

# SQL Составные типы



16

## 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.

Составные типы и работа с ними

Параметры функций составных типов

Функции, возвращающие одну строку

Функции, возвращающие множество строк

## Составной тип

набор именованных атрибутов (полей)  
то же, что табличная строка, но без ограничений целостности

## Создание

явное объявление нового типа  
неявно при создании таблицы  
неопределенный составной тип record

## Использование

атрибуты как скалярные значения  
операции со значениями составного типа: сравнение, проверка  
на NULL, использование с подзапросами

Составной тип — это набор атрибутов, каждый из которых имеет свое имя и свой тип. Составной тип можно рассматривать как табличную строку. Часто он называется «записью» (а в Си-подобных языках такой тип называется «структурой»).

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

Составной тип — объект базы данных, его объявление регистрирует новый тип в системном каталоге, после чего он становится полноценным типом SQL. При создании таблицы автоматически создается и одноименный составной тип, представляющий строку этой таблицы. Важное отличие состоит в том, что в составном типе нет ограничений целостности.

<https://postgrespro.ru/docs/postgresql/16/sql-createtype>

Атрибуты составного типа могут использоваться как отдельные значения (хотя атрибут, в свою очередь, тоже может иметь составной тип).

Составной тип можно использовать как любой другой тип SQL, например, создавать столбцы таблиц этого типа и т. п. Значения составного типа можно сравнивать между собой, проверять на неопределенность (NULL), использовать с подзапросами в таких конструкциях, как IN, ANY/SOME, ALL.

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

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

## Явное объявление составного типа

Первый способ ввести составной тип — явным образом объявить его:

```
=> CREATE TYPE currency AS (  
    amount numeric,  
    code   text  
);  
  
CREATE TYPE  
  
=> \dT  
  
List of data types  
Schema | Name | Description  
-----+-----+-----  
public | currency |  
(1 row)
```

Такой тип можно использовать точно так же, как любой другой тип SQL. Например, мы можем создать таблицу со столбцами такого типа:

```
=> CREATE TABLE transactions(  
    account_id integer,  
    debit       currency,  
    credit      currency,  
    date_entered date DEFAULT current_date  
);  
  
CREATE TABLE
```

Нужно ли так делать — вопрос сложный, универсальных рецептов тут не существует. В каких-то случаях это может быть полезно; в каких-то удобнее действовать в реляционных рамках: выделить сущность, представляемую типом, в отдельную таблицу и ссылаться на нее. Это позволяет избежать избыточности данных (нормализация) и упростить индексирование (в случае составного типа скорее всего потребуется индекс по выражению).

В целом PostgreSQL обладает достаточно большим количеством встроенных типов данных, так что, вероятно, необходимость в создании собственного типа будет возникать не часто, если не сказать редко.

---

## Конструирование значений составных типов

Значения составного типа можно формировать в виде строки, внутри которой в скобках перечислены значения. Обратите внимание, что строковые значения заключаются в двойные кавычки:

```
=> INSERT INTO transactions VALUES (1, NULL, '(100.00,"RUB")');  
  
INSERT 0 1
```

Другой способ — конструктор ROW:

```
=> INSERT INTO transactions VALUES (2, ROW(80.00,'RUB'), NULL);  
  
INSERT 0 1
```

Если составной тип содержит более одного поля, то слово ROW можно опустить:

```
=> INSERT INTO transactions VALUES (3, (20.00,'RUB'), NULL);  
  
INSERT 0 1
```

```
=> SELECT * FROM transactions;  
  
 account_id | debit | credit | date_entered  
-----+-----+-----+-----  
          1 |      | (100.00,RUB) | 2024-10-13  
          2 | (80.00,RUB) |      | 2024-10-13  
          3 | (20.00,RUB) |      | 2024-10-13  
(3 rows)
```

---

## Атрибуты составного типа как отдельные значения

Обращение к отдельному атрибуту составного типа — по сути то же, что и обращению к столбцу таблицы, ведь строка таблицы — это и есть составной тип:

```
=> SELECT t.account_id FROM transactions t;
```

```

account_id
-----
      1
      2
      3
(3 rows)

```

Как правило, требуется брать составное значение в скобки, например, чтобы отличать атрибут записи от столбца таблицы:

```
=> SELECT (t.debit).amount, (t.credit).amount FROM transactions t;
```

```

amount | amount
-----+-----
      | 100.00
 80.00 |
 20.00 |
(3 rows)

```

Или в случае, когда используется выражение:

```
=> SELECT ((10.00, 'RUB')::currency).amount;
```

```

amount
-----
 10.00
(1 row)

```

Составное значение не обязательно связано с каким-то конкретным типом, оно может быть неопределенной записью псевдотипа record:

```
=> SELECT (10.00, 'RUB')::record;
```

```

      row
-----
(10.00,RUB)
(1 row)

```

Но получится ли обратиться к атрибуту такой записи?

```
=> SELECT ((10.00, 'RUB')::record).amount;
```

```

ERROR:  could not identify column "amount" in record data type
LINE 1: SELECT ((10.00, 'RUB')::record).amount;
              ^

```

Нет, поскольку атрибуты такого типа безымянные.

## Неявный составной тип для таблиц

Более частое на практике применение составных типов — упрощение работы функций с таблицами.

При создании таблицы неявно создается и одноименный составной тип. Например, места в кинотеатре:

```
=> CREATE TABLE seats(
      line text,
      number integer
);
```

```
CREATE TABLE
```

```
=> INSERT INTO seats VALUES
      ('A', 42), ('B', 1), ('C', 27);
```

```
INSERT 0 3
```

Команда \dT «прячет» такие неявные типы, но при желании их можно увидеть непосредственно в таблице pg\_type:

```
=> SELECT typtype FROM pg_type WHERE typname = 'seats';
```

```

typtype
-----
 c
(1 row)

```

- c — composite, то есть составной тип.

## Операции над значениями составных типов

Значения составных типов можно сравнивать между собой. Это происходит поэлементно (примерно так же, как строки сравниваются посимвольно):

```
=> SELECT * FROM seats s WHERE s < ('B',52)::seats;
```

line	number
A	42
B	1

(2 rows)

Осторожно: существует много тонкостей, связанных с неопределенными значениями внутри записей.

Также работает проверка на неопределенность IS [NOT] NULL и сравнение IS [NOT] DISTINCT FROM.

Составные типы можно использовать с подзапросами, что бывает очень удобно.

Добавим таблицу с билетами:

```
=> CREATE TABLE tickets(  
    line text,  
    number integer,  
    movie_start date  
);
```

CREATE TABLE

```
=> INSERT INTO tickets VALUES  
    ('A', 42, current_date),  
    ('B', 1, current_date+1);
```

INSERT 0 2

Теперь, например, можно написать такой запрос для поиска мест в билетах на сегодняшний сеанс:

```
=> SELECT * FROM seats WHERE (line, number) IN (  
    SELECT line, number FROM tickets WHERE movie_start = current_date  
);
```

line	number
A	42

(1 row)

Без возможности использовать подзапрос пришлось бы явно соединять таблицы.

Подпрограмма может принимать параметры составного типа

Способ реализации вычисляемых полей

взаимозаменяемость `table.column` и `column(table)`

Другие способы

представления

столбцы `GENERATED ALWAYS`

Разумеется, подпрограммы могут принимать параметры составных типов.

Интересно, что для доступа к столбцу таблицы можно использовать не только привычную форму «таблица.столбец», но и функциональную: «столбец(таблица)». Это позволяет создавать вычисляемые поля, определяя функцию, принимающую на вход составной тип.

<https://postgrespro.ru/docs/postgresql/16/xfunc-sql>

Это несколько курьезный способ, поскольку того же результата можно добиться более явно с помощью представления. Стандарт SQL также предусматривает генерируемые (`GENERATED ALWAYS`) столбцы, хотя в PostgreSQL эта возможность пока реализована не в соответствии со стандартом — столбцы не вычисляются на лету, а сохраняются в таблице.

<https://postgrespro.ru/docs/postgresql/16/ddl-generated-columns>

## Параметры составного типа

Определим функцию, принимающую значение составного типа и возвращающую текстовый номер места.

```
=> CREATE FUNCTION seat_no(seat seats) RETURNS text
IMMUTABLE LANGUAGE sql
RETURN seat.line || seat.number;
```

CREATE FUNCTION

Обратите внимание, что в общем случае конкатенация имеет категорию изменчивости `stable`, а не `immutable`, поскольку для некоторых типов данных приведение к строке может давать разные результаты в зависимости от настроек.

```
=> SELECT seat_no(ROW('A',42));
```

```
seat_no
-----
A42
(1 row)
```

Что удобно, такой функции можно передавать непосредственно строку таблицы:

```
=> SELECT s.line, s.number, seat_no(s.*) FROM seats s;
```

```
line | number | seat_no
-----+-----+-----
A    |      42 | A42
B    |       1 | B1
C    |      27 | C27
(3 rows)
```

Можно обойтись и без «звездочки»:

```
=> SELECT s.line, s.number, seat_no(s) FROM seats s;
```

```
line | number | seat_no
-----+-----+-----
A    |      42 | A42
B    |       1 | B1
C    |      27 | C27
(3 rows)
```

Синтаксисом допускается обращение к функции как к столбцу таблицы (и наоборот, к столбцу как к функции):

```
=> SELECT s.line, number(s), s.seat_no FROM seats s;
```

```
line | number | seat_no
-----+-----+-----
A    |      42 | A42
B    |       1 | B1
C    |      27 | C27
(3 rows)
```

Таким образом можно использовать функции как вычисляемые «на лету» столбцы таблиц.

Что, если и в таблице окажется столбец с тем же именем? Раньше в любом случае предпочтение отдавалось столбцу, а начиная с версии 11 выбор зависит от синтаксической формы.

Разумеется, такого же эффекта можно добиться, определив представление.

```
=> CREATE VIEW seats_v AS
    SELECT s.line, s.number, seat_no(s) FROM seats s;
```

CREATE VIEW

```
=> SELECT line, number, seat_no FROM seats_v;
```

```
line | number | seat_no
-----+-----+-----
A    |      42 | A42
B    |       1 | B1
C    |      27 | C27
(3 rows)
```



А начиная с версии 12, PostgreSQL позволяет при создании таблиц объявить «настоящие» вычисляемые столбцы. Правда, в отличие от стандарта SQL, такие столбцы не вычисляются на лету, а сохраняются в таблице:

```
=> CREATE TABLE seats2(  
    line text,  
    number integer,  
    seat_no text GENERATED ALWAYS AS (seat_no(ROW(line,number))) STORED  
);
```

CREATE TABLE

```
=> \d seats2
```

Table "public.seats2"				
Column	Type	Collation	Nullable	Default
line	text			
number	integer			
seat_no	text			generated always as (seat_no(ROW(line, number))) stored

```
=> INSERT INTO seats2 (line, number)  
    SELECT line, number FROM seats;
```

INSERT 0 3

```
=> SELECT * FROM seats2;
```

line	number	seat_no
A	42	A42
B	1	B1
C	27	C27

(3 rows)

Если мы впоследствии решили, что следует явно задавать значение поля, то нам достаточно удалить выражение:

```
=> ALTER TABLE seats2 ALTER COLUMN seat_no DROP EXPRESSION;
```

ALTER TABLE

Существующие данные в столбце остались, но столбец теперь не вычисляемый, а "обычный":

```
=> SELECT * FROM seats2;
```

line	number	seat_no
A	42	A42
B	1	B1
C	27	C27

(3 rows)

```
=> \d seats2
```

Table "public.seats2"				
Column	Type	Collation	Nullable	Default
line	text			
number	integer			
seat_no	text			

## «Однострочные» функции

Возвращают значение составного типа

Обычно вызываются в списке выборки запроса

При вызове в предложении FROM возвращают  
однострочную таблицу

Функции могут не только принимать параметры составного типа, но и возвращать значение составного типа.

Обычно функции вызываются в списке выборки запроса (предложение SELECT).

Но функцию можно вызвать и в предложении FROM, как будто таблицу из одной строки.

## Функции, возвращающие одно значение

Напишем функцию, конструирующую и возвращающую табличную строку по отдельным компонентам.

Такую функцию можно объявить как RETURNS seats:

```
=> CREATE FUNCTION seat(line text, number integer) RETURNS seats
IMMUTABLE LANGUAGE sql
RETURN ROW(line, number)::seats;
```

CREATE FUNCTION

```
=> SELECT seat('A', 42);
```

```
seat
-----
(A,42)
(1 row)
```

Мы получаем результат составного типа. Его можно «развернуть» в однострочную таблицу:

```
=> SELECT (seat('A', 42)).*;
```

```
line | number
-----+-----
A    |      42
(1 row)
```

Имена столбцов и их типы получены здесь из описания составного типа seats.

Но функцию можно вызывать не только в списке выборки запроса или в условиях, как часть выражения. К функции можно обратиться и в предложении FROM, как к таблице:

```
=> SELECT * FROM seat('A', 42);
```

```
line | number
-----+-----
A    |      42
(1 row)
```

При этом мы тоже получаем однострочную таблицу.

Кстати, можно ли подобным образом вызвать функцию, возвращающую отдельное (скалярное) значение?

```
=> SELECT * FROM abs(-1.5);
```

```
abs
-----
1.5
(1 row)
```

Да, так тоже можно: мы получили одну строку, состоящую из одного столбца.

---

Другой вариант, который мы уже видели в теме «SQL. Функции» — объявить выходные параметры.

Заодно отметим, что в запросе не обязательно собирать составной тип из отдельных полей — это будет проделано автоматически:

```
=> DROP FUNCTION seat(text, integer);
```

DROP FUNCTION

```
=> CREATE FUNCTION seat(line INOUT text, number INOUT integer)
IMMUTABLE LANGUAGE sql
RETURN (line, number);
```

CREATE FUNCTION

```
=> SELECT seat('A', 42);
```

```
seat
-----
(A,42)
(1 row)
```

```
=> SELECT * FROM seat('A', 42);
```

```

line | number
-----+-----
A    |      42
(1 row)

```

Получаем тот же результат — но имена и типы полей в данном случае получены из выходных параметров функции, а сам составной тип остается анонимным.

И еще один вариант — объявить функцию как возвращающую псевдотип `record`, который обозначает составной тип «вообще», без уточнения его структуры.

```
=> DROP FUNCTION seat(text, integer);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION seat(line text, number integer) RETURNS record
IMMUTABLE LANGUAGE sql
RETURN (line, number);
```

```
CREATE FUNCTION
```

```
=> SELECT seat('A',42);
```

```

seat
-----
(A,42)
(1 row)

```

Но вызвать такую функцию в предложении `FROM` уже не получится, поскольку возвращаемый составной тип не просто анонимный, но и количество и типы его полей заранее (на этапе разбора запроса) неизвестны:

```
=> SELECT * FROM seat('A',42);
```

```

ERROR:  a column definition list is required for functions returning "record"
LINE 1: SELECT * FROM seat('A',42);
              ^

```

В этом случае при вызове функции структуру составного типа придется уточнить — мы явно указываем названия и типы столбцов:

```
=> SELECT * FROM seat('A',42) AS (line text, number integer);
```

```

line | number
-----+-----
A    |      42
(1 row)

```

При написании функций допустим любой из этих трех вариантов, но лучше сразу подумать об использовании: будет ли удобен анонимный тип и уточнение структуры типа при вызове.

Объявляются как RETURNS SETOF или RETURNS TABLE

Могут возвращать несколько строк

Обычно вызываются в предложении FROM

Можно использовать как представление с параметрами

особенно удобно в сочетании с подстановкой тела функции в запрос

Мы знаем, что к функциям можно обращаться в предложении FROM, однако до сих пор результатом была одна строка. Конечно, интереснее было бы иметь функции, возвращающие множество строк (табличные функции) — и их действительно можно определять.

Табличные функции естественно вызывать в предложении FROM и можно рассматривать их как своеобразное представление.

(Формально PostgreSQL позволяет вызвать такую функцию и в списке выборки, но так лучше не делать.)

Как и с обычными функциями, в ряде случаев планировщик может подставить тело функции в основной запрос. Это позволяет создавать «представления с параметрами» без накладных расходов.

[https://wiki.postgresql.org/wiki/Inlining\\_of\\_SQL\\_functions](https://wiki.postgresql.org/wiki/Inlining_of_SQL_functions)

## Функции, возвращающие множество строк (табличные функции)

Напишем функцию, которая вернет все места в прямоугольном зале заданного размера.

```
=> CREATE FUNCTION rect_hall(max_line integer, max_number integer)
RETURNS SETOF seats
IMMUTABLE LANGUAGE sql
BEGIN ATOMIC
    SELECT chr(line+64), number
    FROM generate_series(1,max_line) AS line,
         generate_series(1,max_number) AS number;
END;

CREATE FUNCTION
```

Ключевое отличие — слово SETOF. В таком случае функция возвращает не первую строку последнего запроса, как обычно, а все строки последнего запроса.

```
=> SELECT * FROM rect_hall(max_line => 2, max_number => 3);
```

line	number
A	1
A	2
A	3
B	1
B	2
B	3

(6 rows)

Вместо SETOF seats можно использовать и SETOF record:

```
=> DROP FUNCTION rect_hall(integer, integer);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION rect_hall(max_line integer, max_number integer)
RETURNS SETOF record
IMMUTABLE LANGUAGE sql
BEGIN ATOMIC
    SELECT chr(line+64), number
    FROM generate_series(1,max_line) AS line,
         generate_series(1,max_number) AS number;
END;
```

```
CREATE FUNCTION
```

Но в этом случае, как мы видели, при вызове функции придется уточнять структуру составного типа:

```
=> SELECT * FROM rect_hall(max_line => 2, max_number => 3)
    AS (a_line text, a_number integer);
```

a_line	a_number
A	1
A	2
A	3
B	1
B	2
B	3

(6 rows)

А можно объявить функцию с выходными параметрами. Но SETOF record все равно придется написать, чтобы показать, что функция возвращает не одну строку, а множество:

```
=> DROP FUNCTION rect_hall(integer, integer);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION rect_hall(
    max_line integer, max_number integer,
    OUT p_line text, OUT p_number integer
)
RETURNS SETOF record
IMMUTABLE LANGUAGE sql
BEGIN ATOMIC
    SELECT chr(line+64), number
    FROM generate_series(1,max_line) AS line,
         generate_series(1,max_number) AS number;
END;

CREATE FUNCTION

=> SELECT * FROM rect_hall(max_line => 2, max_number => 3);
```

p_line	p_number
A	1
A	2
A	3
B	1
B	2
B	3

(6 rows)

Еще один равнозначный (и к тому же описанный в стандарте SQL) способ объявить табличную функцию — указать слово TABLE:

```
=> DROP FUNCTION rect_hall(integer, integer);

DROP FUNCTION

=> CREATE FUNCTION rect_hall(max_line integer, max_number integer)
RETURNS TABLE(t_line text, t_number integer)
LANGUAGE sql
BEGIN ATOMIC
    SELECT chr(line+64), number
    FROM generate_series(1,max_line) AS line,
         generate_series(1,max_number) AS number;
END;

CREATE FUNCTION

=> SELECT * FROM rect_hall(max_line => 2, max_number => 3);
```

t_line	t_number
A	1
A	2
A	3
B	1
B	2
B	3

(6 rows)

Иногда в запросах бывает полезно пронумеровать строки в том порядке, в котором они получены от функции. Для этого есть специальная конструкция:

```
=> SELECT *
FROM rect_hall(max_line => 2, max_number => 3) WITH ORDINALITY;
```

t_line	t_number	ordinality
A	1	1
A	2	2
A	3	3
B	1	4
B	2	5
B	3	6

(6 rows)

При использовании функции в предложении FROM, перед ней неявно подразумевается ключевое слово LATERAL, что позволяет функции обращаться к столбцам таблиц, стоящих в запросе слева от нее. Иногда это позволяет упростить формулировку запросов.

Например, напомним функцию, конструирующую зал наподобие амфитеатра, в котором дальние ряды имеют больше мест, чем ближние:

```
=> CREATE FUNCTION amphitheatre(max_line integer)
RETURNS TABLE(t_line text, t_number integer)
IMMUTABLE LANGUAGE sql
BEGIN ATOMIC
    SELECT chr(line + 64), number
    FROM generate_series(1,max_line) AS line, -- <--+
         generate_series(1, --
                        line -----+
                        ) AS number;
END;
```

CREATE FUNCTION

```
=> SELECT * FROM amphitheatre(3);
```

t_line	t_number
A	1
B	1
B	2
C	1
C	2
C	3

(6 rows)

Интересно, что к функции, возвращающей множество строк, можно обратиться и в списке выборки запроса:

```
=> SELECT rect_hall(3,4);
```

rect_hall
(A,1)
(A,2)
(A,3)
(A,4)
(B,1)
(B,2)
(B,3)
(B,4)
(C,1)
(C,2)
(C,3)
(C,4)

(12 rows)

В некоторых случаях это смотрится логично, но иногда результат может удивить. Например, сколько строк вернет такой запрос?

```
=> SELECT rect_hall(2,3), rect_hall(2,2);
```

rect_hall	rect_hall
(A,1)	(A,1)
(A,2)	(A,2)
(A,3)	(B,1)
(B,1)	(B,2)
(B,2)	
(B,3)	

(6 rows)

Получается 6 строк, а в версиях до 10 получалось наименьшее общее кратное числа строк, возвращенных каждой функцией (12 в данном случае).

Еще хуже, что запрос может вернуть меньше строк, чем ожидалось, если функция не вернет ни одной строки при каком-то значении параметров.

Поэтому такой способ вызова не стоит использовать.

## Функции как представления с параметрами

Как мы видели, функцию можно использовать во фразе FROM как таблицу или представление. Но при этом мы дополнительно получаем возможность использовать параметры, что в ряде случаев бывает удобно.

Единственная сложность с таким подходом состоит в том, что при обращении к функции (Function Scan) запросы из нее сначала выполняются полностью, и только затем к результату применяются дополнительные условия из запроса.



```

=> EXPLAIN (costs off)
SELECT * FROM rect_hall(3,4) WHERE t_line = 'A';

      QUERY PLAN
-----
Function Scan on rect_hall
  Filter: (t_line = 'A'::text)
(2 rows)

```

Если бы функция содержала сложный, долгий запрос, это могло бы стать проблемой.

В некоторых случаях тело функции может подставляться в вызывающий запрос. Для табличных функций ограничения более мягкие:

- функция написана на языке SQL;
- функция сама не должна быть изменчивой (VOLATILE) и не должна содержать вызовов таких функций;
- функция не должна быть строгой (STRICT);
- тело должно содержать единственный оператор SELECT (но он может быть сложным);
- и ряд других ограничений.

В нашем случае дело в том, что последний раз мы объявили функцию как изменчивую, не указав категорию изменчивости явно.

```

=> ALTER FUNCTION rect_hall(integer, integer) IMMUTABLE;

```

```

ALTER FUNCTION

```

```

=> EXPLAIN (costs off)
SELECT * FROM rect_hall(3,4) WHERE t_line = 'A';

      QUERY PLAN
-----
Nested Loop
  -> Function Scan on generate_series line
      Filter: (chr((line + 64)) = 'A'::text)
  -> Function Scan on generate_series number
(4 rows)

```

Теперь нет вызова функции как такового, а условие подставлено внутрь запроса, что более эффективно.

Составной тип объединяет значения других типов

Упрощает и обогащает работу функций с таблицами

Позволяет создавать вычисляемые поля  
и представления с параметрами

Функции могут возвращать множество строк



1. Создайте функцию `onhand_qty` для подсчета имеющихся в наличии книг. Функция принимает параметр составного типа `books` и возвращает целое число.

Используйте эту функцию в представлении `catalog_v` в качестве «вычисляемого поля».

Проверьте, что приложение отображает количество книг.

2. Создайте табличную функцию `get_catalog` для поиска книг. Функция принимает значения полей формы поиска («имя автора», «название книги», «есть на складе») и возвращает подходящие книги в формате `catalog_v`.

Проверьте, что в «Магазине» начал работать поиск и просмотр.

1.

```
FUNCTION onhand_qty(book books) RETURNS integer
```

2.

```
FUNCTION get_catalog(
    author_name text, book_title text, in_stock boolean
)
RETURNS TABLE(
    book_id integer, display_name text, onhand_qty integer
)
```

При решении хотелось бы воспользоваться уже готовым представлением `catalog_v`, просто наложив ограничения на строки. Но в этом представлении и название книги, и авторы находятся в одном поле, к тому же в сокращенном виде. Очевидно, что поиск автора «Лев» по полю «Л .Н. Толстой» не даст результата.

Можно было бы повторить в функции `get_catalog` запрос из `catalog_v`, но это дублирование кода, что плохо. Поэтому расширьте представление `catalog_v`, добавив в него дополнительные поля: заголовок книги и полный список авторов.

Проверьте, что корректно обрабатываются пустые поля на форме. Когда клиент вызывает функцию `get_catalog`, передает ли он в этом случае пустые строки или неопределенные значения?

## 1. Функция onhand\_qty

```
=> CREATE FUNCTION onhand_qty(book books) RETURNS integer
STABLE LANGUAGE sql
BEGIN ATOMIC
    SELECT coalesce(sum(o.qty_change),0)::integer
    FROM operations o
    WHERE o.book_id = book.book_id;
END;

CREATE FUNCTION

=> CREATE OR REPLACE VIEW catalog_v AS
SELECT b.book_id,
       book_name(b.book_id, b.title) AS display_name,
       b.onhand_qty
FROM   books b
ORDER BY display_name;

CREATE VIEW
```

## 2. Функция get\_catalog

Расширяем catalog\_v заголовком книги и полным списком авторов (приложение игнорирует неизвестные ему поля).

Функция, возвращающая полный список авторов:

```
=> CREATE FUNCTION authors(book books) RETURNS text
STABLE LANGUAGE sql
BEGIN ATOMIC
    SELECT string_agg(
        a.last_name ||
        ' ' ||
        a.first_name ||
        coalesce(' ' || nullif(a.middle_name, ''), ''),
        ','
        ORDER BY ash.seq_num
    )
FROM   authors a
JOIN   authorship ash ON a.author_id = ash.author_id
WHERE  ash.book_id = book.book_id;
END;

CREATE FUNCTION
```

Используем эту функцию в представлении catalog\_v. Такое представление уже существует; мы модифицируем его — изменим порядок столбцов и запрос:

```
=> DROP VIEW catalog_v;

DROP VIEW

=> CREATE VIEW catalog_v AS
SELECT b.book_id,
       b.title,
       b.onhand_qty,
       book_name(b.book_id, b.title) AS display_name,
       b.authors
FROM   books b
ORDER BY display_name;

CREATE VIEW
```

Функция get\_catalog теперь использует расширенное представление:

```
=> CREATE FUNCTION get_catalog(  
    author_name text,  
    book_title text,  
    in_stock boolean  
)  
RETURNS TABLE(book_id integer, display_name text, onhand_qty integer)  
STABLE LANGUAGE sql  
BEGIN ATOMIC  
    SELECT cv.book_id,  
           cv.display_name,  
           cv.onhand_qty  
    FROM   catalog_v cv  
    WHERE  cv.title ILIKE '%' || coalesce(book_title, '') || '%'  
    AND    cv.authors ILIKE '%' || coalesce(author_name, '') || '%'  
    AND    (in_stock AND cv.onhand_qty > 0 OR in_stock IS NOT TRUE)  
    ORDER BY display_name;  
END;  
  
CREATE FUNCTION
```

1. Напишите функцию, переводящую строку, содержащую число в шестнадцатеричной системе, в обычное целое число.  
Например: `convert('FF') → 255`
2. Добавьте в функцию второй необязательный параметр — основание системы счисления (по умолчанию — 16).  
Например: `convert('0110', 2) → 6`
3. Табличная функция `generate_series` не работает со строковыми типами. Предложите свою функцию для генерации последовательностей строк из заглавных английских букв.

1. Например:

`convert('FF') → 255`

Для решения пригодятся: табличная функция `regexp_split_to_table`, функции `upper` и `reverse`, конструкция `WITH ORDINALITY`.

Другое решение возможно с помощью рекурсивного запроса.

Проверить реализацию можно, используя шестнадцатеричные константы: `SELECT X'FF'::integer;`

2. Например:

`convert('0110', 2) → 6`

3. Считайте, что на вход подаются строки равной длины. Например:

`generate_series('AA', 'ZZ') →`

```
→ 'AA'
   'AB'
   'AC'
   ...
   'ZY'
   'ZZ'
```

## 1. Функция для шестнадцатеричной системы

```
=> CREATE DATABASE sql_row;
```

```
CREATE DATABASE
```

```
=> \c sql_row
```

You are now connected to database "sql\_row" as user "student".

Сначала для удобства определим функцию для одной цифры:

```
=> CREATE FUNCTION digit(d text) RETURNS integer
IMMUTABLE LANGUAGE sql
RETURN ascii(d) - CASE
    WHEN d BETWEEN '0' AND '9' THEN ascii('0')
    ELSE ascii('A') - 10
END;
```

```
CREATE FUNCTION
```

Теперь основная функция:

```
=> CREATE FUNCTION convert(hex text) RETURNS integer
IMMUTABLE LANGUAGE sql
BEGIN ATOMIC
    WITH s(d,ord) AS (
        SELECT *
        FROM regexp_split_to_table(reverse(upper(hex)), '') WITH ORDINALITY
    )
    SELECT sum(digit(d) * 16^(ord-1))::integer FROM s;
END;
```

```
CREATE FUNCTION
```

```
=> SELECT convert('0FE'), convert('0FF'), convert('100');
```

```
convert | convert | convert
-----+-----+-----
    254 |    255 |    256
(1 row)
```

## 2. Функция для любой системы счисления

Предполагаем, что основание системы счисления от 2 до 36, то есть число записывается цифрами от 0 до 9, либо буквами от A до Z. В этом случае изменения минимальные.

```
=> DROP FUNCTION convert(text);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION convert(num text, radix integer DEFAULT 16) RETURNS integer
IMMUTABLE LANGUAGE sql
BEGIN ATOMIC
    WITH s(d,ord) AS (
        SELECT *
        FROM regexp_split_to_table(reverse(upper(num)), '') WITH ORDINALITY
    )
    SELECT sum(digit(d) * radix^(ord-1))::integer FROM s;
END;
```

```
CREATE FUNCTION
```

```
=> SELECT convert('101100', 2), convert('2C'), convert('54', 8);
```

```
convert | convert | convert
-----+-----+-----
    44 |    44 |    44
(1 row)
```

Заметим, что в PostgreSQL начиная с версии 16 есть возможность записи целочисленных констант не только в десятичном, но и в двоичном, шестнадцатеричном и восьмеричном виде. Такая возможность закреплена в современном стандарте SQL:

```
=> SELECT 0b101100 AS bin, 0x2C AS hex, 0o54 AS oct;
```

```

bin | hex | oct
-----+-----+-----
44 | 44 | 44
(1 row)

```

### 3. Функция generate\_series для строк

Сначала напомним вспомогательные функции, переводящие строку в числовое представление и обратно.

Первая очень похожа на функцию из предыдущего задания:

```

=> CREATE FUNCTION text2num(s text) RETURNS integer
IMMUTABLE LANGUAGE sql
BEGIN ATOMIC
    WITH s(d,ord) AS (
        SELECT *
        FROM regexp_split_to_table(reverse(s), '') WITH ORDINALITY
    )
    SELECT sum( (ascii(d)-ascii('A')) * 26^(ord-1))::integer FROM s;
END;

```

```
CREATE FUNCTION
```

Обратную функцию напомним с помощью рекурсивного запроса:

```

=> CREATE FUNCTION num2text(n integer, digits integer) RETURNS text
IMMUTABLE LANGUAGE sql
BEGIN ATOMIC
    WITH RECURSIVE r(num,txt, level) AS (
        SELECT n/26, chr( n%26 + ascii('A') )::text, 1
        UNION ALL
        SELECT r.num/26, chr( r.num%26 + ascii('A') ) || r.txt, r.level+1
        FROM r
        WHERE r.level < digits
    )
    SELECT r.txt FROM r WHERE r.level = digits;
END;

```

```
CREATE FUNCTION
```

```

=> SELECT num2text( text2num('ABC'), length('ABC') );

 num2text
-----
ABC
(1 row)

```

Теперь функцию generate\_series для строк можно переписать, используя generate\_series для целых чисел.

```

=> CREATE FUNCTION generate_series(start text, stop text) RETURNS SETOF text
IMMUTABLE LANGUAGE sql
BEGIN ATOMIC
    SELECT num2text( g.n, length(start)) FROM generate_series(text2num(start), text2num(stop)) g(n);
END;

```

```
CREATE FUNCTION
```

```

=> SELECT generate_series('AZ','BC');

 generate_series
-----
AZ
BA
BB
BC
(4 rows)

```

```
=> \c postgres
```

You are now connected to database "postgres" as user "student".

```
=> DROP DATABASE sql_row;
```

```
DROP DATABASE
```