

Оптимизация запросов Профилирование



Авторские права

© Postgres Professional, 2019–2022

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

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.

Профилирование как инструмент для поиска узких мест

Выбор подзадачи для профилирования

Средства построения профиля

Профилирование

- выделение подзадач
- продолжительность
- количество выполнений

Что оптимизировать?

- чем больше доля подзадачи в общем времени выполнения, тем больше потенциальный выигрыш
- необходимо учитывать затраты на оптимизацию
- полезно взглянуть на задачу шире

В предыдущих темах мы разобрались с тем, как работают запросы, из каких «кирпичиков» строится план выполнения и что влияет на выбор того или иного плана. Это самое сложное и важное. Поняв механизмы, с помощью логики и здравого смысла можно разобраться в любой возникшей ситуации и понять, эффективно ли выполняется запрос и что можно сделать, чтобы улучшить показатели.

Но как найти тот запрос, который имеет смысл оптимизировать?

В принципе, решение любой задачи оптимизации (не только в контексте СУБД) начинается с профилирования, хоть этот термин и не всегда употребляется явно. Мы должны разбить задачу, вызывающую нарекания, на подзадачи и измерить, какую часть общего времени они занимают. Также полезна информация о числе выполнений каждой из подзадач.

Чем больше доля подзадачи в общем времени, тем больше выигрыш от оптимизации именно этой подзадачи. На практике приходится учитывать и ожидаемые затраты на оптимизацию: получить потенциальный выигрыш может оказаться нелегко.

Бывает так, что подзадача выполняется быстро, но часто (это, например, характерно для запросов, генерируемых ORM-ами). Может оказаться невозможным ускорить выполнение отдельных запросов, но стоит задаться вопросом: должна ли подзадача выполняться *так* часто? Такой вопрос может привести к необходимости изменения архитектуры, что всегда непросто, но в итоге дать существенный выигрыш.

Всю активность системы

полезно администратору для поиска ресурсоемких задач
мониторинг, расширение pg_profile

Отдельную задачу, вызывающую нарекания

полезно для решения конкретной проблемы
чем точнее, тем лучше: широкий охват размывает картину

Профиль общей активности может многое сказать администратору, который занят не решением конкретной проблемы, а поиском наиболее ресурсоемких задач, оптимизация которых, вероятно, снизит нагрузку на систему.

Такой профиль должна показывать система мониторинга.

Можно также воспользоваться расширением pg_profile. Оно основано на статистических представлениях PostgreSQL, информация из которых сохраняется в хранилище снимков активности. Изучая и сравнивая между собой снимки, можно выявлять проблемы и причины их возникновения. Автор расширения — Андрей Зубков (https://github.com/zubkov-andrei/pg_profile).

Для решения конкретной проблемы необходимо строить профиль так, чтобы измерения затрагивали только действия, необходимые для воспроизведения именно этой проблемы. Если, например, пользователь жалуется на то, что «окно открывается целую минуту», бессмысленно смотреть на всю активность в базе данных за эту минуту: в эти цифры попадут действия, не имеющие никакого отношения к открытию окна.

Время выполнения

- имеет смысл для пользователя
- крайне нестабильный показатель

Страничный ввод-вывод

- стабилен по отношению ко внешним факторам
- мало что говорит пользователям

В каких единицах измерять ресурсы? Самая осмысленная для конечного пользователя характеристика — это **время** отклика: сколько прошло времени «от нажатия на кнопку» до «получения результата».

Однако с технической точки зрения смотреть на время не всегда удобно. Оно сильно зависит от массы внешних факторов: от наполненности кеша, от текущей загруженности сервера. Если поиском проблемы занимаются не на основном, а на тестовом сервере с иными характеристиками, то добавляется и разница в аппаратном обеспечении, в настройках, в профиле нагрузки.

В этом смысле может оказаться удобнее смотреть на **ВВОД-ВЫВОД**, то есть на число прочитанных и записанных страниц. Этот показатель более стабилен. Он не зависит от характеристик «железа» и от нагрузки, то есть будет совпадать на разных серверах при условии одинаковых данных.

Поэтому очень важно анализировать проблемы на полном наборе данных. Для этого, например, можно использовать Database Lab Engine — средство быстрого создания тонких клонов, разработанное Postgres.ai: <https://github.com/postgres-ai/database-lab-engine>.

Ввод-вывод, как правило, адекватно отражает объем работы, необходимый для выполнения запроса, поскольку основное время уходит на чтение и обработку страниц с данными. Хотя такой показатель не имеет смысла для конечного пользователя.

Подзадачи

клиентская часть
сервер приложений
сервер баз данных ← *проблема часто, но не всегда, именно здесь*
сеть

Как профилировать

технически трудно, нужны разнообразные средства мониторинга
подтвердить предположение обычно несложно

Для пользователя имеет смысл время отклика. Это означает, что в профиль, вообще говоря, должна входить не только СУБД, но и клиентская часть, и сервер приложений, и передача данных по сети.

Часто проблемы с производительностью кроются именно в СУБД, поскольку один неадекватно построенный план запроса может увеличивать время на порядки. Но это не всегда так. Проблема может оказаться в том, что у клиента медленное соединение с сервером, что клиентская часть долго отрисовывает полученные данные и т. п.

К сожалению, получить такой полный профиль достаточно трудно. Для этого все компоненты информационной системы должны быть снабжены подсистемами мониторинга и трассировки, учитывающими особенности именно этой системы. Но обычно несложно измерить общее время отклика (хотя бы и секундомером) и сравнить с общим временем работы СУБД; убедиться в отсутствии существенных сетевых задержек и т. п. Если же этого не проделать, то вполне может оказаться, что мы будем искать ключи там, где светлее, а не там, где их потеряли. Собственно, весь смысл профилирования и состоит в том, чтобы выяснить, что именно нуждается в оптимизации.

Далее мы будем считать установленным, что проблема именно в СУБД.

Расширение PL Profiler

- стороннее расширение
- предназначено только для функций на PL/pgSQL
- профилирование отдельного скрипта или сеанса
- отчет о работе в html, включая flame graph

7

При выполнении пользователем действия, выполняется обычно не один, а несколько запросов. Как определить тот запрос, который имеет смысл оптимизировать? Для этого нужен профиль, детализированный до запросов.

Однако серверная часть приложения, выполняющаяся в СУБД, может состоять не только из SQL-запросов, но и содержать процедурный код. Если код написан на языке PL/pgSQL, то для его профилирования можно воспользоваться сторонним расширением PL Profiler (основной автор – Ян Вик: <https://github.com/bigsql/plprofiler>).

Расширение позволяет профилировать как отдельно выполняемые скрипты, так и снимать профиль работающего сеанса. Результат работы формируется как html-файл, содержащий необходимую информацию. В том числе он включает изображение вызовов в виде «языков пламени» (flame graph).

Поскольку курс посвящен оптимизации запросов SQL, мы не будем рассматривать это расширение, но упоминаем его для полноты картины.

Журнал сообщений сервера

включается конфигурационными параметрами:

log_min_duration_statements = 0 — время и текст всех запросов

log_line_prefix — идентифицирующая информация

сложно включить для отдельного сеанса

большой объем; при увеличении порога времени теряем информацию

не отслеживаются вложенные запросы

(можно использовать расширение *auto_explain*)

анализ внешними средствами, такими, как *pgBadger*

Для получения профиля по выполняемым SQL-запросам есть два основных средства, встроенных в PostgreSQL: журнал сообщений сервера и статистика.

В журнале сообщений с помощью конфигурационных параметров можно включить вывод информации о запросах и времени их выполнения. Обычно для этого используется параметр *log_min_duration_statement*, хотя есть и другие.

Как локализовать в журнале те запросы, которые относятся к действиям пользователя? Имело бы смысл включать и выключать параметр для конкретного сеанса, но штатных средств для этого не предусмотрено. Можно фильтровать общий поток сообщений, выделяя только нужные; для этого удобно в параметр *log_line_prefix* вывести дополнительную идентифицирующую информацию. Использование пула соединений еще больше усложняет ситуацию.

Для анализа вложенных запросов (когда в запросе вызывается функция, содержащая другие запросы) потребуется модуль *auto_explain* (<https://postgrespro.ru/docs/postgrespro/13/auto-explain>).

Следующая проблема — анализ журнала. Это осуществляется внешними средствами, из которых стандартом де-факто является расширение *pgBadger* (<https://github.com/darold/pgbadger>).

Разумеется, можно выводить в журнал и собственные сообщения, если они могут помочь делу.

Расширение pg_stat_statements

подробная информация о запросах в представлении
(в том числе о вложенных)

ограниченный размер хранилища

запросы считаются одинаковыми «с точностью до констант»,
даже если имеют разные планы выполнения

идентификация только по имени пользователя и базе данных

Второй способ — статистика, а именно расширение pg_stat_statements (<https://postgrespro.ru/docs/postgresql/13/pgstatstatements>).

Расширение собирает достаточно подробную информацию о выполняемых запросах (в том числе в терминах ввода-вывода страниц) и отображает ее в представлении pg_stat_statements.

Поскольку число различных запросов может быть очень большим, размер хранилища ограничен конфигурационным параметром; в нем остаются наиболее часто используемые запросы.

При этом «одинаковыми» считаются запросы, имеющие одинаковое (с точностью до констант) дерево разбора. Надо иметь в виду, что такие запросы могли иметь разные планы выполнения и выполняться разное время.

К сожалению, есть сложности и с идентификацией: запросы можно отнести к конкретному пользователю и базе данных, но не к сеансу.

Расширение pg_stat_statements

Подключим расширение pg_stat_statements, с помощью которого будем строить профиль:

```
=> CREATE EXTENSION pg_stat_statements;

CREATE EXTENSION

=> ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements';

ALTER SYSTEM
```

Подключение библиотеки требует перезагрузки.

```
student$ sudo pg_ctlcluster 13 main restart

student$ psql demo
```

Настроим расширение так, чтобы собиралась информация о всех запросах, в том числе вложенных:

```
=> SET pg_stat_statements.track = 'all';

SET
```

Создадим представление, чтобы смотреть на собранную статистику выполнения операторов:

```
=> CREATE VIEW statements_v AS
SELECT substring(regexp_replace(query, ' +', ' ', 'g') FOR 55) AS query,
       calls,
       round(total_exec_time)/1000 AS time_sec,
       shared_blks_hit + shared_blks_read + shared_blks_written AS shared_blks
FROM pg_stat_statements
ORDER BY total_exec_time DESC;

CREATE VIEW
```

Здесь мы для простоты показываем только часть полей и выводим только общее число страниц кеша.

Профиль выполнения

Рассмотрим задачу. Требуется построить отчет, выводящий сводную таблицу количества перевезенных пассажиров: строками отчета должны быть модели самолетов, а столбцами — категории обслуживания.

Сначала создадим функцию, возвращающую число пассажиров для заданной модели и категории обслуживания:

```
=> CREATE FUNCTION qty(aircraft_code char, fare_conditions varchar)
RETURNS bigint AS $$
SELECT count(*)
FROM flights f
JOIN boarding_passes bp ON bp.flight_id = f.flight_id
JOIN seats s ON s.aircraft_code = f.aircraft_code AND s.seat_no = bp.seat_no
WHERE f.aircraft_code = qty.aircraft_code AND s.fare_conditions = qty.fare_conditions;
$$ STABLE LANGUAGE sql;

CREATE FUNCTION
```

Для отчета создадим функцию, возвращающую набор строк:

```
=> CREATE FUNCTION report()
RETURNS TABLE(model text, economy bigint, comfort bigint, business bigint)
AS $$
DECLARE
    r record;
BEGIN
    FOR r IN SELECT a.aircraft_code, a.model FROM aircrafts a ORDER BY a.model LOOP
        report.model := r.model;
        report.economy := qty(r.aircraft_code, 'Economy');
        report.comfort := qty(r.aircraft_code, 'Comfort');
        report.business := qty(r.aircraft_code, 'Business');
        RETURN NEXT;
    END LOOP;
END;
$$ STABLE LANGUAGE plpgsql;

CREATE FUNCTION
```

Теперь исследуем работу предложенной реализации. Сбросим статистику:

```
=> SELECT pg_stat_statements_reset();
```

```
pg_stat_statements_reset
```

(1 row)

```
=> SELECT * FROM report();
```

model	economy	comfort	business
Аэробус A319-100	337129	0	70232
Аэробус A320-200	0	0	0
Аэробус A321-200	649388	0	127982
Боинг 737-300	589901	0	59829
Боинг 767-300	817621	0	127947
Боинг 777-300	895896	132571	83080
Бомбардье CRJ-200	1155683	0	0
Сессна 208 Караван	111096	0	0
Сухой Суперджет-100	2425034	0	342423

(9 rows)

Посмотрим, какую статистику мы получили:

```
=> SELECT * FROM statements_v \gx
```

```
-[ RECORD 1 ]-----
query      | SELECT * FROM report()
calls      | 1
time_sec   | 19.911
shared_blks| 922122
-[ RECORD 2 ]-----
query      | SELECT count(*)
           | FROM flights f
           | JOIN boarding_passes
calls      | 27
time_sec   | 19.866
shared_blks| 921621
-[ RECORD 3 ]-----
query      | SELECT a.aircraft_code, a.model FROM aircrafts a ORDER
calls      | 1
time_sec   | 0.002
shared_blks| 23
-[ RECORD 4 ]-----
query      | SELECT pg_stat_statements_reset()
calls      | 1
time_sec   | 0
shared_blks| 0
```

- Первым идет основной запрос, который детализируется ниже. К сожалению, нет признака, вложенный это запрос или внешний.
- Вторым идет запрос из функции qty — он вызывался 27 раз.
- Третий — запрос, по которому работает цикл.

EXPLAIN ANALYZE

подзадачи — узлы плана
продолжительность — actual time
или страничный ввод-вывод — buffers
количество выполнений — loops

Особенности

помимо наиболее ресурсоемких узлов, кандидаты на оптимизацию — узлы с большой ошибкой прогноза кардинальности
любое вмешательство может привести к полной перестройке плана
иногда приходится довольствоваться простым EXPLAIN

11

Так или иначе, среди выполненных запросов мы находим тот, который будем оптимизировать. Как работать с самим запросом? Тут тоже поможет профиль, который выдает команда EXPLAIN ANALYZE.

Подзадачами такого профиля являются узлы плана (план — не плоский список, а дерево). Продолжительность выполнения узла и число его повторений покажет time и loops из блока значений actual. С помощью параметра buffers можно получить объем ввода-вывода.

Кроме перечисленного, план выполнения содержит дополнительную — и крайне важную — информацию об ожидании оптимизатора относительно кардинальности каждого шага. Как правило, если нет серьезной ошибки в прогнозе кардинальности, то и план будет построен адекватный (если нет, то надо менять глобальные настройки). Поэтому стоит обращать внимание не только на наиболее ресурсоемкие узлы, но и на узлы с большой (на порядок или выше) ошибкой. Смотреть надо на наиболее вложенный проблемный узел, поскольку ошибка будет распространяться от него выше по дереву.

Бывают ситуации, когда запрос выполняется так долго, что выполнить EXPLAIN ANALYZE не представляется возможным. В таком случае придется довольствоваться простым EXPLAIN и пытаться разобраться в причине неэффективности без полной информации.

Работать с большими планами в текстовом виде не всегда удобно. Для лучшей наглядности можно пользоваться сторонними инструментами, из которых отметим <https://explain.tensor.ru/about>.

Профиль одного запроса

Попробуем теперь вывести тот же отчет одним запросом, и посмотрим на его выполнение командой EXPLAIN с параметрами analyze и buffers:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
WITH t AS (
  SELECT f.aircraft_code,
    count(*) FILTER (WHERE s.fare_conditions = 'Economy') economy,
    count(*) FILTER (WHERE s.fare_conditions = 'Comfort') comfort,
    count(*) FILTER (WHERE s.fare_conditions = 'Business') business
  FROM flights f
    JOIN boarding_passes bp ON bp.flight_id = f.flight_id
    JOIN seats s ON s.aircraft_code = f.aircraft_code AND s.seat_no = bp.seat_no
  GROUP BY f.aircraft_code
)
SELECT a.model,
  coalesce(t.economy,0) economy,
  coalesce(t.comfort,0) comfort,
  coalesce(t.business,0) business
FROM aircrafts a
  LEFT JOIN t ON a.aircraft_code = t.aircraft_code
ORDER BY a.model;
```

QUERY PLAN

```
-----
Sort (actual rows=9 loops=1)
  Sort Key: ((ml.model -> lang()))
  Sort Method: quicksort Memory: 25kB
  Buffers: shared hit=4073 read=56839, temp read=20572 written=20572
  -> Hash Left Join (actual rows=9 loops=1)
    Hash Cond: (ml.aircraft_code = t.aircraft_code)
    Buffers: shared hit=4073 read=56839, temp read=20572 written=20572
    -> Seq Scan on aircrafts_data ml (actual rows=9 loops=1)
      Buffers: shared hit=1
    -> Hash (actual rows=8 loops=1)
      Buckets: 1024 Batches: 1 Memory Usage: 9kB
      Buffers: shared hit=4072 read=56839, temp read=20572 written=20572
      -> Subquery Scan on t (actual rows=8 loops=1)
        Buffers: shared hit=4072 read=56839, temp read=20572 written=20572
        -> HashAggregate (actual rows=8 loops=1)
          Group Key: f.aircraft_code
          Batches: 1 Memory Usage: 24kB
          Buffers: shared hit=4072 read=56839, temp read=20572 written=20572
          -> Hash Join (actual rows=7925812 loops=1)
            Hash Cond: ((f.aircraft_code = s.aircraft_code) AND ((bp.seat_no)::text = (s.seat_no)::text))
            Buffers: shared hit=4072 read=56839, temp read=20572 written=20572
            -> Hash Join (actual rows=7925812 loops=1)
              Hash Cond: (bp.flight_id = f.flight_id)
              Buffers: shared hit=4064 read=56839, temp read=20572 written=20572
              -> Seq Scan on boarding_passes bp (actual rows=7925812 loops=1)
                Buffers: shared hit=1440 read=56839
              -> Hash (actual rows=214867 loops=1)
                Buckets: 131072 Batches: 4 Memory Usage: 3127kB
                Buffers: shared hit=2624, temp written=549
                -> Seq Scan on flights f (actual rows=214867 loops=1)
                  Buffers: shared hit=2624
            -> Hash (actual rows=1339 loops=1)
              Buckets: 2048 Batches: 1 Memory Usage: 79kB
              Buffers: shared hit=8
              -> Seq Scan on seats s (actual rows=1339 loops=1)
                Buffers: shared hit=8

Planning:
  Buffers: shared hit=45
Planning Time: 0.736 ms
Execution Time: 6646.134 ms
(40 rows)
```

Обратите внимание, как уменьшилось время выполнения запроса и насколько меньше страниц пришлось прочитать благодаря устранению избыточных чтений (верхняя строчка Buffers).

Можно посчитать общее количество страниц во всех задействованных таблицах:

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

sum
60911
(1 row)

Это число может служить грубой оценкой сверху для запроса, которому требуются все строки: обработка существенно большего числа страниц может говорить о том, что данные перебираются по несколько раз.

Видно, что в данном случае план близок к оптимальному. Его тоже можно улучшить, но уже не так радикально (очевидный момент — недостаток оперативной памяти для хеш-соединений).

Для поиска задач, нуждающихся в оптимизации, используется профилирование

Средства профилирования зависят от задачи:

- журнал сообщений сервера и `pg_stat_statements`
- `EXPLAIN ANALYZE`

1. Выполните первую версию отчета, показанного в демонстрации, включив вывод текста и времени выполнения запросов в журнал сообщений.
2. Посмотрите, какая информация попала в журнал сообщений.
3. Повторите предыдущие пункты, включив расширение `auto_explain` с выводом вложенных запросов.

1. Отчет

```
=> CREATE FUNCTION qty(aircraft_code char, fare_conditions varchar)
RETURNS bigint AS $$
    SELECT count(*)
    FROM flights f
        JOIN boarding_passes bp ON bp.flight_id = f.flight_id
        JOIN seats s ON s.aircraft_code = f.aircraft_code AND s.seat_no = bp.seat_no
    WHERE f.aircraft_code = qty.aircraft_code AND s.fare_conditions = qty.fare_conditions;
$$ STABLE LANGUAGE sql;
```

CREATE FUNCTION

```
=> CREATE FUNCTION report()
RETURNS TABLE(model text, economy bigint, comfort bigint, business bigint)
AS $$
DECLARE
    r record;
BEGIN
    FOR r IN SELECT a.aircraft_code, a.model FROM aircrafts a ORDER BY a.model LOOP
        report.model := r.model;
        report.economy := qty(r.aircraft_code, 'Economy');
        report.comfort := qty(r.aircraft_code, 'Comfort');
        report.business := qty(r.aircraft_code, 'Business');
        RETURN NEXT;
    END LOOP;
END;
$$ STABLE LANGUAGE plpgsql;
```

CREATE FUNCTION

Включаем вывод операторов и времени их выполнения в журнал:

```
=> SET log_min_duration_statement = 0;
```

SET

```
=> SELECT * FROM report();
```

model	economy	comfort	business
Аэробус A319-100	337129	0	70232
Аэробус A320-200	0	0	0
Аэробус A321-200	649388	0	127982
Боинг 737-300	589901	0	59829
Боинг 767-300	817621	0	127947
Боинг 777-300	895896	132571	83080
Бомбардье CRJ-200	1155683	0	0
Сессна 208 Караван	111096	0	0
Сухой Суперджет-100	2425034	0	342423

(9 rows)

2. Сообщения в журнале

```
postgres$ tail -n 1 /var/log/postgresql/postgresql-13-main.log
```

```
2022-07-03 15:22:58.407 MSK [169647] postgres@demo LOG: duration: 18962.884 ms statement: SELECT * FROM report();
```

Вложенные SQL-операторы не выводятся.

3. Расширение auto_explain

```
=> LOAD 'auto_explain';
```

LOAD

```
=> RESET log_min_duration_statement;
```

RESET

```
=> SET auto_explain.log_min_duration = 0;
```

SET

```
=> SET auto_explain.log_nested_statements = on;
```

SET

```
=> SELECT * FROM report();
```

model	economy	comfort	business
Аэробус A319-100	337129	0	70232
Аэробус A320-200	0	0	0
Аэробус A321-200	649388	0	127982
Боинг 737-300	589901	0	59829
Боинг 767-300	817621	0	127947
Боинг 777-300	895896	132571	83080
Бомбардье CRJ-200	1155683	0	0
Сессна 208 Караван	111096	0	0
Сухой Суперджет-100	2425034	0	342423

(9 rows)

Выведем несколько последних строк журнала сообщений:

```
postgres$ tail -n 30 /var/log/postgresql/postgresql-13-main.log
```

```

Finalize Aggregate (cost=106684.20..106684.21 rows=1 width=8)
-> Gather (cost=106683.99..106684.20 rows=2 width=8)
    Workers Planned: 2
    -> Partial Aggregate (cost=105683.99..105684.00 rows=1 width=8)
        -> Hash Join (cost=4417.66..105491.35 rows=77055 width=0)
            Hash Cond: ((bp.seat_no)::text = (s.seat_no)::text)
            -> Parallel Hash Join (cost=4401.39..104374.33 rows=412804 width=7)
                Hash Cond: (bp.flight_id = f.flight_id)
                -> Parallel Seq Scan on boarding_passes bp (cost=0.00..91303.77 rows=3302477 width=7)
                -> Parallel Hash (cost=4203.90..4203.90 rows=15799 width=8)
                    -> Parallel Seq Scan on flights f (cost=0.00..4203.90 rows=15799 width=8)
                    Filter: (aircraft_code = $1)
            -> Hash (cost=15.64..15.64 rows=50 width=7)
                -> Bitmap Heap Scan on seats s (cost=5.41..15.64 rows=50 width=7)
                    Recheck Cond: (aircraft_code = $1)
                    Filter: ((fare_conditions)::text = ($2)::text)
                    -> Bitmap Index Scan on seats_pkey (cost=0.00..5.39 rows=149 width=0)
                        Index Cond: (aircraft_code = $1)
2022-07-03 15:23:16.050 MSK [169647] postgres@demo CONTEXT: SQL function "qty" statement 1
PL/pgSQL function report() line 9 at assignment
2022-07-03 15:23:16.052 MSK [169647] postgres@demo LOG: duration: 0.091 ms plan:
Query Text: SELECT a.aircraft_code, a.model FROM aircrafts a ORDER BY a.model
Sort (cost=3.51..3.53 rows=9 width=36)
    Sort Key: ((ml.model ->> lang()))
    -> Seq Scan on aircrafts_data ml (cost=0.00..3.36 rows=9 width=36)
2022-07-03 15:23:16.052 MSK [169647] postgres@demo CONTEXT: PL/pgSQL function report() line 5 at FOR over SELECT rows
2022-07-03 15:23:16.052 MSK [169647] postgres@demo LOG: duration: 17342.440 ms plan:
Query Text: SELECT * FROM report();
Function Scan on report (cost=0.25..10.25 rows=1000 width=56)

```

В журнал попадают вложенные запросы и планы выполнения — собственно, это и есть основная функция расширения.