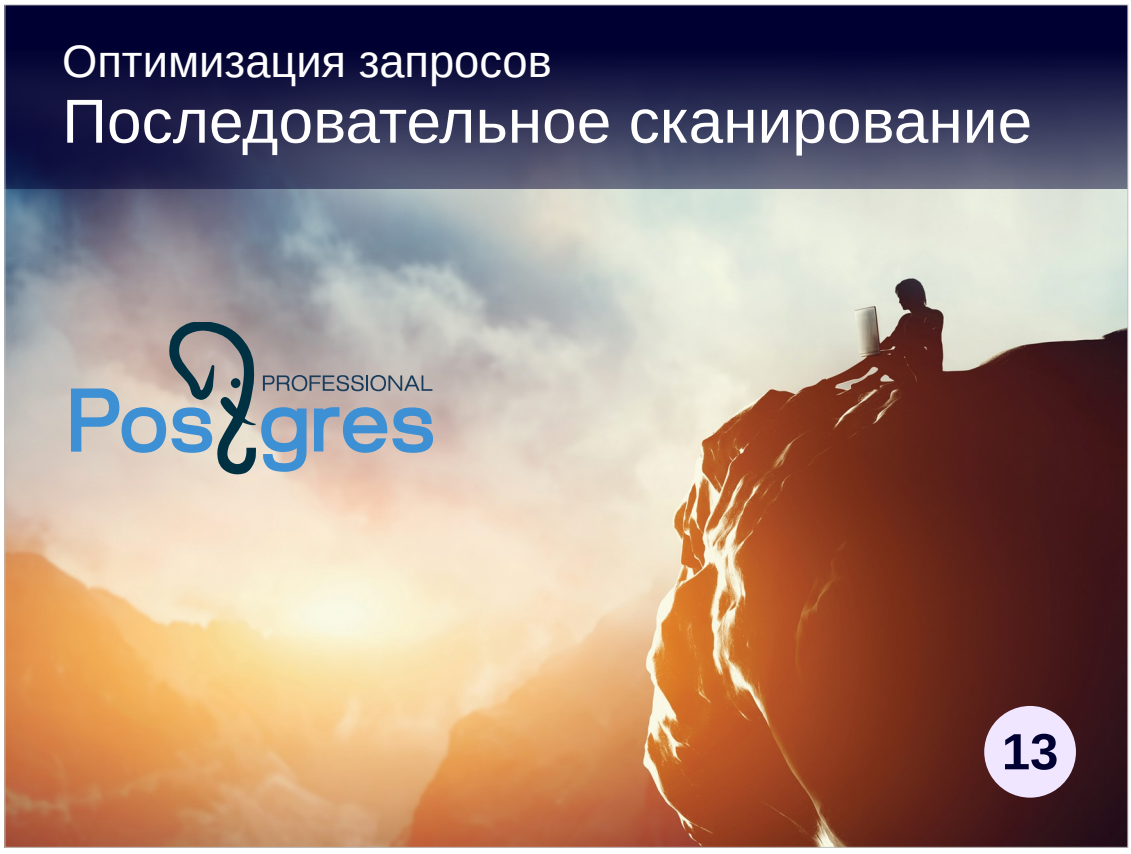


Авторы: Егор Рогов, Павел Лузанов, Павел Толмачев, Илья Баштанов

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

Темы

Последовательное сканирование (Seq Scan)

Параллельные планы выполнения

Параллельное сканирование (Parallel Seq Scan)

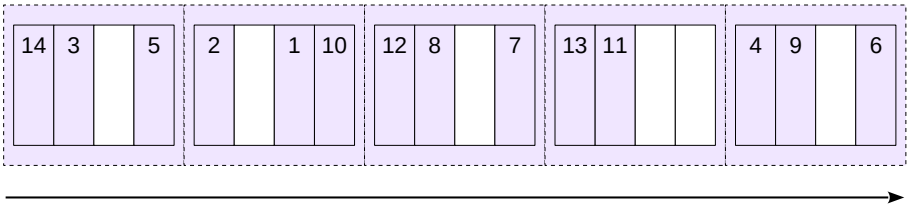
Агрегация при параллельном выполнении

Команда EXPLAIN

Seq Scan

Последовательное чтение всех страниц

- страницы читаются в кеш (используется буферное кольцо)
- проверяется видимость версий строк
- данные возвращаются в произвольном порядке
- время сканирования зависит от физического размера файла



В распоряжении оптимизатора имеется несколько способов доступа к данным. Самый простой из них — последовательное сканирование таблицы. Файл (или файлы) таблицы читается постранично от начала до конца. При этом рассматриваются все версии строк на каждой странице: удовлетворяют ли они условиям запроса и соблюдены ли правила видимости.

Напомним, что чтение происходит через буферный кеш. Чтобы большая таблица не вытеснила все полезные данные, для последовательного сканирования создается «кольцо буферов» небольшого размера. При этом другие процессы, одновременно выполняющие последовательное сканирование той же таблицы, «присоединяются» к тому же кольцу и тем самым экономят дисковые чтения (если процесс присоединился не сразу, он затем отдельно дочитывает начальные страницы таблицы).

Последовательное чтение файла позволяет использовать тот факт, что операционная система обычно читает данные порциями больше, чем размер страницы: с большой вероятностью несколько следующих страниц уже окажутся в кеше ОС.

Последовательное сканирование эффективно работает, когда надо прочитать всю таблицу или значительную ее часть (если селективность условия низка). Если же из всей таблицы нужна только небольшая часть записей, более предпочтительными являются методы доступа, использующие индекс.

Последовательное сканирование

В плане выполнения запроса последовательное сканирование представлено узлом Seq Scan:

```
=> EXPLAIN SELECT * FROM flights;

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(1 row)
```

В скобках приведены важные значения:

- cost — оценка стоимости;
- rows — оценка числа строк, возвращаемых операцией;
- width — оценка размера одной записи в байтах.

Стоимость указывается в некоторых условных единицах и состоит из двух компонент.

Первое число показывает начальную стоимость вычисления узла. Для последовательного сканирования это ноль — чтобы начать возвращать данные, никакой подготовки не требуется.

Второе число показывает полную стоимость для получения всех данных. Как оно получается?

Оптимизатор PostgreSQL учитывает дисковый ввод-вывод и ресурсы процессора. Составляющая ввода-вывода рассчитывается как произведение числа страниц в таблице на условную стоимость чтения одной страницы:

```
=> SELECT relpages, current_setting('seq_page_cost'),
  relpages * current_setting('seq_page_cost')::real AS total
FROM pg_class WHERE relname='flights';

 relpages | current_setting | total
----------+-----------------+-------
     2624 | 1               |  2624
(1 row)
```

Составляющая ресурсов процессора складывается из стоимости обработки каждой строки:

```
=> SELECT reltuples, current_setting('cpu_tuple_cost'),
  reltuples * current_setting('cpu_tuple_cost')::real AS total
FROM pg_class WHERE relname='flights';

 reltuples | current_setting |  total
-----------+-----------------+---------
    214867 | 0.01            | 2148.67
(1 row)
```

Агрегация

Рассмотрим более сложный запрос с агрегатной функцией (на небольшой таблице):

```
=> EXPLAIN SELECT count(*) FROM seats;

                          QUERY PLAN
---------------------------------------------------------------
 Aggregate  (cost=24.74..24.75 rows=1 width=8)
   ->  Seq Scan on seats  (cost=0.00..21.39 rows=1339 width=0)
(2 rows)
```

План состоит из двух узлов. Верхний — Aggregate, в котором происходит вычисление count, — получает данные от нижнего — Seq Scan.

Обратите внимание на стоимость узла Aggregate: начальная стоимость практически равна полной. Это означает, что узел не может выдать результат, пока не обработает все данные (что вполне логично).

Разница между оценкой для Aggregate и верхней оценкой для Seq Scan — стоимость работы собственно узла Aggregate. Она вычисляется исходя из оценки ресурсов на выполнение элементарной операции:

```
=> SELECT reltuples, current_setting('cpu_operator_cost'),
  reltuples * current_setting('cpu_operator_cost')::real AS total
FROM pg_class WHERE relname='seats';
```

```
 reltuples | current_setting |   total
-----------+-----------------+-----------
      1339 | 0.0025          | 3.3474998
(1 row)
```

Параллельные планы

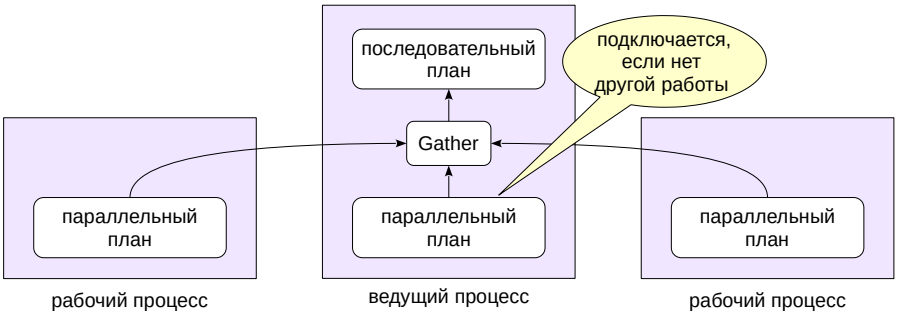
Ведущий процесс

выполняет последовательную часть плана

запускает рабочие процессы и получает от них данные

Рабочие процессы

одновременно работают над параллельной частью плана



PostgreSQL поддерживает параллельное выполнение запросов. Идея состоит в том, что ведущий процесс, выполняющий запрос, порождает (с помощью postmaster, конечно) несколько рабочих процессов, которые одновременно выполняют одну и ту же «параллельную» часть плана. Результаты этого выполнения передаются ведущему процессу, который собирает их в узле Gather.

Если рабочие процессы не успевают загрузить ведущего данными, ведущий процесс тоже подключается к выполнению того же самого параллельного плана.

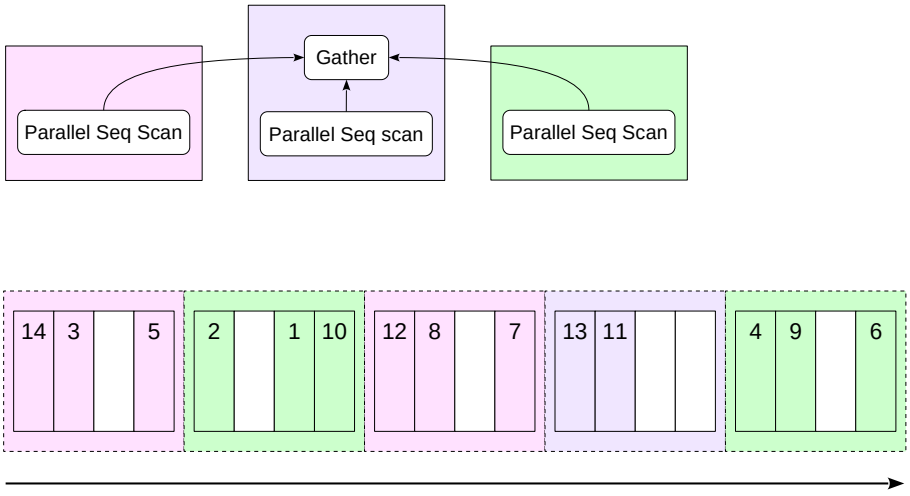
Разумеется, запуск процессов и пересылка данных требуют определенных ресурсов, поэтому далеко не каждый запрос выполняется параллельно.

Кроме того, даже при параллельном выполнении не все шаги плана запроса могут быть распараллелены. Часть операций может выполняться ведущим процессом в одиночку, последовательно.

https://postgrespro.ru/docs/postgresql/13/parallel-query

Заметим, что в PostgreSQL нет другого теоретически возможного режима распараллеливания, при котором несколько процессов составляют конвейер для обработки данных (грубо говоря, отдельные узлы плана выполняются отдельными процессами). Разработчики PostgreSQL сочли такой режим неэффективным.

Parallel Seq Scan



страницы читаются последовательно, но разными процессами

Примером параллельного выполнения является Parallel Seq Scan — «параллельное последовательное сканирование».

Название звучит противоречиво, но, тем не менее, отражает суть операции. Страницы таблицы *читаются последовательно,* в том же самом порядке, в котором они читались бы при обычном последовательном сканировании. Однако запросы на чтение выдаются несколькими *параллельно* работающими процессами. Процессы синхронизируются между собой, чтобы их запросы шли в правильном порядке.

Преимущество такого сканирования проявляется в том, что параллельные процессы одновременно обрабатывают свои страницы. Чтобы эффект оправдал дополнительные накладные расходы на пересылку данных от процесса к процессу, обработка должна быть достаточно ресурсоемкой. Хорошим примером является агрегация данных, поскольку для нее необходимы ресурсы процессора, а пересылать требуется только одно итоговое число. В таком случае параллельное выполнение запроса может занять существенно меньше времени, чем последовательное.

Параллельное последовательное сканирование

Рассмотрим пример плана с параллельным последовательным сканированием:

```
=> EXPLAIN SELECT count(*) FROM bookings;
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=25442.58..25442.59 rows=1 width=8)
   ->  Gather  (cost=25442.36..25442.57 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=24442.36..24442.37 rows=1 width=8)
               ->  Parallel Seq Scan on bookings  (cost=0.00..22243.29 rows=879629 width=0)
(5 rows)
```

Все, что находится ниже узла Gather — параллельная часть плана. Она выполняется в каждом из рабочих процессов (которых запланировано два) и, возможно, в ведущем процессе.

Узел Gather и все узлы выше выполняются только в ведущем процессе. Это последовательная часть плана.

Начнем разбираться снизу вверх. Узел Parallel Seq Scan представляет сканирование таблицы в параллельном режиме.

В поле rows показана оценка числа строк, которые обработает один рабочий процесс. Всего их запланировано 2, и еще часть работы выполнит ведущий, поэтому общее число строк делится на 2.4 (доля ведущего процесса уменьшается с ростом числа рабочих процессов).

```
=> SELECT round(reltuples / 2.4) "rows"
FROM pg_class WHERE relname = 'bookings';
```

```
  rows
--------
 879629
(1 row)
```

По умолчанию ведущий процесс участвует в выполнении параллельной части плана:

```
=> SHOW parallel_leader_participation;
```

```
 parallel_leader_participation
-------------------------------
 on
(1 row)
```

Если ведущий процесс становится узким местом, его можно разгрузить:

```
=> SET parallel_leader_participation = off;
```

```
SET
```

```
=> EXPLAIN SELECT count(*) FROM bookings;
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=27641.65..27641.66 rows=1 width=8)
   ->  Gather  (cost=27641.44..27641.65 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=26641.44..26641.45 rows=1 width=8)
               ->  Parallel Seq Scan on bookings  (cost=0.00..24002.55 rows=1055555 width=0)
(5 rows)
```

В этом случае общее число строк делится на число запланированных рабочих процессов (2):

```
=> SELECT round(reltuples / 2) "rows"
FROM pg_class WHERE relname = 'bookings';
```

```
  rows
---------
 1055555
(1 row)
```

Вернем значение по умолчанию для параметра parallel_leader_participation:

```
=> RESET parallel_leader_participation;
```

```
RESET
```

Для оценки узла Parallel Seq Scan компонента ввода-вывода берется полностью (таблицу все равно придется прочитать страница за страницей), а ресурсы процессора делятся между процессами (на 2.4 в данном случае).

```
=> SELECT round(
  (
          relpages * current_setting('seq_page_cost')::real +
          reltuples * current_setting('cpu_tuple_cost')::real / 2.4
  )::numeric,
  2
) "cost" FROM pg_class WHERE relname = 'bookings';
```

```
   cost
----------
 22243.29
(1 row)
```

```
=> EXPLAIN SELECT count(*) FROM bookings;
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=25442.58..25442.59 rows=1 width=8)
   ->  Gather  (cost=25442.36..25442.57 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=24442.36..24442.37 rows=1 width=8)
               ->  Parallel Seq Scan on bookings  (cost=0.00..22243.29 rows=879629 width=0)
(5 rows)
```

Следующий узел — Partial Aggregate — выполняет агрегацию данных, полученных рабочим процессом, то есть в данном случае подсчитывает количество строк. Оценка выполняется уже известным образом (и добавляется к оценке сканирования таблицы):

```
=> SELECT round((reltuples / 2.4 * current_setting('cpu_operator_cost')::real)::numeric, 2) "cost"
FROM pg_class WHERE relname='bookings';
```

```
  cost
---------
 2199.07
(1 row)
```

Следующий узел — Gather — выполняется ведущим процессом. Он отвечает за запуск рабочих процессов и получение от них данных.

Запуск процессов и пересылка каждой строки данных оцениваются следующими значениями:

```
=> SELECT current_setting('parallel_setup_cost') parallel_setup_cost,
current_setting('parallel_tuple_cost') parallel_tuple_cost;
```

```
 parallel_setup_cost | parallel_tuple_cost
---------------------+---------------------
 1000                | 0.1
(1 row)
```

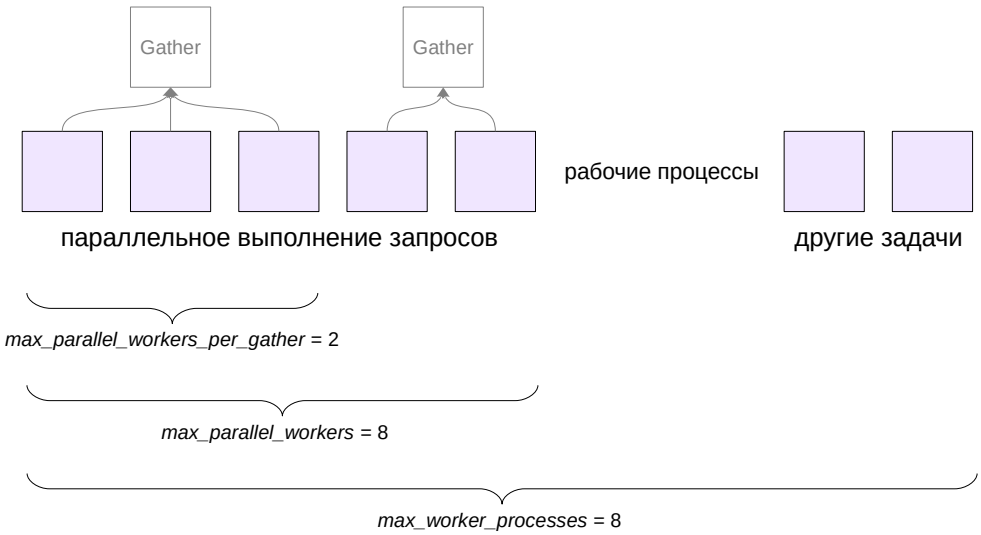
В данном случае пересылается всего одна строка и основная стоимость приходится на запуск.

```
=> EXPLAIN SELECT count(*) FROM bookings;
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=25442.58..25442.59 rows=1 width=8)
   ->  Gather  (cost=25442.36..25442.57 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=24442.36..24442.37 rows=1 width=8)
               ->  Parallel Seq Scan on bookings  (cost=0.00..22243.29 rows=879629 width=0)
(5 rows)
```

Последний узел — Finalize Aggregate — агрегирует полученные частичные агрегаты. Поскольку для этого надо сложить всего три числа, оценка минимальна.

Общие настройки



8

Параллельным выполнением управляет довольно много параметров.

Сначала рассмотрим те, что ограничивают число рабочих процессов.

Вообще механизм рабочих процессов используется не только для параллельного выполнения запросов. Например, рабочие процессы задействованы в логической репликации, ими могут пользоваться расширения. Общее число одновременно выполняющихся рабочих процессов ограничено параметром *max_worker_processes* (по умолчанию 8).

Число одновременно выполняющихся рабочих процессов, занимающихся параллельными планами, ограничено параметром *max_parallel_workers* (по умолчанию тоже 8).

Число одновременно выполняющихся рабочих процессов, обслуживающих один ведущий процесс, ограничено параметром *max_parallel_workers_per_gather* (по умолчанию 2).

Значения этих параметров следует изменить в зависимости от возможностей аппаратуры, объема данных и загруженности системы. Например, даже если в базе данных есть большие таблицы и запросы могли бы выиграть от распараллеливания, но при этом в системе нет свободных ядер, то параллельное выполнение не будет иметь смысла.

Число рабочих процессов

Равно нулю (параллельный план не строится)

если *размер таблицы* < *min_parallel_table_scan_size* = 8MB

Фиксировано

если для таблицы указан параметр хранения *parallel_workers*

Вычисляется по формуле

$1 + \lfloor \log_3 (\textit{размер таблицы} / \textit{min\_parallel\_table\_scan\_size}) \rfloor$

Но не больше, чем *max_parallel_workers_per_gather*

Сколько рабочих процессов будет использоваться?

Планировщик вообще не будет рассматривать параллельное сканирование, если физический размер таблицы меньше значения параметра *min_parallel_table_scan_size*.

Обычно число процессов вычисляется по формуле, приведенной на слайде. Она означает, что при увеличении таблицы в три раза будет добавляться очередной процесс. Например, для стандартного значения *min_parallel_table_scan_size* = 8MB:

таблица	процессы	таблица	процессы
8MB	1	216MB	4
24MB	2	648MB	5
72MB	3	1.9GB	6

Число процессов можно и явно указать в параметре хранения *parallel_workers* таблицы.

При этом число процессов в любом случае не будет превышать значения параметра *max_parallel_workers_per_gather*. Если при выполнении запроса доступное число процессов окажется меньше запланированного, будут использоваться только доступные (вплоть до последовательного выполнения, если пул полностью исчерпан).

Число рабочих процессов

Планировщик не рассматривает параллельные планы для таблиц размером меньше, чем:

```
=> SHOW min_parallel_table_scan_size;
```

```
 min_parallel_table_scan_size
------------------------------
 8MB
(1 row)
```

Если запросить информацию из таблицы немного большего размера (flights, 19 Мбайт), будет запланирован один дополнительный процесс:

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

```
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Finalize Aggregate (actual time=83.442..86.916 rows=1 loops=1)
   ->  Gather (actual time=83.436..86.910 rows=2 loops=1)
         Workers Planned: 1
         Workers Launched: 1
         ->  Partial Aggregate (actual time=79.203..79.204 rows=1 loops=2)
               ->  Parallel Seq Scan on flights (actual time=1.463..73.050 rows=107434 loops=2)
 Planning Time: 0.081 ms
 Execution Time: 86.957 ms
(8 rows)
```

При запросе данных из большой таблицы (bookings, 105 Мбайт) расчетное количество — три процесса.

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

```
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Finalize Aggregate (actual time=343.684..347.006 rows=1 loops=1)
   ->  Gather (actual time=343.551..346.996 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate (actual time=337.666..337.667 rows=1 loops=3)
               ->  Parallel Seq Scan on bookings (actual time=3.899..297.876 rows=703703 loops=3)
 Planning Time: 0.052 ms
 Execution Time: 347.035 ms
(8 rows)
```

Однако запланировано два процесса, так как параметр max_parallel_workers_per_gather ограничивает число процессов параллельного участка плана и сейчас действует значение по умолчанию (2).

```
=> SHOW max_parallel_workers_per_gather;
```

```
 max_parallel_workers_per_gather
---------------------------------
 2
(1 row)
```

Ослабив ограничение, получим план с тремя процессами:

```
=> SET max_parallel_workers_per_gather = 5;
```

```
SET
```

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

```
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Finalize Aggregate (actual time=265.059..265.116 rows=1 loops=1)
   ->  Gather (actual time=265.050..265.110 rows=4 loops=1)
         Workers Planned: 3
         Workers Launched: 3
         ->  Partial Aggregate (actual time=238.581..238.582 rows=1 loops=4)
               ->  Parallel Seq Scan on bookings (actual time=0.026..90.309 rows=527778 loops=4)
 Planning Time: 0.061 ms
 Execution Time: 265.141 ms
(8 rows)
```

Для конкретной таблицы параметром хранения parallel_workers можно задать рекомендованное количество процессов:

```
=> ALTER TABLE bookings SET (parallel_workers = 4);
```

```
ALTER TABLE
```

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

```
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Finalize Aggregate (actual time=249.775..251.730 rows=1 loops=1)
   ->  Gather (actual time=239.730..251.718 rows=5 loops=1)
         Workers Planned: 4
         Workers Launched: 4
         ->  Partial Aggregate (actual time=212.798..212.799 rows=1 loops=5)
               ->  Parallel Seq Scan on bookings (actual time=0.025..106.839 rows=422222 loops=5)
 Planning Time: 0.088 ms
 Execution Time: 251.755 ms
(8 rows)
```

Если установить параметр хранения в ноль, планировщик всегда будет выбирать последовательное сканирование данной таблицы.

Но в любом случае количество запланированных процессов не будет превышать max_parallel_workers_per_gather, независимо от параметра хранения.

Восстановим начальные значения параметров:

```
=> ALTER TABLE bookings RESET (parallel_workers);
```

```
ALTER TABLE
```

```
=> RESET max_parallel_workers_per_gather;
```

```
RESET
```

Не распараллеливаются

Запросы на запись

- а также запросы с блокировкой строк

Курсоры

- в том числе запросы в цикле FOR в PL/pgSQL

Запросы с функциями PARALLEL UNSAFE

Запросы в функциях,
вызванных из распараллеленного запроса

Не каждый запрос может выполняться в параллельном режиме.

Не распараллеливаются запросы, изменяющие или блокирующие данные (UPDATE, DELETE, SELECT FOR UPDATE и т. п.).

Не распараллеливаются запросы, выполнение которых может быть приостановлено — это относится к запросам в курсорах, в том числе в циклах FOR PL/pgSQL.

Не распараллеливаются запросы, содержащие функции, помеченные как PARALLEL UNSAFE (все основные стандартные функции безопасны; список небезопасных можно получить из таблицы pg_proc по условию proparallel = 'u').

Не распараллеливаются запросы, содержащиеся в функциях, которые вызываются из распараллеленного запроса (чтобы не допустить рекурсивного разрастания).

Часть из этих ограничений может быть снята в следующих версиях PostgreSQL.

https://postgrespro.ru/docs/postgresql/13/when-can-parallel-query-be-used

Только последовательно

Чтение результатов общих табличных выражений (CTE)

Чтение результатов нераскрываемых подзапросов

Обращения к временным таблицам

Вызовы функций PARALLEL RESTRICTED

Функции, использующие вложенные транзакции

В целом, чем большую часть плана удается выполнить параллельно, тем больший возможен эффект. Однако есть ряд операций, которые в целом не препятствуют распараллеливанию, но сами могут выполняться только последовательно в ведущем процессе.

К ним относятся:

- чтение результатов общих табличных выражений (подзапросов в предложении WITH);

- чтение результатов других нераскрываемых подзапросов (которые представляются в плане узлами, например, SubPlan);

- обращения ко временным таблицам (так как они доступны только ведущему процессу);

- вызовы функций, помеченных как PARALLEL RESTRICTED (список можно получить из таблицы pg_proc по условию proparallel = 'r').

Если в запросе вызывается функция, использующая вложенные транзакции (например, функции на PL/pgSQL с обработкой исключений), запрос завершится с ошибкой. Такие функции должны быть помечены как PARALLEL RESTRICTED.

https://postgrespro.ru/docs/postgresql/13/parallel-safety

Пометки параллельности для функций и агрегатов

Для функций существует три типа пометок параллельности:

- SAFE — не препятствует параллельному выполнению запроса;
- UNSAFE — запрещает параллельное выполнение запроса (функция изменяет состояние базы данных, транзакции или конфигурационных параметров);
- RESTRICTED — запрос может выполняться параллельно, но функция может выполняться только в ведущем процессе (функция обращается к состоянию сеанса: к временным таблицам, курсорам, подготовленным операторам и т. п.).

Пользовательские функции по умолчанию получают пометку UNSAFE.

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.

Итоги

Последовательное сканирование читает всю таблицу

- эффективно при низкой селективности

Параллельное выполнение в ряде случаев позволяет ускорить запрос

Практика

1. Убедитесь, что запрос, вычисляющий среднюю стоимость перелета, выполняется параллельно.
2. Проверьте, как выполняется тот же запрос, если чтение из таблицы перелетов оформлено как общее табличное выражение.
3. В демонстрации запрос всех строк из таблицы рейсов выполнялся последовательно. Почему? Может ли такой запрос в принципе быть распараллелен или он запрещает параллельное выполнение?
   Воспользуйтесь параметром *force_parallel_mode*, чтобы ответить на этот вопрос.

2. Используйте указание MATERIALIZE, чтобы планировщик не раскрывал общее табличное выражение.

3. Параметр *force_parallel_mode* можно использовать только для проверки, допускает ли запрос распараллеливание, но не для реальной работы. С этим параметром PostgreSQL распараллеливает любой запрос, если только это возможно, даже если в этом нет никакого смысла. Количество параллельных процессов всегда выбирается равным единице, так что ускорения при таком режиме в любом случае не будет.

1. Средняя стоимость перелета

```
=> EXPLAIN SELECT avg(amount) FROM ticket_flights;
```

```
                                              QUERY PLAN
-----------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=114640.79..114640.80 rows=1 width=32)
   ->  Gather  (cost=114640.57..114640.77 rows=2 width=32)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=113640.57..113640.57 rows=1 width=32)
               ->  Parallel Seq Scan on ticket_flights  (cost=0.00..104899.05 rows=3496605 width=6)
(5 rows)
```

Планировщик выбирает параллельный план.

2. Общее табличное выражение

```
=> EXPLAIN
WITH tf AS MATERIALIZED (
  SELECT * FROM ticket_flights
)
SELECT avg(amount) FROM tf;
```

```
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=342668.19..342668.20 rows=1 width=32)
   CTE tf
     ->  Seq Scan on ticket_flights  (cost=0.00..153851.52 rows=8391852 width=32)
   ->  CTE Scan on tf  (cost=0.00..167837.04 rows=8391852 width=16)
(4 rows)
```

Здесь узел CTE представляет подзапрос в общем табличном выражении.

Затем выполняется узел CTE Scan: по сути он работает так же, как и Seq Scan, но просматривает не таблицу, а полученный и материализованный результат выполнения подзапроса. Эта операция выполняется последовательно.

При этом сам подзапрос в общем табличном выражении может работать параллельно:

```
=> EXPLAIN
WITH x AS MATERIALIZED (
  SELECT avg(amount) FROM ticket_flights
)
SELECT * FROM x;
```

```
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 CTE Scan on x  (cost=114640.80..114640.82 rows=1 width=32)
   CTE x
     ->  Finalize Aggregate  (cost=114640.79..114640.80 rows=1 width=32)
           ->  Gather  (cost=114640.57..114640.77 rows=2 width=32)
                 Workers Planned: 2
                 ->  Partial Aggregate  (cost=113640.57..113640.57 rows=1 width=32)
                       ->  Parallel Seq Scan on ticket_flights  (cost=0.00..104899.05 rows=3496605 width=6)
(7 rows)
```

3. Последовательный план

```
=> EXPLAIN SELECT * FROM flights;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(1 row)
```

Размер таблицы достаточно большой, чтобы планировщик рассмотрел параллельный план:

```
=> SELECT pg_size_pretty(pg_table_size('flights')) size;
```

```
 size
-------
 21 MB
(1 row)
```

Используя конфигурационный параметр можно убедиться, что запрос в принципе не запрещает параллельное выполнение, но планировщик отказывается от такой возможности из-за ее неэффективности:

```
=> SET force_parallel_mode = on;

SET

=> EXPLAIN SELECT * FROM flights;

                              QUERY PLAN
----------------------------------------------------------------------
 Gather  (cost=1000.00..27259.37 rows=214867 width=63)
   Workers Planned: 1
   Single Copy: true
   ->  Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(4 rows)
```

В данном случае дело в том, что пересылка всех строк таблицы между процессами невыгодна.