Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

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

Agenda

Motivation

Declaration and opening

Operations with cursors

Loops over cursors and query results

Passing a cursor to clients

Motivation

A cursor implies iterative processing

- a full result set takes too much memory
- only some part of the result set is needed, but its size is unknown in advance
- it is required to allow the client to control the selection
- row-by-row processing is really needed (such cases are quite rare)

We have already learned about the concept of cursors in “Architecture. A General Overview of PostgreSQL.” In that lecture, 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 the PL/pgSQL language.

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. It can be achieved via cursors.

For example, a full SELECT result can take too much memory, so it has to be processed piece by piece. Or the size of the result set is unknown in advance, i.e., it is required to stop the query at some point. Or control over the query has to be delegated to the client.

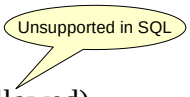
(However, we would like to emphasize once again that although such row-by-row processing may be really required from time to time, the same outcome can often be achieved using pure SQL; as a result, the code will become simpler and will work faster.)

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 (parameters are also allowed)
- the actual parameter values are passed when the cursor is being opened

Distinctive features

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

As we have seen, 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 at the cursor's opening time.

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

These methods are equivalent; which one to use is a matter of taste. In both cases, a query can have implicit parameters, which are derived from PL/pgSQL variables.

As noted above, a query opened with a cursor is prepared automatically.

https://postgrespro.com/docs/postgresql/12/plpgsql-cursors#PLPGSQL-CURSOR-DECLARATIONS

https://postgrespro.com/docs/postgresql/12/plpgsql-cursors#PLPGSQL-CURSOR-OPENING

Declaration and Opening

Let's 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
```

An unbound variable:

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

```
DO
```

A bound variable: the query is already specified at the declaration stage. The cur variable still has the same type: refcursor.

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

```
DO
```

If a variable is bound, the corresponding query can have parameters.

Note how naming ambiguities are resolved in the next two examples.

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

```
DO
```

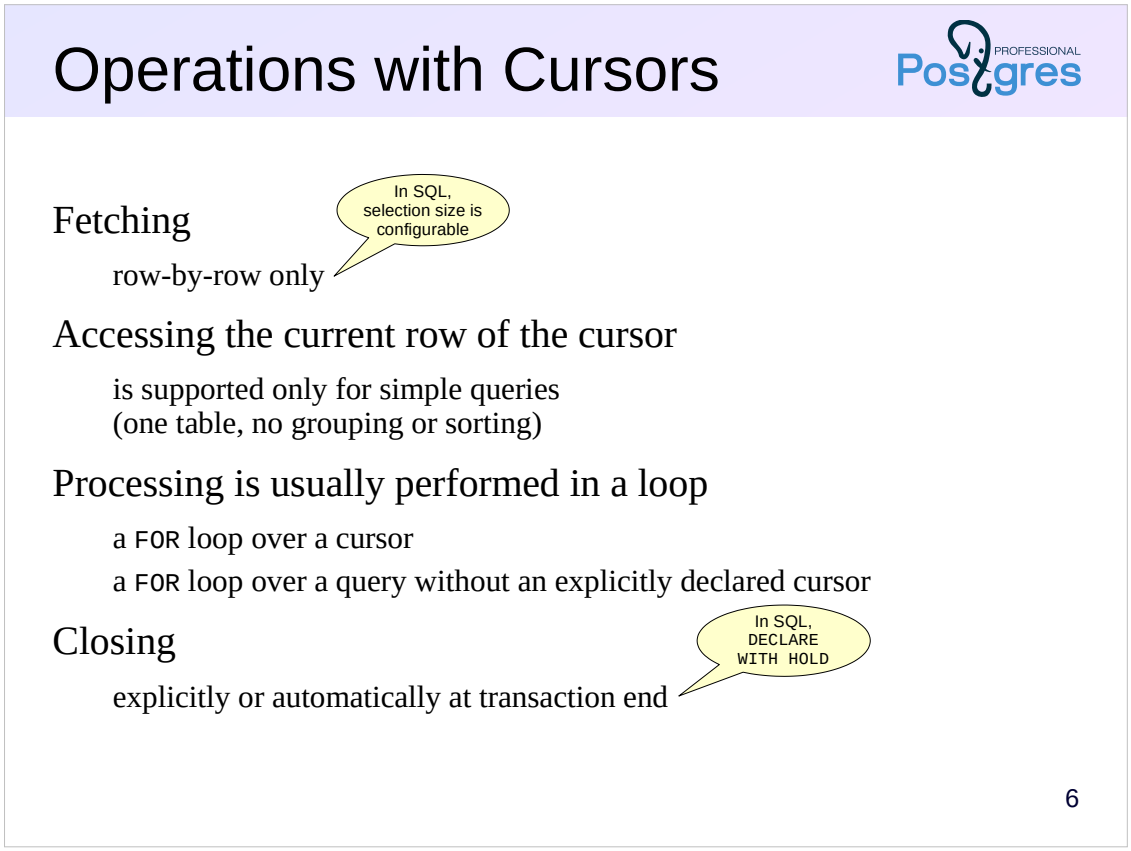
PL/pgSQL variables are also (implicit) parameters of the cursor.

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

```
DO
```

Apart from the SELECT command, a query can also be formed by any other command that returns a result (such as INSERT, UPDATE, DELETE with the RETURNING clause).

Operations with Cursors



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 in such commands 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 `FOR` command that implements them. We have already seen an integer flavor of `FOR` in the “PL/pgSQL. Overview and Programming Structures” lecture; 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 is specified in the command itself.

A cursor can be explicitly closed by a `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/12/plpgsql-cursors#PLPGSQL-CURSOR-USING

https://postgrespro.com/docs/postgresql/12/plpgsql-cursors#PLPGSQL-CURSOR-FOR-LOOP

Cursor Operations

The current row is read from the cursor by the FETCH command. To get to the next row, you can use MOVE.

What will be displayed on the screen?

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

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

Data selection is usually performed in a loop, which can be set up as follows:

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

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

But to avoid writing multiple commands, you can use a FOR loop over the cursor, which does exactly the same thing:

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

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

Moreover, you can do without an explicit cursor at all if iterating through a loop is all that you actually need.

It is convenient to enclose the query into parentheses, although it is not mandatory.

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

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

As with any loop, you can add a label, which can be useful for nested loops.

What will be displayed?

```
=> 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
```

Once the loop is complete, the FOUND variable shows whether any rows have been processed:

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

```
NOTICE:  Has at least one iteration completed? f
DO
```

The current row of the cursor bound to a simple query (over a single table, without grouping or sorting) can be referred to using the CURRENT OF clause. A typical use case is processing a batch of jobs with changing status.

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

```
DO
```

```
=> SELECT * FROM t;
```

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

Note that CURRENT OF does not support FOR loops over queries, as such loops do not explicitly use cursors. Clearly, you can get the same result by using UPDATE or DELETE with the unique table ID (WHERE id=rec.id). But CURRENT OF works faster and does not require an index.

Note that in many cases, you can replace a loop with a single SQL operator: it is easier and faster. Very often, loops are used simply because they have a more familiar “procedural” programming style. But this style is not a good choice for

databases.

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
```

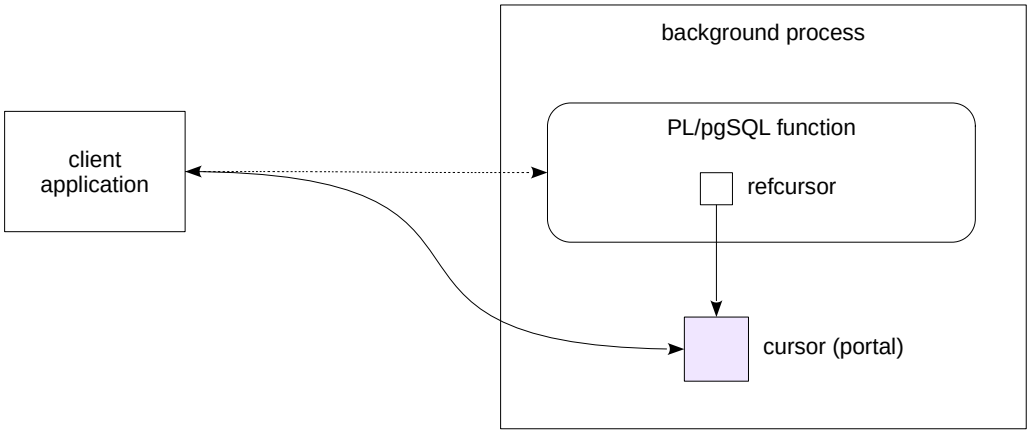
Such a loop can be replaced by one simple command:

```
=> 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 a Client



As we have already said, 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 *portal* term is used.

Thus, 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 a Client

Let's open a cursor and check the value of the cursor variable:

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

```
NOTICE:  <unnamed portal 10>
DO
```

It is the name of the cursor (portal) that has been opened on the server. Its name has been generated automatically.

You can set the name explicitly if you like (but it must be unique):

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

```
NOTICE:  cursor12345
DO
```

Taking advantage of this capability, we can create a function that opens the 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, calls the function, gets the cursor name, and can start reading data from this cursor. The exact way to do it depends on the programming language. In psql, it is done as follows:

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT t_cur() AS curname \gset
```

```
=> \echo :curname
```

```
<unnamed portal 11>
```

```
=> FETCH :"curname"; -- quotes are required because of special characters in the name
```

```
 id |        s
----+-----------------
  1 | One (processed)
(1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

You can also allow the client to set the name of the cursor if it is more convenient:

```
=> 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 becomes simpler:

```
=> BEGIN;
```

```
BEGIN
```

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

```
 t_cur
-------

(1 row)
```

```
=> FETCH cursor12345;
```

```
 id |        s
----+-----------------
  1 | One (processed)
(1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

A function can use OUT parameters to return several open cursors. This way, a single function call can provide the client with data from different tables, if required.

Alternatively, you can select all the required data at once at the server side and put it together as a JSON or XML document. We will cover these formats in DEV2.

Summary

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

Practice

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 writing the `book_name` function in SQL.
   Which implementation do you prefer: PL/pgSQL or SQL?

Task 1. For example:

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

Task 1. The book_name Function (Truncating the List of Authors)

Let's create a more generic function with an additional parameter that defines the maximum number of authors in a title.

Since we are going to change the function signature (the number and/or types of its input parameters), we have to delete this function and then create it anew. In this case, the function has a dependency: it is used in the catalog_v view. This view will also have to be recreated (in real life, all these actions must be performed in one and the same transaction, so that the changes come into effect atomically).

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

```
NOTICE:  drop cascades to view catalog_v
DROP FUNCTION
```

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

Task 2. An Alternative Implementation in Pure SQL

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

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)
```

Practice

1. It is required to distribute energy expenses 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 is required to return a single sorted sequence of integers from both sources.

Task 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 function 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
```

Task 2. A possible function 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
```

Task 1. Distributing Expenses

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

```
CREATE DATABASE
```

```
=> \c plpgsql_cursors
```

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

Create a 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 (10),(10),(10);
```

```
INSERT 0 3
```

Create a 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 the result:

```
=> SELECT distribute_expenses(100.00);
```

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

(1 row)
```

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

```
 id | employees | expenses
----+-----------+----------
  1 |        10 |    33.33
  2 |        10 |    33.34
  3 |        10 |    33.33
(3 rows)
```

Naturally, it is possible to use other algorithms, such as moving all rounding errors to a separate row.

In the DEV2 course, we provide another solution that employs user-defined aggregate functions.

Task 2. Merging Sorted Sequences

This implementation assumes that a number cannot be NULL.

```
=> 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
```

Check the implementation.

```
=> BEGIN;
```

```
BEGIN
```

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

```
DECLARE CURSOR
```

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

```
DECLARE CURSOR
```

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

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

```
=> COMMIT;
```

```
COMMIT
```