT x, array\_agg(x) OVER w  
    FROM generate\_series(1,3) AS x  
    WINDOW w AS (  
      ORDER BY x  
      ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING  
    );

Текущую строку не обязательно включать во фрейм:

SELECT x, sum(x) OVER w  
   FROM generate\_series(1,3) AS x  
   WINDOW w AS (  
     ORDER BY x  
     ROWS BETWEEN 1 PRECEDING AND CURRENT ROW EXCLUDE CURRENT ROW  
   );

Смотрите, как удобно. Можно просто получить соседнее значение из выборки. Далее мы рассмотрим еще более простой способ это сделать.

Еще есть партиции. Рассмотрим их на более интересных данных:

CREATE TABLE employ (  
     "department" TEXT,  
     "name" TEXT,  
     "salary" INT);  
  
INSERT INTO employ  
   SELECT 'dep\_' || chr(d), 'emp\_' || chr(d) || e, d\*10 + e\*1  
   FROM generate\_series(ascii('a'), ascii('c')) AS d,  
        generate\_series (1,3) AS e;  
  
SELECT \* FROM employ;  
  
 department |  name  | salary  
==============================  
 dep\_a      | emp\_a1 |    971  
 dep\_a      | emp\_a2 |    972  
 dep\_a      | emp\_a3 |    973  
 dep\_b      | emp\_b1 |    981  
 dep\_b      | emp\_b2 |    982  
 dep\_b      | emp\_b3 |    983  
 dep\_c      | emp\_c1 |    991  
 dep\_c      | emp\_c2 |    992  
 dep\_c      | emp\_c3 |    993  
  
SELECT name, salary, array\_agg(salary) OVER w  
   FROM employ  
   WINDOW w AS (PARTITION BY department);  
  
  name  | salary |   array\_agg  
=================================  
 emp\_a1 |    971 | {971,972,973}  
 emp\_a2 |    972 | {971,972,973}  
 emp\_a3 |    973 | {971,972,973}  
 emp\_b1 |    981 | {981,982,983}  
 emp\_b2 |    982 | {981,982,983}  
 emp\_b3 |    983 | {981,982,983}  
 emp\_c1 |    991 | {991,992,993}  
 emp\_c2 |    992 | {991,992,993}  
 emp\_c3 |    993 | {991,992,993}

О партиции можно думать, как о такой специальной штуке, ограничивающей фрейм. Здесь мы партицируем данные по отделам. Ни один из фреймов не выходит за границы своей партиции. В остальном все работает так же, как раньше:

SELECT name, salary, sum(salary) OVER w  
   FROM employ  
   WINDOW w AS (  
     PARTITION BY department  
     ORDER BY salary  
     ROWS BETWEEN 1 PRECEDING AND CURRENT ROW EXCLUDE CURRENT ROW  
   );  
  
  name  | salary | sum  
=======================  
 emp\_a1 |    971 |  
 emp\_a2 |    972 | 971  
 emp\_a3 |    973 | 972  
 emp\_b1 |    981 |  
 emp\_b2 |    982 | 981  
 emp\_b3 |    983 | 982  
 emp\_c1 |    991 |  
 emp\_c2 |    992 | 991  
 emp\_c3 |    993 | 992

Партицировать можно и по условию:

SELECT INT4(x > 3), x, array\_agg(x) OVER w  
   FROM generate\_series(1,6) AS x  
   WINDOW w AS (  
     PARTITION BY x > 3  
     ORDER BY x  
     *-- попробуйте убрать следующую строчку; объясните результат*  
     ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
   );  
  
 INT4 | x | array\_agg  
======================  
    0 | 1 | {1,2,3}  
    0 | 2 | {1,2,3}  
    0 | 3 | {1,2,3}  
    1 | 4 | {4,5,6}  
    1 | 5 | {4,5,6}  
    1 | 6 | {4,5,6}

Выше говорилось о существовании [специализированных оконных функций](https://www.postgresql.org/docs/current/functions-window.html). Самая простая из них — это row\_number():

SELECT row\_number() OVER w, name, salary, sum(salary) OVER w  
   FROM employ  
   WINDOW w AS (  
     PARTITION BY department  
     ORDER BY salary  
     ROWS BETWEEN 1 PRECEDING AND CURRENT ROW EXCLUDE CURRENT ROW  
   );  
  
 row\_number |  name  | salary | sum  
====================================  
          1 | emp\_a1 |    971 |  
          2 | emp\_a2 |    972 | 971  
          3 | emp\_a3 |    973 | 972  
          1 | emp\_b1 |    981 |  
          2 | emp\_b2 |    982 | 981  
          3 | emp\_b3 |    983 | 982  
          1 | emp\_c1 |    991 |  
          2 | emp\_c2 |    992 | 991  
          3 | emp\_c3 |    993 | 992

Функция возвращает номер строки в партиции. Нумерация начиная с единицы.

Еще две полезные функции — это lag() и lead():

SELECT name, salary, lag(salary, 1) OVER w, lead(salary, 1) OVER w  
   FROM employ  
   WINDOW w AS (  
     PARTITION BY department  
     ORDER BY salary  
     ROWS BETWEEN 1 PRECEDING AND CURRENT ROW EXCLUDE CURRENT ROW  
   );  
  
  name  | salary | lag | lead  
==============================  
 emp\_a1 |    971 |     |  972  
 emp\_a2 |    972 | 971 |  973  
 emp\_a3 |    973 | 972 |  
 emp\_b1 |    981 |     |  982  
 emp\_b2 |    982 | 981 |  983  
 emp\_b3 |    983 | 982 |  
 emp\_c1 |    991 |     |  992  
 emp\_c2 |    992 | 991 |  993  
 emp\_c3 |    993 | 992 |

Функции позволяют подсмотреть вперед или назад на заданное число строк в рамках партиции.

Также есть first\_value() и last\_value():

SELECT name,  
          salary,  
          first\_value(salary) OVER w,  
          last\_value(salary) OVER w  
   FROM employ  
   WINDOW w AS (  
     PARTITION BY department  
     ORDER BY salary  
     ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
   );  
  
  name  | salary | first\_value | last\_value  
============================================  
 emp\_a1 |    971 |         971 |        973  
 emp\_a2 |    972 |         971 |        973  
 emp\_a3 |    973 |         971 |        973  
 emp\_b1 |    981 |         981 |        983  
 emp\_b2 |    982 |         981 |        983  
 emp\_b3 |    983 |         981 |        983  
 emp\_c1 |    991 |         991 |        993  
 emp\_c2 |    992 |         991 |        993  
 emp\_c3 |    993 |         991 |        993

Заметьте, что условие BETWEEN ... было переписано. Дело в том, что lag() и lead() работают на уровне партиции, и им не важно, какое условие было указано в BETWEEN .... Но функции first\_value() и last\_value() работают с фреймом и учитывают эти условия.

Еще существует nth\_value(). Функция также работает с фреймом:

SELECT name, salary, nth\_value(salary, 2) OVER w  
   FROM employ  
   WINDOW w AS (  
     PARTITION BY department  
     ORDER BY salary  
     ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
   );  
  
  name  | salary | nth\_value  
=============================  
 emp\_a1 |    971 |       972  
 emp\_a2 |    972 |       972  
 emp\_a3 |    973 |       972  
 emp\_b1 |    981 |       982  
 emp\_b2 |    982 |       982  
 emp\_b3 |    983 |       982  
 emp\_c1 |    991 |       992  
 emp\_c2 |    992 |       992  
 emp\_c3 |    993 |       992

Довольно часто используется функция rank():

UPDATE employ SET salary = 970 WHERE salary < 980;  
  
SELECT name, salary, row\_number() OVER w, rank() OVER w  
   FROM employ  
   WINDOW w AS (PARTITION BY department ORDER BY salary DESC);  
  
  name  | salary | row\_number | rank  
=====================================  
 emp\_a1 |    970 |          1 |    1  
 emp\_a3 |    970 |          2 |    1  
 emp\_a2 |    970 |          3 |    1  
 emp\_b3 |    983 |          1 |    1  
 emp\_b2 |    982 |          2 |    2  
 emp\_b1 |    981 |          3 |    3  
 emp\_c3 |    993 |          1 |    1  
 emp\_c2 |    992 |          2 |    2  
 emp\_c1 |    991 |          3 |    3

Функция возвращает номер строки в соответствии с указанным порядком сортировки. Часть синтаксиса про ROWS BETWEEN ... здесь была опущена, поскольку она не влияла на результат.

Наконец, никто не говорил, что нельзя указать несколько OVER (...):

 SELECT name,  
           salary,  
           rank() OVER w1 AS company\_rank,  
           rank() OVER w2 AS department\_rank  
   FROM employ  
   WINDOW w1 AS (ORDER BY salary DESC),  
          w2 AS (PARTITION BY department ORDER BY salary DESC);  
  
  name  | salary | company\_rank | department\_rank  
==================================================  
 emp\_c3 |    993 |            1 |               1  
 emp\_c2 |    992 |            2 |               2  
 emp\_c1 |    991 |            3 |               3  
 emp\_b3 |    983 |            4 |               1  
 emp\_b2 |    982 |            5 |               2  
 emp\_b1 |    981 |            6 |               3  
 emp\_a1 |    970 |            7 |               1  
 emp\_a2 |    970 |            7 |               1  
 emp\_a3 |    970 |            7 |               1

Это далеко не все возможности оконных функций. Однако их должно хватать для очень многих практических задач.

В общем ближе к сути — кто не знает, в PostgreSQL как и в любых нормальных БД, есть поддержка оконных функций. Речь не о OS Windows если что. Оконные функции — это такой особый вид функций, расширяющих аналитическую часть СУБД. Они не уменьшают и не увеличивают количество строк, как GROUP BY или JOIN, а добавляют действительно новую функциональность, например:

- Посчитать сумму с накоплением (нарастающий итог)  
- Вывести список сотрудников с колонкой — процент отклонения ЗП сотрудника от средней в отделе. При этом в таблице нужны все сотрудники всех отделов  
- Посчитать общее кол-во записей без учета limit, offset  
- Получить номер строки выборки. А если показалось легко, то усложните до — получить номер строки сотрудника, среди его отдела с обратной сортировкой по имени. И все в одной таблице  
  
.

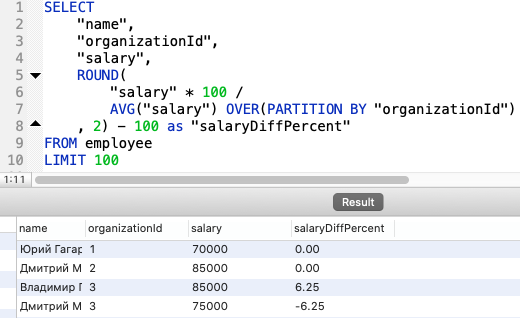
Оконные функции имеют свои дополнительные функции, а так же могут использоваться вместе с функциями агрегации.

## ОЗНАКОМИТЬСЯ

## Это оптимальнее подзапросов (как правило)

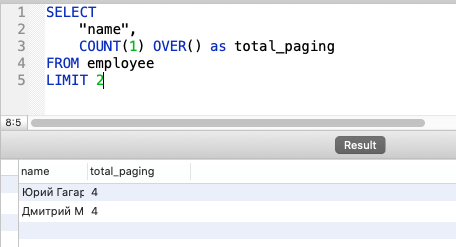
В SQL всегда можно разными путями решить задачу и этот не исключение

Тут считаем процентное отклонение зарплаты сотрудника относительно его отдела/компании (organizationId):

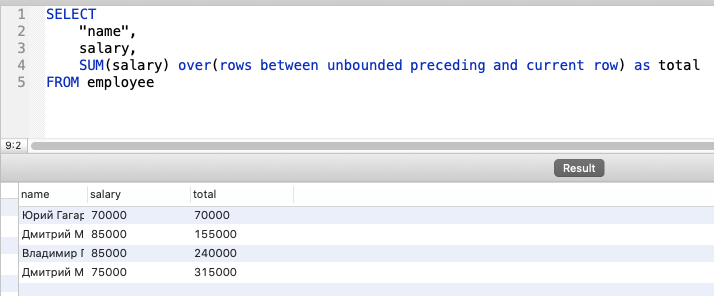


картинка моего запроса для примера

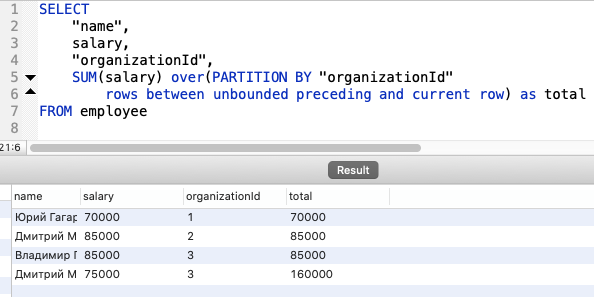
Вот простейший пример как получить данные с лимитом и сразу же общее количество строк в выборке без учета LIMIT / OFFSET. При это Postgres гарантирует, что сделает все это за один проход по данным.



## Нарастающий итог в SQL (сумма с накоплением)



А это тоже самое, но в рамках organizationId:



Получить разницу с минимальной зарплатой в отделе

