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

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 их 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
- обычно 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 из них 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 состоит из имени, необязательных параметров, типа возвращаемого значения и тела. Тело записывается в виде строковой константы, которая содержит код на выбранном языке. За счет этого определение функции выглядит одинаково независимо от выбранного языка. Тело-строка сохраняется в системном каталоге и интерпретируется каждый раз, когда функция вызывается. Начиная с версии PostgreSQL 14 для кода на SQL появилась возможность производить разбор заранее, в системном каталоге при этом сохраняется не исходный текст, а результат разбора. Еще один способ избежать интерпретации времени выполнения — написать функцию на языке Си, но 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/16/sql-createfunction

https://postgrespro.ru/docs/postgresql/16/sql-syntax-calling-funcs

Функции без параметров

Вот простой пример функции без параметров, содержащей один оператор:

```
=> 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)
```

Давайте взглянем на то, как тело функции хранится в системном каталоге.

```
=> \pset xheader_width 60
```

```
Expanded header width is 60.
```

```
=> SELECT proname, prosrc, prosqlbody FROM pg_proc
WHERE proname = 'hello_world' \gx
```

```
-[ RECORD 1 ]------------------------
proname    | hello_world
prosrc     |  SELECT 'Hello, world!';
prosqlbody |
```

Мы видим сохраненную в исходном виде тело-строку.

А теперь реализуем современную возможность пересоздать нашу функцию в другом виде — в стиле стандарта SQL. В нашем случае телом функции может быть один оператор вида RETURN <выражение>:

```
=> CREATE OR REPLACE FUNCTION hello_world() RETURNS text
LANGUAGE sql
RETURN 'Hello, world!';
```

```
CREATE FUNCTION
```

Снова заглянем в системный каталог — тело функции сохранено по-другому:

```
=> SELECT proname, prosrc, left(prosqlbody, 100) AS body FROM pg_proc WHERE proname = 'hello_world' \gx
```

```
-[ RECORD 1 ]-----------------------------------------------
proname | hello_world
prosrc  |
body    | {QUERY :commandType 1 :querySource 0 :canSetTag true :utilityStmt <>
:resultRelation 0 :hasAggs fals
```

Исходный код в этом случае не хранится, получить его можно командой \sf:

```
=> \sf hello_world
```

```
CREATE OR REPLACE FUNCTION public.hello_world()
 RETURNS text
 LANGUAGE sql
RETURN 'Hello, world!'::text
```

В случае, если тело функции состоит из нескольких операторов SQL, в качестве результата возвращается значение из первой строки, которую вернул последний оператор. Если код такой функции написан в стиле стандарта SQL, потребуется использовать конструкцию BEGIN ATOMIC ... END, которая охватывает выполняемый блок операторов:

```
=> CREATE OR REPLACE FUNCTION hello_world() RETURNS text
LANGUAGE sql
BEGIN ATOMIC
  SELECT 'First Line';
  SELECT 'Second Line';
END;
```

```
CREATE FUNCTION
```

Пробуем вызов:

```
=> SELECT hello_world();
```

```
 hello_world
-------------
 Second Line
(1 row)
```

Обратите внимание на особенности синтаксиса стиля стандарта SQL — в отличие от традиционного «строчного»:

- нет конструкции AS, содержащей код функции в виде строки;
- может использоваться новое ключевое слово RETURN для возврата значения;
- указание LANGUAGE sql не является обязательным;
- при создании функции ее код разбирается, а результат разбора сохраняется в pg_proc.prosqlbody (в традиционной нотации текст функции сохраняется в pg_proc.prosrc).

Это лучше соответствует стандарту и в большей мере совместимо с другими реализациями SQL. Теперь при вызове функции ее команды заново не интерпретируются, а используется заранее разобранный вариант.

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
LANGUAGE sql
BEGIN ATOMIC COMMIT; END;
```

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

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

Функции с входными параметрами

Пример функции с одним параметром:

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

```
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
LANGUAGE sql
RETURN 'Hello, ' || $1 || '!'; -- номер вместо имени
```

```
CREATE FUNCTION
```

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

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

Но так лучше не делать, это неудобно.

Удалим функцию и создадим заново, добавив еще два параметра — приветствие и обращение.

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

```
DROP FUNCTION
```

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

```
=> CREATE FUNCTION hello(IN name text, IN greet text DEFAULT 'Dear', IN title text DEFAULT 'Mr')
RETURNS text
LANGUAGE sql
RETURN 'Hello, ' || greet || ' ' || title || ' ' || name || '!';
```

```
CREATE FUNCTION
```

```
=> SELECT hello('Alice', 'Charming', 'Mrs'); -- указаны второй и третий параметры
```

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

Обратите внимание, что параметры со значениями по умолчанию должны идти в конце всего списка. При вызове функции значения фактических параметров, определенных как default, можно опускать, тогда остальные default-параметры, идущие в списке после, также получат значения по умолчанию

```
=> SELECT hello('Bob', 'Excellent'); -- указан только первый default-параметр
```

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

```
=> SELECT hello('Bob'); -- опущены оба параметра, имеющие значение по умолчанию
```

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

Такой способ удобен, когда порядок параметров неочевиден, особенно если их много; также можно явно указать значение одного из параметров по умолчанию.

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

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

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

Если функция должна возвращать неопределенное значение, когда 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, text);
```

```
DROP FUNCTION
```

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

```
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-параметре — что позволяет большинство традиционных языков программирования.

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
)
LANGUAGE sql
RETURN 'Hello, ' || name || '!';
```

```
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
LANGUAGE sql
RETURN 'Hello, ' || name || '!';
```

```
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)
LANGUAGE sql
RETURN 'Hello, ' || name || '!';
```

```
CREATE FUNCTION
```

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

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

Обратите внимание, что, в отличие от многих языков программирования, 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)
LANGUAGE sql
RETURN ('Hello, ' || name || '!', current_time);

CREATE FUNCTION
```

Здесь возвращаемое RETURN выражение пришлось взять в скобки.

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

                hello
-------------------------------------
 ("Hello, Alice!",04:32:30.726895+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 выполняться каждый раз 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* выполнить 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/16/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
LANGUAGE sql VOLATILE
RETURN (SELECT count(*) FROM t);
```

```
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 при использовании более строгих уровней изоляции такого не происходит потому, что в этих случаях всегда используется снимок основного запроса. В предыдущем же примере (volatile, Read Committed) основной запрос и запросы в функции используют разные снимки данных.

```
=> 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
LANGUAGE sql VOLATILE
RETURN random();
```

```
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:

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

 generate_series
-----------------
               2
               4
               5
               6
               7
               8
              10
(7 rows)


=> \g

 generate_series
-----------------
               1
               2
               3
               5
               8
               9
(6 rows)


=> \g

 generate_series
-----------------
               1
               2
               3
               5
               7
(5 rows)


=> \g

 generate_series
-----------------
               5
               6
               8
               9
              10
(5 rows)


=> \g

 generate_series
-----------------
               1
               2
               4
               7
               8
               9
(6 rows)
```

Здесь с разной вероятностью получаем от 0 до 10 строк.

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)
```

Результатом запроса будет либо 0, либо 10 строк.

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

 generate_series
-----------------
(0 rows)


=> \g

 generate_series
-----------------
(0 rows)


=> \g

 generate_series
-----------------
(0 rows)


=> \g

 generate_series
-----------------
(0 rows)


=> \g

 generate_series
-----------------
               1
               2
               3
               4
               5
               6
               7
               8
               9
              10
(10 rows)


=> \g

 generate_series
-----------------
(0 rows)
```

Наконец, изменчивость immutable позволяет вычислить значение функции еще на этапе планирования, поэтому во время выполнения вычисление фильтра уже не требуется:

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

ALTER FUNCTION

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

            QUERY PLAN
----------------------------------
 Function Scan on generate_series
(1 row)


=> \g
```

```
            QUERY PLAN
---------------------------------
 Function Scan on generate_series
(1 row)


=> \g

            QUERY PLAN
---------------------------------
 Function Scan on generate_series
(1 row)


=> \g

            QUERY PLAN
---------------------------------
 Function Scan on generate_series
(1 row)


=> \g

            QUERY PLAN
---------------------------------
 Function Scan on generate_series
(1 row)


=> \g

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


=> \g

            QUERY PLAN
---------------------------------
 Function Scan on generate_series
(1 row)
```

Для immutable получаем случайный план!

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 функция random(), но не rnd(). Она будет вызываться 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 функцией 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, middle_name text
)
RETURNS text
```

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

3. `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). В нормально построенном процессе разработки весь код должен находиться 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.)

Практика+

3. Напишите функцию, находящую корни квадратного уравнения.

12

Во всех заданиях обратите особое внимание на категорию изменчивости функций.

3. Для уравнения вида $y = ax^2 + bx + c$ вычисляется 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).