

Оптимизация запросов

Подходы к настройке



16

Авторские права

© Postgres Professional, 2019–2024

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

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

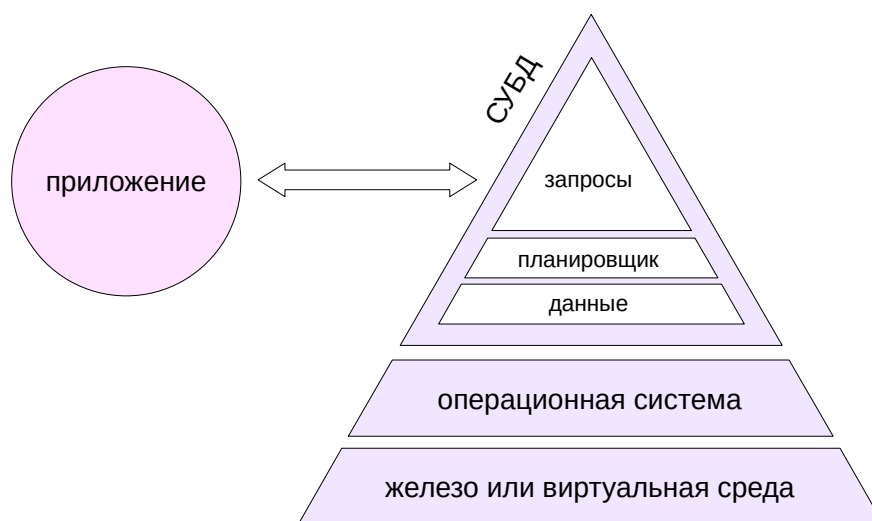
Что настраивать?

Настройка сервера

Настройка приложения

Запросы

Что настраивать?



Производительность работы информационной системы обеспечивается на разных уровнях.

Мы совсем не рассматриваем два важных уровня: оборудование (которое может быть виртуализировано, и тогда к картине добавляется еще один слой) и операционную систему. В частности, работа СУБД с данными происходит через файловую систему, а эффективность и надежность операций ввода-вывода зависит от дисковой подсистемы. Поэтому прежде всего нужно обеспечить необходимые ресурсы и настроить ОС для их максимального использования.

Сервер баз данных настраивается так, чтобы служебные задачи не занимали больше ресурсов, чем необходимо, и чтобы большинство запросов работали приемлемо. На этом уровне мы можем не только настраивать конфигурационные параметры (из которых рассмотрим лишь малую часть), но и управлять расположением данных. Наша задача состоит в том, чтобы обеспечить разумное распределение ресурсов и адекватную работу планировщика.

Далее дело доходит до настройки отдельных запросов. В теме «Профилирование» мы говорили, что разработчик обычно думает над оптимизацией запросов, которые в данный момент пишет, а администратор — над оптимизацией запросов, оказывающих наибольшее влияние на сервер в целом. В этой теме мы рассмотрим некоторые приемы оптимизации запросов.

Но запросы, составляющие рабочую нагрузку, инициируются приложением. Поэтому и настройка приложения не менее важна, чем настройка СУБД.

Использование ресурсов

Физическое расположение данных

Статистика и настройки оптимизатора

Фоновые рабочие процессы

max_parallel_workers_per_gather = 2

max_parallel_workers = 8

max_worker_processes = 8

Ввод-вывод

effective_io_concurrency = 1

maintenance_io_concurrency = 10



Начнем с настроек, которые определяют, какими ресурсами может распоряжаться СУБД.

PostgreSQL может использовать несколько ядер процессоров при параллельной обработке. Для этого запускаются дополнительные фоновые рабочие процессы. Подробно параллельное выполнение запросов было рассмотрено в теме «Параллельный доступ», а использование фоновых процессов для прикладной разработки — в теме «Фоновые процессы» курса DEV2.

Из настроек, связанных с использованием ввода-вывода, отметим параметр *effective_io_concurrency*. Он сообщает системе количество независимых дисков в дисковом массиве. Фактически этот параметр влияет только на количество страниц, которые будут предварительно считываться в кеш при сканировании по битовой карте.

Похожий параметр, действующий для некоторых служебных задач, называется *maintenance_io_concurrency*.

Память

<code>work_mem</code>	= 4MB	
<code>hash_mem_multiplier</code>	= 2	
<code>maintenance_work_mem</code>	= 64MB	
<code>shared_buffers</code>	= 128MB	
<code>effective_cache_size</code>	= 4GB	

Ряд настроек влияет на выделение и использование памяти.

Параметр `work_mem` определяет размер доступной памяти для отдельных операций (рассматривался в соответствующих темах).

Также он влияет и на выбор плана. Например, при небольших значениях предпочтение отдается сортировке, которая лучше работает при недостатке памяти, чем хеширование. Поэтому для узлов, использующих хеширование, размер рабочей памяти можно увеличить с помощью параметра-мультипликатора `hash_mem_multiplier`.

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

Параметр `shared_buffers` определяет размер буферного кеша экземпляра. Настройка этого параметра рассматривается в курсе DBA2, но точно можно сказать, что значение по умолчанию крайне мало.

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

Табличные пространства

разнесение данных по разным физическим устройствам
`ALTER TABLESPACE SET ...`

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.

С помощью **табличных пространств** можно управлять размещением объектов по физическим устройствам ввода-вывода. Например, активно используемые данные хранить на SSD-дисках, а архивные — на более медленных HDD.

Часть параметров сервера, которые зависят от характеристик носителя, (например, *random_page_cost*) можно устанавливать на уровне табличных пространств.

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/16/ddl-partitioning>

<https://postgrespro.ru/docs/postgresql/16/postgres-fdw>

Актуальность

настройка автоочистки и автоанализа

autovacuum_max_workers, autovacuum_analyze_scale_factor, ...

Точность

default_statistics_target = 100

индекс по выражению, расширенная статистика

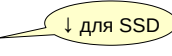
Планировщик опирается в своих оценках на статистику, поэтому статистика должна собираться достаточно часто. Достигается это настройкой процесса автоочистки и автоанализа. Детали настройки подробно рассматриваются в курсе DBA2.

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 будет полезным индекс по выражению, обладающий собственной статистикой. В отдельных случаях можно использовать расширенную статистику.

Ввод-вывод

seq_page_cost = 1.0
random_page_cost = 4.0



Время процессора

cpu_tuple_cost, *cpu_index_tuple_cost*, *cpu_operator_cost*
стоимость пользовательских функций

Имеется большое число настроек для стоимости элементарных операций, из которых, как мы уже видели, в итоге складывается стоимость плана запроса. Такие настройки имеет смысл изменять, если запросы, несмотря на точную оценку кардинальности, выполняются не самым эффективным образом.

Параметры *seq_page_cost* и *random_page_cost* связаны со вводом-выводом и определяют относительную стоимость чтения одной страницы при последовательном 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. Для SSD-дисков (а также в случаях, когда все данные с большой вероятностью будут закешированы), значение этого параметра необходимо значительно уменьшать, например, до 1.1.

Параметры времени процессора определяют веса для учета стоимости обработки прочитанных данных. Обычно эти параметры не меняют, поскольку трудно спрогнозировать эффект изменений.

Но, как мы видели в теме «Функции», иногда полезно задать стоимость пользовательской функции в единицах *cpu_operator_cost*.

<https://postgrespro.ru/docs/postgresql/16/runtime-config-query>

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=1202928 width=21) (actual rows=1187454  
loops=1)  
  Filter: (book_ref < '9000'::bpchar)  
    Rows Removed by Filter: 923656  
  Planning Time: 14.118 ms  
  Execution Time: 255.214 ms  
(5 rows)
```

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

Так ли удачно это решение? Проверим, скомандовав планировщику не использовать последовательное сканирование, если есть другие способы:

```
=> SET enable_seqscan = off; -- действует до конца сеанса
```

SET

Аналогичные enable-параметры есть и для прочих способов соединения, методов доступа и многих других операций. Эти параметры довольно грубо вмешиваются в работу планировщика, но весьма полезны для отладки и экспериментов.

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

QUERY PLAN

```
-----  
-----  
Index Scan using bookings_pkey on bookings (cost=0.43..41921.67 rows=1202928 width=21)  
(actual rows=1187454 loops=1)  
  Index Cond: (book_ref < '9000'::bpchar)  
  Planning Time: 0.100 ms  
  Execution Time: 191.071 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 таком случае стоит либо уменьшить значение `random_page_cost`, чтобы планировщик не завышал стоимость индексного доступа, либо увеличить `effective_cache_size`, чтобы сделать индексное сканирование более привлекательным.

```
=> RESET enable_seqscan; -- отменяем
```

RESET

Клиенты и соединения

Схема данных

Много сеансов

пул соединений

Курсоры

если нужна часть выборки

cursor_tuple_fraction

Много коротких запросов

подготовленные операторы

перенос логики в SQL

Если приложение открывает слишком много соединений или устанавливает их слишком часто и на короткое время, может потребоваться установить между приложением и СУБД пул соединений. Однако пул накладывает на приложение определенные ограничения. Подробно этот вопрос рассмотрен в курсе DEV2.

Курсоры следует использовать, только если нужно получить небольшую часть выборки и ее размер зависит от действий пользователя. В этом случае параметр *cursor_tuple_fraction* подскажет планировщику, что нужно оптимизировать получение первых строк выборки.

Если приложение выполняет слишком большое количество мелких запросов (каждый из которых сам по себе выполняется эффективно), общая эффективность будет очень невысока. Это часто происходит при использовании средств объектно-реляционного отображения (ORM). В теме «Профилирование» мы рассматривали похожий пример с функцией, которая вызывалась в цикле.

В таких случаях со стороны СУБД практически нет средств для оптимизации (кроме подготовленных операторов, если запросы повторяются). 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, если приложение уже написано, переделать его бывает очень затратно.

Нормализация — устранение избыточности в данных

упрощает запросы и проверку согласованности

Денормализация — привнесение избыточности

может повысить производительность, но требует синхронизации

индексы

предрасчитанные 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 база данных.

Другой пример — материализованные представления. Их также надо обновлять, например, по расписанию или другим способом. Подробно это рассматривалось в теме «Материализация».

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.

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 диапазоновых 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=310601.20..677476.01 rows=7925688 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..153887.09 rows=8392709 width=32)
    -> Hash  (cost=137535.88..137535.88 rows=7925688 width=25)
        -> Seq Scan on boarding_passes bp  (cost=0.00..137535.88 rows=7925688 width=25)
(5 rows)
```

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

```
=> BEGIN;
```

```
BEGIN
```

```
=> 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

```
-----
Gather  (cost=164413.25..358122.02 rows=331 width=57)
  Workers Planned: 2
    -> Parallel Hash Join  (cost=163413.25..357088.92 rows=138 width=57)
        Hash Cond: ((tf.flight_id = bp.flight_id) AND (tf.ticket_no = bp.ticket_no))
        -> Parallel Seq Scan on ticket_flights tf  (cost=0.00..104929.62 rows=3496962
width=32)
        -> Parallel Hash  (cost=91302.70..91302.70 rows=3302370 width=25)
            -> Parallel Seq Scan on boarding_passes bp  (cost=0.00..91302.70
rows=3302370 width=25)
(7 rows)
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

Пути оптимизации

Короткие запросы

Длинные запросы

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.

Читают небольшую часть данных, выдают мало строк

Характерны для OLTP

Важно получать ответ быстро

Характерные особенности плана

- условия с высокой селективностью

- индексы

- соединения методом вложенного цикла

Запросы можно (довольно условно) разделить на две группы, имеющие свои особенности и требующие разных подходов к оптимизации: «короткие» и «длинные».

К первой относятся запросы, характерные для OLTP-систем. Такие запросы читают немного данных и выдают одну или несколько строк. Они могут обращаться и к большим таблицам, но в этом случае имеют условия с высокой селективностью, позволяющие читать лишь малую часть данных.

Обычно короткие запросы обслуживают пользовательский интерфейс, и поэтому важно, чтобы они выполнялись как можно быстрее. Приоритет отдается времени отклика.

Это обеспечивается тем, что необходимые запросам данные читаются либо из очень небольших таблиц (одна-две страницы), либо по индексу. Соединения, как правило, выполняются методом вложенного цикла, который не требует подготовительных действий (в отличие от хеш-соединения) и не вызывает проблем при небольшом количестве соединяемых строк.

Короткие запросы

Сразу отключим параллельное выполнение. Для коротких запросов оно бесполезно, а параллельные планы сложнее читать.

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

SET

Рассмотрим запрос, который выводит посадочные талоны, выданные на рейсы, отправляющиеся в течение часового интервала:

```
=> EXPLAIN (analyze, timing off) SELECT bp.*
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id
WHERE date_trunc('hour', f.scheduled_departure) = '2017-06-01 12:00:00';
```

QUERY PLAN

```
-----
Hash Join (cost=5860.43..164201.65 rows=39616 width=25) (actual rows=1799 loops=1)
  Hash Cond: (bp.flight_id = f.flight_id)
  -> Seq Scan on boarding_passes bp (cost=0.00..137535.88 rows=7925688 width=25)
      (actual rows=7925812 loops=1)
  -> Hash (cost=5847.00..5847.00 rows=1074 width=4) (actual rows=51 loops=1)
      Buckets: 2048 Batches: 1 Memory Usage: 18kB
      -> Seq Scan on flights f (cost=0.00..5847.00 rows=1074 width=4) (actual
rows=51 loops=1)
      Filter: (date_trunc('hour'::text, scheduled_departure) = '2017-06-01
12:00:00+03'::timestamp with time zone)
      Rows Removed by Filter: 214816
Planning Time: 0.401 ms
Execution Time: 4358.258 ms
(10 rows)
```

Этот запрос можно отнести к разряду коротких: из двух больших таблиц требуется гораздо меньше процента данных. Однако в плане мы видим, что таблицы читаются полностью. Кроме того, наблюдается большая разница между прогнозируемой и фактической кардинальностью.

Начнем оптимизацию с самого вложенного узла, Seq Scan по таблице рейсов flights. Из-за чего планировщик ошибается в оценке кардинальности?

Дело в функции date_trunc: для нее нет вспомогательной функции планировщика, поэтому берется фиксированная оценка селективности 5%. В темах «Базовая статистика» и «Расширенная статистика» мы исправляли подобные ситуации с помощью индекса по выражению и расширенной статистики. Но в данном случае достаточно переписать условие так, чтобы слева от оператора находилось поле таблицы:

```
=> EXPLAIN (analyze, timing off) SELECT *
FROM flights
WHERE scheduled_departure >= '2017-06-01 12:00:00'
AND scheduled_departure < '2017-06-01 13:00:00';
```

QUERY

PLAN

```
-----
Index Scan using flights_flight_no_scheduled_departure_key on flights
(cost=0.42..5560.92 rows=22 width=63) (actual rows=51 loops=1)
  Index Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp with time
zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time zone))
Planning Time: 1.115 ms
Execution Time: 34.219 ms
(4 rows)
```

Теперь оценка больше похожа на правду, а запрос заодно стал использовать индекс. Хорошо ли это?

В целом хорошо, но индекс создан по столбцам flight_no и scheduled_departure, а условие есть только на второй столбец. В теме «Методы доступа» мы выяснили, что в этом случае индекс будет сканироваться полностью, что нельзя считать эффективным способом доступа к данным.

Создадим новый индекс:

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

CREATE INDEX

Проверим наш запрос:

```
=> EXPLAIN (analyze, timing off) SELECT bp.*
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id
WHERE f.scheduled_departure >= '2017-06-01 12:00:00'
AND f.scheduled_departure < '2017-06-01 13:00:00';
```

QUERY PLAN

```
-----
-----
-----
Nested Loop (cost=10.28..13068.31 rows=812 width=25) (actual rows=1799 loops=1)
-> Bitmap Heap Scan on flights f (cost=4.65..86.93 rows=22 width=4) (actual rows=51
loops=1)
    Recheck Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp with
time zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time zone))
    Heap Blocks: exact=50
    -> Bitmap Index Scan on flights_scheduled_departure_idx (cost=0.00..4.64
rows=22 width=0) (actual rows=51 loops=1)
        Index Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp
with time zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time
zone))
    -> Bitmap Heap Scan on boarding_passes bp (cost=5.63..588.51 rows=155 width=25)
(actual rows=35 loops=51)
        Recheck Cond: (f.flight_id = flight_id)
        Heap Blocks: exact=49
        -> Bitmap Index Scan on boarding_passes_flight_id_seat_no_key (cost=0.00..5.59
rows=155 width=0) (actual rows=35 loops=51)
            Index Cond: (flight_id = f.flight_id)
Planning Time: 0.355 ms
Execution Time: 33.460 ms
(13 rows)
```

В принципе, мы достигли желаемого: планировщик использует индексный доступ, данные соединяются методом вложенного цикла. Проверим еще количество страниц, которые потребовалось прочитать:

```
=> EXPLAIN (analyze, buffers, costs off, timing off) SELECT bp.*
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id
WHERE f.scheduled_departure >= '2017-06-01 12:00:00'
AND f.scheduled_departure < '2017-06-01 13:00:00';
```

QUERY PLAN

```
-----
-----
-----
Nested Loop (actual rows=1799 loops=1)
    Buffers: shared hit=259
    -> Bitmap Heap Scan on flights f (actual rows=51 loops=1)
        Recheck Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp with
time zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time zone))
        Heap Blocks: exact=50
        Buffers: shared hit=53
        -> Bitmap Index Scan on flights_scheduled_departure_idx (actual rows=51 loops=1)
            Index Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp
with time zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time
zone))
            Buffers: shared hit=3
    -> Bitmap Heap Scan on boarding_passes bp (actual rows=35 loops=51)
        Recheck Cond: (f.flight_id = flight_id)
        Heap Blocks: exact=49
        Buffers: shared hit=206
        -> Bitmap Index Scan on boarding_passes_flight_id_seat_no_key (actual rows=35
loops=51)
            Index Cond: (flight_id = f.flight_id)
            Buffers: shared hit=157
Planning:
    Buffers: shared hit=16
Planning Time: 0.213 ms
Execution Time: 0.678 ms
(20 rows)
```

Возможным улучшением будет исключение доступа к таблице flights за счет сканирования только индекса. Удалим созданный ранее индекс и сделаем другой с дополнительным столбцом, который нужен для соединения:

```
=> DROP INDEX flights_scheduled_departure_idx;
```

```
DROP INDEX
```

```
=> CREATE INDEX ON flights(scheduled_departure, flight_id);
```

```
CREATE INDEX
```

```
=> EXPLAIN (analyze, buffers, costs off, timing off) SELECT bp.*  
FROM flights f  
JOIN boarding_passes bp ON bp.flight_id = f.flight_id  
WHERE f.scheduled_departure >= '2017-06-01 12:00:00'  
AND f.scheduled_departure < '2017-06-01 13:00:00';
```

QUERY

PLAN

```
-----  
-----  
Nested Loop (actual rows=1799 loops=1)  
  Buffers: shared hit=207 read=3  
  -> Index Only Scan using flights_scheduled_departure_flight_id_idx on flights f  
(actual rows=51 loops=1)  
    Index Cond: ((scheduled_departure >= '2017-06-01 12:00:00+03'::timestamp with  
time zone) AND (scheduled_departure < '2017-06-01 13:00:00+03'::timestamp with time zone))  
    Heap Fetches: 0  
    Buffers: shared hit=1 read=3  
    -> Bitmap Heap Scan on boarding_passes bp (actual rows=35 loops=51)  
      Recheck Cond: (f.flight_id = flight_id)  
      Heap Blocks: exact=49  
      Buffers: shared hit=206  
      -> Bitmap Index Scan on boarding_passes_flight_id_seat_no_key (actual rows=35  
loops=51)  
        Index Cond: (flight_id = f.flight_id)  
        Buffers: shared hit=157  
  
Planning:  
  Buffers: shared hit=40 read=1  
Planning Time: 0.509 ms  
Execution Time: 1.121 ms  
(17 rows)
```

Мы выиграли еще около 20% чтений. Это может быть важно для запросов, которые выполняются часто, но вряд ли это наш случай. Скорее всего, включение в индекс всех полей таблицы boarding_passes будет избыточным — накладные расходы перевесят выигрыш в производительности. Остановимся на достигнутом.

Читают большие таблицы целиком

Характерны для OLAP

Важно не читать одни и те же данные повторно

Характерные особенности плана

- условия с низкой селективностью

- полное сканирование

- соединение хешированием

- агрегация

- параллельное выполнение

Длинные запросы встречаются в OLAP-системах. Они характеризуются тем, что для получения результата должны прочитать большое количество данных. При этом не важно, сколько строк возвращает запрос: в результате агрегации может остаться и одна.

Такие запросы могут читать большие таблицы целиком, а условия обычно низкоселективны. Поэтому вместо индексного доступа более эффективным становится последовательное сканирование, а вместо соединения вложенным циклом — хеш-соединение. Тем более, что для длинного запроса важно выдать весь результат за разумное время, а не первые строки как можно раньше.

Очень важно следить, чтобы одни и те же данные не читались в запросе несколько раз. Это может происходить по самым разным причинам (из-за коррелированных подзапросов, из-за использования функций и т. п.), которые приводят к появлению вложенных циклов, явных или неявных.

Разумеется, в плане длинного запроса могут встречаться и индексный доступ (при наличии высокоселективных условий), и соединение вложенным циклом (при соединении небольшого количества строк).

Благодаря тому, что длинные запросы обрабатывают много данных и обычно агрегируют их (пользователям редко бывают нужны миллионы строк), они могут выигрывать от параллельного выполнения.

Длинные запросы

Задача: посчитать количество пассажиров, летевших рейсами, и вылет, и прибытие которых состоялись с опозданием от одной минуты до четырех часов.

Вот запрос, решающий задачу, но время его выполнения нас не устраивает:

```
=> \timing on
```

Timing is on.

```
=> SELECT count(*)
FROM flights f
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN boarding_passes bp ON bp.flight_id = tf.flight_id AND bp.ticket_no = tf.ticket_no
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'
  AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';
```

```
count
-----
6940272
(1 row)
```

Time: 133023,560 ms (02:13,024)

```
=> \timing off
```

Timing is off.

Короткий это запрос или длинный?

Узнаем селективность условий:

```
=> SELECT count(*) FROM flights f;
```

```
count
-----
214867
(1 row)
```

```
=> SELECT count(*) FROM flights f
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'
  AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';
```

```
count
-----
173846
(1 row)
```

Условия отбирают примерно 80% строк, все таблицы в запросе большие. Очевидно, это длинный запрос.

Проверим его план:

```
=> EXPLAIN
SELECT count(*)
FROM flights f
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN boarding_passes bp ON bp.flight_id = tf.flight_id AND bp.ticket_no = tf.ticket_no
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'
  AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';
```

QUERY PLAN

```
-----  
-----  
-----  
Aggregate (cost=12190.22..12190.23 rows=1 width=8)  
  -> Nested Loop (cost=6.19..12189.76 rows=184 width=0)  
        Join Filter: (f.flight_id = tf.flight_id)  
        -> Nested Loop (cost=5.63..12076.87 rows=184 width=22)  
              -> Seq Scan on flights f (cost=0.00..9070.01 rows=5 width=4)  
                    Filter: (((actual_departure - scheduled_departure) >=  
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=  
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >= '00:01:00'::interval)  
AND ((actual_arrival - scheduled_arrival) <= '04:00:00'::interval))  
              -> Bitmap Heap Scan on boarding_passes bp (cost=5.63..599.82 rows=155  
width=18)  
                    Recheck Cond: (f.flight_id = flight_id)  
                    -> Bitmap Index Scan on boarding_passes_flight_id_seat_no_key  
(cost=0.00..5.59 rows=155 width=0)  
                          Index Cond: (flight_id = f.flight_id)  
              -> Index Only Scan using ticket_flights_pkey on ticket_flights tf  
(cost=0.56..0.60 rows=1 width=18)  
                    Index Cond: ((ticket_no = bp.ticket_no) AND (flight_id = bp.flight_id))  
(12 rows)
```

Здесь мы видим индексный доступ и соединения вложенным циклом — операции, совсем не подходящие для длинного запроса. Почему так получилось?

Проблема в ошибке оценки селективности условий. Планировщик считает, что из таблицы flights будет выбрано всего пять строк.

Попробуем быстро проверить гипотезу, что переход на другой способ соединения поможет делу. Попросим планировщик не использовать вложенный цикл, если это возможно:

```
=> SET enable_nestloop = off;
```

SET

```
=> EXPLAIN (analyze, buffers, timing off, costs off)  
SELECT count(*)  
FROM flights f  
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id  
  JOIN boarding_passes bp ON bp.flight_id = tf.flight_id AND bp.ticket_no = tf.ticket_no  
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'  
      AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';
```


QUERY PLAN

```

-----
-----
-----
Aggregate (actual rows=1 loops=1)
  Buffers: shared hit=7083 read=123780, temp read=102066 written=102066
  -> Hash Join (actual rows=6940272 loops=1)
    Hash Cond: ((tf.flight_id = f.flight_id) AND (tf.ticket_no = bp.ticket_no))
    Buffers: shared hit=7083 read=123780, temp read=102066 written=102066
    -> Seq Scan on ticket_flights tf (actual rows=8391852 loops=1)
      Buffers: shared read=69960
    -> Hash (actual rows=6940272 loops=1)
      Buckets: 262144 (originally 1024) Batches: 64 (originally 1) Memory
Usage: 7780kB
      Buffers: shared hit=7083 read=53820, temp written=34991
      -> Hash Join (actual rows=6940272 loops=1)
        Hash Cond: (bp.flight_id = f.flight_id)
        Buffers: shared hit=7083 read=53820
        -> Seq Scan on boarding_passes bp (actual rows=7925812 loops=1)
          Buffers: shared hit=4459 read=53820
        -> Hash (actual rows=173846 loops=1)
          Buckets: 262144 (originally 1024) Batches: 1 (originally 1)
Memory Usage: 8160kB
          Buffers: shared hit=2624
          -> Seq Scan on flights f (actual rows=173846 loops=1)
            Filter: (((actual_departure - scheduled_departure) >=
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >= '00:01:00'::interval)
AND ((actual_arrival - scheduled_arrival) <= '04:00:00'::interval))
            Rows Removed by Filter: 41021
            Buffers: shared hit=2624

Planning:
  Buffers: shared hit=52
Planning Time: 0.484 ms
Execution Time: 30803.275 ms
(26 rows)

```

Обращения к таблицам автоматически сменились на последовательное чтение. Запрос ускорился, ему потребовалось прочитать примерно 130 тысяч блоков (против 29 миллионов у первой версии!).

```
=> RESET enable_nestloop;
```

```
RESET
```

Нам осталось более аккуратно объяснить планировщику, какой план стоит использовать. Самый мягкий способ — предоставить более точную статистику. В данном случае подойдет статистика по выражению, которую мы рассматривали в теме «Расширенная статистика»:

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

```
CREATE STATISTICS
```

```
=> CREATE STATISTICS ON (actual_arrival - scheduled_arrival) FROM flights;
```

```
CREATE STATISTICS
```

```
=> ANALYZE flights;
```

```
ANALYZE
```

Проверим план:

```

=> EXPLAIN
SELECT count(*)
FROM flights f
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN boarding_passes bp ON bp.flight_id = tf.flight_id AND bp.ticket_no = tf.ticket_no
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'
AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';

```

QUERY PLAN

```
-----  
-----  
-----  
Aggregate (cost=759183.95..759183.96 rows=1 width=8)  
-> Hash Join (cost=11616.26..744878.36 rows=5722237 width=0)  
    Hash Cond: (tf.flight_id = f.flight_id)  
    -> Merge Join (cost=1.12..649931.87 rows=7925688 width=8)  
        Merge Cond: ((tf.ticket_no = bp.ticket_no) AND (tf.flight_id =  
bp.flight_id))  
        -> Index Only Scan using ticket_flights_pkey on ticket_flights tf  
(cost=0.56..292346.85 rows=8392709 width=18)  
        -> Index Only Scan using boarding_passes_pkey on boarding_passes bp  
(cost=0.56..275989.88 rows=7925688 width=18)  
    -> Hash (cost=9070.01..9070.01 rows=155131 width=4)  
        -> Seq Scan on flights f (cost=0.00..9070.01 rows=155131 width=4)  
            Filter: (((actual_departure - scheduled_departure) >=  
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=  
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >=  
'00:01:00'::interval) AND ((actual_arrival - scheduled_arrival) <=  
'04:00:00'::interval))  
(10 rows)
```

Оценки исправились, теперь во всех узлах они довольно точно соответствуют реальным цифрам. Однако планировщик выбрал не тот план, что мы видели: он перестал использовать соединение вложенным циклом, но оставил индексный доступ и применил соединение слиянием. В результате запрос стал выполняться еще быстрее, хотя ему и потребовалось прочитать более двух миллионов страниц. Хорошо это или плохо?

Это зависит от обстоятельств. Если мы уверены, что размера кеша хватит, чтобы в нем находились данные наших крупных таблиц — хорошо. В конце концов, запрос действительно стал выполняться быстрее! Однако в реальной системе за кеш будут конкурировать данные многих таблиц. Если реальное обращение к страницам окажется медленнее ожидаемого, это повод вернуться к настройкам сервера БД с помощью параметров `random_page_cost` и `effective_cache_size`.

Однако зададимся вопросом: не читаем ли мы лишние данные? Знание предметной области позволяет нам прийти к выводу, что таблица `ticket_flights` вовсе не нужна в запросе. Таблицу `flights` можно соединить непосредственно с `boarding_passes` — внешние ключи проверяют корректность данных, но не требуются для соединения. Добавление `ticket_flights` никак не ограничивает данные (не может быть ситуации, при которой посадочный талон есть, а перелет отсутствует).

Вообще, переформулирование запроса — самый разнообразный и вариативный способ влияния на производительность. Некоторые эквивалентные преобразования планировщик умеет выполнять сам, но мы можем:

- переписать условия так, чтобы они могли (или наоборот, не могли) использовать индекс;
- заменить коррелированные подзапросы (которые, по сути, являются вложенным циклом) соединениями;
- устранить лишние сканирования (как в этом примере, или, скажем, за счет применения оконных функций);
- использовать недоступные планировщику трансформации, такие, как преобразование условий OR в UNION;
- и т. п.

Итак, упрощаем запрос:

```
=> EXPLAIN (analyze, buffers, timing off, costs off)  
SELECT count(*)  
FROM flights f  
    JOIN boarding_passes bp ON bp.flight_id = f.flight_id  
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'  
    AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';
```

QUERY PLAN

```

-----
Aggregate (actual rows=1 loops=1)
  Buffers: shared hit=7115 read=53788, temp read=11797 written=11797
  -> Hash Join (actual rows=6940272 loops=1)
    Hash Cond: (bp.flight_id = f.flight_id)
    Buffers: shared hit=7115 read=53788, temp read=11797 written=11797
    -> Seq Scan on boarding_passes bp (actual rows=7925812 loops=1)
      Buffers: shared hit=4491 read=53788
    -> Hash (actual rows=173846 loops=1)
      Buckets: 262144 Batches: 2 Memory Usage: 5118kB
      Buffers: shared hit=2624, temp written=253
      -> Seq Scan on flights f (actual rows=173846 loops=1)
        Filter: (((actual_departure - scheduled_departure) >=
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >= '00:01:00'::interval)
AND ((actual_arrival - scheduled_arrival) <= '04:00:00'::interval))
        Rows Removed by Filter: 41021
        Buffers: shared hit=2624

Planning:
  Buffers: shared hit=16
Planning Time: 0.253 ms
Execution Time: 7355.845 ms
(18 rows)

```

Потребовалось прочитать всего 61 тысячу страниц. Проверим, сколько всего страниц в таблицах:

```

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

sum
-----
60903
(1 row)

```

Теперь мы уверены, что не читаем лишних данных.

Можно ли еще что-то улучшить?

План запроса говорит о том, что выполнялось двухпроходное соединение хешированием. Можно увеличить значение параметра `work_mem`, чтобы хеш-соединению было достаточно одного прохода без временных файлов.

```

=> SET work_mem = '8MB'; -- имеет смысл увеличить для всего сервера

```

SET

```

=> EXPLAIN (analyze, timing off, costs off)
SELECT count(*)
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id
WHERE f.actual_departure - f.scheduled_departure BETWEEN interval '1 min' AND interval '4 hours'
AND f.actual_arrival - f.scheduled_arrival BETWEEN interval '1 min' AND interval '4 hours';

```

QUERY PLAN

```

-----
Aggregate (actual rows=1 loops=1)
  -> Hash Join (actual rows=6940272 loops=1)
        Hash Cond: (bp.flight_id = f.flight_id)
        -> Seq Scan on boarding_passes bp (actual rows=7925812 loops=1)
        -> Hash (actual rows=173846 loops=1)
              Buckets: 262144 Batches: 1 Memory Usage: 8160kB
              -> Seq Scan on flights f (actual rows=173846 loops=1)
                    Filter: (((actual_departure - scheduled_departure) >=
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >= '00:01:00'::interval)
AND ((actual_arrival - scheduled_arrival) <= '04:00:00'::interval))
                    Rows Removed by Filter: 41021
Planning Time: 0.284 ms
Execution Time: 1489.752 ms
(11 rows)

```

В начале темы мы отключали параллельные планы. Однако для длинных запросов параллельные планы могут иметь смысл. Конечно, при условии наличия свободных ядер, как рассматривалось в теме «Параллельный доступ». Посмотрим:

=> **RESET** max_parallel_workers_per_gather;

RESET

=> **EXPLAIN** (analyze, timing off, costs off)

SELECT count(*)

FROM flights f

JOIN boarding_passes bp **ON** bp.flight_id = f.flight_id

WHERE f.actual_departure - f.scheduled_departure **BETWEEN** interval '1 min' **AND** interval '4 hours'

AND f.actual_arrival - f.scheduled_arrival **BETWEEN** interval '1 min' **AND** interval '4 hours';

QUERY PLAN

```

-----
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (actual rows=1 loops=3)
              -> Parallel Hash Join (actual rows=2313424 loops=3)
                    Hash Cond: (bp.flight_id = f.flight_id)
                    -> Parallel Seq Scan on boarding_passes bp (actual rows=2641937
loops=3)
                    -> Parallel Hash (actual rows=57949 loops=3)
                          Buckets: 262144 Batches: 1 Memory Usage: 8896kB
                          -> Parallel Seq Scan on flights f (actual rows=57949 loops=3)
                                Filter: (((actual_departure - scheduled_departure) >=
'00:01:00'::interval) AND ((actual_departure - scheduled_departure) <=
'04:00:00'::interval) AND ((actual_arrival - scheduled_arrival) >= '00:01:00'::interval)
AND ((actual_arrival - scheduled_arrival) <= '04:00:00'::interval))
                                Rows Removed by Filter: 13674
Planning Time: 4.249 ms
Execution Time: 1616.052 ms
(15 rows)

```

Мы видим, что планировщик использует параллельный план, но в виртуальной машине с одним ядром он, конечно, не дает никаких преимуществ.

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/16/pg-hint-plan> (автор — Киотаро Хоригучи). Но нельзя забывать, что использование подсказок, сильно ограничивающих свободу планировщика, может навредить в будущем, когда распределение данных изменится.

Настройка выполняется на разных уровнях системы

Доступен широкий спектр методов влияния
на план выполнения запросов

Разные типы запросов требуют разных подходов

Ничто не заменит голову и здравый смысл

1. Оптимизируйте запрос, выводящий контактную информацию пассажиров, купивших билеты бизнес-класса, рейсы которых были задержаны более чем на 5 часов:

```
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. Оптимизируйте запрос, вычисляющий среднюю стоимость билетов на перелеты, выполняемые различными типами самолетов.

Начните с варианта, предложенного в комментарии.

2.

```
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
)
FROM aircrafts a
```

1. Оптимизация короткого 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=215149.77..366708.12 rows=289911 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..139110.69 rows=2949878
width=104) (actual rows=1336684 loops=1)
    -> Materialize (cost=215149.30..216598.85 rows=289911 width=14) (actual rows=2
loops=1)
      -> Sort (cost=215149.30..215874.07 rows=289911 width=14) (actual rows=2
loops=1)
        Sort Key: tf.ticket_no
        Sort Method: quicksort Memory: 25kB
        -> Hash Join (cost=6742.28..183890.76 rows=289911 width=14) (actual
rows=2 loops=1)
          Hash Cond: (tf.flight_id = f.flight_id)
            -> Seq Scan on ticket_flights tf (cost=0.00..174865.38 rows=869736
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: 16.369 ms
          Execution Time: 3151.427 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 индекс по выражению 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..177166.68 rows=8 width=104) (actual rows=2 loops=1)
  -> Hash Join (cost=12.39..177160.87 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..174865.38 rows=869736 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..0.73 rows=1 width=104)
(actual rows=1 loops=2)
                Index Cond: (ticket_no = tf.ticket_no)
Planning Time: 4.153 ms
Execution Time: 1457.255 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..837.39 rows=8 width=104) (actual rows=2 loops=1)
  -> Nested Loop (cost=0.85..831.57 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..409.49 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..0.73 rows=1 width=104)
(actual rows=1 loops=2)
            Index Cond: (ticket_no = tf.ticket_no)
Planning Time: 21.421 ms
Execution Time: 7.052 ms
(12 rows)
```

Время выполнения уменьшилось до миллисекунд.

2. Оптимизация длинного запроса

Очевидно, мы имеем дело с длинным запросом: в нем присутствует большая таблица перелетов `ticket_flights`, при этом ни на одну таблицу не накладывается никаких условий. Выполним запрос и посмотрим на план его выполнения:

```
=> EXPLAIN (analyze, buffers, costs off, 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
)
FROM aircrafts a;
```

QUERY PLAN

```
-----
Seq Scan on aircrafts_data ml (actual rows=9 loops=1)
  Buffers: shared hit=24480 read=558817
  SubPlan 1
    -> Aggregate (actual rows=1 loops=9)
      Buffers: shared hit=24480 read=558816
      -> Hash Join (actual rows=932428 loops=9)
        Hash Cond: (tf.flight_id = f.flight_id)
        Buffers: shared hit=24480 read=558816
        -> Seq Scan on ticket_flights tf (actual rows=8391852 loops=8)
          Buffers: shared hit=864 read=558816
        -> Hash (actual rows=23874 loops=9)
          Buckets: 65536 (originally 32768) Batches: 1 (originally 1)

Memory Usage: 2629kB
      Buffers: shared hit=23616
      -> Seq Scan on flights f (actual rows=23874 loops=9)
        Filter: (aircraft_code = ml.aircraft_code)
        Rows Removed by Filter: 190993
        Buffers: shared hit=23616

Planning:
  Buffers: shared hit=57 read=7 dirtied=1
Planning Time: 25.172 ms
Execution Time: 10730.474 ms
(21 rows)
```

В плане запроса видим, что оценки кардинальности вполне адекватны во всех узлах, большие таблицы сканируются последовательно и применяется хеш-соединение.

Однако при этом запрос читает в несколько раз больше страниц, чем необходимо:

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

 sum
-----
72584
(1 row)
```

Причина в коррелированном подзапросе, который выполняется для каждой из девяти моделей самолетов, образуя неявный вложенный цикл. Раскроем подзапрос, добавив группировку.

Может показаться заманчивым избавиться от таблицы самолетов aircrafts, ведь код самолета можно получить непосредственно из таблицы рейсов flights. Однако в этом случае из результатов пропадет модель, не совершившая ни одного рейса. Поэтому необходимо левое соединение.

Кроме того, сразу вспомним, что в демонстрации мы увеличивали work_mem, чтобы избежать двухпроходного соединения.

```
=> SET work_mem = '8MB';
```

```
SET
```

```
=> EXPLAIN (analyze, buffers, 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
GROUP BY a.aircraft_code;
```

QUERY PLAN

```

-----
HashAggregate (cost=257728.41..257728.54 rows=9 width=36) (actual rows=9 loops=1)
  Group Key: ml.aircraft_code
  Batches: 1 Memory Usage: 24kB
  Buffers: shared hit=2765 read=69820
  -> Hash Right Join (cost=7459.71..215769.15 rows=8391852 width=10) (actual
rows=8456132 loops=1)
    Hash Cond: (f.aircraft_code = ml.aircraft_code)
    Buffers: shared hit=2765 read=69820
    -> Hash Right Join (cost=7458.51..183366.07 rows=8391852 width=10) (actual
rows=8456131 loops=1)
      Hash Cond: (tf.flight_id = f.flight_id)
      Buffers: shared hit=2764 read=69820
      -> Seq Scan on ticket_flights tf (cost=0.00..153878.52 rows=8391852
width=10) (actual rows=8391852 loops=1)
        Buffers: shared hit=140 read=69820
        -> Hash (cost=4772.67..4772.67 rows=214867 width=8) (actual rows=214867
loops=1)
          Buckets: 262144 Batches: 1 Memory Usage: 10442kB
          Buffers: shared hit=2624
          -> Seq Scan on flights f (cost=0.00..4772.67 rows=214867 width=8)
(actual rows=214867 loops=1)
            Buffers: shared hit=2624
            -> Hash (cost=1.09..1.09 rows=9 width=4) (actual rows=9 loops=1)
              Buckets: 1024 Batches: 1 Memory Usage: 9kB
              Buffers: shared hit=1
              -> Seq Scan on aircrafts_data ml (cost=0.00..1.09 rows=9 width=4)
(actual rows=9 loops=1)
                Buffers: shared hit=1
Planning:
  Buffers: shared hit=26 read=2
Planning Time: 1.851 ms
Execution Time: 4929.575 ms
(26 rows)

```

Лишние чтения пропали, запрос ускорился.