

# Язык SQL

## Лекция 8

# Повышение производительности

---

**Е. П. Моргунов**

Сибирский государственный университет науки и технологий  
имени академика М. Ф. Решетнева

г. Красноярск

Институт информатики и телекоммуникаций

[emorgunov@mail.ru](mailto:emorgunov@mail.ru)

**Компания Postgres Professional**

г. Москва

На вашем компьютере уже должна быть развернута база данных demo.

- Войдите в систему как пользователь postgres:

```
su - postgres
```

- Должен быть запущен сервер баз данных PostgreSQL:

```
pg_ctl start -D /usr/local/pgsql/data
```

- Для проверки запуска сервера выполните команду

```
pg_ctl status -D /usr/local/pgsql/data
```

или

```
ps -ax | grep postgres | grep -v grep
```

- Если у вас база данных demo была модифицирована, то для ее восстановления выполните команду

```
psql -f demo_small.sql -U postgres (для ОС Debian)
```

```
psql -f demo_small.sql (для ОС Xubuntu)
```

- Запустите утилиту `psql` и подключитесь к базе данных `demo`

```
psql -d demo -U postgres
```

(для ОС Debian)

```
psql -d demo
```

(для ОС Xubuntu)

- Назначьте схему `bookings` в качестве текущей

```
demo=# set search_path = bookings;
```

## 8.1. Основные понятия

- Метод доступа характеризует тот способ, который используется для просмотра таблиц и извлечения только тех строк, которые соответствуют критерию отбора.
- Существуют различные методы доступа: последовательный просмотр (sequential scan), при котором индекс не используется, и группа методов, основанных на использовании индекса. К ней относятся: просмотр по индексу (index scan), просмотр исключительно на основе индекса (index only scan) и просмотр на основе битовой карты (bitmap scan).
- Поскольку и таблицы, и индексы хранятся на диске, то для работы с ними эти объекты считываются в память, в которой они представлены разбитыми на отдельные фрагменты, называемые **страницами**. Эти страницы имеют специальную структуру.
- Размер страниц по умолчанию составляет 8 килобайт.

## Последовательный просмотр (sequential scan)

- При выполнении последовательного просмотра (sequential scan) обращения к индексам не происходит, а строки извлекаются из табличных страниц в соответствии с критерием отбора.
- В том случае, когда в запросе нет предложения WHERE, тогда извлекаются все строки таблицы.
- Данный метод применяется, когда требуется выбрать все строки таблицы или значительную их часть, т. е. когда так называемая *селективность выборки низка*. В таком случае обращение к индексу не ускорит процесс просмотра, а возможно даже и замедлит.

## Просмотр на основе индекса (index scan)

- Просмотр на основе индекса (index scan) предполагает обращение к индексу, созданному для данной таблицы.
- Поскольку в индексе для каждого ключевого значения содержатся уникальные идентификаторы строк в таблицах, то после отыскания в индексе нужного ключа производится обращение к соответствующей странице таблицы и извлечение искомой строки по ее идентификатору.
- При этом нужно учитывать, что хотя записи в индексе упорядочены, но *обращения к страницам таблицы происходят хаотически*, поскольку строки в таблицах не упорядочены.
- В таком случае при *низкой селективности* выборки, т. е. когда из таблицы отбирается значительное число строк, использование индексного поиска может не только не давать ускорения работы, но даже и снижать производительность.

## Просмотр исключительно на основе индекса (index only scan)

- Просмотр исключительно на основе индекса (index only scan), как следует из названия метода, не должен, казалось бы, требовать обращения к строкам таблицы, поскольку все данные, которые нужно получить с помощью запроса, в этом случае присутствуют в индексе.
- Однако в индексе нет информации о *видимости строк транзакциям* — нельзя быть уверенным, что данные, полученные из индекса, видны текущей транзакции.
- Поэтому сначала выполняется обращение к **карте видимости (visibility map)**, которая существует *для каждой таблицы*.
- В ней одним битом отмечены страницы, на которых содержатся только те версии строк, которые видны всем без исключения транзакциям.

## Просмотр исключительно на основе индекса (index only scan) (продолжение)

- Если полученная из индекса версия строки находится на такой странице, значит, эта строка видна текущей транзакции и *обращаться к самой таблице не требуется*.
- Поскольку размер карты видимости очень мал, то в результате сокращается объем операций ввода/вывода.
- Если же строка находится на странице, не отмеченной в карте видимости, тогда происходит обращение и к таблице; в результате никакого выигрыша по быстродействию в сравнении с обычным индексным поиском не достигается.
- Просмотр исключительно на основе индекса особенно эффективен, когда *выбираемые данные изменяются редко*.
- Он может применяться, когда в предложении SELECT указаны только имена столбцов, по которым создан индекс.

## Просмотр на основе битовой карты (bitmap scan)

- Просмотр на основе битовой карты (bitmap scan) является модификацией просмотра на основе индекса.
- Данный метод позволяет оптимизировать индексный поиск за счет того, что сначала производится поиск в индексе *для всех искомых строк* и формирование так называемой битовой карты, в которой указывается, в каких страницах таблицы эти строки содержатся.
- После того как битовая карта сформирована, выполняется извлечение строк из страниц таблицы, но при этом *обращение к каждой странице производится только один раз*.

- Другим важным понятием является способ соединения наборов строк (join). Набор строк может быть получен из таблицы с помощью одного из методов доступа, описанных выше.
- Набор строк может быть получен не только из одной таблицы, а может быть результатом соединения других наборов.
- Важно различать способ *соединения таблиц* (JOIN) и способ *соединения наборов строк*.
- Первое понятие относится к языку SQL и является высокоуровневым, логическим, оно не касается вопросов реализации.
- А второе относится именно к реализации, это — механизм непосредственного выполнения соединения наборов строк.
- Принципиально важным является то, что за один раз соединяются только два набора строк.
- Существует три способа соединения: вложенный цикл (nested loop), хеширование (hash join) и слияние (merge join).
- Они имеют свои особенности, которые PostgreSQL учитывает при выполнении конкретных запросов.

## Вложенный цикл (nested loop)

- Суть способа «вложенный цикл» в том, что перебираются строки из «внешнего» набора и для каждой из них выполняется поиск соответствующих строк во «внутреннем» наборе.
- Если соответствующие строки найдены, то выполняется их соединение со строкой из «внешнего» набора.
- При этом способы выбора строк из обоих наборов могут быть различными.
- Метод поддерживает соединения как на основе равенства значений атрибутов (эквисоединения), так и *любые другие виды условий*.
- Поскольку он не требует подготовительных действий, то *способен быстро приступить к непосредственной выдаче результата*.
- Метод эффективен для *небольших выборок*.

## Соединение хешированием (hash join)

- При соединении хешированием строки одного набора помещаются в хеш-таблицу, содержащуюся в памяти, а строки из второго набора перебираются, и для каждой из них проверяется наличие соответствующих строк в хеш-таблице.
- Ключом хеш-таблицы является тот столбец, по которому выполняется соединение наборов строк.
- *Как правило, число строк в том наборе, на основе которого строится хеш-таблица, меньше, чем во втором наборе.*
- Это позволяет уменьшить ее размер и ускорить процесс обращения к ней.
- Данный метод работает только при выполнении *эквисоединений*, поскольку для хеш-таблицы имеет смысл только проверка на равенство проверяемого значения одному из ее ключей.
- Метод эффективен для *больших выборок*.

## Соединение методом слияния (merge join)

- Соединение методом слияния производится аналогично сортировке слиянием.
- В этом случае оба набора строк должны быть предварительно отсортированы по тем столбцам, по которым производится соединение.
- Затем параллельно читаются строки из обоих наборов и сравниваются значения столбцов, по которым производится соединение.
- При совпадении значений формируется результирующая строка.
- Этот процесс продолжается до исчерпания строк в обоих наборах.
- Этот метод, как и метод соединения хешированием, работает только при выполнении *эквисоединений*.
- Он пригоден для работы с *большими наборами строк*.

## 8.2. Методы просмотра таблиц

- Прежде чем приступить к непосредственному выполнению каждого запроса, PostgreSQL формирует план его выполнения.
- Чтобы достичь хорошей производительности, этот план должен учитывать свойства данных.
- Планированием занимается специальная подсистема — **планировщик (planner)**.
- Просмотреть план выполнения запроса можно с помощью команды EXPLAIN.
- Для детального понимания планов выполнения сложных запросов требуется опыт. Мы изложим лишь основные приемы работы с этой командой.

- Структура плана запроса представляет собой *дерево*, состоящее из так называемых **узлов плана (plan nodes)**.
- Узлы на нижних уровнях дерева отвечают за просмотр и выдачу строк таблиц, которые осуществляются с помощью методов доступа, описанных выше.
- Если конкретный запрос требует выполнения операций агрегирования, соединения таблиц, сортировки, то над узлами выборки строк будут располагаться дополнительные узлы дерева плана.
- Например, для соединения наборов строк будут использоваться способы, которые мы только что рассмотрели.
- Для каждого узла дерева плана команда EXPLAIN выводит по одной строке, при этом выводятся также оценки стоимости выполнения операций на каждом узле, которые делает планировщик.
- В случае необходимости для конкретных узлов могут выводиться дополнительные строки. Самая первая строка плана содержит *общую оценку стоимости* выполнения данного запроса.

**EXPLAIN SELECT \* FROM aircrafts;**

- В ответ получим план выполнения запроса:

оценка ресурсов, требуемых для того, чтобы *приступить* к выводу данных

оценка *общей стоимости* выполнения запроса

оценка *среднего размера* извлекаемых строк

QUERY PLAN

-----  
Seq Scan on aircrafts (cost=0.00..1.09 rows=9 width=52)  
(1 строка)

это условные единицы – важны *соотношения стоимостей*

оценка *общего числа* извлекаемых строк

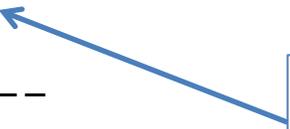
- Поскольку в этом запросе нет предложения WHERE, он должен просмотреть все строки таблицы, поэтому планировщик выбирает последовательный просмотр (sequential scan).
- В скобках приведены важные параметры плана.

- Первая оценка равна нулю, поскольку никакие дополнительные операции с выбранными строками не предполагаются, и PostgreSQL может сразу же выводить прочитанные строки.
- Формируя эту оценку, планировщик исходит из предположения, что данный узел плана запроса выполняется до конца, т. е. извлекаются все имеющиеся строки таблицы. Исключения: в запросе SELECT предложение LIMIT.
- Обе оценки вычисляются на основе ряда параметров сервера баз данных.
- Для каждого запроса планировщик формирует несколько планов.
- При сравнении различных вариантов плана, как правило, для выполнения выбирается тот, который имеет наименьшую общую стоимость выполнения запроса.
- Оценки числа строк и их размера планировщик получает на основе статистики, накапливаемой в специальных системных таблицах.

- В том случае, когда нас не интересуют численные оценки, можно воспользоваться параметром COSTS OFF:

```
EXPLAIN ( COSTS OFF ) SELECT * FROM aircrafts;  
          QUERY PLAN
```

```
-----  
Seq Scan on aircrafts  
(1 строка)
```



обратите внимание  
на скобки

- Сформируем запрос с предложением WHERE:

```
EXPLAIN SELECT * FROM aircrafts WHERE model ~ 'Air';
```

Оценка числа строк изменилась с 9 на 1. Но планировщик немного ошибся — фактически их будет три.

QUERY PLAN

-----  
Seq Scan on aircrafts (cost=0.00..1.11 rows=1 width=52)

→ **Filter:** (model ~ 'Air'::text)  
(2 строки)

это число строк в плане, а не в выборке из таблицы

узел, описывающий критерий отбора строк

- Добавим сортировку данных:

```
EXPLAIN SELECT * FROM aircrafts ORDER BY aircraft_code;
```

Для сортировки требуется некоторое время, поэтому вывод начинается не сразу. В оценку 1,23 входит и оценка стоимости получения выборки — 1,09.

QUERY PLAN

Sort (cost=1.23..1.26 rows=9 width=52)

Sort Key: aircraft\_code

-> Seq Scan on aircrafts (cost=0.00..1.09 rows=9 width=52)

(3 строки)

дополнительный узел плана

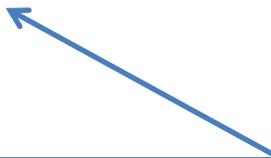
Хотя по столбцу aircraft\_code создан индекс, но он не используется.

Когда таблица очень маленькая, то обращение к индексу не даст выигрыша в скорости, а лишь *добавит* к операциям чтения страниц, в которых хранятся строки таблиц, еще и *операции чтения страниц с записями индекса*.

```
EXPLAIN SELECT * FROM bookings ORDER BY book_ref;  
QUERY PLAN
```

---

```
Index Scan using bookings_pkey on bookings  
  (cost=0.42..8511.24, rows=262788 width=21)  
(1 строка)
```



Первая оценка стоимости в плане — не нулевая. Хотя индекс уже упорядочен, и дополнительная сортировка не требуется, но для того, чтобы найти в индексе первую строку в соответствии с требуемым порядком, тоже нужно некоторое время.

- Если к сортировке добавить еще и условие отбора строк, то это отразится в дополнительной строке верхнего (и единственного) узла плана.

**EXPLAIN**

```
SELECT * FROM bookings
```

```
WHERE book_ref > '0000FF' AND book_ref < '000FFF'
```

```
ORDER BY book_ref;
```

QUERY PLAN

-----  
Index Scan using bookings\_pkey on bookings  
(cost=0.42..9.50 rows=54, width=21)

Index Cond: ((book\_ref > '0000FF'::bpchar) AND  
(book\_ref <, '000FFF'::bpchar))

(2 строки)

Поскольку столбец, по которому производится отбор строк, является индексируемым, то их отбор реализуется не через Filter, а через Index Cond.

```
EXPLAIN SELECT * FROM seats WHERE aircraft_code = 'SU9';
```

сканируются страницы таблицы seats на основе битовой карты

QUERY PLAN

```
-----  
Bitmap Heap Scan on seats (cost=5.03..14.24 rows=97  
width=15)
```

```
Recheck Cond: (aircraft_code = 'SU9'::bpchar)
```

```
-> Bitmap Index Scan on seats_pkey (cost=0.00..5.00  
rows=97 width=0)
```

```
Index Cond: (aircraft_code = 'SU9'::bpchar)
```

(4 строки)

строится битовая карта

для отбора строк в соответствии с предложением WHERE используется индекс

сами строки на этом этапе еще не выбираются

```
EXPLAIN
SELECT book_ref
FROM bookings
WHERE book_ref < '000FFF'
ORDER BY book_ref;
```

QUERY PLAN

```
-----
Index Only Scan using bookings_pkey on bookings
      (cost=0.42..9.42 rows=57, width=7)
  Index Cond: (book_ref < '000FFF'::bpchar)
(2 строки)
```

- В этом плане только один узел — Index Only Scan. Здесь также первая оценка стоимости не нулевая, т. к. отыскание в индексе наименьшего значения требует некоторого времени.

- Посмотрим, как отражаются в планах выполнения запросов различные агрегатные функции. Начнем с простого подсчета строк.

**EXPLAIN**

```
SELECT count( * ) FROM seats
WHERE aircraft_code = 'SU9';
```

QUERY PLAN

стоимость стадии  
агрегирования примерно  
 $14.48 - 14.24 = 0.24$

-----  
Aggregate (cost=14.48..14.49 rows=1 width=8)

→ Bitmap Heap Scan on seats (cost=5.03..14.24  
rows=97 width=0)

Recheck Cond: (aircraft\_code = 'SU9'::bpchar)

→ Bitmap Index Scan on seats\_pkey

(cost=0.00..5.00 rows=97, width=0)

Index Cond: (aircraft\_code = 'SU9'::bpchar)

(5 строк)

Выполняется обращение к страницам таблицы (хотя никакие значения атрибутов не выбираются), чтобы проверить *видимость версий строк* для разных транзакций.

- А в этом примере агрегирование связано уже с вычислениями на основе значений конкретного столбца, а не просто с подсчетом строк.

```
EXPLAIN SELECT avg( total_amount ) FROM bookings;
```

```
QUERY PLAN
```

```
-----  
Aggregate (cost=4958.85..4958.86 rows=1 width=32)  
  -> Seq Scan on bookings (cost=0.00..4301.88 rows=262788  
        width=6)
```

(2 строки)

## 8.3. Методы формирования соединений наборов строк

- Для получения списка мест в салонах самолетов Airbus с указанием класса обслуживания сформируем запрос, в котором соединяются две таблицы: «Места» (seats) и «Самолеты» (aircrafts).

**EXPLAIN**

```
SELECT a.aircraft_code, a.model,  
       s.seat_no, s.fare_conditions  
FROM seats s JOIN aircrafts a  
       ON s.aircraft_code = a.aircraft_code  
WHERE a.model ~ '^Air'  
ORDER BY s.seat_no;
```

# Метод вложенного цикла (nested loop) (2)

сортировка по ключу

QUERY PLAN

```
-----  
Sort (cost=23.28..23.65 rows=149 width=59)  
  Sort Key: s.seat_no  
    -> Nested Loop (cost=5.43..17.90 rows=149 width=59)  
      -> Seq Scan on aircrafts a (cost=0.00..1.11  
        rows=1 width=48)  
        Filter: (model ~ '^Air'::text)  
      -> Bitmap Heap Scan on seats s (cost=5.43..15.29  
        rows=149, width=15)  
        Recheck Cond: (aircraft_code =  
          a.aircraft_code)  
      -> Bitmap Index Scan on seats_pkey  
        (cost=0.00..5.39, rows=149 width=0)  
        Index Cond: (aircraft_code =  
          a.aircraft_code)
```

внешний  
дочерний  
узел

внутренний  
дочерний  
узел

формирование битовой  
карты

(9 строк)

текущее значение атрибута aircraft\_code,  
по которому выполняется соединение

- Узел Nested Loop, в котором выполняется соединение, имеет два дочерних узла: внешний — Seq Scan и внутренний — Bitmap Heap Scan.
- Во внешнем узле последовательно сканируется таблица aircrafts с целью отбора строк согласно условию Filter: (model ~'^Air'::text).
- *Для каждой из отобранных строк* во внутреннем дочернем узле (Bitmap Heap Scan) выполняется поиск в таблице seats по индексу с использованием битовой карты.
- Она формируется в узле Bitmap Index Scan с учетом условия Index Cond: (aircraft\_code = a.aircraft\_code), т. е. *для текущего значения* атрибута aircraft\_code, по которому выполняется соединение.
- На верхнем уровне плана сформированные строки сортируются по ключу (Sort Key: s.seat\_no).

- Получим список маршрутов с указанием модели самолета, выполняющего рейсы по этим маршрутам. Воспользуемся таблицами «Маршруты» (routes) и «Самолеты» (aircrafts).

**EXPLAIN**

```
SELECT r.flight_no, r.departure_airport_name,  
       r.arrival_airport_name, a.model  
FROM routes r JOIN aircrafts a  
     ON r.aircraft_code = a.aircraft_code  
ORDER BY flight_no;
```

# Соединение хешированием (hash join) (2)

сортировка по ключу

QUERY PLAN

```
-----  
Sort (cost=24.25..24.31 rows=21 width=124)  
  Sort Key: r.flight_no  
    -> Hash Join (cost=1.20..23.79 rows=21 width=124)  
      Hash Cond: (r.aircraft_code = a.aircraft_code)  
        -> Seq Scan on routes r (cost=0.00..20.64 rows=464  
          width=108)  
          -> Hash (cost=1.09..1.09 rows=9 width=48)  
            -> Seq Scan on aircrafts a (cost=0.00..1.09  
              rows=9 width=48)
```

поиск в  
хеш-  
таблице

(7 строк)

Формируется хеш-таблица, ключами которой являются значения атрибута aircraft\_code, т. к. именно по нему выполняется соединение таблиц.

**ВАЖНО!** Число строк в таблице aircrafts меньше, чем в routes.

- На самом внутреннем уровне плана последовательно сканируется (Seq Scan) таблица `aircrafts`, и формируется хеш-таблица, ключами которой являются значения атрибута `aircraft_code`, т. к. именно по нему выполняется соединение таблиц.
- Затем последовательно сканируется (Seq Scan) таблица `routes`, и для *каждой ее строки* выполняется поиск значения атрибута `aircraft_code` среди ключей хеш-таблицы: Hash Cond: (`r.aircraft_code = a.aircraft_code`).
- Если такой поиск успешен, значит, формируется комбинированная результирующая строка выборки.
- На верхнем уровне плана сформированные строки сортируются.
- Обратите внимание, что хеш-таблица создана на основе той таблицы, *число строк в которой меньше*, т. е. `aircrafts`. Таким образом, поиск в ней будет выполняться быстрее, чем если бы хеш-таблица была создана на основе таблицы `routes`.

- Для иллюстрации воспользуемся простым запросом, построенным на основе таблиц «Билеты» (tickets) и «Перелеты» (ticket\_flights). Он выбирает *для каждого билета все перелеты, включенные в него*. Конечно, это очень упрощенный запрос, в реальной ситуации он не представлял бы большой практической пользы, но в целях упрощения плана и повышения наглядности, воспользуемся им.

**EXPLAIN**

```
SELECT t.ticket_no, t.passenger_name,  
       tf.flight_id, tf.amount  
FROM tickets t JOIN ticket_flights tf  
     ON t.ticket_no = tf.ticket_no  
ORDER BY t.ticket_no;
```

отдельный узел для  
сортировки не требуется

QUERY PLAN

-----  
Merge Join (cost=**1.51**..98276.90 rows=1045726 width=40)  
Merge Cond: (t.ticket\_no = tf.ticket\_no)  
-> Index Scan using **tickets\_pkey** on tickets t  
(cost=0.42..**17230.42**, rows=366733 width=30)  
-> Index Scan using **ticket\_flights\_pkey**  
on ticket\_flights tf, (cost=0.42..**67058.74**  
rows=1045726 width=24)

(4 строки)

Верхний узел (Merge Join)  
получает наборы строк этих  
таблиц уже в отсортированном  
виде.

$1.51 < 17230.42$  и  $1.51 < 67058.74$ ,  
значит, вывод результирующих строк  
начнется еще задолго до завершения  
сканирования исходных таблиц.

- Два внутренних узла дерева плана отвечают за сканирование таблиц `tickets` и `ticket_flights` по индексам (Index Scan). Таким образом, верхний узел (Merge Join) получает наборы строк этих таблиц *уже в отсортированном виде*, поэтому не требуется отдельного узла для сортировки результирующих строк.
- Первая оценка в узле Merge Join равна 1,51, что значительно меньше вторых оценок, вычисленных планировщиком для двух нижних узлов, а именно: 17230,42 и 67058,74.
- Напомним, что первая оценка говорит, сколько ресурсов будет затрачено (сколько времени, в условных единицах, пройдет) до начала вывода первых результатов выполнения операции на данном уровне дерева плана.
- Вторая оценка показывает общее количество ресурсов, требующихся для полного завершения операции на данном уровне дерева плана.
- Таким образом, можно заключить, что *вывод результирующих строк начнется еще задолго до завершения сканирования исходных таблиц*.

## 8.4. Управление планировщиком

- Для управления планировщиком предусмотрен целый ряд параметров.
- Их можно изменить на время текущего сеанса работы с помощью команды SET.
- Изменять параметры в производственной базе данных следует только в том случае, когда вы *обоснованно считаете*, что планировщик ошибается.
- Однако для того чтобы научиться видеть ошибки планировщика, нужен большой опыт.

**ВАЖНО!** Следует рассматривать приведенные далее команды управления планировщиком лишь *с позиции изучения потенциальных возможностей* управления им, а не как рекомендацию к бездумному изменению этих параметров в реальной работе.

- Чтобы запретить планировщику использовать метод соединения на основе хеширования, нужно сделать так:  
`SET enable_hashjoin = off;`
- Чтобы запретить планировщику использовать метод соединения слиянием, нужно сделать так:  
`SET enable_mergejoin = off;`
- А для того чтобы запретить планировщику использовать соединение методом вложенного цикла, нужно сделать так:  
`SET enable_nestloop = off;`
- По умолчанию все эти параметры имеют значение «on» (включено).

**ВАЖНО!** Необходимо уточнить, что в результате выполнения вышеприведенных команд *не накладывается полного запрета* на использование конкретного метода соединения наборов строк. Методу просто назначается *очень высокая стоимость*, но планировщик все равно сохраняет возможность маневра, и даже такой «запрещенный» метод может быть использован.

- Давайте запретим планировщику использовать метод соединения слиянием:

```
SET enable_mergejoin = off;
```

```
SET
```

- Теперь повторим предыдущий запрос:

```
EXPLAIN
```

```
SELECT t.ticket_no, t.passenger_name,  
       tf.flight_id, tf.amount  
FROM tickets t JOIN ticket_flights tf  
     ON t.ticket_no = tf.ticket_no  
ORDER BY t.ticket_no;
```

оценки стоимости выполнения запроса стали значительно выше

QUERY PLAN

```
-----  
Sort (cost=226400.55..229014.87 rows=1045726 width=40)  
Sort Key: t.ticket_no  
-> Hash Join (cost=16824.49..64658.49 rows=1045726  
width=40)  
Hash Cond: (tf.ticket_no = t.ticket_no)  
-> Seq Scan on ticket_flights tf  
      (cost=0.00..18692.26, rows=1045726 width=24)  
-> Hash (cost=9733.33..9733.33 rows=366733  
width=30)  
      -> Seq Scan on tickets t (cost=0.00..9733.33  
rows=366733, width=30)
```

соединение  
хешированием

(7 строк)

При этом вывод результирующих строк начнется значительно позднее, чем при использовании метода соединения слиянием: значение параметра cost для верхнего узла дерева плана — cost=226400.55..229014.87.

- В команде EXPLAIN можно указать опцию ANALYZE, что позволит выполнить запрос и вывести на экран фактические затраты времени на выполнение запроса и число фактически выбранных строк.
- При этом, хотя запрос и выполняется, сами результирующие строки не выводятся.
- Сначала разрешим планировщику использовать метод соединения слиянием:

```
SET enable_mergejoin = on;
```

```
SET
```

- Повторим предыдущий запрос с опцией ANALYZE.

```
EXPLAIN ANALYZE
```

```
SELECT t.ticket_no, t.passenger_name,  
       tf.flight_id, tf.amount  
FROM tickets t JOIN ticket_flights tf ON  
       t.ticket_no = tf.ticket_no  
ORDER BY t.ticket_no;
```

# Какой получается теперь план запроса?

## QUERY PLAN

```
-----  
Merge Join (cost=1.51..98276.90 rows=1045726 width=40)  
→ (actual time=0.087..10642.643 rows=1045726 loops=1)  
Merge Cond: (t.ticket_no = tf.ticket_no)  
-> Index Scan using tickets_pkey on tickets t  
    (cost=0.42..17230.42 rows=366733 width=30)  
    (actual time=0.031..762.460 rows=366733 loops=1)  
-> Index Scan using ticket_flights_pkey  
    on ticket_flights tf  
    (cost=0.42..67058.74 rows=1045726 width=24)  
    (actual time=0.006..7743.705 rows=1045726 loops=1)
```

факт

← время формирования плана

← время выполнения запроса

Planning time: 122.347 ms

Execution time: 10948.791 ms

(6 строк)

loops – фактическое число повторений  
того или иного узла дерева плана

- Фактические затраты времени измеряются в миллисекундах, а оценки стоимости — в условных единицах, поэтому плановые оценки и фактические затраты *совпасть не могут*.
- Важнее обратить внимание на то, насколько точно планировщик оценил *число обрабатываемых строк*, а также на фактическое число повторений того или иного узла дерева плана — это параметр `loops`.
- В данном запросе каждый узел плана был выполнен ровно один раз, поскольку выбор строк из обоих соединяемых наборов производился по индексу, поэтому достаточно одного прохода по каждому набору.
- Число выбираемых строк было оценено точно, поскольку таблицы связаны по внешнему ключу, и в выборку включаются все их строки (нет предложения `WHERE`).
- Фактические затраты времени на разных компьютерах будут различаться.
- Будет другим и фактическое время при повторном выполнении запроса на одном и том же компьютере (буферизация и кэширование, может изменяться фактическая нагрузка на сервер).

- Если модифицировать запрос, добавив предложение WHERE, то точного совпадения оценки числа выбираемых строк и фактического их числа уже не будет.

```
EXPLAIN ANALYZE
```

```
SELECT t.ticket_no, t.passenger_name,  
       tf.flight_id, tf.amount  
FROM tickets t JOIN ticket_flights tf  
     ON t.ticket_no = tf.ticket_no  
WHERE amount > 50000  
ORDER BY t.ticket_no;
```

# Что получается теперь?

метод тот же, но  
добавилась сортировка

## QUERY PLAN

```
-----  
Merge Join (cost=27391.09..46664.80 rows=75126 width=40)  
  (actual time=2133.715..3117.200 rows=72647 loops=1)  
Merge Cond: (t.ticket_no = tf.ticket_no)  
-> Index Scan using tickets_pkey on tickets t  
  (cost=0.42..17230.42 rows=366733 width=30)  
  (actual time=0.009..318.517 rows=366733 loops=1)  
-> Sort (cost=27390.66..27578.48 rows=75126 width=24)  
  (actual time=2132.781..2173.526 rows=72647  
  loops=1)
```

работает  
ANALYZE

Sort Key: tf.ticket\_no  
Sort Method: external sort Disk: 2768kB

последоват.  
сканирование

-> Seq Scan on ticket\_flights tf  
(cost=0.00..21306.58 rows=75126 width=24)  
(actual time=0.351..332.313 rows=72647  
loops=1)

WHERE

Filter: (amount > '50000':numeric)  
Rows Removed by Filter: 973079

работает  
ANALYZE

Planning time: 1.415 ms  
Execution time: 3135.869 ms  
(11 строк)

оценки числа строк (rows) были  
довольно точными

- Обратимся еще раз к запросу, который мы уже рассматривали выше, и выполним его с опцией ANALYZE.
- В плане этого запроса нас будет интересовать фактический параметр loops.

**EXPLAIN ANALYZE**

```
SELECT a.aircraft_code, a.model,  
       s.seat_no, s.fare_conditions  
FROM seats s JOIN aircrafts a  
     ON s.aircraft_code = a.aircraft_code  
WHERE a.model ~ '^Air'  
ORDER BY s.seat_no;
```

## QUERY PLAN

```
-----  
Sort (cost=23.28..23.65 rows=149 width=59)  
  (actual time=3.423..3.666 rows=426 loops=1)  
  Sort Key: s.seat_no  
  Sort Method: quicksort Memory: 46kB  
  -> Nested Loop (cost=5.43..17.90 rows=149 width=59)  
    (actual time=0.236..0.993 rows=426 loops=1)  
    -> Seq Scan on aircrafts a (cost=0.00..1.11 rows=1  
      width=48)  
      (actual time=0.100..0.112 rows=3 loops=1)  
      Filter: (model ~ '^Air'::text)  
      Rows Removed by Filter: 6  
    -> Bitmap Heap Scan on seats s (cost=5.43..15.29  
      rows=149, width=15)  
      (actual time=0.080..0.154 rows=142 loops=3)  
      Recheck Cond: (aircraft_code = a.aircraft_code)  
      Heap Blocks: exact=6  
      -> Bitmap Index Scan on seats_pkey  
        (cost=0.00..5.39, rows=149 width=0)  
        (actual time=0.064..0.064 rows=142 loops=3)  
        Index Cond: (aircraft_code = a.aircraft_code)
```

метод  
сортировки

Planning time: 0.554 ms  
Execution time: 3.840 ms  
(14 строк)

Из таблицы aircrafts были фактически  
выбраны три строки, и для каждой из них  
выполняется поиск в таблице seats.

- Как видно из плана, значение параметра loops для узла, выполняющего сканирование таблицы seats по индексу с построением битовой карты, равно трем.
- Это объясняется тем, что из таблицы aircrafts были фактически выбраны три строки, и для каждой из них выполняется поиск в таблице seats.
- Для подсчета общих затрат времени на выполнение операций сканирования по индексу за три цикла нужно значение параметра actual time умножить на значение параметра loops.
- Таким образом, для узла дерева плана Bitmap Index Scan получим:
$$0,064 \times 3 = 0,192.$$
- Подобные вычисления общих затрат времени на промежуточных уровнях дерева плана могут помочь выявить наиболее ресурсоемкие операции.
- Согласно этому плану, сортировка на верхнем уровне плана выполнялась в памяти с использованием метода quicksort:  
Sort Method: quicksort Memory: 46kB

# А если запрос модифицирует данные?

```
BEGIN;  
BEGIN  
EXPLAIN ANALYZE  
UPDATE aircrafts  
SET range = range + 100  
WHERE model ~ '^Air';
```

Нужно воспользоваться  
транзакцией ...

## QUERY PLAN

```
-----  
Update on aircrafts (cost=0.00..1.11 rows=1 width=58)  
  (actual time=0.299..0.299 rows=0 loops=1)  
-> Seq Scan on aircrafts (cost=0.00..1.11 rows=1  
    width=58)  
    (actual time=0.111..0.121 rows=3 loops=1)  
    Filter: (model ~ '^Air'::text)  
    Rows Removed by Filter: 6
```

```
Planning time: 0.235 ms  
Execution time: 0.414 ms
```

(6 строк)

```
ROLLBACK;
```

```
ROLLBACK
```

... с откатом изменений.

- В документации приводится важное предостережение о том, что нельзя экстраполировать, т. е. распространять, пусть даже и с некоторыми поправками, оценки, полученные для таблиц небольшого размера, на таблицы большого размера.
- Это объясняется тем, что *оценки*, вычисляемые планировщиком, *не являются линейными*.
- Одна из причин заключается в том, что для таблиц разных размеров могут быть выбраны разные планы.
- Например, для маленькой таблицы может быть выбрано последовательное сканирование, а для большой — сканирование по индексу.

## 8.5. Оптимизация запросов

- При принятии решения о том, что выполнение какого-либо запроса нужно оптимизировать (ускорить его выполнение), следует учитывать не только абсолютное время его выполнения, но и частоту его использования.
- Повлиять на скорость выполнения запроса можно различными способами:
  - обновление статистики, на основе которой планировщик строит планы;
  - изменение исходного кода запроса;
  - изменение схемы данных, связанное с денормализацией: создание материализованных представлений и временных таблиц, создание индексов, использование вычисляемых столбцов таблиц;

- изменение параметров планировщика, управляющих выбором порядка соединения наборов строк: использование общих табличных выражений (запросы с предложением WITH), использование фиксированного порядка соединения (параметр `join_collapse_limit = 1`), запрет раскрытия подзапросов и преобразования их в соединения таблиц (параметр `from_collapse_limit = 1`);
- изменение параметров планировщика, управляющих выбором метода доступа (`enable_seqscan`, `enable_indexscan`, `enable_indexonlyscan`, `enable_bitmapscan`)
- и способа соединения наборов строк (`enable_nestloop`, `enable_hashjoin`, `enable_mergejoin`);
- изменение параметров планировщика, управляющих использованием ряда операций: агрегирование на основе хеширования (`enable_hashagg`), материализация временных наборов строк (`enable_material`), выполнение явной сортировки при наличии других возможностей (`enable_sort`).

- Необходимым условием для того, чтобы планировщик выбрал правильный план, является наличие *актуальной статистики*.
- Если вы предполагаете, что планировщик опирается на неактуальную статистику, можно ее принудительно обновить с помощью команды **ANALYZE**.
- Например, обновить статистику для таблицы `aircrafts` можно так:  
**ANALYZE aircrafts;**  
ANALYZE

- В качестве примера ситуации, в которой оптимизация запроса представляется обоснованной, рассмотрим следующую задачу.
- Предположим, что необходимо определить степень загруженности кассиров нашей авиакомпании в сентябре 2016 г. Для этого, в частности, требуется выявить распределение числа операций бронирования по числу билетов, оформленных в рамках этих операций.
- Другими словами, это означает, что нужно подсчитать число операций бронирования, в которых был оформлен только один билет, число операций, в которых было оформлено два билета и т. д.

- Эту задачу можно переформулировать так: для каждой строки, отобранной из таблицы «Бронирования» (bookings), нужно подсчитать соответствующие строки в таблице «Билеты» (tickets).
- Речь идет о строках, в которых значение поля book\_ref такое же, что и в текущей строке таблицы bookings.
- Буквальное следование такой формулировке задачи приводит к получению запроса с коррелированным подзапросом в предложении SELECT.
- Но это еще не окончательное решение. Теперь нужно сгруппировать полученный набор строк по значениям числа оформленных билетов.

```
EXPLAIN
SELECT num_tickets, count( * ) AS num_bookings
FROM ( SELECT b.book_ref,
           ( SELECT count( * )
             FROM tickets t
             WHERE t.book_ref = b.book_ref
           )
      FROM bookings b
      WHERE date_trunc( 'mon', book_date ) = '2016-09-01'
    ) AS count_tickets( book_ref, num_tickets )
GROUP by num_tickets
ORDER BY num_tickets DESC;
```

очень большие оценки общей стоимости выполнения запроса

QUERY PLAN

```
GroupAggregate (cost=14000017.12..27994373.35 rows=1314
                width=16)
  Group Key: ((SubPlan 1))
  -> Sort (cost=14000017.12..14000020.40 rows=1314
          width=8)
      Sort Key: ((SubPlan 1)) DESC
      -> Seq Scan on bookings b
          (cost=0.00..13999949.05 rows=1314 width=8)
          Filter: (date_trunc('mon'::text, book_date) =
                  '2016-09-01 00:00:00+08'::timestamp
                  with time zone)
```

подзапрос

```
SubPlan 1
-> Aggregate (cost=10650.17..10650.18 rows=1
            width=8)
    -> Seq Scan on tickets t
        (cost=0.00..10650.16 rows=2 width=0)
        Filter: (book_ref = b.book_ref)
```

(10 строк)

- Что можно сделать для ускорения выполнения запроса?
- Давайте создадим индекс для таблицы tickets по столбцу book\_ref, по которому происходит поиск в ней.

```
CREATE INDEX tickets_book_ref_key ON tickets ( book_ref );  
CREATE INDEX
```

- Повторим запрос, добавив параметр ANALYZE в команду EXPLAIN.

сравните

## QUERY PLAN

```
-----  
GroupAggregate (cost=22072.70..38484.52 rows=1314 width=16)  
  (actual time=3656.554..3787.562 rows=5 loops=1)  
  Group Key: ((SubPlan 1))  
    -> Sort (cost=22072.70..22075.99 rows=1314 width=8)  
      (actual time=3656.533..3726.969 rows=165534 loops=1)  
      Sort Key: ((SubPlan 1)) DESC  
      Sort Method: external merge Disk: 2912kB  
      -> Seq Scan on bookings b (cost=0.00..22004.64 rows=1314 width=8)  
        (actual time=0.219..3332.162 rows=165534 loops=1)  
        Filter: (date_trunc('mon'::text, book_date) =  
          '2016-09-01 00:00:00+08'::timestamp with time zone)  
        Rows Removed by Filter: 97254  
        SubPlan 1  
          -> Aggregate (cost=12.46..12.47 rows=1 width=8)  
            (actual time=0.016..0.016 rows=1 loops=165534)  
            -> Index Only Scan using tickets_book_ref_key on tickets t  
              (cost=0.42..12.46 rows=2 width=0)  
              (actual time=0.013..0.014 rows=1 loops=165534)  
              Index Cond: (book_ref = b.book_ref)  
              Heap Fetches: 230699
```

сравните

ПОИСК  
ТОЛЬКО ПО  
ИНДЕКСУ

Planning time: 0.290 ms  
Execution time: 3788.690 ms  
(15 строк)

запрос стал выполняться быстрее

# Что получается в результате?

```
num_tickets | num_bookings
-----+-----
          5 |             13
          4 |            536
          3 |           7966
          2 |          47573
          1 |         109446
```

(5 строк)

- Кроме создания индекса есть и другой способ: замена коррелированного подзапроса соединением таблиц.

**EXPLAIN ANALYZE**

```
SELECT num_tickets, count( * ) AS num_bookings  
FROM ( SELECT b.book_ref, count( * )  
        FROM bookings b, tickets t  
        WHERE date_trunc( 'mon', b.book_date ) =  
                '2016-09-01' AND  
                t.book_ref = b.book_ref  
        GROUP BY b.book_ref  
    ) AS count_tickets( book_ref, num_tickets )  
GROUP by num_tickets  
ORDER BY num_tickets DESC;
```

# Какой теперь будет план? (1)

## QUERY PLAN

```
-----  
GroupAggregate (cost=16966.67..16978.53 rows=200 width=16)  
  (actual time=4092.258..4219.477 rows=5 loops=1)  
  Group Key: count_tickets.num_tickets  
  -> Sort (cost=16966.67..16969.96 rows=1314 width=8)  
    (actual time=4092.236..4161.294 rows=165534 loops=1)  
    Sort Key: count_tickets.num_tickets DESC  
    Sort Method: external merge Disk: 2912kB  
    -> Subquery Scan on count_tickets  
      (cost=16858.57..16898.61 rows=1314 width=8)  
      (actual time=3176.113..3862.133 rows=165534 loops=1)  
      -> GroupAggregate  
        (cost=16858.57..16885.47 rows=1314 width=15)  
        (actual time=3176.111..3765.157 rows=165534 loops=1)  
        Group Key: b.book_ref  
        -> Sort (cost=16858.57..16863.16 rows=1834 width=7)  
          (actual time=3176.098..3552.334 rows=230699 loops=1)  
          Sort Key: b.book_ref  
          Sort Method: external merge Disk: 3824kB  
          -> Hash Join (cost=5632.24..16759.16 rows=1834 width=7)  
            (actual time=498.701..1091.509 rows=230699 loops=1)  
            Hash Cond: (t.book_ref = b.book_ref)
```

**ORDER BY  
num\_tickets DESC**

**подзапрос**

**соединение  
хешированием**

# Какой теперь будет план? (2)

индекс по таблице  
tickets  
игнорируется

путем  
последовательного  
просмотра создается  
хеш для таблицы  
bookings: она  
меньше

```
-> Seq Scan on tickets t
(cost=0.00..9733.33 rows=366733 width=7)
(actual time=0.047..170.792 rows=366733
loops=1)
-> Hash (cost=5615.82..5615.82 rows=1314
width=7)
(actual time=498.624..498.624
rows=165534 loops=1)
Buckets: 262144 (originally 2048)
Batches: 2 (originally 1)
Memory Usage: 3457kB
-> Seq Scan on bookings b
(cost=0.00..5615.82 rows=1314
width=7)
(actual time=0.019..267.728
rows=165534 loops=1)
Filter: (date_trunc('mon'::text,
book_date) =
'2016-09-01 00:00:00+08'::timestamp
with time zone)
Rows Removed by Filter: 97254
```

Planning time: 2.183 ms  
Execution time: 4221.133 ms  
(21 строка)

- Время выполнения модифицированного запроса оказывается несколько большим, чем в предыдущем случае, когда в запросе присутствовал коррелированный подзапрос.
- Таким образом, можно заключить, что для ускорения работы оригинального запроса можно было либо *создать индекс*, либо *модифицировать сам запрос, даже не создавая индекса*.
- Другие методы оптимизации выполнения запросов представлены в разделе «Контрольные вопросы и задания». Рекомендуем вам самостоятельно с ними ознакомиться и поэкспериментировать.
- Перед выполнением упражнений нужно восстановить измененные значения параметров:

```
SET enable_hashjoin = on;
```

```
SET
```

```
SET enable_nestloop = on;
```

```
SET
```

- Подробно с технологиями оптимизации запросов можно ознакомиться с помощью учебного курса DBA2 компании Postgres Professional <https://postgrespro.ru/education/courses/DBA2>

1. Грофф, Дж. SQL. Полное руководство : пер. с англ. / Джеймс Р. Грофф, Пол Н. Вайнберг, Эндрю Дж. Оппель. – 3-е изд. – М. : Вильямс, 2015. – 960 с.
2. Лузанов, П. PostgreSQL для начинающих / П. Лузанов, Е. Рогов, И. Лёвшин ; Postgres Professional. – М., 2017. – 146 с.
3. Моргунов, Е. П. Язык SQL. Базовый курс : учеб.-практ. пособие. / Е. П. Моргунов ; под ред. Е. В. Рогова, П. В. Лузанова ; Postgres Professional. – М., 2017. – 257 с.
4. PostgreSQL [Электронный ресурс] : официальный сайт / The PostgreSQL Global Development Group. – <http://www.postgresql.org>.
5. Postgres Professional [Электронный ресурс] : российский производитель СУБД Postgres Pro : официальный сайт / Postgres Professional. – <http://postgrespro.ru>.
6. Учебные курсы по администрированию PostgreSQL / Е. В. Рогов, П. В. Лузанов ; Postgres Professional. – <https://postgrespro.ru/education/courses>.

Для выполнения практических заданий необходимо использовать книгу:

Моргунов, Е. П. Язык SQL. Базовый курс : учеб.-практ. пособие / Под ред. Е. В. Рогова, П. В. Лузанова ; Postgres Professional. – М., 2017. – 257 с.

<https://postgrespro.ru/education/books/sqlprimer>

1. Изучить материал главы 10. Запросы к базе данных выполнять с помощью утилиты `psql`, описанной в главе 2, параграф 2.2.