

# SQL Composite Types



16

## Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

## Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

## Feedback

Please send your feedback, comments and suggestions to:

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

## Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Composite Types and How to Use Them

Composite Type Arguments

Functions Returning a Single Row

Functions Returning a Set of Rows

# Composite Types



## Composite type

- set of named attributes (fields)
- same as a table row, but without constraints

## Creating a composite type

- explicit declaration of a new type
- implicitly when a table is created
- record: a placeholder composite type

## Using a composite type

- attributes as scalar values
- operations on composite type values: comparison, check for NULL, use with subqueries

3

A composite type represents a set of attributes, with each attribute having its own name and type. A composite type is similar to a table row in many ways. It is often called a *record* (similar to a *structure* in C-like languages).

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

A composite type is a database object. When it is declared, a new type is registered in the system catalog, making it a full-fledged SQL type. A table creation automatically produces a composite type with the same name. This type represents a row of the table. An important difference is that composite types do not have constraints.

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

Composite type attributes can be used as regular scalar values (although each attribute can also be of a composite type itself).

A composite type can be used just like any other SQL type; for example, you can create table columns of this type. Composite values can be compared, checked for NULL, used with subqueries in clauses like IN, ANY/SOME, ALL.

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

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

## Explicit Declaration of a Composite Type

The first way to introduce a composite type is to explicitly declare it.

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

Such a type can be used just as any other SQL type. For example, we can create a table that has some columns of this type:

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

Whether it is a good idea is not an easy question: there are no universal solutions here. In some cases, it can be quite useful; in other situations it's more convenient to follow the relational data model, i.e., move the entity represented by this type into a separate table and add references to this table. It enables you to avoid data redundancy (normalization) and simplify indexing (a composite type is likely to require an index on expression).

In general, PostgreSQL offers quite a lot of built-in data types, so the need for a custom type is unlikely to arise too often.

---

## Constructing Composite Type Values

Composite type values can be constructed in the form of a string, with all the attributes listed in brackets. Note that the attributes of the string type are enclosed in double quotes:

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

Another option is to use the ROW constructor:

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

If the composite type contains more than one field, you can omit the ROW keyword:

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

```
=> SELECT * FROM transactions;  
  
 account_id | debit      | credit      | date_entered  
-----+-----+-----+-----  
          1 |           | (100.00,EUR) | 2025-11-27  
          2 | (80.00,EUR) |           | 2025-11-27  
          3 | (20.00,EUR) |           | 2025-11-27  
(3 rows)
```

---

## Using Composite Type Attributes as Separate Values

Accessing a separate attribute of a composite type is virtually the same operation as accessing a table column, since a table row actually represents a composite type:

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

```

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

```

In most cases, the composite value has to be enclosed into brackets, e.g., to distinguish between a type attribute and a table column:

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

```

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

```

Or if you are using an expression:

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

```

amount
-----
 10.00
(1 row)

```

A composite value does not necessarily belong to a particular type, it can be a value of generic record pseudotype:

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

```

      row
-----
(10.00, EUR)
(1 row)

```

But can you access an attribute of such a value?

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

```

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

```

No, because attributes of such a type have no name.

---

## An Implicit Composite Type for Tables

In practice, composite types are typically used to facilitate the use of functions for table processing.

When a table is created, a composite type with same name is created implicitly. For example, seats in the cinema:

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

```
CREATE TABLE
```

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

```
INSERT 0 3
```

The \dT command hides such implicit types, but you can take a look at them in the pg\_type table if you like:

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

```

typtype
-----
c
(1 row)

```

Here c stands for a composite type.

---

## Operations on Composite Values

Composite type values can be compared with each other. It is done element by element (similar to string comparison, which is performed symbol by symbol):

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

line	number
A	42
B	1

(2 rows)

Beware multiple intricacies of using null values within records.

PostgreSQL also supports IS [NOT] NULL and IS [NOT] DISTINCT FROM predicates for composite values.

Composite types can be used in subqueries, which happens to be very convenient.

Let's add a table with tickets:

```
=> 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

Now we can write the following query to search for seats in tickets for a today's screening:

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

line	number
A	42

(1 row)

We would have to explicitly join tables if we could not use a subquery.

# Routine Arguments



A routine can take composite type arguments

Implementing computed fields

`table.column` and `column(table)` are interchangeable

Other options

views

GENERATED ALWAYS columns

5

Naturally, routines can take arguments of composite types.

It's worth noting that apart from the usual `table.column` notation, you can access a table column using the functional form: `column(table)`. It allows us to create computed fields by declaring a function that takes a composite value as an argument.

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

This approach is a bit odd because there is a more straightforward way to get the same outcome by using a view. The SQL standard also defines GENERATED ALWAYS columns, but their implementation in PostgreSQL does not fully comply with the standard yet: generated columns are STORED in a table instead of being generated on the fly. Such VIRTUAL columns are to be introduced in PostgreSQL 18.

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

## Composite Type Parameters

Let's declare a function that takes a composite value as an input parameter and returns a string with a seat number.

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

CREATE FUNCTION

Note that concatenation is stable in general, not immutable: casting some data types to a string can give different results depending on the current settings.

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

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

It comes in handy that such functions allow you to pass a table row as an argument:

```
=> 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)
```

We can also do without an asterisk:

```
=> 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)
```

The syntax allows calling a function as if it were a table column (and vice versa, you can access a column as if it were a function):

```
=> 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)
```

Using this syntax, you can use functions like table columns computed on the fly.

What if the table contains a column with the same name? Previously, the column would always have priority; starting from version 11, the choice depends on the syntactic form.

---

Clearly, you can get the same outcome by creating a view.

```
=> 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)
```



And starting from version 12, you can declare “true” computed columns as a part of a table. But instead of being computed on the fly as defined by the SQL standard, they are stored in the table:

```
=> 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)
```

If we later wish to define the value explicitly, we can just drop the expression:

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

ALTER TABLE

The data is still in the column, but the column is no longer computed.

```
=> 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 | | |
```

# Single Row Functions

Return a composite value

Usually called in SELECT list

When called within a FROM clause, return a one-row table

Functions can both take arguments of a composite type and return composite type values.

Functions are usually called in SELECT lists, but it is possible to call a function within a FROM clause, as if it were a one-row table.

## Functions Returning One Row

Let's create a function that constructs a table row from separate components and returns it.

Such a function can be declared as 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)
```

The returned result is of a composite type. It can be “unfolded” into a one-row table:

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

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

Column names and types are received from the definition of the seats composite type here.

Apart from calling a function in the SELECT list or as part of an expression, you can also call it in the FROM clause, as if it were a table:

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

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

As a result, we are getting a one-row table again.

By the way, can we use the same calling method for a function that returns a single (scalar) value?

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

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

Yes, it's also possible: we get a single column with a single row.

---

Another approach that we have already seen in the SQL. Functions lecture is to define output parameters.

Note that you do not have to manually construct the composite type from separate fields in the query; it will be done automatically:

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

DROP FUNCTION

```
=> CREATE FUNCTION seat(line INOUT text, number INOUT integer)
IMMUTABLE LANGUAGE sql
BEGIN ATOMIC SELECT line, number; END;
```

CREATE FUNCTION

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

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

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

```

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

```

We get the same outcome, but names and types of the columns are taken from the function input parameters, while the composite type itself remains anonymous.

---

And one more approach is to declare a function that returns the record pseudotype, which denotes a composite type in general, without specifying its structure.

```
=> 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)

```

But you won't be able to call such a function in the FROM clause not just because the return type is anonymous, but also because the number and types of the fields in the returned composite type are not known in advance (at the parsing stage):

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

```

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

```

In this case, you have to specify the exact structure of the composite type when calling a function:

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

```

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

```

You can use any of these three approaches when creating functions. But you should keep in mind the expected use cases from the very beginning: whether it will be convenient to use anonymous types and specify the structure of the type during function calls.

# Set Returning Functions



Declared as RETURNS SETOF or RETURNS TABLE

Can return multiple rows

Usually called in a FROM clause

Can be used as a view with arguments

very convenient when combined with function inlining

9

We have tried calling functions in a FROM clause, but have only seen one-row outputs so far. However, there is nothing stopping us from declaring functions that would return a set of rows: the so-called table functions or set returning functions (SRF).

It's only natural to call these functions in a FROM clause, turning them into a pseudo-views to some extent. (Technically, PostgreSQL allows calling such functions in SELECT lists as well, but it is not recommended.)

Like with regular functions, the planner can sometimes inline the function body into the main query. It allows creating “views with arguments” without additional overhead.

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

## Set Returning Functions

Let's create a function that returns all seats in a rectangular cinema hall of the specified size.

```
=> 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

The key difference is the SETOF usage. In this case, instead of returning the first row of the last query, as usual, the function returns all the rows of the last query.

```
=> 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)

Instead of SETOF seats you can also use 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

But as we have already seen, in this case you have to specify the structure of the composite type when calling a function:

```
=> 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)

Or we could declare a function with output parameters. But SETOF record would still be required to show that the function returns a set of rows, not a single row:

```
=> 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)

Another equivalent way to declare a set returning function (which is even defined by the SQL standard) is to use the TABLE keyword:

```
=> 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)

It is sometimes useful to enumerate the rows returned by the query, in the order they were received from the function. There is a special clause for that:

```
=> 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)

When a function is used in a FROM clause, the LATERAL keyword is assumed to implicitly precede it, which allows this function to access columns of the tables that were mentioned in the query to the left of the function. It can sometimes simplify query definitions.

For example, let's create a function that distributes seats in the cinema like in an amphitheatre, with front rows having fewer seats than back rows:

```
=> 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)

It's interesting that you can call a function returning a set of rows as part of the SELECT list:

```
=> 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)

It seems logical in some cases, but occasionally the result can surprise you. For example, how many rows will be returned by the following query?

```
=> 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)

We get six rows, while prior to version 10 we would get the least common multiple of the number of rows returned by each function (12 in this case).

What's even worse is that the query can return fewer rows than expected if the function returns no rows when passed some particular parameters.

So using this calling method is not recommended.

## Functions as Views with Parameters

As we have already seen, a function can be used in the FROM clause, as if it were a table or a view. We can also pass additional parameters in this case, which is sometimes very convenient.

The only issue with this approach is that when a function is called (Function Scan), the queries in it must be executed completely first, and additional conditions defined in the query can only be applied after that.



```

=> 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)

```

It could become a problem if the function performed a long and complex query.

In some cases, a function body can be inlined, i.e., inserted into the calling query. The requirements for set returning functions are more relaxed. The main restrictions are:

- the function must be written in the SQL language
- the function itself must not be volatile and must not call other volatile functions
- the function must not be STRICT
- the function body must contain only one SELECT statement (although it can introduce a complex query)

In this case, we did not specify the volatility category when creating the function, so it was implicitly declared volatile.

```

=> 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)

```

There is virtually no function call now, and the condition is inserted into the query itself, which is more efficient.

- Composite types combine values of other types
- Simplify and enrich function operations on tables
- Implement computed fields and views with parameters
- Functions can return multiple rows



1. Create a function `onhand_qty` to calculate books in stock.  
The function takes a composite type parameter (`books`) and returns an integer number.  
Use this function in the `catalog_v` view as a computed field.  
Verify that the application can now display the number of books.
2. Create a set returning function `get_catalog` for book search.  
The function takes values from the search fields (author, book title, in stock) and returns matching books in the `catalog_v` format.  
Verify that you can now search for books and browse in the Store tab.

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  
)
```

The obvious solution is to use the existing `catalog_v` view, just with some row filters. But this view displays book titles and authors in the same field, and authors' names are abbreviated. Clearly, searching for "Reuel" in the "J. R. R. Tolkien" field will yield no results.

The `get_catalog` function could repeat the query from the `catalog_v` view, but it is code duplication, which is a bad practice. So you should extend the `catalog_v` view by adding the following fields: the book title and the full list of authors.

Verify that the empty fields in the form are handled correctly. When calling the `get_catalog` function, does the client pass empty strings or null values?

## Task 1. The onhand\_qty Function

```
=> 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
```

## Task 2. The get\_catalog Function

Extend the catalog\_v view by adding book titles and the full list of authors (the application ignores extra fields).

This function returns the full list of authors:

```
=> CREATE FUNCTION authors(book books) RETURNS text
STABLE LANGUAGE sql
BEGIN ATOMIC
    SELECT string_agg(
        a.first_name ||
        coalesce(' ' || nullif(a.middle_name,''), ' ') || ' ' ||
        a.last_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
```

Let's use this function in the catalog\_v view. The view already exists, but we will recreate it with a different column order and use a new query:

```
=> 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
```

The get\_catalog function now uses the extended view:

```

=> 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. Create a function that converts a string representation of a hexadecimal number into a regular integer number.  
For example: `convert('FF') → 255`
2. Extend this function with an optional argument that defines the base of a numeral system (16 by default).  
For example: `convert('0110', 2) → 6`
3. The *generate\_series* set returning function does not work with string types. Create your own function that generates string sequences of uppercase Latin letters.

1. For example:

`convert('FF') → 255`

To solve this problem, you can use a `regexp_split_to_table` set returning function, upper and reverse functions, and `WITH ORDINALITY` clause.

Another option is to use a recursive query.

You can check the implementation using hexadecimal constants:

```
SELECT X'FF'::integer;
```

2. For example:

`convert('0110', 2) → 6`

3. Assume that the input strings have the same length. For example:

`generate_series('AA', 'ZZ') →`

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

## 1. A Hexadecimal System Function

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

```
CREATE DATABASE
```

```
=> \c sql_row
```

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

Define a function for a single digit first:

```
=> 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
```

Now, the main function:

```
=> 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. A Function for Any Base

Assume the base is from 2 to 36: a digit may be a number from 0 to 9 or a letter from A to Z. In this case, the changes are minimal.

```
=> 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)
```

Note that in PostgreSQL 16 and onwards, integer constants can be not only decimal, but also binary, hexadecimal, and octal. This feature comes from the modern SQL standard:

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

```

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

```

### 3. The generate\_series Function for Strings

First, create two auxiliary functions that convert a string to a numeric representation and vice versa.

The first one is very similar to the function from the previous assignment:

```

=> 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

The inverse function will use a recursive query:

```

=> 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)

```

Now, create generate\_series for strings using generate\_series for integers.

```

=> 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