

# Расширяемость Агрегатные и оконные функции



## Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

## Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

## Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## Отказ от ответственности

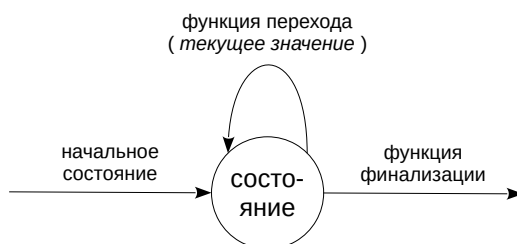
Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Создание пользовательских агрегатных функций  
Механизм работы оконных функций и их создание  
Параллельное выполнение агрегатных функций

## Построчная обработка агрегируемой выборки

состояние

функции перехода и финализации



В PostgreSQL имеется достаточно много встроенных агрегатных функций. Часть из них определена стандартом (такие, как `min`, `max`, `sum`, `count` и т. п.), часть является расширением стандарта.

<https://postgrespro.ru/docs/postgresql/16/functions-aggregate>

Но иногда может оказаться полезным создать собственную агрегатную функцию. Такая функция работает очень просто.

Имеется некоторое состояние, представленное значением какого-либо типа данных, которое инициализируется определенным значением (например, тип `numeric` и начальное значение 0).

Для каждой строки агрегируемой выборки вызывается функция перехода, которая получает значение из текущей строки и должна обновить состояние (например, прибавляет текущее значение к состоянию).

В конце вызывается функция финализации, которая преобразует полученное в результате работы состояние в результат агрегатной функции (в нашем примере достаточно просто вернуть число — в итоге получается аналог функции `sum`).

<https://postgrespro.ru/docs/postgresql/16/xaggr>

## Агрегатные функции

```
=> CREATE DATABASE ext_aggregates;
```

CREATE DATABASE

```
=> \c ext_aggregates
```

You are now connected to database "ext\_aggregates" as user "student".

Мы будем писать функцию для получения среднего, аналог встроенной функции avg.

Начнем с того, что создадим таблицу:

```
=> CREATE TABLE test (
    n float,
    grp text
);
```

CREATE TABLE

```
=> INSERT INTO test(n,grp)
VALUES (1,'A'), (2,'A'), (3,'B'), (4,'B'), (5,'B');
```

INSERT 0 5

Состояние должно включать сумму значений и их количество. Для его хранения создадим составной тип:

```
=> CREATE TYPE average_state AS (
    accum float,
    qty float
);
```

CREATE TYPE

Теперь определим функцию перехода. Она возвращает новое состояние на основе текущего, прибавляя текущее значение к сумме и единицу к количеству.

Мы также включим в функцию отладочный вывод, чтобы иметь возможность наблюдать за ее вызовом.

```
=> CREATE FUNCTION average_transition(
    state average_state,
    val float
)
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 IMMUTABLE;
```

CREATE FUNCTION

Функция финализации делит полученную сумму на количество, чтобы получить среднее:

```
=> CREATE FUNCTION average_final(
    state average_state
)
RETURNS float AS $$
BEGIN
    RAISE NOTICE '= %(%)', state.accum, state.qty;
    RETURN CASE
        WHEN state.qty > 0 THEN state.accum/state.qty
    END;
END;
$$ LANGUAGE plpgsql IMMUTABLE;
```

CREATE FUNCTION

И наконец нужно создать агрегат, указав тип состояния и его начальное значение, а также функции перехода и финализации:

```
=> CREATE AGGREGATE average(float) (
    stype = average_state,
    initcond = '(0,0)',
    sfunc = average_transition,
    finalfunc = average_final
);
```

CREATE AGGREGATE

Можно пробовать нашу агрегатную функцию в работе:

```
=> SELECT average(n) FROM test;
```

```
NOTICE:  0(0) + 1
NOTICE:  1(1) + 2
NOTICE:  3(2) + 3
NOTICE:  6(3) + 4
NOTICE: 10(4) + 5
NOTICE:  = 15(5)
average
-----
      3
(1 row)
```

Благодаря отладочному выводу хорошо видно, как изменяется начальное состояние (0,0).

Функция работает и при указании группировки GROUP BY:

```
=> SELECT grp, average(n) FROM test GROUP BY grp;
```

```
NOTICE:  0(0) + 1
NOTICE:  1(1) + 2
NOTICE:  0(0) + 3
NOTICE:  3(1) + 4
NOTICE:  7(2) + 5
NOTICE:  = 12(3)
NOTICE:  = 3(2)
grp | average
-----+-----
 B  |         4
 A  |        1.5
(2 rows)
```

Здесь видно, что используются два разных состояния — свое для каждой группы.

# Оконные функции

Окно определяет рамку (агрегируемую выборку) для каждой строки

OVER ( )

1 2 3 4 5  
▲

1 2 3 4 5  
▲

1 2 3 4 5  
▲

1 2 3 4 5  
▲

1 2 3 4 5  
▲

OVER (PARTITION BY)

1 2 3 4 5  
▲

1 2 3 4 5  
▲

1 2 3 4 5  
▲

1 2 3 4 5  
▲

1 2 3 4 5  
▲

Оконные функции работают подобно агрегатным, вычисляя значение на основе некоторой выборки. Эта выборка называется *рамкой*.

Но, в отличие от агрегатных функций, строки не сворачиваются в одну общую, а вместо этого значение вычисляется для каждой строки выборки.

Окно задается в предложении OVER после имени функции.

Если окно указано как OVER(), то оконная функция вычисляется на основе всех строк — одинаково для каждой строки выборки.

Если определение окна включает фразу PARTITION BY, то оконная функция вычисляется на основе групп строк (аналогично группировке GROUP BY). В этом случае для всех строк одной группы будет получено одинаковое значение функции.

## OVER()

Созданная нами агрегатная функция работает как оконная без всяких изменений:

```
=> SELECT n, average(n) OVER() FROM test;
```

```
NOTICE:  0(0) + 1
NOTICE:  1(1) + 2
NOTICE:  3(2) + 3
NOTICE:  6(3) + 4
NOTICE: 10(4) + 5
NOTICE: 15(5)
```

n	average
1	3
2	3
3	3
4	3
5	3

(5 rows)

Обратите внимание, что, поскольку рамка для всех строк одинакова, значение вычисляется только один раз, а не для каждой строки.

И для предложения PARTITION BY:

```
=> SELECT n, grp, average(n) OVER(PARTITION BY grp) FROM test;
```

```
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)
```

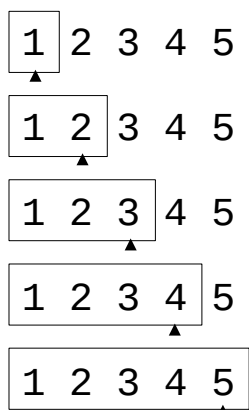
n	grp	average
1	A	1.5
2	A	1.5
3	B	4
4	B	4
5	B	4

(5 rows)

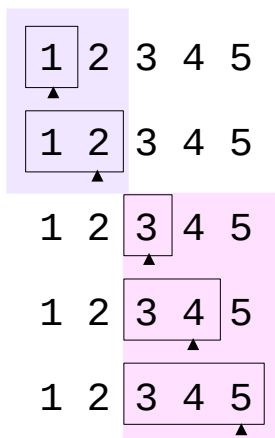
Здесь все работает точно так же, как для обычной группировки GROUP BY.

«Голова» рамки может двигаться

OVER (ORDER BY)



OVER (PARTITION BY ORDER BY)



Как только в определении окна мы указываем предложение ORDER BY, упорядочивающее строки выборки, предполагается, что рамка окна охватывает строки от самой первой до текущей. Это можно записать и явным образом: ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

Например, если в качестве оконной функции используется sum, мы получаем сумму «нарастающим итогом».

Конечно, для каждой группы строк, определяемых предложением PARTITION BY, рамка будет располагаться в пределах этой группы.

## OVER(ORDER BY)

Если добавить к определению окна предложение ORDER BY, получим рамку, «хвост» которой стоит на месте, а голова движется вместе с текущей строкой:

```
=> SELECT n, average(n) OVER(ORDER BY n) FROM test;
```

```
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)
```

n	average
1	1
2	1.5
3	2
4	2.5
5	3

(5 rows)

Снова не понадобилось никаких изменений — все работает. Здесь видно, как каждая следующая строка последовательно добавляется к состоянию и вызывается функция финализации.

Полная форма того же запроса выглядит так:

```
SELECT n, average(n) OVER(
    ORDER BY n                -- сортировка
    ROWS BETWEEN UNBOUNDED PRECEDING -- от самого начала
    AND CURRENT ROW           -- до текущей строки
)
FROM test;
```

То же самое работает и в сочетании с PARTITION BY:

```
=> SELECT n, grp, average(n) OVER(PARTITION BY grp ORDER BY n)
FROM test;
```

```
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)
```

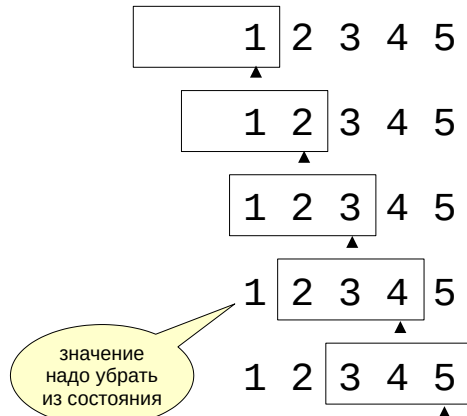
n	grp	average
1	A	1
2	A	1.5
3	B	3
4	B	3.5
5	B	4

(5 rows)

# Скользящая рамка

Могут двигаться «голова» и «хвост» рамки одновременно

OVER (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)



Рамку можно указать явно в предложении ROWS BETWEEN. В том числе двигаться может не только «голова» рамки, но и ее «хвост».

<https://postgrespro.ru/docs/postgresql/16/sql-expressions#SYNTAX-WINDOW-FUNCTIONS>

Если в предыдущих примерах рамка только расширялась (в нее добавлялись все новые значения), то теперь ранее добавленные значения могут «уходить» из рамки.

Чтобы пользовательская агрегатная функция работала эффективно в таком режиме, нужно реализовать функцию инверсии, которая устраняет значение из состояния.

## OVER(ROWS BETWEEN)

С помощью фразы ROWS BETWEEN можно задать любую необходимую конфигурацию рамки, указывая (в частности):

- UNBOUNDED PRECEDING — с самого начала;
- n PRECEDING — n предыдущих;
- CURRENT ROW — текущая строка;
- n FOLLOWING — n следующих;
- UNBOUNDED FOLLOWING — до самого конца.

Рассмотрим вычисление «скользящего среднего» для трех значений. В отличие от предыдущих примеров из состояния должно «вычитаться» значение, уходящее из рамки, но у нас есть только функция «добавления». Единственный способ выполнить запрос — пересчитывать всю рамку заново:

```
=> SELECT n, average(n) OVER(ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
FROM test;
```

```
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)
```

```
 n | average
---+-----
 1 |         1
 2 |        1.5
 3 |         2
 4 |         3
 5 |         4
(5 rows)
```

---

Это, конечно, неэффективно, но мы можем написать недостающую функцию «инверсии»:

```
=> CREATE FUNCTION average_inverse(
    state average_state,
    val float
) 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 IMMUTABLE;
```

```
CREATE FUNCTION
```

Нужно указать эту функцию в определении агрегата:

```
=> DROP AGGREGATE average(float);
```

```
DROP AGGREGATE
```

```
=> CREATE AGGREGATE average(float) (
    -- обычный агрегат
    stype      = average_state,
    initcond   = '(0,0)',
    sfunc      = average_transition,
    finalfunc  = average_final,
    -- вариант с обратной функцией
    mstype     = average_state,
    minitcond  = '(0,0)',
    msfunc     = average_transition,
    minvfunc   = average_inverse,
    mfinalfunc = average_final
);
```

```
CREATE AGGREGATE
```

Пробуем:

```
=> SELECT n, average(n) OVER(ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
FROM test;
```

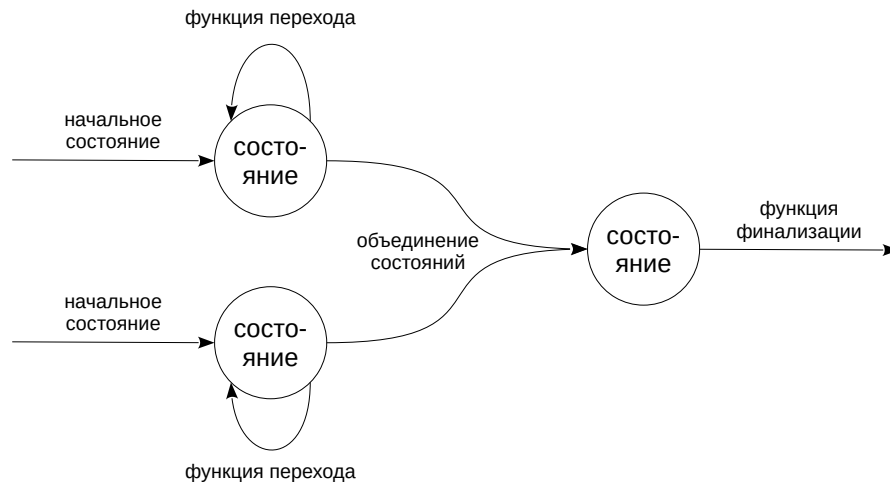
```
NOTICE:  0(0) + 1
NOTICE:  = 1(1)
NOTICE:  1(1) + 2
NOTICE:  = 3(2)
NOTICE:  3(2) + 3
NOTICE:  = 6(3)
NOTICE:  6(3) - 1
NOTICE:  5(2) + 4
NOTICE:  = 9(3)
NOTICE:  9(3) - 2
NOTICE:  7(2) + 5
NOTICE:  = 12(3)
```

n	average
1	1
2	1.5
3	2
4	3
5	4

(5 rows)

Теперь лишние операции не выполняются.

## Объединение состояний параллельных процессов



Агрегатные функции могут выполняться в параллельном режиме. Основной процесс, выполняющий запрос, может создать несколько фоновых рабочих процессов, каждый из которых будет выполнять параллельную обработку части данных в узле плана. Затем полученные результаты передаются в основной процесс, который собирает их и формирует общий результат.

Чтобы пользовательские агрегатные функции могли работать параллельно, нужно реализовать функцию объединения двух состояний в одно общее.

## Параллелизм

Таблица с пятью строчками, конечно, слишком мала для параллельного выполнения. Возьмем больше данных:

```
=> CREATE TABLE big (  
    n float  
);  
  
CREATE TABLE  
  
=> INSERT INTO big  
    SELECT random()*10::integer FROM generate_series(1,1_000_000);  
  
INSERT 0 1000000  
  
=> ANALYZE big;  
  
ANALYZE
```

Встроенные агрегатные функции могут выполняться в параллельном режиме:

```
=> EXPLAIN SELECT sum(n) FROM big;  
  
               QUERY PLAN  
-----  
Finalize Aggregate  (cost=10633.55..10633.56 rows=1 width=8)  
-> Gather  (cost=10633.33..10633.54 rows=2 width=8)  
    Workers Planned: 2  
    -> Partial Aggregate  (cost=9633.33..9633.34 rows=1 width=8)  
        -> Parallel Seq Scan on big  (cost=0.00..8591.67 rows=416667 width=8)  
(5 rows)
```

А наша функция — нет:

```
=> EXPLAIN SELECT average(n) FROM big;  
  
               QUERY PLAN  
-----  
Aggregate  (cost=264425.25..264425.26 rows=1 width=8)  
-> Seq Scan on big  (cost=0.00..14425.00 rows=1000000 width=8)  
JIT:  
  Functions: 3  
  Options: Inlining false, Optimization false, Expressions true, Deforming true  
(5 rows)
```

Чтобы поддержать параллельное выполнение, требуется еще одна функция для объединения двух состояний:

```
=> CREATE 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 IMMUTABLE;
```

CREATE FUNCTION

Кроме того, уберем отладочный вывод из функции перехода:

```
=> CREATE OR REPLACE FUNCTION average_transition(  
    state average_state,  
    val float  
)  
RETURNS average_state  
LANGUAGE sql IMMUTABLE  
RETURN ROW(state accum+val, state qty+1)::average_state;
```

CREATE FUNCTION

Пересоздадим агрегат, указав новую функцию и подтвердив безопасность параллельного выполнения:

```
=> DROP AGGREGATE average(float);
```

```
DROP AGGREGATE
```

```
=> CREATE AGGREGATE average(float) (  
    -- обычный агрегат  
    stype      = average_state,  
    initcond   = '(0,0)',  
    sfunc      = average_transition,  
    finalfunc   = average_final,  
    combinefunc = average_combine,  
    parallel   = safe,  
    -- вариант с обратной функцией  
    mstype     = average_state,  
    minitcond  = '(0,0)',  
    msfunc     = average_transition,  
    minvfunc   = average_inverse,  
    mfinalfunc = average_final  
);
```

```
CREATE AGGREGATE
```

Теперь наша функция тоже работает параллельно:

```
=> EXPLAIN SELECT average(n) FROM big;
```

QUERY PLAN

```
-----  
Finalize Aggregate (cost=113759.38..113759.39 rows=1 width=8)  
-> Gather (cost=113758.42..113758.63 rows=2 width=32)  
    Workers Planned: 2  
        -> Partial Aggregate (cost=112758.42..112758.43 rows=1 width=32)  
            -> Parallel Seq Scan on big (cost=0.00..8591.67 rows=416667 width=8)  
  
JIT:  
  Functions: 5  
  Options: Inlining false, Optimization false, Expressions true, Deforming true  
(8 rows)
```

```
=> SELECT average(n) FROM big;
```

```
NOTICE:  0(0) & 1677204.5364560492(335560)  
NOTICE: 1677204.5364560492(335560) & 1665859.2531424211(333576)  
NOTICE: 3343063.7895984706(669136) & 1654883.505679228(330864)  
NOTICE: = 4997947.295277699(1000000)  
      average  
-----  
  4.997947295277699  
(1 row)
```

Здесь видно, что три процесса поделили работу примерно поровну, и затем три состояния были попарно объединены.

В оконном режиме параллельное выполнение не поддерживается, в том числе и для встроенных функций.

PostgreSQL позволяет создавать агрегатные и оконные функции

Агрегатные и оконные функции дают возможность использовать процедурную обработку в стиле SQL



1. Напишите отчет, выводящий складские остатки по каждой книге в денежном выражении (используя закупочную, а не розничную цену). Оформите отчет как фоновое задание. Учтите, что поступления происходят разными партиями по разной цене, а продажи никак не привязаны к партиям. Поэтому считайте, что в первую очередь продаются книги из более старых партий.
2. Дополните расширение bookfmt функциями min и max для формата издания. Обновите версию расширения и убедитесь, что функции появились в базе данных.

1. Начните с запроса к таблице operations, выводящего только поступления книг, но добавьте столбец, показывающий общее количество проданных книг данного поступления.

В качестве примера книги, для которой на складе остались экземпляры из нескольких партий, можно взять книгу с book\_id = 15.

Напишите оконную функцию, вычисляющую при использовании в режиме «нарастающего итога» остаток книг данного поступления на складе (или решите задачу, используя имеющиеся оконные функции).

2. Расширение bookfmt было написано в практике к теме «Создание расширений».

## 1. Отчет по складским остаткам

Начнем с простого запроса, который выводит поступления книг и добавляет к каждой строке количество проданных экземпляров данной книги:

```
=> WITH sold(book_id, qty) AS (  
    -- продажи книг  
    SELECT book_id,  
           -sum(qty)  
    FROM   operations  
    WHERE  qty < 0  
    GROUP BY book_id  
)  
SELECT o.book_id,  
       o.qty,  
       s.qty as sold_qty,  
       o.price  
FROM   operations o  
       LEFT JOIN sold s ON s.book_id = o.book_id  
WHERE  o.qty > 0  
AND    o.book_id = 15 -- для примера  
ORDER BY o.book_id, o.at;
```

book_id	qty	sold_qty	price
15	36	515	1159
15	40	515	1290
15	35	515	1360
15	32	515	1210
15	37	515	1659
15	29	515	1286
15	31	515	1585
15	48	515	1686
15	35	515	1669
15	28	515	1345
15	33	515	1517
15	30	515	1582
15	28	515	1458
15	32	515	1522
15	32	515	1276
15	21	515	1422
15	28	515	1604

(17 rows)

Нам хотелось бы иметь функцию, которая на основании столбцов qty и sold\_qty вычислит остаток книг от каждого поступления.

Состояние такой агрегатной функции, которую мы будем использовать в режиме «нарастающего итога», включает два целых числа:

- количество книг, уже распределенных между поступлениями;
- количество нераспроданных книг в текущем поступлении.

```
=> CREATE TYPE distribute_state AS (  
    distributed integer,  
    qty integer  
);
```

CREATE TYPE

Функция перехода увеличивает количество распределенных книг на число книг в поступлении, но только до тех пор, пока оно не превышает общего числа проданных книг:

```
=> CREATE FUNCTION distribute_transition(  
    state distribute_state,  
    qty integer,  
    sold_qty integer  
)  
RETURNS distribute_state  
LANGUAGE sql IMMUTABLE  
RETURN ROW(  
    least(state.distributed + qty, sold_qty),  
    qty - (least(state.distributed + qty, sold_qty) - state.distributed)  
)::distribute_state;
```

CREATE FUNCTION

Функция финализации просто возвращает количество нераспроданных книг текущего поступления:

```
=> CREATE FUNCTION distribute_final(
    state distribute_state
)
RETURNS integer
LANGUAGE sql IMMUTABLE
RETURN state.qty;
```

CREATE FUNCTION

Создаем агрегат, указав тип состояния и его начальное значение, а также функции перехода и финализации:

```
=> CREATE AGGREGATE distribute(qty integer, sold_qty integer) (
    stype      = distribute_state,
    initcond   = '(0,0)',
    sfunc      = distribute_transition,
    finalfunc  = distribute_final
);
```

CREATE AGGREGATE

Добавим новую агрегатную функцию к запросу, указав группировку по книгам (book\_id) и сортировку в порядке времени совершения операций:

```
=> WITH sold(book_id, qty) AS (
    -- продажи книг
    SELECT book_id,
           -sum(qty)::integer
    FROM   operations
    WHERE  qty < 0
    GROUP BY book_id
)
SELECT o.book_id,
       o.qty,
       s.qty as sold_qty,
       distribute(o.qty, s.qty) OVER (
           PARTITION BY o.book_id ORDER BY o.at
       ) left_in_stock,
       o.price
FROM   operations o
LEFT JOIN sold s ON s.book_id = o.book_id
WHERE  o.qty > 0
AND    o.book_id = 15 -- для примера
ORDER BY o.book_id, o.at;
```

book_id	qty	sold_qty	left_in_stock	price
15	36	515	0	1159
15	40	515	0	1290
15	35	515	0	1360
15	32	515	0	1210
15	37	515	0	1659
15	29	515	0	1286
15	31	515	0	1585
15	48	515	0	1686
15	35	515	0	1669
15	28	515	0	1345
15	33	515	0	1517
15	30	515	0	1582
15	28	515	0	1458
15	32	515	0	1522
15	32	515	0	1276
15	21	515	12	1422
15	28	515	28	1604

(17 rows)

Осталось убрать условие для конкретной книги, умножить остаток на цену, сгруппировать результат по книгам и оформить запрос в виде функции, чтобы зарегистрировать ее как фоновое задание:

```

=> CREATE FUNCTION stock_task(params jsonb DEFAULT NULL)
RETURNS TABLE(book_id bigint, cost numeric)
LANGUAGE sql STABLE
BEGIN ATOMIC
    WITH sold(book_id, qty) AS (
        -- продажи книг
        SELECT book_id, -sum(qty)::integer FROM operations
        WHERE qty < 0
        GROUP BY book_id
    ), left_in_stock(book_id, cost) AS (
        SELECT o.book_id,
            distribute(o.qty, s.qty) OVER (
                PARTITION BY o.book_id ORDER BY o.at
            ) * price
        FROM operations o
        LEFT JOIN sold s ON s.book_id = o.book_id
        WHERE o.qty > 0
    )
    SELECT book_id, sum(cost) FROM left_in_stock GROUP BY book_id ORDER BY book_id;
END;

```

CREATE FUNCTION

```

=> SELECT register_program('Отчет по складским остаткам', 'stock_task');

```

```

register_program
-----

```

3

(1 row)

```

=> SELECT * FROM stock_task() LIMIT 10;

```

```

book_id | cost
-----+-----

```

1		5661
2		9224
3		13200
4		10400
5		28105
6		9259
7		7455
8		18615
9		15744
10		49676

(10 rows)

Заметим, что задачу можно решить и с помощью стандартных оконных функций:

```
=> WITH sold(book_id, qty) AS (
    -- продажи книг
    SELECT book_id,
           -sum(qty)
    FROM operations
    WHERE qty < 0
    GROUP BY book_id
), received(book_id, qty, cum_qty, price) AS (
    -- поступления книг
    SELECT book_id,
           qty,
           sum(qty) OVER (PARTITION BY book_id ORDER BY at),
           price
    FROM operations
    WHERE qty > 0
), left_in_stock(book_id, qty, price) AS (
    -- оставшиеся на складе книги
    SELECT r.book_id,
           CASE
               WHEN r.cum_qty - s.qty < 0 THEN 0
               WHEN r.cum_qty - s.qty < r.qty THEN r.cum_qty - s.qty
               ELSE r.qty
           END,
           r.price
    FROM received r
           LEFT JOIN sold s ON s.book_id = r.book_id
)
SELECT book_id,
       sum(qty*price)
FROM left_in_stock
GROUP BY book_id
ORDER BY book_id
LIMIT 10; -- ограничим вывод
```

book_id	sum
1	5661
2	9224
3	13200
4	10400
5	28105
6	9259
7	7455
8	18615
9	15744
10	49676

(10 rows)

Разумеется, отчет нетрудно сделать более наглядным, выводя название книги вместо идентификатора.

## 2. Min и max для типа book\_format

Сейчас функции min и max работают не в соответствии с определенным ранее классом операторов:

```
=> SELECT min(format), max(format) FROM books;
```

min	max
60x100/16	84x108/32

(1 row)

А вот правильный порядок:

```
=> SELECT format FROM books GROUP BY format ORDER BY format;
```

```

format
-----
(76,100,32)
(84,108,32)
(60,84,16)
(60,88,16)
(60,90,16)
(66,90,16)
(60,100,16)
(70,90,16)
(70,100,16)
(84,108,16)
(60,90,8)
(11 rows)

```

Создадим версию 1.1 расширения с учетом следующего:

- Состоянием для агрегатной функции является текущее минимальное (максимальное) значение.
- Функция перехода записывает в состояние минимальное (максимальное) из двух значений — запомненного в состоянии и текущего.
- Функция финализации не нужна — возвращается просто текущее состояние.

bookfmt/bookfmt.control

```

default_version = '1.1'
relocatable = true
encoding = UTF8
comment = 'Формат издания'

```

bookfmt/bookfmt--1.0--1.1.sql

```

\echo Use "CREATE EXTENSION bookfmt" to load this file. \quit

CREATE FUNCTION format_min_transition(
    min_so_far book_format,
    val book_format
)
RETURNS book_format
LANGUAGE sql IMMUTABLE
RETURN least(min_so_far, val);

CREATE AGGREGATE min(book_format) (
    stype      = book_format,
    sfunc      = format_min_transition,
    sortop     = <
);

CREATE FUNCTION format_max_transition(
    max_so_far book_format,
    val book_format
)
RETURNS book_format
LANGUAGE sql IMMUTABLE
RETURN greatest(max_so_far, val);

CREATE AGGREGATE max(book_format) (
    stype      = book_format,
    sfunc      = format_max_transition,
    sortop     = >
);

```

При создании агрегата мы дополнительно указали оператор сортировки, реализующий стратегию «меньше» («больше») класса операторов, чтобы планировщик мог оптимизировать вызов наших агрегатных функций, просто получая первое значение из индекса.

bookfmt/Makefile

```

EXTENSION = bookfmt
DATA = bookfmt--0.sql bookfmt--0--1.0.sql bookfmt--1.0--1.1.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)

```

```
student$ sudo make install -C bookfmt
```

```
make: Entering directory '/home/student/bookfmt'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/bin/mkdir -p '/usr/share/postgresql/16/extension'
/usr/bin/install -c -m 644 ./bookfmt.control '/usr/share/postgresql/16/extension/'
/usr/bin/install -c -m 644 ./bookfmt--0.sql ./bookfmt--0--1.0.sql
./bookfmt--1.0--1.1.sql '/usr/share/postgresql/16/extension/'
make: Leaving directory '/home/student/bookfmt'
```

Выполним обновление:

```
=> ALTER EXTENSION bookfmt UPDATE;
```

ALTER EXTENSION

Проверим:

```
=> SELECT min(format)::text, max(format)::text FROM books;
```

min	max
76x100/32	60x90/8

(1 row)

```
=> SELECT min(format)::text, max(format)::text FROM books WHERE false;
```

min	max
-----	-----

(1 row)

1. Напишите агрегатную функцию, вычисляющую средневзвешенное значение для совокупности элементов нескольких типов. Веса для типов элементов передаются функции в виде массива.
2. Бизнес-центр сдает офисы компаниям. В конце месяца администрации БЦ приходит общий счет за электроэнергию, который надо распределить между арендаторами пропорционально площади занимаемых помещений. Выставляемые арендаторам счета необходимо округлить до копеек, но так, чтобы их сумма совпала со значением, указанным в общем счете.

15

1. Средневзвешенное значение совокупности элементов — среднее значение, учитывающее вес, или важность, каждого элемента. Значение вычисляется по формуле

$$(x_1 w(t_1) + x_2 w(t_2) + \dots + x_n w(t_n)) / (w(t_1) + w(t_2) + \dots + w(t_n)), \text{ где}$$

$x_i$  — значение элемента,  $t_i$  — тип элемента,  $w(t_i)$  — вес элемента данного типа.

Требуется написать агрегатную функцию следующего вида:  
`w_agv(x float, t integer, w float[]).`

2. Помещения бизнес-центра можно представить таблицей:

```
rent (
  renter text PRIMARY KEY, -- арендатор
  area integer              -- площадь, м^2
);
```

Пример, показывающий проблему с обычным округлением:

```
INSERT INTO rent VALUES ('A',100), ('B',100), ('C',100);
SELECT round(1000.00 * area / sum(area) OVER ()), 2)
FROM rent;
```

## 1. Средневзвешенное значение

```
=> CREATE DATABASE ext_aggregates;
```

CREATE DATABASE

```
=> \c ext_aggregates
```

You are now connected to database "ext\_aggregates" as user "student".

Рассмотрим задачу на примере данных об оценках за экзаменационные работы. Пусть в экзаменационный билет входят теоретические вопросы (тип 1) и практические задания (тип 2), причем задания имеют больший вес в итоговой оценке.

```
=> CREATE TABLE results (
    student text,
    task_type integer CHECK (task_type IN (1,2)),
    task_score float CHECK (task_score BETWEEN 1 AND 10)
);
```

CREATE TABLE

```
=> INSERT INTO results(student, task_type, task_score) VALUES
    ('Иванов', 1, 4),
    ('Иванов', 1, 6),
    ('Иванов', 2, 10),
    ('Петров', 1, 8),
    ('Петров', 1, 10),
    ('Петров', 2, 5);
```

INSERT 0 6

Состояние опишем как составной тип, состоящий из суммы произведений значений на соответствующий вес и суммы весов:

```
=> CREATE TYPE w_avg_state AS (
    w_accum float,
    w_sum float
);
```

CREATE TYPE

Функция перехода будет получать (кроме состояния) три параметра: текущее значение, его тип и массив весов:

```
=> CREATE FUNCTION w_avg_transition(
    state w_avg_state,
    val float,
    val_type bigint,
    weight float[]
)
RETURNS w_avg_state AS $$
DECLARE
    w float := coalesce(weight[val_type], 0);
BEGIN
    RAISE NOTICE '%(%) + % * % [%]', state.w_accum, state.w_sum, val, w, val_type;
    RETURN ROW (state.w_accum + coalesce(val, 0) * w, state.w_sum + w)::w_avg_state;
END;
$$ LANGUAGE plpgsql IMMUTABLE;
```

CREATE FUNCTION

Функция финализации вычисляет средневзвешенное как отношение суммы произведений значений на соответствующие веса к сумме весов:

```
=> CREATE FUNCTION w_avg_final(
    state w_avg_state
)
RETURNS float AS $$
BEGIN
    RAISE NOTICE '= %(%)', state.w_accum, state.w_sum;
    RETURN CASE
        WHEN state.w_sum > 0 THEN state.w_accum / state.w_sum
    END;
END;
$$ LANGUAGE plpgsql IMMUTABLE;
```

CREATE FUNCTION

Теперь объявляем агрегат:

```
=> CREATE AGGREGATE w_avg(float, bigint, float[]) (
    stype      = w_avg_state,
    initcond   = '(0, 0)',
    sfunc      = w_avg_transition,
    finalfunc   = w_avg_final
);
```

CREATE AGGREGATE

А теперь подсчитаем средневзвешенные оценки за экзамен, сданный студентами, полагая, что решение практической задачи вдвое ценнее ответа на теоретический вопрос:

```
=> SELECT student, w_avg(task_score, task_type, ARRAY[1, 2])
FROM results
GROUP BY student;
```

```
NOTICE:  0(0) + 4 * 1 [1]
NOTICE:  4(1) + 6 * 1 [1]
NOTICE: 10(2) + 10 * 2 [2]
NOTICE:  0(0) + 8 * 1 [1]
NOTICE:  8(1) + 10 * 1 [1]
NOTICE: 18(2) + 5 * 2 [2]
NOTICE:  = 28(4)
NOTICE:  = 30(4)
 student | w_avg
-----+-----
  Петров |      7
  Иванов |     7.5
(2 rows)
```

## 2. Округление копеек

```
=> CREATE TABLE rent (
    renter text PRIMARY KEY,
    area integer
);
```

CREATE TABLE

```
=> INSERT INTO rent VALUES ('A',100), ('B',100), ('C',100);
```

INSERT 0 3

Состояние агрегатной функции будет включать округленную сумму и ошибку округления:

```
=> CREATE TYPE round2_state AS (
    rounded_amount numeric,
    rounding_error numeric
);
```

CREATE TYPE

Функция перехода добавляет к состоянию округленную сумму и ошибку округления. А если ошибка округления переваливает за полкопейки, то добавляет копейку к сумме.

```
=> CREATE FUNCTION round2_transition(
    state round2_state,
    val numeric
)
RETURNS round2_state AS $$
BEGIN
    state.rounding_error :=
        state.rounding_error + val - round(val,2);
    state.rounded_amount :=
        round(val,2) + round(state.rounding_error, 2);
    state.rounding_error :=
        state.rounding_error - round(state.rounding_error, 2);
    RETURN state;
END;
$$ LANGUAGE plpgsql IMMUTABLE;
```

CREATE FUNCTION

Функция финализации возвращает округленную сумму:

```
=> CREATE FUNCTION round2_final(
    state round2_state
)
RETURNS numeric
LANGUAGE sql IMMUTABLE
RETURN state.rounded_amount;
```

CREATE FUNCTION

Объявляем агрегат:

```
=> CREATE AGGREGATE round2(numeric) (  
    stype      = round2_state,  
    initcond   = '(0,0)',  
    sfunc      = round2_transition,  
    finalfunc  = round2_final  
);
```

CREATE AGGREGATE

Пробуем. Нам нужен какой-то определенный, но не важно какой именно, порядок просмотра строк. В данном случае подходит арендатор, поскольку он уникален.

```
=> WITH t AS (  
    SELECT *, sum(area) OVER () total_area  
    FROM rent  
)  
SELECT *, round2( 1000.00 * area / total_area ) OVER (ORDER BY renter)  
FROM t;
```

renter	area	total_area	round2
A	100	300	333.33
B	100	300	333.34
C	100	300	333.33

(3 rows)