Лекция 11. Производительность БД и способы ее измерения

Корытов Павел

23 ноября 2023 г.

Производительность систем

Как правило, при увеличении количества данных системы начинают работать медленеее.

Что можно сделать, чтобы уменьшить вероятность проблем?

- Аккуратнее проектировать БД и архитектуру системы
  - В т.ч. с учётом используемых технологий
- Нагрузочное тестирование

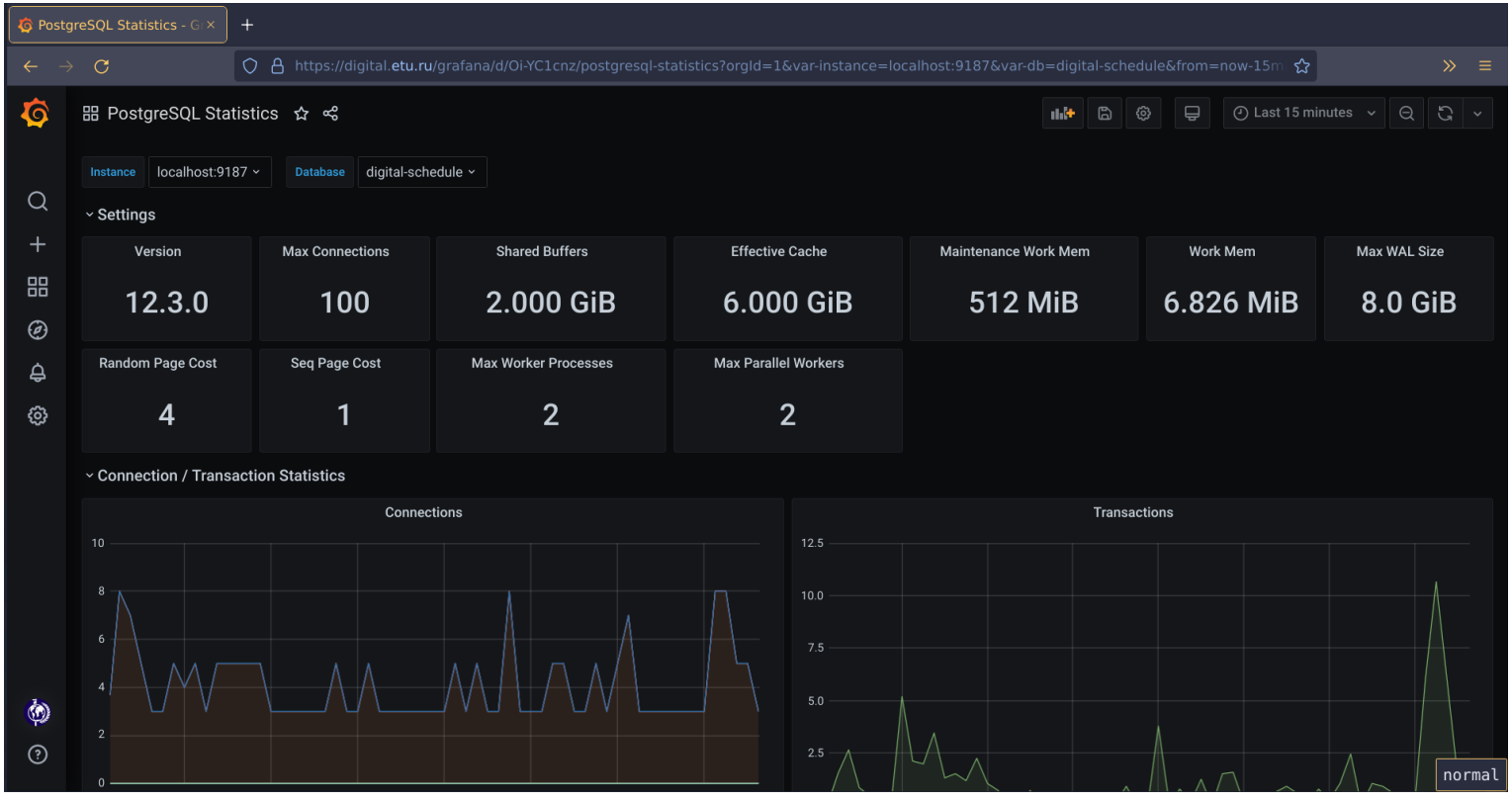
Производительность систем (2)

Но проблемы всё же возникают. Что делать?

- Проактивные методы
  Предотвращение или предвосхщение проблем
  - Мониторинг
- Реактивные методы
  Решение поступающих проблем
  - Профилирование запросов

Мониторинг

Мониторинг - снятие метрик с системы.



Мониторинг. Инфраструктура

За чем стоит следить?

Во-первых, инфраструктура, т.к. БД и системы не развернуты в вакууме.

Если VM/bare-metal server (IaaS):

- CPU
- RAM, swap
- Пространство на диске
- Траффик

Если контейнеры (e.g. Kubernetes, PaaS):

- Аналогичные метрики для потребление ресурсов контейнерами

Мониторинг. Доступность ресурсов

Во-вторых, доступность ресурсов.

Особенно верно в случае контейнеризованного развертывания!

- Доступность контейнера / VM с БД
  - доступность master node, slave nodes
- Доступность эндпоинта с БД
  - доступен ли порт 5432 для сервера PostgreSQL?

Мониторинг. Системы, обращающиеся к БД

В-третьих, БД обычно используются где-то дальше.

Если у вас web-приложение:

- Логи REST-запросов
- Логи сервера, обращающегося к БД
- Телеметрия с клиентов
  - …если соблюдены нормы этики

Цель:

- Определить, какие запросы к системе (пока не к БД) выполняются медленнее остальных
  - и какие запросы начинают выполнятся медленее по мере эксплуатации системы
- По возможности локализовать потенциальные проблемы
  - может быть, у клиента плохое соединение

Мониторинг. БД

В четвертых, снятие метрик непосредственно с БД.

Примеры метрик:

- Количество запросов на чтение/запись
  - Сколько ожидает?
- Размер БД
- Количество конфликтов
- Количество (и наличие!) тупиков
- Заблокированные данные
  - Блокировка таблиц
  - Блокировка отдельных строчек
- Использование:
  - Таблиц
  - Индексов
- …

Мониторинг. Инструменты

Что мы используем:

- prometheus - time-series СУБД
  - node_exporter - собирает метрики с машин
  - redis_exporter - собирает метрики с redis
  - postgres_exporter - собирает метрики с postgres
  - nestjs-prometheus и express-prom-bundle - node.js
  - prometheus-nginxlog-exporter - логи nginx
- Grafana - визуализация
- Uptime Kuma - доступность ресурсов
- pgAdmin - показывает статистику для PostgreSQL
- ClickHouse - OLAP СУБД, можно использовать для детального анализа логов

Ещё есть:

- ELK Stack
- …

Мониторинг. Postgres

В PostgreSQL есть расширение `pg_stat_statements`, собирающее статистику по запросам. Можно найти медленные запросы, но расширение замедлит работу БД.

Чтобы установить: запустить `postgres` с конфигурацией

```
1  postgres -c shared_preload_libraries=pg_stat_statements
```

После этого в нужной БД:

```
1  CREATE EXTENSION pg_stat_statements;
```

Будет создан `VIEW` ”pg_stat_statements”, в котором будут логироваться запросы и время их выполнения.

Сброс статистики:

```
1  SELECT pg_stat_statements_reset();
```

Мониторинг. Как использовать?

- Интересна динамика
  - Какие REST-запросы со временем начинают работать медленее?
  - Растет ли со временем потребление каких-либо ресурсов?
    - RAM (утечки памяти)
    - Подключения к БД
    - Время блокировки таблиц в БД
    - Время транзакций
- Рекомендуется настроить оповещения несколько заранее
  - Например, при заполнении диска на 80%

Реактивные методы

Что-то работает медленно. Что делать?

Во-первых, найти узкое место. Это необязательно будет БД!

- Могут быть ошибки в бизнес-логике
- Неэффективное использование ORM
- …

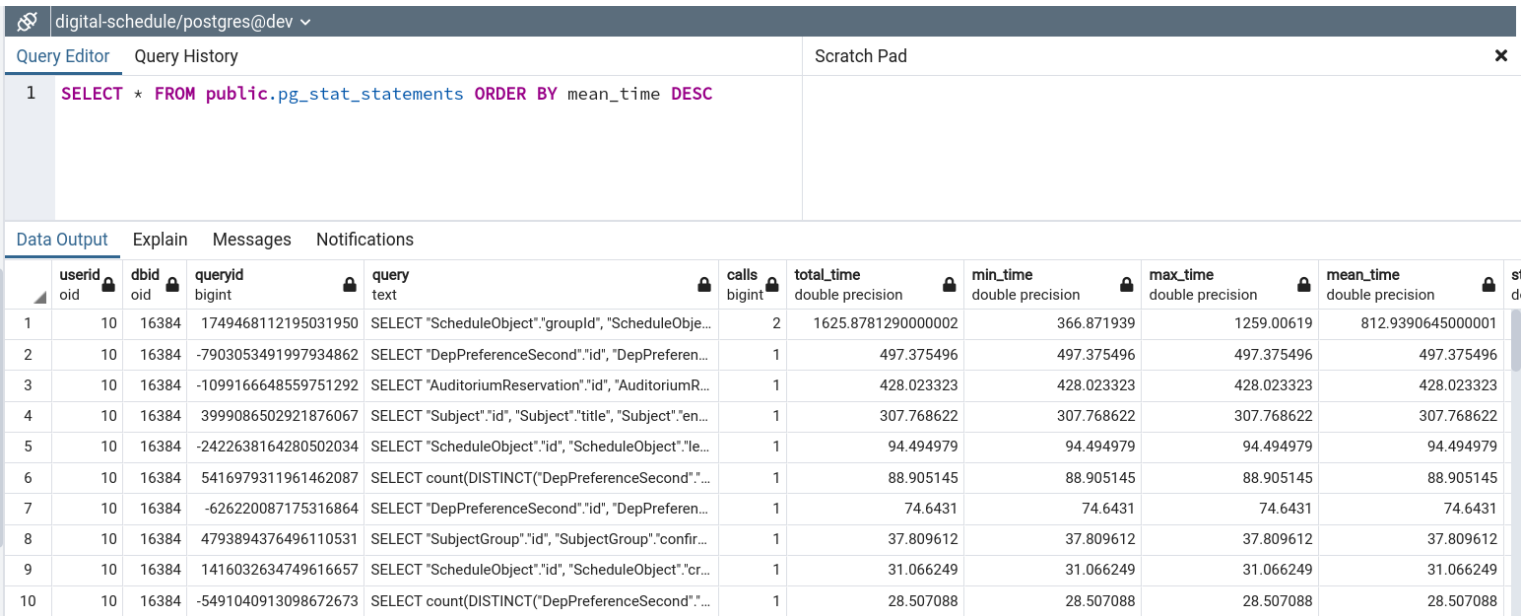
Для начала могут помочь стандартные средства профилирования для технологий вашего приложения.

BENCHMARK

В PostgreSQL измерение времени выполнения - BENCHMARK.

```
1    SELECT BENCHMARK(10, expr);
```

Можно включить логирование медленных запросов с помощью pg_stat_s



	userid oid	dbid oid	queryid bigint	query text	calls bigint	total_time double precision	min_time double precision	max_time double precision	mean_time double precision
1	10	16384	1749468112195031950	SELECT "ScheduleObject"."groupId", "ScheduleObje...	2	1625.8781290000002	366.871939	1259.00619	812.9390645000001
2	10	16384	-7903053491997934862	SELECT "DepPreferenceSecond"."id", "DepPreferen...	1	497.375496	497.375496	497.375496	497.375496
3	10	16384	-1099166648559751292	SELECT "AuditoriumReservation"."id", "AuditoriumR...	1	428.023323	428.023323	428.023323	428.023323
4	10	16384	3999086502921876067	SELECT "Subject"."id", "Subject"."title", "Subject"."en...	1	307.768622	307.768622	307.768622	307.768622
5	10	16384	-2422638164280502034	SELECT "ScheduleObject"."id", "ScheduleObject"."le...	1	94.494979	94.494979	94.494979	94.494979
6	10	16384	5416979311961462087	SELECT count(DISTINCT("DepPreferenceSecond"."...	1	88.905145	88.905145	88.905145	88.905145
7	10	16384	-626220087175316864	SELECT "DepPreferenceSecond"."id", "DepPreferen...	1	74.6431	74.6431	74.6431	74.6431
8	10	16384	4793894376496110531	SELECT "SubjectGroup"."id", "SubjectGroup"."confir...	1	37.809612	37.809612	37.809612	37.809612
9	10	16384	1416032634749616657	SELECT "ScheduleObject"."id", "ScheduleObject"."cr...	1	31.066249	31.066249	31.066249	31.066249
10	10	16384	-5491040913098672673	SELECT count(DISTINCT("DepPreferenceSecond"."...	1	28.507088	28.507088	28.507088	28.507088

План выполнения запроса. EXPLAIN

План выполнения запроса - описание, каким образом СУБД будет выполнять данный запрос.

```
1   EXPLAIN SELECT ...
```

План имеет древовидную структуру:

- На нижнем уровне - операции сканирования, возвращающие строки из таблиц или результаты выполнения операций.
- Затем - операции соединения, агрегации, сортировки

EXPLAIN не выполняет запрос, но только возвращает план, как запрос будет выполнятся.

EXPLAIN ANALYZE выполняет запрос и возвращает соответствие ожидаемой и действительной сложности выполнения.

EXPLAIN

```
1  EXPLAIN SELECT * FROM "AuditoriumReservations"
```

```
1  Seq Scan on "AuditoriumReservations"  (cost=0.00..11333.40
↪  rows=342340 width=156)
```

Здесь цифры:

- `cost=X...Y` - ожидаемая стоимость от начала выполнения операции до получения всех строчек. Ожидается, что операция пройдет до конца
- `rows` - ожидаемое число строк в выводе, если операция пройдет до конца
- `width` - ожидаемая средняя ширина строки на выходе операции (в байтах)

Для таких операций планировщик может смотреть статистику по таблице. Например, здесь 342340 - настоящее число строчек.

EXPLAIN. Планирование запросов (1)

Добавим LIMIT:

```
1  EXPLAIN SELECT * FROM "AuditoriumReservations" LIMIT 1000
```

```
1  Limit  (cost=0.00..33.11 rows=1000 width=156)
2    ->  Seq Scan on "AuditoriumReservations"
     ↪   (cost=0.00..11333.40 rows=342340 width=156)
```

EXPLAIN. (2)

Добавим сортировку:

```
1  EXPLAIN SELECT * FROM "AuditoriumReservations"
2  ORDER BY "auditoriumNumber" DESC
3  LIMIT 1000
```

Кажется, что операция должна быть очень тяжелой, но:

```
1  Limit  (cost=0.42..122.40 rows=1000 width=156)
2    ->  Index Scan Backward using
     ↪   auditorium_reservations_auditorium_number on
     ↪   "AuditoriumReservations"  (cost=0.42..41758.99 rows=342340
     ↪   width=156)
```

Планировщик использует индекс.

EXPLAIN. (3)

Если сортировать по полю без индекса:

```
1  EXPLAIN SELECT * FROM "AuditoriumReservations"
2  ORDER BY "description" DESC
3  LIMIT 1000
```

Тогда действительно сильно тяжелее.

```
1  Limit  (cost=18157.35..18274.02 rows=1000 width=156)
2    ->  Gather Merge  (cost=18157.35..51442.80 rows=285284 width=156)
3          Workers Planned: 2
4          ->  Sort  (cost=17157.32..17513.93 rows=142642 width=156)
5                Sort Key: description DESC
6                ->  Parallel Seq Scan on "AuditoriumReservations"
                 ↪  (cost=0.00..9336.42 rows=142642 width=156)
```

EXPLAIN. (4)

Проверим оценки планировщика. Запрос по индексу:

```
1  EXPLAIN ANALYSE SELECT * FROM "AuditoriumReservations"
2  ORDER BY "auditoriumNumber" DESC
3  LIMIT 1000;
```

```
1  Limit  (cost=0.42..122.40 rows=1000 width=156) (actual
   ↪  time=0.018..0.265 rows=1000 loops=1)
2    ->  Index Scan Backward using
     ↪  auditorium_reservations_auditorium_number on
     ↪  "AuditoriumReservations""  (cost=0.42..41758.99 rows=342340
     ↪  width=156) (actual time=0.017..0.198 rows=1000 loops=1)"
3  Planning Time: 0.093 ms
4  Execution Time: 0.319 ms
```

EXPLAIN. (5)

Запрос без индекса:

```
1  EXPLAIN ANALYSE SELECT * FROM "AuditoriumReservations"
2  ORDER BY "description" DESC
3  LIMIT 1000;
```

```
1  Limit  (cost=18157.35..18274.02 rows=1000 width=156) (actual
   ↪  time=120.684..123.253 rows=1000 loops=1)
2    ->  Gather Merge  (cost=18157.35..51442.80 rows=285284 width=156)
     ↪  (actual time=120.683..126.229 rows=1000 loops=1)
3          Workers Planned: 2
4          Workers Launched: 2
5          ->  Sort  (cost=17157.32..17513.93 rows=142642 width=156)
           ↪  (actual time=108.698..108.753 rows=482 loops=3)
6                Sort Key: description DESC
7                Sort Method: top-N heapsort  Memory: 481kB
8                Worker 0:  Sort Method: top-N heapsort  Memory: 473kB
9                Worker 1:  Sort Method: top-N heapsort  Memory: 503kB
10               ->  Parallel Seq Scan on "AuditoriumReservations"
                 ↪  (cost=0.00..9336.42 rows=142642 width=156) (actual
                 ↪  time=0.015..28.259 rows=114011 loops=3)
11 Planning Time: 0.102 ms
12 Execution Time: 126.386 ms
```

EXPLAIN. (6)

Оценки сходятся:

Запрос	Предварительная оценка	Время (мс)
1	0.42..122.40	0.319
2	18157.35..18274.02	126.386

- `cost` измеряется в условных единицах, поэтому можно сравнивать отношения.
- Предварительная оценка планировщика не всегда сходится с реальным временем выполнения

EXPLAIN. WHERE (1)

Схожие запросы могут быть выполнен по-разному:

```
1  EXPLAIN SELECT * FROM "AuditoriumReservations"
2  WHERE id < 1000
```

Выбирается небольшое подможество строчек. PostgreSQL обращается непосредственно к индексу (b-tree).

```
1  Index Scan using "AuditoriumReservations_pkey" on
   ↪  "AuditoriumReservations"  (cost=0.42..1499.32 rows=1029 width=156)
2    Index Cond: (id < 1000)
```

EXPLAIN. WHERE (2)

Выбирается подможество строчек побольше, но не большинство строчек в таблице

```
1  EXPLAIN SELECT * FROM "AuditoriumReservations"
2  WHERE id < 10000
```

Используется Bitmap Scan

```
1  Bitmap Heap Scan on "AuditoriumReservations"  (cost=1228.31..9867.16
   ↪  rows=58308 width=156)
2    Recheck Cond: (id < 100000)
3    ->  Bitmap Index Scan on "AuditoriumReservations_pkey"
     ↪  (cost=0.00..1213.73 rows=58308 width=0)
4          Index Cond: (id < 100000)
```

2 шага:

- Сначала сканируется индекс и создается bitmap
  - bitmap - массив битов, каждый бит соответствует части индекса (т.е. каким-то кортежам в таблице)
  - если какой-то бит = 1 > этот блок нас интересует
  - на bitmap-ах можно эффективно делать AND, OR и т.п.
- По результатам этой операции забираются данные

EXPLAIN. WHERE (3)

Выбираем ещё больше строчек:

```
1  EXPLAIN SELECT * FROM "AuditoriumReservations"
2  WHERE id < 700000
```

Индекс вообще не используется:

```
1  Seq Scan on "AuditoriumReservations"  (cost=0.00..12189.25 rows=147411
   ↪  width=156)
2    Filter: (id < 700000)
```

EXPLAIN. JOIN (1)

Планировщик может выбрать одну из нескольких стратегий JOIN.

- Nested Loop Join
  Двойной цикл по 2-м таблицам. Нормально использовать, когда таблицы небольшие.
  Раньше была оптимизация Block Nested-Loop Join.
- Hash Join
  Данные одной из таблиц хэшируются и сравниваются с другой. Эффективнее, но есть ограничения:
  - Хэш должен влезать в память (можно [нужно!] настроить `work_mem`)
  - Возможно провести не по всем условиям (но JOIN по ”=” сработает)
- Merge Join
  Отношения сортируются по условию (возможно, с использованием индекса), сканируются параллельно и выбираются совпадающие строчки.

EXPLAIN. JOIN (2)

Как помочь разным стратегиям JOIN:

- Nested Loop Join
  Поможет индекс для внутреннего отношения
- Merge Join
  Помогут индексы по ключам, по которым идет соединение для обоих отношений
- Hash Join
  Никак

EXPLAIN. JOIN (3)

```
1  SELECT * FROM "AuditoriumReservations" AR
2  INNER JOIN "Lessons" L ON AR.id = L."auditoriumReservationId"
3  INNER JOIN "Schedules" S ON S.id = AR."scheduleId"#+end_src
```

Используется Hash Join:

```
1  Hash Join  (cost=2.95..45542.47 rows=304362 width=832)
2    Hash Cond: (ar."scheduleId" = s.id)
3    ->  Merge Join  (cost=1.56..44428.07 rows=348130 width=246)
4          Merge Cond: (ar.id = l."auditoriumReservationId")
5          ->  Index Scan using "AuditoriumReservations_pkey"" on
           ↪   ""AuditoriumReservations" ar  (cost=0.42..26478.79
           ↪   rows=342340 width=156)
6          ->  Index Scan using lessons_auditorium_reservation_id on
           ↪   "Lessons" l  (cost=0.42..14010.63 rows=348130 width=90)
7    ->  Hash  (cost=1.17..1.17 rows=17 width=586)
8          ->  Seq Scan on "Schedules" s  (cost=0.00..1.17 rows=17
           ↪   width=586)
```

EXPLAIN. JOIN (4)

Поменяем условие на эквивалентное:

```
1  EXPLAIN SELECT * FROM "AuditoriumReservations" AR
2  INNER JOIN "Lessons" L ON AR.id >= L."auditoriumReservationId" AND
   ↪  AR.id <= L."auditoriumReservationId"
3  INNER JOIN "Schedules" S ON S.id = AR."scheduleId"
```

```
1  Nested Loop  (cost=1.80..347471452.42 rows=11577258658 width=832)
2    ->  Hash Join  (cost=1.38..12429.29 rows=299300 width=742)
3          Hash Cond: (ar."scheduleId" = s.id)
4          ->  Seq Scan on "AuditoriumReservations" ar
           ↪  (cost=0.00..11333.40 rows=342340 width=156)
5          ->  Hash  (cost=1.17..1.17 rows=17 width=586)
6                ->  Seq Scan on "Schedules" s  (cost=0.00..1.17 rows=17
                 ↪  width=586)
7    ->  Index Scan using lessons_auditorium_reservation_id on "Lessons" l
     ↪  (cost=0.42..774.10 rows=38681 width=90)
8          Index Cond: (("auditoriumReservationId" <= ar.id) AND
           ↪  ("auditoriumReservationId" >= ar.id))
9  JIT:
10   Functions: 14
11   Options: Inlining true, Optimization true, Expressions true,
     ↪  Deforming true
```

Выполняется примерно в 3 раза медленее из-за Nested Loop.

EXPLAIN. JIT

В предыдущем запросе в плане появилась строчка ”JIT”. Это значит, что при планировании будет проведена Just-In-Time компиляция.

Если запустить EXPLAIN ANALYZE:

```
1  JIT:
2    Functions: 16
3    Options: Inlining true, Optimization true, Expressions true,
     ↪  Deforming true
4    Timing: Generation 2.280 ms, Inlining 5.033 ms, Optimization 164.019
     ↪  ms, Emission 101.650 ms, Total 272.982 ms
5  Execution Time: 1135.707 ms
```

На JIT тратится существенное время, но т.к. планировщик (ошибочно) ожидает получить 11577258658 строчек в Nested Join, он считает это ”меньшим злом”.

`SET jit = false` для данного запроса ускоряет его на примерно на 20%.

Что посмотреть

- Как использовать EXPLAIN в PostgreSQL - `https://www.postgresql.org/docs/14/using-explain.html`
- Queries in PostgreSQL - `https://postgrespro.com/blog/pgsql/5969262`

Интересные случаи:

- +1 к LIMIT замедляет запрос в 20000 раз: `https://stackoverflow.com/questions/57753380/why-is-postgres-query-planner-affected-by-limit`