Авторы: Егор Рогов, Павел Лузанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Массивы и работа с ними в PL/pgSQL

Циклы по элементам массивов

Функции с переменным числом параметров
и полиморфные функции

Использование массивов в столбцах таблиц

Тип массива

Массив

набор пронумерованных элементов одного и того же типа

одномерные, многомерные

Создание

использование без явного определения (*имя-типа*`[]`)

неявно при создании базового типа или таблицы (*_имя-типа*)

Использование

элементы как скалярные значения

срезы массива

операции с массивами: сравнение, вхождение, пересечение, конкатенация, использование с ANY и ALL вместо подзапроса, ...

Массив, как и составной тип (запись), не является скалярным и состоит из нескольких элементов другого типа. Но, в отличие от записей, а) все эти элементы имеют одинаковый тип и б) обращение к ним происходит не по имени, а по целочисленному индексу (здесь *индекс* понимается в математическом смысле, а не как индекс БД).

Тип массива не надо специально объявлять, достаточно добавить квадратные скобки к имени типа элементов. (При создании любого базового типа или таблицы автоматически создается соответствующий тип массива. Он получает то же имя, что и тип элемента, но с подчеркиванием впереди. Но такое имя выглядит не столь наглядно, как квадратные скобки.)

Массив является полноценным типом SQL, его можно использовать как любой другой тип: создавать столбцы таблиц этого типа, использовать его для параметров функций и т. п. Элементы массива могут использоваться как обычные скалярные значения. Можно использовать и срезы (slice) массивов.

Массивы можно сравнивать, проверять на неопределенность, определять вхождение элемента и находить пересечение с другими массивами, конкатенировать и пр. Также массивы можно использовать по аналогии с подзапросами в конструкциях ANY/SOME и ALL.

https://postgrespro.ru/docs/postgresql/12/arrays

Различные функции для работы с массивами можно найти в справочном материале, прилагаемом к курсу.

Инициализация и обращение к элементам

Объявление переменной и инициализация массива целиком:

```
=> DO $$
DECLARE
    a integer[]; -- размер игнорируется
BEGIN
    a := ARRAY[10,20,30];
    RAISE NOTICE '%', a;
    -- по умолчанию элементы нумеруются с единицы
    RAISE NOTICE 'a[1] = %, a[2] = %, a[3] = %', a[1], a[2], a[3];
    -- срез массива
    RAISE NOTICE 'Срез [2:3] = %', a[2:3];
END;
$$ LANGUAGE plpgsql;
```

```
NOTICE:  {10,20,30}
NOTICE:  a[1] = 10, a[2] = 20, a[3] = 30
NOTICE:  Срез [2:3] = {20,30}
DO
```

Одномерный массив можно заполнять и поэлементно — при необходимости он автоматически расширяется. Если пропустить какие-то элементы, они получают неопределенные значения.

Что будет выведено?

```
=> DO $$
DECLARE
    a integer[];
BEGIN
    a[2] := 10;
    a[3] := 20;
    a[6] := 30;
    RAISE NOTICE '%', a;
END;
$$ LANGUAGE plpgsql;
```

```
NOTICE:  [2:6]={10,20,NULL,NULL,30}
DO
```

Поскольку нумерация началась не с единицы, перед самим массивом дополнительно выводится диапазон номеров элементов.

Еще один способ получить массив — создать его из подзапроса:

```
=> DO $$
DECLARE
    a integer[];
BEGIN
    a := ARRAY( SELECT n FROM generate_series(1,3) n );
    RAISE NOTICE '%', a;
END;
$$ LANGUAGE plpgsql;
```

```
NOTICE:  {1,2,3}
DO
```

Можно и наоборот, массив преобразовать в таблицу:

```
=> SELECT unnest( ARRAY[1,2,3] );
```

```
 unnest
--------
      1
      2
      3
(3 rows)
```

Интересно, что выражение IN со списком значений преобразуется в поиск по массиву:

```
=> EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) g(id) WHERE id IN (1,2,3);
```

```
                QUERY PLAN
-------------------------------------------
 Function Scan on generate_series g
   Filter: (id = ANY ('{1,2,3}'::integer[]))
(2 rows)
```

Двумерный массив можно рассматривать как массив массивов (массив строк, каждая из которых — массив элементов). Здесь мы использовали другой способ инициализации с помощью строкового литерала.

После инициализации многомерный массив уже нельзя расширить.

```
=> DO $$
DECLARE
    a integer[][] := '{
        { 10, 20, 30},
        {100,200,300}
    }';
BEGIN
    RAISE NOTICE '%', a;
    RAISE NOTICE 'Срез [1:2][2:3] = %', a[1:2][2:3];
    -- расширять нельзя
    a[4][4] := 1;
END;
$$ LANGUAGE plpgsql;
```

```
NOTICE:  {{10,20,30},{100,200,300}}
NOTICE:  Срез [1:2][2:3] = {{20,30},{200,300}}
ERROR:  array subscript out of range
CONTEXT:  PL/pgSQL function inline_code_block line 11 at assignment
```

Массивы и циклы

Обычный цикл по индексам элементов

- array_lower
- array_upper

Цикл FOREACH по элементам массива

- проще, но индексы элементов недоступны

Для итерации по элементам массива вполне можно использовать обычный целочисленный цикл FOR от минимального до максимального индекса массива.

Однако есть и специализированный вариант цикла: FOREACH. В таком варианте переменная цикла пробегает не индексы элементов, а сами элементы. Поэтому переменная должна иметь тот же тип, что и элементы массива (как обычно, если элементами являются записи, то одну переменную составного типа можно заменить несколькими скалярными переменными).

Тот же цикл с фразой SLICE позволяет итерировать срезы массива. Например, для двумерного массива одномерными срезами будут его строки.

https://postgrespro.ru/docs/postgresql/12/plpgsql-control-structures#PLPGSQL-FOREACH-ARRAY

Массивы и циклы

Цикл можно организовать, итерируя индексы элементов массива. Второй параметр функций array_lower и array_upper — номер размерности (единица для одномерных массивов).

```
=> DO $$
DECLARE
    a integer[] := ARRAY[10,20,30];
BEGIN
    FOR i IN array_lower(a,1)..array_upper(a,1) LOOP
        RAISE NOTICE 'a[%] = %', i, a[i];
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
NOTICE:  a[1] = 10
NOTICE:  a[2] = 20
NOTICE:  a[3] = 30
DO
```

Если индексы не нужны, то проще итерировать сами элементы:

```
=> DO $$
DECLARE
    a integer[] := ARRAY[10,20,30];
    x integer;
BEGIN
    FOREACH x IN ARRAY a LOOP
        RAISE NOTICE '%', x;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
NOTICE:  10
NOTICE:  20
NOTICE:  30
DO
```

Итерация индексов в двумерном массиве:

```
=> DO $$
DECLARE
    -- можно и без двойных квадратных скобок
    a integer[] := ARRAY[
        ARRAY[ 10, 20, 30],
        ARRAY[100,200,300]
    ];
BEGIN
    FOR i IN array_lower(a,1)..array_upper(a,1) LOOP -- по строкам
        FOR j IN array_lower(a,2)..array_upper(a,2) LOOP -- по столбцам
            RAISE NOTICE 'a[%][%] = %', i, j, a[i][j];
        END LOOP;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
NOTICE:  a[1][1] = 10
NOTICE:  a[1][2] = 20
NOTICE:  a[1][3] = 30
NOTICE:  a[2][1] = 100
NOTICE:  a[2][2] = 200
NOTICE:  a[2][3] = 300
DO
```

Итерация элементов двумерного массива не требует вложенного цикла:

```
=> DO $$
DECLARE
    a integer[] := ARRAY[
        ARRAY[ 10, 20, 30],
        ARRAY[100,200,300]
    ];
    x integer;
BEGIN
    FOREACH x IN ARRAY a LOOP
        RAISE NOTICE '%', x;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
NOTICE:  10
NOTICE:  20
NOTICE:  30
NOTICE:  100
NOTICE:  200
NOTICE:  300
DO
```

Массивы и подпрограммы

Подпрограммы с переменным числом параметров

- все необязательные параметры должны иметь одинаковый тип
- необязательные параметры передаются в подпрограмму как массив
- последний формальный параметр-массив объявляется как VARIADIC

Полиморфные подпрограммы

- работают со значениями разных типов;
  тип конкретизируется во время выполнения
- используют полиморфные псевдотипы anyarray и anynonarray
- могут иметь переменное число параметров

Массивы позволяют создавать подпрограммы (функции или процедуры) с переменным числом параметров.

В отличие от параметров со значениями по умолчанию, которые при объявлении подпрограммы надо явно перечислить, необязательных параметров может быть сколько угодно и все они передаются подпрограмме в виде массива. Но, как следствие, все они должны иметь один и тот же тип.

При объявлении подпрограммы последний параметр помечается как VARIADIC и должен иметь тип массива.

https://postgrespro.ru/docs/postgresql/12/xfunc-sql#XFUNC-SQL-VARIADIC-FUNCTIONS

Мы уже говорили про полиморфные подпрограммы, которые могут работать с параметрами разных типов. При объявлении подпрограммы указывается специальный полиморфный псевдотип, а конкретный тип уточняется во время выполнения по фактическому типу переданных параметров.

Для массивов есть отдельный полиморфный тип anyarray (и anynonarray для не-массивов).

Этот тип можно использовать совместно с передачей переменного числа аргументов при объявлении VARIADIC-параметра.

https://postgrespro.ru/docs/postgresql/12/xfunc-sql#id-1.8.3.8.18

Массивы и подпрограммы

В теме «SQL. Процедуры» мы рассматривали перегрузку и полиморфизм и создали функцию maximum, которая находила максимальное из трех чисел. Обобщим ее на произвольное число аргументов. Для этого объявим один VARIADIC-параметр:

```
=> CREATE FUNCTION maximum(VARIADIC a integer[]) RETURNS integer
AS $$
DECLARE
    x integer;
    maxsofar integer;
BEGIN
    FOREACH x IN ARRAY a LOOP
        IF x IS NOT NULL AND (maxsofar IS NULL OR x > maxsofar) THEN
            maxsofar := x;
        END IF;
    END LOOP;
    RETURN maxsofar;
END;
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Пробуем:

```
=> SELECT maximum(12, 65, 47);
```

```
 maximum
---------
      65
(1 row)
```

```
=> SELECT maximum(12, 65, 47, null, 87, 24);
```

```
 maximum
---------
      87
(1 row)
```

```
=> SELECT maximum(null, null);
```

```
 maximum
---------

(1 row)
```

Для полноты картины и эта функция может быть сделана полиморфной, чтобы принимать любой тип данных (для которого, конечно, должны быть определены операции сравнения).

- Полиморфные типы anyarray и anyelement всегда согласованы между собой: anyarray = anyelement[];
- Нам нужна переменная, имеющая тип элемента массива. Но объявить ее как anyelement нельзя — она должна иметь реальный тип. Здесь помогает конструкция %TYPE.

```
=> DROP FUNCTION maximum(integer[]);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION maximum(VARIADIC a anyarray, maxsofar OUT anyelement)
AS $$
DECLARE
    x maxsofar%TYPE;
BEGIN
    FOREACH x IN ARRAY a LOOP
        IF x IS NOT NULL AND (maxsofar IS NULL OR x > maxsofar) THEN
            maxsofar := x;
        END IF;
    END LOOP;
END;
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Проверим:

```
=> SELECT maximum(12, 65, 47);
```

```
 maximum
---------
      65
(1 row)
```

```
=> SELECT maximum(12.1, 65.3, 47.6);
```

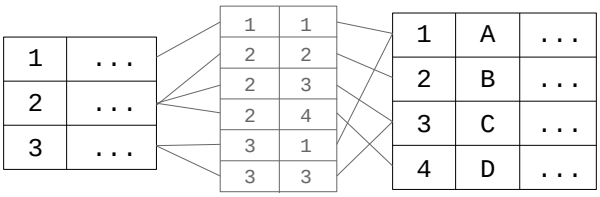
```
 maximum
---------
    65.3
(1 row)
```

Вот теперь у нас получился практически полный аналог выражения greatest!

Массив или таблица?

1	...	{A}
2	...	{B,C,D}
3	...	{A,C}

компактное представление
не требуется соединение
удобно в простых случаях



отдельные таблицы:
многие ко многим
универсальное решение

Классический реляционный подход предполагает, что в таблице хранятся атомарные значения (первая нормальная форма). Язык SQL не имеет средств для «заглядывания внутрь» сложносоставных значений.

Поэтому обычный подход состоит в создании отдельной таблицы, связанной с основной отношением «многие ко многим».

Тем не менее, мы можем создать таблицу со столбцом типа массива. PostgreSQL имеет богатый набор функций для работы с массивами, а поиск элемента в массиве может быть ускорен специальными индексами (такие индексы рассматриваются в курсе DEV2).

Такой подход бывает удобен: получается компактное представление, не требующее соединений. В частности, массивы активно используются в системном каталоге PostgreSQL.

Какое решение выбрать? Зависит от того, какие ставятся задачи, какие требуются операции. Рассмотрим пример.

Массив или таблица?

Представим себе, что мы проектируем базу данных для ведения блога. В блоге есть сообщения, и нам хотелось бы сопоставлять им теги.

Традиционный подход состоит в том, что для тегов надо создать отдельную таблицу, например, так:

```
=> CREATE TABLE posts(
    post_id integer PRIMARY KEY,
    message text
);
```

```
CREATE TABLE
```

```
=> CREATE TABLE tags(
    tag_id integer PRIMARY KEY,
    name text
);
```

```
CREATE TABLE
```

Связываем сообщения и теги отношением многие ко многим через еще одну таблицу:

```
=> CREATE TABLE posts_tags(
    post_id integer REFERENCES posts(post_id),
    tag_id integer REFERENCES tags(tag_id)
);
```

```
CREATE TABLE
```

Наполним таблицы тестовыми данными:

```
=> INSERT INTO posts(post_id,message) VALUES
    (1, 'Перечитывал пейджер, много думал.'),
    (2, 'Это было уже весной, и я отнес елку обратно.');
```

```
INSERT 0 2
```

```
=> INSERT INTO tags(tag_id,name) VALUES
    (1, 'былое и думы'), (2, 'технологии'), (3, 'семья');
```

```
INSERT 0 3
```

```
=> INSERT INTO posts_tags(post_id,tag_id) VALUES
    (1,1), (1,2), (2,1), (2,3);
```

```
INSERT 0 4
```

Теперь мы можем вывести сообщения и теги:

```
=> SELECT p.message, t.name
FROM posts p
     JOIN posts_tags pt ON pt.post_id = p.post_id
     JOIN tags t ON t.tag_id = pt.tag_id
ORDER BY p.post_id, t.name;
```

```
                  message                   |     name
--------------------------------------------+--------------
 Перечитывал пейджер, много думал.          | былое и думы
 Перечитывал пейджер, много думал.          | технологии
 Это было уже весной, и я отнес елку обратно. | былое и думы
 Это было уже весной, и я отнес елку обратно. | семья
(4 rows)
```

Более наглядно будет получить сообщения с тегами, сгруппированными в массив. Для этого используем агрегирующую функцию:

```
=> SELECT p.message, array_agg(t.name ORDER BY t.name) tags
FROM posts p
     JOIN posts_tags pt ON pt.post_id = p.post_id
     JOIN tags t ON t.tag_id = pt.tag_id
GROUP BY p.post_id
ORDER BY p.post_id;
```

```
                   message                    |            tags
----------------------------------------------+----------------------------
 Перечитывал пейджер, много думал.            | {"былое и думы",технологии}
 Это было уже весной, и я отнес елку обратно. | {"былое и думы",семья}
(2 rows)
```

Можем найти все сообщения с определенным тегом:

```
=> SELECT p.message
FROM posts p
     JOIN posts_tags pt ON pt.post_id = p.post_id
     JOIN tags t ON t.tag_id = pt.tag_id
WHERE t.name = 'былое и думы'
ORDER BY p.post_id;
```

```
                   message
-----------------------------------------------
 Перечитывал пейджер, много думал.
 Это было уже весной, и я отнес елку обратно.
(2 rows)
```

Может потребоваться найти все уникальные теги — это совсем просто:

```
=> SELECT t.name
FROM tags t
ORDER BY t.name;
```

```
     name
--------------
 былое и думы
 семья
 технологии
(3 rows)
```

Теперь попробуем подойти к задаче по-другому. Пусть теги будут представлены текстовым массивом прямо внутри таблицы сообщений.

```
=> DROP TABLE posts_tags;
```

```
DROP TABLE
```

```
=> DROP TABLE tags;
```

```
DROP TABLE
```

```
=> ALTER TABLE posts ADD COLUMN tags text[];
```

```
ALTER TABLE
```

Теперь у нас нет идентификаторов тегов, но они нам не очень и нужны.

```
=> UPDATE posts SET tags = '{"былое и думы","технологии"}'
WHERE post_id = 1;
```

```
UPDATE 1
```

```
=> UPDATE posts SET tags = '{"былое и думы","семья"}'
WHERE post_id = 2;
```

```
UPDATE 1
```

Вывод всех сообщений упростился:

```
=> SELECT p.message, p.tags
FROM posts p
ORDER BY p.post_id;
```

```
                   message                    |            tags
----------------------------------------------+----------------------------
 Перечитывал пейджер, много думал.            | {"былое и думы",технологии}
 Это было уже весной, и я отнес елку обратно. | {"былое и думы",семья}
(2 rows)
```

Сообщения с определенным тегом тоже легко найти (используем оператор пересечения &&).

Эта операция может быть ускорена с помощью индекса GIN, и для такого запроса не придется перебирать всю таблицу сообщений.

```
=> SELECT p.message
FROM posts p
WHERE p.tags && '{"былое и думы"}'
ORDER BY p.post_id;

                  message
--------------------------------------------
 Перечитывал пейджер, много думал.
 Это было уже весной, и я отнес елку обратно.
(2 rows)
```

А вот получить список тегов довольно сложно. Для этого требуется развернуть все массивы тегов в большую таблицу и найти в ней уникальные значения, а это тяжелая операция.

```
=> SELECT DISTINCT unnest(p.tags) AS name
FROM posts p;

     name
--------------
 технологии
 былое и думы
 семья
(3 rows)
```

Тут хорошо видно, что имеет место дублирование данных.

Итак, оба подхода вполне могут применяться.

В простых случаях массивы выглядят проще и работают хорошо.

В более сложных сценариях (представьте, что вместе с именем тега мы хотим хранить дату его создания; или требуется проверка ограничений целостности) классический вариант становится более привлекательным.

Итоги

Массив состоит из пронумерованных элементов одного и того же типа данных

Столбец с массивом как альтернатива отдельной таблице: удобные операции и индексная поддержка

Позволяет создавать функции с переменным числом параметров

Практика

1. Создайте функцию add_book для добавления новой книги.

   Функция должна принимать два параметра — название книги и массив идентификаторов авторов — и возвращать идентификатор новой книги.

   Проверьте, что в приложении появилась возможность добавлять книги.

1.

```
FUNCTION add_book(title text, authors integer[])
RETURNS integer
```

1. Функция add_book

```
=> CREATE OR REPLACE FUNCTION add_book(title text, authors integer[])
RETURNS integer
AS $$
DECLARE
    book_id integer;
    id integer;
    seq_num integer := 1;
BEGIN
    INSERT INTO books(title)
        VALUES(title)
        RETURNING books.book_id INTO book_id;
    FOREACH id IN ARRAY authors LOOP
        INSERT INTO authorship(book_id, author_id, seq_num)
            VALUES (book_id, id, seq_num);
        seq_num := seq_num + 1;
    END LOOP;
    RETURN book_id;
END;
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Практика

1. Реализуйте функцию map, принимающую два параметра: массив вещественных чисел и название вспомогательной функции, принимающей один параметр вещественного типа.

   Функция возвращает массив, полученный из исходного применением вспомогательной функции к каждому элементу.
2. Реализуйте функцию reduce, принимающую два параметра: массив вещественных чисел и название вспомогательной функции, принимающей два параметра вещественного типа.

   Функция возвращает вещественное число, полученное последовательной сверткой массива слева направо.
3. Сделайте функции map и reduce полиморфными.

1. Например:

```
map(ARRAY[4.0,9.0],'sqrt') → ARRAY[2.0,3.0]
```

2. Например:

```
reduce(ARRAY[1.0,3.0,2.0,0.5],'greatest') → 3.0
```

В этом случае значение вычисляется как

```
greatest( greatest( greatest(1.0,3.0), 2.0 ), 0.5 )
```

1. Функция map

```
=> CREATE DATABASE plpgsql_arrays;
```

```
CREATE DATABASE
```

```
=> \c plpgsql_arrays
```

```
You are now connected to database "plpgsql_arrays" as user "student".
```

```
=> CREATE FUNCTION map(a INOUT float[], func text)
AS $$
DECLARE
    i integer;
    x float;
BEGIN
    IF cardinality(a) > 0 THEN
        FOR i IN array_lower(a,1)..array_upper(a,1) LOOP
            EXECUTE format('SELECT %I($1)',func) USING a[i] INTO x;
            a[i] := x;
        END LOOP;
    END IF;
END;
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

- INTO a[i] не работает, поэтому нужна отдельная переменная.

```
=> SELECT map(ARRAY[4.0,9.0,16.0],'sqrt');
```

```
   map
---------
 {2,3,4}
(1 row)
```

```
=> SELECT map(ARRAY[]::float[],'sqrt');
```

```
 map
-----
 {}
(1 row)
```

Другой вариант реализации с циклом FOREACH:

```
=> CREATE OR REPLACE FUNCTION map(a float[], func text) RETURNS float[]
AS $$
DECLARE
    x float;
    b float[]; -- пустой массив
BEGIN
    FOREACH x IN ARRAY a LOOP
        EXECUTE format('SELECT %I($1)',func) USING x INTO x;
        b := b || x;
    END LOOP;
    RETURN b;
END;
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> SELECT map(ARRAY[4.0,9.0,16.0],'sqrt');
```

```
   map
---------
 {2,3,4}
(1 row)
```

```
=> SELECT map(ARRAY[]::float[],'sqrt');
```

```
 map
-----

(1 row)
```

2. Функция reduce

```
=> CREATE FUNCTION reduce(a float[], func text) RETURNS float
AS $$
DECLARE
    i integer;
    r float := NULL;
BEGIN
    IF cardinality(a) > 0 THEN
        r := a[array_lower(a,1)];
        FOR i IN array_lower(a,1)+1 .. array_upper(a,1) LOOP
            EXECUTE format('SELECT %I($1,$2)',func) USING r, a[i]
                INTO r;
        END LOOP;
    END IF;
    RETURN r;
END;
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Greatest (как и least) — не функция, а встроенное условное выражение, поэтому из-за экранирования не получится использовать ее напрямую:

```
=> SELECT reduce( ARRAY[1.0,3.0,2.0], 'greatest');
```

```
ERROR:  function greatest(double precision, double precision) does not exist
LINE 1: SELECT "greatest"($1,$2)
               ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.
QUERY:  SELECT "greatest"($1,$2)
CONTEXT:  PL/pgSQL function reduce(double precision[],text) line 9 at EXECUTE
```

Вместо нее используем реализованную в демонстрации функцию maximum.

```
=> CREATE FUNCTION maximum(VARIADIC a anyarray, maxsofar OUT anyelement)
AS $$
DECLARE
    x maxsofar%TYPE;
BEGIN
    FOREACH x IN ARRAY a LOOP
        IF x IS NOT NULL AND (maxsofar IS NULL OR x > maxsofar) THEN
            maxsofar := x;
        END IF;
    END LOOP;
END;
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> SELECT reduce(ARRAY[1.0,3.0,2.0], 'maximum');
```

```
 reduce
--------
      3
(1 row)
```

```
=> SELECT reduce(ARRAY[1.0], 'maximum');
```

```
 reduce
--------
      1
(1 row)
```

```
=> SELECT reduce(ARRAY[]::float[], 'maximum');
```

```
 reduce
--------

(1 row)
```

Вариант с циклом FOREACH:

```
=> CREATE OR REPLACE FUNCTION reduce(a float[], func text) RETURNS float
AS $$
DECLARE
    x float;
    r float;
    first boolean := true;

BEGIN
    FOREACH x IN ARRAY a LOOP
        IF first THEN
            r := x;
            first := false;
        ELSE
            EXECUTE format('SELECT %I($1,$2)',func) USING r, x INTO r;
        END IF;
    END LOOP;
    RETURN r;
END;
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION

=> SELECT reduce(ARRAY[1.0,3.0,2.0], 'maximum');

 reduce
--------
      3
(1 row)


=> SELECT reduce(ARRAY[1.0], 'maximum');

 reduce
--------
      1
(1 row)


=> SELECT reduce(ARRAY[]::float[], 'maximum');

 reduce
--------

(1 row)
```

3. Полиморфные варианты функций

Функция map.

```
=> DROP FUNCTION map(float[],text);

DROP FUNCTION

=> CREATE FUNCTION map(
    a anyarray,
    func text,
    elem anyelement DEFAULT NULL
)
RETURNS anyarray
AS $$
DECLARE
    x elem%TYPE;
    b a%TYPE;
BEGIN
    FOREACH x IN ARRAY a LOOP
        EXECUTE format('SELECT %I($1)',func) USING x INTO x;
        b := b || x;
    END LOOP;
    RETURN b;
END;
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION
```

- Требуется фиктивный параметр типа anyelement, чтобы внутри функции объявить переменную такого же типа.

```
=> SELECT map(ARRAY[4.0,9.0,16.0],'sqrt');
```

```
                            map
-----------------------------------------------------------
 {2.000000000000000,3.000000000000000,4.000000000000000}
(1 row)


=> SELECT map(ARRAY[]::float[],'sqrt');

 map
-----

(1 row)
```

Пример вызова с другим типом данных:

```
=> SELECT map(ARRAY[' a ','  b','c  '],'btrim');

   map
---------
 {a,b,c}
(1 row)
```

Функция reduce.

```
=> DROP FUNCTION reduce(float[],text);

DROP FUNCTION

=> CREATE FUNCTION reduce(
    a anyarray,
    func text,
    elem anyelement DEFAULT NULL
)
RETURNS anyelement
AS $$
DECLARE
    x elem%TYPE;
    r elem%TYPE;
    first boolean := true;
BEGIN
    FOREACH x IN ARRAY a LOOP
        IF first THEN
            r := x;
            first := false;
        ELSE
            EXECUTE format('SELECT %I($1,$2)',func) USING r, x INTO r;
        END IF;
    END LOOP;
    RETURN r;
END;
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION

=> CREATE FUNCTION add(x anyelement, y anyelement) RETURNS anyelement
AS $$
BEGIN
    RETURN x + y;
END;
$$ IMMUTABLE LANGUAGE plpgsql;

CREATE FUNCTION

=> SELECT reduce(ARRAY[1,-2,4], 'add');

 reduce
--------
      3
(1 row)


=> SELECT reduce(ARRAY['a','b','c'], 'concat');

 reduce
--------
 abc
(1 row)
```