

Авторы: Егор Рогов, Павел Лузанов



**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

Исправление неэффективностей

- каким-то образом найти и исправить узкое место
- бывает сложно догадаться, 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 — получить адекватный план



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 отсутствует в явном виде 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 (начиная с версии 10).

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.

# 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. Для SSD-дисков значение этого параметра необходимо значительно уменьшать.

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 кэш при сканировании по битовой карте.

https://postgrespro.ru/docs/postgresql/10/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.

Это может влиять на порядок вычисления выражений в условии WHERE. Например, если условие выглядит как f() AND g(), то сначала будет вычислена наименее дорогая функция: если результат будет ложным, то более дорогую функцию вовсе не придется вычислять.

# 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

Память

*work_mem* = 4MB ↑
*maintenance_work_mem* = 64MB
*effective_cache_size* = 4GB

Запросы с большим количеством соединений

*join_collapse_limit* = 8
*from_collapse_limit* = 8
*geqo_threshold* = 12
и другие параметры *geqo*



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.

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

Параметр *effective_cache_size* подсказывает PostgreSQL общий объем кэшируемых данных (как в буферном кэше, так и в кэше ОС). Чем он больше, тем более предпочтительным будет индексный доступ.

Число различных возможных планов выполнения запроса растет экспоненциально с ростом числа соединений. Начиная с некоторого момента поиск лучшего плана (используется метод динамического программирования с дополнительными эвристическими ограничениями) начинает занимать слишком много времени.

Планировщик перебирает различные порядки соединения только для первых *join_collapse_limit* соединений (и для первых *from_collapse_limit* подзапросов в предложении FROM).

Если же число соединений достигает *geqo_threshold*, планировщик переключается на использование генетического алгоритма.

https://postgrespro.ru/docs/postgresql/10/geqo-pg-intro

# Настройки

Локализация влияния

ALTER DATABASE SET …
ALTER ROLE [IN DATABASE …] SET …
ALTER FUNCTION SET …
SET [LOCAL] …

Отладка

*enable_seqscan*
*enable_indexscan*, *enable_bitmapscan*, *enable_indexonlyscan*
*enable_nestloop*, *enable_hashjoin*, *enable_mergejoin*
*enable_hashagg*, *enable_sort*
*force_parallel_mode*



Обычно рассмотренные параметры имеет смысл устанавливать на уровне всей системы (или табличных пространств для ввода-вывода).

Однако есть возможность ограничить влияние параметров, устанавливая их для отдельной базы данных или отдельной роли (или же совокупности роли и базы данных). Можно также устанавливать параметры на уровне функции, что позволяет изолировать настройки отдельного запроса и менять их без вмешательства в исходный код. Разумеется, можно менять параметры и на уровне отдельного сеанса или транзакции.

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.

Параметры оказывают достаточно грубое влияние на планировщик, но оказываются весьма полезны для отладки и экспериментов.

# Схема данных

Нормализация — устранение избыточности в данных

- упрощает запросы и проверку согласованности

Денормализация — привнесение избыточности

- может повысить производительность, но требует синхронизации
- индексы
- предрассчитанные поля
- составные типы (массивы, JSON) вместо отдельных таблиц
- материализованные представления
- кэширование результатов вне СУБД (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. Такую информацию необходимо синхронизировать вручную — обычным способом являются триггеры, так что денормализация оказывается локализованной в базе данных и приложению не требуется «думать» об этом.

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

Дублировать данные можно и на уровне приложения, кэшируя результаты выполнения запросов. Это популярный способ, но часто к нему прибегают из-за неправильной работы приложения с базой данных (например, в случае использования ORM). В любом случае, на приложение ложится своевременное обновления кэша, обеспечение разграничения доступа к данным кэша 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. Разумеется, это не универсальный подход.

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.

# Физическое расположение

Табличные пространства

разнесение данных по разным физическим устройствам

Секционирование

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



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 не поддерживает такую возможность, для этого требуются сторонние решения. Обзорно этот вопрос рассматривается в последней теме курса DBA3.

# Изменение запросов

## Управление порядком соединений

материализация подзапросов с помощью CTE

явные соединения (JOIN) и *join_collapse_limit* = 1

подзапросы в предложении FROM и *from_collapse_limit* = 1

## 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.



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.

Если проблема имеющегося запроса видится в неправильном порядке соединения (и нет возможности исправить статистику), можно навязать планировщику необходимый порядок:

- Подзапросы CTE всегда материализуются: подзапрос не раскрывается, его результат вычисляется и помещается либо в оперативную память (в пределах *work_mem*), либо сбрасывается во временный файл.

- При значении параметра join_collapse_limit = 1 планировщик не перебирает порядок соединений — таблицы будут соединяться ровно в том порядке, в котором они указаны в тексте запроса.

- Аналогично и подзапросы не будут раскрываться при значении *from_collapse_limit* = 1.

Иногда требуются более радикальные изменения:

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

- Переписывание коррелированных подзапросов на соединения.

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

# Изменение запросов

Замена процедурного кода декларативным

- чтобы избавиться от большого числа мелких запросов

Подсказки оптимизатору

- отсутствуют, но есть другие средства

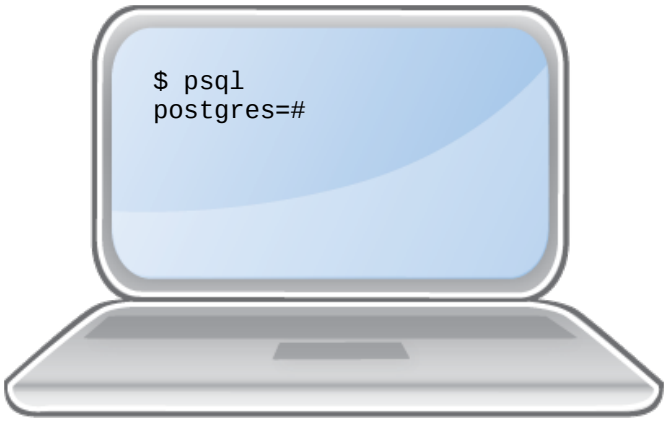


Если приложение выполняет слишком большое количество мелких запросов (каждый из которых сам по себе выполняется эффективно), общая эффективность будет очень невысока. Со стороны СУБД практически нет средств для оптимизации в таких ситуациях (кроме подготовленных операторов и кэширования).

В этом случае 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.

Еще один (традиционный для других СУБД) способ влияния — подсказки оптимизатору — отсутствует в PostgreSQL: https://wiki.postgresql.org/wiki/OptimizerHintsDiscussion. На самом деле часть подсказок существует в виде конфигурационных параметров, а кроме того есть специальные расширения, например https://postgrespro.ru/docs/enterprise/10/pg-hint-plan (автор — Киотаро Хоригучи). Но нельзя  забывать, что использование подсказок, сильно ограничивающих свободу планировщика, может навредить в будущем, когда распределение данных изменится.

# Демонстрация



# 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 со следующего варианта:

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