

# PL/pgSQL Курсоры



16

## **Авторские права**

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

## **Использование материалов курса**

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

## **Обратная связь**

Отзывы, замечания и предложения направляйте по адресу:

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

## **Отказ от ответственности**

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Причины использования

Объявление и открытие курсора

Операции с курсором

Циклы по курсору и по результатам запроса

Передача курсора клиенту

## Курсор подразумевает итеративную обработку

- полная выборка занимает слишком много памяти
- нужна не вся выборка, но размер заранее неизвестен
- способ отдать управление выборкой клиенту
- действительно требуется построчная обработка (обычно нет)

С концепцией курсоров мы уже знакомы по теме «Архитектура. Общее устройство PostgreSQL». Там речь шла о курсорах как о возможности, предоставляемой сервером, и мы смотрели, как к этой возможности обращаться средствами SQL. Теперь поговорим о том, как использовать те же самые курсоры в языке PL/pgSQL.

Зачем вообще может возникнуть необходимость в курсорах? Декларативный SQL в первую очередь предназначен для работы с множествами строк — в этом его сила и преимущество. PL/pgSQL, как процедурный язык, должен работать со строками по одной за раз, используя явные циклы. Такой подход как раз реализуется при помощи курсоров.

Например, полная выборка может занимать слишком много места, так что приходится обрабатывать результаты по частям. Или мы получаем набор данных неизвестного заранее размера — и в процессе выборки нужно вовремя остановиться. Или есть необходимость предоставить управление выборкой клиенту.

(Однако еще раз отметим, что хотя необходимость такой построчной обработки может возникать, в большом количестве случаев ее можно заменить чистым SQL; код в итоге окажется проще и будет быстрее работать.)

## Не связанные с запросом курсорные переменные

объявляется переменная типа `refcursor`  
конкретный запрос указывается при открытии

## Связанные с запросом курсорные переменные

при объявлении указывается запрос (возможно, с параметрами)  
при открытии указываются фактические значения параметров



## Особенности

значение курсорной переменной — имя курсора  
(можно задать явно или сгенерируется автоматически)  
переменные PL/pgSQL в запросе становятся неявными параметрами  
(значения подставляются при открытии курсора)  
запрос предварительно подготавливается

4

Как мы видели, в SQL курсор объявлялся и открывался одновременно командой `DECLARE`. В PL/pgSQL это два отдельных шага. Кроме того, для доступа к курсорам используются так называемые курсорные переменные, имеющие тип `refcursor` и, фактически, содержащие имя курсора (причем если не указывать это имя явно, PL/pgSQL сам позаботится о его уникальности).

Курсорную переменную можно объявить, не связывая ее с конкретным запросом. Тогда при открытии курсора нужно будет указать запрос.

Другой вариант — уже при объявлении переменной указать запрос, возможно, с параметрами. Тогда при открытии курсора указываются только фактические параметры.

Оба способа равноценны; какой использовать — дело вкуса.

И связанные, и несвязанные курсорные переменные инициализируются только при открытии курсора; в обоих случаях соответствующий запрос может иметь неявные параметры — переменные PL/pgSQL.

Напомним, что запрос, открытый с помощью курсора, автоматически подготавливается.

<https://postgrespro.ru/docs/postgresql/16/plpgsql-cursors#PLPGSQL-CURS-OR-DECLARATIONS>

<https://postgrespro.ru/docs/postgresql/16/plpgsql-cursors#PLPGSQL-CURS-OR-OPENING>

## Объявление и открытие

Создадим таблицу:

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

CREATE TABLE

```
=> INSERT INTO t VALUES (1, 'Раз'), (2, 'Два'), (3, 'Три');
```

INSERT 0 3

Несвязанная переменная:

```
=> DO $$
DECLARE
    -- объявление переменной
    cur refcursor;
BEGIN
    -- связывание с запросом и открытие курсора
    OPEN cur FOR SELECT * FROM t;
END
$$;

DO
```

Связанная переменная: запрос указывается уже при объявлении. При этом переменная cur имеет тот же тип refcursor.

```
=> DO $$
DECLARE
    -- объявление и связывание переменной
    cur CURSOR FOR SELECT * FROM t;
BEGIN
    -- открытие курсора
    OPEN cur;
END
$$;

DO
```

В случае связанной переменной запрос может иметь параметры.

Обратите внимание на устранение неоднозначности имен в этом и следующем примерах.

```
=> DO $$
DECLARE
    -- объявление и связывание переменной
    cur CURSOR(id integer) FOR SELECT * FROM t WHERE t.id = cur.id;
BEGIN
    -- открытие курсора с указанием фактических параметров
    OPEN cur(1);
END
$$;

DO
```

Переменные PL/pgSQL также являются (неявными) параметрами курсора.

```
=> DO $$
<<local>>
DECLARE
    id integer := 3;
    -- объявление и связывание переменной
    cur CURSOR FOR SELECT * FROM t WHERE t.id = local.id;
BEGIN
    id := 1;
    -- открытие курсора (значение id берется на этот момент)
    OPEN cur;
END
$$;

DO
```

В качестве запроса можно использовать не только команду SELECT, но и любую другую, возвращающую результат (например, INSERT, UPDATE, DELETE с фразой RETURNING).

# Операции с курсором

## Выборка

только по одной строке

в SQL  
размер выборки  
определяется

## Обращение к текущей строке курсора

только для простых запросов  
(одна таблица, без группировок и сортировок)

## Обычно обработка выполняется в цикле

цикл FOR по курсору  
цикл FOR по запросу без явного определения курсора

## Заккрытие

явно или автоматически при завершении транзакции

в SQL  
DECLARE  
WITH HOLD

Выборка строк из курсора возможна в PL/pgSQL только по одной. Для этого служит команда `FETCH INTO`.

Если запрос достаточно простой (одна таблица, без группировок и сортировок), то в процессе работы с курсором можно обращаться к текущей строке, например, в командах `UPDATE` или `DELETE`.

Процедурная обработка данных подразумевает работу с циклами. Можно организовать перебор и обработку всех строк, возвращаемых курсором, с помощью тех управляющих команд, что мы уже знаем. Но, поскольку часто нужен именно такой цикл, в PL/pgSQL есть специальная команда `FOR`. Целочисленный вариант `FOR` мы уже видели в теме «Обзор и конструкции языка», а этот вариант работает с курсором. Более того, есть еще один вариант цикла `FOR`, в котором не требуется даже объявления курсора — в команде указывается сам запрос.

Курсор можно закрыть явно командой `CLOSE`, но он в любом случае будет закрыт по окончании транзакции (в SQL курсор может оставаться открытым и после завершения транзакции, если указать фразу `WITH HOLD`).

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

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

## Операции с курсором

Чтение текущей строки из курсора выполняется командой FETCH. Если нужно только переместиться на следующую строку, можно воспользоваться другой командой — MOVE.

Что будет выведено на экран?

```
=> DO $$
DECLARE
    cur refcursor;
    rec record; -- можно использовать и несколько скалярных переменных
BEGIN
    OPEN cur FOR SELECT * FROM t ORDER BY id;
    MOVE cur;
    FETCH cur INTO rec;
    RAISE NOTICE '%', rec;
    CLOSE cur;
END
$$;

NOTICE:  (2,Два)
DO
```

Обычно выборка происходит в цикле, который можно организовать так:

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

NOTICE:  (1,Раз)
NOTICE:  (2,Два)
NOTICE:  (3,Три)
DO
```

Но чтобы не писать много команд, в PL/pgSQL имеется цикл FOR по курсору, который делает ровно то же самое:

```
=> DO $$
DECLARE
    cur CURSOR FOR SELECT * FROM t;
    -- переменная цикла не объявляется
BEGIN
    FOR rec IN cur LOOP -- суг должна быть связана с запросом
        RAISE NOTICE '%', rec;
    END LOOP;
END
$$;

NOTICE:  (1,Раз)
NOTICE:  (2,Два)
NOTICE:  (3,Три)
DO
```

Более того, можно вообще обойтись без явной работы с курсором, если цикл — это все, что требуется.

Скобки вокруг запроса не обязательны, но удобны.

```
=> DO $$
DECLARE
    rec record; -- надо объявить явно
BEGIN
    FOR rec IN (SELECT * FROM t) LOOP
        RAISE NOTICE '%', rec;
    END LOOP;
END
$$;
```

```
NOTICE: (1,Раз)
NOTICE: (2,Два)
NOTICE: (3,Три)
DO
```

Как и для любого цикла, здесь можно указать метку, что может оказаться полезным во вложенных циклах:

Что будет выведено?

```
=> 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,Раз), (1,Раз)
NOTICE: (1,Раз), (2,Два)
DO
```

После выполнения цикла переменная FOUND позволяет узнать, была ли обработана хотя бы одна строка:

```
=> DO $$
DECLARE
    rec record;
BEGIN
    FOR rec IN (SELECT * FROM t WHERE false) LOOP
        RAISE NOTICE '%', rec;
    END LOOP;
    RAISE NOTICE 'Была ли как минимум одна итерация? %', FOUND;
END
$$;
```

```
NOTICE: Была ли как минимум одна итерация? f
DO
```

На текущую строку курсора, связанного с простым запросом (по одной таблице, без группировок и сортировок) можно сослаться с помощью предложения CURRENT OF. Типичный случай применения — обработка пакета заданий с изменением статуса каждого из них.

```
=> DO $$
DECLARE
    cur refcursor;
    rec record;
BEGIN
    OPEN cur FOR SELECT * FROM t
        FOR UPDATE; -- строки блокируются по мере обработки
    LOOP
        FETCH cur INTO rec;
        EXIT WHEN NOT FOUND;
        UPDATE t SET s = s || ' (обработано)' WHERE CURRENT OF cur;
    END LOOP;
    CLOSE cur;
END
$$;
```

DO

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

```
id |          s
----+-----
 1 | Раз (обработано)
 2 | Два (обработано)
 3 | Три (обработано)
(3 rows)
```

Заметим, что CURRENT OF не работает с циклом FOR по запросу, поскольку этот цикл не использует курсор явным образом. Конечно, аналогичный результат можно получить, явно указав в команде UPDATE или DELETE уникальный ключ таблицы (WHERE id = rec.id). Но CURRENT OF работает быстрее и не требует наличия индекса.

---

Следует заметить, что в большом числе случаев вместо использования циклов можно выполнить задачу одним



оператором SQL — и это будет проще и еще быстрее. Часто циклы используют просто потому, что это более привычный, «процедурный» стиль программирования. Но для баз данных этот стиль не подходит.

Например:

```
=> 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;
```

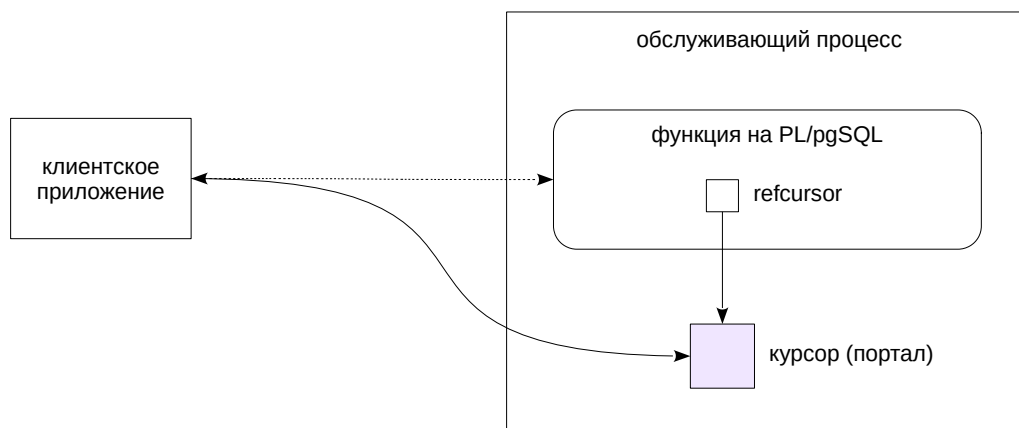
```
BEGIN
NOTICE:  (1,"Раз (обработано)")
NOTICE:  (2,"Два (обработано)")
NOTICE:  (3,"Три (обработано)")
DO
ROLLBACK
```

Такой цикл заменяется одной простой SQL-командой:

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

```
BEGIN
id |          s
----+-----
 1 | Раз (обработано)
 2 | Два (обработано)
 3 | Три (обработано)
(3 rows)
```

```
DELETE 3
ROLLBACK
```



Как уже говорилось, курсорная переменная PL/pgSQL (типа `refcursor`) содержит имя открытого SQL-курсора. Когда говорят о курсоре как о памяти, отведенной в обслуживающем процессе для хранения его состояния, в документации используется термин *портал*.

Таким образом, функция на PL/pgSQL может открыть курсор и вернуть его имя клиенту. Далее клиент сможет работать с курсором так, как будто он сам его и открыл, но доступ будет иметь только к тем данным, которые предоставлены при открытии. Это дает еще одну возможность для организации взаимодействия приложения и базы данных.

## Передача курсора клиенту

Откроем курсор и выведем значение курсорной переменной:

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

NOTICE: <unnamed portal 14>
DO
```

Это имя курсора (портала), который был открыт на сервере. Оно было сгенерировано автоматически.

При желании имя можно задать явно (но оно должно быть уникальным в сеансе):

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

NOTICE: cursor12345
DO
```

Пользуясь этим, можно написать функцию, которая откроет курсор и вернет его имя:

```
=> 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
```

Клиент начинает транзакцию...

```
=> BEGIN;
```

```
BEGIN
```

...вызывает функцию, узнает имя курсора...

```
=> SELECT t_cur();

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

...и получает возможность читать из него данные (кавычки нужны из-за спецсимволов в имени):

```
=> FETCH "<unnamed portal 15>";

 id |          s
-----+-----
  1 | Раз (обработано)
(1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

Для удобства можно позволить клиенту самому устанавливать имя курсора:

```
=> 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

```

Клиентский код упрощается:

```

=> BEGIN;

BEGIN

=> SELECT t_cur('cursor12345');

```

```

t_cur
-----

```

(1 row)

```

=> FETCH cursor12345;

```

```

id | s
---+-----
1  | Раз (обработано)
(1 row)

```

```

=> COMMIT;

```

```

COMMIT

```

Функция может вернуть и несколько открытых курсоров, используя OUT-параметры. Таким образом можно за один вызов функции обеспечить клиента информацией из разных таблиц, если это необходимо.

Альтернативный подход — сразу выбрать все необходимые данные на стороне сервера и сформировать из них документ JSON или XML. Работа с этими форматами рассматривается в курсе DEV2.

Курсор позволяет получать  
и обрабатывать данные построчно

Цикл FOR упрощает работу с курсорами

Обработка в цикле естественна для процедурных языков,  
но этим не стоит злоупотреблять



1. Измените функцию `book_name`: если у книги больше двух авторов, то в названии должны указываться первые два с добавлением «и др.» в конце.  
Проверьте работу функции в SQL и в приложении.
2. Попробуйте написать функцию `book_name` на SQL.  
Какой вариант нравится больше — PL/pgSQL или SQL?

## 1. Например:

Хрестоматия. Пушкин А. С., Толстой Л. Н., Тургенев И. С. →  
→ Хрестоматия. Пушкин А. С., Толстой Л. Н. и др.

## 1. Функция book\_name (сокращение авторов)

Напишем более универсальную функцию с дополнительным параметром — максимальное число авторов в названии.

Поскольку функция меняет сигнатуру (число и/или типы входных параметров), ее необходимо сначала удалить, а потом создать заново. В данном случае у функции есть зависимый объект — представление catalog\_v, в котором она используется. Представление тоже придется пересоздать (в реальной работе все эти действия надо выполнять в одной транзакции, чтобы изменения вступили в силу атомарно).

```
=> 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 OR REPLACE FUNCTION book_name(
    book_id integer,
    title text,
    maxauthors integer DEFAULT 2
)
RETURNS text
AS $$
DECLARE
    r record;
    res text = shorten(title);
BEGIN
    IF (right(res, 3) != '...') THEN res := res || '.'; END IF;
    res := res || ' ';
    FOR r IN (
        SELECT a.last_name, a.first_name, a.middle_name, ash.seq_num
        FROM authors a
            JOIN authorship ash ON a.author_id = ash.author_id
        WHERE ash.book_id = book_name.book_id
        ORDER BY ash.seq_num
    )
    LOOP
        EXIT WHEN r.seq_num > maxauthors;
        res := res || author_name(r.last_name, r.first_name, r.middle_name) || ', ';
    END LOOP;
    res := rtrim(res, ', ');
    IF r.seq_num > maxauthors THEN
        res := res || ' и др.';
    END IF;
    RETURN res;
END
$$ 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
4	Война и мир. Толстой Л. Н.
2	Муму. Тургенев И. С.
5	Путешествия в некоторые удаленные страны... Свифт Д.
1	Сказка о царе Салтане. Пушкин А. С.
3	Трудно быть богом. Стругацкий А. Н., Стругацкий Б. Н.
6	Хрестоматия. Пушкин А. С., Толстой Л. Н. и др.

(6 rows)

## 2. Вариант на чистом SQL

```

=> 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 ' и др.'
        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
4	Война и мир. Толстой Л. Н.
2	Муму. Тургенев И. С.
5	Путешествия в некоторые удаленные страны... Свифт Д.
1	Сказка о царе Салтане. Пушкин А. С.
3	Трудно быть богом. Стругацкий А. Н., Стругацкий Б. Н.
6	Хрестоматия. Пушкин А. С., Толстой Л. Н. и др.

(6 rows)



1. Требуется распределить расходы на электроэнергию по отделам компании пропорционально количеству сотрудников (перечень отделов находится в таблице). Напишите функцию, которая примет общую сумму расходов и запишет распределенные расходы в строки таблицы. Числа округляются до копеек; сумма расходов всех отделов должна в точности совпадать с общей суммой.
2. Напишите табличную функцию, имитирующую сортировку слиянием. Функция принимает две курсорные переменные; оба курсора уже открыты и возвращают упорядоченные по неубыванию целые числа. Требуется выдать общую упорядоченную последовательность чисел из обоих источников.

12

1. В качестве таблицы можно взять:

```
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);
```

Функция:

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

Ожидаемый результат после вызова функции с параметром 100.0:

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

2. Функция:

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

Например, если первый курсор возвращает последовательность 1, 3, 5, а второй — 2, 3, 4, то ожидается результат:

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

## 1. Распределение расходов

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

CREATE DATABASE

```
=> \c plpgsql_cursors
```

You are now connected to database "plpgsql\_cursors" as user "student".

Таблица:

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

CREATE TABLE

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

INSERT 0 3

Функция:

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

CREATE FUNCTION

Проверка:

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

distribute\_expenses  
-----

(1 row)

```
=> SELECT * FROM deps;
```

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

(3 rows)

Разумеется, возможны и другие алгоритмы, например, перенос всех ошибок округления на одну строку и т. п.

В курсе DEV2 рассматривается другое решение этой задачи с помощью пользовательских агрегатных функций.

## 2. Слияние отсортированных наборов

Эта реализация предполагает, что числа не могут иметь неопределенные значения 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

Проверяем.

```
=> 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