

Доступ к данным Параллельный доступ

Postgres
PROFESSIONAL



16

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.

Параллельные планы

Размер пула процессов

Параллельное последовательное сканирование

Параллельный индексный доступ

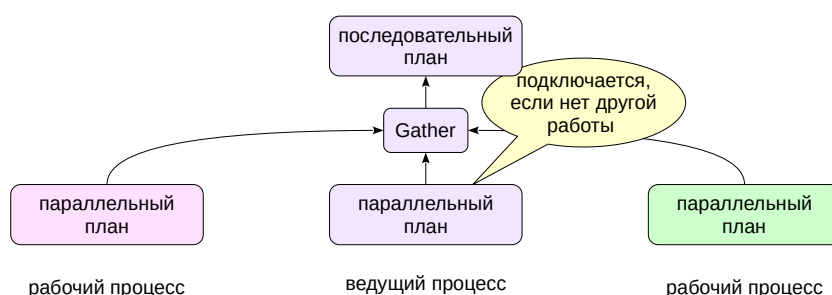
Ведущий процесс

выполняет последовательную часть плана

запускает рабочие процессы и получает от них данные

Рабочие процессы

одновременно работают над параллельной частью плана



3

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.

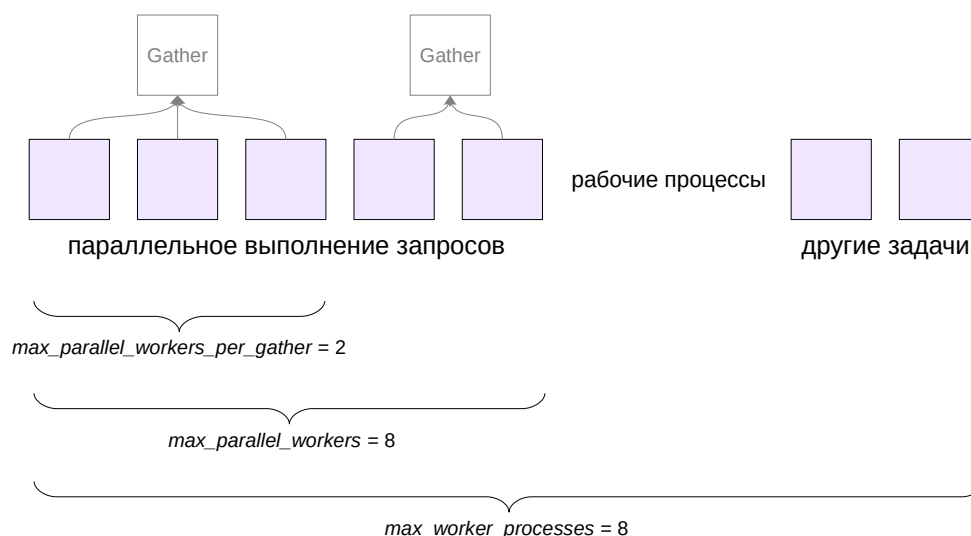
Разумеется, запуск процессов и пересылка данных требуют определенных ресурсов, поэтому далеко не каждый запрос выполняется параллельно.

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/16/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.

Размер пула процессов



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. Рабочие процессы можно использовать в прикладном коде (подробнее об этом — в теме «Фоновые процессы» курса DEV2). 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 загрузки 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.

Параллельное последовательное сканирование

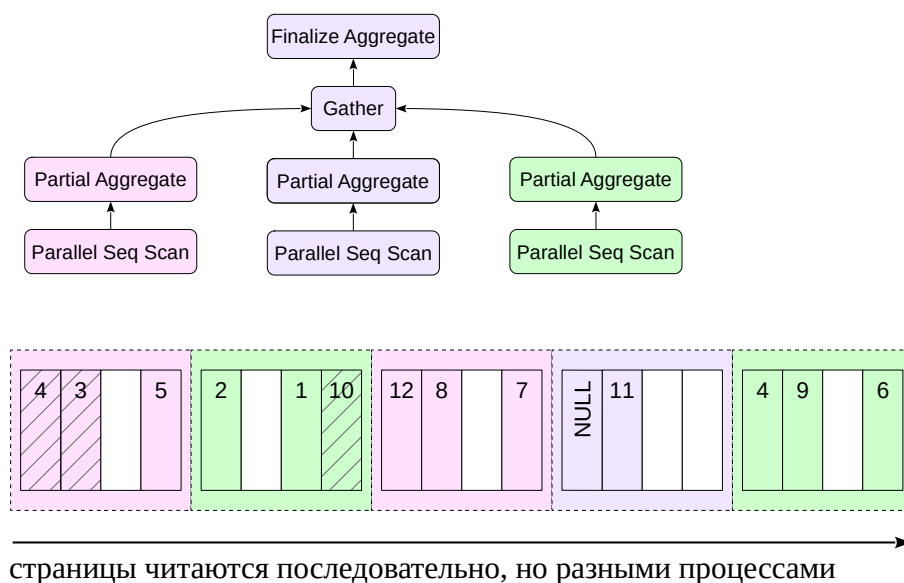
- агрегация

- параллельная агрегация

Число рабочих процессов

Ограничения

Parallel Seq Scan



Примером узла, выполняющегося в параллельном режиме 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.

Агрегация

Рассмотрим запрос с агрегатной функцией на небольшой таблице. Такой запрос выполняется последовательно:

```
=> 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 узел Aggregate, в котором происходит вычисление count, получает данные от дочернего узла Seq Scan.

Обратите внимание на узел Aggregate: его начальная стоимость практически равна полной. Это означает, что узел не может выдать результат (состоящий из одной строки), 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 рассчитывается исходя из того, что надо будет посчитать каждую строку, выданную узлом Seq Scan; одна элементарная операция оценивается значением параметра `cpu_operator_cost`:

```
=> 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)
```

Параллельное последовательное сканирование

Теперь рассмотрим такой же пример, но с большой таблицей. Здесь в плане запроса увидим параллельное последовательное сканирование:

```
=> 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
) AS "cost"
FROM pg_class WHERE relname = 'bookings';

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

Выведем план еще раз:

```
=> 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 * current_setting('cpu_operator_cost')::real / 2.4
  )::numeric,
  2
) AS "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.

И снова посмотрим на план запроса:

```
=> 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.

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

$1 + \lfloor \log_3(\text{размер таблицы} / \text{min_parallel_table_scan_size}) \rfloor$

Всегда не больше *max_parallel_workers_per_gather*

8

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 при последовательном сканировании

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=96.990..97.034 rows=1 loops=1)
  -> Gather (actual time=60.525..97.018 rows=2 loops=1)
      Workers Planned: 1
      Workers Launched: 1
      -> Partial Aggregate (actual time=27.737..27.737 rows=1 loops=2)
          -> Parallel Seq Scan on flights (actual time=0.011..23.625 rows=107434
loops=2)
Planning Time: 0.175 ms
Execution Time: 97.060 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=948.229..950.288 rows=1 loops=1)
  -> Gather (actual time=945.809..950.278 rows=3 loops=1)
      Workers Planned: 2
      Workers Launched: 2
      -> Partial Aggregate (actual time=943.131..943.132 rows=1 loops=3)
          -> Parallel Seq Scan on bookings (actual time=9.823..897.242 rows=703703
loops=3)
Planning Time: 0.066 ms
Execution Time: 950.316 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=243.301..243.359 rows=1 loops=1)  
  -> Gather (actual time=243.293..243.354 rows=4 loops=1)  
        Workers Planned: 3  
        Workers Launched: 3  
        -> Partial Aggregate (actual time=235.895..235.896 rows=1 loops=4)  
              -> Parallel Seq Scan on bookings (actual time=4.260..159.100 rows=527778  
loops=4)  
Planning Time: 0.069 ms  
Execution Time: 243.381 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=250.018..252.028 rows=1 loops=1)  
  -> Gather (actual time=247.955..252.021 rows=5 loops=1)  
        Workers Planned: 4  
        Workers Launched: 4  
        -> Partial Aggregate (actual time=239.298..239.298 rows=1 loops=5)  
              -> Parallel Seq Scan on bookings (actual time=0.031..172.462 rows=422222  
loops=5)  
Planning Time: 0.083 ms  
Execution Time: 252.048 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

10

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 (пометки параллельности рассматриваются в теме «Функции»).

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/16/when-can-parallel-query-be-used>

Чтение результатов общих табличных выражений (СТЕ)

Чтение результатов нераскрываемых подзапросов

Обращения к временным таблицам

Вызовы функций PARALLEL RESTRICTED

Функции, использующие вложенные транзакции

В целом, чем большую часть плана удастся выполнить параллельно, тем больший возможен эффект. Однако есть ряд операций, которые не препятствуют распараллеливанию плана, 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.

Если в запросе вызывается функция, использующая вложенные транзакции (например, функция на PL/pgSQL с обработкой исключений), запрос завершится с ошибкой. Такие функции должны быть помечены как PARALLEL RESTRICTED. Подробнее пометки параллельности рассматриваются в теме «Функции».

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

Только последовательные операции

Некоторые операции всегда выполняются последовательно, например, обращения к временным таблицам.

Для примера создадим временную таблицу tmp_bookings — копию таблицы bookings:

```
=> CREATE TEMP TABLE tmp_bookings AS
      SELECT * FROM bookings;
```

```
SELECT 2111110
```

Используя операцию объединения множеств UNION ALL (без удаления дубликатов), объединим результаты двух запросов — к основной и временной таблицам:

```
=> EXPLAIN (costs off)
      SELECT count(*) FROM bookings
      UNION ALL
      SELECT count(*) FROM tmp_bookings;
```

QUERY PLAN

```
-----
Append
  -> Finalize Aggregate
        -> Gather
              Workers Planned: 2
              -> Partial Aggregate
                    -> Parallel Seq Scan on bookings
  -> Aggregate
        -> Seq Scan on tmp_bookings
(8 rows)
```

Данные из таблицы bookings читаются параллельно (по умолчанию используется два процесса), а из временной таблицы tmp_bookings — последовательно.

Если операция не выполняется параллельно, это не обязательно означает, что она в принципе не распараллеливается: возможно, планировщик счел, что в данном случае последовательное выполнение эффективнее. Параметр debug_parallel_query позволяет проверить, может ли операция выполняться в параллельном режиме.

Без агрегации обращение к таблице bookings выполняется последовательно:

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

QUERY PLAN

```
-----
Seq Scan on bookings
(1 row)
```

Но со включенным параметром мы видим, что в принципе параллельный план возможен:

```
=> SET debug_parallel_query = on;
```

```
SET
```

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

QUERY PLAN

```
-----
Gather
  Workers Planned: 1
  Single Copy: true
  -> Seq Scan on bookings
(4 rows)
```

Параметр служит только для отладки. С ним параллельное выполнение планируется, даже если оно не выгодно, и всегда использует единственный процесс.

```
=> RESET debug_parallel_query;
```

```
RESET
```

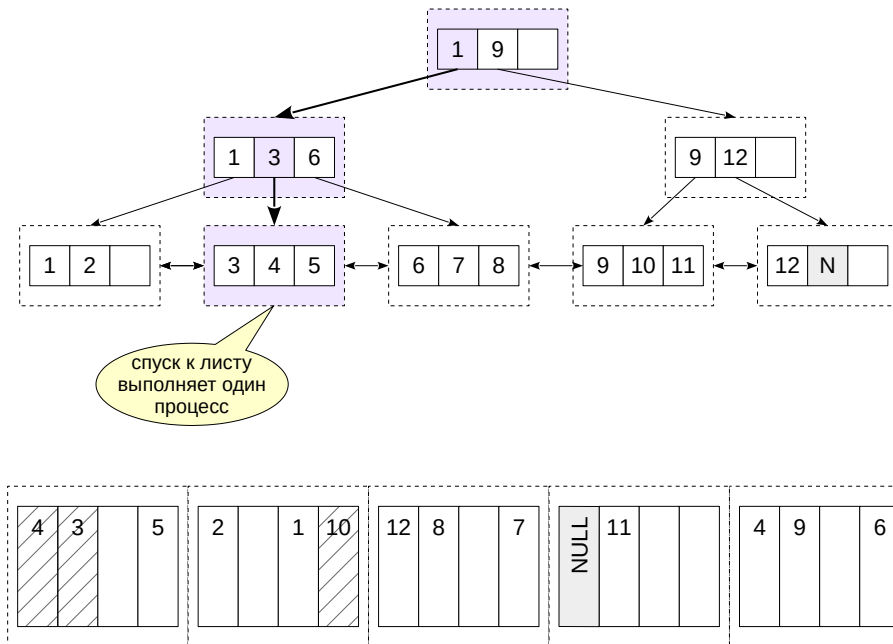

Параллельное сканирование индекса

Параллельное сканирование только индекса

Параллельное сканирование по битовой карте

Число рабочих процессов

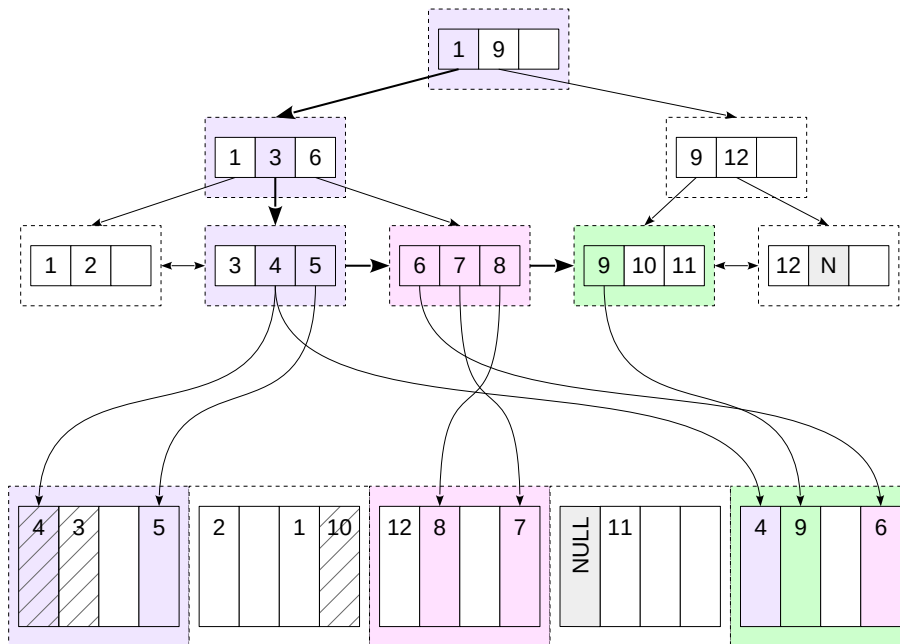
Parallel Index Scan



14

Индексный доступ тоже может выполняться в параллельном режиме. Это происходит в два этапа. Сначала ведущий процесс спускается от корня дерева к листовой странице.

Parallel Index Scan



15

Затем рабочие процессы выполняют параллельное чтение листовых страниц индекса, двигаясь по списку.

Процесс, прочитавший индексную страницу, выполняет и чтение необходимых табличных страниц. При этом может получиться так, что одну и ту же табличную страницу прочитают несколько процессов (этот пример показан на слайде: последняя табличная страница содержит строки, ссылки на которые ведут от нескольких индексных страниц, прочитанных разными процессами). Конечно, сама страница будет находиться в буферном кеше в одном экземпляре.

Параллельное сканирование индекса

В качестве примера параллельного сканирования индекс найдем общую стоимость всех бронирований с кодами, меньшими 400000 (они составляют примерно одну четверть от общего числа):

```
=> EXPLAIN
SELECT sum(total_amount)
FROM bookings WHERE book_ref < '400000';
```

QUERY PLAN

```
-----
Finalize Aggregate  (cost=16863.96..16863.97 rows=1 width=32)
  -> Gather  (cost=16863.74..16863.94 rows=2 width=32)
        Workers Planned: 2
        -> Partial Aggregate  (cost=15863.74..15863.75 rows=1 width=32)
              -> Parallel Index Scan using bookings_pkey on bookings
(cost=0.43..15314.03 rows=219879 width=6)
      Index Cond: (book_ref < '400000'::bpchar)
(6 rows)
```

Аналогичный план мы уже видели при параллельном последовательном сканировании, но в данном случае данные читаются с помощью индекса узлом Parallel Index Scan.

Полная стоимость складывается из стоимостей доступа к индексу и к таблице. Оценка стоимости индексного доступа не делится между процессами, так как индекс читается процессами последовательно, страница за страницей:

```
=> SELECT round(
  (relpages / 4.0) * current_setting('random_page_cost')::real +
  (reltuples / 4.0) * current_setting('cpu_index_tuple_cost')::real +
  (reltuples / 4.0) * current_setting('cpu_operator_cost')::real
) AS index_cost
FROM pg_class WHERE relname = 'bookings_pkey';

index_cost
-----
      9750
(1 row)
```

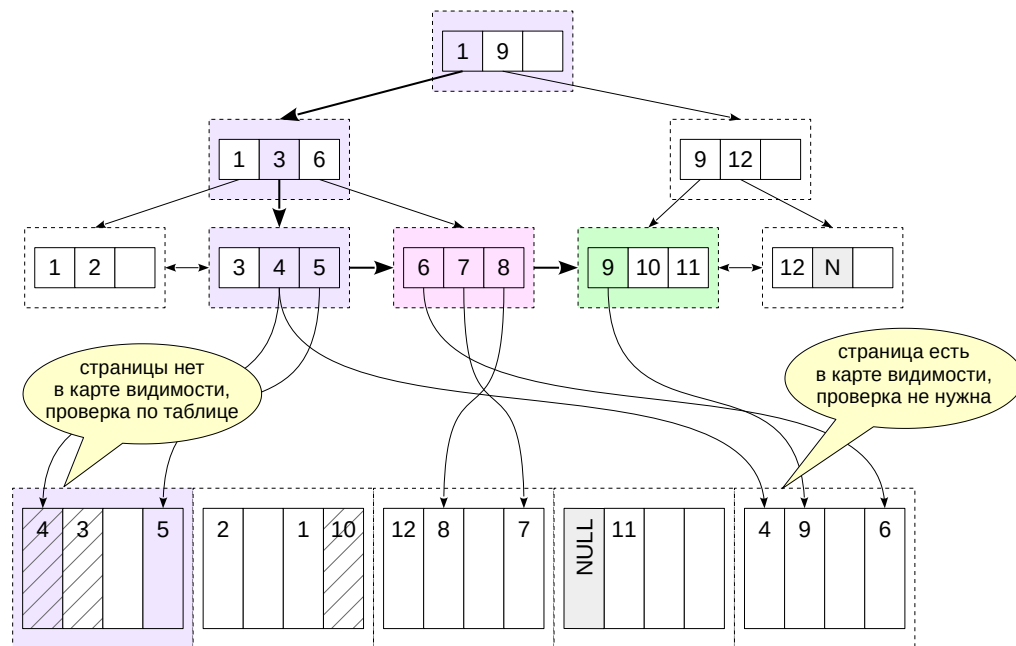
Значением параметра `cpu_operator_cost` оценивается операция сравнения значений («меньше»).

Стоимость доступа к 1/4 табличных страниц (поделенных между процессами) оценивается аналогично последовательному сканированию, поскольку строки физически упорядочены по `book_ref`:

```
=> SELECT round(
  (relpages / 4.0) * current_setting('seq_page_cost')::real +
  (reltuples / 4.0) / 2.4 * current_setting('cpu_tuple_cost')::real
) AS table_cost
FROM pg_class WHERE relname = 'bookings';

table_cost
-----
      5561
(1 row)
```


Parallel Index Only Scan



17

Сканирование только индекса тоже может выполняться параллельно. Это происходит точно так же, как и при обычном индексном сканировании: ведущий процесс спускается от корня к листовой странице, а затем рабочие процессы параллельно сканируют листовые страницы индекса, обращаясь при необходимости к соответствующим страницам таблицы для проверки видимости.

Параллельное сканирование только индекса

Рассмотрим похожий пример, в котором сканирование только индекса выполняется параллельно:

```
=> EXPLAIN
```

```
SELECT count(book_ref)
FROM bookings WHERE book_ref <= '400000';
```

QUERY PLAN

```
-----
Finalize Aggregate  (cost=13498.96..13498.97 rows=1 width=8)
  -> Gather  (cost=13498.74..13498.95 rows=2 width=8)
        Workers Planned: 2
        -> Partial Aggregate  (cost=12498.74..12498.75 rows=1 width=8)
              -> Parallel Index Only Scan using bookings_pkey on bookings
(cost=0.43..11949.05 rows=219879 width=7)
      Index Cond: (book_ref <= '400000'::bpchar)
(6 rows)
```

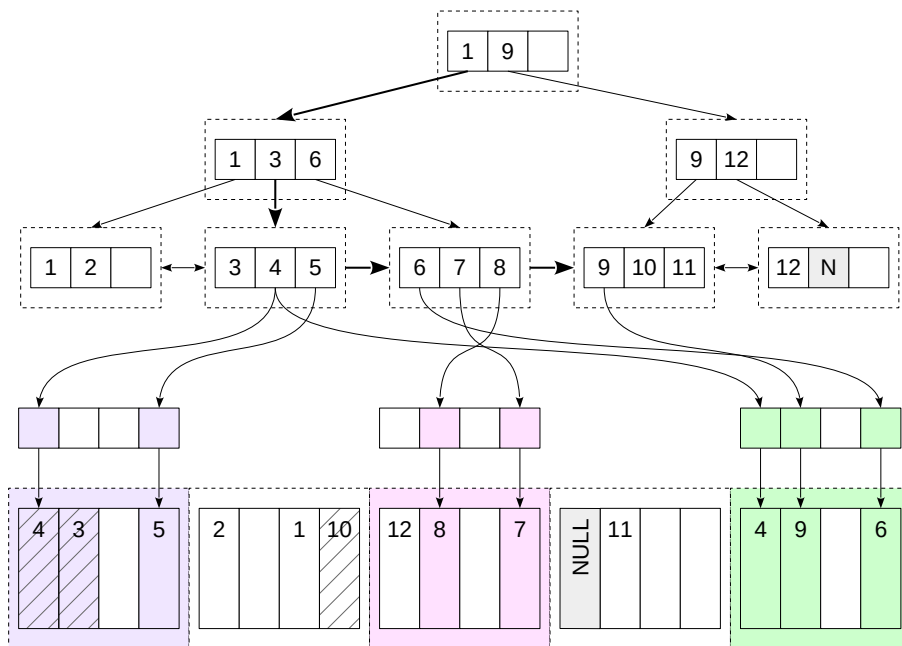
Стоимость доступа к таблице здесь учитывает только обработку строк, без ввода-вывода:

```
=> SELECT round(
  (reltuples / 4.0) / 2.4 * current_setting('cpu_tuple_cost')::real
) AS table_cost
FROM pg_class WHERE relname = 'bookings';

table_cost
-----
      2199
(1 row)
```

Вклад индексного доступа остается прежним.

Parallel Bitmap Heap Scan



19

Сканирование по битовой карте может выполняться параллельно.

Первый этап — сканирование индекса и построение битовой карты — всегда выполняется последовательно ведущим процессом.

А второй этап — сканирование таблицы — выполняется рабочими процессами параллельно. Это происходит аналогично параллельному последовательному сканированию.

Параллельное сканирование по битовой карте

Для построения битовой карты нам понадобится новый индекс для таблицы bookings:

```
=> CREATE INDEX ON bookings(total_amount);
```

CREATE INDEX

Сколько бронирований сделано за последний месяц на сумму до 20 тысяч Р?

```
=> SELECT bookings.now() - INTERVAL '1 months';
```

?column?

```
-----  
2017-07-15 18:00:00+03  
(1 row)
```

```
=> \bind '2017-07-15 18:00:00+03'
```

```
=> EXPLAIN (costs off)  
SELECT count(*) FROM bookings  
WHERE total_amount < 20000 AND book_date > $1;
```

QUERY PLAN

```
-----  
----  
Finalize Aggregate  
  -> Gather  
      Workers Planned: 2  
      -> Partial Aggregate  
          -> Parallel Bitmap Heap Scan on bookings  
              Recheck Cond: (total_amount < '20000'::numeric)  
              Filter: (book_date > '2017-07-15 18:00:00+03'::timestamp with time  
zone)  
          -> Bitmap Index Scan on bookings_total_amount_idx  
              Index Cond: (total_amount < '20000'::numeric)  
(9 rows)
```

Ведущий процесс строит битовую карту в узле Bitmap Index Scan. Сканирование таблицы по построенной битовой карте выполняется параллельно в узле Parallel Bitmap Heap Scan.

Число рабочих процессов



Равно нулю (параллельный план не строится)

если *размер выборки* < *min_parallel_index_scan_size* = 512kB

Фиксировано

если для **таблицы** указан параметр хранения *parallel_workers*

Вычисляется по формуле

$1 + \lfloor \log_3(\text{размер выборки} / \text{min_parallel_index_scan_size}) \rfloor$

Но не больше, чем *max_parallel_workers_per_gather*

21

Число рабочих процессов выбирается примерно так же, как и в случае последовательного сканирования. Объем данных, который предполагается прочитать из индекса (определяемый числом индексных страниц), сравнивается со значением параметра *min_parallel_index_scan_size* (по умолчанию 512kB).

Если в случае последовательного сканирования таблицы объем данных определялся размером всей таблицы, то при индексном доступе планировщик должен спрогнозировать, сколько индексных страниц будет прочитано. Как именно это происходит, рассматривается позже в теме «Базовая статистика».

Если размер выборки меньше, параллельный план не рассматривается оптимизатором. Например, никогда не будет выполняться параллельно доступ к одному значению — в этом случае просто нечего распараллеливать.

Если размер выборки достаточно велик, число рабочих процессов определяется по формуле, если только оно не указано явно в параметре хранения *parallel_workers* таблицы (не индекса!).

Число процессов в любом случае не будет превышать значения параметра *max_parallel_workers_per_gather*.

Параллельные операции используют ресурсы процессора
в нескольких рабочих процессах

PostgreSQL выделяет рабочие процессы из общего пула

Все методы доступа могут работать в параллельном режиме

Выполните запрос, вычисляющий общее количество бронирований, с различными настройками планировщика и сравните планы выполнения:

1. С настройками по умолчанию.
2. Запретив последовательное сканирование.
3. Дополнительно запретив сканирование только индекса.
4. Разрешив все методы доступа, но отключив параллельность.

1. Речь идет о запросе

```
EXPLAIN (costs off)
SELECT count(book_ref) FROM bookings;
```

2. Установите параметр *enable_seqscan* в off.

3. Установите параметр *enable_indexonlyscan* в off.

4. Восстановите значения параметров по умолчанию и установите параметр *max_parallel_workers_per_gather* в 0.

1. План запроса с настройками по умолчанию

```
=> EXPLAIN (costs off)
SELECT count(book_ref) FROM bookings;

               QUERY PLAN
-----
Finalize Aggregate
  -> Gather
      Workers Planned: 2
      -> Partial Aggregate
          -> Parallel Seq Scan on bookings
(5 rows)
```

Параллельное последовательное сканирование оказывается наиболее выгодным.

2. Запрет последовательного сканирования

Отключим Seq Scan и повторим запрос:

```
=> SET enable_seqscan = off;

SET

=> EXPLAIN (costs off)
SELECT count(book_ref) FROM bookings;

               QUERY PLAN
-----
Finalize Aggregate
  -> Gather
      Workers Planned: 2
      -> Partial Aggregate
          -> Parallel Index Only Scan using bookings_pkey on bookings
(5 rows)
```

Индексный доступ медленнее табличного, но в данном случае достаточно просканировать только индекс.

3. Запрет сканирования только индекса

Отключим и Index Only Scan:

```
=> SET enable_indexonlyscan = off;

SET

=> EXPLAIN (costs off)
SELECT count(book_ref) FROM bookings;

               QUERY PLAN
-----
Aggregate
  -> Seq Scan on bookings
(2 rows)
```

Остальные параллельные способы получения данных оказываются невыгодными, лучше вообще не использовать параллельность.

4. Запрет параллельности

Сбросим все параметры и запретим параллельные планы:

```
=> RESET ALL;

RESET

=> SET max_parallel_workers_per_gather = 0;

SET

=> EXPLAIN (costs off)
SELECT count(book_ref) FROM bookings;
```


QUERY PLAN

```
-----  
Aggregate  
  -> Seq Scan on bookings  
(2 rows)
```

Планировщик выбрал последовательное сканирование как самый выгодный способ получить все строки.

Меняя значения конфигурационных параметров, можно исследовать поведение оптимизатора: например, понять, какие альтернативные планы он рассматривал.

=> **RESET ALL;**

RESET