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

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

Темы

B-деревья

Сканирование индекса

Сканирование только индекса

Индексы с дополнительными столбцами

Тонкости сортировки

Устранение дубликатов

B-дерево

Индекс

- вспомогательная структура во внешней памяти
- сопоставляет ключи и идентификаторы строк таблицы

Устройство: дерево поиска

- сбалансированное
- сильно ветвистое
- только сортируемые типы данных (операции «больше», «меньше»)
- результаты автоматически отсортированы

Использование

- ускорение доступа
- поддержка ограничений целостности

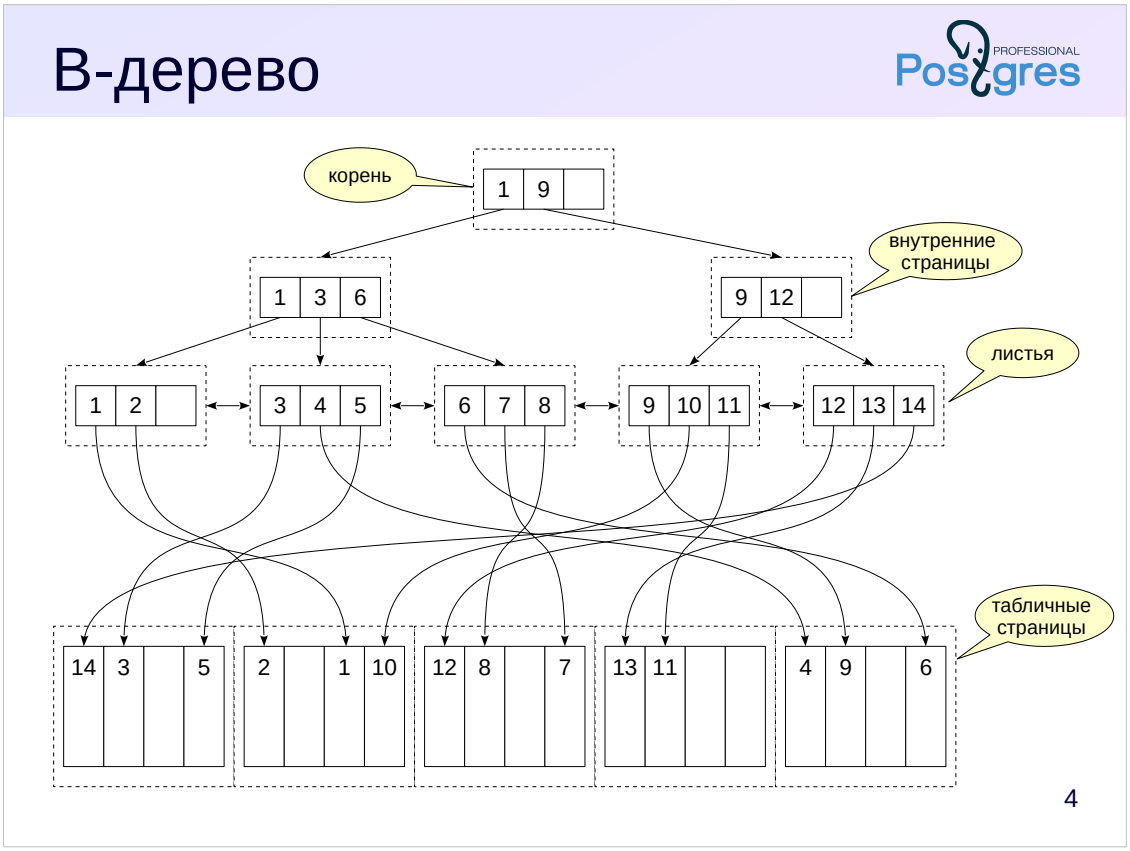
В этом модуле мы будем рассматривать только один из доступных в PostgreSQL типов индексов: B-дерево (B-tree). Это самый часто применяющийся на практике тип индекса. Некоторые другие типы индексов рассмотрены в курсе DEV2.

Как и все индексы в PostgreSQL, B-дерево является вторичной структурой — индекс не содержит в себе никакой информации, которую нельзя было бы получить из самой таблицы. Индекс можно удалить и пересоздать. Если бы не поддержка ограничений целостности (первичные и уникальные ключи), можно было бы сказать, что индексы влияют только на производительность, но не на логику работы.

Любой индекс сопоставляет значения проиндексированных полей (ключи поиска) и идентификаторы строк таблицы. Для этого в индексе B-tree строится упорядоченное дерево ключей, в котором можно быстро найти нужный ключ, а вместе с ним — и ссылку на табличную строку. Но проиндексировать можно только данные, допускающие сортировку (должны быть определены операции «больше», «меньше»). Например, числа, строки и даты могут быть проиндексированы, а точки на плоскости — нет (для них существуют другие типы индексов).

Особенностями B-дерева является его сбалансированность (постоянная глубина) и сильная ветвистость. Хотя размер дерева зависит от проиндексированных столбцов, на практике деревья обычно имеют глубину не больше 4–5.

Кроме поддержки ограничений целостности, задача B-деревьев (как и всех индексов) состоит в ускорении доступа к данным.



На слайде представлен пример B-дерева (верхняя часть рисунка).
У дерева есть корневая, внутренние и листовые страницы.

Страницы состоят из индексных записей, каждая из которых содержит:

- значения столбцов, по которым построен индекс (такие столбцы называются *ключевыми*);

- ссылку на другую страницу индекса или версию строки.

Внутри страницы ключи всегда упорядочены.

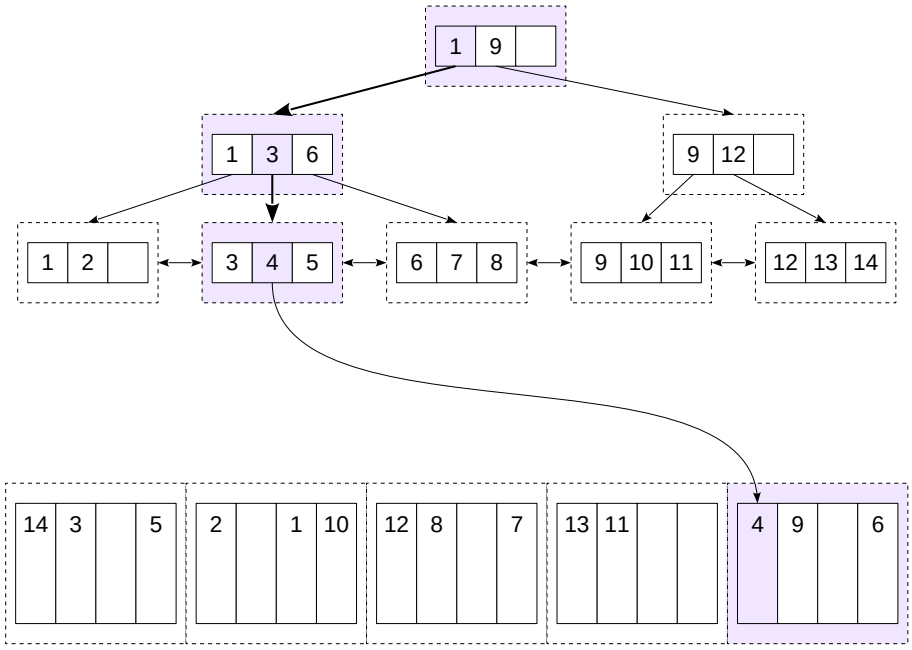
Листовые страницы ссылаются непосредственно на табличные версии строк, содержащие ключи индексирования.

Внутренние страницы ссылаются на нижележащие страницы индекса, а значения ключей определяют диапазон значений, которые можно обнаружить, спустившись по ссылке.

Самая верхняя страница дерева, на которую нет ссылок, называется корнем.

Индексная страница может быть заполнена не полностью. Свободное место используется для вставки в индекс новых значений. Если же на странице не хватает места — она разделяется на две новых страницы. Разделившиеся страницы никогда не объединяются, и это в некоторых случаях может приводить к разрастанию индекса.

Index scan: одно значение



Рассмотрим поиск одного значения с помощью индекса. Например, нам нужно найти в таблице строку, где значение проиндексированного столбца равно четырем.

Начинаем с корня дерева. Индексные записи в корневой странице определяют диапазоны значений ключей в нижележащих страницах: «от 1 до 9» и «9 и больше». Нам подходит диапазон «от 1 до 9», что соответствует строке с ключом 1. Заметим, что ключи хранятся упорядоченно, следовательно, поиск внутри страницы выполняется очень эффективно.

Ссылка из найденной строки приводит нас к странице второго уровня. В ней мы находим диапазон «от 3 до 6» (ключ 3) и переходим к странице третьего уровня.

Эта страница является листовой. В ней мы находим ключ, равный 4, и переходим на страницу таблицы.

В действительности процесс более сложен: мы не говорили про неуникальные ключи, про проблемы одновременного доступа и так далее. Но не будем углубляться в детали реализации.

Обратите внимание: на этой и следующих иллюстрациях цветом выделены те строки и те страницы, которые потребовалось прочитать.

Сканирование индекса

Рассмотрим таблицу бронирований:

```
=> \d bookings
```

```
                      Table "bookings.bookings"
    Column    |           Type           | Collation | Nullable | Default
--------------+--------------------------+-----------+----------+---------
 book_ref     | character(6)             |           | not null |
 book_date    | timestamp with time zone |           | not null |
 total_amount | numeric(10,2)            |           | not null |
Indexes:
    "bookings_pkey" PRIMARY KEY, btree (book_ref)
Referenced by:
    TABLE "tickets" CONSTRAINT "tickets_book_ref_fkey" FOREIGN KEY (book_ref) REFERENCES bookings(book_ref)
```

Столбец book_ref является первичным ключом и для него автоматически был создан индекс bookings_pkey.

Проверим план запроса с поиском одного значения:

```
=> EXPLAIN SELECT * FROM bookings WHERE book_ref = 'CDE08B';
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using bookings_pkey on bookings  (cost=0.43..8.45 rows=1 width=21)
   Index Cond: (book_ref = 'CDE08B'::bpchar)
(2 rows)
```

Выбран метод доступа Index Scan, указано имя использованного индекса. Здесь обращение и к индексу, и к таблице представлено одним узлом плана. Строкой ниже указано условие доступа.

Начальная стоимость индексного доступа — оценка ресурсов для спуска к листовому узлу. Она зависит от логарифма количества листовых узлов (количество операций сравнения, которые надо выполнить) и от высоты дерева. При оценке считается, что необходимые страницы окажутся в кеше, и оцениваются только ресурсы процессора: цифра получается небольшой.

Полная стоимость добавляет оценку чтения необходимых листовых страниц индекса и табличных страниц.

В данном случае, поскольку индекс уникальный, будет прочитана одна индексная страница и одна табличная. Стоимость каждого из чтений оценивается параметром random_page_cost:

```
=> SELECT current_setting('random_page_cost');
```

```
 current_setting
-----------------
 4
(1 row)
```

Его значение обычно больше, чем seq_page_cost, поскольку произвольный доступ стоит дороже (хотя для SSD-дисков этот параметр следует существенно уменьшить).

Итого получаем 8, и еще немного добавляет оценка процессорного времени на обработку строк.

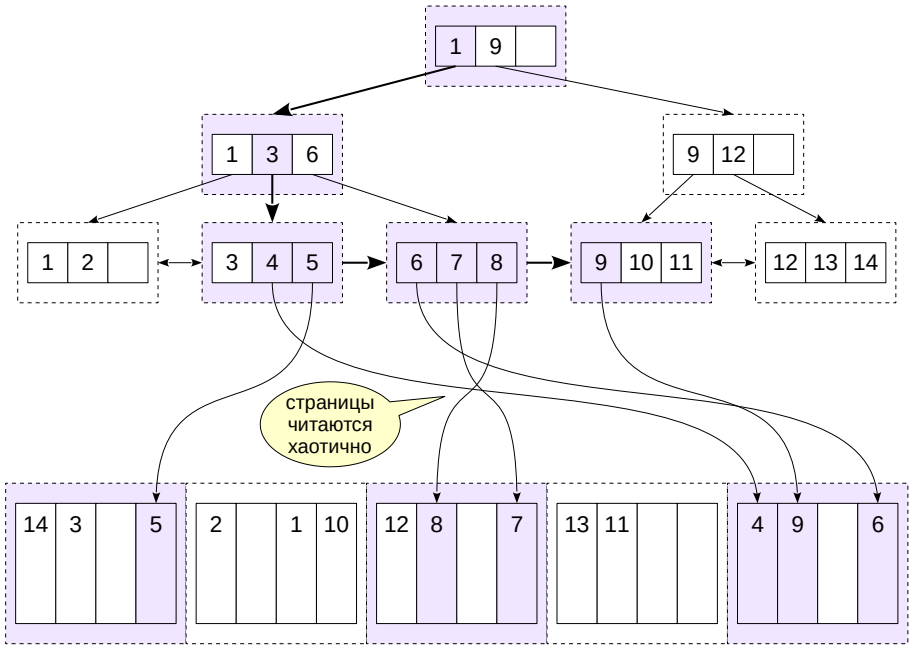
В строке Index Cond плана указываются только те условия, по которым происходит обращение к индексу или которые могут быть проверены на уровне индекса.

Дополнительные условия, которые можно проверить только по таблице, отображаются в отдельной строке Filter:

```
=> EXPLAIN SELECT * FROM bookings
WHERE book_ref = 'CDE08B' AND total_amount > 1000;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using bookings_pkey on bookings  (cost=0.43..8.45 rows=1 width=21)
   Index Cond: (book_ref = 'CDE08B'::bpchar)
   Filter: (total_amount > '1000'::numeric)
(3 rows)
```

Index scan: диапазон



B-дерево позволяет эффективно искать не только отдельные значения, но и диапазоны значений (по условиям «меньше», «больше», «меньше или равно», «больше или равно», а также «between»).

Вот как это происходит. Сначала мы ищем крайний ключ условия. Например, для условия «от 4 до 9» мы можем выбрать значение 4 или 9, а для условия «меньше 9» надо взять 9. Затем спускаемся до листовой страницы индекса так, как мы рассматривали в предыдущем примере, и получаем первое значение из таблицы.

Дальше остается двигаться по листовым страницам индекса вправо (или влево, в зависимости от условия), перебирая строки этих страниц до тех пор, пока мы не встретим ключ, выпадающий из диапазона.

На слайде показан пример поиска значений по условию «x BETWEEN 4 AND 9» или, что тоже самое, «x >= 4 AND x <= 9». Спустившись к значению 4, мы затем перебираем ключи 5, 6, и так далее до 9. Встретив ключ 10, прекращаем поиск.

Нам помогают два свойства: упорядоченность ключей на всех страницах и связанность листовых страниц двунаправленным списком.

Обратите внимание, что к одной и той же табличной странице нам пришлось обращаться несколько раз. Мы прочитали последнюю страницу таблицы (значение 4), затем первую (5), затем опять последнюю (6) и так далее.

Поиск по диапазону

Мы получаем данные из индекса, спускаясь от корня дерева к левому листовому узлу и проходя по списку листовых страниц. Поэтому индексное сканирование всегда возвращает данные в том порядке, в котором они хранятся в дереве индекса и который был указан при его создании:

```
=> SELECT * FROM bookings
WHERE book_ref > '000900' AND book_ref < '000939'
ORDER BY book_ref;
```

```
 book_ref |       book_date        | total_amount
----------+------------------------+--------------
 000906   | 2016-10-10 00:32:00+03 |     73500.00
 000909   | 2017-08-01 04:30:00+03 |     99700.00
 000917   | 2017-07-27 00:01:00+03 |     28000.00
 000930   | 2017-04-04 00:50:00+03 |    260900.00
 000938   | 2016-09-15 15:59:00+03 |    123000.00
(5 rows)
```

Тот же самый индекс может использоваться и для получения строк в обратном порядке:

```
=> SELECT * FROM bookings
WHERE book_ref > '000900' AND book_ref < '000939'
ORDER BY book_ref DESC;
```

```
 book_ref |       book_date        | total_amount
----------+------------------------+--------------
 000938   | 2016-09-15 15:59:00+03 |    123000.00
 000930   | 2017-04-04 00:50:00+03 |    260900.00
 000917   | 2017-07-27 00:01:00+03 |     28000.00
 000909   | 2017-08-01 04:30:00+03 |     99700.00
 000906   | 2016-10-10 00:32:00+03 |     73500.00
(5 rows)
```

В этом случае мы спускаемся от корня дерева к правому листовому узлу, и проходим по списку листовых страниц в обратную сторону.

Обратите внимание на количество страниц (Buffers), которое потребовалось прочитать:

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT * FROM bookings
WHERE book_ref > '000900' AND book_ref < '000939'
ORDER BY book_ref DESC;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan Backward using bookings_pkey on bookings (actual rows=5 loops=1)
   Index Cond: ((book_ref > '000900'::bpchar) AND (book_ref < '000939'::bpchar))
   Buffers: shared hit=5
 Planning:
   Buffers: shared hit=8
(5 rows)
```

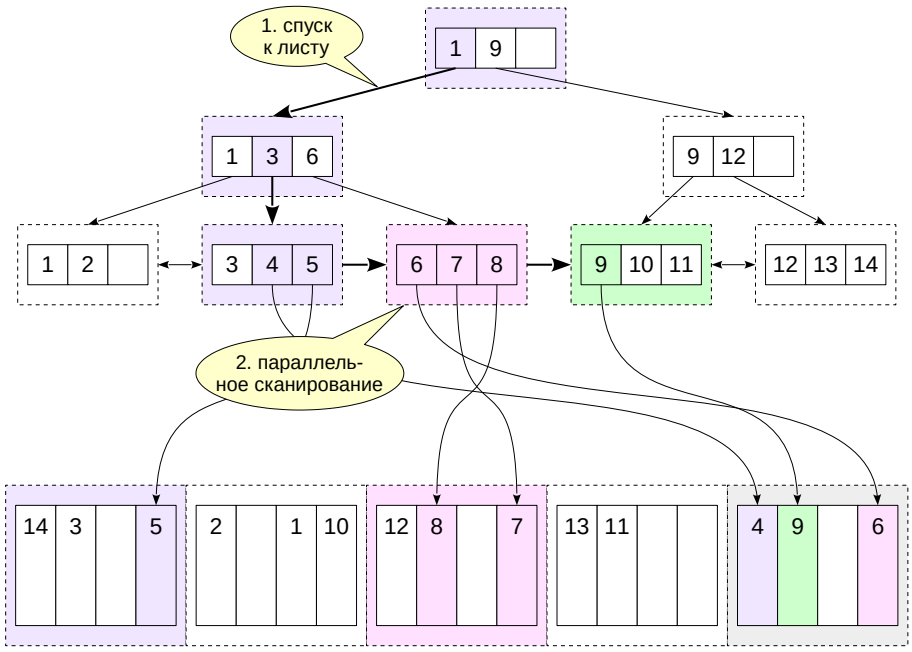
Сравним поиск по диапазону с повторяющимся поиском отдельных значений. Получим тот же результат с помощью конструкции IN, и посмотрим, сколько страниц потребовалось прочитать в этом случае:

```
=> EXPLAIN (analyze, buffers, costs off)
SELECT * FROM bookings
WHERE book_ref IN ('000906','000909','000917','000930','000938')
ORDER BY book_ref DESC;
```

```
                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Index Scan Backward using bookings_pkey on bookings (actual time=0.032..0.049 rows=5 loops=1)
   Index Cond: (book_ref = ANY ('{000906,000909,000917,000930,000938}'::bpchar[]))
   Buffers: shared hit=24
 Planning Time: 0.080 ms
 Execution Time: 0.061 ms
(5 rows)
```

Количество страниц увеличилось, поскольку в этом случае приходится спускаться от корня к каждому значению.

Parallel Index Scan



Индексный доступ может выполняться в параллельном режиме. Это происходит в два этапа. Сначала ведущий процесс спускается от корня дерева к листовой странице. Затем рабочие процессы выполняют параллельное чтение листовых страниц индекса, двигаясь по указателям.

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 иллюстрации: 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

Сканирование индекса может выполняться в параллельном режиме. В качестве примера найдем общую сумму всех бронирований с номерами, меньшими 400000 (их примерно одна четверть от общего числа):

```
=> RESET max_parallel_workers_per_gather;
```

```
RESET
```

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

```
                                                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=16864.81..16864.82 rows=1 width=32)
   ->  Gather  (cost=16864.59..16864.80 rows=2 width=32)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=15864.59..15864.60 rows=1 width=32)
               ->  Parallel Index Scan using bookings_pkey on bookings  (cost=0.43..15314.82 rows=219907 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 — узел Parallel Index Scan.

Полная стоимость складывается из стоимостей доступа к таблице и к индексу. Стоимость доступа к 1/4 табличных страниц (поделенных между процессами) оценивается аналогично последовательному сканированию:

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

```
 round
-------
  5561
(1 row)
```

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
) FROM pg_class WHERE relname = 'bookings_pkey';
```

```
 round
-------
  9750
(1 row)
```

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

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

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

Но не больше, чем *max_parallel_workers_per_gather*

Число рабочих процессов выбирается примерно так же, как и в случае последовательного сканирования. Сравнивается объем данных, который предполагается прочитать из индекса (определяемый числом индексных страниц) 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 «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*.

Число рабочих процессов

Установим максимальное число параллельно работающих процессов, которые могут запускаться одним узлом Gather, в значение четыре:

```
=> SET max_parallel_workers_per_gather = 4;
```

```
SET
```

Посмотрим значение конфигурационного параметра min_parallel_index_scan_size:

```
=> SHOW min_parallel_index_scan_size;
```

```
 min_parallel_index_scan_size
------------------------------
 512kB
(1 row)
```

И выполним следующий запрос:

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

```
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=16244.21..16244.22 rows=1 width=32)
   ->  Gather  (cost=16243.88..16244.19 rows=3 width=32)
         Workers Planned: 3
         ->  Partial Aggregate  (cost=15243.88..15243.89 rows=1 width=32)
               ->  Parallel Index Scan using bookings_pkey on bookings  (cost=0.43..14818.25 rows=170250 width=6)
                     Index Cond: (book_ref < '400000'::bpchar)
(6 rows)
```

Было запланировано три параллельных процесса. Если увеличить значение min_parallel_index_scan_size до 10 мегабайт, планировщик запланирует лишь один процесс:

```
=> SET min_parallel_index_scan_size = '10MB';
```

```
SET
```

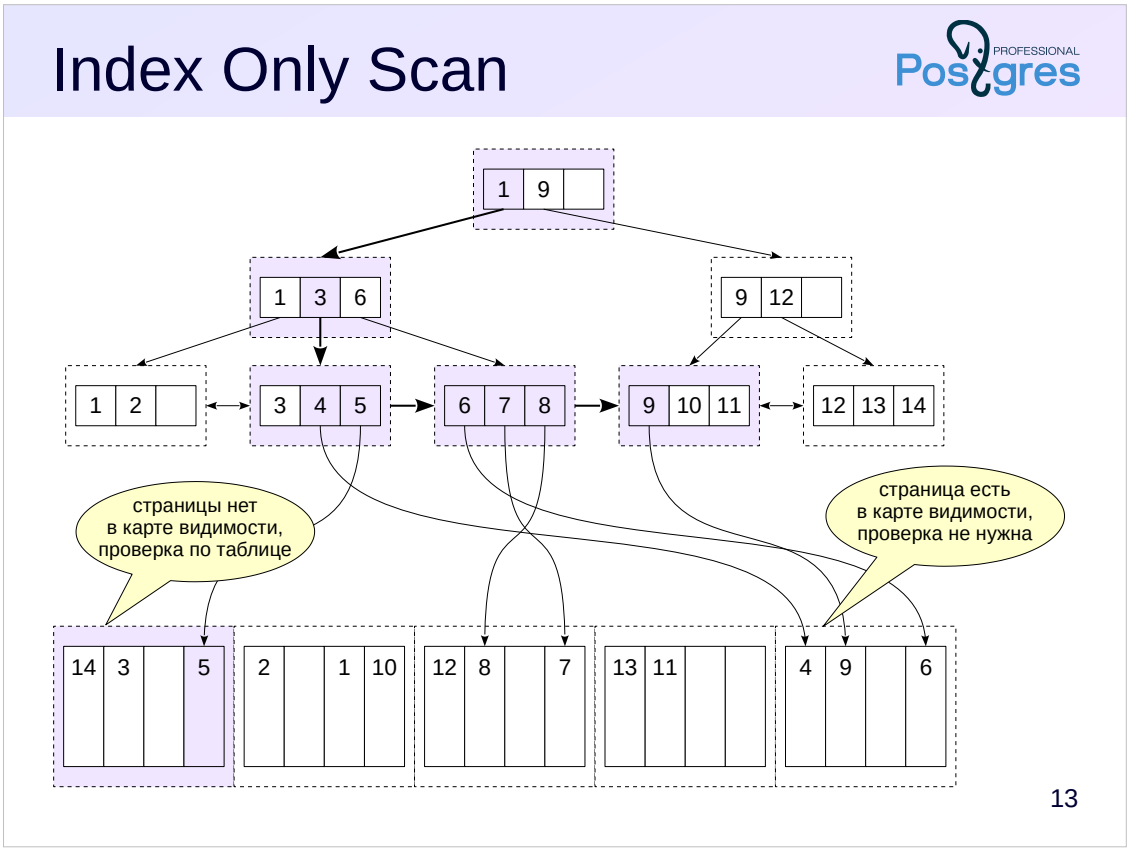
И повторим запрос:

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

```
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=17996.57..17996.58 rows=1 width=32)
   ->  Gather  (cost=17996.46..17996.57 rows=1 width=32)
         Workers Planned: 1
         ->  Partial Aggregate  (cost=16996.46..16996.47 rows=1 width=32)
               ->  Parallel Index Scan using bookings_pkey on bookings  (cost=0.43..16220.31 rows=310456 width=6)
                     Index Cond: (book_ref < '400000'::bpchar)
(6 rows)
```

```
=> RESET min_parallel_index_scan_size;
```

```
RESET
```



Если в запросе требуются только проиндексированные данные, то они уже есть в самом индексе — к таблице в этом случае обращаться не надо. Такой индекс называется *покрывающим* для запроса.

Это хорошая оптимизация, исключающая обращения к табличным страницам. Но, к сожалению, индексные страницы не содержат информацию о видимости строк — чтобы проверить, надо ли показывать найденную в индексе строку, мы вынуждены заглянуть и в табличную страницу, что сводит оптимизацию на нет.

Поэтому критическую роль в эффективности сканирования только индекса играет карта видимости. Если табличная страница содержит гарантированно видимые данные, и это отражено в карте видимости, к такой табличной странице обращаться не надо. Но страницы, не отмеченные в карте видимости, посетить все-таки придется.

Это одна из причин, по которой стоит выполнять очистку (vacuum) достаточно часто — именно этот процесс обновляет карту видимости.

Планировщик не знает точно, сколько табличных страниц потребуют проверки, но учитывает оценку этого числа. При плохом прогнозе планировщик может отказаться от использования сканирования только индекса.

При обработке каждой индексной записи сначала проверяется наличие табличной страницы в карте видимости, а при ее отсутствии читается сама табличная страница.

Сканирование только индекса

Если вся необходимая информация содержится в самом индексе, то нет необходимости обращаться к таблице — за исключением проверки видимости:

```
=> EXPLAIN SELECT book_ref FROM bookings WHERE book_ref <= '100000';

                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Index Only Scan using bookings_pkey on bookings  (cost=0.43..4202.31 rows=147536 width=7)
   Index Cond: (book_ref <= '100000'::bpchar)
(2 rows)
```

Посмотрим план этого запроса с помощью EXPLAIN ANALYZE:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT book_ref FROM bookings WHERE book_ref <= '100000';

                                QUERY PLAN
--------------------------------------------------------------------------
 Index Only Scan using bookings_pkey on bookings (actual rows=132109 loops=1)
   Index Cond: (book_ref <= '100000'::bpchar)
   Heap Fetches: 0
(3 rows)
```

Строка Heap Fetches показывает, сколько версий строк было проверено с помощью таблицы. В данном случае карта видимости содержит актуальную информацию, обращаться к таблице не потребовалось.

Обновим первую строку таблицы:

```
=> UPDATE bookings
SET total_amount = total_amount
WHERE book_ref = '000004';

UPDATE 1
```

Сколько версий строк придется проверить теперь?

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT book_ref FROM bookings WHERE book_ref <= '100000';

                                QUERY PLAN
--------------------------------------------------------------------------
 Index Only Scan using bookings_pkey on bookings (actual rows=132109 loops=1)
   Index Cond: (book_ref <= '100000'::bpchar)
   Heap Fetches: 158
(3 rows)
```

Проверять приходится все версии, попадающие на измененную страницу.

Include-индексы

Индекс с дополнительными неключевыми столбцами

```
CREATE INDEX ... INCLUDE (...)
```

Неключевые столбцы

- не используются при поиске по индексу
- не учитываются ограничением уникальности
- значения хранятся в индексной записи и возвращаются без обращения к таблице

Покрывающий индекс как правило увеличивает эффективность выборки. Чтобы сделать индекс покрывающим, в него может понадобиться добавить столбцы, но это не всегда возможно:

- добавление столбца в уникальный индекс нарушит гарантию уникальности исходных столбцов;

- тип данных добавляемого столбца может не поддерживаться индексом.

В таких случаях начиная с версии PostgreSQL 11 можно добавить к индексу *неключевые столбцы*, указав их в предложении INCLUDE.

https://postgrespro.ru/docs/postgresql/13/sql-createindex

Значения таких столбцов не формируют дерево индекса, а просто хранятся как дополнительные сведения в индексных записях листовых страниц. Поиск по неключевым столбцам не работает, но их значения могут возвращаться без обращения к таблице.

В настоящее время include-индексы поддерживаются только для индексов на основе B-деревьев и для GiST-индексов.

Include-индексы создаются, чтобы индекс стал покрывающим, но не стоит путать эти два термина. Индекс вполне может быть покрывающим для некоторого запроса, но не использовать предложение INCLUDE. А Include-индекс может не быть покрывающим для каких-то запросов.

Include-индексы

Индекс tickets_pkey не является покрывающим для приведенного запроса, поскольку в нем требуется не только столбец ticket_no (есть в индексе), но и book_ref:

```
=> EXPLAIN (analyze, buffers, costs off, summary off)
SELECT ticket_no, book_ref FROM tickets WHERE ticket_no > '0005435990286';
```

```
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Index Scan using tickets_pkey on tickets (actual time=1.846..12.327 rows=7146 loops=1)
   Index Cond: (ticket_no > '0005435990286'::bpchar)
   Buffers: shared hit=72 read=150
 Planning:
   Buffers: shared hit=19 read=7
(5 rows)
```

В данном случае было прочитано 222 страницы (Buffers).

Создадим include-индекс, добавив в него неключевой столбец book_ref, так как он требуется запросу:

```
=> CREATE UNIQUE INDEX ON tickets (ticket_no) INCLUDE (book_ref);
```

```
CREATE INDEX
```

Повторим запрос:

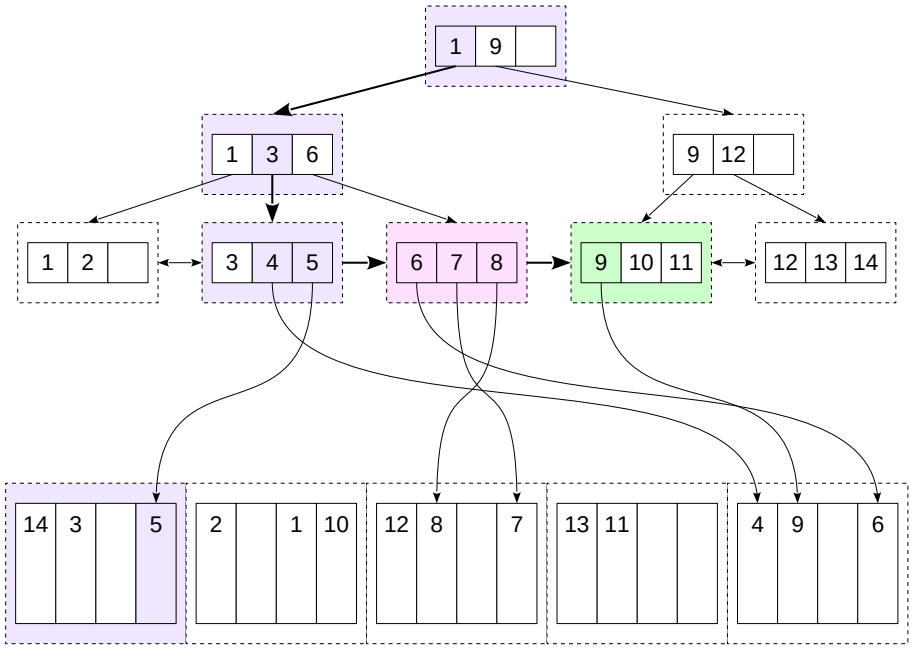
```
=> EXPLAIN (analyze, buffers, costs off, summary off)
SELECT ticket_no, book_ref FROM tickets WHERE ticket_no > '0005435990286';
```

```
                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Index Only Scan using tickets_ticket_no_book_ref_idx on tickets (actual time=0.075..2.691 rows=7146 loops=1)
   Index Cond: (ticket_no > '0005435990286'::bpchar)
   Heap Fetches: 0
   Buffers: shared hit=4 read=35
 Planning:
   Buffers: shared hit=19 read=4
(6 rows)
```

Теперь оптимизатор выбирает метод Index Only Scan и использует только что созданный индекс. Количество прочитанных страниц сократилось. Поскольку карта видимости актуальна, обращаться к таблице не пришлось (Heap Fetches: 0).

В include-индекс можно включать столбцы с типами данных, которые не поддерживаются B-деревом (например, геометрические типы и xml).

Parallel Index Only Scan



Сканирование только индекса 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

Сканирование только индекса также может выполняться параллельно:

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

```
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=12879.21..12879.22 rows=1 width=8)
   ->  Gather  (cost=12878.89..12879.20 rows=3 width=8)
         Workers Planned: 3
         ->  Partial Aggregate  (cost=11878.89..11878.90 rows=1 width=8)
               ->  Parallel Index Only Scan using bookings_pkey on bookings  (cost=0.43..11453.26 rows=170251 width=7)
                     Index Cond: (book_ref <= '400000'::bpchar)
(6 rows)
```

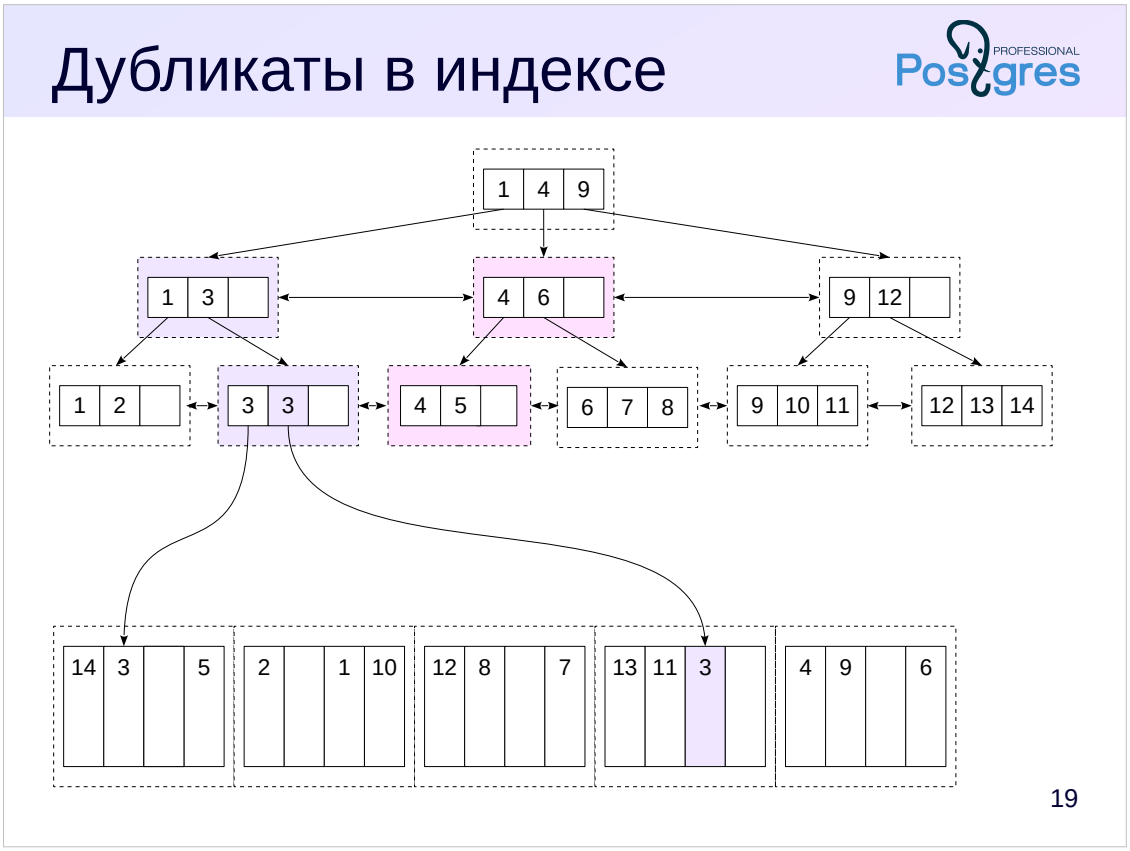
Стоимость доступа к таблице здесь учитывает только обработку строк, без ввода-вывода:

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

```
 round
-------
  2199
(1 row)
```

Вклад индексного доступа остается прежним.

Дальше мы не будем подробно останавливаться на расчете стоимости. Для этого используется достаточно сложная математическая модель; в нашу задачу входит только показать общий принцип.



Дубликаты в индексе — это записи с одинаковыми значениями ключей, указывающие на разные версии строк таблицы.

Дубликаты возникают по двум причинам. Во-первых, *разные строки* могут иметь одинаковые значения проиндексированных столбцов — в этом случае мы имеем дело с неуникальным индексом. Во-вторых, из-за работы механизма многоверсионности могут возникать *разные версии* одной и той же строки — такие дубликаты постоянно возникают (и затем удаляются очисткой) и в уникальных индексах тоже.

Предположим, что в таблицу была добавлена еще одна строка со значением 3.

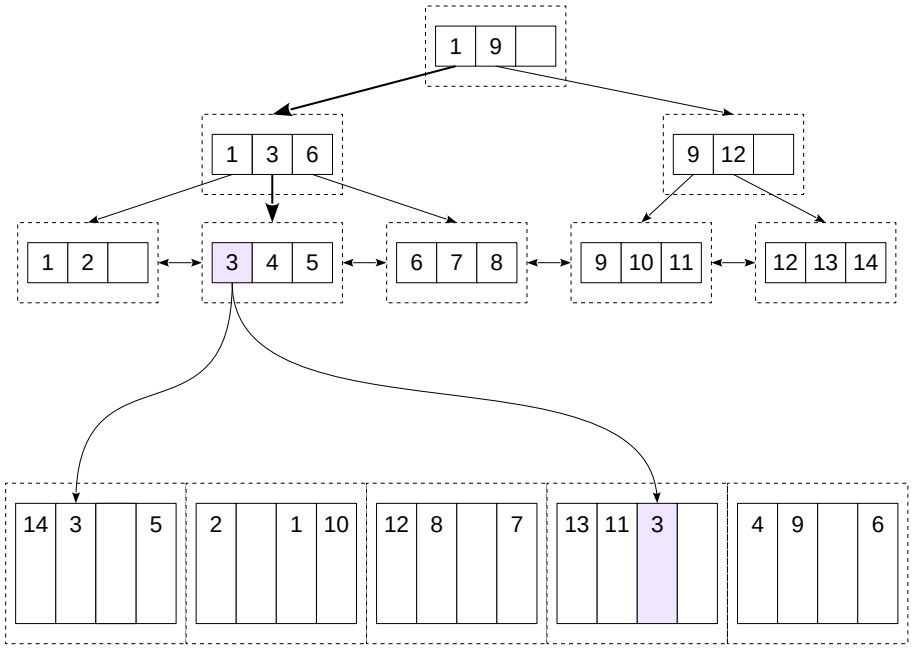
Новую индексную запись невозможно добавить в уже заполненную листовую страницу (с ключами 3, 4, 5) и происходит расщепление: в индексе появляется еще одна листовая страница и записи распределяются между ними.

В вышестоящей внутренней странице (с ключами 1, 3, 6) должна появиться индексная запись, ссылающаяся на новую листовую страницу. Но для этой записи нет места; поэтому внутренняя страница также расщепляется. (Для корректной работы индекса во время расщепления внутренние страницы одного уровня на самом деле тоже связаны двунаправленным списком.)

В корневой странице дерева нашлось место для новой записи, поэтому расщепления корня и увеличения глубины дерева в данном случае не произошло.

На рисунке новые страницы показаны розовым цветом.

Исключение дубликатов



Дубликаты приводят к разрастанию индекса, а расщепленные страницы никогда не объединяются. Начиная с 13-й версии для избежания лишних расщеплений используется механизм *исключения дубликатов*: индексные записи с одинаковыми ключами группируются в одну общую запись со списком ссылок на версии строк.

Для экономии ресурсов группировка записей происходит только в том случае, если иначе страницу пришлось бы расщепить. Поскольку дубликаты возникают в том числе из-за появления версий одних и тех же строк, механизм исключения дубликатов может предотвратить разрастание даже уникальных индексов. Но если известно, что дубликаты не образуются, механизм можно отключить параметром хранения *deduplicate_items*.

В нашем примере дублирующиеся тройки сгруппированы в одну индексную запись. За счет этого удалось избежать расщеплений; размер индекса не изменился.

Исключение дубликатов применяется только с типами данных, значения которых можно сравнивать побитно (не подходят типы numeric, float4, float8, jsonb; для некоторых правил сортировки не подходят текстовые типы). Механизм не работает с Include-индексами (информация о них представлена ниже), не реализована поддержка типов-контейнеров.

https://postgrespro.ru/docs/postgresql/13/btree-implementation#BTREE-DEDUPLICATION

При обновлении версии сервера с помощью pg_upgrade необходимо выполнить REINDEX для существующих индексов, чтобы задействовать для них возможность устранения дубликатов.

Исключение дубликатов

Сравним размер индекса без исключения дубликатов и с исключением. Пример подобран таким образом, что в индексе должно быть много повторяющихся значений.

Сначала создадим индекс, отключив исключение дубликатов с помощью параметра хранения deduplicate_items:

```
=> CREATE INDEX dedup_test ON ticket_flights (fare_conditions)
WITH (deduplicate_items = off);
```

```
CREATE INDEX
```

Посмотрим размер созданного индекса:

```
=> SELECT pg_size_pretty(pg_total_relation_size('dedup_test'));
```

```
 pg_size_pretty
----------------
 187 MB
(1 row)
```

Выполним запрос с помощью индекса, отключив для этого последовательное сканирование.

Обратите внимание на значение Buffers и время выполнения запроса.

```
=> SET enable_seqscan = off;
```

```
SET
```

```
=> EXPLAIN (analyze, buffers, costs off)
SELECT fare_conditions FROM ticket_flights;
```

```
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Index Only Scan using dedup_test on ticket_flights (actual time=0.058..945.659 rows=8391852 loops=1)
   Heap Fetches: 0
   Buffers: shared hit=4 read=23877
 Planning:
   Buffers: shared hit=21 read=2
 Planning Time: 1.833 ms
 Execution Time: 1249.710 ms
(7 rows)
```

Удалим индекс и создадим его заново. Параметр хранения deduplicate_items не указываем, так как по умолчанию он включен:

```
=> DROP INDEX dedup_test;
```

```
DROP INDEX
```

```
=> CREATE INDEX dedup_test ON ticket_flights (fare_conditions);
```

```
CREATE INDEX
```

Опять посмотрим размер индекса:

```
=> SELECT pg_size_pretty(pg_total_relation_size('dedup_test'));
```

```
 pg_size_pretty
----------------
 56 MB
(1 row)
```

Размер сократился более чем в три раза.

Повторно выполним запрос:

```
=> EXPLAIN (analyze, buffers, costs off)
SELECT fare_conditions FROM ticket_flights;
```

```
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------
 Index Only Scan using dedup_test on ticket_flights (actual time=0.043..628.928 rows=8391852 loops=1)
   Heap Fetches: 0
   Buffers: shared hit=5 read=7077
 Planning:
   Buffers: shared hit=5 read=1
 Planning Time: 0.141 ms
 Execution Time: 923.149 ms
(7 rows)
```

Количество Buffers тоже сократилось примерно в три раза. И запрос стал выполняться быстрее.

```
=> RESET enable_seqscan;
```

```
RESET
```

Итоги

Индексы

- ускоряют доступ при высокой селективности
- поддерживают ограничения целостности
- позволяют получить отсортированные данные

Покрывающие индексы

- позволяют экономить на обращении к таблице

Возможен параллельный индексный доступ

Практика

1. Напишите запрос, выбирающий максимальную сумму бронирования.
   Проверьте план выполнения. Какой метод доступа выбрал планировщик? Эффективен ли такой доступ?
2. Создайте индекс по столбцу bookings.total_amount.
   Снова проверьте план выполнения запроса. Какой метод доступа выбрал планировщик теперь?
3. При создании индекса можно указать порядок сортировки столбца. Зачем, если индекс можно просматривать в любом направлении?
4. В демонстрации был создан include-индекс для таблицы билетов. Замените им индекс, поддерживающий первичный ключ таблицы.

1. Этот запрос можно написать как минимум двумя разными способами. Во-первых, можно воспользоваться предложениями ORDER BY и LIMIT 1. Во-вторых, можно вывести максимальное число с помощью агрегатной функции max.

Попробуйте оба варианта.

3. Подсказка: это важно для индексов, построенных по нескольким столбцам.

4. Используйте команду ALTER TABLE … DROP CONSTRAINT для удаления старого ограничения целостности вместе с соответствующим индексом, и команду ALTER TABLE … ADD CONSTRAINT … USING INDEX для создания нового ограничения, используя существующий индекс.

1. Максимальная сумма бронирования

```
=> EXPLAIN SELECT total_amount FROM bookings ORDER BY total_amount DESC LIMIT 1;

                                     QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=27641.46..27641.58 rows=1 width=6)
   ->  Gather Merge  (cost=27641.46..232902.56 rows=1759258 width=6)
         Workers Planned: 2
         ->  Sort  (cost=26641.44..28840.51 rows=879629 width=6)
               Sort Key: total_amount DESC
               ->  Parallel Seq Scan on bookings  (cost=0.00..22243.29 rows=879629 width=6)
(6 rows)
```

Планировщик последовательно сканирует всю таблицу, а затем сортирует данные. Это неэффективный доступ, поскольку нам требуется только одно значение. Но, пока нет индекса, это единственно возможный способ.

2. Индекс

Создаем индекс.

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

CREATE INDEX
```

Повторяем запрос:

```
=> EXPLAIN SELECT total_amount FROM bookings ORDER BY total_amount DESC LIMIT 1;

                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.43..0.46 rows=1 width=6)
   ->  Index Only Scan Backward using bookings_total_amount_idx on bookings  (cost=0.43..54867.08 rows=2111110 width=6)
(2 rows)
```

Теперь планировщик выбрал только индексное сканирование.

Сформулируем запрос по-другому:

```
=> EXPLAIN SELECT max(total_amount) FROM bookings;

                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Result  (cost=0.46..0.47 rows=1 width=32)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.43..0.46 rows=1 width=6)
           ->  Index Only Scan Backward using bookings_total_amount_idx on bookings  (cost=0.43..60144.86 rows=2111110 width=6)
                 Index Cond: (total_amount IS NOT NULL)
(5 rows)
```

Видно, что и в этом случае планировщик выбрал индексное сканирование (добавив условие NOT NULL, поскольку функция max не должна учитывать неопределенные значения).

Новый узел плана, который здесь появляется — InitPlan. Он соответствует подзапросу, который выполняется один раз. То есть, фактически, планировщик переформулировал запрос следующим образом:

```
=> EXPLAIN SELECT (
  SELECT total_amount FROM bookings
  WHERE total_amount IS NOT NULL
  ORDER BY total_amount DESC LIMIT 1
);

                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Result  (cost=0.46..0.47 rows=1 width=16)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.43..0.46 rows=1 width=6)
           ->  Index Only Scan Backward using bookings_total_amount_idx on bookings  (cost=0.43..60144.86 rows=2111110 width=6)
                 Index Cond: (total_amount IS NOT NULL)
(5 rows)
```

Если бы подзапрос выполнялся несколько раз, вместо InitPlan в плане появился бы узел SubPlan.

3. Порядок сортировки при создании индекса

Порядок важен для многоколоночных индексов. Создадим индекс на таблице рейсов по аэропортам вылета и прилета:

```
=> CREATE INDEX dep_arr on flights(departure_airport, arrival_airport);

CREATE INDEX
```

Индекс может использоваться для такого запроса:

```
=> EXPLAIN SELECT * FROM flights ORDER BY departure_airport, arrival_airport;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using dep_arr on flights  (cost=0.29..14417.83 rows=214867 width=63)
(1 row)
```

И для такого:

```
=> EXPLAIN SELECT * FROM flights ORDER BY departure_airport DESC, arrival_airport DESC;

                                      QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan Backward using dep_arr on flights  (cost=0.29..14417.83 rows=214867 width=63)
(1 row)
```

Но не для такого (сортировка в разных направлениях):

```
=> EXPLAIN SELECT * FROM flights ORDER BY departure_airport, arrival_airport DESC;

                            QUERY PLAN
-------------------------------------------------------------------
 Sort  (cost=31883.96..32421.12 rows=214867 width=63)
   Sort Key: departure_airport, arrival_airport DESC
   ->  Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(3 rows)
```

Здесь приходится отдельно выполнять сортировку результатов. В этом случае поможет другой индекс:

```
=> CREATE INDEX dep_asc_arr_desc ON flights(departure_airport, arrival_airport DESC);

CREATE INDEX

=> EXPLAIN SELECT * FROM flights ORDER BY departure_airport, arrival_airport DESC;

                                      QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using dep_asc_arr_desc on flights  (cost=0.29..14417.83 rows=214867 width=63)
(1 row)
```

А также для запроса с обратным порядком сортировки:

```
=> EXPLAIN SELECT * FROM flights ORDER BY departure_airport DESC, arrival_airport;

                                         QUERY PLAN
-----------------------------------------------------------------------------------------------
 Index Scan Backward using dep_asc_arr_desc on flights  (cost=0.29..14417.83 rows=214867 width=63)
(1 row)
```

4. Include-индекс для первичного ключа

В демонстрации был создан такой include-индекс:

```
=> CREATE UNIQUE INDEX tickets_ticket_no_book_ref_idx
ON tickets (ticket_no) INCLUDE (book_ref);

CREATE INDEX
```

Теперь индекс tickets_pkey является избыточным и может быть заменен на новый. Для этого в транзакции удалим старое ограничение целостности (при этом удалится и старый индекс) и добавим новое ограничение, указав имя уже созданного нового индекса. При этом надо учесть наличие внешнего ключа на таблице ticket_flights, которое тоже придется создать заново:

```
=> BEGIN;

BEGIN

=> ALTER TABLE tickets DROP CONSTRAINT tickets_pkey CASCADE;

NOTICE:  drop cascades to constraint ticket_flights_ticket_no_fkey on table ticket_flights
ALTER TABLE

=> ALTER TABLE tickets ADD CONSTRAINT tickets_pkey PRIMARY KEY USING INDEX tickets_ticket_no_book_ref_idx;

NOTICE:  ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index "tickets_ticket_no_book_ref_idx" to "tickets_pkey"
ALTER TABLE

=> ALTER TABLE ticket_flights
ADD FOREIGN KEY (ticket_no) REFERENCES tickets(ticket_no);

ALTER TABLE

=> COMMIT;

COMMIT

=> \d tickets
```

```
                      Table "bookings.tickets"
     Column     |         Type          | Collation | Nullable | Default
----------------+-----------------------+-----------+----------+---------
 ticket_no      | character(13)         |           | not null |
 book_ref       | character(6)          |           | not null |
 passenger_id   | character varying(20) |           | not null |
 passenger_name | text                  |           | not null |
 contact_data   | jsonb                 |           |          |
Indexes:
    "tickets_pkey" PRIMARY KEY, btree (ticket_no) INCLUDE (book_ref)
Foreign-key constraints:
    "tickets_book_ref_fkey" FOREIGN KEY (book_ref) REFERENCES bookings(book_ref)
Referenced by:
    TABLE "ticket_flights" CONSTRAINT "ticket_flights_ticket_no_fkey" FOREIGN KEY (ticket_no) REFERENCES tickets(ticket_no)
```