Авторы: Егор Рогов, Павел Лузанов

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

Темы

Функции и их особенности в базах данных

Параметры и возвращаемое значение

Способы передачи параметров при вызове

Категории изменчивости и оптимизация

Функции в базе данных

Основной мотив: упрощение задачи

- интерфейс (параметры) и реализация (тело функции)
- о функции можно думать вне контекста всей задачи

	Традиционные языки	*PostgreSQL*
побочные эффекты	глобальные переменные	вся база данных (категории изменчивости)
модули	со своим интерфейсом и реализацией	пространства имен, клиент и сервер
сложности	накладные расходы на вызов (подстановка)	скрытие запроса от планировщика (подстановка, подзапросы, представления)

Основная цель появления функций в программировании вообще — упростить решаемую задачу за счет ее декомпозиции на более мелкие подзадачи. Упрощение достигается за счет того, что о функции можно думать, абстрагировавшись от «большой» задачи. Для этого функция определяет четкий интерфейс с внешним миром (параметры и возвращаемое значение). Ее реализация (тело функции) может меняться; вызывающая сторона «не видит» этих изменений и не зависит от них. Этой идеальной ситуации может мешать глобальное состояние (глобальные переменные), и надо учитывать, что в случае БД таким состоянием является вся база данных.

В традиционных языках функции часто объединяются в модули (пакеты, классы для ООП и т. п.), имеющие собственный интерфейс и реализацию. Границы модулей могут проводиться более или менее произвольно. Для PostgreSQL есть жесткая граница между клиентской частью и серверной: серверный код работает с базой, клиентский — управляет транзакциями. Модули (пакеты) отсутствуют, есть только пространства имен.

Для традиционных языков единственный минус широкого использования функций состоит в накладных расходах на ее вызов. Иногда его преодолевают с помощью подстановки (inlining) кода функции в вызывающую программу. Для БД последствия могут быть более серьезные: если в функцию выносится часть запроса, планировщик перестает видеть «общую картину» и не может построить хороший план. В некоторых случаях PostgreSQL умеет выполнять подстановку; альтернативные варианты — использование подзапросов или представлений.

Общие сведения

Объект базы данных

- определение хранится в системном каталоге

Основные составные части определения

- имя
- параметры
- тип возвращаемого значения
- тело

Доступны несколько языков, в том числе SQL

- код в виде строковой константы
- интерпретируется при вызове

Вызывается в контексте выражения

Функции являются такими же объектами базы данных, как, например, таблицы и индексы. Определение функции сохраняется в системном каталоге; поэтому функции в базе данных называют *хранимыми*.

В PostgreSQL доступно большое количество стандартных функций. С некоторыми их которых можно познакомиться в справочном материале «Основные типы данных и функции».

И, конечно, можно писать собственные функции на разных языках программирования. Материал этой темы относится к функциям на любом языке, но примеры будут использовать язык SQL.

Определение функции — довольно ожидаемо — состоит из имени, необязательных параметров, типа возвращаемого значения, и тела. Что может показаться неожиданным — тело записывается в виде строковой константы, которая содержит код на выбранном языке программирования. За счет этого определение функции выглядит одинаково независимо от выбранного языка. Тело-строка сохраняется в системном каталоге и интерпретируется каждый раз, когда функция вызывается. В настоящий момент единственный способ избежать интерпретации — написать функцию на языке Си, но это требуется нечасто, и в данном курсе эта тема не рассматривается.

Функция всегда вызывается в контексте какого-либо выражения. Например, в списке выражений команды SELECT, в условии WHERE, в ограничении целостности CHECK и т. п.

https://postgrespro.ru/docs/postgresql/12/sql-createfunction

https://postgrespro.ru/docs/postgresql/12/sql-syntax-calling-funcs

Функции без параметров

Вот простой пример функции без параметров:

```
=> CREATE FUNCTION hello_world() -- имя и пустой список параметров
RETURNS text                     -- тип возвращаемого значения
AS $$ SELECT 'Hello, world!'; $$ -- тело
LANGUAGE sql;                    -- указание языка
```

```
CREATE FUNCTION
```

Тело удобно записывать в строке, заключенной в кавычки-доллары, как в приведенном примере. Иначе придется заботиться об экранировании кавычек, которые наверняка встретятся в теле функции. Сравните:

```
=> SELECT ' SELECT ''Hello, world!''; ';
```

```
         ?column?
--------------------------
  SELECT 'Hello, world!';
(1 row)
```

```
=> SELECT $$ SELECT 'Hello, world!'; $$;
```

```
         ?column?
--------------------------
  SELECT 'Hello, world!';
(1 row)
```

При необходимости кавычки-доллары могут быть вложенными. Для этого в каждой паре кавычек надо использовать разный текст между долларами:

```
=> SELECT $func$ SELECT $$Hello, world!$$; $func$;
```

```
          ?column?
----------------------------
  SELECT $$Hello, world!$$;
(1 row)
```

Функция вызывается в контексте выражения, например:

```
=> SELECT hello_world(); -- пустые скобки обязательны
```

```
  hello_world
---------------
 Hello, world!
(1 row)
```

В общем случае тело функции может состоять из нескольких операторов SQL. В качестве значения функции возвращается значение из первой строки, которую вернул последний оператор.

Не все операторы SQL можно использовать в функции. Запрещены:

- команды управления транзакциями (BEGIN, COMMIT, ROLLBACK и т. п.);
- служебные команды (такие как VACUUM или CREATE INDEX).

Вот пример неправильной функции. Здесь мы использовали псевдотип void, который говорит о том, что функция не возвращает ничего.

```
=> CREATE FUNCTION do_commit() RETURNS void AS $$
COMMIT;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
=> SELECT do_commit();
```

```
ERROR:  COMMIT is not allowed in a SQL function
CONTEXT:  SQL function "do_commit" during startup
```

Управлять транзакциями можно в процедурах, о чем мы будем говорить в следующей теме.

Функции с входными параметрами

Пример функции с одним параметром:

```
=> CREATE FUNCTION hello(name text) -- формальный параметр
RETURNS text AS $$
SELECT 'Hello, ' || name || '!';
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

При вызове функции мы указываем фактический параметр, соответствующий формальному:

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

При указании типа параметра можно указать и модификатор (например, varchar(10)), но он игнорируется.

Можно определить параметр функции без имени; тогда внутри тела функции на параметры придется ссылаться по номеру. Удалим функцию и создадим новую:

```
=> DROP FUNCTION hello(text); -- достаточно указать тип параметра
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(text)
RETURNS text AS $$
SELECT 'Hello, ' || $1 || '!'; -- номер вместо имени
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

Но так лучше не делать, это неудобно.

Удалим функцию и создадим заново, добавив еще один параметр — обращение.

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

```
DROP FUNCTION
```

Здесь мы использовали необязательное ключевое слово IN, обозначающее входной параметр. Предложение DEFAULT позволяет определить значение по умолчанию для параметра:

```
=> CREATE FUNCTION hello(IN name text, IN title text DEFAULT 'Mr')
RETURNS text AS $$
SELECT 'Hello, ' || title || ' ' || name || '!';
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice', 'Mrs'); -- указаны оба параметра
```

```
       hello
--------------------
 Hello, Mrs Alice!
(1 row)
```

```
=> SELECT hello('Bob'); -- опущен параметр, имеющий значение по умолчанию
```

```
     hello
----------------
 Hello, Mr Bob!
(1 row)
```

До сих пор мы вызывали функцию, указывая фактические параметры позиционным способом — в том порядке, в котором они определены при создании функции. Во многих стандартных функциях имена параметров не заданы, так что этот способ оказывается единственным.

Но если формальным параметрам даны имена, можно использовать их при указании фактических параметров. В этом случае параметры могут указываться в произвольном порядке:

```
=> SELECT hello(title => 'Mrs', name => 'Alice');
```

```
       hello
-------------------
 Hello, Mrs Alice!
(1 row)
```

```
=> SELECT hello(name => 'Bob');
```

```
     hello
----------------
 Hello, Mr Bob!
(1 row)
```

Такой способ удобен, когда порядок параметров неочевиден, особенно если их много.

Можно совмещать оба способа: часть параметров (начиная с первого) указать позиционно, а оставшиеся — по имени:

```
=> SELECT hello('Alice', title => 'Mrs');
```

```
       hello
-------------------
 Hello, Mrs Alice!
(1 row)
```

В случае, когда функция должна возвращать неопределенное значение, если хотя бы один из входных параметров не определен, ее можно объявить как строгую (STRICT). Тело функции при этом вообще не будет выполняться.

```
=> DROP FUNCTION hello(text, text);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(IN name text, IN title text DEFAULT 'Mr')
RETURNS text AS $$
SELECT 'Hello, ' || title || ' ' || name || '!';
$$ LANGUAGE sql STRICT;
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice', NULL);
```

```
 hello
-------

(1 row)
```

Вход и выход

Входные значения

- определяются параметрами с режимом IN и INOUT

Выходное значение

- определяется либо предложением RETURNS,
либо параметрами с режимом INOUT и OUT
- если одновременно указаны обе формы, они должны быть согласованы

Формальные параметры с режимом IN и INOUT считаются *входными*. Значения соответствующих фактических параметров должны быть указаны при вызове функции (либо должны быть определены значения по умолчанию).

Возвращаемое значение можно определить двумя способами:

- использовать предложение RETURNS для указания типа;

- определить *выходные* параметры с режимом INOUT или OUT.

Две эти формы записи эквивалентны. Например, функция с указанием RETURNS integer и функция с параметром OUT integer возвращают целое число.

Можно использовать и оба способа одновременно. В этом случае функция также будет возвращать *одно* целое число. Но при этом типы RETURNS и выходных параметров должны быть согласованы друг с другом.

Таким образом, нельзя написать функцию, которая будет возвращать одно значение, и при этом передавать другое значение в OUT-параметр — что позволяет большинство традиционных языков программирования. В PostgreSQL такая функция будет *возвращать оба значения*.

Функции с выходными параметрами

Альтернативный способ вернуть значение — использовать выходной параметр.

```
=> DROP FUNCTION hello(text, text);
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(
    IN name text,
    OUT text -- имя можно не указывать, если оно не нужно
)
AS $$
SELECT 'Hello, ' || name || '!';
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

Результат тот же самый.

Можно использовать и RETURNS, и OUT-параметр вместе — результат снова будет тем же:

```
=> DROP FUNCTION hello(text); -- OUT-параметры не указываем
```

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(IN name text, OUT text)
RETURNS text AS $$
SELECT 'Hello, ' || name || '!';
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

Или даже так, использовав INOUT-параметр:

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

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(INOUT name text)
AS $$
SELECT 'Hello, ' || name || '!';
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice');
```

```
     hello
---------------
 Hello, Alice!
(1 row)
```

Обратите внимание, что, в отличие от многих языков программирования, в PL/pgSQL фактическое значение, переданное SQL-функции в INOUT-параметре, никак не изменяется: мы передаем входное значение, а выходное возвращается функцией в качестве результата. Поэтому мы можем указать константу, хотя другие языки требовали бы переменную.

В то время как в RETURNS можно указать только одно значение, выходных параметров может быть несколько. Например:

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

```
DROP FUNCTION
```

```
=> CREATE FUNCTION hello(
    IN name text,
    OUT greeting text,
    OUT clock timetz)
AS $$
SELECT 'Hello, ' || name || '!', current_time;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice');
```

```
                hello
-------------------------------------
 ("Hello, Alice!",11:05:20.617553+03)
(1 row)
```

Действительно, наша функция вернула не одно значение, а сразу несколько.

Подробнее о такой возможности и составных типах мы будем говорить в теме «SQL. Составные типы».

Категории изменчивости

Volatile

- возвращаемое значение может произвольно меняться при одинаковых значениях входных параметров
- используется по умолчанию

Stable

- значение не меняется в пределах одного оператора SQL
- функция не может менять состояние базы данных

Immutable

- значение не меняется, функция детерминирована
- функция не может менять состояние базы данных

Каждой функции сопоставлена категория изменчивости, которая определяет свойства возвращаемого значения при одинаковых значениях входных параметров.

Категория volatile говорит о том, что возвращаемое значение может произвольно меняться. Такие функции будут вычисляться при каждом вызове. Если при создании функции категория не указана, назначается именно эта категория.

Категория stable используется для функций, возвращаемое значение которых не меняется в пределах одного SQL-оператора. В частности, такие функции не могут менять состояние БД. Такая функция *может* быть выполнена один раз во время выполнения запроса, а затем будет использоваться вычисленное значение.

Категория immutable еще более строгая: возвращаемое значение не меняется никогда. Такую функцию *можно* вычислить на этапе планирования запроса, а не во время выполнения.

Можно — не означает, что всегда происходит именно так, но планировщик вправе выполнить такие оптимизации. В некоторых (простых) случаях планировщик делает собственные выводы об изменчивости функции, невзирая на указанную явно категорию.

https://postgrespro.ru/docs/postgresql/12/xfunc-volatility

Категории изменчивости и изоляция

В целом использование функций внутри запросов не нарушает установленный уровень изоляции транзакции, но есть два момента, о которых полезно знать.

Во-первых, функции с изменчивостью volatile на уровне изоляции Read Committed приводят к рассогласованию данных внутри одного запроса.

Сделаем функцию, возвращающую число строк в таблице:

```
=> CREATE TABLE t(n integer);
```

```
CREATE TABLE
```

```
=> CREATE FUNCTION cnt() RETURNS bigint
AS $$
    SELECT count(*) FROM t;
$$ VOLATILE LANGUAGE sql;
```

```
CREATE FUNCTION
```

Теперь вызовем ее несколько раз с задержкой, а в параллельном сеансе вставим в таблицу строку.

```
=> BEGIN ISOLATION LEVEL READ COMMITTED;
```

```
BEGIN
```

```
=> SELECT (SELECT count(*) FROM t), cnt(), pg_sleep(1)
FROM generate_series(1,4);
```

```
  => INSERT INTO t VALUES (1);
```

```
  INSERT 0 1
```

```
 count | cnt | pg_sleep
-------+-----+----------
     0 |   0 |
     0 |   0 |
     0 |   1 |
     0 |   1 |
(4 rows)
```

```
=> END;
```

```
COMMIT
```

При изменчивости stable или immutable, либо использовании более строгих уровней изоляции, такого не происходит.

```
=> ALTER FUNCTION cnt() STABLE;
```

```
ALTER FUNCTION
```

```
=> TRUNCATE t;
```

```
TRUNCATE TABLE
```

```
=> BEGIN ISOLATION LEVEL READ COMMITTED;
```

```
BEGIN
```

```
=> SELECT (SELECT count(*) FROM t), cnt(), pg_sleep(1)
FROM generate_series(1,4);
```

```
  => INSERT INTO t VALUES (1);
```

```
  INSERT 0 1
```

```
 count | cnt | pg_sleep
-------+-----+----------
     0 |   0 |
     0 |   0 |
     0 |   0 |
     0 |   0 |
(4 rows)
```

```
=> END;
```

```
COMMIT
```

Второй момент связан с видимостью изменений, сделанных собственной транзакцией.

Функции с изменчивостью volatile видят все изменения, в том числе сделанные текущим, еще не завершенным оператором SQL.

```
=> ALTER FUNCTION cnt() VOLATILE;

ALTER FUNCTION

=> TRUNCATE t;

TRUNCATE TABLE

=> INSERT INTO t SELECT cnt() FROM generate_series(1,5);

INSERT 0 5

=> SELECT * FROM t;

 n
---
 0
 1
 2
 3
 4
(5 rows)
```

Это верно для любых уровней изоляции.

Функции с изменчивостью stable или immutable видят изменения только уже завершенных операторов.

```
=> ALTER FUNCTION cnt() STABLE;

ALTER FUNCTION

=> TRUNCATE t;

TRUNCATE TABLE

=> INSERT INTO t SELECT cnt() FROM generate_series(1,5);

INSERT 0 5

=> SELECT * FROM t;

 n
---
 0
 0
 0
 0
 0
(5 rows)
```

Категории изменчивости и оптимизация

Благодаря дополнительной информации о поведении функции, которую дает указание категории изменчивости, оптимизатор может сэкономить на вызовах функции.

Для экспериментов создадим функцию, возвращающую случайное число:

```
=> CREATE FUNCTION rnd() RETURNS float
AS $$
    SELECT random();
$$ VOLATILE LANGUAGE sql;

CREATE FUNCTION
```

Проверим план выполнения следующего запроса:

```
=> EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;

                   QUERY PLAN
-------------------------------------------------
 Function Scan on generate_series
   Filter: (random() > '0.5'::double precision)
(2 rows)
```

В плане мы видим «честное» обращение к функции generate_series; каждая строка результата сравнивается со случайным числом и при необходимости отбрасывается фильтром.

В этом можно убедиться и воочию (ожидаем в среднем получить 5 строк):

```
=> SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;

 generate_series
-----------------
               1
               2
               4
               6
               9
(5 rows)
```

Функция с изменчивостью stable будет вызвана всего один раз — поскольку мы фактически указали, что ее значение не может измениться в пределах оператора:

```
=> ALTER FUNCTION rnd() STABLE;

ALTER FUNCTION

=> EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;

                      QUERY PLAN
------------------------------------------------------
 Result
   One-Time Filter: (rnd() > '0.5'::double precision)
   ->  Function Scan on generate_series
(3 rows)
```

Наконец, изменчивость immutable позволяет вычислить функции еще на этапе планирования, поэтому во время выполнения никакие фильтры не нужны:

```
=> ALTER FUNCTION rnd() IMMUTABLE;

ALTER FUNCTION

=> EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;

        QUERY PLAN
--------------------------
 Result
   One-Time Filter: false
(2 rows)
```

Ответственность «за дачу заведомо ложных показаний» лежит на разработчике.

Подстановка тела функции в SQL-запрос

В некоторых (очень простых) случаях тело функции на языке SQL может быть подставлено прямо в основной SQL-оператор на этапе разбора запроса. В этом случае время на вызов функции не тратится.

Упрощенно требуется выполнение следующих условий:

- тело функции состоит из одного оператора SELECT;
- нет обращений к таблицам, отсутствуют подзапросы, группировки и т. п.;
- возвращаемое значение должно быть одно;
- вызываемые функции не должны противоречить указанной категории изменчивости.

Пример мы уже видели: наша функция rnd(), объявленная volatile.

Посмотрим еще раз.

```
=> ALTER FUNCTION rnd() VOLATILE;

ALTER FUNCTION

=> EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;

                   QUERY PLAN
-------------------------------------------------
 Function Scan on generate_series
   Filter: (random() > '0.5'::double precision)
(2 rows)
```

В фильтре упоминается только функция random(), которая вызывается напрямую, минуя «обертку» в виде функции rnd().

Итоги

Можно создавать собственные функции
и использовать их так же, как и встроенные

Функции можно писать на разных языках, в том числе SQL

Изменчивость влияет на возможности оптимизации

Иногда функция на SQL может быть подставлена в запрос

Практика

1. Создайте функцию author_name для формирования имени автора. Функция принимает три параметра (фамилия, имя, отчество) и возвращает строку с фамилией и инициалами.

   Используйте эту функцию в представлении authors_v.

2. Создайте функцию book_name для формирования названия книги.
   Функция принимает два параметра (идентификатор книги и заголовок) и возвращает строку, составленную из заголовка и списка авторов в порядке seq_num. Имя каждого автора формируется функций author_name.

   Используйте эту функцию в представлении catalog_v.

Проверьте изменения в приложении.

Напомним, что необходимые функции можно посмотреть в раздаточном материале «Основные типы данных и функции».

1. `FUNCTION author_name(`
```
    last_name text, first_name text, surname text
)
RETURNS text
```

Например: `author_name('Толстой','Лев','Николаевич') →`
```
→ 'Толстой Л. Н.'
```

2. `FUNCTION book_name(book_id integer, title text)`
```
RETURNS text
```

Например: `book_name(3,'Трудно быть богом') →`
```
→ 'Трудно быть богом. Стругацкий А. Н., Стругацкий Б. Н.'
```

Все инструменты позволяют «непосредственно» редактировать хранимые функции. Например, в psql есть команда \ef, открывающая текст функции в редакторе и сохраняющая изменения в базу.

Такой возможностью лучше не пользоваться (или как минимум не злоупотреблять). Нормально построенный процесс разработки предполагает хранение всего кода в файлах под версионным контролем. При необходимости изменить функцию файл редактируется и выполняется (с помощью psql или средствами IDE). Если же менять определение функций сразу в БД, изменения легко потерять. (Вообще же вопрос организации процесса разработки намного сложнее и в курсе мы его не затрагиваем.)

1. Функция author_name

```
=> CREATE OR REPLACE FUNCTION author_name(
    last_name text,
    first_name text,
    middle_name text
) RETURNS text
AS $$
SELECT last_name || ' ' ||
       left(first_name, 1) || '.' ||
       CASE WHEN middle_name != '' -- подразумевает NOT NULL
           THEN ' ' || left(middle_name, 1) || '.'
           ELSE ''
       END;
$$ IMMUTABLE LANGUAGE sql;
```

```
CREATE FUNCTION
```

Категория изменчивости — immutable. Функция всегда возвращает одинаковое значение при одних и тех же входных параметрах.

```
=> CREATE OR REPLACE VIEW authors_v AS
SELECT a.author_id,
       author_name(a.last_name, a.first_name, a.middle_name) AS display_name
FROM   authors a
ORDER BY display_name;
```

```
CREATE VIEW
```

2. Функция book_name

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

```
CREATE FUNCTION
```

Категория изменчивости — stable. Функция возвращает одинаковое значение при одних и тех же входных параметрах, но только в рамках одного SQL-запроса.

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

```
DROP VIEW
```

```
=> CREATE VIEW catalog_v AS
SELECT b.book_id,
       book_name(b.book_id, b.title) AS display_name
FROM   books b
ORDER BY display_name;
```

```
CREATE VIEW
```

Практика

1. Напишите функцию, выдающую случайное время, равномерно распределенное в указанном отрезке.
   Начало отрезка задается временной отметкой (timestamptz), конец — либо временной отметкой, либо интервалом (interval).
2. В таблице хранятся номера автомобилей, введенные кое-как: встречаются как латинские, так и русские буквы в любом регистре; между буквами и цифрами могут быть пробелы.
   Считая, что формат номера «*буква три-цифры две-буквы*», напишите функцию, выдающую число уникальных номеров.
   Например, «K 123 XM» и «k123xm» считаются равными.
3. Напишите функцию, находящую корни квадратного уравнения.

Во всех заданиях обратите особое внимание на категорию изменчивости функций.

2. Сначала напишите функцию «нормализации» номера, то есть приводящую номер к какому-нибудь стандартному виду. Например, без пробелов и только заглавными латинскими буквами.

В номерах используются только 12 русских букв, имеющих латинские аналоги схожего начертания, а именно: АВЕКМНОРСТУХ.

3. Для уравнения $y = ax^2 + bx + c$:

дискриминант $D = b^2 – 4ac$.

- при $D > 0$ два корня $x_{1,2} = (–b \pm \sqrt{D}) / 2a$
- при $D = 0$ один корень $x = –b / 2a$ (в качестве x_2 можно вернуть null)
- при $D < 0$ корней нет (оба корня null).

1. Случайная временная отметка

Функция с двумя временными отметками:

```
=> CREATE FUNCTION rnd_timestamp(t_start timestamptz, t_end timestamptz)
RETURNS timestamptz
AS $$
    SELECT t_start + (t_end - t_start) * random();
$$ VOLATILE LANGUAGE sql;
```

```
CREATE FUNCTION
```

Категория изменчивости — volatile. Используется функция random, поэтому функция будет возвращать разные значения при одних и тех же входных параметрах.

```
=> SELECT current_timestamp,
    rnd_timestamp(
        current_timestamp,
        current_timestamp + interval '1 hour'
    )
FROM generate_series(1,10);
```

```
       current_timestamp       |         rnd_timestamp
-------------------------------+-------------------------------
 2023-07-26 11:11:13.193191+03 | 2023-07-26 11:41:41.90775+03
 2023-07-26 11:11:13.193191+03 | 2023-07-26 12:07:48.158397+03
 2023-07-26 11:11:13.193191+03 | 2023-07-26 12:01:19.254117+03
 2023-07-26 11:11:13.193191+03 | 2023-07-26 11:55:02.500033+03
 2023-07-26 11:11:13.193191+03 | 2023-07-26 11:26:03.176406+03
 2023-07-26 11:11:13.193191+03 | 2023-07-26 11:22:37.967449+03
 2023-07-26 11:11:13.193191+03 | 2023-07-26 12:09:41.149538+03
 2023-07-26 11:11:13.193191+03 | 2023-07-26 11:27:16.67582+03
 2023-07-26 11:11:13.193191+03 | 2023-07-26 11:50:15.106529+03
 2023-07-26 11:11:13.193191+03 | 2023-07-26 11:57:27.417962+03
(10 rows)
```

Вторую функцию (с параметром-интервалом) можно определить через первую:

```
=> CREATE FUNCTION rnd_timestamp(t_start timestamptz, t_delta interval)
RETURNS timestamptz
AS $$
    SELECT rnd_timestamp(t_start, t_start + t_delta);
$$ VOLATILE LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
=> SELECT rnd_timestamp(current_timestamp, interval '1 hour');
```

```
        rnd_timestamp
------------------------------
 2023-07-26 12:08:08.34104+03
(1 row)
```

2. Автомобильные номера

Создадим таблицу с номерами.

```
=> CREATE TABLE cars(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    regnum text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO cars(regnum) VALUES
    ('K 123 XM'), ('k123xm'), ('A 098BC');
```

```
INSERT 0 3
```

Функция нормализации:

```
=> CREATE FUNCTION normalize(regnum text) RETURNS text
AS $$
    SELECT translate(upper(regnum), 'ABEKMHOPCTYX ', 'ABEKMHOPCTYX');
$$ IMMUTABLE LANGUAGE sql;
```

```
CREATE FUNCTION
```

Категория изменчивости — immutable. Функция всегда возвращает одинаковое значение при одних и тех же входных параметрах.

```
=> SELECT normalize(regnum) FROM cars;

 normalize
-----------
 K123XM
 K123XM
 A098BC
(3 rows)
```

Теперь легко найти дубликаты:

```
=> CREATE FUNCTION num_unique() RETURNS bigint
AS $$
    SELECT count(DISTINCT normalize(regnum))
    FROM cars;
$$ STABLE LANGUAGE sql;

CREATE FUNCTION

=> SELECT num_unique();

 num_unique
------------
          2
(1 row)
```

3. Корни квадратного уравнения

```
=> CREATE FUNCTION square_roots(
    a float,
    b float,
    c float,
    x1 OUT float,
    x2 OUT float
)
AS $$
WITH discriminant(d) AS (
    SELECT b*b - 4*a*c
)
SELECT CASE WHEN d >= 0.0 THEN (-b + sqrt(d))/2/a END,
       CASE WHEN d >  0.0 THEN (-b - sqrt(d))/2/a END
FROM discriminant;
$$ IMMUTABLE LANGUAGE sql;

CREATE FUNCTION
```

Категория изменчивости — immutable. Функция всегда возвращает одинаковое значение при одних и тех же входных параметрах.

```
=> SELECT square_roots(1,  0, -4);

 square_roots
--------------
 (2,-2)
(1 row)


=> SELECT square_roots(1, -4,  4);

 square_roots
--------------
 (2,)
(1 row)


=> SELECT square_roots(1,  1,  1);

 square_roots
--------------
 (,)
(1 row)
```