**Лабораторная работа № 6 АГРЕГИРОВАНИЕ ДАННЫХ**

Агрегатные функции вычисляют одно значение над некоторым набором строк. В PostgreSQL имеются следующие агрегатные функции:

* **AVG**: находит среднее значение. Входной параметр должен представлять один из следующих типов: smallint, int, bigint, real, double precision, numeric, interval. Для целочисленнных параметров результатом будет значение типа numeric, для параметров, которые представляют число с плавающей точкой, - значение типа double precision.
* **BIT\_AND**: выполняет операцию побитового умножения (операции логического И) для чисел следующих типов: smallint, int, bigint, bit. Если параметр содержит значение NULL, то возвращается NULL.
* **BIT\_OR**: выполняет операцию побитового сложения (операции логического ИЛИ) для чисел следующих типов: smallint, int, bigint, bit. Если параметр содержит значение NULL, то возвращается NULL.
* **BOOL\_AND**: выполняет операцию логического умножения для значений типа bool. Если входные все значения равны true, то возвращается true, иначе возвращается false.
* **BOOL\_OR**: выполняет операцию логического сложения для значений типа bool. Если входные хотя бы одно из значений равно true, то возвращается true, иначе возвращается false.
* **COUNT(\*)**: находит количество строк в запросе
* **COUNT(expression)**: находит количество строк в запросе, для которых expression не содержит значение NULL.
* **SUM**: находит сумму значений
* **MIN**: находит наименьшее значение
* **MAX**: находит наибольшее значение
* **STRING\_AGG(expression, delimiter)**: соединяет с помощью delimiter все текстовые значения из expression в одну строку.

В качестве параметра все агрегатные функции принимают выражение, которое представляет критерий для определения значений. Зачастую, в качестве выражения выступает название столбца, над значениями которого надо проводить вычисления.

Если в наборе нет строк, то все агрегатные функции за исключением COUNT(\*) возвращают значение NULL.

**Avg**

Функция **Avg** возвращает среднее значение на диапазоне значений столбца таблицы.

Пусть в базе данных у нас есть таблица товаров Products, которая описывается следующими выражениями:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | CREATE TABLE Products  (      Id SERIAL PRIMARY KEY,      ProductName VARCHAR(30) NOT NULL,      Company VARCHAR(20) NOT NULL,      ProductCount INT DEFAULT 0,      Price NUMERIC NOT NULL,      IsDiscounted BOOL  );    INSERT INTO Products (ProductName, Company, ProductCount, Price, IsDiscounted)  VALUES  ('iPhone X', 'Apple', 3, 76000, false),  ('iPhone 8', 'Apple', 2, 71000, true),  ('iPhone 7', 'Apple', 5, 42000, true),  ('Galaxy S9', 'Samsung', 2, 46000, false),  ('Galaxy S8 Plus', 'Samsung', 1, 56000, true),  ('Desire 12', 'HTC', 5, 28000, true),  ('Nokia 9', 'HMD Global', 6, 38000, true); |

Найдем среднюю цену товаров из базы данных:

|  |  |
| --- | --- |
| 1 | SELECT AVG(Price) AS Average\_Price FROM Products; |

Для поиска среднего значения в качестве выражения в функцию передается столбец Price. Для получаемого значения устанавливается псевдоним Average\_Price, хотя можно его и не устанавливать.

Также мы можем применить фильтрацию. Например, найти среднюю цену для товаров какого-то определенного производителя:

|  |  |
| --- | --- |
| 1  2 | SELECT AVG(Price) FROM Products  WHERE Company='Apple'; |

И, кроме того, мы можем находить среднее значение для более сложных выражений. Например, найдем среднюю сумму всех товаров, учитывая их количество:

|  |  |
| --- | --- |
| 1 | SELECT AVG(Price \* ProductCount) FROM Products |

**Count**

Функция **Count** вычисляет количество строк в выборке. Есть две формы этой функции. Первая форма COUNT(\*) подсчитывает число строк в выборке:

|  |  |
| --- | --- |
| 1 | SELECT COUNT(\*) FROM Products; |

Вторая форма функции вычисляет количество строк по определенному столбцу, при этом строки со значениями NULL игнорируются:

|  |  |
| --- | --- |
| 1 | SELECT COUNT(DISTINCT Company) FROM Products; |

Оператор **DISTINCT** указывает, что надо взять именно уникальные значения из столбца Company.

**Min и Max**

Функции **Min** и **Max** возвращают соответственно минимальное и максимальное значение по столбцу. Например, найдем минимальную цену среди товаров:

|  |  |
| --- | --- |
| 1 | SELECT MIN(Price) FROM Products; |

Поиск максимальной цены:

|  |  |
| --- | --- |
| 1 | SELECT MAX(Price) FROM Products; |

Данные функции также игнорируют значения NULL и не учитывают их при подсчете.

**Sum**

Функция **Sum** вычисляет сумму значений столбца. Например, подсчитаем общее количество товаров:

|  |  |
| --- | --- |
| 1 | SELECT SUM(ProductCount) FROM Products; |

Также вместо имени столбца может передаваться вычисляемое выражение. Например, найдем общую стоимость всех имеющихся товаров:

|  |  |
| --- | --- |
| 1 | SELECT SUM(ProductCount \* Price) FROM Products; |

**BOOL\_AND и BOOL\_OR**

Допустим, нам надо узнать, есть ли в таблице товары, которые подлежать скидке, то есть у которых IsDiscounted = true. В этом случае можно выполнить функцию BOOL\_OR, которая возвращает true, если хотя бы одно значение в наборе равно true:

|  |  |
| --- | --- |
| 1 | SELECT BOOL\_OR(IsDiscounted) FROM Products; |

Если нам надо узнать, все ли товары подлежат скидке, то можно применить функцию BOOL\_AND, которая возвращает true, если все значения в наборе равны true:

|  |  |
| --- | --- |
| 1 | SELECT BOOL\_AND(IsDiscounted) FROM Products; |

**STRING\_AGG**

Функция **STRING\_AGG()** объединяет набор строковых значений или значений bytea. Например, выберем названия всех товаров:

|  |  |
| --- | --- |
| 1 | SELECT STRING\_AGG(ProductName, ', ') FROM Products; |

Или выберем всех производителей:

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT STRING\_AGG(DISTINCT Company, ', ') FROM Products;    -- результат операции  -- Apple, HMD Global, HTC, Samsung |

Чтобы выбрать уникальных производителей, здесь также применяется оператор DISTINCT, иначе у нас бы повторялись значения.

**Комбинирование функций**

Объединим применение нескольких функций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT COUNT(\*) AS ProdCount,         SUM(ProductCount) AS TotalCount,         MIN(Price) AS MinPrice,         MAX(Price) AS MaxPrice,         AVG(Price) AS AvgPrice  FROM Products; |

Агрегатные функции получают единственный результат из набора входных значений.

CREATE AGGREGATE ***имя*** ( [ ***режим\_аргумента*** ] [ ***имя\_аргумента*** ] ***тип\_данных\_аргумента*** [ , ... ] ) (

SFUNC = ***функция\_состояния***,

STYPE = ***тип\_данных\_состояния***

[ , SSPACE = ***размер\_данных\_состояния*** ]

[ , FINALFUNC = ***функция\_завершения*** ]

[ , FINALFUNC\_EXTRA ]

[ , COMBINEFUNC = ***комбинирующая\_функция*** ]

[ , SERIALFUNC = ***функция\_сериализации*** ]

[ , DESERIALFUNC = ***функция\_десериализации*** ]

[ , INITCOND = ***начальное\_условие*** ]

[ , MSFUNC = ***функция\_состояния\_движ*** ]

[ , MINVFUNC = ***обратная\_функция\_движ*** ]

[ , MSTYPE = ***тип\_данных\_состояния\_движ*** ]

[ , MSSPACE = ***размер\_данных\_состояния\_движ*** ]

[ , MFINALFUNC = ***функция\_завершения\_движ*** ]

[ , MFINALFUNC\_EXTRA ]

[ , MINITCOND = ***начальное\_условие\_движ*** ]

[ , SORTOP = ***оператор\_сортировки*** ]

[ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]

)

CREATE AGGREGATE ***имя*** ( [ [ ***режим\_аргумента*** ] [ ***имя\_аргумента*** ] ***тип\_данных\_аргумента*** [ , ... ] ]

ORDER BY [ ***режим\_аргумента*** ] [ ***имя\_аргумента*** ] ***тип\_данных\_аргумента*** [ , ... ] ) (

SFUNC = ***функция\_состояния***,

STYPE = ***тип\_данных\_состояния***

[ , SSPACE = ***размер\_данных\_состояния*** ]

[ , FINALFUNC = ***функция\_завершения*** ]

[ , FINALFUNC\_EXTRA ]

[ , INITCOND = ***начальное\_условие*** ]

[ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]

[ , HYPOTHETICAL ]

)

или старый синтаксис

CREATE AGGREGATE ***имя*** (

BASETYPE = ***базовый\_тип***,

SFUNC = ***функция\_состояния***,

STYPE = ***тип\_данных\_состояния***

[ , SSPACE = ***размер\_данных\_состояния*** ]

[ , FINALFUNC = ***функция\_завершения*** ]

[ , FINALFUNC\_EXTRA ]

[ , COMBINEFUNC = ***комбинирующая\_функция*** ]

[ , SERIALFUNC = ***функция\_сериализации*** ]

[ , DESERIALFUNC = ***функция\_десериализации*** ]

[ , INITCOND = ***начальное\_условие*** ]

[ , MSFUNC = ***функция\_состояния\_движ*** ]

[ , MINVFUNC = ***обратная\_функция\_движ*** ]

[ , MSTYPE = ***тип\_данных\_состояния\_движ*** ]

[ , MSSPACE = ***размер\_данных\_состояния\_движ*** ]

[ , MFINALFUNC = ***функция\_завершения\_движ*** ]

[ , MFINALFUNC\_EXTRA ]

[ , MINITCOND = ***начальное\_условие\_движ*** ]

[ , SORTOP = ***оператор\_сортировки*** ]

)

**Встроенные агрегатные функции общего назначения**

| **Функция** | **Типы аргумента** | **Тип результата** | **Частичный режим** | **Описание** |
| --- | --- | --- | --- | --- |
| array\_agg(***выражение***) | любой тип не массива | массив элементов с типом аргумента | Нет | входные значения, включая NULL, объединяются в массив |
| array\_agg(***выражение***) | любой тип массива | тот же, что и тип аргумента | Нет | входные массивы собираются в массив большей размерности (они должны иметь одну размерность и не могут быть пустыми или равны NULL) |
| avg(***выражение***) | smallint, int, bigint, real, double precision, numeric или interval | numeric для любых целочисленных аргументов, double precision для аргументов с плавающей точкой, в противном случае тип данных аргумента | Да | арифметическое среднее для всех входных значений, отличных от NULL |
| bit\_and(***выражение***) | smallint, int, bigint или bit | тот же, что и тип аргумента | Да | побитовое И для всех входных значений, не равных NULL, или NULL, если таких нет |
| bit\_or(***выражение***) | smallint, int, bigint или bit | тот же, что и тип аргумента | Да | побитовое ИЛИ для всех входных значений, не равных NULL, или NULL, если таких нет |
| bool\_and(***выражение***) | bool | bool | Да | true, если все входные значения равны true, и false в противном случае |
| bool\_or(***выражение***) | bool | bool | Да | true, если хотя бы одно входное значение равно true, и false в противном случае |
| count(\*) |  | bigint | Да | количество входных строк |
| count(***выражение***) | any | bigint | Да | количество входных строк, для которых значение ***выражения*** не равно NULL |
| every(***выражение***) | bool | bool | Да | синоним bool\_and |
| json\_agg(***выражение***) | any | json | Нет | агрегирует значения, включая NULL, в виде массива JSON |
| jsonb\_agg(***выражение***) | any | jsonb | Нет | агрегирует значения, включая NULL, в виде массива JSON |
| json\_object\_agg(***имя***, ***значение***) | (any, any) | json | Нет | агрегирует пары имя/значение в виде объекта JSON (NULL допускается в значениях, но не в именах) |
| jsonb\_object\_agg(***имя***, ***значение***) | (any, any) | jsonb | Нет | агрегирует пары имя/значение в виде объекта JSON (NULL допускается в значениях, но не в именах) |
| max(***выражение***) | любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов | тот же, что и тип аргумента | Да | максимальное значение ***выражения*** среди всех входных данных, отличных от NULL |
| min(***выражение***) | любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов | тот же, что и тип аргумента | Да | минимальное значение ***выражения*** среди всех входных данных, отличных от NULL |
| string\_agg(***выражение***, ***разделитель***) | (text, text) или (bytea, bytea) | тот же, что и типы аргументов | Нет | входные данные (исключая NULL) складываются в строку через заданный разделитель |
| sum(***выражение***) | smallint, int, bigint, real, double precision, numeric, interval или money | bigint для аргументов smallint или int, numeric для аргументов bigint, и тип аргумента в остальных случаях | Да | сумма значений ***выражения*** по всем входным данным, отличным от NULL |
| xmlagg(***выражение***) | xml | xml | Нет | соединение XML-значений, отличных от NULL (см. также |

**Агрегатные функции для статистических вычислений**

| **Функция** | **Тип аргумента** | **Тип результата** | **Частичный режим** | **Описание** |
| --- | --- | --- | --- | --- |
| corr(***Y***, ***X***) | double precision | double precision | Да | коэффициент корреляции |
| covar\_pop(***Y***, ***X***) | double precision | double precision | Да | ковариация совокупности |
| covar\_samp(***Y***, ***X***) | double precision | double precision | Да | ковариация выборки |
| regr\_avgx(***Y***, ***X***) | double precision | double precision | Да | среднее независимой переменной (sum(***X***)/***N***) |
| regr\_avgy(***Y***, ***X***) | double precision | double precision | Да | среднее зависимой переменной (sum(***Y***)/***N***) |
| regr\_count(***Y***, ***X***) | double precision | bigint | Да | число входных строк, в которых оба выражения не NULL |
| regr\_intercept(***Y***, ***X***) | double precision | double precision | Да | пересечение с осью OY линии, полученной методом наименьших квадратов по данным (***X***, ***Y***) |
| regr\_r2(***Y***, ***X***) | double precision | double precision | Да | квадрат коэффициента корреляции |
| regr\_slope(***Y***, ***X***) | double precision | double precision | Да | наклон линии, полученной методом наименьших квадратов по данным (***X***, ***Y***) |
| regr\_sxx(***Y***, ***X***) | double precision | double precision | Да | sum(***X***^2) - sum(***X***)^2/***N*** («сумма квадратов» независимой переменной) |
| regr\_sxy(***Y***, ***X***) | double precision | double precision | Да | sum(***X***\****Y***) - sum(***X***) \* sum(***Y***)/***N*** («сумма произведений» независимых и зависимых переменных) |
| regr\_syy(***Y***, ***X***) | double precision | double precision | Да | sum(***Y***^2) - sum(***Y***)^2/***N*** («сумма квадратов» зависимой переменной) |
| stddev(***выражение***) | smallint, int, bigint, real, double precision или numeric | double precision для аргументов с плавающей точкой, numeric для остальных | Да | сохранившийся синоним stddev\_samp |
| stddev\_pop(***выражение***) | smallint, int, bigint, real, double precision или numeric | double precision для аргументов с плавающей точкой, numeric для остальных | Да | стандартное отклонение по генеральной совокупности входных значений |
| stddev\_samp(***выражение***) | smallint, int, bigint, real, double precision или numeric | double precision для аргументов с плавающей точкой, numeric для остальных | Да | стандартное отклонение по выборке входных значений |
| variance(***выражение***) | smallint, int, bigint, real, double precision или numeric | double precision для аргументов с плавающей точкой, numeric для остальных | Да | сохранившийся синоним var\_samp |
| var\_pop(***выражение***) | smallint, int, bigint, real, double precision или numeric | double precision для аргументов с плавающей точкой, numeric для остальных | Да | дисперсия для генеральной совокупности входных значений (квадрат стандартного отклонения) |
| var\_samp(***выражение***) | smallint, int, bigint, real, double precision или numeric | double precision для аргументов с плавающей точкой, numeric для остальных | Да | дисперсия по выборке для входных значений (квадрат отклонения по выборке) |

**Сортирующие агрегатные функции**

| **Функция** | **Тип непосредственного аргумента** | **Тип агрегированного аргумента** | **Тип результата** | **Частичный режим** | **Описание** |
| --- | --- | --- | --- | --- | --- |
| mode() WITHIN GROUP (ORDER BY ***выражение\_сортировки***) |  | любой сортируемый тип | тот же, что у выражения сортировки | Нет | возвращает значение, наиболее часто встречающееся во входных данных (если одинаково часто встречаются несколько значений, произвольно выбирается первое из них) |
| percentile\_cont(***дробь***) WITHIN GROUP (ORDER BY ***выражение\_сортировки***) | double precision | double precision или interval | тот же, что у выражения сортировки | Нет | непрерывный процентиль: возвращает значение, соответствующее заданной дроби по порядку, интерполируя соседние входные значения, если необходимо |
| percentile\_cont(***дроби***) WITHIN GROUP (ORDER BY ***выражение\_сортировки***) | double precision[] | double precision или interval | массив типа выражения сортировки | Нет | множественный непрерывный процентиль: возвращает массив результатов, соответствующих форме параметра ***дроби*** (для каждого элемента не NULL подставляется значение, соответствующее данному процентилю) |
| percentile\_disc(***дробь***) WITHIN GROUP (ORDER BY ***выражение\_сортировки***) | double precision | любой сортируемый тип | тот же, что у выражения сортировки | Нет | дискретный процентиль: возвращает первое значение из входных данных, позиция которого по порядку равна или превосходит указанную дробь |
| percentile\_disc(***дроби***) WITHIN GROUP (ORDER BY ***выражение\_сортировки***) | double precision[] | любой сортируемый тип | массив типа выражения сортировки | Нет | множественный дискретный процентиль: возвращает массив результатов, соответствующих форме параметра ***дроби*** (для каждого элемента не NULL подставляется входное значение, соответствующее данному процентилю) |

### Создание типа PostgreSQL

**CREATE TYPE average\_state AS (  
  accum numeric,  
  qty numeric  
);**

Функция перехода

**CREATE OR REPLACE FUNCTION average\_transition(  
  state average\_state,  
  val numeric  
) RETURNS average\_state AS $$  
BEGIN  
  RAISE NOTICE '%(%) + %', state.accum, state.qty, val;  
  RETURN ROW(state.accum+val, state.qty+1)::average\_state;  
END;  
$$ LANGUAGE plpgsql;**

Функция завершения

**CREATE OR REPLACE FUNCTION average\_final(  
  state average\_state  
) RETURNS numeric AS $$  
BEGIN  
  RAISE NOTICE '= %(%)', state.accum, state.qty;  
  RETURN CASE WHEN state.qty > 0 THEN  
    trim(trailing '0' from (state.accum/state.qty)::text)::numeric  
  END;  
END;  
$$ LANGUAGE plpgsql;**

Агрегирующая функция

**CREATE AGGREGATE average(numeric) (  
  sfunc     = average\_transition,  
  stype     = average\_state,  
  finalfunc = average\_final,  
  initcond  = '(0,0)'  
);**

Оконные функции: OVER()

Хорошая новость: написанная нами агрегатная функция может без всяких изменений работать и как оконная (аналитическая).  
  
Оконная функция отличается от агрегатной тем, что не сворачивает выборку в одну (агрегированную) строку, а вычисляется как бы отдельно для каждой строки. Синтаксически вызов оконной функции отличается наличием конструкции OVER с указанием *рамки,* которая определяет множество строк для обработки. В простейшем случае она так и записывается: OVER(), и это означает, что функция должна обработать все строки. Результат получается такой, как будто мы посчитали обычную агрегатную функцию и записали результат (один и тот же) напротив каждой строки выборки.  
  
Иными словами, рамка статична и охватывает все строки:

1. 2. 3. 4. 5.

+---+ +---+ +---+ +---+ +---+

| 1 | | 1 | | 1 | | 1 | | 1 |

| 2 | | 2 | | 2 | | 2 | | 2 |

| 3 | | 3 | | 3 | | 3 | | 3 |

| 4 | | 4 | | 4 | | 4 | | 4 |

| 5 | | 5 | | 5 | | 5 | | 5 |

+---+ +---+ +---+ +---+ +---+

PostgreSQL

Попробуем:  
  
**SELECT g.x, average(g.x) OVER ()  
FROM generate\_series(1,5) as g(x);**  
NOTICE:  0(0) + 1  
NOTICE:  1(1) + 2  
NOTICE:  3(2) + 3  
NOTICE:  6(3) + 4  
NOTICE:  10(4) + 5  
NOTICE:  = 15(5)  
 x | average  
---+---------  
 1 |       3  
 2 |       3  
 3 |       3  
 4 |       3  
 5 |       3  
(5 rows)  
  
По выводу NOTICE видно, что все происходит точно так же, как и ранее при вычислении обычной агрегатной функции. Получив результат от функции average\_final, PostgreSQL проставляет его в каждой строке.

Oracle

**SELECT average(level) OVER() average  
FROM dual CONNECT BY level <= 5;**  
     LEVEL     AVERAGE  
---------- -----------  
         1           3  
         2           3  
         3           3  
         4           3  
         5           3  
0(0) + 1  
1(1) + 2  
3(2) + 3  
6(3) + 4  
10(4) + 5  
= 15(5) flags:1  
= 15(5) flags:1  
= 15(5) flags:1  
= 15(5) flags:1  
= 15(5) flags:1  
= 15(5) flags:0  
  
Неожиданно. Вместо того, чтобы вычислить результат один раз, Oracle вызывает функцию ODCIAggregateTerminate N+1 раз: сначала для каждой строки с флагом 1 (что означает, что контекст еще пригодится) и затем еще один раз в конце. Значение, полученное при последнем вызове, просто игнорируется.  
  
Вывод такой: если в функции ODCIAggregateTerminate используется вычислительно сложная логика, надо подумать о том, чтобы не делать одну и ту же работу несколько раз.

Оконные функции: OVER(PARTITION BY)

Предложение PARTITION BY в определении рамки похоже на обычную агрегатную конструкцию GROUP BY. Оконная функция с указанием PARTITION BY вычисляется отдельно для каждой группы строк, и результат приписывается к каждой строке выборки.  
  
В таком варианте рамка тоже статична, но для каждой группы она разная. Например, если определены две группы строк (с первой по вторую и с третьей по пятую), то рамку можно представить себе так:

1. 2. 3. 4. 5.

+---+ +---+

| 1 | | 1 |

| 2 | | 2 | +---+ +---+ +---+

+---+ +---+ | 3 | | 3 | | 3 |

| 4 | | 4 | | 4 |

| 5 | | 5 | | 5 |

+---+ +---+ +---+

PostgreSQL

**SELECT g.x/3 part,  
       g.x,  
       average(g.x) OVER (PARTITION BY g.x/3)  
FROM generate\_series(1,5) as g(x);**  
NOTICE:  0(0) + 1  
NOTICE:  1(1) + 2  
NOTICE:  = 3(2)  
NOTICE:  0(0) + 3  
NOTICE:  3(1) + 4  
NOTICE:  7(2) + 5  
NOTICE:  = 12(3)  
 part | x | average  
------+---+---------  
    0 | 1 |     1.5  
    0 | 2 |     1.5  
    1 | 3 |       4  
    1 | 4 |       4  
    1 | 5 |       4  
(5 rows)  
  
Вычисление снова происходит последовательно, но теперь при переходе к другой группе строк состояние сбрасывается в начальное значение (initcond).

Oracle

**SELECT trunc(level/3) part,  
       level,  
       average(level) OVER(PARTITION BY trunc(level/3)) average  
FROM dual CONNECT BY level <= 5;**  
      PART      LEVEL    AVERAGE  
---------- ---------- ----------  
         0          2        1.5  
         0          1        1.5  
         1          4          4  
         1          5          4  
         1          3          4  
0(0) + 2  
2(1) + 1  
= 3(2) flags:1  
= 3(2) flags:1  
0(0) + 4  
4(1) + 5  
9(2) + 3  
= 12(3) flags:1  
= 12(3) flags:1  
= 12(3) flags:1  
= 12(3) flags:0  
  
Занятно, что Oracle решил переставить строки местами. Это может что-то сказать о деталях реализации, но в любом случае — имеет право.

Оконные функции: OVER(ORDER BY)

Если в определение рамки добавить предложение ORDER BY, указывающее порядок сортировки, функция начнет работать в режиме *нарастания* (для функции sum мы бы так и сказали — *нарастающим итогом*).  
  
Для первой строки рамка будет состоять из одной этой строки; для второй — из первой и второй; для третьей — из первой, второй и третьей и так далее. Иными словами, в рамку будут входить строки с первой до текущей.  
  
На самом деле, это можно ровно так и записать: OVER(ORDER BY… ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW), но, поскольку это многословие подразумеваются по умолчанию, его обычно опускают.  
  
Итак, рамка перестает быть статичной: ее голова движется вниз, а хвост остается на месте:

1. 2. 3. 4. 5.

+---+ +---+ +---+ +---+ +---+

| 1 | | 1 | | 1 | | 1 | | 1 |

+---+ | 2 | | 2 | | 2 | | 2 |

+---+ | 3 | | 3 | | 3 |

+---+ | 4 | | 4 |

+---+ | 5 |

+---+

PostgreSQL

**SELECT g.x, average(g.x) OVER (ORDER BY g.x)  
FROM generate\_series(1,5) as g(x);**  
NOTICE:  0(0) + 1  
NOTICE:  = 1(1)  
NOTICE:  1(1) + 2  
NOTICE:  = 3(2)  
NOTICE:  3(2) + 3  
NOTICE:  = 6(3)  
NOTICE:  6(3) + 4  
NOTICE:  = 10(4)  
NOTICE:  10(4) + 5  
NOTICE:  = 15(5)  
 x | average  
---+---------  
 1 |       1  
 2 |     1.5  
 3 |       2  
 4 |     2.5  
 5 |       3  
(5 rows)  
  
Как видим, строки все так же добавляются к контексту по одной, но теперь функция average\_final вызывается после каждого добавления, выдавая промежуточный итог.

Oracle

**SELECT level, average(level) OVER(ORDER BY level) average  
FROM dual CONNECT BY level <= 5;**  
     LEVEL    AVERAGE  
---------- ----------  
         1          1  
         2        1.5  
         3          2  
         4        2.5  
         5          3  
0(0) + 1  
= 1(1) flags:1  
1(1) + 2  
= 3(2) flags:1  
3(2) + 3  
= 6(3) flags:1  
6(3) + 4  
= 10(4) flags:1  
10(4) + 5  
= 15(5) flags:1  
= 15(5) flags:0  
  
На этот раз обе системы работают одинаково.

Оконные функции: OVER(PARTITION BY ORDER BY)

Предложения PARTITION BY и ORDER BY можно комбинировать. Тогда внутри каждой группы строк функция будет работать в режиме нарастания, а при переходе от группы к группе состояние будет сбрасываться в начальное.

1. 2. 3. 4. 5.

+---+ +---+

| 1 | | 1 |

+---+ | 2 | +---+ +---+ +---+

+---+ | 3 | | 3 | | 3 |

+---+ | 4 | | 4 |

+---+ | 5 |

+---+

PostgreSQL

**SELECT g.x/3 part,  
       g.x,  
       average(g.x) OVER (PARTITION BY g.x/3 ORDER BY g.x)  
FROM generate\_series(1,5) as g(x);**  
NOTICE:  0(0) + 1  
NOTICE:  = 1(1)  
NOTICE:  1(1) + 2  
NOTICE:  = 3(2)  
NOTICE:  0(0) + 3  
NOTICE:  = 3(1)  
NOTICE:  3(1) + 4  
NOTICE:  = 7(2)  
NOTICE:  7(2) + 5  
NOTICE:  = 12(3)  
 part | x | average  
------+---+---------  
    0 | 1 |       1  
    0 | 2 |     1.5  
    1 | 3 |       3  
    1 | 4 |     3.5  
    1 | 5 |       4  
(5 rows)

Oracle

**SELECT trunc(level/3) part,  
       level,  
       average(level) OVER(PARTITION BY trunc(level/3) ORDER BY level) average  
FROM dual CONNECT BY level <= 5;**  
      PART    LEVEL     AVERAGE  
---------- ---------- ----------  
     0        1           1  
     0        2         1.5  
     1        3           3  
     1        4         3.5  
     1        5           4  
0(0) + 1  
= 1(1) flags:1  
1(1) + 2  
= 3(2) flags:1  
0(0) + 3  
= 3(1) flags:1  
3(1) + 4  
= 7(2) flags:1  
7(2) + 5  
= 12(3) flags:1  
= 12(3) flags:0

Оконные функции со скользящей рамкой

Во всех примерах, которые мы посмотрели, рамка либо была статической, либо двигалась только ее голова (при использовании предложения ORDER BY). Это давало нам возможность вычислять состояние последовательно, добавляя к контексту строку за строкой.  
  
Но рамку оконной функции можно задать и таким образом, что ее хвост тоже будет смещаться. В нашем примере это будет соответствовать понятию *скользящего среднего.* Например, указание OVER(ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) говорит о том, что для каждой строки результата будут усредняться текущее и два предыдущих значений.

1. 2. 3. 4. 5.

+---+

| | +---+

| | | | +---+

| 1 | | 1 | | 1 | +---+

+---+ | 2 | | 2 | | 2 | +---+

+---+ | 3 | | 3 | | 3 |

+---+ | 4 | | 4 |

+---+ | 5 |

+---+

Сможет ли вычисляться оконная функция в таком случае? Оказывается, сможет, правда неэффективно. Но, написав еще немного кода, можно улучшить ситуацию.

PostgreSQL

Посмотрим:  
  
**SELECT g.x,  
       average(g.x) OVER (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)  
FROM generate\_series(1,5) as g(x);**  
NOTICE:  0(0) + 1  
NOTICE:  = 1(1)  
NOTICE:  1(1) + 2  
NOTICE:  = 3(2)  
NOTICE:  3(2) + 3  
NOTICE:  = 6(3)  
NOTICE:  0(0) + 2  
NOTICE:  2(1) + 3  
NOTICE:  5(2) + 4  
NOTICE:  = 9(3)  
NOTICE:  0(0) + 3  
NOTICE:  3(1) + 4  
NOTICE:  7(2) + 5  
NOTICE:  = 12(3)  
 x | average  
---+---------  
 1 |       1  
 2 |     1.5  
 3 |       2  
 4 |       3  
 5 |       4  
(5 rows)  
  
Вплоть до третьей строки все идет хорошо, потому что хвост фактически не двигается: мы просто добавляем к уже имеющемуся контексту очередное значение. Но, поскольку мы не умеем убирать значение из контекста, для четвертой и пятой строк все приходится пересчитывать полностью, каждый раз возвращаясь к начальному состоянию.  
  
Итак, было бы здорово иметь не только функцию *добавления* очередного значения, но и функцию *удаления* значения из состояния. И действительно, такую функцию можно создать:  
  
**CREATE OR REPLACE FUNCTION average\_inverse(state average\_state, val numeric)  
RETURNS average\_state AS $$  
BEGIN  
  RAISE NOTICE '%(%) - %', state.accum, state.qty, val;  
  RETURN ROW(state.accum-val, state.qty-1)::average\_state;  
END;  
$$ LANGUAGE plpgsql;**  
  
Чтобы оконная функция смогла ей воспользоваться, нужно пересоздать агрегат следующим образом:  
  
**DROP AGGREGATE average(numeric);  
CREATE AGGREGATE average(numeric) (  
  -- обычный вариант  
  sfunc      = average\_transition,  
  stype      = average\_state,  
  finalfunc  = average\_final,  
  initcond   = '(0,0)',  
  -- вариант с “обратной” функцией  
  msfunc     = average\_transition,  
  minvfunc   = average\_inverse,  
  mstype     = average\_state,  
  mfinalfunc = average\_final,  
  minitcond  = '(0,0)'  
);**  
Проверим:  
  
**SELECT g.x,  
       average(g.x) OVER (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)  
FROM generate\_series(1,5) as g(x);**  
NOTICE:  0(0) + 1  
NOTICE:  = 1(1)  
NOTICE:  1(1) + 2  
NOTICE:  = 3(2)  
NOTICE:  3(2) + 3  
NOTICE:  = 6(3)  
NOTICE:  6(3) - 1  
NOTICE:  5(2) + 4  
NOTICE:  = 9(3)  
NOTICE:  9(3) - 2  
NOTICE:  7(2) + 5  
NOTICE:  = 12(3)  
 x | average  
---+---------  
 1 |       1  
 2 |     1.5  
 3 |       2  
 4 |       3  
 5 |       4  
(5 rows)  
  
Вот теперь все в порядке: для четвертой и пятой строк мы удаляем из состояния хвостовое значение и добавляем новое.

Oracle

Тут ситуация аналогична. Созданный вариант аналитической функции работает, но неэффективно:  
  
**SELECT level,  
       average(level) OVER(ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) average  
FROM dual CONNECT BY level <= 5;**  
     LEVEL    AVERAGE  
---------- ----------  
         1          1  
         2        1.5  
         3          2  
         4          3  
         5          4  
0(0) + 1  
= 1(1) flags:1  
1(1) + 2  
= 3(2) flags:1  
3(2) + 3  
= 6(3) flags:1  
0(0) + 2  
2(1) + 3  
5(2) + 4  
= 9(3) flags:1  
0(0) + 3  
3(1) + 4  
7(2) + 5  
= 12(3) flags:1  
= 12(3) flags:0  
  
Функция удаления значения из контекста определяется следующим образом:  
  
**MEMBER FUNCTION ODCIAggregateDelete(self IN OUT AverageImpl, val IN number)  
RETURN number IS  
BEGIN  
  dbms\_output.put\_line(self.accum||'('||self.qty||') - '||val);  
  self.accum := self.accum - val;  
  self.qty := self.qty - 1;  
  RETURN ODCIConst.Success;  
END;**

Полный код для copy-paste

Пересоздавать саму функцию не нужно. Проверим:  
  
**SELECT level,  
       average(level) OVER(ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) average  
FROM dual CONNECT BY level <= 5;**  
     LEVEL    AVERAGE  
---------- ----------  
         1          1  
         2        1.5  
         3          2  
         4          3  
         5          4  
0(0) + 1  
= 1(1) flags:1  
1(1) + 2  
= 3(2) flags:1  
3(2) + 3  
= 6(3) flags:1  
6(3) - 1  
5(2) + 4  
= 9(3) flags:1  
9(3) - 2  
7(2) + 5  
= 12(3) flags:1  
= 12(3) flags:0

Параллельность

И PostgreSQL, и Oracle (Enterprise Edition) умеют вычислять агрегатные функции в параллельном режиме. При этом каждый из параллельных процессов выполняет свою часть работы, формируя промежуточное состояние. Затем основной процесс-координатор получает эти несколько состояний и должен объединить их в одно итоговое.  
  
Для этого нужна еще одна функция *объединения,* которую мы сейчас и напишем. В нашем случае она просто складывает и суммы, и количество значений.

PostgreSQL

Функция выглядит следующим образом:  
  
**CREATE OR REPLACE FUNCTION average\_combine(state1 average\_state, state2 average\_state)  
RETURNS average\_state AS $$  
BEGIN  
  RAISE NOTICE '%(%) & %(%)', state1.accum, state1.qty, state2.accum, state2.qty;  
  RETURN ROW(state1.accum+state2.accum, state1.qty+state2.qty)::average\_state;  
END;  
$$ LANGUAGE plpgsql;**  
Еще мы уберем наш отладочный вывод из функции average\_transition. При параллельном выполнении мы будем суммировать не пять значений, а больше, так что если этого не сделать, мы получим слишком много бесполезной информации.  
  
Поскольку мы убираем вывод, то отпадает и необходимость использовать процедурный язык — напишем функцию на чистом SQL:  
  
**CREATE OR REPLACE FUNCTION average\_transition(state average\_state, val numeric)  
RETURNS average\_state AS $$  
  SELECT ROW(state.accum+val, state.qty+1)::average\_state;  
$$ LANGUAGE sql;**  
Осталось пересоздать агрегат с учетом новой функции и указать, что его можно безопасно использовать в параллельном режиме:  
  
**DROP AGGREGATE average(numeric);  
CREATE AGGREGATE average(numeric) (  
  -- обычный вариант  
  sfunc       = average\_transition,  
  stype       = average\_state,  
  finalfunc   = average\_final,  
  combinefunc = average\_combine,  
  initcond    = '(0,0)',  
  -- вариант с “обратной” функцией  
  msfunc      = average\_transition,  
  minvfunc    = average\_inverse,  
  mstype      = average\_state,  
  mfinalfunc  = average\_final,  
  minitcond   = '(0,0)',  
  -- параллельность  
  parallel = safe  
);**  
Теперь создадим таблицу и заполним ее данными. Тысячи строк будет достаточно.  
  
**CREATE TABLE t(n) AS SELECT generate\_series(1,1000)::numeric;**  
С настройками по умолчанию PostgreSQL не построит параллельный план для такой таблицы — слишком она мала, — но его несложно уговорить:  
  
**SET parallel\_setup\_cost=0;  
SET min\_parallel\_table\_scan\_size=0;**  
**EXPLAIN(costs off) SELECT average(n) FROM t;**  
                QUERY PLAN                  
------------------------------------------  
 Finalize Aggregate  
   ->  Gather  
         Workers Planned: 2  
         ->  Partial Aggregate  
               ->  Parallel Seq Scan on t  
  
В плане запроса видим:

* два запланированных рабочих процесса, выполняющих частичное агрегирование (Partial Aggregate),
* узел Gather, собирающий информацию,
* и итоговое объединение состояний (Finalize Aggregate).

Проверим:  
  
**SELECT average(n) FROM t;**  
NOTICE:  0(0) & 281257(678)  
NOTICE:  281257(678) & 127803(226)  
NOTICE:  409060(904) & 91440(96)  
NOTICE:  = 500500(1000)  
 average  
---------  
   500.5  
(1 row)  
  
Почему функция average\_combine вызывается три раза, а не два? Дело в том, что в PostgreSQL координирующий процесс тоже выполняет часть работы. Поэтому, хотя было запущено два рабочих процесса, реально работа выполнялась в трех. Один из них успел обработать 678 строк, другой 226 и третий — 96 (хотя эти цифры ничего не значат и при другом запуске могут отличаться).

Oracle

Если помните, функцию ODCIAggregateMerge мы уже написали в самом начале, поскольку в Oracle она является обязательной. Документация настаивает, что эта функция необходима не только для параллельной работы, но и для последовательной — хотя мне трудно понять, зачем (и на практике не приходилось сталкиваться с ее выполнением при последовательной обработке).  
  
Все, что остается сделать — объявить функцию безопасной для параллельной работы:  
  
**CREATE OR REPLACE FUNCTION average(val number) RETURN number  
  PARALLEL\_ENABLE  
  AGGREGATE USING AverageImpl;  
/**  
Создаем таблицу:  
  
**CREATE TABLE t(n) AS SELECT to\_number(level) FROM dual CONNECT BY level <= 1000;**  
Уговорить Oracle еще проще, чем PostgreSQL — достаточно написать хинт. Вот какой получается план (вывод сильно урезан для простоты):  
  
**EXPLAIN PLAN FOR SELECT /\*+ PARALLEL(2) \*/ average(n) FROM t;  
SELECT \* FROM TABLE(dbms\_xplan.display);**  
---------------------------------  
| Id  | Operation               |  
---------------------------------  
|   0 | SELECT STATEMENT        |  
|   1 |  SORT AGGREGATE         |  
|   2 |   PX COORDINATOR        |  
|   3 |    PX SEND QC (RANDOM)  |  
|   4 |     SORT AGGREGATE      |  
|   5 |      PX BLOCK ITERATOR  |  
|   6 |       TABLE ACCESS FULL |  
---------------------------------  
  
План также содержит:

* частичную агрегацию (4),
* координатора, получающего частичные контексты (2),
* и итоговое объединение контекстов (1).

**SELECT /\*+ PARALLEL(2) \*/ average(n) FROM t;**  
AVERAGE(N)  
----------  
     500.5  
0(0) & 216153(657)  
216153(657) & 284347(343)  
= 500500(1000) flags:0  
  
В Oracle координатор не участвует в частичной агрегации. Поэтому объединяются только два контекста и по этой же причине мы видим только вывод функции ODCIAggregateMerge.

Пример про округление копеек

И обещанный пример из жизни. Эту функцию я придумал, когда приходилось писать отчеты для бухгалтерии, работающей по РСБУ (правилам российского бухучета).  
  
Самая простая задача, в которой возникает необходимость округления — распределение общих расходов (скажем, 100 рублей) на отделы (скажем, 3 штуки) по какому-то принципу (скажем, поровну):  
  
**WITH depts(name) AS (  
  VALUES ('A'), ('B'), ('C')  
), report(dept,amount) AS (  
  SELECT name, 100.00 / count(\*) OVER() FROM depts  
)  
SELECT dept, round(amount,2) FROM report;**  
 dept | round  
------+-------  
 A    | 33.33  
 B    | 33.33  
 C    | 33.33  
(3 rows)  
  
Этот запрос показывает проблему: суммы надо округлять, но при этом теряется копейка. А  РСБУ этого не прощает.  
  
Задачу можно решать по-разному, но на мой вкус наиболее элегантный способ — оконная функция, которая работает в нарастающем режиме и берет всю борьбу с копейками на себя:  
  
**WITH depts(name) AS (  
  VALUES ('A'), ('B'), ('C')  
), report(dept,amount) AS (  
  SELECT name, 100.00 / count(\*) OVER() FROM depts  
)  
SELECT dept, round2(amount) OVER (ORDER BY dept) FROM report;**  
 dept | round2  
------+--------  
 A    |  33.33  
 B    |  33.34  
 C    |  33.33  
(3 rows)  
  
Состояние такой функции включает ошибку округления (r\_error) и текущее округленное значение (amount). Функция обработки очередного значения увеличивает ошибку округления, и, если она уже превышает полкопейки, добавляет к округленной сумме копеечку:  
  
**state.r\_error := state.r\_error + val - round(val,2);  
state.amount := round(val,2) + round(state.r\_error,2);  
state.r\_error := state.r\_error - round(state.r\_error,2);**  
А функция, выдающая результат, просто возвращает уже готовый state.amount.  
  
Полный код функции приводить не буду: используя уже приведенные примеры написать ее не представляет сложности.  
  
Если вам встречались интересные примеры использования собственных агрегатных или оконных функций — поделитесь ими в комментариях.