PostgreSQL 16: лучшие практики оптимизации

УДК 004.65
ББК 16.35
М74

А81 Аристов Евгений. PostgreSQL 16: лучшие практики оптимизации. – М.: ООО «Сам Полиграфист», 2024. – 316 с.

ISBN 978-5-00227-223-5

В данной книге подробно изучается внутреннее устройство PostgreSQL 16, начиная с выбора сервера для установки, операционной системы, особенностей базы данных и вариантов их тюнинга и заканчивая последними тенденциями оптимизации production-систем.

Книга написана доступным языком с использованием подробных практических примеров. Главы логически выстроены по пути усложнения материала и позволяют разобраться в особенностях функционирования PostgreSQL. Исходные коды и ссылки, используемые в книге, выложены на Github.

В каждой главе поднимается проблематика разных областей архитектуры PostgreSQL, будь то особенности подключения и обеспечения безопасности доступа или тонкая настройка файловой системы, разбираются варианты решения проблем, оптимизация производительности, чек-листы и лучшие практики по использованию той или иной технологии.

Книга будет интересна и полезна широкой аудитории: разработчикам, администраторам баз данных, девопс-инженерам, архитекторам программного обеспечения, в том числе микросервисной архитектуры.

Тираж 120 экз. Заказ № 29763.

Отпечатано в типографии «OneBook.ru» ООО «Сам Полиграфист»
129090 г. Москва, Протопоповский пер., 6
www.onebook.ru

Оглавление

Об авторе

Для начала, хотелось бы поблагодарить за то, что вы выбрали эту книгу. Я в IT уже больше 25 лет. Начинал, как и многие когда-то, с ZX Spectrum. Написал свою первую игру наподобие текстовой Dictator. Тогда всё хранилось на простых аудиокассетах, а язык был Basic. Потом IBM 8086, хотя кто уже помнит те времена. Дальше DOS, первый Windows 3.11, профильный университет, базы данных большие и маленькие, сотни успешно реализованных проектов с применением виртуализации, кластеризации и highload в качестве Архитектора. В какой-то момент понял, что хочу учить людей, нести светлое и доброе в наш мир. Уже больше четырёх лет я читаю свои авторские курсы по Постгресу и другим направлениям во множестве обучающих организаций, институтов и университетов, пишу книги по базам данных.

За это время понял, насколько не хватает хорошей книги по практическому разбору встречаемых проблем, методам их решения и лучших практик по оптимизации производительности самой популярной СУБД[1] – PostgreSQL. В этой книге даются практические рекомендации, проведён глубокий анализ внутреннего устройства для понимания принципов работы этой СУБД и возможностей её тонкой настройки. Рассматриваются частые ошибки при проектировании и использовании PostgreSQL, а также лучшие варианты тюнинга (тонкой настройки).

В 2021 году создал личный сайт-визитку https://aristov.tech, где были только регалии и контакты. Сейчас сайт вырос до обучающего портала, на нём доступны: блог с интересными статьями, менторинг для ускоренного обучения почти по всем направлениям, канал YouTube с обучающими роликами и раздел с моим уникальным курсом по оптимизации PostgreSQL, на основании которого я и решил написать данную книгу, включив в неё часть обучающего контента из курса.

Эта книга написана под новую версию Постгреса – 16. Однако большинство из используемых подходов к решению проблем подойдут как к более старым версиям Постгреса, так и, что более важно, к новым версиям! А многие настройки ОС или примеры оптимизации запросов и к другим СУБД!

Ссылки из данной книги расположены на Github[2] для удобства перехода. В репозитории ссылки будут разбиты по номерам тем, номер ссылки в квадратных скобках [1] нумеруется отдельно для каждой главы. Также на сайте

[1] Система управления базами данных
[2] Мой github URL: https://github.com/aeuge/Postgres16book (дата обращения 16.01.2024)

https://aristov.tech можно заказать[3] оригинальную электронную версию в PDF или бумажную версию с автографом. Также можно заказать предыдущую книгу по развёртыванию отказоустойчивых решений Постгреса – PostgreSQL 14. Оптимизация, Kubernetes, кластера, облака. – М.: ООО «Сам Полиграфист», 2022. – 576 с. Отрывок из неё или текущей книги всегда можно найти на моём сайте и принять решение о заказе.

Спасибо за поддержку Илье @JorianR, который и положил начало моей преподавательской карьеры. Отличный комьюнити-менеджер. Спасибо Алексею Цыкунову @erlong15. Мой наставник, Профессионал с большой буквы, опытнейший DBA[4] и просто замечательный человек. Спасибо любимой жене Светлане @asveta за поддержку. Спасибо редакторам книги Павлу Андрееву @shaadowsky и Андрею Alandre @alandreei. Ну и, конечно же, любимой семье, которая меня поддерживала все эти долгие месяцы.

Просьба оставить отзыв о книге[5] на моём сайте. Любые мнения, пожелания, предложения по доработке приветствуются.

Желаю вам получить удовольствие от прочтения книги и удачи в карьере.

[3] ссылка для заказа книги https://aristov.tech/#orderbook
[4] DataBase Architector/Administrator
[5] https://aristov.tech/Постгрес-feedback

1. PostgreSQL 16. Настройка ВМ[6], ОС[7] и СУБД

Начну книгу я с девиза моего курса по оптимизации Постгреса: «Серебряной пули не существует». Если бы какой-то из вариантов любого тюнинга или параметра был оптимальным – его бы сразу выставляли и больше к нему не обращались. В книге разберём, что есть много вариантов оптимизации, какие-то параметры на самом деле нужно тюнить, какие-то, может быть, тоже, а может быть, нет – они приводят к улучшению одного показателя и к ухудшению другого.

Варианты развёртывания ВМ, установки PostgreSQL, использования docker[8] и Kubernetes[9] в этой книге рассматриваться не будут. Они подробно изложены в моей книге по PostgreSQL 14[10].

В данной главе будут разбираются подводные камни тюнинга железных серверов и ВМ, ОС, PostgreSQL, какие варианты бенчмарков рекомендовано использовать, как правильно измерять производительность.

Есть, например, huge pages, а есть transparent huge pages. В чём разница и что стоит включать, отключать и в каком направлении тюнить. Что делать, если SSD[11]/NVMe[12]-диск или Ceph[13] «упёрлись в полку»[14]?

Даже простой вопрос, что быстрее: count(1), count(*), count(id), count(UUID[15]) или count(field) вызывает сложности. Давайте разбираться.

В книге сейчас и далее используются сгенерированные мной БД[16] по автобусным пассажирским перевозкам в Тайланде[17] для PostgreSQL. Схема данных относительно простая – есть города, автобусные станции, автобусы, маршруты перевозок, расписание и сами перевозки.

ВМ будет состоять из 4 ядер, 16 гигабайт памяти и SSD-диска.

---

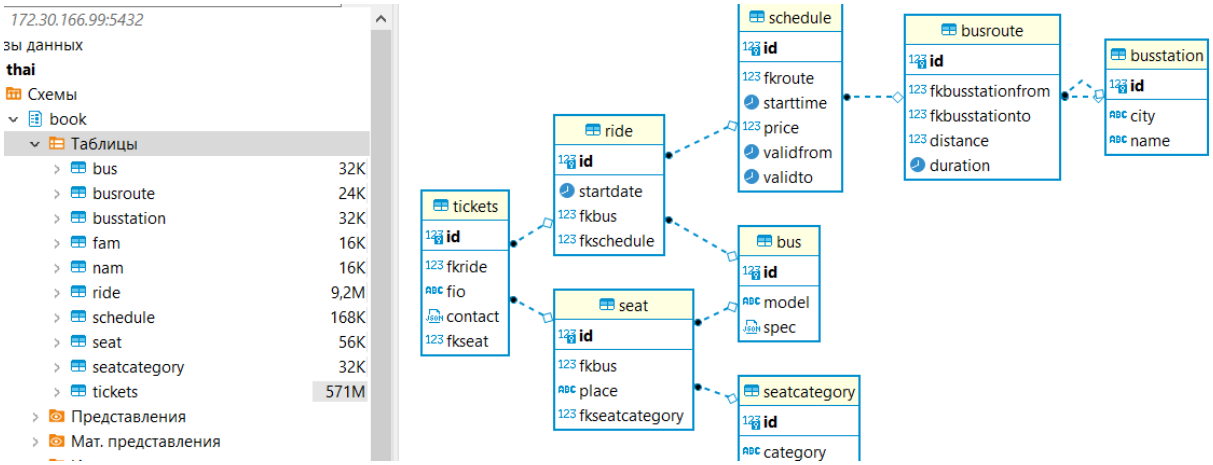
[6] Виртуальная машина
[7] Операционная система
[8] Docker URL: https://www.docker.com/ (дата обращения 05.01.2024) [1]
[9] Что такое Kubernetes URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (дата обращения 05.01.2024) [2]
[10] Аристов Евгений. PostgreSQL 14. Оптимизация, Kubernetes, кластера, облака. – М.: ООО «Сам Полиграфист», 2022. – 576 с. URL: https://aristov.tech/PostgreSQL14preview.pdf (дата обращения 05.01.2024) [3]
[11] SSD URL: https://en.wikipedia.org/wiki/Solid-state_drive (дата обращения 16.01.2024) [4]
[12] NVMe URL: https://en.wikipedia.org/wiki/NVM_Express (дата обращения 16.01.2024) [5]
[13] Ceph URL: https://en.wikipedia.org/wiki/Ceph_(software) (дата обращения 16.01.2024) [6]
[14] Достигли 100% утилизации ресурса и дальше некуда
[15] UUID URL: https://en.wikipedia.org/wiki/Universally_unique_identifier (дата обращения 16.01.2024) [7]
[16] База данных
[17] Тайские перевозки URL: https://github.com/aeuge/postgres16book/tree/main/database (дата обращения 16.01.2024) [8]

Всего предлагается три варианта: маленький, который будет в примерах, на ~6 млн строк (600 МБ[18]), средний на ~60 млн строк (6 ГБ[19]) и большой на ~600 млн строк (60 ГБ). Инструкции разворачивания остальных вариантов расположены на странице «Тайские перевозки» на Github.

Загрузить данные в инстанс PostgreSQL в базу данных **thai** можно довольно легко командой загрузки утилитой wget[20] заархивированного датасета с открытого google storage[21], его разархивирования утилитой tar[22] и перенаправления SQL-кода в PostgreSQL (знак перенаправления <), объединив всё через двойной амперсанд (&&):

```
wget https://storage.googleapis.com/thaibus/thai_small.tar.gz && tar -xf thai_small.tar.gz && psql < thai.sql
```

Схема данных загруженного датасета:



Объём БД можно посмотреть, используя команду **\l+** в утилите **psql**[23], входящей в сборку PostgreSQL, она нам как раз покажет объём загруженных данных. Для этого сначала укажем БД, к которой необходимо подключиться при запуске утилиты:

```
psql -d thai
\l+
```

```
thai      | postgres | UTF8     | C.UTF-8 | C.UTF-8 |                    | 590 MB  | pg_default |
```

[18] Мегабайт
[19] Гигабайт
[20] wget URL: https://en.wikipedia.org/wiki/Wget (дата обращения 16.01.2024) [9]
[21] Облачное хранилище URL: https://cloud.google.com/storage?hl=ru (дата обращения 16.01.2024) [10]
[22] tar URL: https://en.wikipedia.org/wiki/Tar_(computing) (дата обращения 16.01.2024) [11]
[23] psql URL: https://www.postgresql.org/docs/current/app-psql.html (дата обращения 16.01.2024) [12]

Чтобы ответить на вопрос, **что быстрее: count(*) или count(1)** – посмотрим планы наших запросов:

```
EXPLAIN SELECT count(*) FROM book.tickets;
EXPLAIN SELECT count(1) FROM book.tickets;
```

```
thai=# EXPLAIN SELECT count(*) FROM book.tickets;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=86934.59..86934.60 rows=1 width=8)
   ->  Gather  (cost=86934.38..86934.58 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=85934.38..85934.38 rows=1 width=8)
               ->  Parallel Seq Scan on tickets  (cost=0.00..80532.70 rows=2160670 width=0)
(5 rows)

thai=# EXPLAIN SELECT count(1) FROM book.tickets;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=86934.59..86934.60 rows=1 width=8)
   ->  Gather  (cost=86934.38..86934.58 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=85934.38..85934.38 rows=1 width=8)
               ->  Parallel Seq Scan on tickets  (cost=0.00..80532.70 rows=2160670 width=0)
(5 rows)
```

Видим абсолютно идентичные планы, даже с одинаковым количеством «**попугаев**» в **cost** – стоимости запроса (это не шутка, а профессиональный сленг, так как хоть цифры и красивые, но они не связаны напрямую со временем исполнения – нельзя сказать, что 1 миллион cost равен 1 секунде исполнения, можно сравнивать только между собой и примерно представлять, во что обойдётся запрос).

Вроде закрыли вопрос, но пытливый ум должен зацепиться за строку **Parallel Seq Scan** – получается, последовательно просматривается ВСЯ таблица на диске, чтобы посчитать количество строк. Как-то не оптимально. Может, нет индекса? Давайте посмотрим:

```
SELECT relname, pg_size_pretty(pg_relation_size(oid)) FROM pg_class WHERE relname like 'tickets%';
```

```
thai=# SELECT relname, pg_size_pretty(pg_relation_size(oid)) FROM pg_class
 WHERE relname like 'tickets%';
    relname     | pg_size_pretty
----------------+----------------
 tickets        | 461 MB
 tickets_id_seq | 8192 bytes
 tickets_pkey   | 111 MB
(3 rows)
```

А вот и индекс, при этом его размер в четыре с лишним раза меньше файла с данными. Так почему же PostgreSQL выбирает последовательное сканирование, а не сканирование индекса?

Чтобы ответить на этот вопрос, сначала почистим таблицу и посчитаем **статистику** – может быть из-за неё происходит данное поведение – используя утилиту **vacuum** (подробнее будем разбирать в главе 7):

```
VACUUM ANALYZE book.tickets;
```

```
thai=# VACUUM ANALYZE book.tickets;
VACUUM
thai=# EXPLAIN SELECT count(1) FROM book.tickets;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=86934.40..86934.41 rows=1 width=8)
   ->  Gather  (cost=86934.19..86934.40 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=85934.19..85934.20 rows=1 width=8)
               ->  Parallel Seq Scan on tickets  (cost=0.00..80532.55 rows=2160655 width=0)
(5 rows)
```

Ничего не изменилось.

А ответ на этот вопрос довольно интересен – это параметр **random_page_cost**.

Он задаёт приблизительную стоимость чтения одной произвольной страницы с диска. Значение по умолчанию равно 4.0. У твердотельных накопителей лучше выбрать меньшее значение **random_page_cost** – оптимально 1 ~ 1.1.

Переводя на русский, PostgreSQL думает, что будет очень дорого найти запись в индексе и проверить её наличие на диске в файле видимости данных для разных снимков данных **visibility map** (картой видимости, **VM**[24]).

Давайте установим этот **параметр в 1**, так как у меня на BM SSD-диск, и проверим план запроса:

```
SET random_page_cost = 1;
```

```
thai=# SET random_page_cost = 1;
SET
thai=# EXPLAIN SELECT count(1) FROM book.tickets;
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=68157.70..68157.71 rows=1 width=8)
   ->  Gather  (cost=68157.48..68157.69 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=67157.48..67157.49 rows=1 width=8)
               ->  Parallel Index Only Scan using tickets_pkey on tickets  (cost=0.43..61755.84 rows=2160655 width=0)
(5 rows)
```

[24] Visibility map URL: https://www.postgresql.org/docs/current/storage-vm.html (дата обращения 16.01.2024) [13]

Наконец, видим Parallel Index Only Scan и значительно меньшее количество попугаев (cost)!

То есть, когда есть индекс и корректно настроены параметры, Постгрес выбирает параллельный поиск по индексу для подсчёта элементов.

Думаете на этом всё? Нет, осталось обсудить ещё два важных момента.

Первое – разницу count(**index_filed**) и count(**field_without_index**).

Сначала посмотрим структуру таблицы book.tickets из psql, используя встроенную функцию **\d+**, где **+** позволяет получить расширенный вывод команды **\d**:

```
\d+ book.tickets:
```

```
                                                  Table "book.tickets"
 Column  |  Type   | Collation | Nullable |                 Default                  | Storage  | Compression |
---------+---------+-----------+----------+------------------------------------------+----------+-------------+
 id      | bigint  |           | not null | nextval('book.tickets_id_seq'::regclass) | plain    |             |
 fkride  | integer |           |          |                                          | plain    |             |
 fio     | text    |           |          |                                          | extended |             |
 contact | jsonb   |           |          |                                          | extended |             |
 fkseat  | integer |           |          |                                          | plain    |             |
Indexes:
    "tickets_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "tickets_fkride_fkey" FOREIGN KEY (fkride) REFERENCES book.ride(id)
    "tickets_fkseat_fkey" FOREIGN KEY (fkseat) REFERENCES book.seat(id)
Access method: heap
```

А теперь сравним **count(id)** и **count(fio)** (нет индекса по полю fio):

```
thai=# explain select count(id) from book.tickets;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=68157.74..68157.75 rows=1 width=8)
   ->  Gather  (cost=68157.52..68157.73 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=67157.52..67157.53 rows=1 width=8)
               ->  Parallel Index Only Scan using tickets_pkey on tickets  (cost=0.43..61755.88 rows=2160657 width=8)
(5 rows)

thai=# explain select count(fio) from book.tickets;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Finalize Aggregate  (cost=86934.42..86934.43 rows=1 width=8)
   ->  Gather  (cost=86934.21..86934.42 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=85934.21..85934.22 rows=1 width=8)
               ->  Parallel Seq Scan on tickets  (cost=0.00..80532.57 rows=2160657 width=15)
(5 rows)
```

Видим ожидаемое решение PostgreSQL при поиске количества записей для поля без индекса использовать полное сканирование.

Второе - ещё одна неочевидная вещь – **значение поля fio может быть NULL** и тогда это поле **не будет посчитано** в итоговом запросе!

Если попытаемся добавить ограничение **NOT NULL** на поле **fio,** то получим ошибку, так как часть значений NULL:

```
ALTER TABLE book.tickets ALTER COLUMN fio SET NOT NULL;
```

```
thai=# ALTER TABLE book.tickets ALTER COLUMN fio SET NOT NULL;
ERROR:  column "fio" of relation "tickets" contains null values
```

Попытаемся избавиться от NULL, поменяем структуру БД, объявив поле fio NOT NULL, и тогда не окажется вариантов, что запись не будет посчитана, ведь PostgreSQL сможет использовать первичный индекс. Какая разница, по какой колонке считать и зачем ему **seq scan**.

```
UPDATE book.tickets SET fio = 'no' WHERE fio is NULL;
ALTER TABLE book.tickets ALTER COLUMN fio SET NOT NULL;
EXPLAIN SELECT count(fio) FROM book.tickets;
```

```
thai=# UPDATE book.tickets SET fio = 'no' WHERE fio is NULL;
UPDATE 47607
thai=# ALTER TABLE book.tickets ALTER COLUMN fio SET NOT NULL;
ALTER TABLE
thai=# EXPLAIN SELECT count(fio) FROM book.tickets;
                                     QUERY PLAN
------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=87628.57..87628.58 rows=1 width=8)
   ->  Gather  (cost=87628.36..87628.57 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=86628.36..86628.37 rows=1 width=8)
               ->  Parallel Seq Scan on tickets  (cost=0.00..81183.09 rows=2178109 width=15)
(5 rows)
```

Видим, что план провалился – SEQ SCAN.

Вывод: аккуратнее используем выражение внутри COUNT, есть нюансы.

Следующий нюанс работы с COUNT связан с использованием поля типа UUID. Начнём с описания этой технологии.

Общая идея заключается в том, что UUID предоставляет способ генерации уникальных идентификаторов без необходимости зависеть от конкретных контекстов или ресурсов. В PostgreSQL для создания UUID можно использовать extension[25], **uuid-ossp**[26] и/или встроенную функцию **gen_random_uuid()**.

[25] extension URL: https://www.postgresql.org/docs/current/extend-extensions.html (дата обращения 16.01.2024) [14]
[26] uuid-ossp URL: https://www.postgresql.org/docs/current/functions-uuid.html (дата обращения 16.01.2024) [15]

UUID имеет несколько применений:

- **уникальные идентификаторы для записей**: вместо использования автоинкрементных числовых идентификаторов (SERIAL или BIGSERIAL), можно использовать UUID для уникальной идентификации записей в таблицах. Это особенно полезно в распределённых системах, где целостность данных может быть подвергнута риску из-за конфликтов при генерации идентификаторов;
- **распределённые системы и репликация**: UUID обеспечивает гарантию уникальности идентификаторов даже в распределённых системах или при репликации данных, где отсутствует единая централизованная точка генерации идентификаторов;
- **слияние данных**: если нужно объединить данные из разных источников, UUID может быть более надёжным способом идентификации записей, так как вероятность коллизий (совпадения идентификаторов) очень низка;
- **защита личных данных**: UUID может помочь обезличивать данные, так как UUID не содержит информацию о самой записи;
- **публичные идентификаторы**: UUID можно использовать как публичные идентификаторы для ресурсов в API или веб-приложениях;
- **переносимость данных**: UUID позволяет легко перемещать и синхронизировать данные между разными базами данных и системами;
- **замена недоступных идентификаторов**: в некоторых случаях (например, при интеграции с другими системами), идентификаторы разных систем могут быть одинаковыми или неудобными для использования, UUID хорошая альтернативой в таких сценариях.

Кажется, всё отлично с UUID?

Сравним генерацию 10 миллионов строк **UUID четвёртой версии**[27] + случайный текст с аналогичным классическим подходом **bigserial**[28] **(8 байт)**:

```
\timing
CREATE EXTENSION "uuid-ossp";
CREATE TABLE records2 (id int8 not null, filler text);
INSERT INTO records2 SELECT id, repeat(' ', 100) FROM generate_series(1, 10000000) id;
```

[27] UUID_v4 URL: https://gcore.com/dev-tools/gen-uuid-v4 (дата обращения 16.01.2024) [16]
[28] bigint URL: https://www.postgresql.org/docs/current/datatype-numeric.html (дата обращения 16.01.2024) [17]

```
postgres=# CREATE TABLE records2 (id int8 not null, filler text);
CREATE TABLE
Time: 11.994 ms
postgres=# INSERT INTO records2 SELECT id, repeat(' ', 100) FROM generate_series(1, 10000000) id;
INSERT 0 10000000
Time: 19663.504 ms (00:19.664)
```

А теперь тоже самое, но с генерацией UUID:

```
CREATE TABLE records3 (uuid_v4 uuid not null, filler text);
INSERT INTO records3 SELECT gen_random_uuid(), repeat(' ', 100) FROM generate_series(1, 10000000) id;
```

```
postgres=# CREATE TABLE records3 (uuid_v4 uuid not null, filler text);
CREATE TABLE
Time: 4.796 ms
postgres=# INSERT INTO records3 SELECT gen_random_uuid(), repeat(' ', 100) FROM generate_series(1, 10000000) id;
INSERT 0 10000000
Time: 35810.010 ms (00:35.810)
```

Время генерации выше почти в два раза!

Конечно, результаты на разном железе могут отличаться от представленных в книге. Я проводил много замеров и, в среднем, разница в 1,5-3 раза.

Сравним размер сгенерированных данных:

```
\dt+
```

```
postgres=# \dt+
                                    List of relations
 Schema |   Name   | Type  |  Owner   | Persistence | Access method |  Size   | Description
--------+----------+-------+----------+-------------+---------------+---------+-------------
 public | records2 | table | postgres | permanent   | heap          | 1347 MB |
 public | records3 | table | postgres | permanent   | heap          | 1421 MB |
(2 rows)
```

Видим, что размер отличается не сильно, так как размер самих данных значительно превышает размер ключа.

Связано это с особенностью работы функции генерации случайного значения. Она использует системный вызов в Linux, соответственно, тот не сильно быстрый и обращается к встроенному генератору. Поэтому есть задержка, пока генератор выдаст значение. То есть упираемся в вызов функции при генерации значений. Казалось бы, там доли миллисекунды, но, когда умножается на 10 миллионов итераций, в целом, вот такой **overhead**[29].

[29] Overhead URL: https://en.wikipedia.org/wiki/Overhead_(business) (дата обращения 16.01.2024) [18]

Но это не самое страшное. Давайте создадим индексы по этим полям (тоже есть разница, хоть и небольшая) и проверим работу count():

```
CREATE INDEX ON records2 (id);
CREATE INDEX ON records3 (uuid_v4);
```

```
postgres=# CREATE INDEX ON records2 (id);
CREATE INDEX
Time: 4634.612 ms (00:04.635)
postgres=# CREATE INDEX ON records3 (uuid_v4);
CREATE INDEX
Time: 6845.019 ms (00:06.845)
```

Разница 50%. В среднем, от 50 до 100%!

Выполним count:

```
SELECT COUNT(id) FROM records2;
SELECT COUNT(uuid_v4) FROM records3;
```

```
postgres=# SELECT COUNT(id) FROM records2;
  count
----------
 10000000
(1 row)

Time: 472.055 ms
postgres=# SELECT COUNT(uuid_v4) FROM records3;
  count
----------
 10000000
(1 row)

Time: 434.455 ms
```

Практически не видно разницы, в рамках статистической погрешности!

В версии 15 разница была до двух раз не в пользу UUID. Это связано с **visibility map** данных для разных снимков – реализация MVCC[30]. То есть, находя строку в индексе, PostgreSQL идёт в VM и проверят, видно ли для данного снимка данных эту строку. Казалось бы, в любом случае он должен сходить 10 млн раз, но есть один нюанс – в первом случае данные генерируются последовательно и попадаем в ту же страницу VM в памяти, которая лежит закэшированная + есть оптимизация у PostgreSQL на такой случай. А в случае UUID – данные случайны (но, как видим, в версии 16 оптимизировали работу с UUID) и каждый раз надо искать в хеш-таблице нужную страницу, а при отсутствии подтягивать её с диска.

[30] MVCC URL: https://www.postgresql.org/docs/current/mvcc.html (дата обращения 16.01.2024) [19]

В конце статьи ещё посмотрим на размер данных и индексов:

```
CREATE EXTENSION pg_prewarm;
SELECT relname, pg_size_pretty(pg_relation_size(oid)), pg_prewarm(oid) FROM pg_class WHERE relname like 'records%';
```

```
postgres=# CREATE EXTENSION pg_prewarm;
CREATE EXTENSION
Time: 11.378 ms
postgres=# SELECT relname, pg_size_pretty(pg_relation_size(oid)), pg_prewarm(oid) FROM pg_class WHERE relname like 'records%';
       relname        | pg_size_pretty | pg_prewarm
----------------------+----------------+------------
 records2             | 1347 MB        |     172414
 records2_id_idx      | 214 MB         |      27422
 records3             | 1420 MB        |     181819
 records3_uuid_v4_idx | 301 MB         |      38506
(4 rows)

Time: 1961.986 ms (00:01.962)
```

Видим, что хоть индекс для UUID имеет больший размер, но на скорость работы это практически не влияет.

Проблему с более последовательными значениями можно решить, используя генерацию UUID_v7[31] – но пока патч не вошёл в основную версию PostgreSQL. Видимо, как раз из-за устранения такой проблемы обращения к visibility map.

Есть, конечно, вариант получить количество строк быстрее – достать значение из статистики по таблицам, но значение может быть неточным:

```
SELECT count(*) FROM book.tickets;
SELECT reltuples::BIGINT AS estimate
FROM pg_class
WHERE oid = 'book.tickets'::regclass;
```

```
thai=# \timing
Timing is on.
thai=# SELECT count(*) FROM book.tickets;
  count
---------
 5185505
(1 row)

Time: 505.007 ms
thai=# SELECT reltuples::BIGINT AS estimate
thai-# FROM pg_class
thai-# WHERE oid = 'book.tickets'::regclass;
 estimate
----------
  5185576
(1 row)

Time: 2.866 ms
```

Разница по времени выполнения и полученным значениям очевидна: ~1962 мс и ~3 мс.

[31] UUID v7 URL: https://pgxn.org/dist/pg_uuidv7/ (дата обращения 16.01.2024)[20]

Так что даже такой простой вопрос: подсчёт количества записей – имеет кучу нюансов и подводных камней, что уж говорить о более серьёзных вещах.

Переходим к основной теме главы – оптимальный выбор и настройка платформы.

Построение инфраструктуры под данные в PostgreSQL начинается с самого важного и сложного шага – выбора аппаратной архитектуры. Идеально разворачивать на железе, но в текущих реалиях такие ситуации встречаются редко. Обычно никто на них не выделяет финансов/ресурсов. Или, если выделяют, организовывают на них виртуальную инфраструктуру на базе **Kubernetes**, **Proxmos**[32], **Hyper-V**[33], **ESXi**[34] и аналогах.

На что стоит обратить внимание при выборе железа под PostgreSQL или системы виртуализации/контейнеризации.

Во-первых, это разные поколения процессоров. Несмотря на то, что выделяться с высокой долей вероятности будут ВМ, они всё равно реализованы на железе. Они могут дать совершенно разный результат даже просто при смене поколений. То есть нужно иметь в виду, что может быть ситуация, когда переехали на более мощное железо, а PostgreSQL не ускорился, поэтому перед переездом железа – желательно даже перед заказом данного железа – в идеале, протестировать его тем или иным способом.

Во-вторых, все почему-то смотрят только на **IOPS**[35] – количество операций ввода-вывода на системы хранения, чем больше, тем считается лучше, но никто не обращает внимания на **Latency**[36] – задержку, с которой эта информация будет считана или записана.

Пример: 30 тысяч IOPS – значит 30 тысяч блоков по 4 КБ[37] (стандарт, но могут быть и большие блоки). А с какой задержкой они будут считаны/записаны?

Нужно понимать, что совершенно по-разному ведут себя жёсткие диски: NVMe, network-SSD, технологии Ceph, S3 и так далее.

Вернёмся к поведению хранилищ на нашем примере – необходимо учитывать ещё и профиль нагрузки. Начинаем копировать и получаем сотни мегабайт в секунду. Начинаем писать в PostgreSQL и цифры будут значительно

---

[32] Proxmos URL: https://www.proxmox.com/en/ (дата обращения 16.01.2024) [21]
[33] Hyper-V URL: https://en.wikipedia.org/wiki/Hyper-V (дата обращения 16.01.2024) [22]
[34] ESXi URL: https://www.vmware.com/products/esxi-and-esx.html (дата обращения 16.01.2024) [23]
[35] IOPS URL: https://en.wikipedia.org/wiki/IOPS (дата обращения 16.01.2024) [24]
[36] latency URL: https://en.wikipedia.org/wiki/Latency_(engineering) (дата обращения 16.01.2024) [25]
[37] Килобайты

меньше. Опять же, получаем 30 тысяч блоков с задержкой в 10 мс[38] или 1 секунду – совершенно другое время отклика и, как следствие, жалобы пользователей более чем вероятны.

В-третьих, нужно учитывать геолокацию виртуальных машин. Допустим, 70% трафика сосредоточено в Центральной России, но остальные 30% на Дальнем Востоке, нам необходимо минимум одну из реплик держать на Дальнем Востоке хотя бы для чтения. Таким образом сокращаем время отклика на наши запросы с этого региона, снижаем сетевую нагрузку и нагрузку на основной сервер. Поэтому георасределение – очень важная часть проектирования архитектуры. Конечно же, должны быть настроены геобалансеры[39] или учтена эта особенность на уровне кода приложения.

В-четвёртых, необходимо внимательно относиться к используемым настройкам аппаратных ресурсов, в частности, **NUMA**[40] и **HyperThreading**[41] (HT).

Более глубоко подводные камни разбираются на моём курсе оптимизации PostgreSQL[42].

Следующим этапом идёт **выбор ОС** для PostgreSQL.

PostgreSQL имеет ряд проблем с переключением между процессами в Windows. Файловая система NTFS (по умолчанию в Windows) также имеет ряд проблем при работе с PostgreSQL. Таким образом Windows крайне не рекомендован к использованию в production-контуре, да и вообще, в целом, для PostgreSQL.

Остаётся только Linux во всём его многообразии. Разница довольно относительна между дистрибутивами, но здесь, скорее всего, будет зависеть от вашей ИБ или СБ[43].

Примеры в книге будут демонстрироваться на Ubuntu.

---

[38] Миллисекунд

[39] GSLB URL: https://www.nginx.com/resources/glossary/global-server-load-balancing/ (дата обращения 16.01.2024) [26]

[40] NUMA URL: https://en.wikipedia.org/wiki/Non-uniform_memory_access (дата обращения 16.01.2024) [27]

[41] HT URL: https://en.wikipedia.org/wiki/Hyper-threading (дата обращения 16.01.2024) [28]

[42] Курс Оптимизация Постгреса URL: https://aristov.tech/#course (дата обращения 16.01.2024) [29]

[43] информационная безопасность или Служба безопасности

Почему Ubuntu?

Практически самый хорошо поддерживаемый дистрибутив, имеющий регулярные мажорные[44] и минорные обновления, патчи безопасности. Linux, конечно, имеет свои баги в разных версиях. Куда без них? Поэтому тестирование на конкретном дистрибутиве перед внедрением в production просто обязательно!

Также важно проверять обновления перед использованием! Необходимо поддерживать актуальные версии ОС, при этом категорически не рекомендовано переходить сразу на вышедшие новые мажорные версии. Например, на момент написания этой главы последняя мажорная версия Ubuntu 22.04 – при выходе новой версии 24.04 в ней однозначно будут детские ошибки, крайне малоприятно их вылавливать на своих системах. Лучше дождаться второй-третьей минорной версии.

Важно заметить, что в текущих реалиях очень много production крутится в Docker и Kubernetes – они, конечно, дают от 3 до 10 (в среднем) процентов overhead. Но удобство использования инфраструктуры перевешивает минусы и в большинстве случаев это оправданно.

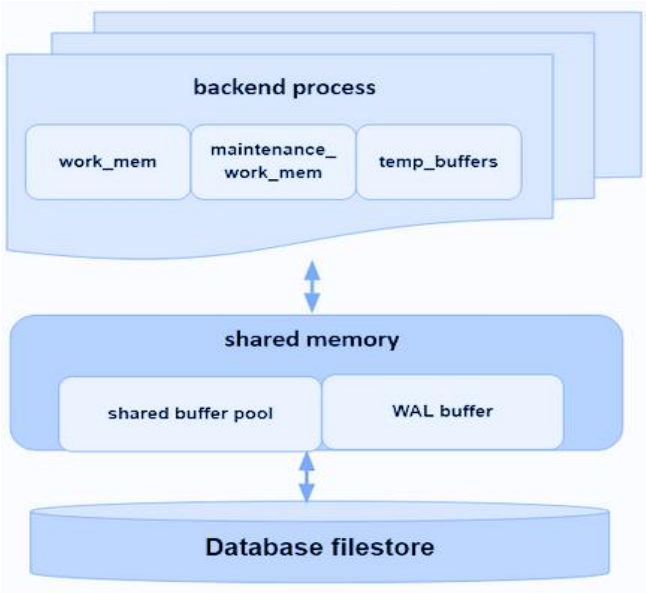
Опять же, могут быть нюансы по совместимости образов хостовой ОС и внутри контейнера. Был случай, когда контейнер с Ubuntu 16 запустили на версии 18 и из-за небольших особенностей ядра получили большую просадку производительности.

Очень важно, что использовать рекомендовано только официальные сборки, причём желательно из вашего локального репозитория[45] и перед загрузкой проверенные вашей СБ!

[44] Major minor URL: https://en.wikipedia.org/wiki/Software_versioning (дата обращения 16.01.2024) [30]
[45] Local registry URL: https://www.docker.com/blog/how-to-use-your-own-registry-2/ (дата обращения 16.01.2024) [31]

Переходим к **настройкам ОС**.

Для начала, вспомним, как PostgreSQL работает с памятью.



Выделяется **shared memory,** в которой хранится кэш данных с диска (более подробно рассмотрим в главе 6), далее выделим под каждую сессию **work_mem** (может выделяться несколько раз) **+ temp_buffers**. Если в сессии используются обслуживающие процессы, то умножаем их на **maintenance_work_mem**, ОС также потребляет ресурсы.

Клиенты подключаются, память тратится, начинает заканчиваться, приходит **OOM killer**[46] и начинает убивать клиентов Постгрес или сам родительский процесс PostgreSQL. Сейчас узнаем, как этого избежать.

ОС имеет такой параметр, как **swappiness**.

То есть, при достижении этого параметра, она передаёт на диск те данные, которые ей кажутся не сильно актуальными. Чем меньше памяти остаётся, тем активнее она начинает это делать. Нужно понимать, что доступ к этой информации на порядки медленнее, чем к оперативной памяти. Тут тоже есть небольшой нюанс. Везде в документации написано, что параметр настраивается от 0 до 100. На самом деле, параметр от 0 до 200[47].

По умолчанию, настроен на 60, то есть остаётся 60% памяти и ОС начинает свопить (swap) – скидывать на диск, как кажется ОС (она не понимает сущность скидываемых данных), ненужную в данный момент информацию,

[46] OOM killer URL: https://www.kernel.org/doc/gorman/html/understand/understand016.html (дата обращения 16.01.2024) [32]

[47] swappiness URL: https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html?highlight=swappiness (дата обращения 16.01.2024) [33]

освобождая место для актуальной, что недопустимо для production-ready-систем.

Есть три основные стратегии, как настроить **swappiness**.

Первая – вообще **отключить swappiness** – в итоге, ускоряемся, так как не используем дисковую память. Но придётся ждать, скорее всего, OOM killer.

Довольно опасная конфигурация, рекомендую применять, только если чётко понимаете, сколько доступно ресурсов и сколько будет коннектов. И если в целом они в сумме не превышают установленный лимит памяти, то можно. Конечно, **обязательно** надо настроить мониторинг. А если начинает использоваться больше 80-90% памяти, то могут быть нюансы с OOM killer.

История из реального кейса.

*Память ВМ 16 ГБ, **max_connections** = **1000, shared_buffers** = 20 ГБ, **work_mem** = 32 МБ, **maintenance_work_mem** = 1024 МБ.*

Что не так с этим конфигом? Почему приходил OOM killer?

Давайте считать: max_connections умножаем на work_mem – без учёта shared_buffers и без учёта maintenance_work_mem уже превышаем 16 ГБ.

Даже один shared_buffers больше всей оперативной памяти ВМ. Shared_buffers начинает заканчиваться, так как PostgreSQL не знает, сколько всего памяти и как настроена ОС. Указали, что 20 ГБ можно тратить – он 20 ГБ и тратит. Это не его проблема, что у ОС памяти нет.

Или запустили reindex сразу, допустим, пять экземпляров в параллели.

Вывод – очень аккуратно выделяем память.

То есть, если бы был swap, просто ушли бы в swap. В данном случае приходит OOM killer. У ОС какие варианты? Или она умирает совсем, или просто убивается процесс, освобождая память.

Привожу пример, как настроить параметры в данном варианте.

Общая формула:

```
ОЗУ[48] > 2Gb + shared_buffers + max_connections * (work_mem
+ temp_buffers) + maintance_parallel_workers *
maintance_work_mem
```

[48] ОЗУ URL: https://en.wikipedia.org/wiki/Computer_data_storage#Primary_storage (дата обращения 16.01.2024) [34]

С учётом того, что work_mem может выделяться несколько раз.

Правда, не весь объём сразу резервируется, но есть шанс OOM killer.

Кого убьёт OOM killer – вопрос интересный, но с этим можно побороться[49], чтобы он убивал хотя бы бэкэнд-процессы, а не основной PostgreSQL, потому что если основной он убил, то всё.

Вторая классическая стратегия.

Используем swap, но устанавливаем **swappiness в 1**. Допустим, есть 100 ГБ памяти, остаётся 1 ГБ и начинает использоваться swap. Но тут проблема, что при резком всплеске можем не успеть. Допустим, одновременно пришло 100 коннектов, всем нужна память… Хотя, в целом, ОС обычно сможет всё-таки обработать. Поэтому параметр «1» – хорошее значение для swappiness, то есть swap практически не используем. Опять же, здесь проблематика, что ОС не знает об особенностях Постгрес, не знает конкретно, что вытеснять – память анонимная и другие факторы. Не факт, что будут вытеснены ненужные вещи.

Третья оптимальная стратегия, которая мне нравится и которую я и использую – это **swappiness 2-5**. Могут быть такие небольшие тормоза, когда остаётся мало свободной памяти, но, с другой стороны, неплохо защищены от OOM killer.

Кроме swappiness, также есть и ряд других параметров, например, группа настроек **vm.dirty_*ratio**, отвечающие за порядок записи на диск.

Данные, которые лежат в дисковом кэше ОС, должны быть записаны на диск. Не факт, что они будут сразу записаны. То есть, они всё-таки какое-то время в каком-то объёме лежат в памяти и по **fsync** должны попадать на диск. По умолчанию стандартный размер 5%. Но возьмём пример и, допустим, есть 256 ГБ ОЗУ, тогда 5% – это 12 ГБ лежит в памяти и при большой нагрузке будет принудительно вызван fsync и их нужно единомоментно записать на диск. NVMe даже по последней спецификации[50] 6 ГБ в секунду пишут. То есть в данном случае нужно 2 секунды для записи 12 ГБ. Поэтому здесь тоже есть ряд нюансов, и если есть единомоментные всплески нагрузки на диск, то можно исследовать параметры в этом направлении.

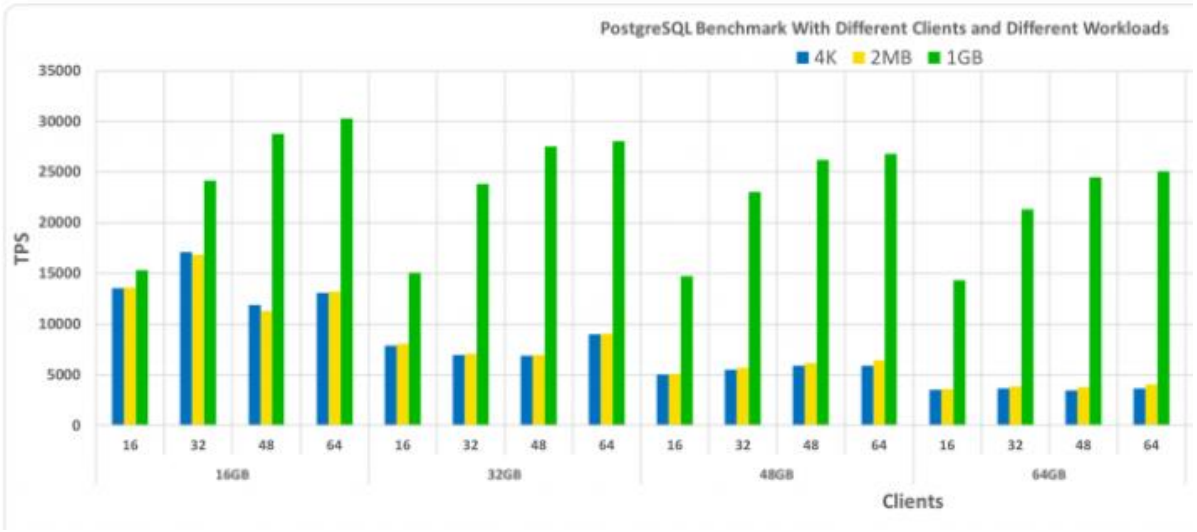
Чем больше размер общей памяти, тем больше хеш-таблица для поиска нужных данных и могут быть побочные эффекты в виде просадки

[49] Tuning for OOM URL: https://www.postgresql.org/docs/current/kernel-resources.html#LINUX-MEMORY-OVERCOMMIT (дата обращения 16.01.2024) [35]
[50] NVMe pci-e 5.0 URL: https://ks-is.com/pcie-3-0-protiv-pcie-4-0-protiv-pcie-5-0-v-chem-raznica (дата обращения 16.01.2024) [36]

производительности. Для борьбы с таким эффектом были придуманы **huge pages**[51]. Допустим, нужен мегабайт данных с диска подряд. Зачем ходить 125 раз на диск по 8 килобайт (стандартная страница), если можно сразу в память подгрузить несколько мегабайт и там уже ими оперировать. Это как раз и есть huge pages или большие страницы. Стандартно выделяется размер от 2 мегабайт до гигабайта и общее количество таких страниц. То есть можно оперировать не маленькими блоками по 8 килобайт, а большими страницами при массивной обработке данных.

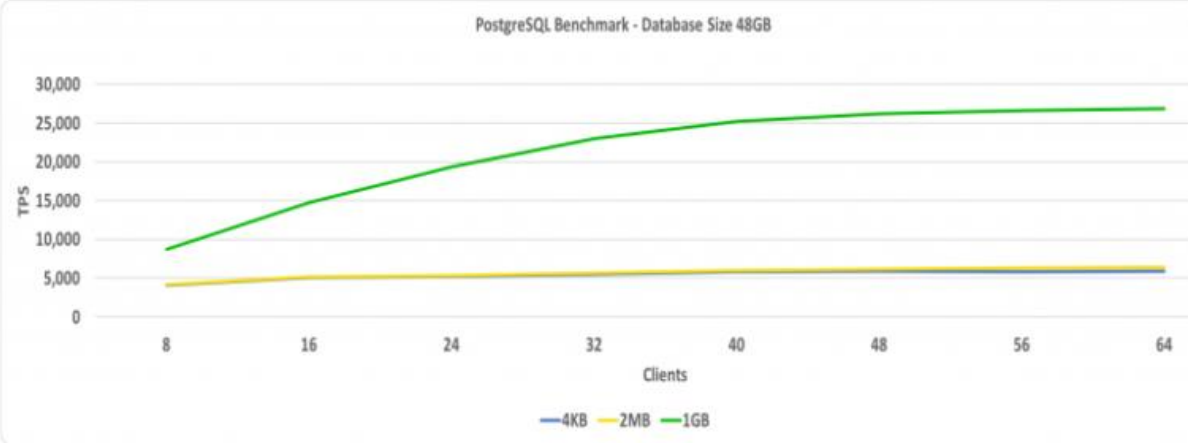
Нужно знать, с какими объёмами будете работать, и в зависимости от них оценить необходимость huge pages, размеры баз данных, размеры клиентских транзакций и другие параметры, которые могут на это повлиять.

На исследовании уважаемой компании Percona[52] увидим разницу:

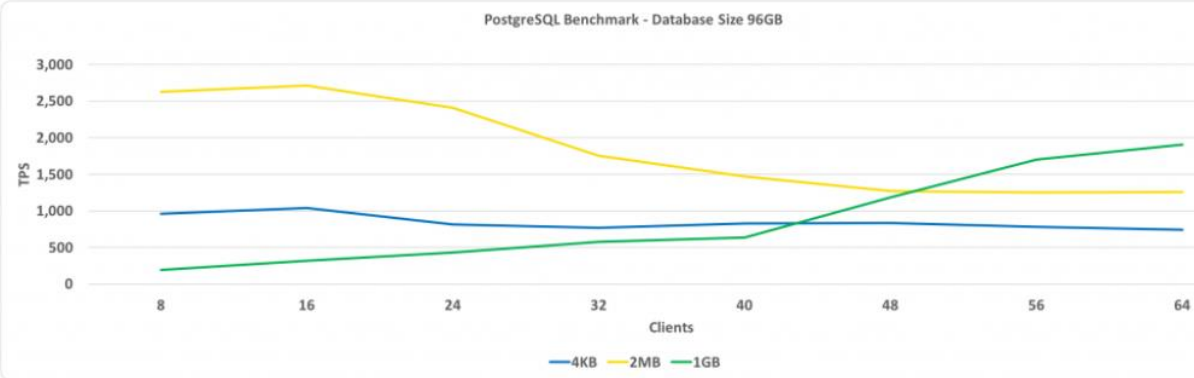


Здесь видим корреляцию между использованием разного размера huge pages, общего размера БД, количества клиентов и скорости работы СУБД.

[51] Huge Pages URL: https://linuxconfig.org/how-to-enable-hugepages-on-linux (дата обращения 16.01.2024) [37]
[52] Huge pages benchmark URL: https://www.percona.com/blog/benchmark-postgresql-with-linux-hugepges/ (дата обращения 16.01.2024) [38]



Особенно этот эффект заметен в исследовании при размере большой страницы 1 ГБ. Опять же, нужно сделать ремарку – это на конкретном железе для конкретного профиля нагрузки.



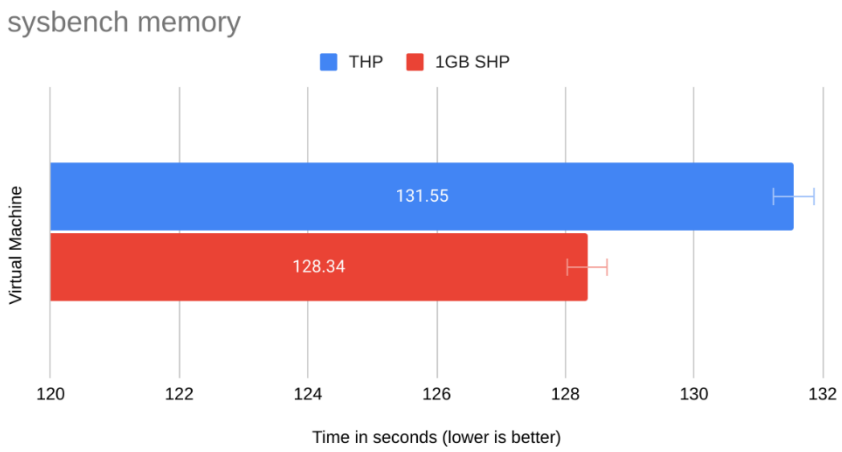
А вот здесь видим уже результат интереснее – размер БД в 96 ГБ превышает выделенный shared_buffers в 64 ГБ. Так что, несмотря на однозначную полезность технологии, многое зависит от профиля нагрузки и других факторов.

Дальше в книге посмотрим, что от профиля нагрузки зависит очень многое.

Есть ряд рекомендованных настроек, которые нужно делать по умолчанию. Есть те, с которыми можно поэкспериментировать на тестовом кластере. И логичное продолжение этой технологии – так называемый **Transparent Huge Pages**[53], чтобы приложение вообще не знало, что оно работает с huge pages. Казалось бы, отличное решение. Операционная система пусть сама разруливает все эти моменты, но, как оказалось, лучшее – враг хорошего.

[53] Transparent HP URL: https://docs.kernel.org/next/admin-guide/mm/transhuge.html (дата обращения 16.01.2024) [39]

Для анализа ситуации посмотрим на исследование компании RedHat[54]:



Видим, что при включении Transparent Huge Pages результат выполнения запроса 131 секунда, а при отключении 128 секунд.

Аналогичные проблемы были и у компании Тинькофф. Пример отличного траблшутинга этих проблем можно почитать в статье на Хабре[55].

К сожалению, объём книги ограничен, но вот на какие параметры ещё стоит обратить внимание (они рассматриваются на курсе):

- **vm.overcommit_***
- **Kernel Same-page Merging**
- **Memory Compression**
- **Memory Ballooning**
- **CPU Affinity**
- **madvise**
- **kernel.sh***
- **net.***

Выбрали нужную архитектуру, подтюнили железо и ОС. Дальше нужно установить PostgreSQL и инициализировать кластер.

Обратите внимание! Неоднократно встречал, что используют русские кодировки при инициализации кластера, например, CP1251. Есть минимальная экономия на объёме хранения данных в символах 1 байт, вместо 2 байт для

[54] Memory benchmarking URL: https://developers.redhat.com/blog/2021/04/27/benchmarking-transparent-versus-1gib-static-huge-page-performance-in-linux-virtual-machines#benchmarks (дата обращения 16.01.2024) [40]
[55] THP problem URL: https://habr.com/ru/companies/tinkoff/articles/446342/ (дата обращения 16.01.2024) [41]

русских символов. Но при этом получаем несовместимость выгрузок и межпроцессного взаимодействия, так как с высокой долей вероятности они будут в UTF-8. Поэтому используем UTF-8 и забываем вообще об этой проблематике. Эта экономия того не стоит.

Ещё один очень интересный момент при инициализации PostgreSQL – нужно указать размер блока, по умолчанию 8 КБ. Как это изменять и на что это влияет, разберём в главе 3. Там есть разные интересные сайд-эффекты при изменении этого размера.

Небольшой чек-лист, на что ещё стоит обратить внимание при установке и инициализации кластера:

- в зависимости от ОС дефолтный Постгрес будет разных версий;
- кастомная сборка с патчами в ядро НЕ рекомендована из-за довольно проблематичного обновления версий СУБД;
- рекомендуется ставить из официального репозитория https://www.postgresql.org;
- по умолчанию на Debian-системах (в том числе Ubuntu) дефолтный кластер автоматически создаётся и запускается, также есть ряд удобных утилит pg_*;
- с 14 версии PostgreSQL дефолтная система шифрования пароля SCRAM-SHA-256, а не MD5, как ранее, не имеет прямой совместимости, аккуратнее при миграции юзеров;
- имеет смысл включить в template0 extension, чтобы не создавать их в каждой БД, также включить shared_preload_libraries = 'pg_stat_statements', правда можем получить небольшую просадку производительности до 15%, разберём в соответствующей главе.

После установки PostgreSQL нужно обязательно подтюнить. Всего параметров PostgreSQL в версии 16 – 361:

```
SELECT count(*) FROM pg_settings;
```

```
postgres=# SELECT count(*) FROM pg_settings;
 count
-------
   361
(1 row)
```

По факту порядка 20 основных[56], которые нужно тюнить вообще сразу же после установки. Давайте посмотрим, на что они влияют и какие значения рекомендованы для начальной настройки:

- **shared_buffers** – используется для кэширования данных.
  Задача: буфер между диском и клиентскими подключениями.
  Рекомендуемое значение для данного параметра – 25% от общей оперативной памяти на сервере. Редко значение больше, чем 40% окажет влияние на производительность.

- **max_connections** – максимальное количество соединений.
  Причины тюнинга: очень маленькое количество по умолчанию – 100.

  Рекомендуемые параметры: если планируется использование PostgreSQL, как DWH, то большое количество соединений не нужно. Данный параметр тесно связан с **work_mem**. Поэтому будьте с ним предельно аккуратны. Для изменения данного параметра придётся перезапускать сервер. Более подробно про коннектинг и проблемы обсудим в следующей главе.

- **effective_cache_size** – служит подсказкой для планировщика, сколько памяти у него в запасе. За счёт данного параметра планировщик может чаще использовать индексы, строить хэш-таблицы.
  При настройке необходимо учитывать другие параметры конфигурации: **shared_buffers**, **work_mem** и **maintenance_work_mem**. Все эти параметры влияют на распределение памяти и производительность PostgreSQL.

  Рекомендуемые параметры: наиболее часто используемое значение 50-75% оперативной памяти на сервере. Требует перезагрузки сервера. Также для оптимального выбора необходимо нагрузочное тестирование.

- **work_mem** – используется для хранения текущего датасета, сортировок, построения хеш-таблиц. Это позволяет выполнять данные операции в памяти, что гораздо быстрее обращения к диску.
  В рамках одного запроса данный параметр может быть использован несколько раз. Если запрос содержит 5 операций сортировки, то память, которая может использоваться для его выполнения, уже может сожрать как минимум work_mem * 5.

[56] main parameters URL: https://github.com/aeuge/postgres16book/blob/main/scripts/parameters.md (дата обращения 16.01.2024) [42]

Так как, скорее всего, на сервере много сессий, то каждая из них может использовать этот параметр по несколько раз, поэтому не рекомендуется делать его слишком большим. Можно выставить небольшое значение для глобального параметра в конфиге и потом, в случае сложных запросов, менять этот параметр локально (для текущей сессии).
Обратите внимание, что при превышении этого параметра будет использовано временное пространство, расположенное на диске – запросы будут выполняться медленнее и при большом запросе с декартовым произведением могут привести к опустошению пустого места на диске и завершаться с ошибкой, также могут способствовать приходу OOM killer в зависимости от конфигурации ОС.
Более подробно будем разбирать в главе 7.

- **maintenance_work_mem** – определяет максимальное количество выделяемой памяти для операций типа VACUUM, CREATE INDEX, CREATE FOREIGN KEY.
  Увеличение этого параметра позволит быстрее выполнять эти операции.
  Не связано с work_mem, поэтому можно ставить в разы больше, чем work_mem.
  Выделяется ТОЛЬКО при вызове обслуживающих операций.

  Рекомендуемые параметры: классический подход установить maintenance_work_mem в диапазоне от 128 МБ до 1 ГБ или даже больше, если у вас есть достаточно оперативной памяти на сервере и выполнение обслуживания занимает существенную долю времени.

- **wal_buffers** – объём разделяемой памяти, который будет использоваться для буферизации данных журнала транзакций (Write-Ahead Log – WAL[57]), ещё не записанных на диск.
  Если у вас большое количество одновременных подключений, увеличение параметра улучшит производительность.
  По умолчанию -1, определяется автоматически как 1/32 от «shared_buffers», но не больше, чем 16 МБ (вручную можно задавать большие значения).

  Рекомендуемые параметры: обычно ставят 16 МБ. Более подробно проблематику и тюнинг WAL будем рассматривать в главе 6.

[57] WAL URL: https://www.postgresql.org/docs/current/wal-intro.html (дата обращения 16.01.2024) [43]

- **min_wal_size / max_wal_size** – тюнинг этих параметров тесно связан с управлением WAL. Эти параметры позволяют настроить размеры журнальных сегментов, которые используются для записи изменений в базу данных перед их фиксацией.

- **min_wal_size** – этот параметр задаёт минимальный размер журнального сегмента, до которого должен «опуститься» WAL перед переиспользованием. Если установить его слишком низко, может возникнуть увеличение количества записей – соответственно, высокий ввод/вывод (I/O), так как PostgreSQL не сможет эффективно переиспользовать журнальные файлы. Рекомендуется установить его на достаточно высокое значение, чтобы уменьшить I/O операции записи, но не слишком высокое, чтобы избежать излишнего потребления места на диске.

- **max_wal_size** – устанавливает максимальный размер журнального сегмента. Если установить его слишком низко, это может привести к тому, что база данных перестанет работать, когда достигнет предела размера журнала, и потребуется архивация WAL для освобождения места. Но если значение установлено слишком высоко, это может привести к тому, что понадобится больше места на диске.

  Рекомендуется выбирать значения для min_wal_size и max_wal_size таким образом, чтобы обеспечить баланс между эффективностью записи и использованием дискового пространства, а также учитывать конкретные характеристики базы данных и потребности в производительности.

- **checkpoint_timeout** – время между сбросами закэшированной изменённой информации из shared_buffers на диск.
  Чем реже происходит сбрасывание грязных буферов на диск, тем дольше будет восстановление БД после сбоя.
  Значение по умолчанию 5 минут, рекомендуемое – от 10 минут до часа.

  Рекомендуемые параметры: необходимо ещё «синхронизировать» параметр **max_wal_size**. Для этого можно поставить **checkpoint_timeout** в выбранный промежуток, включить параметр **log_checkpoints** и по нему отследить, сколько было записано буферов. После чего подогнать параметр **max_wal_size**. Также здесь в тандеме настройки участвует и bg_writer. Более подробно разберём в соответствующей главе.

- **synchronous_commit** – 5 уровней синхронной записи WAL-журналов на диск и синхронного ответа от ведомого сервера в зависимости от выбранного уровня (более подробно будем рассматривать в главе 4).
  Отключаем синхронную запись журнала изменений данных на диск, что позволяет увеличить скорость ответа СУБД от 10% до 3000+% за счёт нивелирования времени подтверждения такой записи каждой транзакции. Конечно, при сбое ВМ можно потерять небольшую часть последних изменений.

  Рекомендуемые параметры: зависит от бизнес-логики и других параметров и задач.

- **random_page_cost** – задаёт приблизительную стоимость чтения одной произвольной страницы с диска. Значение по умолчанию равно 4.0.

  Рекомендуемые параметры: у твердотельных накопителей лучше выбрать меньшее значение random_page_cost – оптимально 1.0 или 1.1.

- **effective_io_concurrency** – задаёт оценку, сколько параллельных асинхронных запросов может выдержать дисковая подсистема. Современные твердотельные накопители эффективно справляются с этой задачей.

  Рекомендуемые параметры: можно ставить 100-300. Правда, если и ОС поддерживает posix_fadvise[58].

- **old_snapshot_threshold** – в теории должен БЫЛ решить проблему со старыми снапшотами[59] при долгих транзакциях, но всё как обычно: хотели как лучше – получилось как всегда[60].
  Еще один отличный кейс – 100+ ядер и долгие транзакции – до 5 раз падение производительности[61].
  Также будет аффектить при AUTOCOMMIT = OFF (увидим в главе 10).
  Проблему пока так никто и не пофиксил, но обещают в версии 17.

---

[58] posix_fadvice URL: https://www.opennet.ru/man.shtml?topic=posix_fadvise&category=2&russian=0 (дата обращения 16.01.2024) [44]

[59] snapshot URL: https://www.percona.com/blog/working-with-snapshots-in-postgresql/ (дата обращения 16.01.2024) [45]

[60] old snapshot problem URL: https://www.postgresql.org/message-id/20230213204507.b7k3fiorgwrahsjx%40awork3.anarazel.de (дата обращения 16.01.2024) [46]

[61] old snapshot problem2 URL: https://github.com/postgres/postgres/blob/REL_13_STABLE/src/backend/utils/time/snapmgr.c#L1808C33-L1808C33 (дата обращения 16.01.2024) [47]

Рекомендуемые параметры: -1 – однозначно отключить!

- max_worker_processes – этот параметр определяет максимальное количество дополнительных рабочих процессов, которые могут быть созданы для выполнения параллельных операций. Включает как фоновые процессы, так и процессы для параллельных выполнений запросов. Значение по умолчанию – 8.

  Рекомендуемые параметры: необходимо настраивать в соответствии с количеством доступных процессорных ядер на сервере.
  Обычно устанавливают значение, равное количеству ядер или немного меньше. Например, на сервере с 16 ядрами можно установить max_worker_processes равным 12.

- max_parallel_workers_per_gather – этот параметр определяет максимальное количество рабочих процессов, которые могут быть использованы для параллельного сбора данных при выполнении операции сортировки или объединения. Значение по умолчанию – 2.

  Рекомендуемые параметры: необходимо настраивать значение на основе доступного количества памяти на сервере и типа запросов, которые выполняются. Если сервер имеет достаточное количество памяти и выполняет запросы с большим объёмом данных, можно увеличить значение параметра. Например, установить max_parallel_workers_per_gather равным 4 или 8.

- max_parallel_workers – этот параметр определяет максимальное общее количество рабочих процессов, которые могут использоваться для всех параллельных операций. Значение по умолчанию – 8.

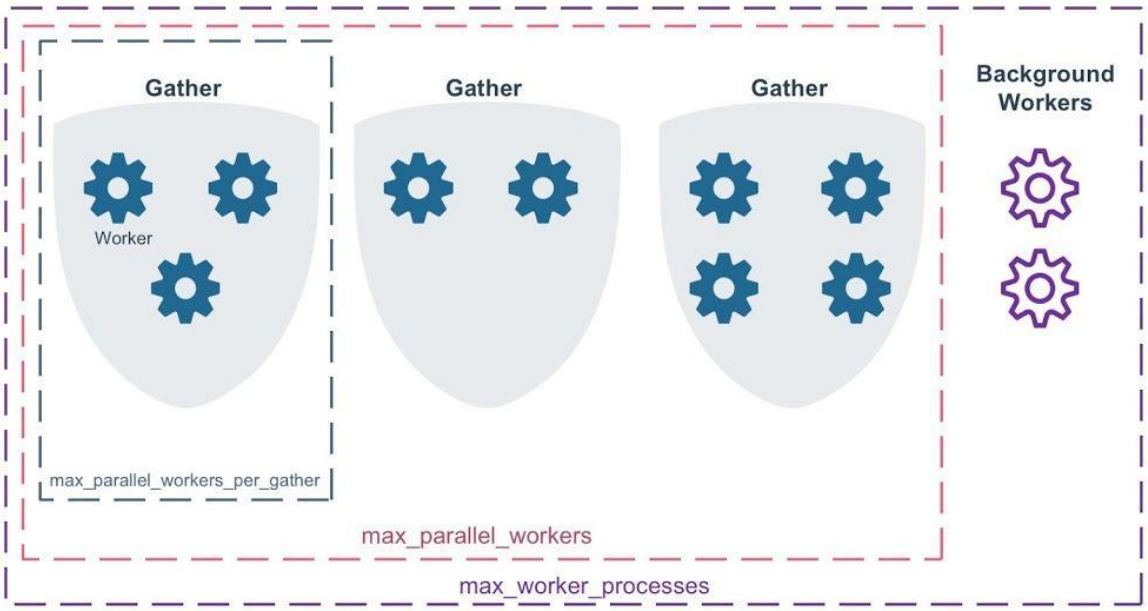
  Рекомендуемые параметры: при настройке необходимо обеспечить баланс между параллельным выполнением запросов и другими задачами на сервере. Если сервер имеет много ядер и большой объём оперативной памяти, можно увеличить значение параметра, например, до 16 или 32.

- max_parallel_maintenance_workers – этот параметр определяет максимальное количество рабочих процессов, которые могут быть использованы для параллельных операций обслуживания, таких как VACUUM и INDEX BUILD. Значение по умолчанию – 2.

Рекомендуемые параметры: необходимо настраивать значение на основе типа обслуживания, которое выполняется на сервере. Если выполняется много обслуживания, можно увеличить значение параметра для ускорения этих операций. Например, установить **max_parallel_maintenance_workers** равным 4.

При настройке параметров, связанных с параллельным выполнением, важно учитывать конкретные характеристики сервера, объём данных и типы запросов, которые будут выполняться. Рекомендуется проводить тестирование и мониторинг производительности после изменения этих параметров, чтобы определить оптимальные значения.

Посмотрим на примере исследования компании DataEgret[62], как распределяются ядра:

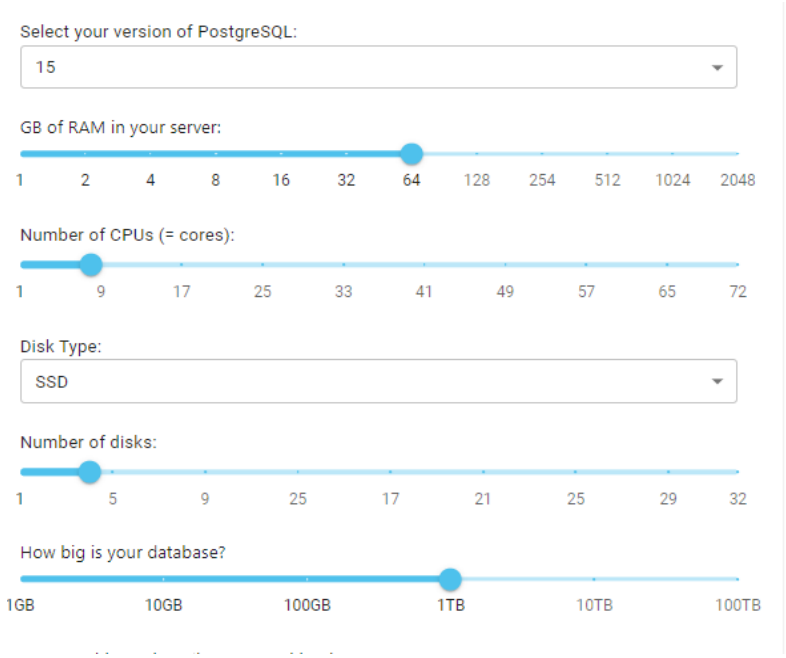


На схеме наглядно видно, какие параметры за что отвечают.

И, всё же, есть ли вариант оптимальной первоначальной настройки для автоматизации процесса?

Ответ – есть и это продвинутый конфигуратор от Cybertec[63], где указываются физические параметры сервера и тип нагрузки:

[62] Исследование распределения ядер URL: https://dataegret.com/2018/04/lets-speed-things-up/ (дата обращения 16.01.2024) [48]
[63] cybertec конфигуратор URL: https://pgconfigurator.cybertec.at/ (дата обращения 16.01.2024) [49]



После этого генерируется список настроек[64], которые достаточно записать в конец конфигурационного файла postgresql.conf и рестартовать инстанс PostgreSQL.

Также есть списки рекомендаций по тюнингу параметров:

- от разработчиков PostgreSQL[65]
- от контрибьютора PostgreSQL компании CrunchyData[66]
- скрипт автоматического checkup от Jfcoz[67]

Переходим к следующей проблематике.

Как измерять? Что измерять? И вообще, какие есть критерии для измерения?

Бенчмарков много разных и все они отличаются. Они могут как открывать транзакцию на каждый запрос, на сессию, так и работать в одной транзакции. Есть нюансы по запуску – сетевые задержки, когда запускаем с этой же ВМ или с сетевой, нюансы по архитектуре, например, pool connector[68] PgBouncer[69] имеет проблему однопоточности (разберём в главе 2). Опять же,

---

[64] default settings URL: https://github.com/aeuge/Postgres16book/blob/main/scripts/01/settings.txt (дата обращения 16.01.2024) [50]

[65] Tuning PostgreSQL URL: https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server (дата обращения 16.01.2024) [51]

[66] Tuning PostgreSQL URL: https://www.crunchydata.com/blog/optimize-postgresql-server-performance (дата обращения 16.01.2024) [52]

[67] Tuner URL: https://github.com/jfcoz/postgresqltuner (дата обращения 16.01.2024) [53]

[68] Pool connector URL: https://habr.com/ru/articles/194142/ (дата обращения 16.01.2024) [54]

[69] Pgbouncer URL: https://www.pgbouncer.org/ (дата обращения 16.01.2024) [55]

можем использовать классический встроенный бенчмарк **pgbench**[70] на миллионе записей или на миллиарде записей. Будет совершенно разный результат.

Каждый бенчмарк имеет свои особенности и характеристики, можно упорядочить требования к бенчмарку и изложить их в следующей редакции:

- открытие транзакции на каждый запрос
- открытие сессии на каждый запрос
- сетевые задержки

Огромное значение имеют:

- объём данных
- характеристики инстанса
- особенности файловых систем
- прогрев данных

Очень важен профиль нагрузки:

- 100% чтение
- или 100% запись – insert и update – абсолютно разная нагрузка
- есть ли индексы при вставке
- классика 90% чтение 9% insert и 1% update

Даже если поменяли тип инстанса BM и получили иной тип процессора – результат может быть непредсказуем. Даже от типа файловой системы много зависит (разберём подробно в главе 3).

В чём измерять производительность? TPS или QPS[71]? Опять же, в чём разница? TPS – transaction per second, QPS – query per second. Транзакция включает один и более запросов. То есть, TPS всегда будет меньше или равен QPS.

Классические варианты бенчмарков для PostgreSQL:

- pgbench
- Яндекс.Танк[72]
- Apache Benchmark[73]
- JMeter[74] для Java

---

[70] Pgbench URL: https://www.postgresql.org/docs/current/pgbench.html (дата обращения 16.01.2024) [56]
[71] TPS & QPS URL: https://www.osp.ru/os/2017/02/13052225 (дата обращения 16.01.2024) [57]
[72] Яндекс танк URL: https://cloud.yandex.ru/ru/marketplace/products/yc/load-testing (дата обращения 16.01.2024) [58]
[73] Ab URL: https://httpd.apache.org/docs/2.4/programs/ab.html (дата обращения 16.01.2024) [59]
[74] JMeter URL: https://jmeter.apache.org/ (дата обращения 16.01.2024) [60]

- sysbench[75]
- netperf[76]

Классические утилиты анализа производительности:

- top[77]
- atop[78]
- htop[79]
- perf-top[80]

Что измерять?

- TPS
- QPS
- Latency
- нагрузку на диск/CPU/сеть

Допустим, есть минимальный Latency, но CPU под 100%, или Latency в два раза больше, а CPU – в 10. Что лучше? Всё относительно.

Понятно, что для клиента лучше, наверное, Latency поменьше, но нам – когда CPU-нагрузка поменьше. Тут тоже вопрос, что лучше, решать бизнес-задачу или нагрузка на систему.

Рекомендация выбрать одну стратегию и метод тестирования и уже сравнивать результаты по одной методике.

Очень важно, чтобы объём БД был соизмерим с реальной продуктовой базой, так как на разных объёмах поведение СУБД, скорее всего, будет отличаться и не всегда можно предсказать, что произойдёт при кратной разнице в размерах данных.

При сдаче системы в эксплуатацию необходимо делать контрольные замеры, прописать методику и закрепить это официальным отчётом.

Вдруг отказ в обслуживании (denial-of-service) случился или другая проблема. Что к этому привело? Берём документацию и смотрим, что, например, при сдаче системы нагрузка на CPU была от 50 до 60%, она держала

[75] Sysbench URL: https://en.wikipedia.org/wiki/Sysbench (дата обращения 16.01.2024) [61]
[76] Netperf URL: https://hewlettpackard.github.io/netperf/ (дата обращения 16.01.2024) [62]
[77] Top URL: https://manpages.ubuntu.com/manpages/xenial/man1/top.1.html (дата обращения 16.01.2024) [63]
[78] Atop URL: https://linux.die.net/man/1/atop (дата обращения 16.01.2024) [64]
[79] Htop URL: https://htop.dev/ (дата обращения 16.01.2024) [65]
[80] Perf URL: https://man7.org/linux/man-pages/man1/perf-top.1.html (дата обращения 16.01.2024) [66]

100 тысяч RPS, а прошло полгода и всё поломалось. Оказалось, нагрузка выросла до 200 тысяч RPS и железо не справилось.

Поэтому рекомендую сохранить параметры, с которыми запустили, чтобы, когда, как вариант, через год случится деградация, могли, с чем сравнить.

Возникает вопрос – в каком виде хранить отчёт?

Классически. Просто снимаете бенчмарк, сохраняете его и записываете конфигурацию ПО и ВМ, размеры базы и так далее. Можно сделать скриншот **grafana**[81]. Также рекомендую иметь какой-то эталон.

Самый классический подход – берём ноутбук, например, у меня i7, там NVMe и запускаем бенчмарк на нём, смотрим – допустим, 10 тысяч RPS. Потом на ВМ запустили, а там 7 тысяч. Как так? ВМ хуже ноутбука? Значит, есть какие-то проблемы с железом или конфигом.

Посчитали в одинаковых условиях: на одном, на втором, на третьем – и пришли уже к какому-то выводу.

Посмотрим на практике.

Напоминаю, что есть ~6 млн записей в тайских перевозках.

Первым делом сгенерируем приблизительно хороший конфиг, потому что PostgreSQL не настроен по умолчанию. Для этого берём конфигуратор (чуть выше прикладывал ссылку) и после добавления строк в postgresql.conf перезагружаем инстанс PostgreSQL (не ВМ) через pg_ctlcluster.

Не забываем подтюнить Linux по указанным ранее параметрам.

Посмотрим, например, на **swappiness** и сконфигурируем параметр. Есть несколько вариантов, как посмотреть этот параметр, рассмотрим два:

```
cat /proc/sys/vm/swappiness
sysctl vm.swappiness
```

```
aeugene@postgres4:~$ cat /proc/sys/vm/swappiness
60
aeugene@postgres4:~$ sysctl vm.swappiness
vm.swappiness = 60
```

Для изменения онлайн достаточно под правами суперадминистратора просто изменить значение:

```
sudo sysctl vm.swappiness=1
```

[81] Grafana URL: https://grafana.com/ (дата обращения 16.01.2024) [67]

Но оно будет действовать только до перезагрузки. Для продолжительной фиксации необходимо внести изменения в файл:

```
sudo nano /etc/sysctl.conf
vm.swappiness=1
```

Посмотрим, что с большими страницами в системе:

```
grep Huge /proc/meminfo
```

```
aeugene@postgres4:~$ grep Huge /proc/meminfo
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
FileHugePages:         0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
```

Видим, что по 2 МБ HugePages по умолчанию. Можно поэкспериментировать с этим параметром и сравнить производительность для конкретной БД. Но в ОС отключены – HugePages_Total=0.

Обратите внимание, что в PostgreSQL также нужно включить использование больших страниц[82]. По умолчанию стоит параметр try. То есть он смотрит, если есть HugePages в операционке – он их включает. Если нет – не включает. В нашем случае они отключены.

```
SELECT name, setting, short_desc FROM pg_settings WHERE name like '%page%';
```

name	setting	short_desc
bgwriter_lru_maxpages	100	Background writer maximum number of LRU pages to flush per round.
full_page_writes	on	Writes full pages to WAL when first modified after a checkpoint.
huge_page_size	0	The size of huge page that should be requested.
huge_pages	try	Use of huge pages on Linux or Windows.

Для бенчмарков будем использовать классический pgbench, но с кастомными настройками. Протестируем кластер на дефолтном конфиге и начнём с генерации тестовой базы на 100 тысяч строк:

```
pgbench -i postgres
```

[82] Huge pages in postgresql URL: https://www.postgresql.org/docs/current/kernel-resources.html#LINUX-HUGE-PAGES (дата обращения 16.01.2024) [68]

```
postgres@postgres4:~$ pgbench -i postgres
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.06 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.30 s (drop tables 0.00 s, create tables 0.03 s, client-side generate 0.11 s,
```

Запустим тест в одного клиента на 10 секунд:

```
progress: 1.0 s, 662.0 tps, lat 1.503 ms stddev 0.210, 0 failed
progress: 2.0 s, 678.0 tps, lat 1.474 ms stddev 0.198, 0 failed
progress: 3.0 s, 622.0 tps, lat 1.607 ms stddev 0.222, 0 failed
progress: 4.0 s, 607.0 tps, lat 1.647 ms stddev 0.227, 0 failed
progress: 5.0 s, 670.0 tps, lat 1.492 ms stddev 0.215, 0 failed
progress: 6.0 s, 711.0 tps, lat 1.405 ms stddev 0.188, 0 failed
progress: 7.0 s, 693.0 tps, lat 1.442 ms stddev 0.191, 0 failed
```

Видим довольно посредственную производительность, даже до 1000 TPS не дошли.

Запустим параллельно 10 клиентов:

```
pgbench -P 1 -c 10 -T 10 postgres
```

```
latency average = 10.832 ms
latency stddev = 7.457 ms
initial connection time = 36.640 ms
tps = 922.411780 (without initial connection time)
```

Уже лучше.

А если запустить ещё в 4 потока по числу ядер по 10 клиентов?

```
pgbench -P 1 -c 10 -j 4 -T 10 postgres
```

```
latency average = 10.178 ms
latency stddev = 7.177 ms
initial connection time = 15.049 ms
tps = 981.726643 (without initial connection time)
```

Давайте проанализируем, что влияет на производительность. Во-первых, эти тестовые данные очень маленькие – 100 тысяч записей. Конечно, можем, например, отключить синхронный коммит и получить кратный рост производительности, пожертвовав надёжностью (последствия отключения синхронного коммита будем рассматривать в соответствующей главе). Во-вторых, под капотом теста довольно странный профиль TPC-B.

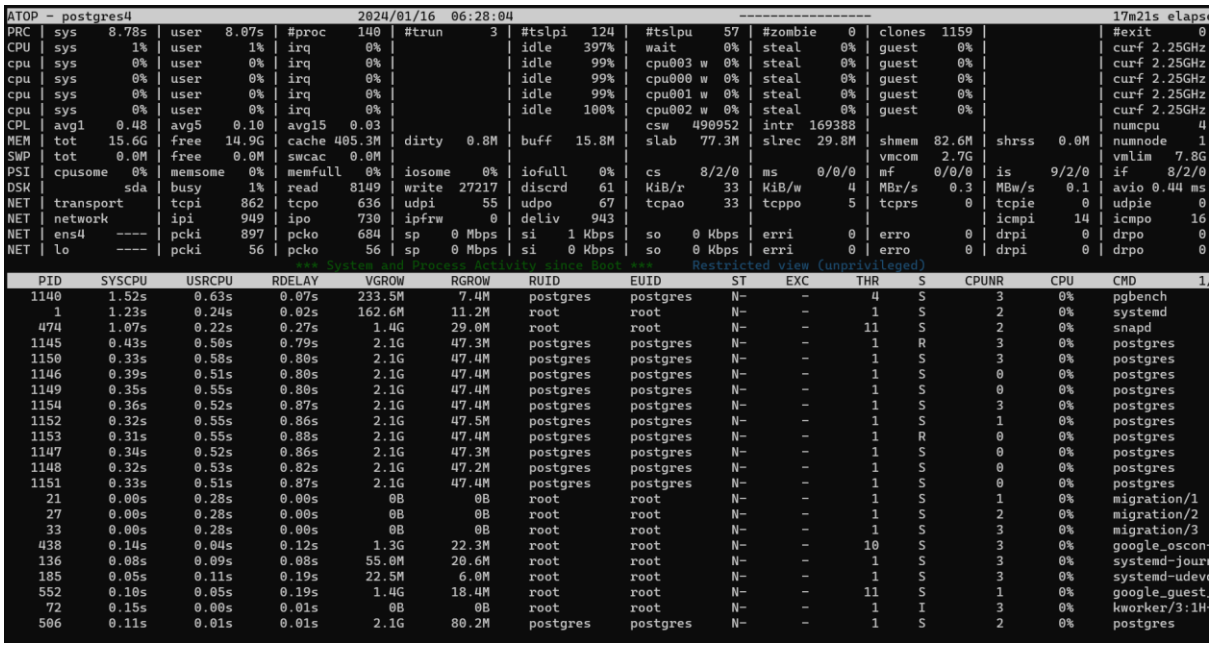
Структура одной транзакции выглядит следующим образом:

```
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

То есть попытка эмуляции изменения баланса. Внутри одной транзакции 5 запросов и из них 4 являются запросами на изменение/добавление данных. То есть 80% пишущая нагрузка (PostgreSQL использует модель MVCC copy-on-write[83]). Данный тип нагрузки встречается очень и очень редко. Обычный профиль нагрузки – 70-90% чтение, остальное update, insert или delete.

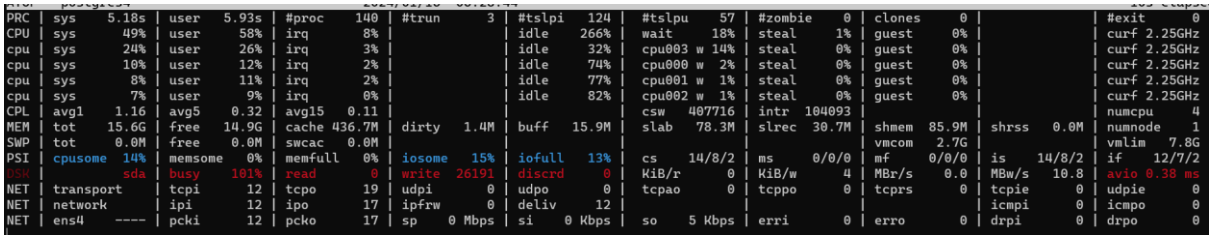
Поэтому в книге я буду использовать свои кастомные скрипты для бенчмарков, которые больше отражают реальные проекты.

Кроме скорости выполнения запросов, очень интересно посмотреть на нагрузку ВМ. Для этого воспользуемся классическими утилитами atop и htop. После нагрузки ВМ посмотрим на информативность утилит:



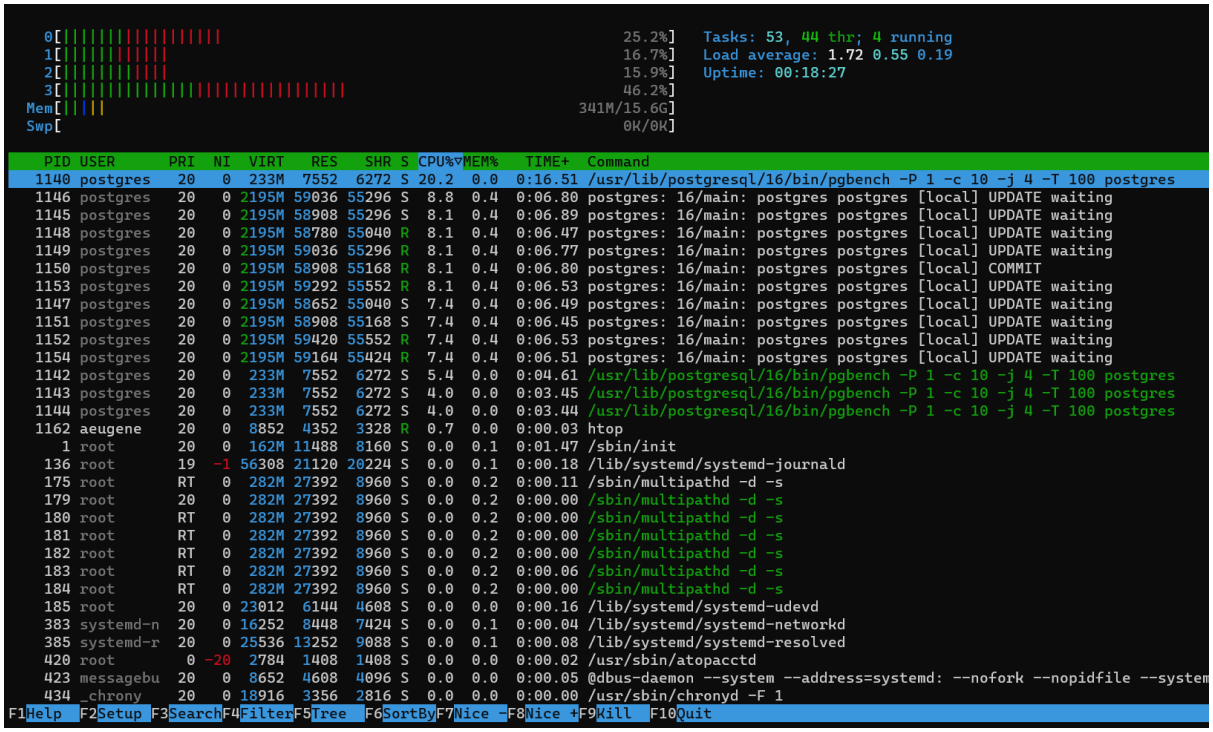
[83] MVCC URL: https://www.postgresql.org/docs/current/mvcc.html (дата обращения 16.01.2024) [69]

Утилита **atop** выводит огромное количество параметров системы, вплоть до прерываний и времени ядра на паразитное переключение процессов. Также важно обратить внимание, что утилита некоторое время собирает данные и потом подсвечивает проблемные места:



В нашем случае упёрлись в SSD-диск – четвёртая снизу (красная) строка.

Утилита **htop** попроще – но более информативно показывает нагрузку на процессор, память и выводит список самых тяжёлых процессов:



В целом, UPDATE и waiting UPDATE идёт – тест не оптимален и, в основном, нагрузка на диск, а процессор простаивает.

Используем более реальную нагрузку для бенчмарка.

Создадим 2 профиля нагрузки к БД тайских перевозок.

Первый – выборка случайной записи на базе в 6 миллионов. Используем перенаправление вывода (>), утилиту чтения из потока/файла (cat) и символ перевода строки (EOL), чтобы создался файл с нужным содержимым в

домашнем каталоге пользователя **postgres** (не забываем про функцию random – присваиваем переменной r значение) – все четыре строки сразу копируем и вставляем в командную строку:

```
cat > ~/workload.sql << EOL
\set r random(1, 5000000)
SELECT id, fkRide, fio, contact, fkSeat FROM book.tickets
WHERE id = :r;
EOL
```

Почему не написали вместо переменной :r сразу функцию random?

Ответ очень тонок – функция random имеет свойство волатильности (volatile)[84], то есть **функция не будет закэширована** и, **внимание**, **будет вызываться для каждой строки из выборки (SELECT)**!

Ещё раз – вместо того, чтобы сравнивать значение с переменной, для каждой строки будет вызываться функция рандом и сравнивать каждый раз с новым значением – итого 6 миллионов раз будет вызвана функция и результат будет совершенно непредсказуем, а также значительно дольше выполняться.

Наконец, запустим бенчмарк:

```
/usr/lib/postgresql/16/bin/pgbench -c 8 -j 4 -T 10 -f ~/workload.sql -U postgres thai
```

```
transaction type: /var/lib/postgresql/workload.sql
scaling factor: 1
query mode: simple
number of clients: 8
number of threads: 4
maximum number of tries: 1
duration: 10 s
number of transactions actually processed: 307972
number of failed transactions: 0 (0.000%)
latency average = 0.260 ms
initial connection time = 10.142 ms
tps = 30827.296690 (without initial connection time)
```

30к[85] TPS на такой скромной ВМ, что очень неплохо.

Рассмотрим второй профиль нагрузки – пишущая нагрузка.

Генерируем новую случайную поездку (также копируем все строки для автоматического формирования нужного файла):

```
cat > ~/workload2.sql << EOL
INSERT INTO book.tickets (fkRide, fio, contact, fkSeat)
```

[84] Volatile URL: https://www.postgresql.org/docs/current/xfunc-volatility.html (дата обращения 16.01.2024) [70]

[85] тысяч

```
    VALUES (
    ceil(random()*100)
    , (array(SELECT fam FROM
book.fam))[ceil(random()*110)]::text || ' ' ||
        (array(SELECT nam FROM
book.nam))[ceil(random()*110)]::text
        ,('{"phone":"+7' || (1000000000::bigint +
floor(random()*9000000000)::bigint)::text || '"}')::jsonb
        , ceil(random()*100));
    EOL
```

Запускаем бенчмарк:

```
    /usr/lib/postgresql/16/bin/pgbench -c 8 -j 4 -T 10 -f
~/workload2.sql -U postgres -p 5432 thai
```

И получаем ошибку:

```
DETAIL:  Failing row contains (5185877, 81, null, {"phone": "+76839846020"}, 47).
pgbench: error: client 5 script 0 aborted in command 0 query 0: ERROR:  null value in column "fio" of relation
 "tickets" violates not-null constraint
```

Мы сделали поле не пустое, а из-за небольшой ошибки в генерации схемы иногда получаются пустые фамилии. *Генерация схемы выходит за рамки книги и рассматривается на курсе, в том числе поиск ошибки.*

Возвращаем возможность NULL в поле fio:

```
    psql -d thai -c "ALTER TABLE book.tickets ALTER COLUMN fio
DROP NOT NULL;"
```

И запустим бенчмарк:

```
transaction type: /var/lib/postgresql/workload2.sql
scaling factor: 1
query mode: simple
number of clients: 8
number of threads: 4
maximum number of tries: 1
duration: 10 s
number of transactions actually processed: 68447
number of failed transactions: 0 (0.000%)
latency average = 1.168 ms
initial connection time = 9.392 ms
tps = 6847.289645 (without initial connection time)
```

Значительно упало количество запросов. Это также связано с тем, что, если встречаются в транзакции только SELECT, то PostgreSQL создаёт

виртуальную транзакцию, а не обычную, и это быстрее. Дальше в книге ещё будем затрагивать этот момент.

Итого, в этой главе рассмотрели сложности выбора архитектуры для развёртывания PostgreSQL, как выбрать ОС и её настроить на оптимальную производительность, какие параметры необходимо отрегулировать при первоначальной установке PostgreSQL и на какие подводные камни нужно обратить внимание. Создали профили для дальнейших бенчмарков и сравнения производительности при изменении того или иного параметра.

Все команды, рассмотренные в этой главе, можно найти на Гитхабе в файле https://github.com/aeuge/Postgres16book/blob/main/scripts/01/tuning.txt