

Доступ к данным Методы доступа



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.

Последовательное сканирование (Seq Scan)

Сканирование индекса (Index Scan)

Сканирование по битовой карте (Bitmap Scan)

Сканирование только индекса (Index-Only Scan)

Сравнение эффективности методов доступа

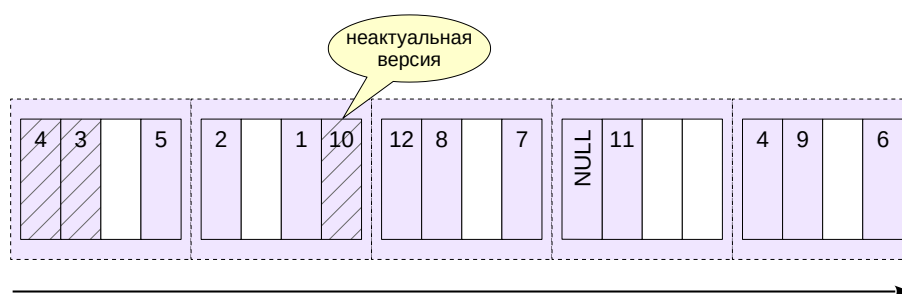
Последовательное чтение всех страниц

страницы читаются в кеш

проверяется видимость версий строк

данные возвращаются в произвольном порядке

время сканирования зависит от физического размера файла



3

В распоряжении оптимизатора имеется несколько способов доступа к данным. Самый простой из них — последовательное сканирование таблицы. Файл (или файлы) основного слоя таблицы читается постранично от начала до конца. Напомним, что чтение происходит через буферный кеш (для временных таблиц — через локальный кеш сеанса).

Последовательное чтение файла позволяет использовать тот факт, что операционная система обычно читает данные порциями большими, 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).

При последовательном сканировании на каждой странице рассматриваются все версии строк — в том числе и неактуальные (мертвые) версии, которые еще не удалены процедурой очистки (процессом автоочистки). Если в файлах таблицы остается много мертвых, но не вычищенных версий строк, это будет приводить к замедлению скорости работы последовательного сканирования. В частности поэтому своевременное выполнение очистки может существенно ускорить выполнение некоторых запросов.

Подробнее про версии строк, снимки данных и очистку ненужных версий строк рассказано в курсе DBA2.

<https://postgrespro.ru/docs/postgresql/16/mvcc>

<https://postgrespro.ru/docs/postgresql/16/sql-vacuum>

Последовательное сканирование

В плане выполнения запроса последовательное сканирование представлено узлом Seq Scan:

```
=> EXPLAIN (buffers) SELECT * FROM flights;
```

```
               QUERY PLAN
-----
Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=63)
Planning:
  Buffers: shared hit=91 read=17 dirtied=6
(3 rows)
```

В скобках приведены важные значения:

- cost — оценка стоимости;
- rows — оценка числа строк, возвращаемых операцией;
- width — оценка среднего размера одной строки 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 получения всех данных.

Как считается полная стоимость?

Оптимизатор PostgreSQL учитывает ввод-вывод и ресурсы процессора. Составляющая ввода-вывода рассчитывается как произведение числа страниц в таблице и условной стоимости 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)
```

Последовательное сканирование и мертвые версии строк

Создадим копию таблицы flights, отключив для нее автоочистку, чтобы мертвые версии строк не удалялись автоматически:

```
=> CREATE TABLE flights_copy
    WITH (autovacuum_enabled = false)
    AS SELECT * FROM flights;
```

```
SELECT 214867
```

И проверим размер полученной таблицы:

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

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

Удалим все строки из этой таблицы:

```
=> DELETE FROM flights_copy;
```

DELETE 214867

Мертвые версии строк останутся в файле таблицы до ближайшей очистки, поэтому размер таблицы останется прежним:

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

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

Получим план запроса с параметрами analyze и buffers — обратите внимание на количество страниц в Buffers и стоимость:

```
=> EXPLAIN (analyze, buffers, timing off)
SELECT count(*) FROM flights_copy;
```

QUERY PLAN

```
-----
Aggregate (cost=3968.80..3968.81 rows=1 width=8) (actual rows=1 loops=1)
  Buffers: shared hit=2624
  -> Seq Scan on flights_copy (cost=0.00..3699.84 rows=107584 width=0) (actual rows=0
loops=1)
    Buffers: shared hit=2624
Planning:
  Buffers: shared hit=3 read=3
Planning Time: 1.081 ms
Execution Time: 14.252 ms
(8 rows)
```

Прочитано более двух тысяч страниц.

Очистим таблицу вручную и снова проверим размер таблицы:

```
=> VACUUM flights_copy;
```

VACUUM

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

pg_size_pretty
-----
16 kB
(1 row)
```

Мертвые версии строк удалены, размер таблицы уменьшился. Снова получим план запроса:

```
=> EXPLAIN (analyze, buffers, timing off)
SELECT count(*) FROM flights_copy;
```

QUERY PLAN

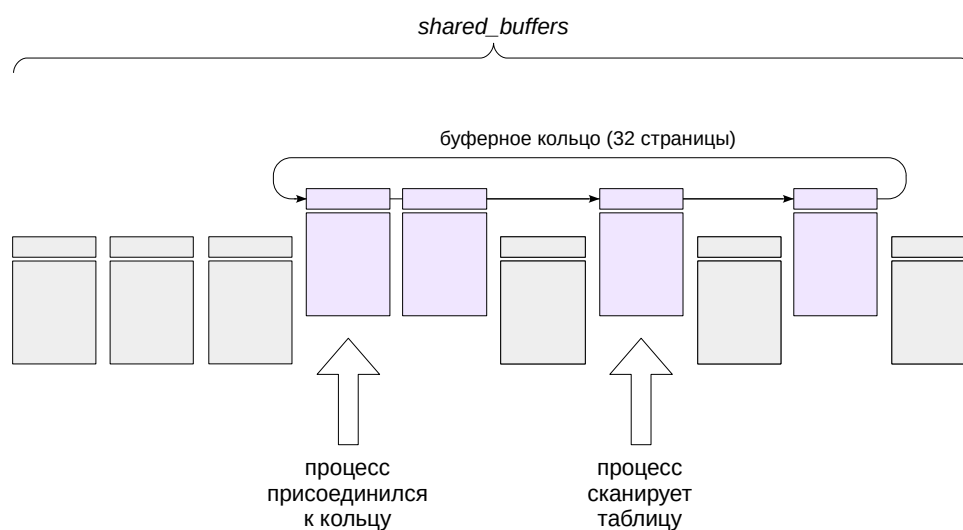
```
-----
Aggregate (cost=0.00..0.01 rows=1 width=8) (actual rows=1 loops=1)
  -> Seq Scan on flights_copy (cost=0.00..0.00 rows=1 width=0) (actual rows=0 loops=1)
Planning:
  Buffers: shared hit=2
Planning Time: 0.077 ms
Execution Time: 0.023 ms
(6 rows)
```

Таблица пуста — не прочитано ни одной версии строки, стоимость плана минимальна.

Время, затраченное на обработку неактуальных версий строк, может быть существенным для таблиц большего размера.

```
=> DROP TABLE flights_copy;
```

DROP TABLE



При последовательном сканировании через буферный кеш проходит большое количество страниц с «одноразовыми» данными, которые могут вытеснять полезные страницы из буферов. Чтобы этого не происходило, используются так называемые буферные кольца. Для последовательного чтения большой таблицы (размер которой превышает четверть буферного кеша) из всего кеша используются только 32 страницы, в пределах которых действует вытеснение. При этом остальные данные в буфере не страдают.

Если в процессе чтения меняются биты-подсказки или сами данные, появляется грязный буфер. Такой буфер отцепляется от кольца и будет вытеснен на общих основаниях, а в кольцо добавляется новый буфер. Такая стратегия рассчитана на то, что в основном данные читаются, а не меняются.

Если в процессе чтения таблицы другому процессу потребуется та же таблица, он не начинает читать ее с начала, а подключается к имеющемуся буферному кольцу. После окончания сканирования процесс дочитывает «пропущенное» начало таблицы.

При работе с временными таблицами через локальный кеш механизм буферных колец не используется.

Массовое вытеснение

Сбросим текущие значения статистики ввода-вывода:

```
=> SELECT pg_stat_reset_shared('io');
```

```
pg_stat_reset_shared
-----
```

(1 row)

Прочитаем все строки из таблицы, размер которой больше четверти shared_buffers:

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

QUERY PLAN

```
-----
Seq Scan on tickets (actual time=2.756..6116.525 rows=2949857 loops=1)
  Buffers: shared read=49415
Planning:
  Buffers: shared hit=52 read=6 dirtied=2
Planning Time: 13.467 ms
Execution Time: 6265.022 ms
(6 rows)
```

Прочитано 49415 страниц.

Накопленную статистику показывает представление pg_stat_io:

```
=> SELECT reads, hits, reuses
FROM pg_stat_io
WHERE context = 'bulkread' -- сканирование больших таблиц
AND object = 'relation'
AND backend_type = 'client backend';
```

```
reads | hits | reuses
-----+-----+-----
49415 |    0 | 49383
(1 row)
```

Всего было прочитано (reads) 49415 страниц, из них (reuses) 49383 страниц были загружены в кеш с вытеснением. Таким образом, для сканирования использовалось буферное кольцо размером $49415 - 49383 = 32$ страницы.

Снова сбросим статистику и повторно обратимся к той же таблице:

```
=> SELECT pg_stat_reset_shared('io');
```

```
pg_stat_reset_shared
-----
```

(1 row)

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

QUERY PLAN

```
-----
Seq Scan on tickets (actual time=0.798..921.254 rows=2949857 loops=1)
  Buffers: shared hit=32 read=49383
Planning Time: 0.050 ms
Execution Time: 1019.984 ms
(4 rows)
```

```
=> SELECT reads, hits, reuses
FROM pg_stat_io
WHERE context = 'bulkread' -- сканирование больших таблиц
AND object = 'relation'
AND backend_type = 'client backend';
```

reads	hits	reuses
49383	32	49351

(1 row)

Теперь видим, что 32 страницы уже находились в кеше после прошлого чтения (hits), так что серверу пришлось прочитать оставшиеся 49383 страниц (reads), и при этом из кеша вытеснилось 49351 страниц (reuses). Снова размер кольца составляет $49383 - 49351 = 32$ буферов, но в них уже находились другие страницы, а не те, что использовались в первый раз.

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 поиска автоматически отсортированы

Использование

ускорение доступа
поддержка ограничений целостности

В этой теме мы будем рассматривать только один из доступных в PostgreSQL типов индексов: В-дерево (B-tree). Это наиболее часто применяющийся на практике тип индекса. Некоторые другие типы индексов рассмотрены далее в этом курсе в теме «Типы индексов».

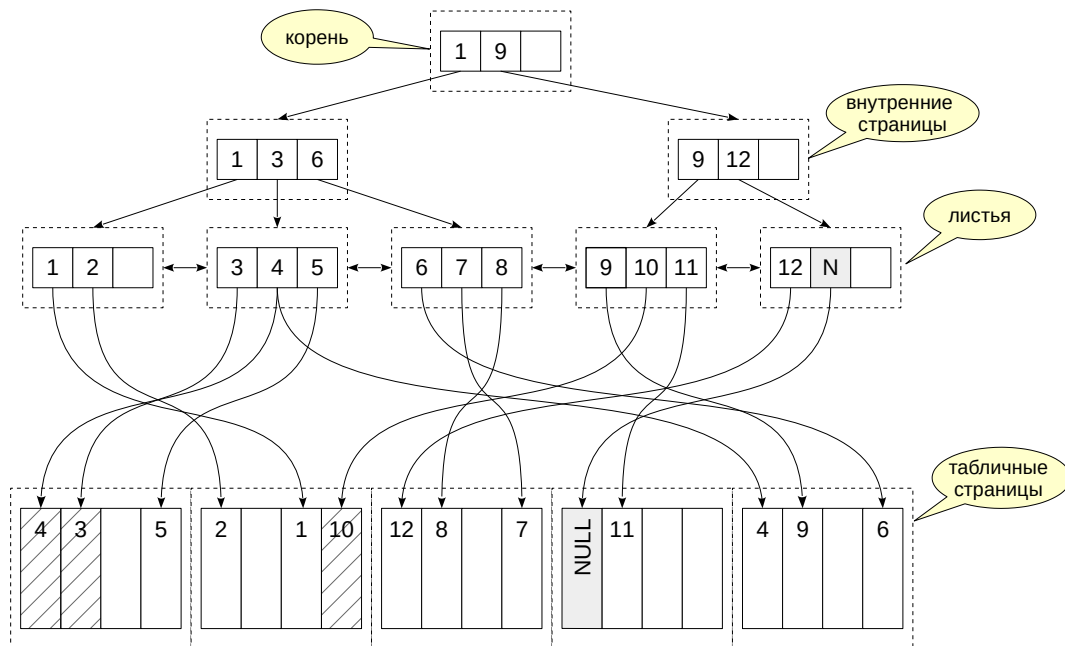
Как и все индексы в PostgreSQL, В-дерево является вторичной структурой — индекс не содержит в себе никакой информации, которую нельзя было бы получить из самой таблицы, однако занимает дополнительное место на диске. Индекс можно удалить и пересоздать. Индексы нужны для ускорения операций, затрагивающих небольшую часть таблицы: выборки малого количества строк, а также поддержки ограничений целостности (первичных и уникальных ключей).

Индексы сопоставляют 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 ссылки на версии строк.

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). При индексации текстовых строк нужно учитывать особенности правил сортировки (подробнее см. курс DBA2, тема «Локализация»).

Особенностями В-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.

В-дерево



8

В верхней части слайда приведен пример В-дерева. Его страницы состоят из индексных записей, каждая из которых содержит:

- *ключ* – значения столбцов, по которым построен индекс (такие столбцы называются *ключевыми*);
- ссылку на другую страницу индекса или ссылки на версии строк.

Внутри страницы ключи всегда упорядочены.

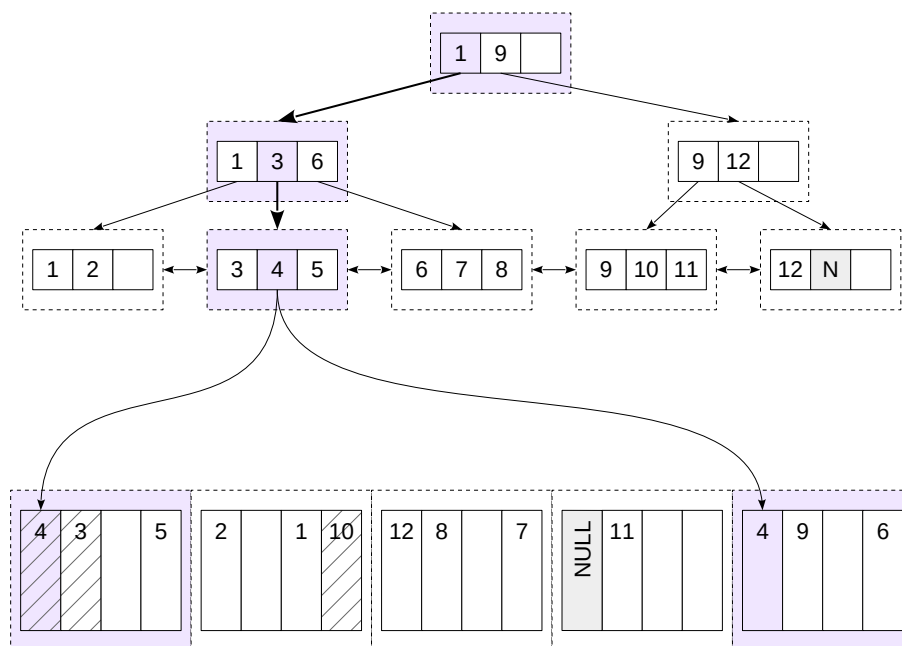
Листовые страницы ссылаются непосредственно на табличные версии строк, содержащие ключи индексирования. Эти страницы связаны двунаправленным списком, чтобы было удобно перебирать ключи в порядке возрастания или убывания.

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

Верхняя страница дерева, на которую нет ссылок, называется *корнем*.

Индексная страница может быть заполнена частично. Свободное место используется для вставки в индекс новых записей. 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.

По умолчанию считается, что неопределенные значения «больше» обычных, и они хранятся в дереве справа. Этот порядок можно изменить при создании индекса с помощью предложений `NULLS LAST` и `NULLS FIRST`.



Рассмотрим, как происходит поиск одного значения с помощью индекса. Например, нам нужно найти в таблице строку, в которой 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 записи 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.

Заметьте, что ключи могут повторяться, причем даже в уникальном индексе: из-за работы механизма многоверсионности могут возникать разные версии одной и той же строки. Для экономии места ключи хранятся в индексной странице в одном экземпляре.

Прежде чем вернуть найденные версии строк, необходимо проверить их видимость.

На иллюстрациях цветом выделены записи и 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 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, модель рассчитывает на то, что 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.

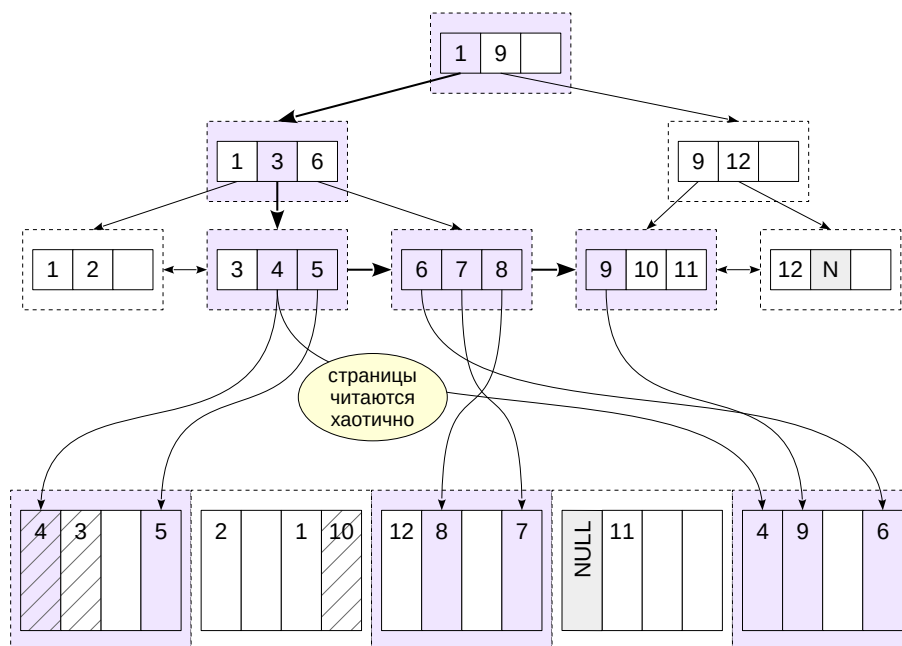
В строке `Index Cond` плана указываются только те условия, по которым происходит обращение к индексу или которые могут быть проверены на уровне индекса.

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



11

В-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 записи 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. Результат поиска автоматически получается отсортированным.

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

```
=> EXPLAIN (costs off)
SELECT * FROM bookings
WHERE book_ref > '000900' AND book_ref < '000939'
ORDER BY book_ref;
```

QUERY PLAN

```
-----
Index Scan using bookings_pkey on bookings
  Index Cond: ((book_ref > '000900'::bpchar) AND (book_ref < '000939'::bpchar))
(2 rows)
```

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

```
=> 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=4 read=1
Planning:
  Buffers: shared hit=12
(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.

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:

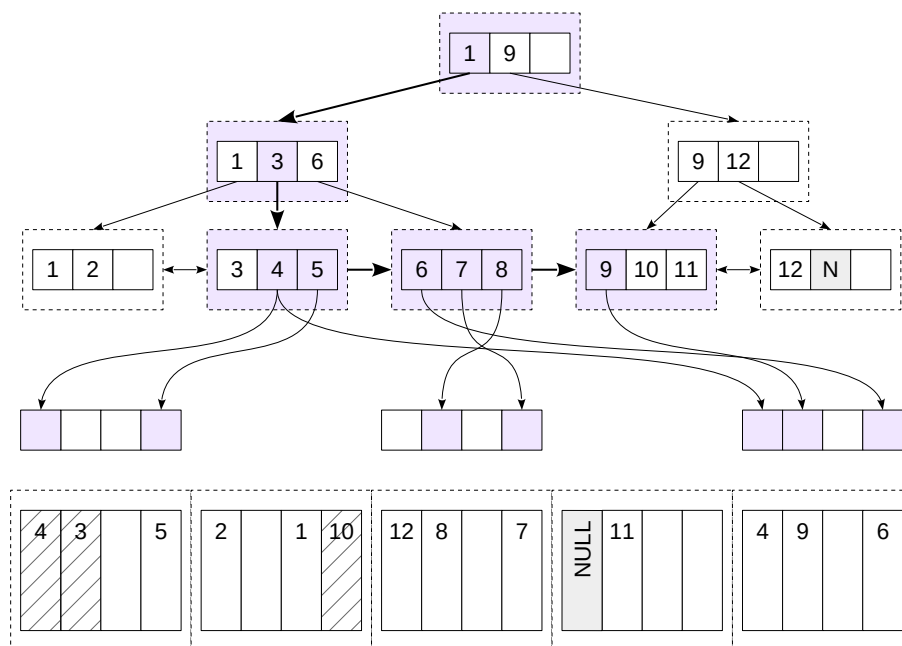
```
=> EXPLAIN (analyze, buffers, costs off)
SELECT * FROM bookings
WHERE book_ref IN ('000906', '000909', '000917', '000930', '000938')
ORDER BY book_ref DESC;
```

QUERY PLAN

```
-----
Index Scan Backward using bookings_pkey on bookings (actual time=0.028..0.044 rows=5
loops=1)
  Index Cond: (book_ref = ANY ('{000906,000909,000917,000930,000938}'::bpchar[]))
  Buffers: shared hit=24
Planning Time: 0.078 ms
Execution Time: 0.054 ms
(5 rows)
```

Количество страниц увеличилось, поскольку в этом случае приходится спускаться от корня к каждому значению. В версии 17 этот запрос выполняется так же эффективно, как и поиск по диапазону.

Bitmap Index Scan



13

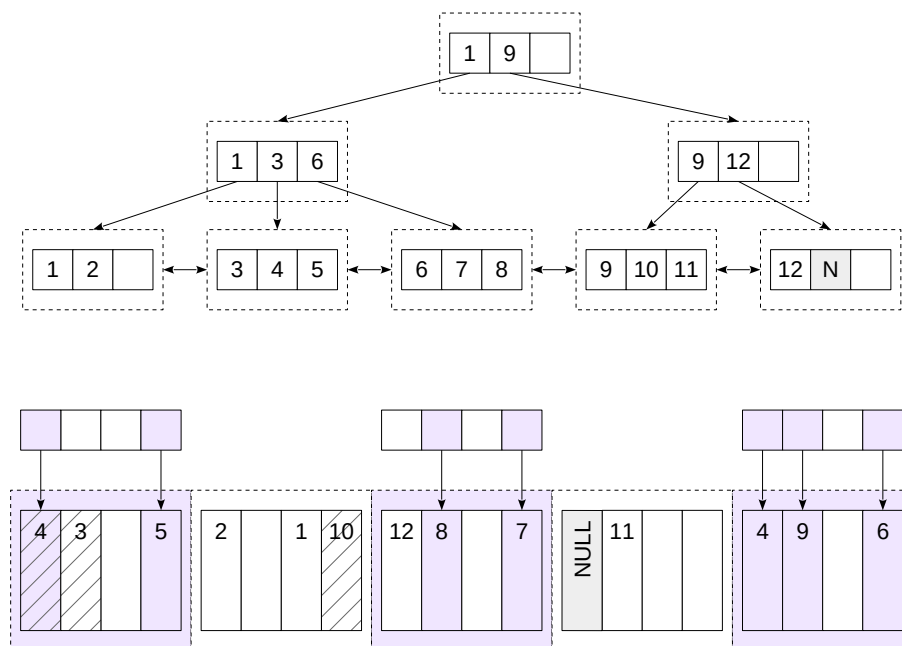
Множественный просмотр одних и тех же табличных страниц крайне неэффективен. Даже в лучшем случае, если нужная страница находится в буферном кеше, ее нужно найти и заблокировать (см. курс DBA2: тема «Буферный кеш» модуля «Журналирование»), а в худшем приходится иметь дело со случайными чтениями с диска.

Чтобы не тратить ресурсы на повторный просмотр табличных страниц, применяется еще один способ доступа — сканирование по битовой карте. Он похож на обычный индексный доступ, но происходит в два этапа.

Сначала сканируется индекс (Bitmap Index Scan) и в локальной памяти процесса строится битовая карта. Битовая карта состоит из фрагментов. Фрагменты соответствуют табличным страницам, а каждый бит фрагмента соответствует версии строки в этой странице. При построении битовой карты в ней отмечаются те версии строк, которые удовлетворяют условию и должны быть прочитаны.

За счет разделения битовой карты на фрагменты карта, в которой отмечено немного версий, будет занимать мало места.

Bitmap Heap Scan



14

Когда индекс просканирован и битовая карта готова, начинается сканирование таблицы (Bitmap Heap Scan). При этом:

- используется собственный механизм предвыборки, при котором асинхронно читаются *effective_io_concurrency* страниц (по умолчанию значение равно единице);
- на одной странице может проверяться несколько версий строк, но каждая страница просматривается ровно один раз.

Сканирование по битовой карте

Будем рассматривать таблицу бронирований bookings. Создадим на ней два дополнительных индекса:

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

```
CREATE INDEX
```

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

```
CREATE INDEX
```

Посмотрим, какой метод доступа будет выбран для поиска диапазона.

```
=> EXPLAIN  
SELECT * FROM bookings WHERE total_amount < 5000;
```

QUERY PLAN

```
-----  
--  
Bitmap Heap Scan on bookings  (cost=80.77..8966.56 rows=4173 width=21)  
  Recheck Cond: (total_amount < '5000'::numeric)  
    -> Bitmap Index Scan on bookings_total_amount_idx  (cost=0.00..79.73 rows=4173  
width=0)  
      Index Cond: (total_amount < '5000'::numeric)  
(4 rows)
```

Чтобы упорядочить сканирование большого числа страниц, планировщик выбрал сканирование по битовой карте. Этот метод состоит из двух узлов:

- Bitmap Index Scan читает индекс и строит битовую карту;
- Bitmap Heap Scan читает табличные страницы, используя построенную карту.

Обратите внимание на стоимости: карта должна быть построена полностью, прежде чем ее можно будет использовать.

Объединение битовых карт

Кроме того, что битовая карта позволяет избежать повторных чтений табличных страниц, с ее помощью можно объединять несколько условий:

```
=> EXPLAIN (costs off)  
SELECT * FROM bookings  
WHERE total_amount < 5000 OR total_amount > 500000;
```

QUERY PLAN

```
-----  
Bitmap Heap Scan on bookings  
  Recheck Cond: ((total_amount < '5000'::numeric) OR (total_amount > '500000'::numeric))  
    -> BitmapOr  
      -> Bitmap Index Scan on bookings_total_amount_idx  
          Index Cond: (total_amount < '5000'::numeric)  
      -> Bitmap Index Scan on bookings_total_amount_idx  
          Index Cond: (total_amount > '500000'::numeric)  
(7 rows)
```

Здесь сначала были построены две битовые карты — по одной на каждое условие, а затем объединены побитовой операцией «или».

Таким же образом могут быть использованы и разные индексы:

```
=> EXPLAIN (costs off)  
SELECT * FROM bookings  
WHERE total_amount < 5000  
   OR book_date = bookings.now() - INTERVAL '1 day';
```

QUERY PLAN

```
-----  
-----  
Bitmap Heap Scan on bookings  
  Recheck Cond: ((total_amount < '5000'::numeric) OR (book_date = ('2017-08-15  
18:00:00+03'::timestamp with time zone - '1 day'::interval)))  
    -> BitmapOr  
      -> Bitmap Index Scan on bookings_total_amount_idx  
            Index Cond: (total_amount < '5000'::numeric)  
      -> Bitmap Index Scan on bookings_book_date_idx  
            Index Cond: (book_date = ('2017-08-15 18:00:00+03'::timestamp with time  
zone - '1 day'::interval))  
(7 rows)
```

Битовая карта без потери точности

пока размер карты не превышает *work_mem*,
информация хранится с точностью до версии строки

Битовая карта с потерей точности

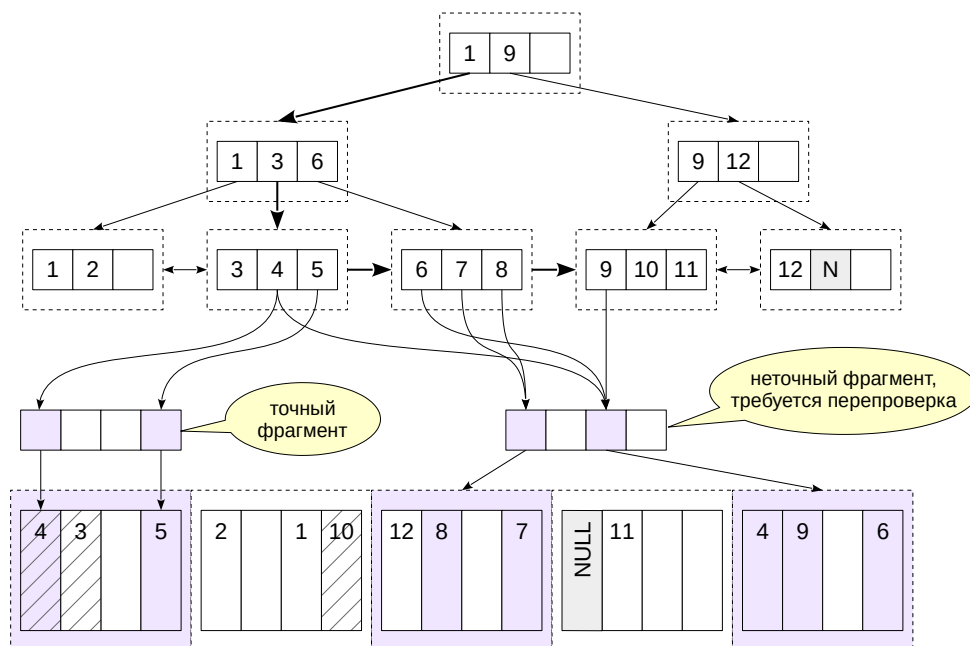
если память закончилась, происходит огрубление
части уже построенной карты до отдельных страниц
требуется примерно 1 МБ памяти на 64 ГБ данных;
ограничение *work_mem* может быть превышено

Битовая карта хранится в локальной памяти обслуживающего процесса, и под ее хранение выделяется *work_mem* байт. Временные файлы никогда не используются.

Если карта перестает помещаться в *work_mem*, часть ее фрагментов «огрубляется» — каждый бит начинает соответствовать целой странице, а не отдельной версии строки (lossy bitmap). Стоимость обработки таких фрагментов увеличивается. Освободившееся место используется для того, чтобы продолжить строить карту.

В принципе, при сильно ограниченном *work_mem* и большой выборке битовая карта может не поместиться в памяти, даже если в ней совсем не останется информации на уровне версий строк. В таком случае ограничение *work_mem* нарушается — под карту будет дополнительно выделено столько памяти, сколько необходимо.

Неточные фрагменты



17

На рисунке показана битовая карта, состоящая из двух фрагментов. Первый фрагмент — точный, каждый его бит соответствует одной версии строки. Второй фрагмент — неточный, в нем каждый бит соответствует целой странице.

Неточные фрагменты требуют перепроверки условий для всех версий строк в табличной странице, а это сказывается на производительности. Если две битовые карты объединяются, причем фрагмент хотя бы одной из карт неточен, то и результирующий фрагмент тоже вынужденно будет неточным. Поэтому размер *work_mem* играет большую роль для эффективности сканирования по битовой карте.

Неточные фрагменты

Узел Bitmap Heap Scan показывает условие перепроверки (Recheck Cond). Сама же перепроверка выполняется не всегда, а только при потере точности, когда битовая карта не помещается в память.

Повторим запрос, изменив условие: будем искать бронирования с суммой до 5 тысяч ₽, сделанные за последний месяц.

```
=> 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 (analyze, costs off, timing off)
SELECT count(*) FROM bookings
WHERE total_amount < 5000 AND book_date > $1;
```

QUERY PLAN

```
-----
Aggregate (actual rows=1 loops=1)
  -> Bitmap Heap Scan on bookings (actual rows=129 loops=1)
        Recheck Cond: ((total_amount < '5000'::numeric) AND (book_date > '2017-07-15
18:00:00+03'::timestamp with time zone))
        Heap Blocks: exact=129
  -> BitmapAnd (actual rows=0 loops=1)
        -> Bitmap Index Scan on bookings_total_amount_idx (actual rows=1471
loops=1)
                Index Cond: (total_amount < '5000'::numeric)
        -> Bitmap Index Scan on bookings_book_date_idx (actual rows=178142
loops=1)
                Index Cond: (book_date > '2017-07-15 18:00:00+03'::timestamp with
time zone)
Planning Time: 0.154 ms
Execution Time: 44.765 ms
(11 rows)
```

Строка «Heap Blocks: exact» говорит о том, что все фрагменты битовой карты построены с точностью до строк — перепроверка не выполняется.

Уменьшим размер выделяемой памяти.

```
=> SET work_mem = '64kB';
```

```
SET
```

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

```
=> EXPLAIN (analyze, costs off, timing off)
SELECT count(*) FROM bookings
WHERE total_amount < 5000 AND book_date > $1;
```

QUERY PLAN

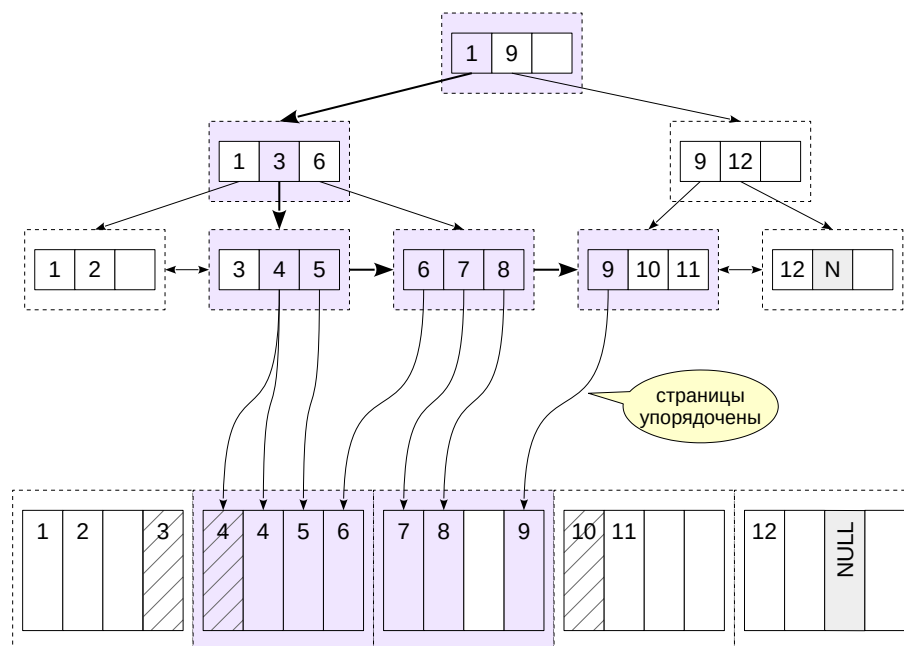
```
-----  
Aggregate (actual rows=1 loops=1)  
  -> Bitmap Heap Scan on bookings (actual rows=129 loops=1)  
        Recheck Cond: ((total_amount < '5000'::numeric) AND (book_date > '2017-07-15  
18:00:00+03'::timestamp with time zone))  
        Rows Removed by Index Recheck: 89895  
        Heap Blocks: exact=811 lossy=568  
        -> BitmapAnd (actual rows=0 loops=1)  
              -> Bitmap Index Scan on bookings_total_amount_idx (actual rows=1471  
loops=1)  
                    Index Cond: (total_amount < '5000'::numeric)  
              -> Bitmap Index Scan on bookings_book_date_idx (actual rows=178142  
loops=1)  
                    Index Cond: (book_date > '2017-07-15 18:00:00+03'::timestamp with  
time zone)  
        Planning Time: 0.148 ms  
        Execution Time: 27.004 ms  
(12 rows)
```

Здесь появились lossy-фрагменты битовой карты — с точностью до страниц. Также указано, сколько строк не прошло перепроверку условия (Rows Removed by Index Recheck).

Восстановим значение параметра.

```
=> RESET work_mem;
```

```
RESET
```



Если данные в таблице физически упорядочены, обычное индексное сканирование не будет читать табличную страницу повторно. В таком (нечастом на практике) случае метод сканирования по битовой карте теряет смысл, проигрывая обычному индексному сканированию.

Разумеется, планировщик это тоже учитывает (как именно — рассматривается в теме «Базовая статистика»).

Кластеризация

Если строки таблицы упорядочены так же, как и индекс, битовая карта становится излишней. Продемонстрируем это с помощью команды CLUSTER.

Сейчас строки таблицы физически упорядочены по номеру бронирования:

```
=> SELECT * FROM bookings LIMIT 10;
```

book_ref	book_date	total_amount
000004	2016-08-13 15:40:00+03	55800.00
00000F	2017-07-05 03:12:00+03	265700.00
000010	2017-01-08 19:45:00+03	50900.00
000012	2017-07-14 09:02:00+03	37900.00
000026	2016-08-30 11:08:00+03	95600.00
00002D	2017-05-20 18:45:00+03	114700.00
000034	2016-08-08 05:46:00+03	49100.00
00003F	2016-12-12 15:02:00+03	109800.00
000048	2016-09-17 01:57:00+03	92400.00
00004A	2016-10-13 21:57:00+03	29000.00

(10 rows)

Переупорядочим строки в соответствии с индексом по столбцу total_amount.

Пока идет процесс, обратите внимание:

- Команда CLUSTER устанавливает исключительную блокировку, поскольку полностью перестраивает таблицу (как VACUUM FULL);
- Строки упорядочиваются, но не поддерживаются в упорядоченном виде — в процессе работы кластеризация будет ухудшаться.

```
=> CLUSTER bookings USING bookings_total_amount_idx;
```

CLUSTER

```
=> VACUUM ANALYZE bookings;
```

VACUUM

Убедимся, что строки упорядочены по стоимости:

```
=> SELECT * FROM bookings LIMIT 10;
```

book_ref	book_date	total_amount
00F39E	2017-02-12 04:11:00+03	3400.00
0103E1	2017-04-03 09:32:00+03	3400.00
013695	2016-11-01 09:30:00+03	3400.00
0158C0	2017-04-24 07:47:00+03	3400.00
01AD57	2017-03-15 16:11:00+03	3400.00
020E97	2017-01-02 12:25:00+03	3400.00
021FD3	2017-05-27 10:20:00+03	3400.00
0278C7	2017-02-04 17:42:00+03	3400.00
029452	2016-12-10 13:51:00+03	3400.00
0355DD	2016-09-19 13:22:00+03	3400.00

(10 rows)

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

```
SELECT * FROM bookings
```

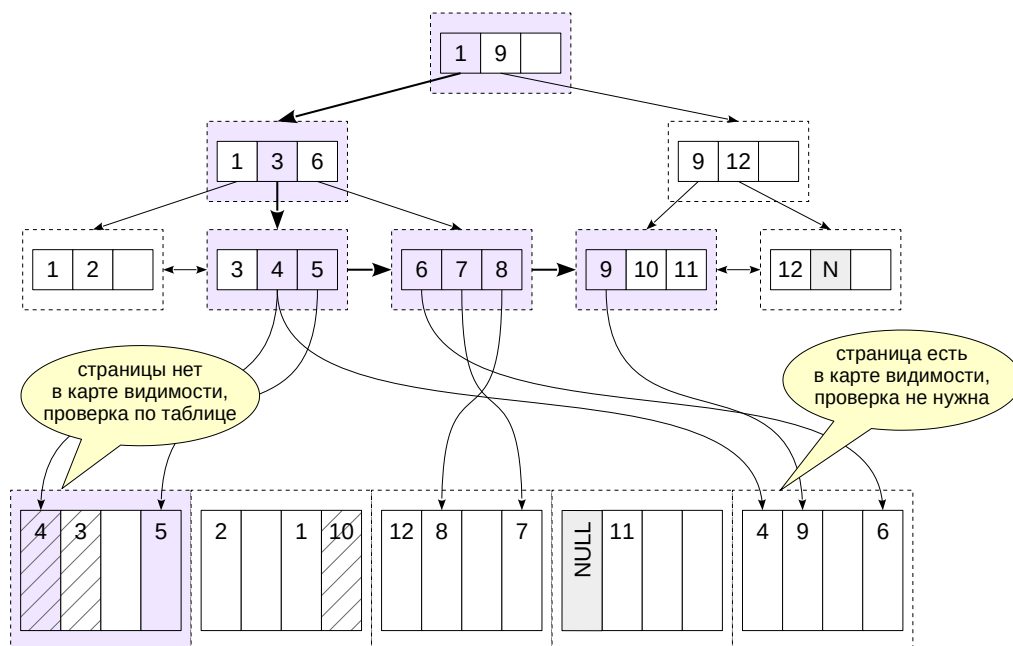
```
WHERE total_amount < 5000;
```

QUERY PLAN

Index Scan using bookings_total_amount_idx on bookings
Index Cond: (total_amount < '5000'::numeric)
(2 rows)

До кластеризации использовалось сканирование по битовой карте, но теперь проще и выгодней сделать обычное индексное сканирование.

Index-Only Scan



21

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

Это хорошая оптимизация, исключая 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 которым стоит выполнять очистку 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 в листовом узле сначала проверяется наличие табличной страницы в карте видимости, а при ее отсутствии читается сама табличная страница.

<https://postgrespro.ru/docs/postgresql/16/indexes-index-only-scans>

Сканирование только индекса

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

```
=> EXPLAIN (costs off)
SELECT total_amount FROM bookings
WHERE total_amount > 200000;
```

QUERY PLAN

```
-----
Index Only Scan using bookings_total_amount_idx on bookings
    Index Cond: (total_amount > '200000'::numeric)
(2 rows)
```

Посмотрим план этого запроса с помощью EXPLAIN ANALYZE:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT total_amount FROM bookings
WHERE total_amount > 200000;
```

QUERY PLAN

```
-----
Index Only Scan using bookings_total_amount_idx on bookings (actual rows=141535 loops=1)
    Index Cond: (total_amount > '200000'::numeric)
    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 = '00F39E';
```

UPDATE 1

Сколько версий строк придется проверить теперь?

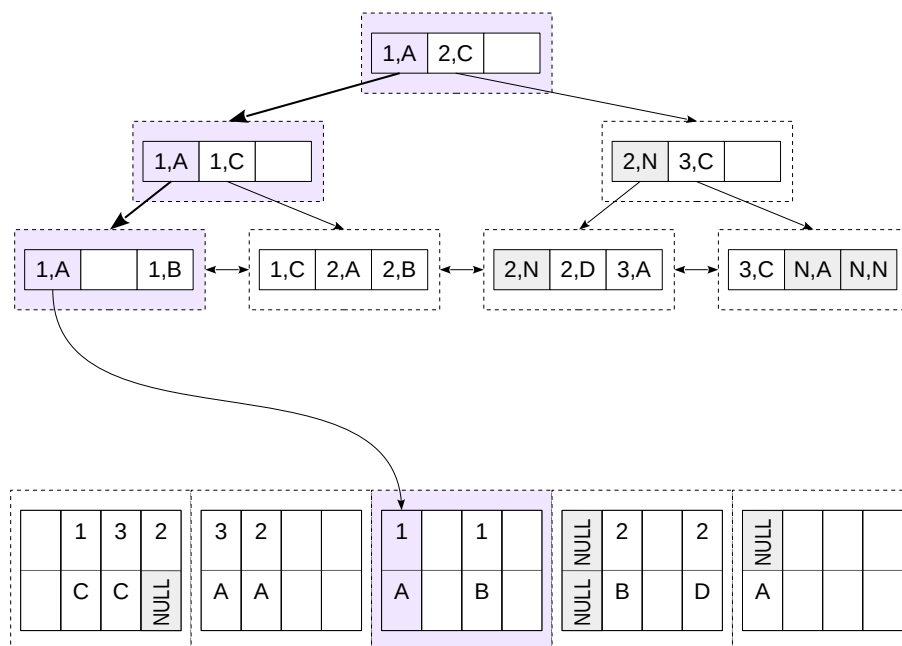
```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT total_amount FROM bookings
WHERE total_amount > 200000;
```

QUERY PLAN

```
-----
Index Only Scan using bookings_total_amount_idx on bookings (actual rows=141535 loops=1)
    Index Cond: (total_amount > '200000'::numeric)
    Heap Fetches: 88
(3 rows)
```

Проверять пришлось все версии, попавшие на измененную страницу.

Многоколоночный индекс

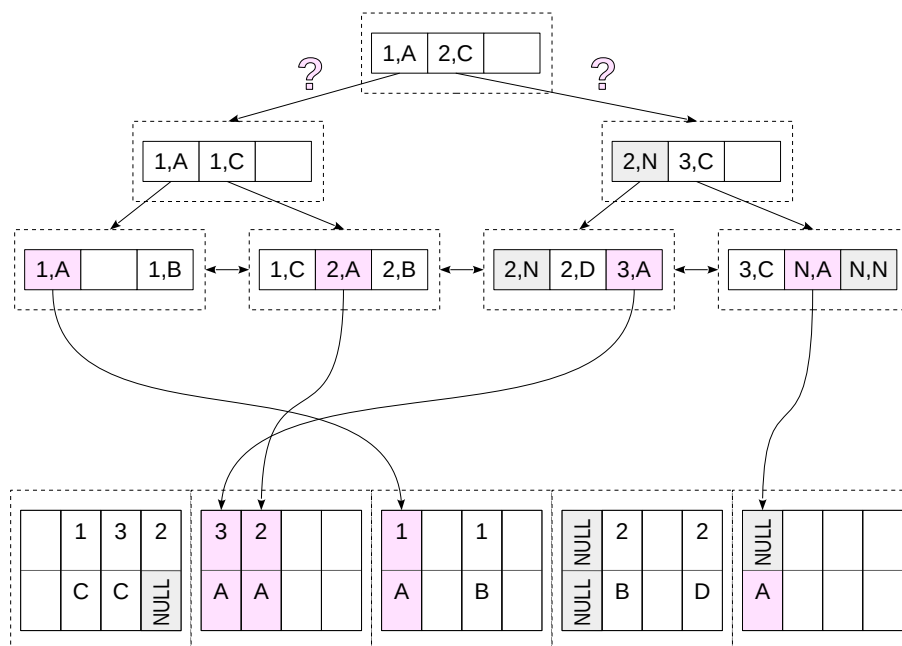


23

Индекс можно создать по нескольким столбцам. В этом случае имеет значение порядок следования столбцов и порядок сортировки.

На рисунке приведен пример многоколоночного индекса, построенного по двум столбцам (оба — по возрастанию значений). Такой индекс ускоряет поиск, если в запросе есть условие на один или несколько *первых* ключей, поскольку в индексных записях ключи отсортированы сначала по первому столбцу, затем по второму и так далее.

В примере, показанном на слайде, поиск происходит по первому и второму столбцам; индекс подошел бы и для условия только на первый столбец.



Однако если запрос содержит условие только на второй столбец, индекс оказывается бесполезным. На слайде видно, что листовые записи, в которых второй столбец равен «A», могут оказаться в любом месте индекса — у нас нет способа спуститься к ним от корня дерева.

Иногда в таких случаях планировщик все-таки прибегает к только индексному сканированию, но при этом просматривается весь индекс целиком.

Аналогично, индекс не может выдавать записи в порядке, отличном от указанного при создании индекса. Например, показанный на слайде индекс не может выдавать записи, отсортированные по первому столбцу в порядке возрастания, а по второму — в порядке убывания.

Многоколоночные индексы

На таблице перелетов ticket_flights создан индекс по столбцам ticket_no, flight_id. Запрос по обоим столбцам использует индекс:

```
=> EXPLAIN SELECT *
FROM ticket_flights
WHERE ticket_no = '0005432000284' AND flight_id = 187662;
```

QUERY PLAN

```
-----
-
Index Scan using ticket_flights_pkey on ticket_flights (cost=0.56..8.58 rows=1 width=32)
  Index Cond: ((ticket_no = '0005432000284'::bpchar) AND (flight_id = 187662))
(2 rows)
```

Запрос только по номеру билета — тоже:

```
=> EXPLAIN SELECT *
FROM ticket_flights
WHERE ticket_no = '0005432000284';
```

QUERY PLAN

```
-----
--
Index Scan using ticket_flights_pkey on ticket_flights (cost=0.56..16.57 rows=3
width=32)
  Index Cond: (ticket_no = '0005432000284'::bpchar)
(2 rows)
```

Но для запроса по номеру рейса такой индекс не годится:

```
=> EXPLAIN SELECT *
FROM ticket_flights
WHERE flight_id = 187662;
```

QUERY PLAN

```
-----
Gather (cost=1000.00..114680.93 rows=105 width=32)
  Workers Planned: 2
    -> Parallel Seq Scan on ticket_flights (cost=0.00..113670.43 rows=44 width=32)
        Filter: (flight_id = 187662)
(4 rows)
```

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

26

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 можно добавить к индексу *неключевые столбцы*, указав их в предложении INCLUDE.

<https://postgrespro.ru/docs/postgresql/16/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 быть получены без обращения к таблице.

В настоящее время include-индексы поддерживаются только для индексов на основе B-деревьев, GiST и SP-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 требуется вернуть не только столбец ticket_no, который есть в индексе, но и book_ref, которого в индексе нет:

```
=> 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=0.632..17.006 rows=7146 loops=1)
  Index Cond: (ticket_no > '0005435990286'::bpchar)
  Buffers: shared hit=74 read=153
Planning:
  Buffers: shared read=4
(5 rows)
```

Buffers показывает количество прочитанных страниц (hit+read).

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:

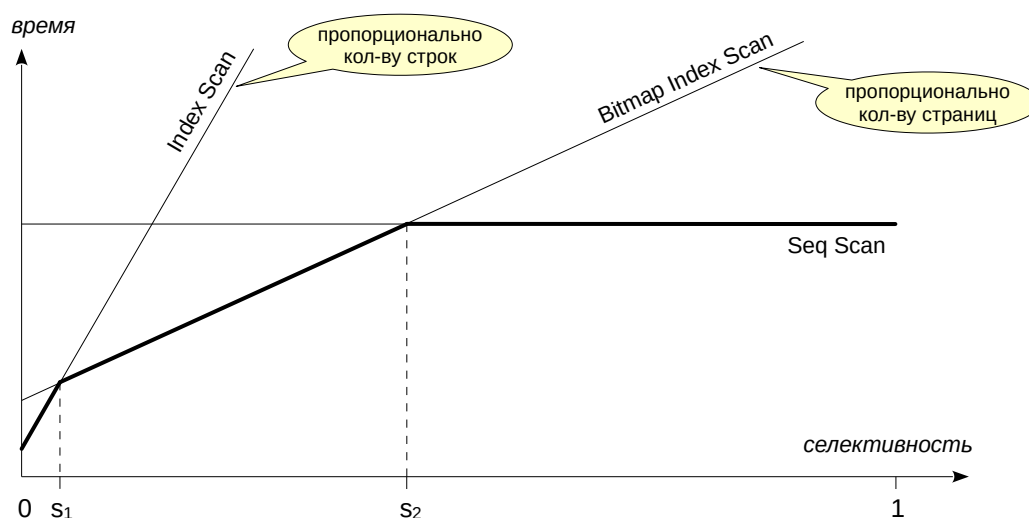
```
=> EXPLAIN (analyze, buffers, costs off, summary off)
SELECT ticket_no, book_ref FROM tickets
WHERE ticket_no > '0005435990286';
```

QUERY PLAN

```
-----
Index Only Scan using tickets_ticket_no_book_ref_idx on tickets (actual
time=0.066..2.624 rows=7146 loops=1)
  Index Cond: (ticket_no > '0005435990286'::bpchar)
  Heap Fetches: 0
  Buffers: shared hit=4 read=35
Planning:
  Buffers: shared hit=20 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).



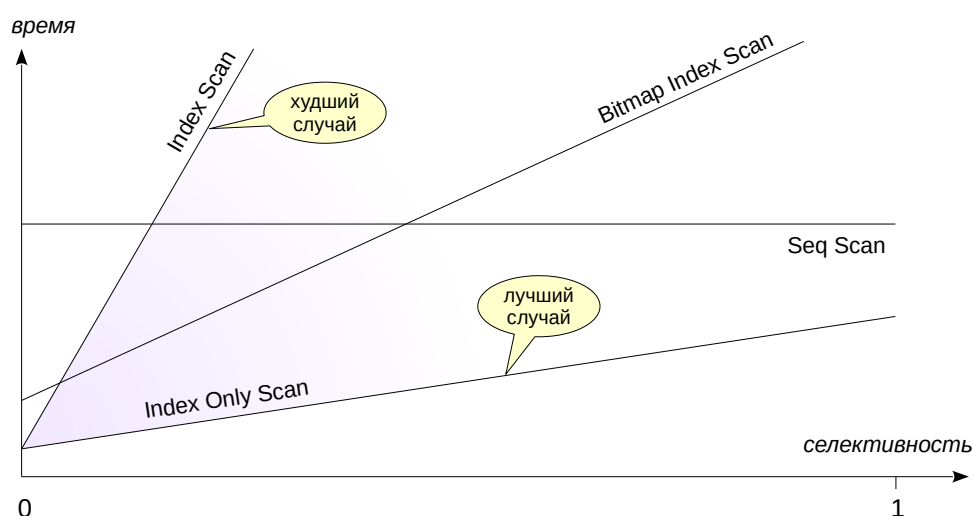
Индексное сканирование лучше всего работает при очень высокой селективности, когда по индексу выбирается одно или несколько значений.

При средней селективности обычно лучше всего показывает себя сканирование по битовой карте. Хотя этот способ требует предварительного построения битовой карты, он выигрывает у индексного сканирования, поскольку обходится без повторных чтений одних и тех же страниц (если только данные в таблице не расположены физически в нужном порядке, что бывает нечасто). В худшем случае производительность индексного доступа пропорциональна числу выбираемых строк, а сканирования по битовой карте — числу страниц.

При низкой селективности лучше всего работает последовательное сканирование: если надо выбрать все или почти все табличные строки, обращение к индексным страницам только увеличивает накладные расходы. Этот эффект усиливается в случае вращающихся дисков, где скорость произвольного чтения существенно ниже скорости чтения последовательно расположенных страниц.

Значение селективности, при котором становится выгодно переключиться на другой метод доступа, сильно зависит от конкретной таблицы и конкретного индекса. Планировщик учитывает множество параметров, чтобы выбрать наиболее подходящий способ.

Еще одно замечание: при индексном доступе результат возвращается отсортированным, что в ряде случаев увеличивает привлекательность такого доступа даже при низкой селективности.



Эффективность сканирования только индекса сильно зависит от актуальности карты видимости и от того, сколько страниц действительно содержат только актуальные версии строк.

В лучшем случае этот метод доступа может быть эффективнее последовательного сканирования даже при низкой селективности (если индекс меньше, чем таблица, и особенно на SSD-дисках).

В худшем случае, когда требуется проверять видимость каждой строки, метод вырождается в обычное индексное сканирование.

Поэтому планировщику приходится учитывать состояние карты видимости: при неблагоприятном прогнозе исключительно индексное сканирование не применяется из-за опасности получить замедление вместо ускорения.

Оптимизатор располагает несколькими методами доступа

- последовательное сканирование
- сканирование индекса
- сканирование только индекса
- сканирование по битовой карте

Модель стоимости учитывает множество параметров

1. Убедитесь, что при повторном выполнении запрос, выбирающий все строки из таблицы `flights`, читает данные из кеша СУБД.
2. В демонстрации был создан `include`-индекс для таблицы билетов `tickets`. Замените им индекс, поддерживающий первичный ключ.
3. Создайте индекс по столбцу `amount` таблицы перелетов `ticket_flights`. Найдите перелеты стоимостью более 120 000 руб. (примерно 1 % строк). Какой метод доступа был выбран?
4. Повторите п. 3 для стоимости менее 42 000 руб. (около 90 % строк).

1. Используйте команду `EXPLAIN` с параметрами `analyze` и `buffers`.
4. Попробуйте также запретить выбранный метод доступа (параметр `enable_seqscan`) и сравнить скорости выполнения.

1. Запрос всех строк таблицы flights

```
=> EXPLAIN (analyze, buffers)
SELECT * FROM flights;
```

QUERY PLAN

```
-----
Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=63) (actual
time=0.018..49.701 rows=214867 loops=1)
  Buffers: shared read=2624
Planning:
  Buffers: shared hit=91 read=17 dirtied=6
Planning Time: 2.170 ms
Execution Time: 57.372 ms
(6 rows)
```

Все страницы прочитаны с диска в общую память (Buffers: shared read=2624). Возможно, часть страниц была получена из кеша операционной системы, но PostgreSQL об этом ничего не знает.

Повторим запрос:

```
=> EXPLAIN (analyze, buffers)
SELECT * FROM flights;
```

QUERY PLAN

```
-----
Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=63) (actual
time=0.006..12.440 rows=214867 loops=1)
  Buffers: shared hit=2624
Planning Time: 0.049 ms
Execution Time: 21.487 ms
(4 rows)
```

Так как мы только что обращались к таблице, в буферном кеше сохранились ее страницы. Поэтому сервер читает данные из буферного кеша общей памяти (Buffers: shared hit=2624).

2. Include-индекс для первичного ключа

Чтобы изменения выполнялись атомарно, сделаем их в транзакции.

```
=> BEGIN;
```

```
BEGIN
```

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 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:

```
=> 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
```

```
=> \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)

Вернемся к исходному состоянию:

=> **ROLLBACK;**

ROLLBACK

3. Выборка 1% строк

=> **CREATE INDEX ON ticket_flights(amount);**

CREATE INDEX

=> **EXPLAIN (analyze)**

SELECT * FROM ticket_flights WHERE amount > 120000;

QUERY PLAN

```

-----
Bitmap Heap Scan on ticket_flights  (cost=1669.13..76466.99 rows=88864 width=32) (actual
time=14.753..519.338 rows=83988 loops=1)
  Recheck Cond: (amount > '120000'::numeric)
  Heap Blocks: exact=3394
   -> Bitmap Index Scan on ticket_flights_amount_idx  (cost=0.00..1646.91 rows=88864
width=0) (actual time=13.538..13.539 rows=83988 loops=1)
     Index Cond: (amount > '120000'::numeric)
  Planning Time: 8.094 ms
  Execution Time: 523.765 ms
(7 rows)

```

Выбрано сканирование по битовой карте, она уместилась в оперативную память без потери точности.

Для небольшой выборки сканирование по битовой карте эффективнее, чем последовательное.

4. Выборка 90% строк

=> **EXPLAIN (analyze)**

SELECT * FROM ticket_flights WHERE amount < 42000;

QUERY PLAN

```

-----
Seq Scan on ticket_flights  (cost=0.00..174858.15 rows=7513553 width=32) (actual
time=11.054..2234.824 rows=7540479 loops=1)
  Filter: (amount < '42000'::numeric)
  Rows Removed by Filter: 851373
  Planning Time: 0.091 ms
  Execution Time: 2491.209 ms
(5 rows)

```

Запретим последовательное сканирование.

=> **SET enable_seqscan = off;**

SET

=> **EXPLAIN (analyze)**

SELECT * FROM ticket_flights WHERE amount < 42000;

QUERY PLAN

```
-----  
-----  
Bitmap Heap Scan on ticket_flights (cost=140762.47..310478.38 rows=7513553 width=32)  
(actual time=936.871..11134.269 rows=7540479 loops=1)  
  Recheck Cond: (amount < '42000'::numeric)  
  Rows Removed by Index Recheck: 298921  
  Heap Blocks: exact=36281 lossy=33034  
-> Bitmap Index Scan on ticket_flights_amount_idx (cost=0.00..138884.08 rows=7513553  
width=0) (actual time=929.298..930.019 rows=7540479 loops=1)  
  Index Cond: (amount < '42000'::numeric)  
Planning Time: 0.083 ms  
Execution Time: 11393.126 ms  
(8 rows)
```

Точная битовая карта не умещается в оперативную память, во многих страницах пришлось проверять все версии строк.

Для большой выборки полное сканирование выгоднее.

Чтобы уменьшить влияние случайных факторов, запросы всегда следует повторять несколько раз, усредняя результаты.

```
=> RESET ALL;
```

```
RESET
```

```
=> DROP INDEX ticket_flights_amount_idx;
```

```
DROP INDEX
```