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

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

Темы

Пути оптимизации

Статистика

Настройки, влияющие на планирование и выполнение

Схема данных

Физическое расположение данных

Изменение запросов

Пути оптимизации

Цель оптимизации — получить адекватный план

Исправление неэффективностей

- каким-то образом найти и исправить узкое место
- бывает сложно определить, в чем проблема
- часто приводит к борьбе с планировщиком

Правильный расчет кардинальности

- добиться правильного расчета кардинальности в каждом узле и положиться на планировщик
- если план все еще неадекватный, настраивать глобальные параметры

Цель оптимизации запроса — получить адекватный план выполнения. Есть разные пути, которыми можно идти к этой цели.

Можно посмотреть в план запроса, понять причину неэффективного выполнения и сделать что-то, исправляющее ситуацию. К сожалению, проблема не всегда бывает очевидной, а исправление часто сводится к борьбе с оптимизатором.

Если идти таким путем, то хочется иметь возможность целиком или частично отключить планировщик и самому создать план выполнения. Такая возможность называется *подсказками* (хинтами) и в явном виде отсутствует в PostgreSQL.

Другой подход состоит в том, чтобы добиться корректного расчета кардинальности в каждом узле плана. Для этого, конечно, нужна аккуратная статистика, но этого часто бывает недостаточно.

Если идти таким путем, то мы не боремся с планировщиком, а помогаем ему принять верное решение. К сожалению, это часто оказывается слишком сложной задачей.

Если при правильно оцененной кардинальности планировщик все равно строит неэффективный план, это повод заняться настройкой глобальных конфигурационных параметров.

Обычно имеет смысл применять оба способа, смотря по ситуации и сообразуясь со здравым смыслом.

Дальше мы рассмотрим некоторые возможные приемы оптимизации.

Статистика

Актуальность

настройка автоочистки и автоанализа

autovacuum_max_workers, autovacuum_analyze_scale_factor, …

Точность

default_statistics_target = 100

индекс по выражению

расширенная статистика (например, для коррелированных предикатов)

Использование имеющейся статистики

планировщик не всегда может сделать правильные выводы из имеющихся данных

переформулирование запроса, временные таблицы

В первую очередь стоит еще раз напомнить, что первый шаг к адекватному плану — актуальная статистика. Для этого статистика должна собираться достаточно часто, а достигается это настройкой автоочистки и автоанализа. Детали настройки подробно рассматриваются в курсе DBA2.

Кроме того, статистика должна быть достаточно точной. Абсолютной точности достичь не получится (и не нужно), но погрешности не должны приводить к построению некорректных планов. Признаком неактуальной, неточной статистики будет серьезное несоответствие ожидаемого и реального числа строк в листовых узлах плана.

Для увеличения точности может потребоваться изменить значение *default_statistics_target* (глобально или для отдельных столбцов таблиц). Иногда может оказаться полезным индекс по выражению, обладающий собственной статистикой. В отдельных случаях можно использовать расширенную статистику.

Наличие точной актуальной статистики необходимо, но не достаточно для построения хорошего плана. Планировщик может не суметь сделать правильные выводы из имеющейся статистики; часто ошибки связаны с неверным расчетом селективности соединений или агрегаций.

Иногда можно немного переформулировать запрос, чтобы исправить ситуацию. Может помочь вынесение части запроса во временную таблицу, чтобы следующий этап учитывал получившееся количество строк, хотя это и ограничивает свободу планировщика и чревато накладными расходами.

Статистика

Сразу отключим параллельное выполнение. В виртуальной машине с одним ядром оно не поможет, а параллельные планы сложнее читать.

```
=> SET max_parallel_workers_per_gather = 0;
```

```
SET
```

Простой пример запроса, который не позволяет использовать имеющуюся статистику из-за вызова функции. Мы пытаемся выбрать все рейсы за июнь 2017 года:

```
=> EXPLAIN (analyze, timing off, summary off) SELECT *
FROM flights
WHERE date_trunc('month',scheduled_departure) = '2017-06-01';
```

```
                                                   QUERY PLAN                                                    
-----------------------------------------------------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5847.00 rows=1074 width=63) (actual rows=16235 loops=1)
   Filter: (date_trunc('month'::text, scheduled_departure) = '2017-06-01 00:00:00+03'::timestamp with time zone)
   Rows Removed by Filter: 198632
(3 rows)
```

Обратите внимание на разницу между прогнозируемой и фактической кардинальностью.

В теме «Статистика» для исправления статистики мы воспользовались индексом по выражению. Но в данном случае достаточно переписать запрос:

```
=> EXPLAIN (analyze, timing off, summary off) SELECT *
FROM flights
WHERE scheduled_departure >= '2017-06-01'
  AND scheduled_departure <  '2017-07-01';
```

```
                                                                                QUERY PLAN                                                                                
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5847.00 rows=16679 width=63) (actual rows=16235 loops=1)
   Filter: ((scheduled_departure >= '2017-06-01 00:00:00+03'::timestamp with time zone) AND (scheduled_departure < '2017-07-01 00:00:00+03'::timestamp with time zone))
   Rows Removed by Filter: 198632
(3 rows)
```

В таком виде запрос сможет использовать и обычный индекс по столбцу scheduled_departure, если его создать.

Настройки стоимости

Ввод-вывод

можно (и нужно) указывать на уровне табличных пространств

seq_page_cost = 1.0

random_page_cost = 4.0 ↓ для SSD

effective_io_concurrency = 1

Имеется большое число настроек, которые позволяют задать стоимости элементарных операций, из которых, как мы уже видели, в итоге складывается стоимость плана запроса. Такие настройки имеет смысл изменять, если запрос, в котором планировщик точно спрогнозировал кардинальности, тем не менее выполняется не самым эффективным образом.

С вводом-выводом связаны настройки, задающие веса для «условных единиц», в которых выражается стоимость.

Это параметры *seq_page_cost* и *random_page_cost,* определяющие стоимость чтения одной страницы при последовательном доступе и при произвольном доступе.

Значение *seq_page_cost* равно единице и его не стоит изменять. Высокое значение параметра *random_page_cost* отражает реалии HDD-дисков. Значение этого параметра необходимо значительно уменьшать для SSD-дисков (а также в случаях, когда все данные с большой вероятностью будут закешированы).

Параметр *effective_io_concurrency* можно увеличить до числа независимых дисков в дисковом массиве. Фактически этот параметр влияет только на количество страниц, которые будут предварительно считаны в кеш при сканировании по битовой карте.

https://postgrespro.ru/docs/postgresql/13/runtime-config-query

Настройки стоимости

Время процессора

cpu_tuple_cost	= 0.01
cpu_index_tuple_cost	= 0.005
cpu_operator_cost	= 0.0025

CREATE FUNCTION … COST *стоимость*

Параметры для времени процессора определяют веса для учета времени обработки прочитанных данных. Обычно этот вклад меньше стоимости ввода-вывода, но не всегда. Можно указать стоимость обработки одной табличной строки (*cpu_tuple_cost*), одной строки индекса (*cpu_index_tuple_cost*) и одной операции (*cpu_operator_cost;* например, операции сравнения).

Также можно задать стоимость пользовательской функции в единицах *cpu_operator_cost.* По умолчанию функции на Си получают оценку 1, а на других языках — 100.

Настройки стоимости

Параллельное выполнение

parallel_setup_cost = 1000
parallel_tuple_cost = 0.1

Курсоры

cursor_tuple_fraction = 0.1

Параметр *parallel_setup_cost* указывает стоимость развертывания инфраструктуры для параллельной обработки: выделение общей памяти и порождение рабочих процессов. Эта величина добавляется к общей стоимости запроса. Параметр *parallel_tuple_cost* определяет стоимость пересылки одной строки данных от процесса к процессу.

Остальные параметры, относящиеся к параллельному выполнению, обсуждались ранее (они ограничивают количество параллельных процессов и устанавливают минимальный размер выборки, при которой планировщик рассматривает параллельные планы).

Напомним также, что оценка стоимости состоит из двух компонент: cost1..cost2. При обычном выполнении запросов планировщик выбирает план, минимизируя cost2, то есть оценку ресурсов для получения полной выборки данных. А при использовании курсоров учитывается значение параметра *cursor_tuple_fraction*: чем меньше значение этой доли, тем сильнее планировщик ориентируется на быстрое получение первых результатов. Говоря точнее, минимизируется значение cost1 + (cost2 − cost1) × *cursor_tuple_fraction.*

Настройки стоимости

Пример запроса с корректной оценкой кардинальности:

```
=> EXPLAIN (analyze, timing off) SELECT *
FROM bookings
WHERE book_ref < '9000';
```

```
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Seq Scan on bookings  (cost=0.00..39835.88 rows=1202926 width=21) (actual rows=1187454 loops=1)
   Filter: (book_ref < '9000'::bpchar)
   Rows Removed by Filter: 923656
 Planning Time: 0.840 ms
 Execution Time: 501.773 ms
(5 rows)
```

Планировщик выбрал последовательное сканирование, и сделал это на основе полной информации.

Так ли удачно это решение? Проверим, отключив возможность последовательного сканирования.

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

```
SET
```

```
=> EXPLAIN (analyze, timing off) SELECT *
FROM bookings
WHERE book_ref < '9000';
```

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Index Scan using bookings_pkey on bookings  (cost=0.43..41921.64 rows=1202926 width=21) (actual rows=1187454 loops=1)
   Index Cond: (book_ref < '9000'::bpchar)
 Planning Time: 0.209 ms
 Execution Time: 213.747 ms
(4 rows)
```

Результат, скорее всего, окажется в пользу индексного сканирования, поскольку все данные закешированы и произвольный доступ выполняется быстро. Такая же ситуация возможна при использовании быстрых SSD-дисков. Делать выводы на основании одного запроса неправильно, но систематическая ошибка должна послужить поводом к изменению глобальных настроек. В данном случае стоит уменьшить значение random_page_cost, чтобы планировщик не завышал стоимость индексного доступа.

```
=> RESET enable_seqscan;
```

```
RESET
```

Настройки памяти

Память

work_mem = 4MB ↑
hash_mem_multiplier = 1
maintenance_work_mem = 64MB
effective_cache_size = 4GB

Ряд настроек влияет на выделение памяти. Параметр *work_mem* определяет размер доступной памяти для отдельных операций (рассматривался в соответствующих темах). Также он влияет и на выбор плана. При небольших значениях предпочтение отдается сортировке (а не хешированию), поскольку ее алгоритм менее чувствителен к недостатку памяти.

Для узлов, использующих хеширование, размер рабочей памяти можно увеличить с помощью мультипликатора — параметра *hash_mem_multiplier*.

Параметр *maintenance_work_mem* влияет на скорость построения индексов и на работу служебных процессов.

Параметр *effective_cache_size* подсказывает PostgreSQL общий объем кешируемых данных (как в буферном кеше, так и в кеше ОС). Чем он больше, тем более предпочтительным будет индексный доступ. Этот параметр не влияет на реальное выделение памяти.

Локализация настроек

На уровне базы данных

ALTER DATABASE SET …

На уровне роли и базы данных

ALTER ROLE [IN DATABASE …] SET …

На уровне процедуры или функции

ALTER ROUTINE SET …

На уровне сеанса или транзакции

SET [LOCAL] …

Обычно рассмотренные параметры устанавливаются в файлах postgresql.conf или postgresql.auto.conf и влияют на всю систему.

Однако есть возможность ограничить влияние параметров, устанавливая их на других уровнях.

Часть параметров, касающихся ввода-вывода, можно установить для табличных пространств.

Остальные параметры можно устанавливать для отдельной базы данных или отдельной роли (или совокупности роли и базы данных). Можно также устанавливать параметры на уровне процедур и функций, что позволяет изолировать настройки отдельного запроса и менять их без вмешательства в исходный код.

Часть параметров можно устанавливать на уровне отдельного сеанса или транзакции.

Локализация настроек

В ряде случаев может оказаться удобным оформить запросы в виде хранимых подпрограмм (например, с целью предоставить к ним доступ приложению). В этом случае дополнительным преимуществом может быть возможность установки параметров для конкретных подпрограмм.

Вот пример функции, возвращающей имена всех пассажиров с номерами рейсов за месяц в порядке возрастания даты вылета:

```
=> CREATE FUNCTION get_passengers_and_flights(d timestamptz)
RETURNS TABLE(passenger_name text, flight_no text)
AS $$
  SELECT t.passenger_name, f.flight_no
  FROM tickets t
    JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
    JOIN flights f ON f.flight_id = tf.flight_id
  WHERE f.scheduled_departure >= date_trunc('month', d)
    AND f.scheduled_departure  < date_trunc('month', d) + interval '1 month'
  ORDER BY f.scheduled_departure, t.passenger_name;
$$ LANGUAGE sql;

CREATE FUNCTION
```

Такая функция будет работать медленно из-за необходимости сортировки большого объема данных:

```
=> \timing on

Timing is on.

=> SELECT * FROM get_passengers_and_flights('2017-06-01') LIMIT 3;

    passenger_name    | flight_no
----------------------+-----------
 ALBERT EGOROV        | PG0328
 ALEKSANDRA KUZNECOVA | PG0328
 ALEKSANDR BORISOV    | PG0328
(3 rows)

Time: 14411,853 ms (00:14,412)
```

Ситуацию можно немного улучшить, увеличив объем рабочей памяти для этого конкретного запроса:

```
=> ALTER FUNCTION get_passengers_and_flights SET work_mem = '32MB';

ALTER FUNCTION
Time: 14,079 ms

=> SELECT * FROM get_passengers_and_flights('2017-06-01') LIMIT 3;

    passenger_name    | flight_no
----------------------+-----------
 ALBERT EGOROV        | PG0328
 ALEKSANDRA KUZNECOVA | PG0328
 ALEKSANDR BORISOV    | PG0328
(3 rows)

Time: 10524,037 ms (00:10,524)

=> \timing off

Timing is off.
```

Отладка

Отключение определенных способов доступа

enable_seqscan
enable_indexscan, *enable_bitmapscan*, *enable_indexonlyscan*

Отключение определенных методов соединений

enable_nestloop, *enable_hashjoin*, *enable_mergejoin*

Отключение определенных операций

enable_hashagg, *enable_sort*, *enable_incremental_sort*

Проверка возможности распараллеливания

force_parallel_mode

Ряд параметров могут использоваться для запрещения определенных методов доступа, способов соединений и других операций. Установка этих параметров в значение off не запрещает операции, но устанавливает им очень большую стоимость. Таким образом, планировщик будет пытаться обойтись без них, но может применить в безвыходной ситуации.

Эти параметры оказывают достаточно грубое влияние на планировщик, но оказываются весьма полезны для отладки и экспериментов. Точечно их иногда можно применять как аналог подсказок планировщику.

На слайде показаны не все параметры этого семейства.

Параметр *force_parallel_mode* никогда не применяется при реальном выполнении запросов, но полезен для проверки возможности распараллеливания.

Схема данных

Нормализация — устранение избыточности в данных

- упрощает запросы и проверку согласованности

Денормализация — привнесение избыточности

- может повысить производительность, но требует синхронизации
- индексы
- предрассчитанные поля (генерируемые столбцы или триггеры)
- материализованные представления
- кеширование результатов в приложении

На логическом уровне база данных должна быть нормализованной: неформально, в хранимых данных не должно быть избыточности. Если это не так, мы имеем дело с ошибкой проектирования: будет сложно проверять согласованность данных, возможны различные аномалии при изменении данных и т. п.

Однако на уровне хранения некоторое дублирование может дать существенный выигрыш в производительности — ценой того, что избыточные данные необходимо синхронизировать с основными.

Самый частый способ денормализации — индексы (хотя о них обычно не думают в таком контексте). Индексы обновляются автоматически.

Можно дублировать некоторые данные (или результаты расчета на основе этих данных) в столбцах таблиц. Можно использовать генерируемые столбцы или синхронизировать данные с помощью триггеров. Так или иначе, за денормализацию отвечает базе данных.

Другой пример — материализованные представления. Их также надо обновлять, например, по расписанию или другим способом.

Дублировать данные можно и на уровне приложения, кешируя результаты выполнения запросов. Это популярный способ, но часто к нему прибегают из-за неправильной работы приложения с базой данных (например, в случае использования ORM). На приложение ложится задача по своевременному обновлению кеша, обеспечению разграничения доступа к данным кеша и т. п.

Схема данных

Типы данных

- выбор подходящих типов
- составные типы (массивы, JSON) вместо отдельных таблиц

Ограничения целостности

- помимо обеспечения целостности данных, могут учитываться планировщиком для устранения ненужных соединений, улучшения оценок селективности и других оптимизаций
- первичный ключ и уникальность — уникальный индекс
- внешний ключ
- отсутствие неопределенных значений
- проверка CHECK (*constraint_exclusion*)

Важен правильный выбор **типов данных** из всего многообразия, предлагаемого PostgreSQL. Например, представление интервалов дат не двумя отдельными столбцами, а с помощью диапазонных типов (daterange, tstzrange) позволяет использовать индексы GiST и SP-GiST для таких операций, как пересечение интервалов.

В ряде случаев эффект дает использование составных типов (таких как массивы или JSON) вместо классического подхода с созданием отдельной таблицы. Это позволяет сэкономить на соединении и не требует хранения большого количества служебной информации в заголовках версий строк. Но такое решение следует принимать с большой осторожностью, поскольку оно имеет свои минусы.

Ограничения целостности важны сами по себе, но в некоторых случаях планировщик может учитывать их и для оптимизации.

- Ограничения первичного ключа и уникальности сопровождаются построением уникальных индексов и гарантируют, что все значения столбцов ключа различны (точная статистика). Такие гарантии позволяют и более эффективно выполнять соединения.

- Наличие внешнего ключа и ограничений NOT NULL дает гарантии, которые позволяют в ряде случаев устранять лишние внутренние соединения (что особенно важно при использовании представлений), а также улучшает оценку селективности в случае, если соединение происходит по нескольким столбцам.

- Ограничение CHECK с использованием параметра *constraint_exclusion* позволяет не сканировать таблицы (или секции), если в них гарантированно нет требуемых данных.

Схема данных

Ссылочное ограничение целостности особенно важно при составном ключе. Пример точной оценки строк, которое будет получено в результате соединения:

```
=> EXPLAIN SELECT *
FROM ticket_flights tf
  JOIN boarding_passes bp ON tf.flight_id = bp.flight_id
                         AND tf.ticket_no = bp.ticket_no;

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Hash Join  (cost=310609.60..677434.34 rows=7925944 width=57)
   Hash Cond: ((tf.flight_id = bp.flight_id) AND (tf.ticket_no = bp.ticket_no))
   ->  Seq Scan on ticket_flights tf  (cost=0.00..153851.52 rows=8391852 width=32)
   ->  Hash  (cost=137538.44..137538.44 rows=7925944 width=25)
         ->  Seq Scan on boarding_passes bp  (cost=0.00..137538.44 rows=7925944 width=25)
(5 rows)
```

Однако если удалить внешний ключ, оценка становится неадекватной. Это еще одно проявление проблемы коррелированных предикатов:

```
=> ALTER TABLE boarding_passes
DROP CONSTRAINT boarding_passes_ticket_no_fkey;

ALTER TABLE

=> EXPLAIN SELECT *
FROM ticket_flights tf
  JOIN boarding_passes bp ON tf.flight_id = bp.flight_id
                         AND tf.ticket_no = bp.ticket_no;

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Hash Join  (cost=310609.60..677434.34 rows=311 width=57)
   Hash Cond: ((tf.flight_id = bp.flight_id) AND (tf.ticket_no = bp.ticket_no))
   ->  Seq Scan on ticket_flights tf  (cost=0.00..153851.52 rows=8391852 width=32)
   ->  Hash  (cost=137538.44..137538.44 rows=7925944 width=25)
         ->  Seq Scan on boarding_passes bp  (cost=0.00..137538.44 rows=7925944 width=25)
(5 rows)
```

Физическое расположение

Табличные пространства

разнесение данных по разным физическим устройствам

Секционирование

разделение таблицы на отдельно управляемые части
для упрощения администрирования и ускорения доступа

Шардирование

размещение секций на разных серверах для масштабирования
нагрузки на чтение и запись

секционирование и расширение postgres-fdw
или сторонние решения

Физическая организация данных может сильно влиять на производительность. К возможностям такой организации можно отнести распределение объектов по табличным пространствам и секционирование.

Табличные пространства можно использовать для того, чтобы управлять размещением объектов по физическим устройствам ввода-вывода. Например, активно используемые данные хранить на SSD-дисках, а архивные — на более медленных HDD.

Секционирование позволяет организовать работу с данными очень большого объема. Основная выгода для производительности состоит в замене полного сканирования всей таблицы сканированием отдельной секции. Заметим, что секции тоже можно размещать по различным табличным пространствам.

Еще один вариант — размещение секций на разных серверах (шардирование) с возможностью выполнения распределенных запросов. Стандартный PostgreSQL содержит только базовые механизмы, необходимые для организации шардирования: секционирование и расширение postgres-fdw. Более эффективно и полноценно шардинг реализуется с помощью внешних решений. Обзорно этот вопрос рассматривается в последней теме курса DBA3.

https://postgrespro.ru/docs/postgresql/13/ddl-partitioning

https://postgrespro.ru/docs/postgresql/13/postgres-fdw

Порядок соединений

Автоматический выбор порядка соединений

- если число соединяемых таблиц не превышает *join_collapse_limit* — планировщик выбирает лучший порядок соединений
- при большем количестве рассматриваются не все варианты

Ручное управление порядком соединений

- материализация подзапросов
  с помощью CTE или временных таблиц
- явные соединения (JOIN) и *join_collapse_limit* = 1
- подзапросы в предложении FROM и *from_collapse_limit* = 1

Число различных возможных планов выполнения запроса растет экспоненциально с ростом числа соединений. Начиная с некоторого момента (как правило, когда количество соединяемых таблиц превышает значение *join_collapse_limit*) планировщик вынужден сокращать пространство поиска, чтобы не тратить слишком много времени и оперативной памяти. В таком случае за качество планирования нельзя поручиться, и, скорее всего, потребуется ручное вмешательство.

Подробнее об этом можно прочитать в статье Павла Толмачева: https://habr.com/ru/company/postgrespro/blog/662021/

Наиболее удобным способом влияния является разделение таблиц на группы, каждая из которых будет планироваться независимо. Автор запроса может выделить такие группы с помощью материализации общих табличных выражений (или с помощью временных таблиц, хотя такой способ обычно менее предпочтителен).

Другой способ — указать планировщику, что таблицы должны соединяться в том порядке, в котором они синтаксически перечислены в запросе. Для этого надо установить параметр *join_collapse_limit* = 1 и использовать явные соединения с помощью ключевого слова JOIN. Аналогично и подзапросы не будут раскрываться при значении *from_collapse_limit* = 1. Однако такой способ перекладывает на автора запроса слишком много ответственности. Планировщик сможет выбирать только то, какой из наборов строк поставить в соединении внешним, а какой — внутренним.

Материализация CTE

Подзапросы CTE могут быть материализованы. Материализацией управляет планировщик, но такое поведение можно гарантировать, указав ключевое слово MATERIALIZED. В этом случае подзапрос не раскрывается, его результат вычисляется и помещается либо в оперативную память (в пределах work_mem), либо сбрасывается во временный файл.

Выполним следующий запрос, в котором используется CTE:

```
=> EXPLAIN (costs off)
WITH b AS (
  SELECT * FROM bookings
)
SELECT * FROM b
WHERE b.book_ref = '000112';
```

```
                 QUERY PLAN
-------------------------------------------
 Index Scan using bookings_pkey on bookings
   Index Cond: (book_ref = '000112'::bpchar)
(2 rows)
```

В этом случае планировщик не материализует табличное выражение — оно раскрывается, что дает возможность использовать условие для индексного доступа.

Теперь выполним тот же запрос, но с материализацией:

```
=> EXPLAIN (costs off)
WITH b AS MATERIALIZED (
  SELECT * FROM bookings
)
SELECT * FROM b
WHERE b.book_ref = '000112';
```

```
               QUERY PLAN
----------------------------------------
 CTE Scan on b
   Filter: (book_ref = '000112'::bpchar)
   CTE b
     ->  Seq Scan on bookings
(4 rows)
```

Теперь сначала вычисляется табличное выражение, и только затем — остальной запрос. В этом случае условие не может попасть внутри табличного выражения и индексный доступ не работает.

Параметр join_collapse_limit

Значение по умолчанию параметра join_collapse_limit выбрано так, чтобы соединение такого количества таблиц не требовало чрезмерных ресурсов:

```
=> SHOW join_collapse_limit;
```

```
 join_collapse_limit
---------------------
 8
(1 row)
```

Выполним запрос, в котором используется явное соединение таблиц с помощью ключевого слова JOIN:

```
=> EXPLAIN (costs on)
SELECT *
FROM tickets t
 JOIN ticket_flights tf ON (tf.ticket_no = t.ticket_no)
 JOIN flights f ON (f.flight_id = tf.flight_id);
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Hash Join  (cost=171645.57..778545.78 rows=8391852 width=199)
   Hash Cond: (tf.ticket_no = t.ticket_no)
   ->  Hash Join  (cost=9767.51..302691.07 rows=8391852 width=95)
         Hash Cond: (tf.flight_id = f.flight_id)
         ->  Seq Scan on ticket_flights tf  (cost=0.00..153851.52 rows=8391852 width=32)
         ->  Hash  (cost=4772.67..4772.67 rows=214867 width=63)
               ->  Seq Scan on flights f  (cost=0.00..4772.67 rows=214867 width=63)
   ->  Hash  (cost=78913.25..78913.25 rows=2949825 width=104)
         ->  Seq Scan on tickets t  (cost=0.00..78913.25 rows=2949825 width=104)
(9 rows)
```

В выбранном плане сначала соединяются таблицы рейсов (flights) и перелетов (ticket_flights), а затем результат соединяется с билетами (tickets).

Установив параметр join_collapse_limit в единицу, можно зафиксировать порядок соединений:

```
=> SET join_collapse_limit = 1;
```

```
SET
```

```
=> EXPLAIN (costs on)
SELECT *
FROM tickets t
 JOIN ticket_flights tf ON (tf.ticket_no = t.ticket_no)
 JOIN flights f ON (f.flight_id = tf.flight_id);
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Hash Join  (cost=171645.57..860497.78 rows=8391852 width=199)
   Hash Cond: (tf.flight_id = f.flight_id)
   ->  Hash Join  (cost=161878.06..498584.23 rows=8391852 width=136)
         Hash Cond: (tf.ticket_no = t.ticket_no)
         ->  Seq Scan on ticket_flights tf  (cost=0.00..153851.52 rows=8391852 width=32)
         ->  Hash  (cost=78913.25..78913.25 rows=2949825 width=104)
               ->  Seq Scan on tickets t  (cost=0.00..78913.25 rows=2949825 width=104)
   ->  Hash  (cost=4772.67..4772.67 rows=214867 width=63)
         ->  Seq Scan on flights f  (cost=0.00..4772.67 rows=214867 width=63)
(9 rows)
```

Теперь таблицы соединяются друг с другом в том порядке, в котором они перечислены в запросе, несмотря на то, что итоговая стоимость запроса выше.

```
=> RESET join_collapse_limit;
```

```
RESET
```

С похожим параметром from_collapse_limit предлагается познакомиться в практике.

Изменение запросов

Альтернативные способы выполнения

- планировщик не всегда рассматривает все возможные трансформации
- раскрытие коррелированных подзапросов
- устранение лишних таблиц
- замена UNION на OR и обратно
- и т. п.

Замена процедурного кода декларативным

- чтобы избавиться от большого числа мелких запросов

Самый разнообразный и вариативный способ влияния на производительность запроса — его переформулирование. Теоретически планировщик должен уметь выполнять эквивалентные преобразования (ведь SQL — декларативный язык), но на практике это возможно в довольно ограниченных пределах.

Иногда требуются ручные изменения:

- Переписывание коррелированных подзапросов на соединения.

- Устранение из запроса лишних сканирований таблиц (например, за счет применения оконных функций).

- Использование недоступных планировщику трансформаций. Например, операции для работы со множествами в настоящее время не трансформируются (UNION в OR и т. п.).

Если приложение выполняет слишком большое количество мелких запросов (каждый из которых сам по себе выполняется эффективно), общая эффективность будет очень невысока.

В таких случаях со стороны СУБД практически нет средств для оптимизации (кроме подготовленных операторов и кеширования). Единственный действенный метод — избавление от процедурного кода в приложении и перенос его на сервер БД в виде небольшого числа крупных SQL-команд. Это дает планировщику возможность применить более эффективные способы доступа и соединений и избавляет от многочисленных пересылок данных от клиента к серверу и обратно. К сожалению, это очень затратный способ.

Пример оптимизации запроса

Рассмотрим теперь пример оптимизации запроса, в котором будут использованы различные рассмотренные ранее методы.

Задача: вычислить среднюю стоимость билетов на перелеты, выполняемые различными типами самолетов. При этом исключить самые дешевые и самые дорогие билеты.

Начинаем со следующего запроса, который решает задачу, но работает медленно:

```
=> \timing on
```

```
Timing is on.
```

```
=> SELECT a.aircraft_code, (
  SELECT round(avg(tf.amount))
  FROM flights f
    JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  WHERE f.aircraft_code = a.aircraft_code
    AND tf.amount > (SELECT min(amount) FROM ticket_flights)
    AND tf.amount < (SELECT max(amount) FROM ticket_flights)
)
FROM aircrafts a
GROUP BY a.aircraft_code;
```

```
 aircraft_code | round
---------------+-------
 319           | 51546
 320           |
 321           | 15666
 733           | 16545
 763           | 34262
 773           | 23783
 CN1           |  6586
 CR2           | 13228
 SU9           | 14265
(9 rows)

Time: 22524,558 ms (00:22,525)
```

```
=> \timing off
```

```
Timing is off.
```

Посмотрим схему плана выполнения:

```
=> EXPLAIN (costs off)
SELECT a.aircraft_code, (
  SELECT round(avg(tf.amount))
  FROM flights f
    JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  WHERE f.aircraft_code = a.aircraft_code
    AND tf.amount > (SELECT min(amount) FROM ticket_flights)
    AND tf.amount < (SELECT max(amount) FROM ticket_flights)
)
FROM aircrafts a
GROUP BY a.aircraft_code;
```

```
                                QUERY PLAN
--------------------------------------------------------------------------
 Group
   Group Key: ml.aircraft_code
   ->  Index Only Scan using aircrafts_pkey on aircrafts_data ml
   SubPlan 3
     ->  Aggregate
           InitPlan 1 (returns $0)
             ->  Aggregate
                   ->  Seq Scan on ticket_flights
           InitPlan 2 (returns $1)
             ->  Aggregate
                   ->  Seq Scan on ticket_flights ticket_flights_1
           ->  Hash Join
                 Hash Cond: (tf.flight_id = f.flight_id)
                 ->  Seq Scan on ticket_flights tf
                       Filter: ((amount > $0) AND (amount < $1))
                 ->  Hash
                       ->  Seq Scan on flights f
                             Filter: (aircraft_code = ml.aircraft_code)
(18 rows)
```

- Основной запрос состоит из сканирования aircrafts_data по индексу и группировки.
- Узел SubPlan соответствует подзапросу, вычисляющему стоимость — он выполняется в цикле.
- Узлы InitPlan соответствуют подзапросам, вычисляющим минимум и максимум — они выполняются один раз (но для каждого выполнения SubPlan, в который они вложены).

Можно заметить, что для вычисления минимума и максимума выполняется полное сканирование таблицы ticket_flights. Создание индекса должно ускорить эту операцию:

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

```
CREATE INDEX
```

```
=> EXPLAIN (analyze, timing off)
SELECT a.aircraft_code, (
  SELECT round(avg(tf.amount))
  FROM flights f
    JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  WHERE f.aircraft_code = a.aircraft_code
    AND tf.amount > (SELECT min(amount) FROM ticket_flights)
    AND tf.amount < (SELECT max(amount) FROM ticket_flights)
)
FROM aircrafts a
GROUP BY a.aircraft_code;
```

```
                                                                                           QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Group  (cost=0.14..635571.89 rows=9 width=36) (actual rows=9 loops=1)
   Group Key: ml.aircraft_code
   ->  Index Only Scan using aircrafts_pkey on aircrafts_data ml  (cost=0.14..12.27 rows=9 width=4) (actual rows=9 loops=1)
         Heap Fetches: 9
   SubPlan 5
     ->  Aggregate  (cost=70617.72..70617.73 rows=1 width=32) (actual rows=1 loops=9)
           InitPlan 2 (returns $1)
             ->  Result  (cost=0.46..0.47 rows=1 width=32) (actual rows=1 loops=1)
                   InitPlan 1 (returns $0)
                     ->  Limit  (cost=0.43..0.46 rows=1 width=6) (actual rows=1 loops=1)
                           ->  Index Only Scan using ticket_flights_amount_idx on ticket_flights  (cost=0.43..239033.85 rows=8391852 width=6) (actual rows=1 loops=1)
                                 Index Cond: (amount IS NOT NULL)
                                 Heap Fetches: 0
           InitPlan 4 (returns $3)
             ->  Result  (cost=0.46..0.47 rows=1 width=32) (actual rows=1 loops=1)
                   InitPlan 3 (returns $2)
                     ->  Limit  (cost=0.43..0.46 rows=1 width=6) (actual rows=1 loops=1)
                           ->  Index Only Scan Backward using ticket_flights_amount_idx on ticket_flights ticket_flights_1  (cost=0.43..239033.85 rows=8391852 width=6) (actual rows=1 l
                                 Index Cond: (amount IS NOT NULL)
                                 Heap Fetches: 0
           ->  Hash Join  (cost=6540.08..70603.66 rows=5245 width=6) (actual rows=925529 loops=9)
                 Hash Cond: (tf.flight_id = f.flight_id)
                 ->  Bitmap Heap Scan on ticket_flights tf  (cost=894.51..64847.95 rows=41959 width=10) (actual rows=8329761 loops=8)
                       Recheck Cond: ((amount > $1) AND (amount < $3))
                       Rows Removed by Index Recheck: 29618
                       Heap Blocks: exact=295184 lossy=264280
                       ->  Bitmap Index Scan on ticket_flights_amount_idx  (cost=0.00..884.02 rows=41959 width=0) (actual rows=8329761 loops=8)
                             Index Cond: ((amount > $1) AND (amount < $3))
                 ->  Hash  (cost=5309.84..5309.84 rows=26858 width=4) (actual rows=23874 loops=9)
                       Buckets: 65536 (originally 32768)  Batches: 1 (originally 1)  Memory Usage: 2629kB
                       ->  Seq Scan on flights f  (cost=0.00..5309.84 rows=26858 width=4) (actual rows=23874 loops=9)
                             Filter: (aircraft_code = ml.aircraft_code)
                             Rows Removed by Filter: 190993
 Planning Time: 8.074 ms
 Execution Time: 46567.284 ms
(35 rows)
```

Все только ухудшилось. Почему?

Теперь все три обращения к таблице ticket_flights выполняются с помощью индекса. Но в случае tf планировщик сильно ошибается с оценкой кардинальности: вместо ожидаемых 40 тыс строк читаются практически все строки из 8 млн. Очевидно, что использовать для этого индекс неэффективно.

Запретить индексный доступ можно разными средствами. Например, переписать условие так, чтобы индекс не мог применяться (добавить 0 к amount слева от оператора сравнения).

Но лучше исправить оценку планировщика. Сейчас он не знает значений, которые будут выбраны подзапросами, потому оценивает «из общих соображений». Но значения можно вычислить заранее и подставить в основной запрос.

Чтобы получить согласованные данные, используем уровень изоляции Repeatable Read:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
=> SELECT max(amount) AS a_max, min(amount) AS a_min
FROM ticket_flights \gset
```

```
=> EXPLAIN (analyze, timing off)
SELECT a.aircraft_code, (
  SELECT round(avg(tf.amount))
  FROM flights f
    JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  WHERE f.aircraft_code = a.aircraft_code
    AND tf.amount > :a_min
    AND tf.amount < :a_max
)
FROM aircrafts a
GROUP BY a.aircraft_code;
```

```
                                                          QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------
 Group  (cost=0.14..2032981.77 rows=9 width=36) (actual rows=9 loops=1)
   Group Key: ml.aircraft_code
   ->  Index Only Scan using aircrafts_pkey on aircrafts_data ml  (cost=0.14..12.27 rows=9 width=4) (actual rows=9 loops=1)
         Heap Fetches: 9
   SubPlan 1
     ->  Aggregate  (cost=225885.48..225885.50 rows=1 width=32) (actual rows=1 loops=9)
           ->  Hash Join  (cost=5645.56..223286.72 rows=1039506 width=6) (actual rows=925529 loops=9)
                 Hash Cond: (tf.flight_id = f.flight_id)
                 ->  Seq Scan on ticket_flights tf  (cost=0.00..195810.78 rows=8316168 width=10) (actual rows=8329761 loops=8)
                       Filter: ((amount > 3000.00) AND (amount < 203300.00))
                       Rows Removed by Filter: 62091
                 ->  Hash  (cost=5309.84..5309.84 rows=26858 width=4) (actual rows=23874 loops=9)
                       Buckets: 65536 (originally 32768)  Batches: 1 (originally 1)  Memory Usage: 2629kB
                       ->  Seq Scan on flights f  (cost=0.00..5309.84 rows=26858 width=4) (actual rows=23874 loops=9)
                             Filter: (aircraft_code = ml.aircraft_code)
                             Rows Removed by Filter: 190993
 Planning Time: 4.737 ms
 Execution Time: 18890.170 ms
(18 rows)
```

Теперь оценки кардинальности исправились и планировщик сам предпочел полное сканирование таблицы tf.

Однако запрос все еще выполняется не самым эффективным образом. Мы читаем одни и те же таблицы несколько раз в цикле из-за того, что сумма вычисляется в коррелированном подзапросе. Планировщик не может раскрыть его самостоятельно, но мы можем сделать это вручную. Обратите внимание на необходимость левых соединений.

```
=> EXPLAIN (analyze, timing off)
SELECT a.aircraft_code, round(avg(tf.amount))
FROM aircrafts a
  LEFT JOIN flights f ON f.aircraft_code = a.aircraft_code
  LEFT JOIN ticket_flights tf ON tf.flight_id = f.flight_id
    AND tf.amount > :a_min
    AND tf.amount < :a_max
GROUP BY a.aircraft_code;
```

```
                                                         QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=381685.36..381685.49 rows=9 width=36) (actual rows=9 loops=1)
   Group Key: ml.aircraft_code
   Batches: 1  Memory Usage: 24kB
   ->  Hash Right Join  (cost=8299.71..340104.52 rows=8316168 width=10) (actual rows=8394051 loops=1)
         Hash Cond: (f.aircraft_code = ml.aircraft_code)
         ->  Hash Right Join  (cost=8298.51..307993.66 rows=8316168 width=10) (actual rows=8394050 loops=1)
               Hash Cond: (tf.flight_id = f.flight_id)
               ->  Seq Scan on ticket_flights tf  (cost=0.00..195810.78 rows=8316168 width=10) (actual rows=8329761 loops=1)
                     Filter: ((amount > 3000.00) AND (amount < 203300.00))
                     Rows Removed by Filter: 62091
               ->  Hash  (cost=4772.67..4772.67 rows=214867 width=8) (actual rows=214867 loops=1)
                     Buckets: 131072  Batches: 4  Memory Usage: 3127kB
                     ->  Seq Scan on flights f  (cost=0.00..4772.67 rows=214867 width=8) (actual rows=214867 loops=1)
         ->  Hash  (cost=1.09..1.09 rows=9 width=4) (actual rows=9 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on aircrafts_data ml  (cost=0.00..1.09 rows=9 width=4) (actual rows=9 loops=1)
 Planning Time: 0.433 ms
 Execution Time: 8073.575 ms
(18 rows)
```

Вот теперь запрос выполняется эффективно, необходимые данные читаются только один раз.

Можно еще немного улучшить результат, увеличив значение параметра work_mem, чтобы хеш-соединение таблиц flights и ticket_flights выполнялось за один проход без временных файлов.

```
=> SET work_mem = '16MB';
```

```
SET
```

```
=> EXPLAIN (analyze, costs off, timing off, buffers) SELECT a.aircraft_code, round(avg(tf.amount))
FROM aircrafts a
  LEFT JOIN flights f ON f.aircraft_code = a.aircraft_code
  LEFT JOIN ticket_flights tf ON tf.flight_id = f.flight_id
    AND tf.amount > :a_min
    AND tf.amount < :a_max
GROUP BY a.aircraft_code;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 HashAggregate (actual rows=9 loops=1)
   Group Key: ml.aircraft_code
   Batches: 1  Memory Usage: 24kB
   Buffers: shared hit=16281 read=56277
   ->  Hash Right Join (actual rows=8394051 loops=1)
         Hash Cond: (f.aircraft_code = ml.aircraft_code)
         Buffers: shared hit=16281 read=56277
         ->  Hash Right Join (actual rows=8394050 loops=1)
               Hash Cond: (tf.flight_id = f.flight_id)
               Buffers: shared hit=16280 read=56277
               ->  Seq Scan on ticket_flights tf (actual rows=8329761 loops=1)
                     Filter: ((amount > 3000.00) AND (amount < 203300.00))
                     Rows Removed by Filter: 62091
                     Buffers: shared hit=13656 read=56277
               ->  Hash (actual rows=214867 loops=1)
                     Buckets: 262144  Batches: 1  Memory Usage: 10442kB
                     Buffers: shared hit=2624
                     ->  Seq Scan on flights f (actual rows=214867 loops=1)
                           Buffers: shared hit=2624
         ->  Hash (actual rows=9 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               Buffers: shared hit=1
               ->  Seq Scan on aircrafts_data ml (actual rows=9 loops=1)
                     Buffers: shared hit=1
 Planning:
   Buffers: shared hit=16 read=4
 Planning Time: 2.610 ms
 Execution Time: 7137.125 ms
(28 rows)
```

На этом можно остановиться, поскольку запрос не делает лишних чтений:

```
=> SELECT sum(relpages) FROM pg_class
WHERE relname IN ('aircrafts','flights','ticket_flights');
```

```
  sum
-------
 72557
(1 row)
```

Дальнейших путей для оптимизации запроса не видно (хотя и остаются пути для оптимизации настроек сервера, что является темой курса DBA2).

```
=> END;
```

```
COMMIT
```

Подсказки оптимизатору

Отсутствуют в явном виде

- хотя средства влияния имеются: конфигурационные параметры, материализация CTE и другие

Сторонние расширения

- pg_hint_plan

Еще один (традиционный для других СУБД) способ влияния — подсказки оптимизатору — отсутствует в PostgreSQL: Это принципиальное решение сообщества: https://wiki.postgresql.org/wiki/OptimizerHintsDiscussion.

На самом деле часть подсказок все-таки неявно существует в виде конфигурационных параметров и других средств.

Кроме того, есть специальные расширения, например https://postgrespro.ru/docs/enterprise/13/pg-hint-plan (автор — Киотаро Хоригучи). Но нельзя забывать, что использование подсказок, сильно ограничивающих свободу планировщика, может навредить в будущем, когда распределение данных изменится.

Итоги

Доступен широкий спектр методов влияния на план выполнения запросов

Не все методы применимы во всех случаях

Методы, оказывающие глобальное влияние, следует применять с осторожностью

Ничто не заменит голову и здравый смысл

Практика

1. Оптимизируйте запрос, выводящий контактную информацию пассажиров, летевших бизнес-классом, рейсы которых были задержаны более чем на 5 часов. Начните с варианта, предложенного в комментарии.
2. Проверьте план выполнения запроса, приведенного в комментарии, при значении параметра *from_collapse_limit* по умолчанию и при значении, равном единице.

1.

```
SELECT t.*
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
  JOIN flights f ON f.flight_id = tf.flight_id
WHERE tf.fare_conditions = 'Business'
  AND f.actual_departure >
      f.scheduled_departure + interval '5 hour';
```

2.

```
SELECT *
FROM
  (
    SELECT *
    FROM ticket_flights tf
      JOIN tickets t ON tf.ticket_no = t.ticket_no
  ) ttf
  JOIN flights f ON f.flight_id = ttf.flight_id;
```

1. Оптимизация запроса

Отключим параллельное выполнение.

```
=> SET max_parallel_workers_per_gather = 0;
```

```
SET
```

```
=> EXPLAIN (analyze, timing off)
SELECT t.*
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
  JOIN flights f ON f.flight_id = tf.flight_id
WHERE tf.fare_conditions = 'Business'
  AND f.actual_departure > f.scheduled_departure + interval '5 hour';
```

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Merge Join  (cost=215411.95..367011.40 rows=292315 width=104) (actual rows=2 loops=1)
   Merge Cond: (t.ticket_no = tf.ticket_no)
   ->  Index Scan using tickets_pkey on tickets t  (cost=0.43..139109.80 rows=2949825 width=104) (actual rows=1336684 loops=1)
   ->  Materialize  (cost=215411.52..216873.10 rows=292315 width=14) (actual rows=2 loops=1)
         ->  Sort  (cost=215411.52..216142.31 rows=292315 width=14) (actual rows=2 loops=1)
               Sort Key: tf.ticket_no
               Sort Method: quicksort  Memory: 25kB
               ->  Hash Join  (cost=6742.28..183875.47 rows=292315 width=14) (actual rows=2 loops=1)
                     Hash Cond: (tf.flight_id = f.flight_id)
                     ->  Seq Scan on ticket_flights tf  (cost=0.00..174831.15 rows=876949 width=18) (actual rows=859656 loops=1)
                           Filter: ((fare_conditions)::text = 'Business'::text)
                           Rows Removed by Filter: 7532196
                     ->  Hash  (cost=5847.00..5847.00 rows=71622 width=4) (actual rows=1 loops=1)
                           Buckets: 131072  Batches: 1  Memory Usage: 1025kB
                           ->  Seq Scan on flights f  (cost=0.00..5847.00 rows=71622 width=4) (actual rows=1 loops=1)
                                 Filter: (actual_departure > (scheduled_departure + '05:00:00'::interval))
                                 Rows Removed by Filter: 214866
 Planning Time: 24.336 ms
 Execution Time: 3331.920 ms
(19 rows)
```

Здесь мы имеем дело с запросом, характерным для OLTP — небольшое число строк, для получения которых надо выбрать только небольшую часть данных. Поэтому общее направление оптимизации — переход от полного сканирования и соединения хешированием к индексам и вложенным циклам.

Заметим, что рейс, задержанный более чем на 5 часов, всего один, а планировщик сканирует всю таблицу. Можно построить функциональный индекс на разности двух столбцов и немного переписать условие:

```
=> CREATE INDEX ON flights ((actual_departure - scheduled_departure));
```

```
CREATE INDEX
```

```
=> ANALYZE flights;
```

```
ANALYZE
```

```
=> EXPLAIN (analyze, timing off)
SELECT t.*
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
  JOIN flights f ON f.flight_id = tf.flight_id
WHERE tf.fare_conditions = 'Business'
  AND f.actual_departure - f.scheduled_departure > interval '5 hour';
```

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=12.82..177151.37 rows=8 width=104) (actual rows=2 loops=1)
   ->  Hash Join  (cost=12.39..177145.58 rows=8 width=14) (actual rows=2 loops=1)
         Hash Cond: (tf.flight_id = f.flight_id)
         ->  Seq Scan on ticket_flights tf  (cost=0.00..174831.15 rows=876949 width=18) (actual rows=859656 loops=1)
               Filter: ((fare_conditions)::text = 'Business'::text)
               Rows Removed by Filter: 7532196
         ->  Hash  (cost=12.37..12.37 rows=2 width=4) (actual rows=1 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Index Scan using flights_expr_idx on flights f  (cost=0.42..12.37 rows=2 width=4) (actual rows=1 loops=1)
                     Index Cond: ((actual_departure - scheduled_departure) > '05:00:00'::interval)
   ->  Index Scan using tickets_pkey on tickets t  (cost=0.43..0.72 rows=1 width=104) (actual rows=1 loops=2)
         Index Cond: (ticket_no = tf.ticket_no)
 Planning Time: 15.487 ms
 Execution Time: 2227.160 ms
(14 rows)
```

Теперь займемся таблицей ticket_flights, которая тоже сканируется полностью, хотя из нее читается незначительная часть строк.

Помог бы индекс по классам обслуживания fare_conditions, но лучше создать индекс по столбцу flight_id, что позволит эффективно выполнять соединение вложенным циклом с flights:

```
=> CREATE INDEX ON ticket_flights(flight_id);
```

```
CREATE INDEX
```

```
=> EXPLAIN (analyze, timing off)
SELECT t.*
FROM tickets t
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
  JOIN flights f ON f.flight_id = tf.flight_id
WHERE tf.fare_conditions = 'Business'
  AND f.actual_departure - f.scheduled_departure > interval '5 hour';
```

```
                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=1.28..814.80 rows=8 width=104) (actual rows=2 loops=1)
   ->  Nested Loop  (cost=0.85..809.00 rows=8 width=14) (actual rows=2 loops=1)
         ->  Index Scan using flights_expr_idx on flights f  (cost=0.42..12.37 rows=2 width=4) (actual rows=1 loops=1)
               Index Cond: ((actual_departure - scheduled_departure) > '05:00:00'::interval)
         ->  Index Scan using ticket_flights_flight_id_idx on ticket_flights tf  (cost=0.43..398.21 rows=11 width=18) (actual rows=2 loops=1)
               Index Cond: (flight_id = f.flight_id)
               Filter: ((fare_conditions)::text = 'Business'::text)
               Rows Removed by Filter: 10
   ->  Index Scan using tickets_pkey on tickets t  (cost=0.43..0.72 rows=1 width=104) (actual rows=1 loops=2)
         Index Cond: (ticket_no = tf.ticket_no)
 Planning Time: 8.366 ms
 Execution Time: 8.635 ms
(12 rows)
```

Время выполнения уменьшилось до миллисекунд.

2. Параметр from_collapse_limit

Сначала посмотрим, как работает запрос со значением from_collapse_limit по умолчанию:

```
=> SHOW from_collapse_limit;
```

```
 from_collapse_limit
---------------------
 8
(1 row)
```

```
=> EXPLAIN (costs off, summary off, settings)
SELECT *
FROM
  (
    SELECT *
    FROM ticket_flights tf
      JOIN tickets t ON tf.ticket_no = t.ticket_no
  ) ttf
  JOIN flights f ON f.flight_id = ttf.flight_id;
```

```
                                           QUERY PLAN
-----------------------------------------------------------------------------------------------
 Hash Join
   Hash Cond: (tf.ticket_no = t.ticket_no)
   ->  Hash Join
         Hash Cond: (tf.flight_id = f.flight_id)
         ->  Seq Scan on ticket_flights tf
         ->  Hash
               ->  Seq Scan on flights f
   ->  Hash
         ->  Seq Scan on tickets t
 Settings: jit = 'off', max_parallel_workers_per_gather = '0', search_path = 'bookings, public'
(10 rows)
```

В подзапросе соединяются две таблицы, но оптимизатор раскрывает подзапрос и выбирает порядок соединения для всех трех таблиц. Первыми здесь соединяются таблицы перелетов (ticket_flights) и рейсов (flights).

Теперь уменьшим значение from_collapse_limit до единицы и посмотрим на новый план того же запроса:

```
=> SET from_collapse_limit = 1;
```

```
SET
```

```
=> EXPLAIN (costs off, summary off, settings)
SELECT *
FROM
  (
    SELECT *
    FROM ticket_flights tf
      JOIN tickets t ON tf.ticket_no = t.ticket_no
  ) ttf
  JOIN flights f ON f.flight_id = ttf.flight_id;
```

```
                                                      QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
 Hash Join
   Hash Cond: (tf.ticket_no = t.ticket_no)
   ->  Hash Join
         Hash Cond: (tf.flight_id = f.flight_id)
         ->  Seq Scan on ticket_flights tf
         ->  Hash
               ->  Seq Scan on flights f
   ->  Hash
         ->  Seq Scan on tickets t
 Settings: from_collapse_limit = '1', jit = 'off', max_parallel_workers_per_gather = '0', search_path = 'bookings, public'
(10 rows)
```

План запроса поменялся — сначала соединяются две таблицы из подзапроса, который теперь не раскрывается.