

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

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.

What are Cursors

Declaration and Opening

Operations with Cursors

Loops over Cursors and Query Results

Passing a Cursor to Clients

What are Cursors



A cursor implies iterative processing

- a full output takes too much memory
- only some part of the output is needed, but its size is unknown beforehand
- allows the client to control the output size
- you really need row-by-row processing (you usually don't)

3

We have already learned about the concept of cursors in the Architecture. PostgreSQL overview topic. In that topic, cursors were presented as a server feature, and we explained how to access them using SQL. Now let's talk about how to use cursors in PL/pgSQL.

Why do we need cursors at all? SQL is a declarative language; first and foremost, it is designed to work with sets of rows, which is its strength and advantage. Being a procedural language, PL/pgSQL has to work with data row by row, using explicit loops. This can be achieved via cursors.

For example, a full SELECT result may take up too much memory, so it has to be processed piece by piece, or the required size of the output is unknown beforehand, for example, the query must be stopped at some point, or you need to let the client manage the output.

(Although row-by-row processing may be required from time to time, the same outcome can often be achieved using pure SQL, with simpler and more efficient code.)

Declaration and Opening

Unbound cursor variables

- a variable of the *refcursor* type is declared
- the actual query is specified when the cursor is being opened

Bound cursor variables

- the query is specified at declaration time (possibly with parameters)
- the arguments are passed when the cursor is opened

Not supported in SQL

Features

- the value of the cursor variable is the cursor name
(can be specified explicitly or generated automatically)
- PL/pgSQL variables become implicit parameters of the query
(their arguments are filled in when the cursor is opened)
- the query is prepared

4

SQL uses a single DECLARE command that both declares and opens a cursor. In PL/pgSQL, these are two different steps. Besides, there are so-called cursor variables that are used for cursor access. These variables have the *refcursor* type and virtually contain the name of the cursor (if you do not provide this name explicitly, PL/pgSQL ensures that it is unique).

A cursor variable can be declared without being bound to a particular query. Then you have to specify the query when you open the cursor.

Alternatively, you can specify the query (including parameters) when declaring a variable. Then, you'll only have to pass the arguments when opening the cursor.

These methods are equivalent; which one to use is a matter of taste. Both bound and unbound cursor variables are initialized only when the cursor is being opened. In both cases, a query can have implicit parameters, which are derived from PL/pgSQL variables.

A query opened with a cursor is prepared automatically.

<https://postgrespro.com/docs/postgresql/16/plpgsql-cursors#PLPGSQL-CURSOR-DECLARATIONS>

<https://postgrespro.com/docs/postgresql/16/plpgsql-cursors#PLPGSQL-CURSOR-OPENING>

Declaration and Opening

Create a table:

```
=> CREATE TABLE t(id integer, s text);
```

CREATE TABLE

```
=> INSERT INTO t VALUES (1, 'One'), (2, 'Two'), (3, 'Three');
```

INSERT 0 3

Unbound variable:

```
=> DO $$
DECLARE
    -- declare a variable
    cur refcursor;
BEGIN
    -- bind with a query and open
    OPEN cur FOR SELECT * FROM t;
END
$$;
```

DO

Bound variable: the query is added during declaration. The variable cur is still of the refcursor type.

```
=> DO $$
DECLARE
    -- declare a variable and bind
    cur CURSOR FOR SELECT * FROM t;
BEGIN
    -- open the cursor
    OPEN cur;
END
$$;
```

DO

For bound variables, the query may be parameterized.

Note the name disambiguation in this example and the next one:

```
=> DO $$
DECLARE
    -- declare a variable and bind
    cur CURSOR(id integer) FOR SELECT * FROM t WHERE t.id = cur.id;
BEGIN
    -- pass arguments and open the cursor
    OPEN cur(1);
END
$$;
```

DO

PL/pgSQL variables also serve as (implicit) cursor parameters.

```
=> DO $$
<<local>>
DECLARE
    id integer := 3;
    -- declare a variable and bind
    cur CURSOR FOR SELECT * FROM t WHERE t.id = local.id;
BEGIN
    id := 1;
    -- open the cursor (the value of id is passed at this moment)
    OPEN cur;
END
$$;
```

DO

You can use not only the SELECT command, but also any other command that returns a result (for example, INSERT, UPDATE, DELETE with the RETURNING keyword) as a query.

Operations with Cursors

Fetch

row by row only

in SQL
the batch size
is configurable

Access the current row

for simple queries only (one table, no grouping or sorting)

Processing is usually performed in a loop

a FOR loop over a cursor

a FOR loop over a query without an explicitly declared cursor

Close

explicitly or automatically at transaction end

in SQL
DECLARE
WITH HOLD

PL/pgSQL allows fetching data from a cursor row by row only. It is done via the `FETCH INTO` command.

If the query is simple enough (it works with one table, without grouping or sorting), it is possible to access the current row of the cursor within commands such as `UPDATE` or `DELETE`.

Procedural processing implies looping through data. The rows returned by a cursor can be iterated through and processed using control commands that are already familiar to us. But since such loops are required quite often, PL/pgSQL offers a special flavor of the `FOR` command that implements them. We have already seen `FOR` working with integers in the PL/pgSQL. Overview and Programming Structures section; this one works with cursors. Moreover, there is one more flavor of the `FOR` loop that does not require cursor declaration at all: the query itself is specified within the command.

A cursor can be explicitly closed by the `CLOSE` command, but it will be closed anyway once the transaction is complete (in SQL, a cursor can remain open even after the transaction has finished if you have specified `WITH HOLD`).

<https://postgrespro.com/docs/postgresql/16/plpgsql-cursors#PLPGSQL-CURSOR-USING>

<https://postgrespro.com/docs/postgresql/16/plpgsql-cursors#PLPGSQL-CURSOR-FOR-LOOP>

Operations with Cursors

The FETCH command reads the next row from the cursor position. To move to the next row without reading anything, use the MOVE command.

What will this example return?

```
=> DO $$
DECLARE
    cur refcursor;
    rec record; -- you can use several scalar variables instead
BEGIN
    OPEN cur FOR SELECT * FROM t ORDER BY id;
    MOVE cur;
    FETCH cur INTO rec;
    RAISE NOTICE '%', rec;
    CLOSE cur;
END
$$;

NOTICE:  (2,Two)
DO
```

Usually, such data retrieval is done within a loop. For example:

```
=> DO $$
DECLARE
    cur refcursor;
    rec record;
BEGIN
    OPEN cur FOR SELECT * FROM t;
    LOOP
        FETCH cur INTO rec;
        EXIT WHEN NOT FOUND; -- FOUND: was the row retrieved?
        RAISE NOTICE '%', rec;
    END LOOP;
    CLOSE cur;
END
$$;

NOTICE:  (1,One)
NOTICE:  (2,Two)
NOTICE:  (3,Three)
DO
```

PL/pgSQL offers a cursor FOR loop that does the same thing, but with much less code:

```
=> DO $$
DECLARE
    cur CURSOR FOR SELECT * FROM t;
    -- loop variable is not declared
BEGIN
    FOR rec IN cur LOOP -- cur should be bound to a query
        RAISE NOTICE '%', rec;
    END LOOP;
END
$$;

NOTICE:  (1,One)
NOTICE:  (2,Two)
NOTICE:  (3,Three)
DO
```

Moreover, you can forego explicit cursor commands altogether if the loop is all you really need.

Parentheses around the query are optional, but convenient.

```
=> DO $$
DECLARE
    rec record; -- should be declared explicitly
BEGIN
    FOR rec IN (SELECT * FROM t) LOOP
        RAISE NOTICE '%', rec;
    END LOOP;
END
$$;
```

```
NOTICE: (1,One)
NOTICE: (2,Two)
NOTICE: (3,Three)
DO
```

As with any loop, you can place a label here, which can be useful in nested loops.

What will be output here?

```
=> DO $$
DECLARE
    rec_outer record;
    rec_inner record;
BEGIN
    <<outer>>
    FOR rec_outer IN (SELECT * FROM t ORDER BY id) LOOP
        <<inner>>
        FOR rec_inner IN (SELECT * FROM t ORDER BY id) LOOP
            EXIT outer WHEN rec_inner.id = 3;
            RAISE NOTICE '%, %', rec_outer, rec_inner;
        END LOOP inner;
    END LOOP outer;
END
$$;
```

```
NOTICE: (1,One), (1,One)
NOTICE: (1,One), (2,Two)
DO
```

After the loop is done, the FOUND variable will tell if at least one row got processed:

```
=> DO $$
DECLARE
    rec record;
BEGIN
    FOR rec IN (SELECT * FROM t WHERE false) LOOP
        RAISE NOTICE '%', rec;
    END LOOP;
    RAISE NOTICE 'Was there at least one iteration? %', FOUND;
END
$$;
```

```
NOTICE: Was there at least one iteration? f
DO
```

The current row of the cursor bound to a simple query (single table, no grouping or sorting) can be referenced with the CURRENT OF clause. One typical usecase for this is batch processing tasks and updating their status live.

```
=> DO $$
DECLARE
    cur refcursor;
    rec record;
BEGIN
    OPEN cur FOR SELECT * FROM t
        FOR UPDATE; -- the rows are locked as they are processed
    LOOP
        FETCH cur INTO rec;
        EXIT WHEN NOT FOUND;
        UPDATE t SET s = s || ' (processed)' WHERE CURRENT OF cur;
    END LOOP;
    CLOSE cur;
END
$$;
```

DO

```
=> SELECT * FROM t;

id |          s
----+-----
 1 | One (processed)
 2 | Two (processed)
 3 | Three (processed)
(3 rows)
```

Note that CURRENT OF does not work with FOR loops over queries, because they do not use the cursor explicitly. You can explicitly specify a unique table key ('WHERE id = rec.id'), of course, but CURRENT OF is faster and does not need an index.

With all that said, a single SQL statement can do the job better and faster than a set of loops. Loops are often overused because of their familiar, procedural programming vibe. But databases are built different.

For example:

```
=> BEGIN;
DO $$
DECLARE
    rec record;
BEGIN
    FOR rec IN (SELECT * FROM t) LOOP
        RAISE NOTICE '%', rec;
        DELETE FROM t WHERE id = rec.id;
    END LOOP;
END
$$;
ROLLBACK;

BEGIN
NOTICE:  (1,"One (processed)")
NOTICE:  (2,"Two (processed)")
NOTICE:  (3,"Three (processed)")
DO
ROLLBACK
```

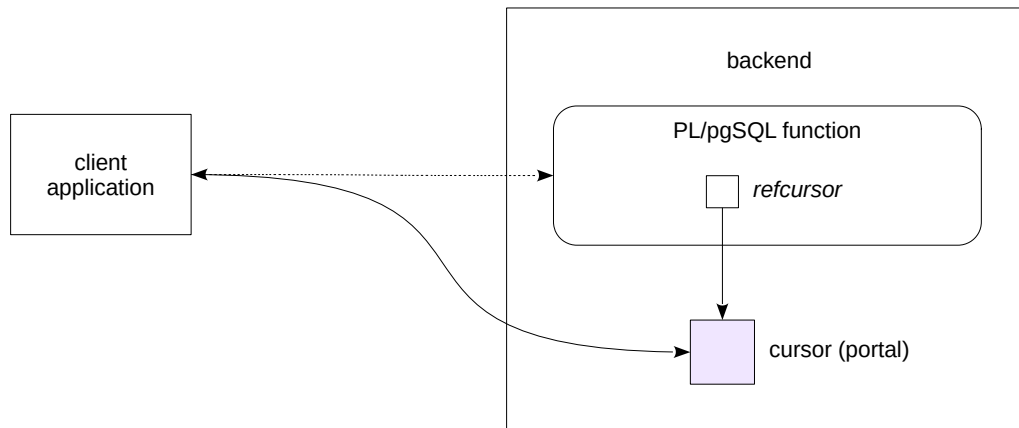
This loop is equivalent to one simple SQL command:

```
=> BEGIN;
DELETE FROM t RETURNING *;
ROLLBACK;

BEGIN
id |          s
-----+-----
  1 | One (processed)
  2 | Two (processed)
  3 | Three (processed)
(3 rows)

DELETE 3
ROLLBACK
```

Passing a Cursor to Clients



A PL/pgSQL cursor variable (of the *refcursor* type) contains the name of an open SQL cursor. To denote the memory allocated in the backend for keeping the cursor state, the term *portal* is used.

This way, a PL/pgSQL function can open a cursor and return its name to the client. Then, the client will be able to work with the cursor as if it has been opened by this client, but will have access only to the provided data. It adds one more way of setting up the interface between the database and the application.

Passing a Cursor to Clients

Open a cursor and print the value of the cursor variable:

```
=> DO $$  
DECLARE  
    cur refcursor;  
BEGIN  
    OPEN cur FOR SELECT * FROM t;  
    RAISE NOTICE '%', cur;  
END  
$$;  
  
NOTICE: <unnamed portal 14>  
DO
```

This is the name of the cursor (portal) that was opened on the server. It was generated automatically.

You can name it explicitly if you want (but it must be unique for the session):

```
=> DO $$  
DECLARE  
    cur refcursor := 'cursor12345';  
BEGIN  
    OPEN cur FOR SELECT * FROM t;  
    RAISE NOTICE '%', cur;  
END  
$$;  
  
NOTICE: cursor12345  
DO
```

With this, you can create a function that takes a cursor and returns its name:

```
=> CREATE FUNCTION t_cur() RETURNS refcursor  
AS $$  
DECLARE  
    cur refcursor;  
BEGIN  
    OPEN cur FOR SELECT * FROM t;  
    RETURN cur;  
END  
$$ VOLATILE LANGUAGE plpgsql;  
  
CREATE FUNCTION
```

The client starts a transaction...

```
=> BEGIN;
```

```
BEGIN
```

...calls the function, gets the cursor name...

```
=> SELECT t_cur();  
  
      t_cur  
-----  
<unnamed portal 15>  
(1 row)
```

...and starts reading from it (the quotation marks are necessary because of the special characters in the name):

```
=> FETCH "<unnamed portal 15>";  
  
 id |          s  
----+-----  
  1 | One (processed)  
(1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

You can let the client set the cursor name:

```
=> DROP FUNCTION t_cur();
```

DROP FUNCTION

```
=> CREATE FUNCTION t_cur(cur refcursor) RETURNS void
AS $$
BEGIN
    OPEN cur FOR SELECT * FROM t;
END
$$ VOLATILE LANGUAGE plpgsql;
```

CREATE FUNCTION

The client code is simplified:

```
=> BEGIN;
```

BEGIN

```
=> SELECT t_cur('cursor12345');
```

```
  t_cur
-----
```

(1 row)

```
=> FETCH cursor12345;
```

```
 id |      s
----+-----
```

```
  1 | One (processed)
```

(1 row)

```
=> COMMIT;
```

COMMIT

The function can also return several open cursors using OUT parameters. This makes it possible for the client to read data from multiple tables within one function call.

An alternative is to bundle the data from different sources into a JSON or XML on the server and then return it. This is covered in the DEV2 course.

Takeaways



A cursor allows fetching and processing data row by row

A FOR loop can simplify cursor handling

Processing data in loops is common for procedural languages, but should not be overused



1. Modify the *book_name* function: if the book has more than two authors, the title should include only the first two, while the rest are to be replaced with “et al.”
Check that the function works fine in SQL and in the application.
2. Try rewriting the *book_name* function in SQL.
Which implementation do you prefer: PL/pgSQL or SQL?

1. For example:

101 Famous Poems. Alexander S. Pushkin, William Shakespeare, Ivan A. Bunin →
→ 101 Famous Poems. Alexander S. Pushkin, William Shakespeare, et al.

1. book_name Function (Author Abbreviations)

A more universal function can be created that takes an additional parameter: the maximum number of authors in a book name.

Since the function changes its signature (the number and/or types of input parameters), it must first be deleted and then recreated. In this case, the function has a dependent object, the catalog_v view, in which it is used. The view will also have to be recreated (in reality, all these actions must be performed in a single transaction for the changes to take effect atomically).

```
=> DROP FUNCTION book_name(integer,text) CASCADE;
```

NOTICE: drop cascades to 2 other objects

DETAIL: drop cascades to view catalog_v

drop cascades to function get_catalog(text,text,boolean)

DROP FUNCTION

```
=> CREATE FUNCTION book_name(
    book_id integer,
    title text,
    maxauthors integer DEFAULT 2
)
RETURNS text
AS $$
DECLARE
    r record;
    res text := shorten(title);
BEGIN
    IF (right(res, 3) != '...') THEN res := res || '.'; END IF;
    res := res || ' ';
    FOR r IN (
        SELECT a.last_name, a.first_name, a.middle_name, ash.seq_num
        FROM authors a
            JOIN authorship ash ON a.author_id = ash.author_id
        WHERE ash.book_id = book_name.book_id
        ORDER BY ash.seq_num
    )
    LOOP
        EXIT WHEN r.seq_num > maxauthors;
        res := res || author_name(r.last_name, r.first_name, r.middle_name) || ', ';
    END LOOP;
    res := rtrim(res, ', ');
    IF r.seq_num > maxauthors THEN
        res := res || ' et al.';
    END IF;
    RETURN res;
END
$$ STABLE LANGUAGE plpgsql;
```

CREATE FUNCTION

```
=> CREATE OR REPLACE 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

```
=> SELECT book_id, display_name FROM catalog_v;
```

book_id	display_name
7	101 Famous Poems. Alexander S. Pushkin, Ivan A. Bunin et al.
4	Dark Avenues. Ivan A. Bunin
3	Good Omens. Neil Gaiman, Terry Pratchett
2	Romeo and Juliet. William Shakespeare
1	The Tale of Tsar Saltan. Alexander S. Pushkin
6	Three Men in a Boat (To Say Nothing of the... Jerome K. Jerome
5	Travels into Several Remote Nations of the... Jonathan Swift

(7 rows)

Also remember to recreate the get_catalog function which was deleted:

```
=> 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
```

2. A Pure SQL Version

```
=> CREATE OR REPLACE FUNCTION book_name(
    book_id integer,
    title text,
    maxauthors integer DEFAULT 2
)
RETURNS text
STABLE LANGUAGE sql
BEGIN ATOMIC
SELECT shorten(book_name.title) ||
    CASE WHEN (right(shorten(book_name.title), 3) != '...') THEN '. '::text ELSE ' ' END ||
    string_agg(
        author_name(a.last_name, a.first_name, a.middle_name), ', '
        ORDER BY ash.seq_num
    ) FILTER (WHERE ash.seq_num <= maxauthors) ||
    CASE
        WHEN max(ash.seq_num) > maxauthors THEN ' et al.'
        ELSE ''
    END
FROM   authors a
    JOIN authorship ash ON a.author_id = ash.author_id
WHERE  ash.book_id = book_name.book_id;
END;

CREATE FUNCTION
```

```
=> SELECT book_id, display_name FROM catalog_v;

book_id | display_name
-----+-----
7 | 101 Famous Poems. Alexander S. Pushkin, Ivan A. Bunin et al.
4 | Dark Avenues. Ivan A. Bunin
3 | Good Omens. Neil Gaiman, Terry Pratchett
2 | Romeo and Juliet. William Shakespeare
1 | The Tale of Tsar Saltan. Alexander S. Pushkin
6 | Three Men in a Boat (To Say Nothing of the... Jerome K. Jerome
5 | Travels into Several Remote Nations of the... Jonathan Swift
(7 rows)
```


1. Energy expenses must be distributed between different departments in proportion to their headcount (the list of departments is stored in a table).
Create a function that takes the total energy cost as an argument and saves the distributed expenses in different table rows. The values are rounded to cents; the sum of expenses of all departments must exactly match the total cost.
2. Create a set returning function that simulates merge sort. The function should take two cursor variables; both cursors are already open and return sorted integers in non-decreasing order. It must return a single sorted sequence of integers from both sources.

1. We can use the following table:

```
CREATE TABLE depts(
  id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  employees integer,
  expenses numeric(10,2)
);
INSERT INTO depts(employees) VALUES (10),(10),(10);
```

A possible implementation:

```
FUNCTION distribute_expenses(amount numeric) RETURNS void;
```

If 100.00 is taken as an argument, the expected result is:

```
expenses
-----
33.33
33.34
33.33
```

2. A possible implementation:

```
FUNCTION merge(c1 refcursor, c2 refcursor) RETURNS SETOF integer;
```

For example, if the first cursor returns the sequence 1, 3, 5, and the second cursor returns the sequence 2, 3, 4, the expected result is as follows:

```
merge
-----
1
2
3
3
4
5
```

1. Expenses Distribution

```
=> CREATE DATABASE plpgsql_cursors;
```

```
CREATE DATABASE
```

```
=> \c plpgsql_cursors
```

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

Table:

```
=> CREATE TABLE depts(  
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
    employees integer,  
    expenses numeric(10,2)  
);
```

```
CREATE TABLE
```

```
=> INSERT INTO depts(employees) VALUES (20),(10),(30);
```

```
INSERT 0 3
```

Function:

```
=> CREATE FUNCTION distribute_expenses(amount numeric) RETURNS void  
AS $$  
DECLARE  
    depts_cur CURSOR FOR  
        SELECT employees FROM depts FOR UPDATE;  
    total_employees numeric;  
    expense numeric;  
    rounding_err numeric := 0.0;  
    cent numeric;  
BEGIN  
    SELECT sum(employees) FROM depts INTO total_employees;  
    FOR dept IN depts_cur LOOP  
        expense := amount * (dept.employees / total_employees);  
        rounding_err := rounding_err + (expense - round(expense,2));  
  
        cent := round(rounding_err,2);  
        expense := expense + cent;  
        rounding_err := rounding_err - cent;  
  
        UPDATE depts SET expenses = round(expense,2)  
        WHERE CURRENT OF depts_cur;  
    END LOOP;  
END  
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Check:

```
=> SELECT distribute_expenses(100.0);
```

```
distribute_expenses  
-----
```

```
(1 row)
```

```
=> SELECT * FROM depts;
```

```
id | employees | expenses  
---+-----+-----  
 1 |         20 |    33.33  
 2 |         10 |    16.67  
 3 |         30 |    50.00  
(3 rows)
```

Of course, other algorithms are also possible, for example, assigning all rounding errors to a single row, etc.

The DEV2 course proposes another solution using custom aggregate functions.

2. Merging Sorted Sets

This implementation assumes that numbers cannot have undefined NULL values.

```
=> CREATE FUNCTION merge(c1 refcursor, c2 refcursor)
RETURNS SETOF integer
AS $$
DECLARE
    a integer;
    b integer;
BEGIN
    FETCH c1 INTO a;
    FETCH c2 INTO b;
    LOOP
        EXIT WHEN a IS NULL AND b IS NULL;
        IF a < b OR b IS NULL THEN
            RETURN NEXT a;
            FETCH c1 INTO a;
        ELSE
            RETURN NEXT b;
            FETCH c2 INTO b;
        END IF;
    END LOOP;
END
$$ VOLATILE LANGUAGE plpgsql;
```

CREATE FUNCTION

Test:

```
=> BEGIN;
```

BEGIN

```
=> DECLARE c1 CURSOR FOR
        SELECT * FROM (VALUES (1),(3),(5));
```

DECLARE CURSOR

```
=> DECLARE c2 CURSOR FOR
        SELECT * FROM (VALUES (2),(3),(4));
```

DECLARE CURSOR

```
=> SELECT * FROM merge('c1','c2');
```

```
merge
-----
    1
    2
    3
    3
    4
    5
(6 rows)
```

```
=> COMMIT;
```

COMMIT

```
=> \c postgres
```

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

```
=> DROP DATABASE plpgsql_cursors;
```

DROP DATABASE