

Оптимизация запросов Функции



Авторские права

© Postgres Professional, 2019–2024

Авторы: Егор Рогов, Павел Лузанов, Павел Толмачев, Илья Баштанов

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

Использование материалов курса

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

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

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

Категории изменчивости

Подстановка кода функции в текст запроса

Вызов табличных функций

Настройки COST и ROWS

Вспомогательные функции планировщика

Конфигурационные параметры

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.

<https://postgrespro.ru/docs/postgresql/16/xfunc-volatility>

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 обращение к табличной функции generate_series в узле Function Scan. Каждая строка результата сравнивается со случайным числом и при необходимости отбрасывается фильтром, в котором вычисляется скалярная функция random.

В этом можно убедиться и воочию:

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

```
generate_series
-----
1
3
4
6
7
8
9
(7 rows)
```

```
=> \g
```

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

```
=> \g
```

```
generate_series
-----
1
2
7
8
9
(5 rows)
```

```
=> \g
```

```
generate_series
-----
1
2
4
5
7
9
10
(7 rows)
```

=> \g

```
generate_series
-----
1
2
8
(3 rows)
```

Здесь с разной вероятностью получаем от 0 до 10 строк.

Функция с категорией изменчивости 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)
```

Узел Result формирует строку выборки, а выражение One-Time Filter вычисляется один раз, так что результатом запроса будет либо 0, либо 10 строк.

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

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

=> \g

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

=> \g

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

Наконец, категория Immutable позволяет вычислить значение функции еще на этапе планирования, поэтому во время выполнения вычисление условия 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

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

```
=> \g
```

QUERY PLAN

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

Для Immutable получаем случайный план!

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

Скалярные функции на SQL

- один оператор SELECT без предложения FROM,
- возвращает одно значение
- вызываемые функции не должны быть изменчивее вызывающей
- и др.

Табличные функции на SQL

- один оператор SELECT
- категория Immutable или Stable
- функция не STRICT
- и др.

Оптимизатор PostgreSQL умеет подставлять (inline) тело функции в SQL-запрос. Это работает как со скалярными, так и с табличными функциями.

В обоих случаях есть много ограничений: функция должна быть написана на языке SQL, использовать единственный оператор SELECT и т. д. Скалярная функция, к тому же, не должна обращаться к таблицам базы данных и вызывать функции, имеющие менее строгую категорию, а табличная — быть стабильной или постоянной.

Важным преимуществом подстановки тела функции в запрос является то, что функция становится прозрачной для планировщика. Например, дополнительные условия в теле основного запроса могут быть применены к запросу из тела функции, что позволит как можно раньше отфильтровать лишние строки.

https://wiki.postgresql.org/wiki/Inlining_of_SQL_functions

Подстановка кода функций в SQL-запрос

Тело очень простых скалярных функций на языке SQL может быть подставлено прямо в основной SQL-оператор на этапе разбора запроса. В этом случае время на вызов функции не тратится.

Пример мы уже видели: наша функция rnd().

Проверим, какой будет план запроса в случае, когда категория изменчивости функции rnd (Stable) не соответствует категории изменчивости функции random (Volatile):

```
=> 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)
```

В фильтре упоминается функция rnd().

Поменяем категорию изменчивости функции на Volatile:

```
=> 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)
```

Теперь в фильтре упоминается функция random(), но не rnd(). Она будет вызываться напрямую, минуя «обертку» в виде функции rnd().

Возможностей для подстановки табличных функций гораздо больше. Например, в таких функциях допускаются обращения к таблицам.

```
=> CREATE FUNCTION flights_from(airport_name text)
RETURNS SETOF flights
AS $$
  SELECT f.*
  FROM flights f
  JOIN airports a ON f.departure_airport = a.airport_code
  WHERE a.airport_name = flights_from.airport_name;
$$
LANGUAGE sql STABLE;
```

```
CREATE FUNCTION
```

При подстановке табличные функции работают наподобие представлений с параметрами. Планировщик оптимизирует весь запрос, функция прозрачна для него:

```
=> EXPLAIN (costs off)
SELECT *
FROM flights_from('Оренбург')
WHERE status = 'Arrived';
```


QUERY PLAN

Hash Join

Hash Cond: (f.departure_airport = ml.airport_code)

-> Seq Scan on flights f

Filter: ((status)::text = 'Arrived'::text)

-> Hash

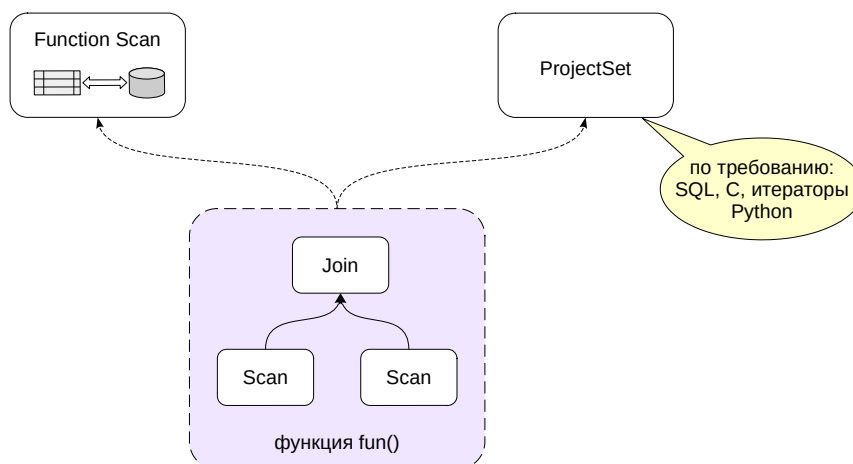
-> Seq Scan on airports_data ml

Filter: ((airport_name ->> lang()) = '0pen6ыр'::text)

(7 rows)

SELECT * FROM fun()

SELECT fun()



Когда табличная функция вызывается в предложении FROM, за ее вычисление отвечает узел Function Scan. При этом сначала все строки, возвращаемые функцией, материализуются, и только затем передаются родительскому узлу плана. Таково текущее ограничение реализации.

Если же функция вызывается в предложении SELECT, в плане она выполняется в узле ProjectSet. В этом случае функция может воспользоваться интерфейсом возвращения строк «по требованию» (value-per-call), без материализации.

Так работают функции на языке SQL, большинство встроенных функций (написанных на C) и функции на PL/Python, возвращающие итератор. Функции на других языках программирования тоже могут использовать этот интерфейс, если такая возможность в них реализована.

Табличные функции

Вызовем функцию `generate_series` в предложении `FROM` с ограничением на количество строк (`LIMIT`):

```
=> \set timing on
```

```
=> EXPLAIN (analyze, costs off)
SELECT * FROM generate_series(1,10_000_000)
LIMIT 10;
```

QUERY PLAN

```
-----
Limit (actual time=875.922..875.938 rows=10 loops=1)
  -> Function Scan on generate_series (actual time=875.919..875.929 rows=10 loops=1)
Planning Time: 0.042 ms
Execution Time: 1021.103 ms
(4 rows)
```

В плане запроса видим узел `Function Scan` — сначала были получены все строки из функции, и только потом наложено ограничение `LIMIT`.

А теперь повторим запрос, только вызовем функцию из предложения `SELECT`:

```
=> EXPLAIN (analyze, costs off)
SELECT generate_series(1,10_000_000)
LIMIT 10;
```

QUERY PLAN

```
-----
Limit (actual time=0.003..0.005 rows=10 loops=1)
  -> ProjectSet (actual time=0.003..0.003 rows=10 loops=1)
    -> Result (actual time=0.001..0.001 rows=1 loops=1)
Planning Time: 0.041 ms
Execution Time: 0.014 ms
(5 rows)
```

Теперь в плане появился узел `ProjectSet`, который формирует десять строк выборки — в данном случае оптимизатору удалось получать строки по требованию. Время выполнения запроса сократилось на порядки.

Узел `Result` здесь представляет опущенное в запросе предложение `FROM` — он передает родительскому узлу ровно одну строку.

```
=> \set timing off
```

Однако не все функции могут возвращать строки по одной. Например, обращение к любой функции на языке PL/pgSQL возвращает все строки результата:

```
=> CREATE FUNCTION plpgsql_rows() RETURNS SETOF integer
AS $$
BEGIN
  RETURN QUERY
    SELECT * FROM generate_series(1,10_000_000);
END
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Вызовем функцию из предложения `SELECT`:

```
=> \timing on
```

Timing is on.

```
=> EXPLAIN (analyze, costs off)
SELECT plpgsql_rows() LIMIT 10;
```

QUERY PLAN

```
-----
Limit (actual time=2103.501..2103.504 rows=10 loops=1)
  -> ProjectSet (actual time=2103.500..2103.501 rows=10 loops=1)
    -> Result (actual time=0.000..0.001 rows=1 loops=1)
Planning Time: 0.031 ms
Execution Time: 2156.213 ms
(5 rows)
```

Time: 2156,484 ms (00:02,156)

```
=> \timing off
```

Timing is off.

В плане — узел ProjectSet, но теперь серверу приходится получить результат функции полностью, а затем наложить ограничение LIMIT. Время выполнения это хорошо показывает.

В предложении SELECT может быть несколько вызовов табличных функций, а сами функции могут вкладываться друг в друга:

```
=> SELECT generate_series(1, generate_series(1,3)), unnest(ARRAY['A','B','C']);
```

generate_series	unnest
1	A
1	B
2	B
1	C
2	C
3	C

(6 rows)

```
=> EXPLAIN (verbose, costs off)
```

```
SELECT generate_series(1, generate_series(1,3)), unnest(ARRAY['A','B','C']);
```

QUERY PLAN

```
ProjectSet
  Output: generate_series(1, (generate_series(1, 3))), (unnest('{A,B,C}'::text[]))
  -> ProjectSet
    Output: generate_series(1, 3), unnest('{A,B,C}'::text[])
    -> Result
(5 rows)
```

Нижний узел ProjectSet формирует выборку из результатов выполнения двух табличных функций: generate_series(1,3) и unnest. В этой выборке оказываются три строки.

Верхний узел ProjectSet формирует итоговую выборку, вычисляя внешний вызов generate_series.

Без учета материализации такой запрос эквивалентен следующему запросу, в котором функции вызываются в предложении FROM:

```
=> SELECT g2.i, u.c
```

```
FROM generate_series(1,3) WITH ORDINALITY AS g1(i)
FULL JOIN LATERAL unnest(ARRAY['A','B','C']) WITH ORDINALITY AS u(c)
ON g1.ordinality = u.ordinality
CROSS JOIN LATERAL generate_series(1, g1.i) AS g2(i);
```

i	c
1	A
1	B
2	B
1	C
2	C
3	C

(6 rows)

Настройки COST и ROWS

```
CREATE FUNCTION fun( )
```

```
SELECT * FROM fun( )
```

Function Scan (rows=1000 cost=100)

оценки
по умолчанию

```
CREATE FUNCTION fun( ) ROWS 12 COST 123
```

```
SELECT * FROM fun( )
```

Function Scan (rows=12 cost=123)

ROWS

COST

Обычно (если не удалось подставить тело функции в запрос) оптимизатор не имеет возможности анализировать код функции и воспринимает ее как «черный ящик».

Однако можно дать оптимизатору приблизительную информацию о стоимости вызова функции и числе возвращаемых строк.

Параметр COST задает стоимость пользовательской функции в единицах *cpu_operator_cost*. По умолчанию функции на С получают оценку 1, а на других языках — 100.

Параметр ROWS указывает примерное число возвращаемых строк.

COST и ROWS можно указывать как при создании функции, так и для уже существующих функций.

<https://postgrespro.ru/docs/enterprise/16/sql-createfunction>

Настройки COST и ROWS

Напишем табличную функцию на языке PL/pgSQL, выводящую дни недели:

```
=> CREATE FUNCTION days_of_week() RETURNS SETOF text
AS $$
BEGIN
    FOR i IN 7 .. 13 LOOP
        RETURN NEXT to_char(to_date(i::text, 'J'), 'TMDy');
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

```
=> SELECT * FROM days_of_week();
```

```
days_of_week
-----
Пн
Вт
Ср
Чт
Пт
Сб
Вс
(7 rows)
```

План запроса:

```
=> EXPLAIN
SELECT * FROM days_of_week();
```

```
              QUERY PLAN
-----
Function Scan on days_of_week (cost=0.25..10.25 rows=1000 width=32)
(1 row)
```

Стоимость выполнения функции считается постоянной, для пользовательских функций по умолчанию она равна стоимости 100 операторов:

```
=> SELECT 100 * current_setting('cpu_operator_cost')::float;

?column?
-----
      0.25
(1 row)
```

Но это значение можно изменить:

```
=> ALTER FUNCTION days_of_week COST 1000;
```

ALTER FUNCTION

```
=> EXPLAIN
SELECT * FROM days_of_week();
```

```
              QUERY PLAN
-----
Function Scan on days_of_week (cost=2.50..12.50 rows=1000 width=32)
(1 row)
```

В плане изменилась начальная стоимость.

Сервер оценивает кардинальность результата этой функции как 1000, хотя фактическое значение равно семи.

С помощью указания ROWS можно подсказать серверу ориентировочное количество строк, которое вернет функция:

```
=> ALTER FUNCTION days_of_week ROWS 10;
```

ALTER FUNCTION

Повторим запрос:

```
=> EXPLAIN
```

```
SELECT * FROM days_of_week();
```

QUERY PLAN

```
-----  
Function Scan on days_of_week (cost=2.50..2.60 rows=10 width=32)  
(1 row)
```

Теперь сервер считает, что функция вернет 10 строк, поэтому уменьшилась полная стоимость узла:

```
=> SELECT 1000 * current_setting('cpu_operator_cost')::float  
      + 10 * current_setting('cpu_tuple_cost')::float;
```

```
?column?
```

```
-----  
2.6  
(1 row)
```

Измененные значения можно увидеть в системном каталоге:

```
=> SELECT procost, prorows FROM pg_proc WHERE proname='days_of_week';
```

```
procost | prorows  
-----+-----  
1000 | 10  
(1 row)
```

Или с помощью метакоманды psql:

```
=> \sf days_of_week
```

```
CREATE OR REPLACE FUNCTION bookings.days_of_week()  
  RETURNS SETOF text  
  LANGUAGE plpgsql  
  COST 1000 ROWS 10  
AS $function$  
BEGIN  
  FOR i IN 7 .. 13 LOOP  
    RETURN NEXT to_char(to_date(i::text,'J'),'TMDy');  
  END LOOP;  
END;  
$function$
```

CREATE FUNCTION fun(x) ROWS 18

SELECT * FROM fun(1)

Function Scan (rows=18)

ROWS

SELECT * FROM fun(2)

Function Scan (rows=18)

CREATE FUNCTION fun(x) SUPPORT fun_support

SELECT * FROM fun(1)

Function Scan (rows=12)

fun_support(1)

SELECT * FROM fun(2)

Function Scan (rows=24)

fun_support(2)

Параметры COST и ROWS позволяют задать стоимость и количество строк, возвращаемых функцией, как постоянные величины. Но константы не всегда дают желаемый результат.

В PostgreSQL есть возможность для функции написать *вспомогательную функцию*, которая предоставляет планировщику информацию, зависящую от значений аргументов основной (*целевой*) функции.

Вспомогательная функция может по значениям аргументов целевой функции выдавать:

- оценку ее стоимости;
- оценку числа возвращаемых строк;
- выражение, эквивалентное вызову функции.

Дополнительно для функций, возвращающих boolean:

- оценку селективности;
- эквивалентный предикат с индексируемым оператором.

Вспомогательная функция должна быть написана на языке C.

<https://postgrespro.ru/docs/postgresql/16/xfunc-optimization>

Вспомогательные функции планировщика

Посмотрим план запроса с вызовом функции `generate_series`:

```
=> EXPLAIN
```

```
SELECT n FROM generate_series(1,5) n;
```

QUERY PLAN

```
-----  
Function Scan on generate_series n  (cost=0.00..0.05 rows=5 width=4)  
(1 row)
```

В отличие от функции `days_of_week`, оптимизатор сразу правильно оценивает количество возвращаемых строк. Более того, оценка зависит от параметров функции:

```
=> EXPLAIN
```

```
SELECT n FROM generate_series(1,15) n;
```

QUERY PLAN

```
-----  
Function Scan on generate_series n  (cost=0.00..0.15 rows=15 width=4)  
(1 row)
```

Благодаря вспомогательной функции (она может быть написана только на языке C) планировщик получает дополнительную информацию, которую использует для вычисления селективности условий, кардинальности функции или ее стоимости.

Посмотреть, имеется ли вспомогательная функция, можно в таблице `pg_proc`:

```
=> SELECT left(pg_get_function_arguments(p.oid), 57) proargtypes, prosupport  
FROM pg_proc p  
WHERE p.proname = 'generate_series';
```

proargtypes	prosupport
integer, integer, integer	generate_series_int4_support
integer, integer	generate_series_int4_support
bigint, bigint, bigint	generate_series_int8_support
bigint, bigint	generate_series_int8_support
numeric, numeric, numeric	-
numeric, numeric	-
timestamp without time zone, timestamp without time zone,	-
timestamp with time zone, timestamp with time zone, inter	-
timestamp with time zone, timestamp with time zone, inter	-

(9 rows)

Как видно, вспомогательные функции существуют не для всех перегруженных вариантов функции `generate_series`.

Посмотрим на план запроса, генерирующего ряд дат:

```
=> EXPLAIN SELECT *
```

```
FROM generate_series(now(), now() + interval '5 day', '1 day');
```

QUERY PLAN

```
-----  
Function Scan on generate_series  (cost=0.01..10.01 rows=1000 width=8)  
(1 row)
```

Без вспомогательной функции и указания `ROWS` оптимизатор не имеет информации о числе строк в результате и поэтому использует значение по умолчанию (1000).

С каждой версией в PostgreSQL появляются новые вспомогательные функции.

CREATE FUNCTION fun() PARALLEL ...

UNSAFE

небезопасные для распараллеливания
(по умолчанию)

RESTRICTED

ограниченно распараллеливаемые

SAFE

безопасные для распараллеливания

13

В теме «Параллельная обработка» было рассказано про то, что не каждый запрос может выполняться в параллельном режиме. Поскольку оптимизатор не может проанализировать тело функции, он рассчитывает на пометки параллельности, определяя по ним возможность параллельной обработки.

При создании функции (или позже) можно указать одну из трех пометок:

- **UNSAFE** — запрещаются параллельные планы, если в запросе есть вызов функции;
- **RESTRICTED** — разрешаются параллельные планы, но запрещено вызывать функцию в параллельной части плана;
- **SAFE** — безопасна для параллельной обработки.

По умолчанию используется пометка **UNSAFE**.

Пометки параллельности указываются также для пользовательских агрегатных функций.

<https://postgrespro.ru/docs/postgresql/16/parallel-safety#PARALLEL-LABELING>

Пометки параллельности

Пометки параллельности можно увидеть в столбце proparallel таблицы pg_proc (r=restricted, s=safe, u=unsafe):

```
=> SELECT proparallel, count(*)
FROM pg_proc
GROUP BY proparallel;
```

proparallel	count
r	185
s	3027
u	91

(3 rows)

Все основные стандартные функции безопасны.

Пометки также показывает метакоманда \df+ утилиты psql (поле Parallel):

```
=> \x
```

Expanded display is on.

```
=> \df+ random
```

```
List of functions
-[ RECORD 1 ]-----+-----
Schema          | pg_catalog
Name            | random
Result data type | double precision
Argument data types |
Type           | func
Volatility      | volatile
Parallel        | restricted
Owner           | postgres
Security        | invoker
Access privileges |
Language        | internal
Internal name    | drandom
Description      | random value
```

```
=> \x
```

Expanded display is off.

Проверим, как пометка параллельности влияет на план выполнения запроса.

Напишем функцию, вычисляющую стоимость билета. Она помечена как безопасная для параллельного выполнения:

```
=> CREATE FUNCTION ticket_amount(ticket_no char(13)) RETURNS numeric
LANGUAGE plpgsql STABLE PARALLEL SAFE
AS $$
BEGIN
    RETURN (SELECT sum(amount)
            FROM ticket_flights tf
            WHERE tf.ticket_no = ticket_amount.ticket_no
            );
END;
$$;
```

```
CREATE FUNCTION
```

Запрос проверяет, что общая стоимость бронирований совпадает с общей стоимостью билетов:

```
=> EXPLAIN (costs off)
SELECT (SELECT sum(ticket_amount(ticket_no)) FROM tickets) =
       (SELECT sum(total_amount) FROM bookings);
```

QUERY PLAN

```
-----
Result
  InitPlan 1 (returns $1)
    -> Finalize Aggregate
      -> Gather
        Workers Planned: 2
      -> Partial Aggregate
        -> Parallel Seq Scan on tickets
  InitPlan 2 (returns $3)
    -> Finalize Aggregate
      -> Gather
        Workers Planned: 2
      -> Partial Aggregate
        -> Parallel Seq Scan on bookings
(13 rows)
```

План запроса состоит из двух частей: в узле InitPlan 1 выполняется подзапрос с агрегацией по tickets, а подзапрос в узле InitPlan 2 выполняет агрегацию по bookings.

Сейчас оба подзапроса выполняются параллельно.

Поменяем пометку параллельности на UNSAFE:

```
=> ALTER FUNCTION ticket_amount PARALLEL UNSAFE;
```

```
ALTER FUNCTION
```

```
=> EXPLAIN (costs off)
SELECT (SELECT sum(ticket_amount(ticket_no)) FROM tickets) =
       (SELECT sum(total_amount) FROM bookings);
```

QUERY PLAN

```
-----
Result
  InitPlan 1 (returns $0)
    -> Aggregate
      -> Seq Scan on tickets
  InitPlan 2 (returns $1)
    -> Aggregate
      -> Seq Scan on bookings
(7 rows)
```

Теперь оба подзапроса выполняются последовательно — пометка запрещает параллельные планы.

А теперь пометим функцию как ограниченно распараллеливаемую (RESTRICTED):

```
=> ALTER FUNCTION ticket_amount PARALLEL RESTRICTED;
```

```
ALTER FUNCTION
```

```
=> EXPLAIN (costs off)
SELECT (SELECT sum(ticket_amount(ticket_no)) FROM tickets) =
       (SELECT sum(total_amount) FROM bookings);
```

QUERY PLAN

```
-----
Result
  InitPlan 1 (returns $0)
    -> Aggregate
      -> Seq Scan on tickets
  InitPlan 2 (returns $2)
    -> Finalize Aggregate
      -> Gather
        Workers Planned: 2
      -> Partial Aggregate
        -> Parallel Seq Scan on bookings
(10 rows)
```

Подзапрос с функцией выполняется последовательно ведущим процессом, для второго подзапроса выбран параллельный план.

Конфигурационные параметры

В ряде случаев может оказаться удобным оформить запросы в виде хранимых подпрограмм (например, с целью предоставить к ним доступ приложению). В этом случае дополнительным преимуществом может быть возможность установки параметров для конкретных подпрограмм.

Рассмотрим в качестве примера запрос:

```
=> EXPLAIN (analyze, costs off, timing off)
SELECT count(*) FROM bookings;
```

QUERY PLAN

```
-----
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (actual rows=1 loops=3)
              -> Parallel Seq Scan on bookings (actual rows=703703 loops=3)
Planning Time: 0.158 ms
Execution Time: 416.424 ms
(8 rows)
```

Допустим, мы хотим использовать параллельные планы, но именно этот запрос собираемся выполнять последовательно. Тогда мы можем установить параметр на уровне функции:

```
=> CREATE FUNCTION count_bookings() RETURNS bigint
AS $$
SELECT count(*) FROM bookings;
$$ LANGUAGE sql STABLE;
```

CREATE FUNCTION

```
=> ALTER FUNCTION count_bookings SET max_parallel_workers_per_gather = 0;
```

ALTER FUNCTION

О том, как проверить план запроса, выполняющегося внутри функции, мы говорили в теме «Профилирование». Воспользуемся расширением `auto_explain`:

```
=> LOAD 'auto_explain';
```

LOAD

```
=> SET auto_explain.log_min_duration = 0;
```

SET

```
=> SET auto_explain.log_nested_statements = on;
```

SET

Выполним запрос:

```
=> SELECT count_bookings();
```

```
count_bookings
-----
          2111110
(1 row)
```

Выведем последние строки журнала сообщений:

```
student$ tail -n 10 /var/log/postgresql/postgresql-16-main.log
```

```
2025-02-05 11:16:07.075 MSK [72164] postgres@demo LOG:  duration: 124.707 ms  plan:
```

```
  Query Text:
```

```
  SELECT count(*) FROM bookings;
```

```
    Aggregate  (cost=39835.88..39835.89 rows=1 width=8)
```

```
      -> Seq Scan on bookings  (cost=0.00..34558.10 rows=2111110 width=0)
```

```
2025-02-05 11:16:07.075 MSK [72164] postgres@demo CONTEXT:  SQL function "count_bookings"
statement 1
```

```
2025-02-05 11:16:07.075 MSK [72164] postgres@demo LOG:  duration: 125.410 ms  plan:
```

```
  Query Text: SELECT count_bookings();
```

```
  Result  (cost=0.00..0.26 rows=1 width=8)
```

Функция — черный ящик для планировщика,
если ее тело не подставляется в запрос

Вызов табличных функций обычно материализуется

Планировщику можно дать дополнительную информацию

- категорию изменчивости функции

- кардинальность и стоимость

- пометку параллельности

Вспомогательные функции помогают оптимизировать
вызовы встроенных функций

1. Отключите материализацию общего табличного выражения, в котором вызывается функция random. Объясните результат.
2. Напишите функцию-обертку на SQL для запроса
`SELECT * FROM generate_series (1, 10_000_000).`
Рассмотрите три варианта: сам исходный запрос и вызовы функции в предложениях FROM и SELECT. Сравните планы выполнения запросов и использование временных файлов.
Что меняется при установке категории изменчивости Stable?
3. Какую категорию изменчивости имеет функция `days_of_week` из демонстрации? Какова ее изменчивость на самом деле?

3. Определение функции:

```
CREATE FUNCTION days_of_week() RETURNS SETOF text
AS $$
BEGIN
    FOR i IN 7 .. 13 LOOP
        RETURN NEXT to_char(to_date(i::text, 'J'), 'TMDy');
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

1. Материализация изменчивых функций в CTE

Общее табличное выражение с изменчивой функцией всегда материализуется:

```
=> EXPLAIN (costs off)
WITH c AS (
  SELECT random()
)
SELECT * FROM c;

QUERY PLAN
-----
CTE Scan on c
  CTE c
    -> Result
(3 rows)
```

В этом случае указание NOT MATERIALIZED не действует:

```
=> EXPLAIN (costs off)
WITH c AS NOT MATERIALIZED (
  SELECT random()
)
SELECT * FROM c;

QUERY PLAN
-----
CTE Scan on c
  CTE c
    -> Result
(3 rows)
```

2. Функция-обертка

```
=> CREATE FUNCTION sql_rows_lab() RETURNS SETOF integer
AS $$
  SELECT * FROM generate_series(1,10_000_000);
$$ LANGUAGE sql;
```

CREATE FUNCTION

По умолчанию функция имеет категорию изменчивости Volatile.

Базовый запрос:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM generate_series(1,10_000_000);

QUERY PLAN
-----
Function Scan on generate_series (actual rows=10000000 loops=1)
  Buffers: temp read=17090 written=17090
  Planning Time: 0.021 ms
  Execution Time: 1496.112 ms
(4 rows)
```

Поскольку функция вызывается в предложении FROM, происходит материализация. Памяти work_mem не хватает, все строки сбрасываются на диск (temp written) и затем считываются (temp read).

Вызов функции в предложении SELECT:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT sql_rows_lab();

QUERY PLAN
-----
ProjectSet (actual rows=10000000 loops=1)
  Buffers: temp read=17090 written=17090
  -> Result (actual rows=1 loops=1)
  Planning Time: 0.026 ms
  Execution Time: 3513.213 ms
(5 rows)
```


В узле ProjectSet нет материализации, поэтому цифры temp written/read остались без изменений. Однако время выполнения сильно увеличилось: оно тратится на передачу десяти миллионов строк от узла к узлу по одной строке.

Вызов функции в предложении FROM:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM sql_rows_lab();

QUERY PLAN

-----
Function Scan on sql_rows_lab (actual rows=10000000 loops=1)
  Buffers: temp read=34180 written=34180
Planning Time: 0.020 ms
Execution Time: 3285.512 ms
(4 rows)
```

Здесь к узлу Function Scan внутри функции добавляется еще один в основном запросе, поэтому количество использованных временных страниц удваивается.

Сменим категорию изменчивости функции:

```
=> ALTER FUNCTION sql_rows_lab STABLE;
```

ALTER FUNCTION

При вызове функции из предложения SELECT ничего не меняется:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT sql_rows_lab();

QUERY PLAN

-----
ProjectSet (actual rows=10000000 loops=1)
  Buffers: shared hit=8, temp read=17090 written=17090
  -> Result (actual rows=1 loops=1)
Planning Time: 0.017 ms
Execution Time: 3428.832 ms
(5 rows)
```

Вызовем функцию из предложения FROM:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM sql_rows_lab();

QUERY PLAN

-----
Function Scan on generate_series (actual rows=10000000 loops=1)
  Buffers: temp read=17090 written=17090
Planning Time: 0.048 ms
Execution Time: 1510.315 ms
(4 rows)
```

Теперь тело функции подставляется в основной запрос.

3. Дни недели

Функция была создана следующей командой:

```
=> CREATE FUNCTION days_of_week() RETURNS SETOF text
AS $$
BEGIN
  FOR i IN 7 .. 13 LOOP
    RETURN NEXT to_char(to_date(i::text, 'J'), 'TMDy');
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

Категория изменчивости не указана, поэтому подразумевается Volatile.

Может показаться, что это постоянная функция (Immutable), поскольку у нее нет параметров и список дней недели не меняется.

```
=> SELECT * FROM days_of_week();
```

```
days_of_week
-----
Пн
Вт
Ср
Чт
Пт
Сб
Вс
(7 rows)
```

Однако названия дней недели зависят от настройки локализации. Текущее значение:

```
=> \dconfig lc_time
```

```
List of configuration parameters
Parameter | Value
-----+-----
lc_time   | ru_RU.UTF-8
(1 row)
```

Изменим настройку:

```
=> SET lc_time = 'en_US.UTF8';
```

```
SET
```

```
=> SELECT * FROM days_of_week();
```

```
days_of_week
-----
Mon
Tue
Wed
Thu
Fri
Sat
Sun
(7 rows)
```

Теперь функция возвращает названия дней недели на английском языке, поэтому правильным будет задать категорию Stable:

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

```
ALTER FUNCTION
```