Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

Contact Us
Please send your feedback to: edu@postgrespro.ru

Agenda

Composite types and how to work with them

Composite type parameters

Functions returning a single row

Functions returning a set of rows

Composite Types

A composite type

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

Creation

- an explicit declaration of a new type
- implicit creation together with a table
- the record type, which serves as a placeholder for composite types

Usage

- attributes as scalar values
- operations on composite type values: comparison, check for `NULL`, usage in subqueries

A composite type represents a set of attributes, with each attribute having its own name and type. A composite type can be compared to a table row. It is often called a “record” (or a “structure” in C-like languages).

https://postgrespro.com/docs/postgresql/12/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 the row of the table; an important difference is that composite types do not have constraints.

https://postgrespro.com/docs/postgresql/12/sql-createtype

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

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

https://postgrespro.com/docs/postgresql/12/functions-comparisons

https://postgrespro.com/docs/postgresql/12/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's 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) | 2021-10-19
          2 | (80.00,EUR) |              | 2021-10-19
          3 | (20.00,EUR) |              | 2021-10-19
(3 rows)
```

Using Composite Type Attributes as Scalar 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 some 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 an indefinite value of the 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 data entries.

PostgreSQL also supports IS [NOT] NULL and IS [NOT] DISTINCT FROM clauses 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 movie:

```
=> 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.

Composite Type Parameters

A function can take parameters of composite types

Implementing computed fields using functions

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

Other ways to set up computed fields

- views
- `GENERATED ALWAYS` columns

Naturally, functions can take parameters of composite types.

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

https://postgrespro.com/docs/postgresql/12/xfunc-sql

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

https://postgrespro.com/docs/postgresql/12/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
AS $$
    SELECT seat.line || seat.number;
$$ IMMUTABLE LANGUAGE sql;
```

```
CREATE FUNCTION
```

Note that concatenation is normally stable, 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 the actual table row as a parameter:

```
=> 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 table 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 computed columns proper when creating a table. But instead of being computed on the fly as defined by the SQL standard, they are simply 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
```

```
=> 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)
```

One-Row Functions

Return a composite value

Are usually called in the context of `SELECT` lists

When called in the `FROM` clause, return a one-row table

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

Functions are usually called in the context of `SELECT` lists.

But it is also possible to call a function in the `FROM` clause, as if it were a one-row table.

Functions with a Single Return Value

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

Such a function can be declared as RETURNS seats:

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

```
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 scalar value?

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

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

Yes, it's also possible.

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)
AS $$
    SELECT line, number;
$$ IMMUTABLE LANGUAGE sql;
```

```
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
AS $$
    SELECT line, number;
$$ IMMUTABLE LANGUAGE sql;
```

```
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 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 seats(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

Are declared as `RETURNS SETOF` or `RETURNS TABLE`

Can return several rows

Are usually called in the `FROM` clause

Can be used as a view with parameters

- it is especially convenient when combined with function inlining

As we know, functions can be called in the `FROM` clause, but we have only seen one-row results so far. It is certainly more interesting to have functions that return sets of rows, and we can declare them as well.

It is natural to call set-returning functions in the `FROM` clause; they can be considered as some kind of a view. (Formally, PostgreSQL allows calling such functions in `SELECT` lists as well, but it is not recommended.)

Like with regular functions, the planner can sometimes perform inlining, i.e., insert the function body into the main query. It allows creating “views with parameters” without additional overhead.

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
AS $$
    SELECT chr(line+64), number
    FROM generate_series(1,max_line) AS lines(line),
         generate_series(1,max_number) AS numbers(number);
$$ IMMUTABLE LANGUAGE sql;
```

```
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
AS $$
    SELECT chr(line+64), number
    FROM generate_series(1,max_line) AS lines(line),
         generate_series(1,max_number) AS numbers(number);
$$ IMMUTABLE LANGUAGE sql;
```

```
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 seats(line text, number integer);
```

```
 line | 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 line text, OUT number integer
)
RETURNS SETOF record
AS $$
    SELECT chr(line+64), number
    FROM generate_series(1,max_line) AS lines(line),
         generate_series(1,max_number) AS numbers(number);
$$ IMMUTABLE LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
=> 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)
```

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(line text, number integer)
AS $$
    SELECT chr(line+64), number
    FROM generate_series(1,max_line) AS lines(line),
         generate_series(1,max_number) AS numbers(number);
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
=> 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)
```

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;
```

```
 line | 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(line text, number integer)
AS $$
    SELECT chr(line+64), number
    FROM generate_series(1,max_line) AS lines(line), -- <--+
         generate_series(1, --                           |
                         line ---------------------------+
                        ) AS numbers(number);
$$ IMMUTABLE LANGUAGE sql;
```

```
CREATE FUNCTION
```

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

```
 line | 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 all 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 a Function Scan must be completed before additional conditions defined in the query can be applied.

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

          QUERY PLAN
------------------------------
 Function Scan on rect_hall
   Filter: (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, e.g., 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 operator (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 line = 'A';
```

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

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

Summary

Composite types combine values of other types

Simplify and enrich function operations on tables

Implement computed fields and views with parameters

Practice

1. Create a function `onhand_qty` to calculate books in stock.
   The function takes a parameter of a composite type (`books`) and returns an integer number.

   Use this function in the `catalog_v` view as a computed field.

   Check 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.

   Check that you can now browse and search for books in "Store."

Task 1.

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

Task 2.

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

To solve this problem, it is tempting to use the already available `catalog_v` view and simply filter out some rows. But this view displays book titles and authors in the same field, and authors' middle names are abbreviated. It is clear that the search for "Klapka" in the "Jerome K. Jerome" field will not return any results.

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

Check 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 OR REPLACE FUNCTION onhand_qty(book books) RETURNS integer
AS $$
    SELECT coalesce(sum(o.qty_change),0)::integer
    FROM operations o
    WHERE o.book_id = book.book_id;
$$ STABLE LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
=> DROP VIEW IF EXISTS catalog_v;
```

```
DROP VIEW
```

```
=> CREATE 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

Let's extend the catalog_v view by adding book titles and the full list of authors (the application ignores unknown fields).

Here is the function that returns the full list of authors:

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

```
CREATE FUNCTION
```

Let's use this function in the catalog_v view:

```
=> 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 OR REPLACE FUNCTION get_catalog(
    author_name text,
    book_title text,
    in_stock boolean
)
RETURNS TABLE(book_id integer, display_name text, onhand_qty integer)
AS $$
    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;
$$ STABLE LANGUAGE sql;

CREATE FUNCTION
```

Practice

1. Create a function that converts a string representation of a hexadecimal number into a regular integer number.
2. Extend this function with an optional parameter that defines the base of a numeral system (16 by default).
3. The set-returning function `generate_series` does not support text types. Create your own function that generates string sequences of uppercase Latin letters.

Task 1. For example:

```
convert('FF') → 255
```

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

Another option is to use a recursive query.

You can check the implementation using hexadecimal constants:

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

Task 2. For example:

```
convert('0110',2) → 6
```

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

```
generate_series('AA','ZZ') →
 → 'AA'
   'AB'
   'AC'
   ...
   'ZY'
   'ZZ'
```

Task 1. A Function for the Hexadecimal Numeral System

For convenience, let's first declare a function for a single digit.

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

```
CREATE FUNCTION
```

Now let's create the main function:

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

```
CREATE FUNCTION
```

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

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

Task 2. A Function for Any Numeral System

Let's assume that the numeral system base is from 2 to 36, i.e., we can use digits from 0 to 9 and letters from A to Z to write a number. In this case, the required changes are minimal.

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

```
DROP FUNCTION
```

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

```
CREATE FUNCTION
```

```
=> SELECT convert('0110',2), convert('0FF'), convert('Z',36);
```

```
 convert | convert | convert
---------+---------+---------
       6 |     255 |      35
(1 row)
```

Task 3. The generate_series Function for Strings

First, let's create some auxiliary functions that convert a string into a numeric representation and back.

The first function looks very similar to the one in the previous assignment:

```
=> CREATE FUNCTION text2num(s text) RETURNS integer
AS $$
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;
$$ IMMUTABLE LANGUAGE sql;

CREATE FUNCTION
```

To create the inverse function, let's use a recursive query:

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

CREATE FUNCTION

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

 num2text
----------
 ABC
(1 row)
```

Now the generate_series function for strings can be rewritten using the generate_series function for integer numbers.

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

CREATE FUNCTION

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

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