

Функции

Зачем нужны функции SQL

Функция – объект БД, принимающий аргументы и возвращающий результат

Функции (хранимые процедуры) – компилируемы и хранятся на стороне БД

Разграничение работы Frontend и Backend

Переиспользуемость функций разными клиентскими приложениями

Управление безопасностью через регулирование доступа к функциям

Модульное программирование. Что если нужна генерация случайных чисел в разных SQL-скриптах? (использование процедурного расширения SQL – plpgsql)

Функция в Postgres

- Могут содержать SELECT, INSERT, UPDATE, DELETE
- Не могут содержать COMMIT, SAVEPOINT, VACCUUM (utility)

(функции транзакционны автоматически: если произошла ошибка внутри функции, все что было сделано – будет откатано)

Делятся:

- SQL-функции
- Процедурные (PL/pgSQL – основной диалект)
- Серверные функции
- Собственные C-функции

Синтаксис

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
{ LANGUAGE lang_name
  | TRANSFORM { FOR TYPE type_name } [, ... ]
  | WINDOW
  | { IMMUTABLE | STABLE | VOLATILE }
  | [ NOT ] LEAKPROOF
  | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
  | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
  | PARALLEL { UNSAFE | RESTRICTED | SAFE }
  | COST execution_cost
  | ROWS result_rows
  | SUPPORT support_function
  | SET configuration_parameter { TO value | = value | FROM CURRENT }
  | AS 'definition'
  | AS 'obj_file', 'link_symbol'
  | sql_body
} ...
```

DROP FUNCTION IF EXISTS *name_function*

Синтаксис

```
CREATE FUNCTION add(integer, integer) RETURNS integer
  AS 'select $1 + $2;'
  LANGUAGE SQL
  IMMUTABLE
  RETURNS NULL ON NULL INPUT;
```

IMMUTABLE указывает, что функция не может изменять базу данных и всегда возвращает один и тот же результат при задании одних и тех же значений аргументов.

RETURNS NULL ON NULL INPUT указывает, что функция всегда возвращает значение null всякий раз, когда какой-либо из ее аргументов имеет значение null. Если этот параметр указан, функция не выполняется при наличии пустых аргументов; вместо этого автоматически предполагается нулевой результат.

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
  AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
  LANGUAGE SQL;

SELECT * FROM dup(42);
```

Синтаксис

То же самое можно сделать более подробно с явно названным составным типом:

```
CREATE TYPE type_result AS (f1 int, f2 text);  
CREATE function result(int) RETURNS type_result  
AS  
$$  
SELECT $1, CAST ($1 as TEXT) || ' is text'  
$$  
LANGUAGE SQL;  
SELECT * from result(10)
```

```
CREATE OR REPLACE function result(int) RETURNS TABLE (f1 int, f2 text)  
AS  
$$  
SELECT $1, CAST ($1 as TEXT) || 'is text'  
$$  
LANGUAGE SQL;
```

Однако функция отличается от предыдущих примеров, поскольку она фактически возвращает набор записей, а не только одну запись

Синтаксис

```
CREATE OR REPLACE FUNCTION get_sum_price() RETURNS bigint AS  
$$  
SELECT SUM(price)  
FROM products  
$$ LANGUAGE SQL;
```

```
SELECT get_sum_price() AS sum_price;
```

```
CREATE OR REPLACE FUNCTION get_avg_price() RETURNS float8 AS  
$$  
SELECT AVG(price)  
FROM products  
$$ LANGUAGE SQL;
```

```
SELECT get_avg_price() AS avg_price;
```

Синтаксис

- IN – входящие аргументы
- OUT – исходящие аргументы
- VARIADIC – массив входящих параметров
- DEFAULT *value* (значение параметра можно не передавать)

```
CREATE OR REPLACE FUNCTION get_product_price_by_name(prod_name varchar) RETURNS real AS  
$$  
    SELECT price  
    FROM products  
    WHERE product_name = prod_name  
$$ LANGUAGE SQL;
```

```
SELECT get_product_price_by_name('Шоколад') as price;
```


Синтаксис

Наименьшая и наибольшая цена

```
CREATE OR REPLACE FUNCTION get_price_max_min(OUT max_price real, OUT min_price real) AS  
$$  
    SELECT MAX(price), MIN(price)  
    FROM products  
$$ LANGUAGE SQL;
```

SELECT get_price_max_min(); // результат в одной колонке {,}

	get_price_max_min record
1	(10000,2000)

SELECT *
FROM get_price_max_min(); // результат в разных колонках

	max_price real	min_price real
1	10000	2000

Синтаксис

Наименьшая и наибольшая цена

```
CREATE OR REPLACE FUNCTION get_price_by_categories(IN id_category int, OUT max_price real, OUT min_price real) AS
```

```
$$
```

```
    SELECT MAX(price), MIN(price)
```

```
    FROM products
```

```
    WHERE category = id_category
```

```
$$ LANGUAGE SQL;
```

```
SELECT *
```

```
FROM get_price_by_categories(1);
```

```
CREATE OR REPLACE FUNCTION get_price_by_categories_default  
(id_category int DEFAULT 1, OUT max_price real, OUT min_price real)
```

```
SELECT *
```

```
FROM get_price_by_categories_default();
```

Синтаксис

- RETURNS SETOF *data_type* – возврат n- значений типа
- RETURNS SETOF *table* – если нужно вернуть все столбцы из таблицы или пользовательского типа
- RETURNS SETOF *record* - только когда типы колонок в результирующем наборе заранее неизвестны
- RETURNS TABLE (*column_name data_type,...*) – тоже что и SETOF *table*, но имеем возможность явно указать возвращаемые столбцы

Модификатор SETOF указывает, что функция будет возвращать набор элементов, а не один элемент

СИНТАКСИС

```
CREATE OR REPLACE FUNCTION get_average_prices_by_prod_categories()  
    RETURNS SETOF double precision AS
```

```
$$
```

```
    SELECT AVG(price)  
    FROM products  
    GROUP BY category_id
```

```
    SELECT *  
    FROM get_average_prices_by_prod_categories() AS  
    average_prices;
```

```
$$ LANGUAGE SQL;
```

```
CREATE OR REPLACE FUNCTION get_sum_avg_prices_by_prod_categories(OUT sum_price real, OUT  
avg_price float8)
```

```
    RETURNS SETOF RECORD AS
```

```
$$
```


```
    SELECT SUM(price), AVG(price)  
    FROM products  
    GROUP BY category_id
```

```
    SELECT sum_price AS sum, avg_price AS avg  
    FROM get_sum_avg_prices_by_prod_categories()
```

```
$$ LANGUAGE SQL;
```

СИНТАКСИС

```
CREATE OR REPLACE FUNCTION get_price_by_categories(VARIADIC id_categories int[])  
RETURNS SETOF character(20) AS  
$$  
SELECT product_name  
FROM products  
WHERE category_id IN (SELECT unnest( id_categories ))  
$$ LANGUAGE SQL;
```



разворачивает массив в виде
столбца таблицы

```
SELECT * FROM get_price_by_categories(1,2,...);
```

Синтаксис

```
CREATE OR REPLACE FUNCTION get_sum_avg_prices_by_prod_categories()  
    RETURNS SETOF RECORD AS
```

```
$$
```

```
    SELECT SUM(price), AVG(price)
```

```
    FROM products
```

```
    GROUP BY category_id
```

```
$$ LANGUAGE SQL;
```



КОГДА НЕ ЗНАЕМ, ЧТО ВОЗВРАЩАЕМ

```
SELECT *
```

```
FROM get_sum_avg_prices_by_prod_categories() – ОШИБКА: должен быть список с определением столбцов  
(анонимный тип)
```

```
SELECT *
```

```
FROM get_sum_avg_prices_by_prod_categories()
```

```
AS (sum_price real, avg_price float8)
```

Синтаксис

```
CREATE OR REPLACE FUNCTION get_customers_by_country(customer_country varchar)
    RETURNS TABLE(char_code char, company_name varchar) AS
```

```
$$
```

```
    SELECT customer_id, company_name
```

```
    FROM customers
```

```
    WHERE country = customer_country
```

```
$$ LANGUAGE SQL;
```

char_code= customer_id

company_name= company_name

```
SELECT * FROM get_customers_by_country('Россия');
```

```
CREATE OR REPLACE FUNCTION get_customers_by_country_table(customer_country varchar)
    RETURNS SETOF customers AS
```

```
$$
```

```
-- Не работает: SELECT company_name, contact_name
```

```
SELECT *
```

```
FROM customers
```

```
WHERE country = customer_country
```

```
$$ LANGUAGE SQL;
```

customers – имя таблицы

PL/pgSQL – это процедурный язык СУБД PostgreSQL. Он может использоваться для создания обычных функций и триггерных функций.

Этот язык позволяет дополнить язык SQL управляющими структурами. С его помощью можно выполнять сложные вычисления.

Функции, написанные на этом языке, могут использоваться везде, где могли бы использоваться встроенные функции языка SQL, например, в индексных выражениях при создании индексов.

Данный язык позволяет повысить эффективность работы приложения с базой данных за счет того, что в рамках одной процедуры, написанной на этом языке, могут быть сгруппированы несколько SQL-операторов, которые хранятся на сервере.

Поэтому клиентскому приложению не требуется выполнять эти SQL-операторы по одному, организуя каждый раз взаимодействие с сервером и тем самым увеличивая сетевой трафик.

Также не выполняется передача промежуточных результатов вычислений от сервера к клиенту, тем самым также сокращается число взаимодействий клиента и сервера, что позволяет ускорить обработку данных.

- Функции на языке PL/pgSQL оформляются в виде блоков (в квадратных скобках указаны необязательные элементы):

[<<метка>>]

[DECLARE

объявления]

BEGIN

операторы

END [метка] ;

- Внутри блока могут содержаться вложенные блоки, которые удобно использовать для отражения логической структуры функции. Переменные, объявленные во вложенном блоке, скрывают одноименные переменные, объявленные во внешнем блоке.
- Все ключевые слова являются нечувствительными к регистру символов, поэтому их можно вводить как в верхнем, так и в нижнем регистре.

Синтаксис

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$  
    BEGIN  
        RETURN i + 1;  
    END;  
$$ LANGUAGE plpgsql;
```

BEGIN/END – тело метода

Создание переменных

Прогон циклов и развитая логика

Возврат значения через RETURN (вместо SELECT) или RETURN QUERY

```
CREATE OR REPLACE FUNCTION get_sum_of_goods() RETURNS bigint AS  
$$  
BEGIN  
    RETURN SUM(price)  
    FROM products;  
END;  
SELECT get_sum_of_goods();  
$$ LANGUAGE plpgsql;
```

Синтаксис

```
CREATE OR REPLACE FUNCTION get_price_max_min(OUT max_price real, OUT min_price real)
AS
$$
BEGIN
    max_price:= MAX(price) FROM products;
    min_price:= MIN(price) FROM products;
END;
$$ LANGUAGE plpgsql
```

```
CREATE OR REPLACE FUNCTION get_price_max_min(OUT max_price real, OUT min_price real) AS
$$
BEGIN
    SELECT MAX(price), MIN(price)
    INTO max_price, min_price
    FROM products;
END;
$$ LANGUAGE plpgsql
```

Синтаксис

```
CREATE OR REPLACE FUNCTION get_customers_by_country(customer_country varchar)
RETURNS SETOF customers AS $$
BEGIN
    RETURN QUERY
    SELECT *
    FROM customers
    WHERE country = customer_country;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT * FROM get_customers_by_country('Россия')
```

- Переменные в языке PL/pgSQL могут иметь любой тип данных, имеющийся в PostgreSQL, например integer, varchar и т. д.

DEFAULT означает присваивание :=

```
quantity integer DEFAULT 32;
```

- строковое значение нужно заключить в одинарные кавычки

```
url varchar := 'http://mysite.com';
```

- можно создать константу и инициализировать ее

```
user_id CONSTANT integer := 10;
```

- переменная для хранения значения поля user_id из таблицы users, Такой оператор избавляет нас от необходимости знать тип данных этого поля

```
user_id users.user_id%TYPE;
```

Синтаксис

IF логическое-выражение THEN

операторы

[ELSIF логическое-выражение THEN операторы [ELSIF логическое-выражение THEN операторы ...]]

[ELSE операторы]

END IF;

CREATE OR REPLACE FUNCTION assessment_of_student(grade int) RETURNS text AS

\$\$

DECLARE

assessment text = '';

BEGIN

IF grade BETWEEN 0 AND 59 THEN assessment := 'неудовлетворительно';

ELSIF grade BETWEEN 60 AND 69 THEN assessment := 'удовлетворительно';

ELSIF grade BETWEEN 70 AND 89 THEN assessment := 'хорошо';

ELSE assessment := 'отлично';

END IF;

RETURN assessment;

END;

\$\$ LANGUAGE plpgsql;

select assessment_of_student(70)

assessment_of_student	
	text
1	хорошо

Синтаксис

WHILE логическое-выражение
LOOP
 операторы
END LOOP;

LOOP
 EXIT WHEN логическое-выражение
 операторы
END LOOP;

FOR цель IN запрос LOOP
 операторы
END LOOP [метка];

Переменная *запрос* может быть переменной-кортежем, переменной типа record или разделённым запятыми списком скалярных переменных.

Переменной *цель* последовательно присваиваются строки результата запроса, и для каждой строки выполняется тело цикла

FOR i IN 1..10
LOOP
 -- внутри цикла переменная i будет иметь значения 1,2,3,4,5,6,7,8,9,10
END LOOP;

FOR i IN REVERSE 10..1 BY 2
LOOP
 -- внутри цикла переменная i будет иметь значения 10,8,6,4,2
END LOOP;

- Для вывода сообщений пользователю или для генерирования ошибок служит команда RAISE. Покажем один из вариантов ее син-таксиса.

```
RAISE [ уровень ] 'формат' [, выражение [, ... ]];
```

- Здесь уровень означает степень серьезности сообщения: DEBUG, LOG, INFO, NOTICE, WARNING и EXCEPTION. По умолчанию используется EXCEPTION, что означает формирование ошибки. Параметр 'формат' служит для формирования текста сообщения, за этим параметром могут следовать переменные, значения которых подставляются в строку 'формат' в те позиции, которые обозначены символом «%».

Приведем простой пример:

```
RAISE NOTICE 'Товар % продается по цене %',  
            mviews.product_name,  
            mviews.price;
```


Синтаксис

```
CREATE FUNCTION fun1() RETURNS integer AS
$$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Выполнение функции..';
    FOR mviews IN
        SELECT * FROM products ORDER BY 1
    LOOP
        RAISE NOTICE 'Товар % продается по цене %',
            mviews.product_name,
            mviews.price;

    END LOOP;
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

ЗАМЕЧАНИЕ: Выполнение функции..

ЗАМЕЧАНИЕ: Товар Шоколад продается по цене 10000

ЗАМЕЧАНИЕ: Товар Карамель продается по цене 2000

ЗАМЕЧАНИЕ: Товар Батончик продается по цене 6000

Successfully run. Total query runtime: 64 msec.

1 rows affected.

Синтаксис

Цикл по результатам запроса

```
CREATE FUNCTION refresh_mviews() RETURNS SETOF products AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Выполнение функции..';

    FOR mviews IN
        SELECT * FROM products ORDER BY 1
    LOOP
        mviews.price=mviews.price*0.5
        RETURN NEXT product;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Синтаксис

```
create function get_product_info (p_n text) RETURNS TABLE (category bigint, price real) AS  
$$
```

```
DECLARE
```

```
tmp char(1);
```

```
product RECORD;
```

```
BEGIN
```

```
SELECT 'x' INTO tmp FROM products WHERE product_name = p_n;
```

```
IF NOT FOUND THEN
```

```
RAISE NOTICE 'Продукта % нет в базе данных', p_n;
```

```
RETURN;
```

```
END IF;
```

```
FOR product IN SELECT * FROM products
```

```
WHERE product_name=p_n
```

```
LOOP
```

```
RETURN QUERY SELECT product.category_id, product.price;
```

```
END LOOP;
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

возвращаем
таблицу

Тип RECORD. Ее
структура заранее не
определена

```
SELECT * FROM get_product_info('Мороженое');
```

ЗАМЕЧАНИЕ: Продукта Мороженое нет в базе данных

Successfully run. Total query runtime: 49 msec.
0 rows affected.

```
SELECT * FROM get_product_info('Шоколад');
```

	category bigint	price real
1	1	10000