Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк
Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Процедуры и их отличие от функций

Входные и выходные параметры

Перегрузка и полиморфизм

Подпрограммы

Функции

- вызываются в контексте выражения
- не могут управлять транзакциями
- возвращают результат

Процедуры

- вызываются оператором CALL
- могут управлять транзакциями
- могут возвращать результат

Процедуры были введены в PostgreSQL 11. Основная причина их появления состоит в том, что функции не могут управлять транзакциями. Функции вызываются в контексте какого-либо выражения, которое вычисляется как часть уже начатого оператора (например, SELECT) в уже начатой транзакции. Нельзя завершить транзакцию и начать новую «посередине» выполнения оператора.

Процедуры всегда вызываются специальным оператором CALL. Если этот оператор сам начинает новую транзакцию (а не вызывается из уже начатой), то в процедуре можно использовать команды управления транзакциями.

К сожалению, процедуры, написанные на языке SQL, лишены сейчас возможности использовать команды COMMIT и ROLLBACK, хотя те из них, что оформлены в новом стиле стандарта SQL, вероятно смогут это делать в будущем. Поэтому пример процедуры, управляющей транзакциями, придется отложить до темы «PL/pgSQL. Выполнение запросов».

Иногда можно услышать, что процедура отличается от функции тем, что не возвращает результат. Но это не так — процедуры тоже могут возвращать результат, если необходимо.

Функции и процедуры имеют общее пространство имен и вместе называются *подпрограммами (routine)*.

https://postgrespro.ru/docs/postgresql/16/sql-createprocedure

https://postgrespro.ru/docs/postgresql/16/sql-call

Процедуры без параметров

Начнем с примера простой процедуры без параметров.

```
=> CREATE TABLE t(a float);
```

```
CREATE TABLE
```

```
=> CREATE PROCEDURE fill()
AS $$
    TRUNCATE t;
    INSERT INTO t SELECT random() FROM generate_series(1,3);
$$ LANGUAGE sql;
```

```
CREATE PROCEDURE
```

Чтобы вызвать процедуру, необходимо использовать специальный оператор:

```
=> CALL fill();
```

```
CALL
```

Результат работы виден в таблице:

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

```
           a
----------------------
   0.33549384886195055
   0.27391692859232375
 0.0007135169058296587
(3 rows)
```

А теперь переопределим нашу процедуру в стиле стандарта SQL:

```
=> CREATE OR REPLACE PROCEDURE fill()
LANGUAGE sql
BEGIN ATOMIC
    DELETE FROM t;  -- команда TRUNCATE пока что не поддерживается в таких подпрограммах
    INSERT INTO t SELECT random() FROM generate_series(1,3);
END;
```

```
CREATE PROCEDURE
```

Убедимся в ее работоспособности:

```
=> CALL fill();
```

```
CALL
```

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

```
          a
---------------------
  0.5138209108690146
  0.9624081296523503
 0.12926453407925398
(3 rows)
```

И попробуем в процедуре выполнить фиксацию транзакции:

```
=> CREATE OR REPLACE PROCEDURE fill()
LANGUAGE sql
BEGIN ATOMIC
    DELETE FROM t;
    INSERT INTO t SELECT random() FROM generate_series(1,3);
    COMMIT;
END;
```

```
ERROR:  COMMIT is not yet supported in unquoted SQL function body
```

Обратите внимание, что мы получили ошибку о недопустимой команде еще на этапе определения подпрограммы.

Переименуем таблицу, с которой работает наша процедура:

```
=> ALTER TABLE t RENAME TO ta;
```

```
ALTER TABLE
```

Вызов ниже не приведет к ошибке — таблица в определении процедуры в системном каталоге теперь представлена не по имени, а по идентификатору, который был получен еще на этапе создания подпрограммы.

```
=> CALL fill();
```

```
CALL
```

Ту же самую задачу, что выполняла процедура, можно решить и с помощью функции, возвращаемое значение которой определяется последним оператором. Можно объявить тип результата void, если фактически функция ничего не возвращает, или вернуть что-то осмысленное.

Возвратим таблице прежнее имя и определим функцию:

```
=> ALTER TABLE ta RENAME TO t;
```

```
ALTER TABLE
```

```
=> CREATE FUNCTION fill_avg() RETURNS float
LANGUAGE sql
BEGIN ATOMIC
    DELETE FROM t;
    INSERT INTO t SELECT random() FROM generate_series(1, 3);
    SELECT avg(a) FROM t;
END;
```

```
CREATE FUNCTION
```

В любом случае функция вызывается в контексте какого-либо выражения:

```
=> SELECT fill_avg();
```

```
     fill_avg
------------------
 0.610768252246219
(1 row)
```

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

```
         a
--------------------
 0.8349286559297802
 0.6017907659033424
 0.3955853349055345
(3 rows)
```

Чего нельзя достичь с помощью функции — это управления транзакциями. Но и в процедурах на языке SQL, как мы видели, это пока не поддерживается (зато поддерживается при использовании других языков).

Процедуры с параметрами

Добавим в процедуру входной параметр — число строк:

```
=> DROP PROCEDURE fill();
```

```
DROP PROCEDURE
```

```
=> CREATE PROCEDURE fill(nrows integer)
LANGUAGE sql
BEGIN ATOMIC
    DELETE FROM t;
    INSERT INTO t SELECT random() FROM generate_series(1, nrows);
END;
```

```
CREATE PROCEDURE
```

Точно так же, как и в случае функций, при вызове процедур фактические параметры можно передавать позиционным способом или по имени:

```
=> CALL fill(nrows => 5);
```

```
CALL
```

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

```
          a
--------------------
  0.8687305915657457
  0.5746613027422047
 0.23648642151784105
 0.20511705727523166
 0.12191686146548864
(5 rows)
```

Процедуры могут также иметь OUT- и INOUT-параметры, с помощью которых можно возвращать значения:

```
=> DROP PROCEDURE fill(integer);
```

```
DROP PROCEDURE
```

```
=> CREATE PROCEDURE fill(IN nrows integer, OUT average float)
LANGUAGE sql
BEGIN ATOMIC
    DELETE FROM t;
    INSERT INTO t SELECT random() FROM generate_series(1, nrows);
    SELECT avg(a) FROM t; -- как в функции
END;
```

```
CREATE PROCEDURE
```

Попробуем:

```
=> CALL fill(5, NULL /* значение не используется, но его необходимо указать*/);
```

```
      average
--------------------
 0.5339114517423708
(1 row)
```

Перегрузка

Несколько подпрограмм с одним и тем же именем

- подпрограмма однозначно определяется сигнатурой — именем и входными параметрами
- тип возвращаемого значения и выходные параметры игнорируются
- подходящая подпрограмма выбирается во время выполнения в зависимости от фактических параметров

Команда CREATE OR REPLACE

- при несовпадении типов входных параметров создаст новую перегруженную подпрограмму
- при совпадении — изменит существующую подпрограмму, но нельзя поменять тип подпрограммы, тип возвращаемого значения, типы OUT-параметров

Перегрузка — это возможность использования одного и того же имени для нескольких подпрограмм, отличающихся типами параметров IN и INOUT.

Сигнатура подпрограммы — ее имя и типы входных параметров. При вызове PostgreSQL находит подпрограмму, соответствующую сигнатуре. Возможны ситуации, когда подходящую подпрограмму невозможно определить однозначно; в таком случае во время выполнения возникнет ошибка.

Однако в сигнатуру, в частности, не входят:

- тип подпрограммы (процедура или функция);
- типы параметров OUT;
- тип возвращаемого значения.

Перегрузку надо учитывать, выполняя команду CREATE OR REPLACE (FUNCTION или PROCEDURE). Если сигнатура не совпадает ни с одной из существующих, будет создана новая перегруженная подпрограмма, а при совпадении — изменена существующая.
В последнем случае не разрешается изменять тип подпрограммы, тип OUT-параметров или тип возвращаемого значения, однако можно изменить язык и другие свойства. Поэтому иногда приходится удалять подпрограмму и создавать ее заново, в этом случае потребуется также удалить зависящие от нее объекты: другие подпрограммы, представления, триггеры и т. п. (DROP ROUTINE ... CASCADE).

https://postgrespro.ru/docs/postgresql/16/xfunc-overload

Полиморфизм

Подпрограмма, принимающая параметры разных типов

- формальные параметры используют полиморфные псевдотипы (например, `anyelement` или `anycompatible`)
- конкретный тип данных выбирается во время выполнения по типу фактических параметров

В некоторых случаях удобно не создавать несколько перегруженных подпрограмм для разных типов, а написать одну, принимающую параметры любого (или почти любого) типа.

Для этого в качестве типа формального параметра указывается специальный *полиморфный псевдотип*. Пока мы рассмотрим два из них — anyelement и anycompatible — но позже встретимся и с другими.

Полиморфные псевдотипы в определении позволяют использовать в качестве параметров любые типы, при этом конкретный тип, с которым будет работать подпрограмма, выбирается во время выполнения по типу фактического параметра.

Если в определении указано несколько полиморфных параметров типа anyelement, то все фактические параметры неявно приводятся к типу первого из них. Если же указано несколько параметров типа anycompatible, то типы фактических параметров приводятся к некоторому общему типу.

Если подпрограмма объявлена с возвращаемым значением полиморфного типа, то она должна иметь по крайней мере один входной полиморфный параметр. Конкретный тип возвращаемого значения также определяется исходя из типа фактического входного параметра. Для подпрограмм в стиле стандарта SQL возможности использовать полиморфные типы данных для аргументов нет.

https://postgrespro.ru/docs/postgresql/16/extend-type-system#EXTEND-TYPES-POLYMORPHIC

https://postgrespro.ru/docs/postgresql/16/xfunc-sql#XFUNC-SQL-POLYMORPHIC-FUNCTIONS

Перегруженные подпрограммы

Перегрузка работает одинаково и для функций, и для процедур. Они имеют общее пространство имен.

В качестве примера напишем функцию, возвращающую большее из двух целых чисел. (Похожее выражение есть в SQL и называется greatest, но мы напишем собственную функцию.)

```
=> CREATE FUNCTION maximum(a integer, b integer) RETURNS integer
LANGUAGE sql
RETURN CASE WHEN a > b THEN a ELSE b END;
```

```
CREATE FUNCTION
```

Проверим:

```
=> SELECT maximum(10, 20);
```

```
 maximum
---------
      20
(1 row)
```

Допустим, мы решили сделать аналогичную функцию для трех чисел. Благодаря перегрузке, не надо придумывать для нее какое-то новое название:

```
=> CREATE FUNCTION maximum(a integer, b integer, c integer)
RETURNS integer
LANGUAGE sql
RETURN CASE
         WHEN a > b THEN maximum(a, c)
         ELSE maximum(b, c)
       END;
```

```
CREATE FUNCTION
```

Теперь у нас две функции с одним именем, но разным числом параметров:

```
=> \df maximum
```

```
                              List of functions
 Schema |  Name   | Result data type |       Argument data types       | Type
--------+---------+------------------+---------------------------------+------
 public | maximum | integer          | a integer, b integer            | func
 public | maximum | integer          | a integer, b integer, c integer | func
(2 rows)
```

И обе работают:

```
=> SELECT maximum(10, 20), maximum(10, 20, 30);
```

```
 maximum | maximum
---------+---------
      20 |      30
(1 row)
```

Команда CREATE OR REPLACE позволяет создать подпрограмму или заменить существующую, не удаляя ее. Поскольку в данном случае функция с такой сигнатурой уже существует, она будет заменена:

```
=> CREATE OR REPLACE FUNCTION maximum(a integer, b integer, c integer)
RETURNS integer
LANGUAGE sql
RETURN CASE
         WHEN a > b THEN
           CASE WHEN a > c THEN a ELSE c END
         ELSE
           CASE WHEN b > c THEN b ELSE c END
         END;
```

```
CREATE FUNCTION
```

Пусть наша функция работает не только для целых чисел, но и для вещественных. Как этого добиться? Можно определить еще такую функцию:

```
=> CREATE FUNCTION maximum(a real, b real) RETURNS real
LANGUAGE sql
RETURN CASE WHEN a > b THEN a ELSE b END;
```

```
CREATE FUNCTION
```

Теперь у нас три функции с одинаковым именем:

```
=> \df maximum
```

```
                              List of functions
 Schema |  Name   | Result data type |       Argument data types       | Type
--------+---------+------------------+---------------------------------+------
 public | maximum | integer          | a integer, b integer            | func
 public | maximum | integer          | a integer, b integer, c integer | func
 public | maximum | real             | a real, b real                  | func
(3 rows)
```

Две из них имеют одинаковое количество параметров, но отличаются их типами:

```
=> SELECT maximum(10, 20), maximum(1.1, 2.2);
```

```
 maximum | maximum
---------+---------
      20 |     2.2
(1 row)
```

Если подпрограмма перегружена несколько раз, то для получения информации только о некоторых из них можно указать в команде \df типы интересующих параметров:

```
=> \df maximum real
```

```
                        List of functions
 Schema |  Name   | Result data type | Argument data types | Type
--------+---------+------------------+---------------------+------
 public | maximum | real             | a real, b real      | func
(1 row)
```

Дальше нам придется определить функции для всех остальных типов данных и повторить все то же самое для трех параметров. Притом что операторы в теле этих функций будут одни и те же.

Полиморфные функции

Здесь нам помогут полиморфные типы anyelement и anycompatible. Это псевдотипы, взамен которых при вызове и интерпретации функции будет подставлен тип фактического параметра. Разумеется, в случае определения подпрограммы в стиле стандарта SQL ее код будет разобран еще на этапе создания, и воспользоваться полиморфными псевдотипами не удастся.

Удалим все три наши функции...

```
=> DROP FUNCTION maximum(integer, integer);
```

```
DROP FUNCTION
```

```
=> DROP FUNCTION maximum(integer, integer, integer);
```

```
DROP FUNCTION
```

```
=> DROP FUNCTION maximum(real, real);
```

```
DROP FUNCTION
```

...и затем создадим новую:

```
=> CREATE FUNCTION maximum(a anyelement, b anyelement)
RETURNS anyelement
AS $$
    SELECT CASE WHEN a > b THEN a ELSE b END;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Такая функция должна принимать любой тип данных (а работать будет с любым типом, для которого определен оператор «больше»).

Получится?

```
=> SELECT maximum('A', 'B');
```

```
ERROR:  could not determine polymorphic type because input has type unknown
```

Увы, нет. В данном случае строковые литералы могут быть типа char, varchar, text — конкретный тип нам неизвестен. Но можно применить явное приведение типов:

```
=> SELECT maximum('A'::text, 'B'::text);
```

```
 maximum
---------
 B
(1 row)
```

Еще пример с другим типом:

```
=> SELECT maximum(now(), now() + interval '1 day');

            maximum
-------------------------------
 2024-01-19 12:42:31.951555+03
(1 row)
```

Тип результата функции всегда будет тот же, что и тип параметров.

Но можно продвинуться еще дальше — сделать так, чтобы можно было использовать в полиморфных подпрограммах не абсолютно одинаковые типы, а совместимые, то есть те, что могут быть приведены друг к другу неявно. Для этого нужно использовать полиморфный псевдотип anycompatible.

Удалим нашу функцию и взамен нее создадим другую:

```
=> DROP FUNCTION maximum;

DROP FUNCTION

=> CREATE FUNCTION maximum(a anycompatible, b anycompatible)
RETURNS anycompatible
AS $$
    SELECT CASE WHEN a > b THEN a ELSE b END;
$$ LANGUAGE sql;

CREATE FUNCTION
```

Повторим наш опыт с параметрами-литералами:

```
=> SELECT maximum('A', 'B');

 maximum
---------
 B
(1 row)
```

Получилось!

Но если типы параметров не совпадают и не могут быть неявно приведены к некоторому общему типу, то будет ошибка:

```
=> SELECT maximum(1, 'A');

ERROR:  invalid input syntax for type integer: "A"
LINE 1: SELECT maximum(1, 'A');
                          ^
```

В этом примере такое ограничение выглядит естественно, хотя в некоторых случаях оно может оказаться и неудобным.

Определим теперь функцию с тремя параметрами, но так, чтобы третий можно было не указывать:

```
=> CREATE FUNCTION maximum(
    a anycompatible,
    b anycompatible,
    c anycompatible DEFAULT NULL
) RETURNS anycompatible
AS $$
SELECT CASE
         WHEN c IS NULL THEN
             x
         ELSE
             CASE WHEN x > c THEN x ELSE c END
       END
FROM (
    SELECT CASE WHEN a > b THEN a ELSE b END
) max2(x);
$$ LANGUAGE sql;

CREATE FUNCTION

=> SELECT maximum(10, 11.21, 3e3);
```

```
 maximum
---------
    3000
(1 row)
```

Так работает. А так?

```
=> SELECT maximum(10, 11.21);
```

```
ERROR:  function maximum(integer, numeric) is not unique
LINE 1: SELECT maximum(10, 11.21);
               ^
HINT:  Could not choose a best candidate function. You might need to add explicit type
casts.
```

А так произошел конфликт перегруженных функций:

```
=> \df maximum
```

```
                                                 List of functions
 Schema |  Name   | Result data type |                            Argument data types
                     | Type
--------+---------+------------------+-----------------------------------------------------
---------------------+------
 public | maximum | anycompatible    | a anycompatible, b anycompatible
                     | func
 public | maximum | anycompatible    | a anycompatible, b anycompatible, c anycompatible
DEFAULT NULL::unknown | func
(2 rows)
```

Невозможно понять, имеем ли мы в виду функцию с двумя параметрами, или с тремя (но просто не указали последний).

Мы решим этот конфликт просто — удалим первую функцию за ненадобностью.

```
=> DROP FUNCTION maximum(anycompatible, anycompatible);
```

```
DROP FUNCTION
```

```
=> SELECT maximum(10, 11.21), maximum(10, 11.21, 3e3);
```

```
 maximum | maximum
---------+---------
   11.21 |    3000
(1 row)
```

Теперь все работает. А в теме «PL/pgSQL. Массивы» мы узнаем, как определять подпрограммы с произвольным числом параметров.

Итоги

Можно создавать и использовать собственные процедуры

В отличие от функций, процедуры вызываются оператором CALL и могут управлять транзакциями

Для процедур и функций поддерживаются перегрузка и полиморфизм

Практика

1. В таблице authors имена, фамилии и отчества авторов по смыслу должны быть уникальны, но это условие никак не проверяется. Напишите процедуру, удаляющую возможные дубликаты авторов.
2. Чтобы необходимость в подобной процедуре не возникала, создайте ограничение целостности, которое не позволит появляться дубликатам в будущем.

1. В приложении возможность добавлять авторов появится в теме «PL/pgSQL. Выполнение запросов». А пока для проверки можно добавить дубликаты в таблицу вручную.

1. Устранение дубликатов

В целях проверки добавим второго Пушкина:

```
=> INSERT INTO authors(last_name, first_name, middle_name)
    VALUES ('Пушкин', 'Александр', 'Сергеевич');
```

```
INSERT 0 1
```

```
=> SELECT last_name, first_name, middle_name, count(*)
FROM authors
GROUP BY last_name, first_name, middle_name;
```

```
 last_name  | first_name | middle_name | count
------------+------------+-------------+-------
 Свифт      | Джонатан   |             |     1
 Стругацкий | Борис      | Натанович   |     1
 Пушкин     | Александр  | Сергеевич   |     2
 Стругацкий | Аркадий    | Натанович   |     1
 Толстой    | Лев        | Николаевич  |     1
 Тургенев   | Иван       | Сергеевич   |     1
(6 rows)
```

Задачу устранения дубликатов можно решить разными способами. Например, так:

```
=> CREATE PROCEDURE authors_dedup()
LANGUAGE sql
BEGIN ATOMIC
DELETE FROM authors
WHERE author_id IN (
    SELECT author_id
    FROM (
        SELECT author_id,
               row_number() OVER (
                   PARTITION BY first_name, last_name, middle_name
                   ORDER BY author_id
               ) AS rn
        FROM authors
    ) t
    WHERE t.rn > 1
);
END;
```

```
CREATE PROCEDURE
```

```
=> CALL authors_dedup();
```

```
CALL
```

```
=> SELECT last_name, first_name, middle_name, count(*)
FROM authors
GROUP BY last_name, first_name, middle_name;
```

```
 last_name  | first_name | middle_name | count
------------+------------+-------------+-------
 Свифт      | Джонатан   |             |     1
 Стругацкий | Борис      | Натанович   |     1
 Пушкин     | Александр  | Сергеевич   |     1
 Стругацкий | Аркадий    | Натанович   |     1
 Толстой    | Лев        | Николаевич  |     1
 Тургенев   | Иван       | Сергеевич   |     1
(6 rows)
```

2. Ограничение целостности

Вспомним, что отчество может быть неопределенным (NULL), а по умолчанию для ограничений целостности UNIQUE действует правило, что неопределенные значения считаются различными. Таким образом, ограничение

```
UNIQUE(first_name, last_name, middle_name)
```

не помешает добавить второго Джонатана Свифта без отчества.

Однако мы можем такое поведение изменить:

```
=> ALTER TABLE authors ADD CONSTRAINT authors_nodup UNIQUE NULLS NOT DISTINCT (
    first_name, last_name, middle_name
);
```

```
ALTER TABLE
```

Проверим:

```
=> INSERT INTO authors(last_name, first_name)
    VALUES ('Свифт', 'Джонатан');
```

```
ERROR:  duplicate key value violates unique constraint "authors_nodup"
DETAIL:  Key (first_name, last_name, middle_name)=(Джонатан, Свифт, null) already exists.
```

```
=> INSERT INTO authors(last_name, first_name, middle_name)
    VALUES ('Пушкин', 'Александр', 'Сергеевич');
```

```
ERROR:  duplicate key value violates unique constraint "authors_nodup"
DETAIL:  Key (first_name, last_name, middle_name)=(Александр, Пушкин, Сергеевич) already
exists.
```

Практика

1. Получится ли создать в одной и той же схеме и имеющие одно и то же имя: а) процедуру с одним входным параметром, б) функцию с одним входным параметром того же типа, возвращающую некоторое значение? в) процедуру с одним входным и одним выходным параметром? Проверьте.
2. В таблице хранятся вещественные числа (например, результаты каких-либо измерений). Напишите процедуру нормализации данных, которая умножает все числа на определенный коэффициент так, чтобы все значения попали в интервал от −1 до 1.
   Процедура должна возвращать выбранный коэффициент.

2. В качестве коэффициента возьмите максимальное абсолютное значение из таблицы.

1. Перегрузка процедур и функций

```
=> CREATE DATABASE sql_proc;
```

```
CREATE DATABASE
```

```
=> \c sql_proc
```

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

Не получится, так как в сигнатуру подпрограммы входит только имя и тип входных параметров (возвращаемое значение игнорируется), и при этом процедуры и функции имеют общее пространство имен.

```
=> CREATE PROCEDURE test(IN x integer)
LANGUAGE sql
RETURN 1;
```

```
CREATE PROCEDURE
```

```
=> CREATE FUNCTION test(IN x integer) RETURNS integer
LANGUAGE sql
RETURN 1;
```

```
ERROR:  function "test" already exists with same argument types
```

В некоторых сообщениях, как и в этом, вместо слова «процедура» используется «функция», поскольку во многом они устроены одинаково.

```
=> CREATE OR REPLACE PROCEDURE test(IN x integer, OUT y integer)
LANGUAGE sql
RETURN x;
```

```
ERROR:  cannot change whether a procedure has output parameters
HINT:  Use DROP PROCEDURE test(integer) first.
```

Такую процедуру тоже создать нельзя, так как уже имеется процедура с такой же сигнатурой, а изменять выходные параметры (и факт их наличия) для имеющейся подпрограммы также запрещено. Нам предлагают удалить подпрограмму, чтобы затем создать ее заново.

2. Нормализация данных

Таблица с тестовыми данными:

```
=> CREATE TABLE samples(a float);
```

```
CREATE TABLE
```

```
=> INSERT INTO samples(a)
    SELECT (0.5 - random())*100 FROM generate_series(1,10);
```

```
INSERT 0 10
```

Процедуру можно написать, используя один SQL-оператор:

```
=> CREATE PROCEDURE normalize_samples(INOUT coeff float)
LANGUAGE sql
BEGIN ATOMIC
   WITH c(coeff) AS (
       SELECT 1/max(abs(a))
       FROM samples
   ),
   upd AS (
       UPDATE samples
       SET a = a * c.coeff
       FROM c
   )
   SELECT coeff FROM c;
END;
```

```
CREATE PROCEDURE
```

```
=> CALL normalize_samples(NULL);
```

```
        coeff
---------------------
 0.02161050802622934
(1 row)
```

```
=> SELECT * FROM samples;
```

```
          a
----------------------
  0.31602898268788376
 -0.07324710356368555
   0.4258755136278238
  0.15854584091830776
  0.24864054058642118
   0.1536963640711513
  -0.3057260266767897
  -0.9790164692458508
 -0.48071022388392104
                    1
(10 rows)
```