Применение алгоритмов сжатия для ускорения загрузки веб-приложений.

1. Вводная часть (дает читателю понять проблему)

Основная задача - обосновать и реализовать алгоритм ддинамического управления параметрами сжатия для оптимизации загрузки и работы Web-приложений в условиях изменяющейся сетевой нагрузки. В рамках работы предполагается разработка системы, способной адаптировать параметры сжатия в реальном времени для обеспечения бесперебойного функционирования и быстрого отклика приложений, учитывая изменение числа пользователей и интенсивности их активности.

2. Описание объекта исследования (ограничить объем исследования)

В ходе данной работы мы будем тестировать алгоритмы сжатия на примере REST API (Representational State Transfer Application Programming Interface / Программный интерфейс приложения для передачи репрезентативного состояния) SPA (Single page application / Одностраничное веб-приложение) веб-приложения, выполняющего роль видеохостинга.

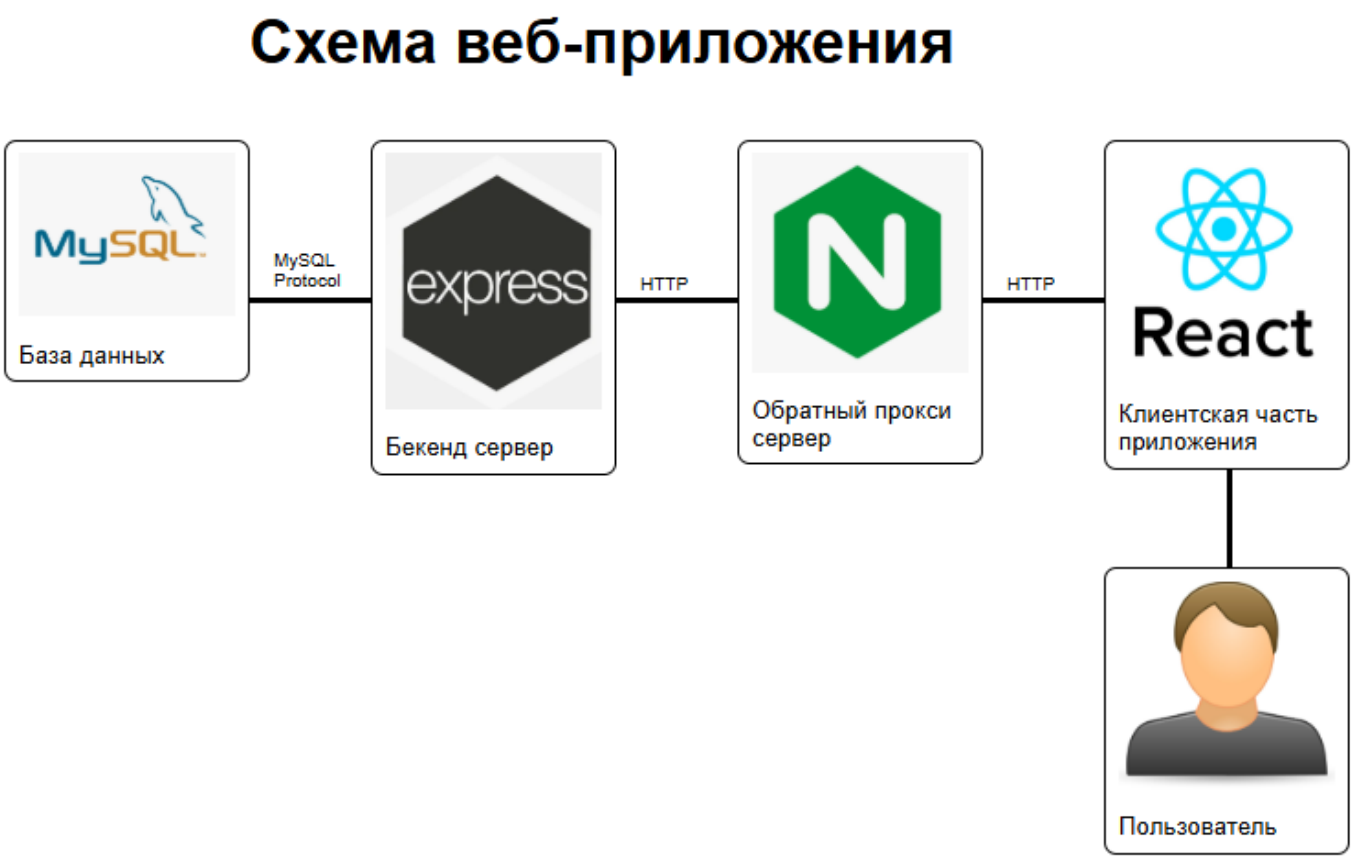
Видеохостинг должен выполнять следующую бизнес-логику:

- Возможость добавлять, удалять, просматривать видео
- Аутентифицировать пользователей
- Подбирать персональные рекомендации
- Оставлять комментарии и ставить лайки под видео

Стек технологий

Используется следующий стек технологий:

- Express JS - бекенд сервер
- React JS - фронтэнд часть приложения
- MySQL - реляционная база данных
- Nginx - обратный прокси сервер



Вкатце опишу для чего нужен каждый элемент смехы:

- **Бекенд** сервер обрабатывает запросы от пользователя, отвечает за аутентификацию, реализует бизнес логику. Например: подобрать рекомендации видео для конкретного пользователя, загрузить/удалить видео от пользователя с правами администратора
- **Фронтэнд** - это часть веб-приложения, что находится на стороне пользователя. Пользователь загружает статические файлы сайта, которые в последующем исполняются в браузере. HTML отвечает за разметку, CSS добавляет стили, а JS отвечает за функциональность - делает сайт живым.
- **База данных** необходима для удобного хранения большого колличества данных
- **Обратный прокси сервер** выполняет вспомогательные операции с запросами. Помимо перенаправления запросов, он также может выполнять сжатие, кеширование, предоставлять доступ к файлам балансировать нагрузку

Характеристи железа сервера

Все части веб приложения, разумеется кроме клиентской части, находятся на удалённом VPS сервере со следующими характеристиками:

- CPU 1 vCPU
- RAM 2 GB
- Storage 20 GB
- Speed 500 Mbps

Сценарий работы пользователя

Пользователь переходит по ссылке или вводит в строку поиска `http://example-videohosting.ru`. В этом случае происходит HTTP Get запрос, по которому обычно находится `index.html` файл.

Обычно этот файл выглядит следующим образом:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Videohosting</title>
    <link rel="stylesheet" href="./main.css">
    <script src="./main.js"></script>
</head>
<body>
    <img src="./assets/images/image.png">
    ...
</body>
</html>
```

После загрузки файла браузер начинает его парсить: строится DOM (Document Obejct Model / представление HTML-документа в виде дерева тегов), подгружатся CSS файлы (`<link rel="stylesheet" href="./main.css">`), подгружаются и исполняются JS файлы, подгружатся и рендарятся изображения. Когда этот процесс завершится, можно считать, что приложение полностью загружено и готово к использованию.

Далее пользователь может выполнять HTTP запросы для отправки форм, подзагрузки данных с бекенд сервера.

Статические данные - это данные, которые неизменны для каждого пользователя. К ним можно отнести HTML, JS, CSS файлы сайта и в целом все изображения

Динамические данные - это данные, меняющиеся во время работы приложения. К ним можно отнести информацию о видео (видео могут добавляться/удаляться, изменяется количество просмотров, лайков и т.д)

Мы отдельно рассмотрим сжатие статических и динамических данных.

Далее чем пойдёт об алгоритмах и форматах кодирования, но перед этим необходимо разобраться, что такое алгоритм Хаффмана

Алгоритм Хаффмана

Префиксный код - код, удовлетворяющий условию: если в коде есть слово a, то слова ab, где b - непустая строка, не существует

Идея алгоритма состоит в том, что если мы знаем вероятности появления символов, можно описать процедуру построения оптимальных префиксных кодов. Символам с наибольшей вероятностью ставятся в соотвествие более короткие коды.

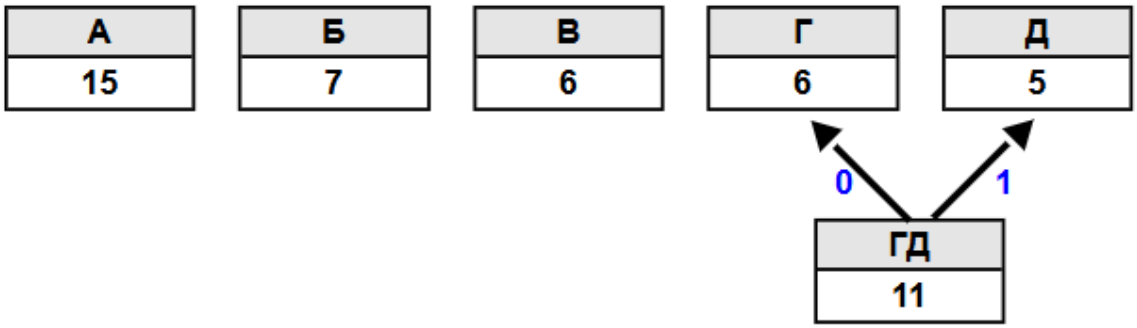
Алгоритм на входе получает таблицу частотностей символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана:

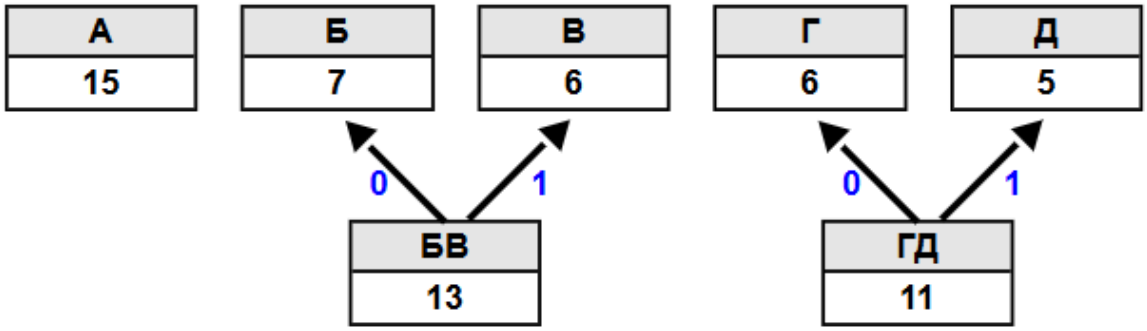
1. Символы входного алфавита образуют список свободных узлов. Каждый лист имеет вес, который может быть равен либо вероятности, либо количеству вхождений символа в сжимаемое сообщение.
2. Выбираются два свободных узла дерева с наименьшими весами.
3. Создается их родитель с весом, равным их суммарному весу.
4. Родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка.
5. Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой — бит 0. Битовые значения ветвей, исходящих от корня, не зависят от весов потомков.
6. Шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

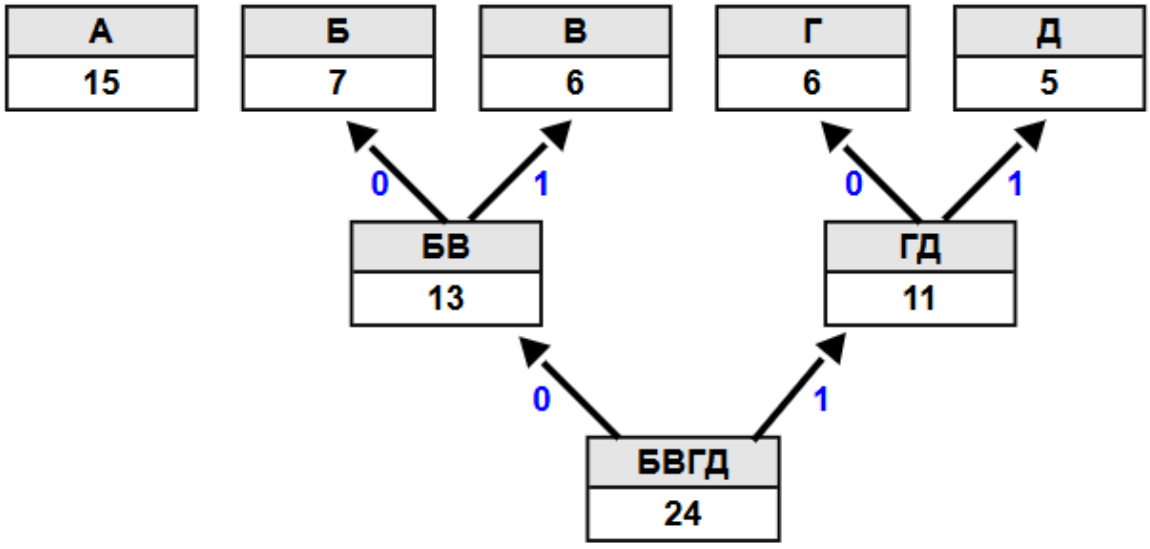
Построим дерево на конкретном примере, пусть в сообщении имеются символы А, Б, В, Г, Д с частотнастями:

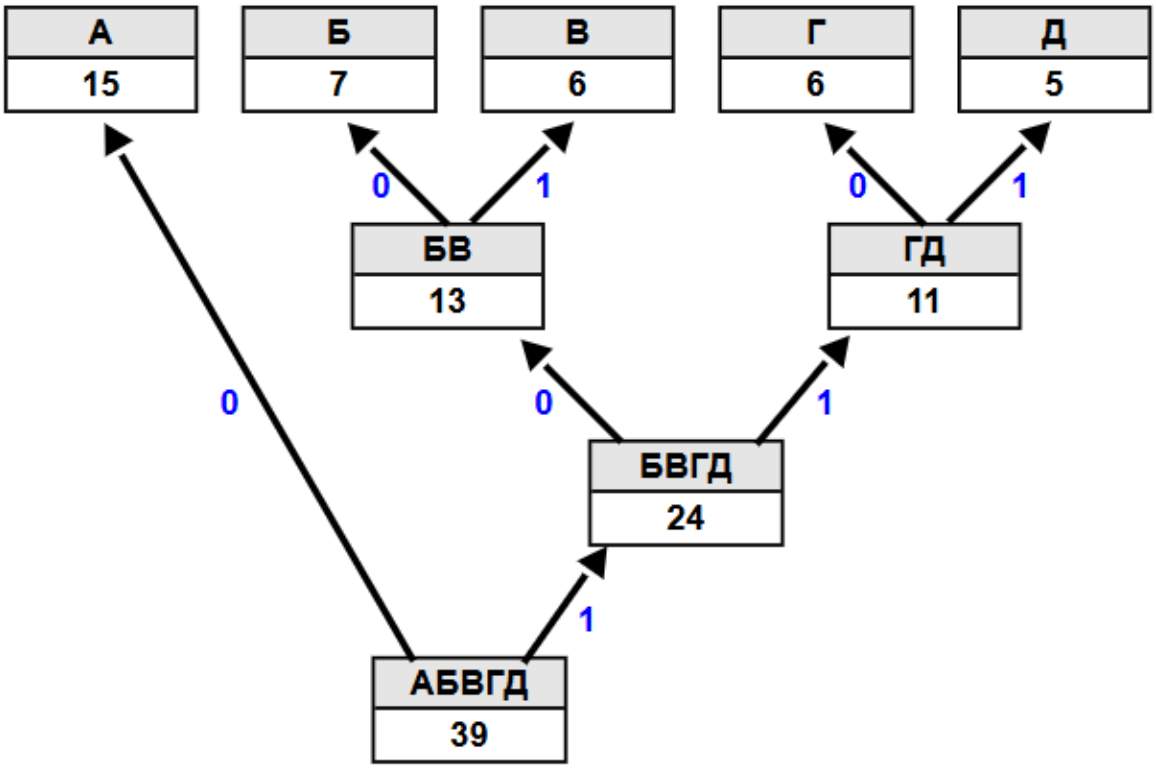
- А 15
- Б 7
- В 6
- Г 6
- Д 5

А	Б	В	Г	Д
15	7	6	6	5




А
15
Б
7
В
6
Г
6
Д
5
0
1
0
1
БВ
13
ГД
11


А
15
Б
7
В
6
Г
6
Д
5
0
1
0
1
БВ
13
ГД
11
0
1
БВГД
24



В итоге получим следующие префиксные коды:

- А 0
- Б 100
- В 101
- Г 110
- Д 111

О форматах изображений

Перед тем как приступить к описанию форматов приведу критерии, по которым я буду их сравнивать:

- **Степень сжатия** - отношение размеров исходного (формат PNG) и сжатого изображений
- **Скорость компрессии/декомпрессии** - как быстро сервер сожмёт изображение и как быстро клиент его разархивирует и отрендерит
- **Поддерживаемость разными браузерами** - важный параметр, так как формат с низким процентом поддреживаемостм не может использоваться в реальном приложении из-за риска потерять большую часть пользователей. Для оценки этого параметра я буду использовать данные с сайта `https://caniuse.com/`

На выбор имеется пять форматов:

- JPEG
- PNG
- WebP
- AVIF

- HEIC
- SVG

Поговорим про каждый из них по отдельности

JPEG

Filename extension: jpg jpeg jpe jif jfif jfi MIME type: image/jpeg JPEG (Joint Photographic Expert Group) поддерживается всеми браузерами, один из самых популярных форматов изображений. Поддерживаются изображения с линейным размером не более 65535 x 65536 пикселов.

Алгоритм JPEG наиболее эффективен для сжатия фотографий, содержащих реалистичные сцены с плавными переходами яркости и цвета. Для хранененния чертежей, текстовой и знаковой графики лучше использовать PNG, GIF, либо использовать режим сжатия Lossless JPG.

Уделим этому формату больше внимания чем остальным, так как он зарекомендавал себя временем и применяется много больше других. Позволяет сжимать изображения как с потерями, так и без потерь. На примере JPEG, мы также постараемся понять, как можно регулировать степень сжатия.

При сжатии изображение преоразуется из цветового пространства RGB в $YC_{b}C_{r}$. Стандарт ISO/IEC 10918-1 не регламентирует выбор именно YCbCr, допуская другие виды преобразования.

Y - компонента яркости, C_{B}, C_{R} - синяя и красная цветоразностные компоненты

Преобразование YCbCr можно представить следующими формулами:

$ Y = 0 + (0.299 * R) + (0.587 * G) + (0.114 * B) $

$ C_{B} = 128 - (0.168736 * R) - (0.331264 * G) + (0.5 * B) $

$ C_{R} = 128 + (0.5 * R) - (0.418688 * G) - (0.081312 * B) $

И обратно:

$ R = Y + 1.402 * (C_{R} - 128) $

$ G = Y - 0.34414 * (C_{B} - 128) - 0.71414 * (C_{R} - 128) $

$ B = Y + 1.772 * (C_{B} - 128) $

После преобразования $RGB -> YC_{B}C_{R}$ для каналов C_{B}, C_{R}, отвечающих за цвет может выполняться «прореживание», где каждому блоку из 4 пикселей (2x2) ставится в соответсвие усреднённое значение C_{B}, C_{R} (Схема прореживания 4:2:0). При этом для каждого блока 2x2 вместо 12 значений (4Y, 4C_{B}, 4C_{R}) ставится в соответствие 6 значений (4Y, C_{B}, C_{R}). Если к изображению предъявляются более высокие требования по качеству, могут применяться схемы по сжатия (4:4:0), (4:2:2) или не применятся вовсе. (4:4:4)

Стандарт допускает прореживание блоков не 2*2, а 4*1 или 1*4, но на практике такие схемы применяются довольно редко.

После этого компоненты Y, C_{B}, $C_{R} разбивается на блоки 8*8. Каждый такой блок подвергается дискретному косинусному преобразованию (ДКП)

Формула для вычисления коэффициентов ДКП $F(u,v)$ блока 8×8:

Коэффициенты ДКП $F(u,v)$ вычисляются по формуле:

$$ F(u,v) = \frac{C(u)C(v)}{4} \sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cdot \cos\left(\frac{(2x+1)u\pi}{16}\right) \cos\left(\frac{(2y+1)v\pi}{16}\right) $$

где:

- $f(x,y)$ — значение пикселя в позиции (x,y),
- u,v — частотные координаты (от 0 до 7),
- $C(k)$ — нормировочный коэффициент: $$ C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{при } k=0, \\ 1 & \text{иначе}. \end{cases} $$

Далее к полученной матрице применяется квантование (зависит от степени сжатия):

$ Q(u, v) = round(\frac{F(u, v)}{Q_{table}(u, v)}) $

После этого многие высокочастотные коэффициенты становятся нулевыми.

Далее полученные коэффициенты записываются в массив и кодируются с помощью кодов Хаффмана.

PNG

Filename extension: png MIME type: image/png PNG (Portable Network Graphics) — это растровый формат изображения, который предлагает сжатие без потерь. Еще одна особенность PNG формата — изображение может содержать прозрачные области. Поддерживается всеми браузерами Согласно сайту caniuse.com поддерживается 92,6% браузеров весной 2025 года.

WebP

Filename extension: webp

MIME type: image/webp

WebP — это формат изображений, разработанный Google, который предлагает сжатие изображений с потерями и без потерь. Он призван обеспечить высокое качество изображений при меньших размерах файлов.

Согласно сайту caniuse.com поддерживается 95,92% браузеров весной 2025 года.

AVIF

AVIF (AV1 Image File Format) — это современный формат изображений, основанный на технологии сжатия AV1. Он предназначен для обеспечения высокого качества изображений при более низком размере файлов. Согласно сайту caniuse.com поддерживается 92,6% браузеров.

Согласно сайту caniuse.com поддерживается 93,71% браузеров весной 2025 года.

HEIF/HEIC

Filename extension: heif heic

MIME type: image/heif image/heic

HEIF (High Efficiency Image Format) — это современный формат изображений, который обеспечивает высокую эффективность сжатия и поддержку различных функций, таких как анимация, HDR, прозрачность и многослойность.

HEIC (High Efficiency Image Container) — это контейнерный формат файла, который используется в том числе и для хранения изображений в формате HEIF, как пример можно привести Live Photos сделанные на iPhone.

Согласно сайту caniuse.com поддерживается 13,99% браузеров весной 2025 года.

SVG

Filename extension: svg MIME type: image/svg+xml

SVG (Scalable Vector Graphics) — это формат изображений, основанный на XML, который описывает двумерные векторные графики с использованием векторных объектов, таких как линии, кривые, формы и текст.

Согласно сайту caniuse.com поддерживается 96,99% браузеров весной 2025 года.

О алгоритмах сжатия текстовых данных

На данный момент, браузером поддерживаются два основных алгоритма сжатия текстовых файлов (css, html, js):

- Gzip
- Brotli

Перед тем как приступить к описанию приведённых алгоритмов, следует рассказать о семействе алгоритмов LZ, в частности LZ77, так как оба алгоритма основываются на нём. Алгоритмы словарного сжатия Зива-Лемпела появились во второй половине 70-х гг. Это были так называемые алгоритмы LZ77 и LZ78, разработанные совместно Зивом (Ziv) и Лемпелом (Lempel). В дальнейшем первоначальные схемы подвергались множественным изменениям, в результате чего мы сегодня имеем десятки достаточно самостоятельных алгоритмов и бессчетное количество модификаций. LZ77 и LZ78 являются универсальными алгоритмами сжатия, в которых словарь формируется на основании уже обработанной части входного потока, т. е. адаптивно. Принципиальным отличием является лишь способ формирования фраз.

Алгоритм LZ77

Алгоритм LZ77 является родоначальником целого семейства словарных схем - так называемых алгоритмов со скользящим словарем, или скользящим окном. Действительно, в LZ77 в качестве словаря используется блок уже закодированной последовательности. Как правило, по мере выполнения обработки положение этого блока относительно начала последовательности постоянно меняется, словарь "скользит" по входному потоку данных. Скользящее окно имеет длину N, т. е. в него помещается N символов, и состоит из двух частей:

- последовательности длины W=N-n уже закодированных символов, которая и является словарем;

- упреждающего буфера, или буфера предварительного просмотра (lookahead), длины n; обычно n на порядки меньше W. Пусть к текущему моменту времени мы уже закодировали t символов, последние W символом будут составлять наш словарь. На каждой итерации алгоритма мы ищем самое длинное вхождение префиксной строки упреждающего буфера, начиная с t+1 символа в словаре + упреждающем буфере, но важно, чтобы часть строки лежала в словаре. Полученная в результате поиска фраза кодируется с помощью двух чисел:

1. смещения (offset) от начала буфера
2. длины соответствия, или совпадения Смещение и длина соответствия играют роль указателя (ссылки), однозначно определяющего фразу. Дополнительно в выходной поток записывается символ s (подумайте зачем), непосредственно следующий за совпавшей строкой буфера.

Что касается декодирования сжатых данных, то оно осуществляется путем простой замены кода на блок символов, состоящий из фразы словаря и явно передаваемого символа. Естественно, декодер должен выполнять Те же действия по изменению окна, что и кодер. Фраза словаря элементарно опре- деляется по смещению и длине, поэтому важным свойством LZ77 и прочих алгоритмов со скользящим окном является очень быстрая работа декодера. Алгоритм декодирования может иметь следующий вид:

Алгоритмы со скользящим окном характеризуются сильной несимметричностью по времени - кодирование значительно медленнее декодирования, поскольку при сжатии много времени тратится на поиск фраз.

Формат Deflate

Формат словарного сжатия Deflate, предложенный Кацем (Katz), используется популярным архиватором GZIP. Сжатие осуществляется с помощью алгоритма типа LZH, иначе говоря, указатели и литералы кодируются по методу Хаффмана. Формат специфицирует только работу декодера, т. е. определяет алгоритм декодирования, и не налагает серьезных ограничений на реализацию кодера. В принципе в качестве алгоритма сжатия может применяться любой работающий со скользящим окном, лишь бы он исходил из стандартной процедуры обновления словаря для алгоритмов семейства LZ77 и использовал задаваемые форматом типы кодов Хаффмана Закодированные в соответствии с форматом Deflate данные представляют собой набор блоков, порядок которых совпадает с последовательностью соответствующих блоков исходных данных. Используется 3 типа блоков закодированных данных:

1. состоящие из несжатых данных;
2. использующие фиксированные коды Хаффмана;
3. использующие динамические коды Хаффмана. Длина блоков первого типа не может превышать 64 Кб, относительно других ограничений по размеру нет. Каждый блок типа 2 и 3 состоит из двух частей:

- описания двух таблиц кодов Хаффмана, использованных для кодирования данных блока;
- собственно закодированных данных. Коды Хаффмана каждого блока не зависят от использованных в предыдущих блоках. Само описание динамически создаваемых кодов Хаффмана является, в свою очередь, также сжатым с помощью фиксированных кодов Хаффмана, таблица которых задается форматом. Алгоритм словарного сжатия может использовать в качестве словаря часть предыдущего блока (блоков), но величина смещения не может быть больше 32 Кб. Данные в компактном представлении состоят из кодов элементов двух типов:
- литералов (одиночных символов);

- указателей имеющихся в словаре фраз; указатели состоят из пары <длина совпадения, смещение>

Длина совпавшей строки не может превышать 258 байт, а смещение фразы - 32 Кб. Литералы и длины совпадения кодируются с помощью одной таблицы кодов Хаффмана, а для смещений используется другая таблица; иначе говоря, литералы и длины совпадения образуют один алфавит. Именно эти таблицы кодов и передаются в начале блока третьего типа.

Алгоритм декодирования Deflate

Сжатые данные декодируются по следующему алгоритму: TODO

Алгоритм словарного сжатия для DEFLATE

Как уже указывалось, формат Deflate не имеет четкой спецификации алгоритма словарного сжатия. Разработчики могут использовать какие-то свои алгоритмы, подходящие для решения специфических задач. В качестве примера приведу свободный от патентов алгоритм сжатия для Deflate, используемый в разрабатываемой Info-ZIP group утилите Zip.

Формат сжатия Gzip (GNU zip)

Это утилита для сжатия и распаковки файлов, которая широко используется в UNIX-системах. Формат файла gzip состоит из 3 основных частей:

1. Заголовок: Содержит информацию о типе файла, имени оригинального файла, времени создания, уровне сжатия и других параметрах.
2. Тело: Содержит сжатые данные, выполненные с помощью алгоритма DEFLATE.
3. Контрольная сумма (CRC-32) и Размер оригинала: Эти данные предоставляют возможность проверки целостности и правильности распаковки данных.

Заголовок файла содержит следующие ключевые поля:

- ID1 и ID2: Идентификаторы, указывающие на формат gzip (значения - 0x1F и 0x8B).
- CM: Метод сжатия (в gzip используется значение 8 для DEFLATE). *
- FLG: Биты флагов, которые указывают наличие дополнительных полей и информации.
- MTIME: Время последней модификации оригинального файла.
- XFL: Дополнительная информация о методе компрессии.
- OS: Платформа, на которой был создан/сжат файл (gzip).

Другие значения для этого поля теоретически могут быть использованы для обозначения различных методов сжатия, но сам формат gzip и его стандартная реализация подразумевают использование только DEFLATE. На практике, если gzip файл содержит метод сжатия, отличный от DEFLATE, стандартные утилиты для работы с gzip файлами могут его не поддерживать и не распознать.

За счёт чего можно добиться разных степеней сжатия?

Т.к формат Deflate не имеет четкой спецификации алгоритма словарного сжатия, разработчики могут использовать различные модификации LZH, подбирая параметры для обеспечения желаемого

соотношения скорости и коэффициента сжатия. В приведённом примере алгоритма сжатия DEFLATE, можно менять match_len(t+1) = L

Инструменты разработчика браузера

Браузеры и JavaScript

На сегодняшний день существует два вида браузеров:

- Основанные на движке Chromium V8 от Google (https://github.com/v8/v8) такие, как Goooogle Chrome, Yandex Browser, Microsoft Edge и другие
- Firefox, использующий Quantum

Дело в том, что JavaScript, использующийся в веб приложениях, - скриптованный язык, существует спецификация ESMA Script (https://262.ecma-international.org/), в которой написано, что должен делать язык, но реализация не указана. Реализация фукнций языка ложится на разработчиков браузеров. Здесь, в отличие от Си или Java, не существует компиляторов или Java Virtual Machine. Разрабочики точно не знают, что происходит "под капотом".

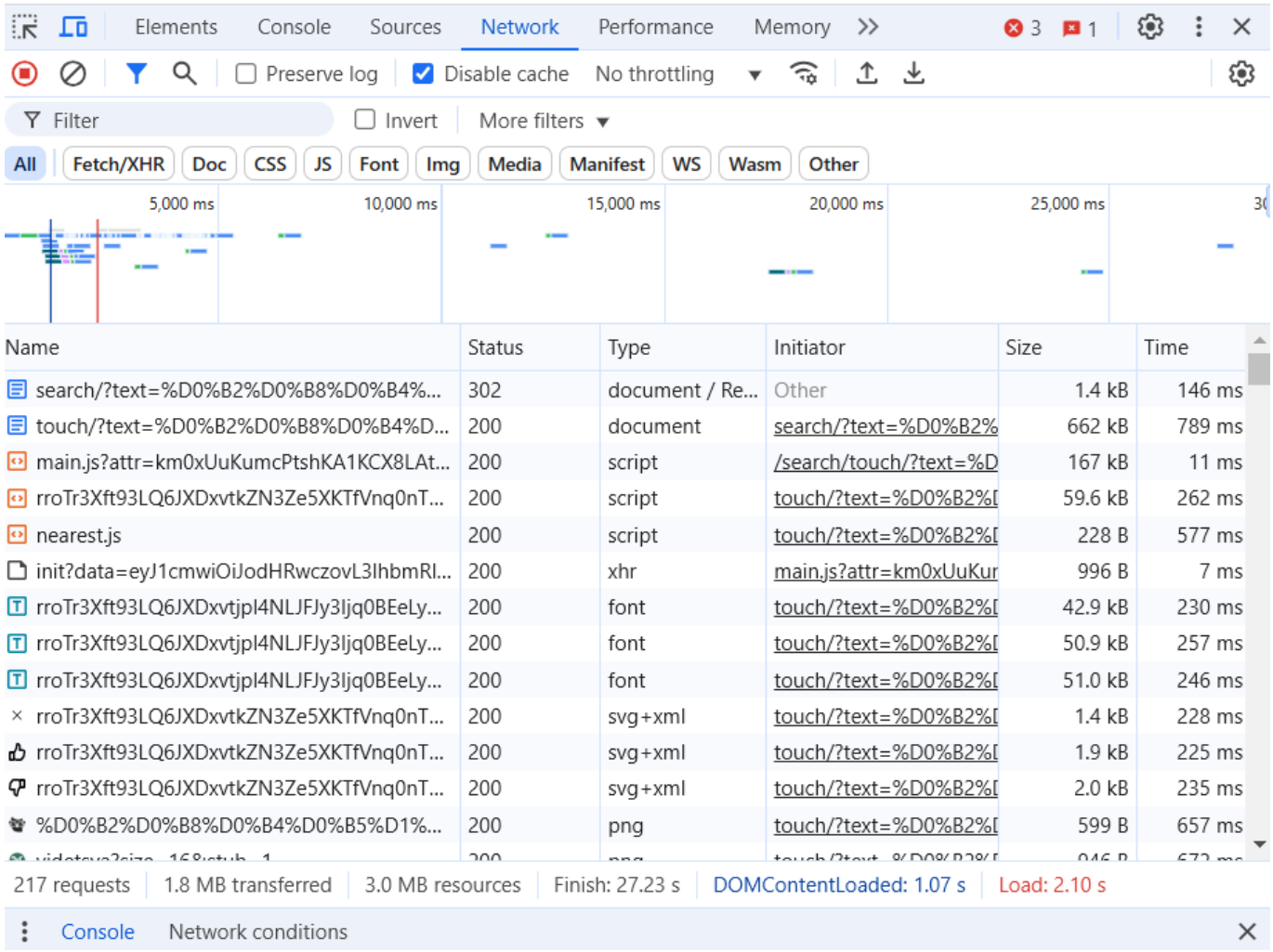
Для получения информации о времени выполнения, скорости загрузки, использования памяти можно использовать инструменты разработчика (DevTools). Её можно открыть в любом браузере, нажав F12

Какие функции предоставляет консоль разработчика

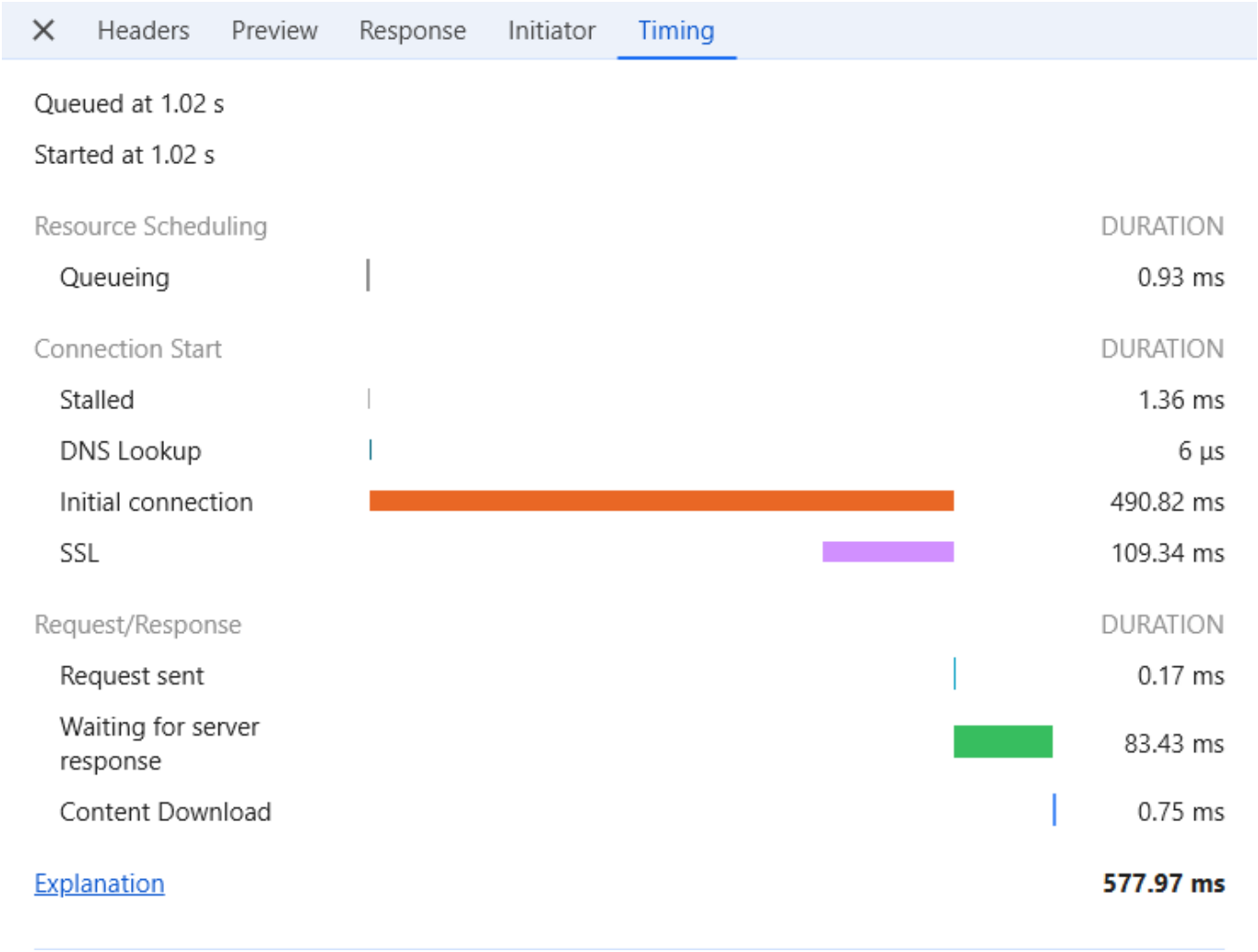
Реализации DevTools отличаются в зависимости от браузера. Но в целом, DevTools у браузеров на Chromium очень похожи. Я буду брать примеры из Google Chrome

Нас интересуют две вкладки:

Networks



На этой вкладке можно посмотреть все сетевые запросы, выполненные браузером на данной странице, узнать статус запроса, тип, размер ответа и общее время, потраченное на запрос,

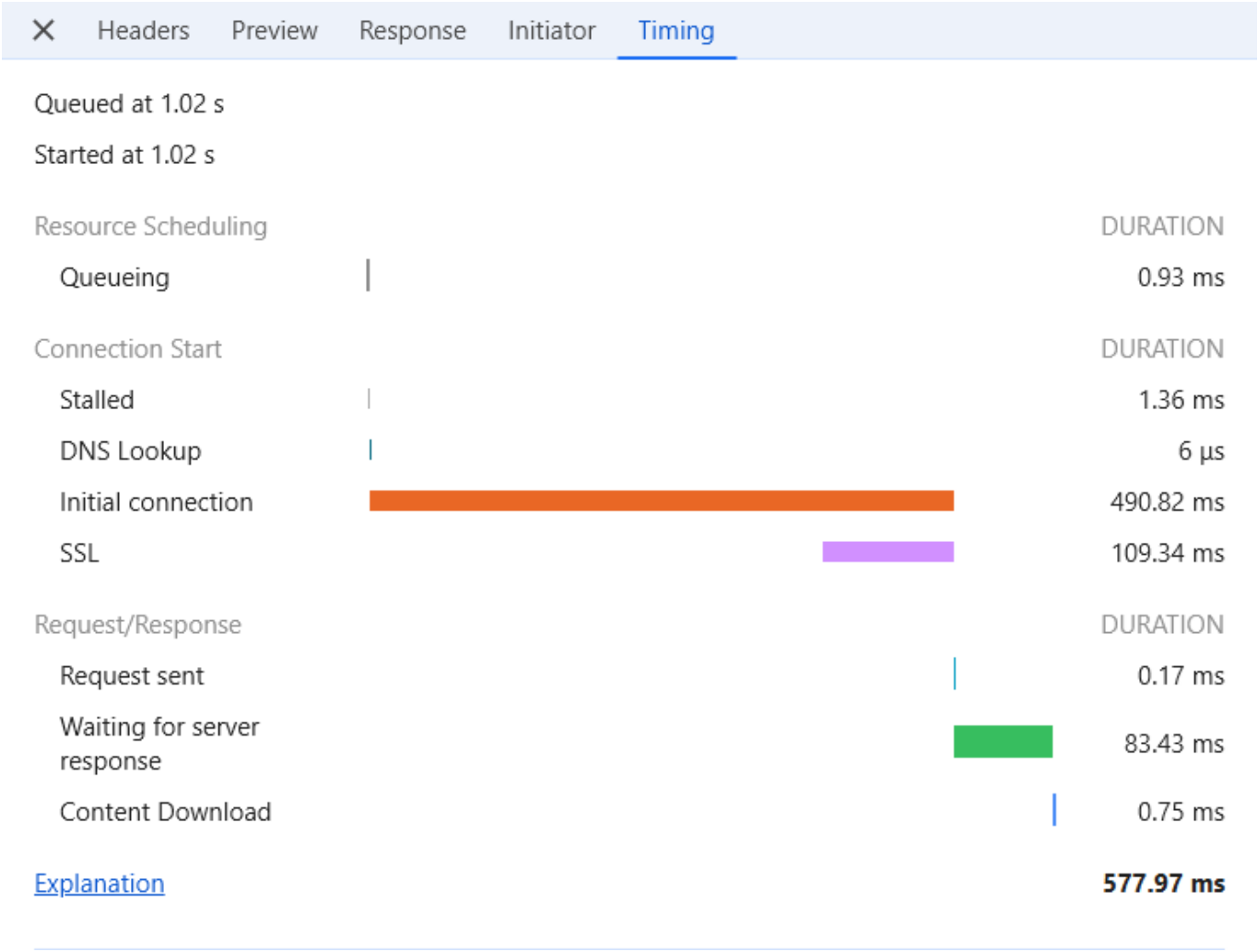


Если нажать на конкретный запрос, можно будет узнать такие параметры, как

- Время, когда запрос был отправлен (время отсчитывается от момента перехода на сайт)
- Queueing Очередь . Браузер ставит запросы в очередь перед началом соединения и когда: Есть запросы с более высоким приоритетом. Приоритет запроса определяется такими факторами, как тип ресурса, а также его расположение в документе. Для этого источника уже открыто шесть TCP-соединений, что является пределом. (Применимо только к HTTP/1.0 и HTTP/1.1.) Браузер ненадолго выделяет место в дисковом кеше.
- Stalled Запрос мог быть остановлен после начала соединения по любой из причин, описанных в Queueing
- Initial connection, физическая удалённость сервера, временные затраты на TSL/SSL handshake, если используется HTTPs
- Время, затраченное, на отправку запроса
- Время ожидание ответа от сервера
- Сколько времени затрачено на загрузку содержимого запроса

Как можно заметить, по этим параметрам не удаётся опередилить время, затраченное на декодирование запроса

Performance



Largest Contentful Paint (LCP) (https://web.dev/articles/lcp) сообщает о времени рендеринга самого большого изображения, текстового блока или видео, видимого в окне просмотра, по отношению к моменту, когда пользователь впервые перешёл на страницу.

На этой вкладке мы так же можем применить throttling к процессору или к сети

Для сети существует несколько режимов:

2. Low-end Mobile (регламентировано для 3G, медленное ограничение):

- Скорость загрузки (Download): 400 Kbps.
- Скорость отправки (Upload): 400 Kbps.
- Задержка (Latency): ~400 мс.

3. Regular 3G

- Скорость загрузки (Download): 750 Kbps.
- Скорость отправки (Upload): 250 Kbps.
- Задержка (Latency): ~100 мс.

4. Good 3G

- Скорость загрузки (Download): 1.5 Mbps.
- Скорость отправки (Upload): 750 Kbps.

- Задержка (Latency): ~40 мс.

Можно настроить и свою конфигурацию

Тротлинг процессора

Троттлинг процессора (CPU throttling) в DevTools используется для замедления работы процессора в целях эмуляции менее мощных устройств или реальных условий работы пользователей. Это особенно полезно для тестирования производительности, чтобы понять, как ваш сайт или приложение будет работать на слабых устройствах, таких как смартфоны, или в условиях высокой нагрузки.

Непроверенная информация !

Тротлинг в DevTools реализован программно. Браузер замедляет выполнение базовых функций JavaScript (sort, map, for) в указанное число раз (4x, 6x, 20x). Исходя из этого, можно предположить, что замедление CPU не отразиться на времени, затраченное на декодирование файлов

Также можно замерить, какое замедление процессора нужно применить для эмуляции мобильных устройств. На моём компьютере это оказалось от 2x (для среднего сегмента смартфонов) и 6x (для low сегмента).

3. Измерения и анализ полученных результатов

Сравнение форматов изображений

Изначально все изображения в png

Конвертация из PNG в JPEG с качеством 0.8, из PNG в WebP на сайте `https://image.online-convert.com/convert/` Из PNG в Avif на сайте `https://converter.app/png-to-avif` Из PNG в Heic на сайте `https://png2heic.com/`

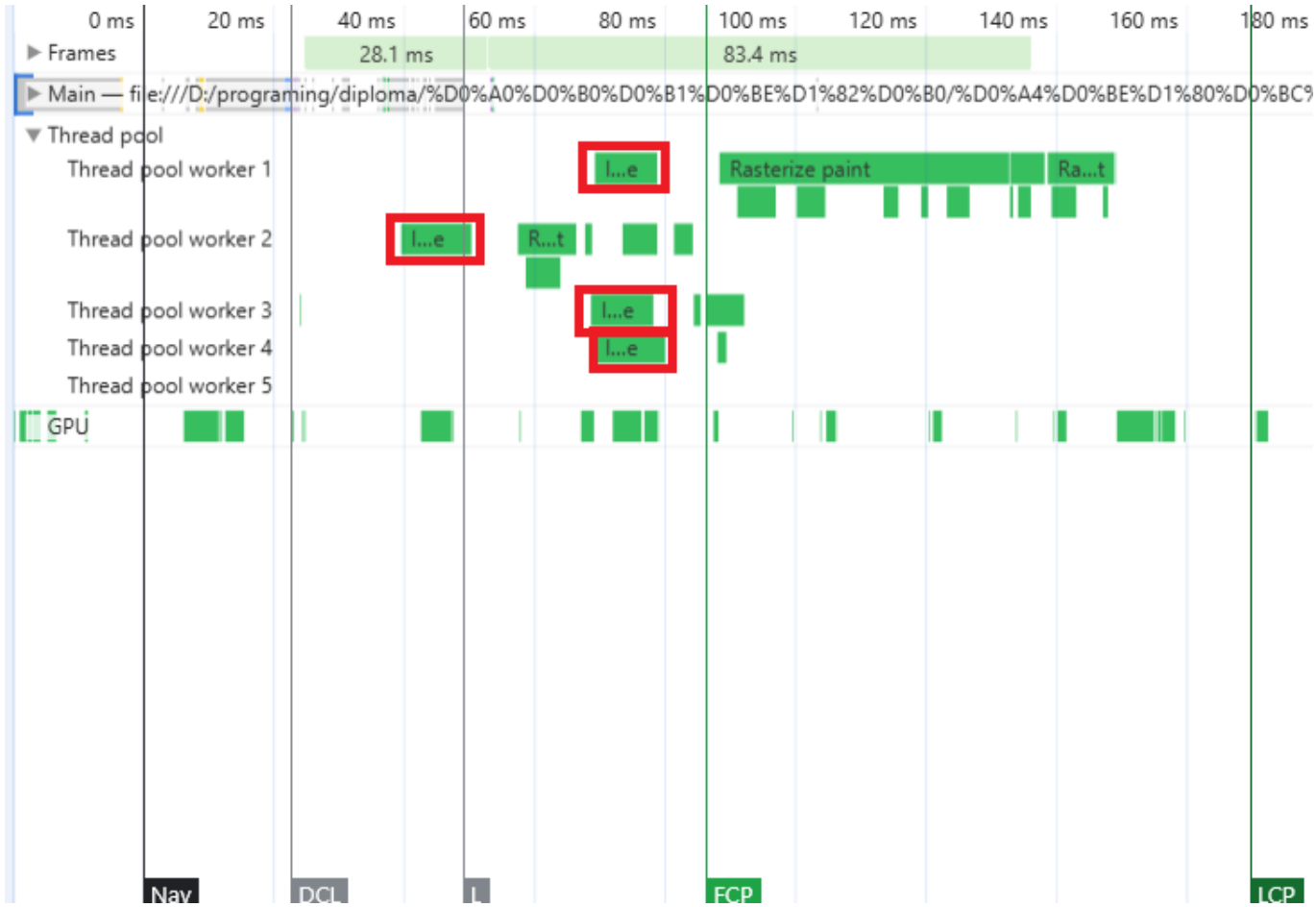
Выбраны 10 различных изображений (людей, природы, котов) в формате PNG и сконвертированы в JPG, Webp, Avif и Heic

Вес 10-ти изображений в форматах:

- PNG 18,0 Мб
- JPEG 1,66 Мб
- WebP 1,42 Мб
- Avif 1,00 Мб
- Heic 1,04 Мб

Измерение времени декодирования

Я начал с того, что попытался получить информацию о времени декодирования напрямую из DevTools. Для формата JPEG получилась следующая картина:



На данной диграмме мы видим временные интервалы соответствующие декодированию и растеризации изображений. По оси X отложено время в милисекундах, по оси Y представлены задействованные потоки процессора

Как можно заметить, браузер умеет декодировать изображения параллельно.

В основном реализуется следующах схема:

1. Браузер скачивает файл
2. Декодирует
3. Растеризует

Но, т.к JPEG поддерживает последовательное декодирование, может получится так, что браузер сначала декодирует часть изображения, растеризует декодированную часть, декодирует ещё одну часть и так далее. На практике я наблюдал разбиение изображения на три части

Для изображения весом 344kB я получил время декодирования: $T_{dec}=13,4 \pm 1,08мс$, время расторизации: $T_{ras}=8,9 \pm 0,91мс$

Дела обстаят хуже с другими форматами. Браузер не даёт информации о времени декодирования.

Для avif:

Время растеризации для avif $T_{ras}=34,9 \pm 3,21мс$, что заметно хуже чем у JPG

Мы можем измерить время, потраченное на декодирование + растеризацию косвенно. Предположим, что LCP измеряется по следующей формуле

$ T_{LCP} = T_{проч} + T_{down} + T_{dec+ras} $

где

- $T_{проч}$ - время на рендеринг DOM, время на отправку запроса HTTP и прочие функции браузера, которые не зависят от формата
- T_{down} - время на загрузку файла, для JPEG $T_{down} = 1,3 \pm 0,2 мс$, для PNG $T_{down} = 2,4 \pm 0,3 мс$. Этим можно пренебречь
- $T_{dec+ras}$ - время на декодирование и растеризацию

Тогда замерив LCP для разных форматов, можно найти разность $T_{dec+ras}$.

Результаты для LCP следующие:

Формат	**$\overline{LCP}$, мс**	**σ, мс**
JPEG	73,5	8,05
PNG	141,3	7,51
AVIF	139,5	6,6
WebP	125,3	6,51

Считая, что для JPEG $T_{dec+ras} = 22,8 \pm 2мс$. JPEG $T_{dec+ras}$:

Формат	**$\overline{LCP}$, мс**	**σ, мс**
JPEG	22,8	2

Формат	$\overline{LCP}$, мс	σ, мс
PNG	90,6	9,51
AVIF	88,8	8,6
WebP	74,6	8,51

Сжатие статических файлов браузера с помощью Gzip, Brotli

Что измеряем

Для алгоритмов gzip, brotli будут измерены

1. Степень сжатия
2. Время сжатия
3. Время декомпрессии

В качестве исходных данным будут использованы минифицированные версии трёх веб-приложений:

1. Dating App (DA)
2. Videohosting (VH)
3. Angular conduit (AC)

Технические характеристики устройства

Процессор: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz Оперативная память: 8,00 ГБ Операционная система: Windowws 11, версия 24H2, WSL Linux Ubuntu 20.04

Условия проведения эксперимента, обработка данных

Для сжатия использована утилита gzipper (https://www.npmjs.com/package/gzipper) на node.js Для декопрессии использованы утилиты gunzip, brotli на Linux

Для замеров времени декомпрессии использован hyperfine

При измерении скорости для каждой точки было проведено 20 измерений и применён метод усечённого среднего (10%) Так же построены минимумы по времени

Для точности измерений были отключены фоновые процессы операционной системы

В веб-приложениях были удалены файлы с изображениями, т.к они уже сжаты с помощью специальных алгортмов для изображений (jpg, webp, avif)

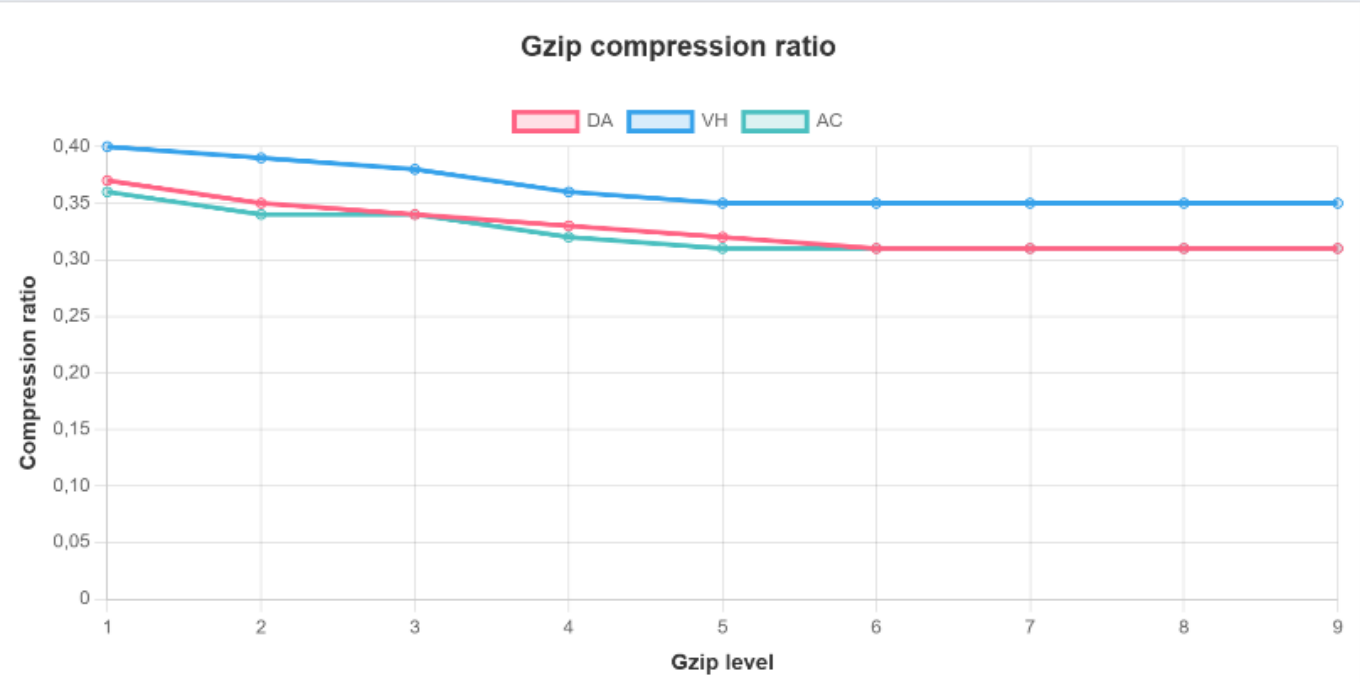
Изначальный размер

Изначальный размер приложений:

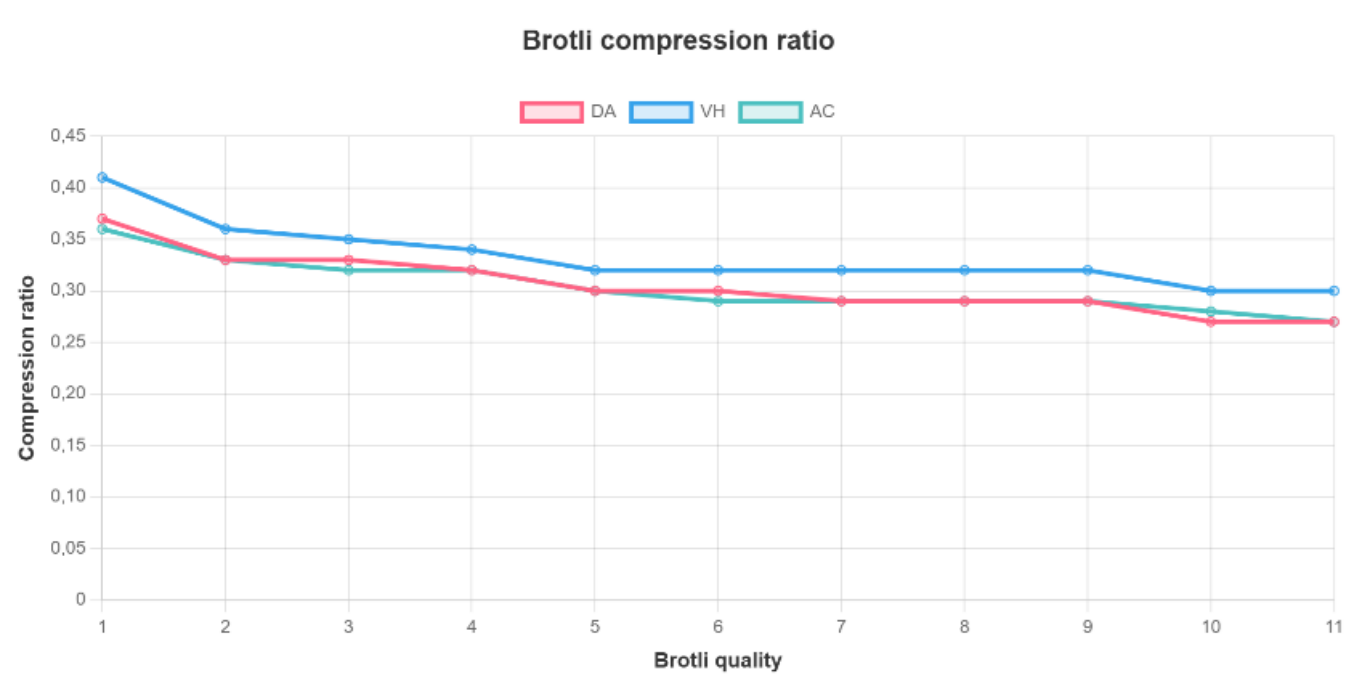
1. Dating App (DA) 999КБ
2. Videohosting (VH) 479КБ
3. Angular conduit (AC) 456КБ

Измерение степени сжатия

Для Gzip



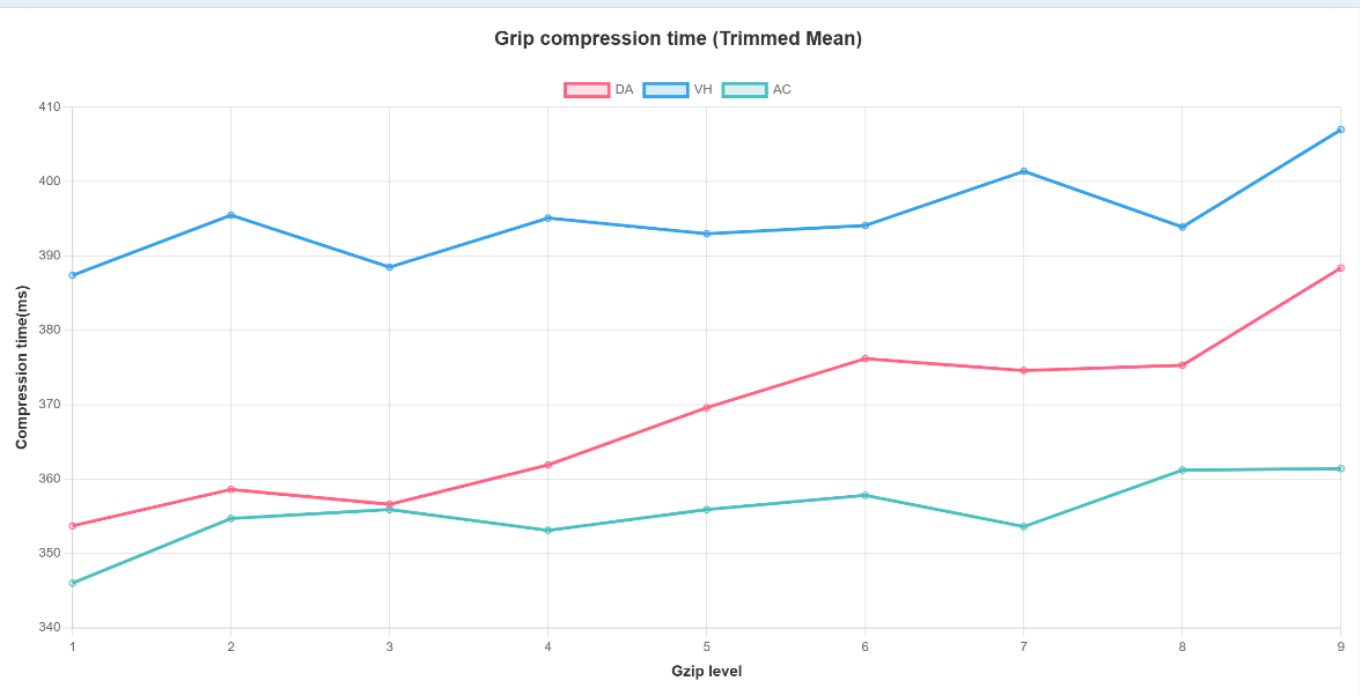
для Brotli



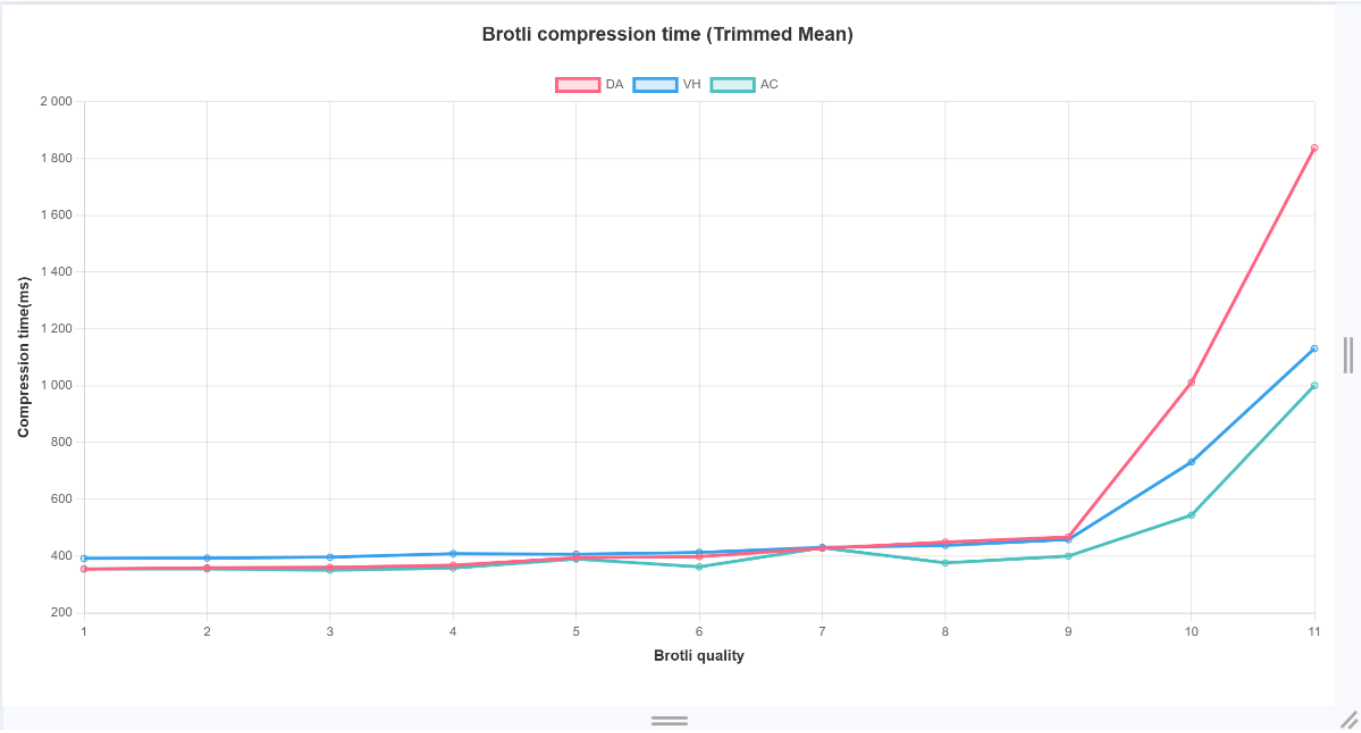
Измерение времени сжатия

Замеры времени сжатия производились с помощью gzipper. Подряд вводились команды из консоли, после каждого сжатия, ожидание 1-2 секунды

Для Gzip

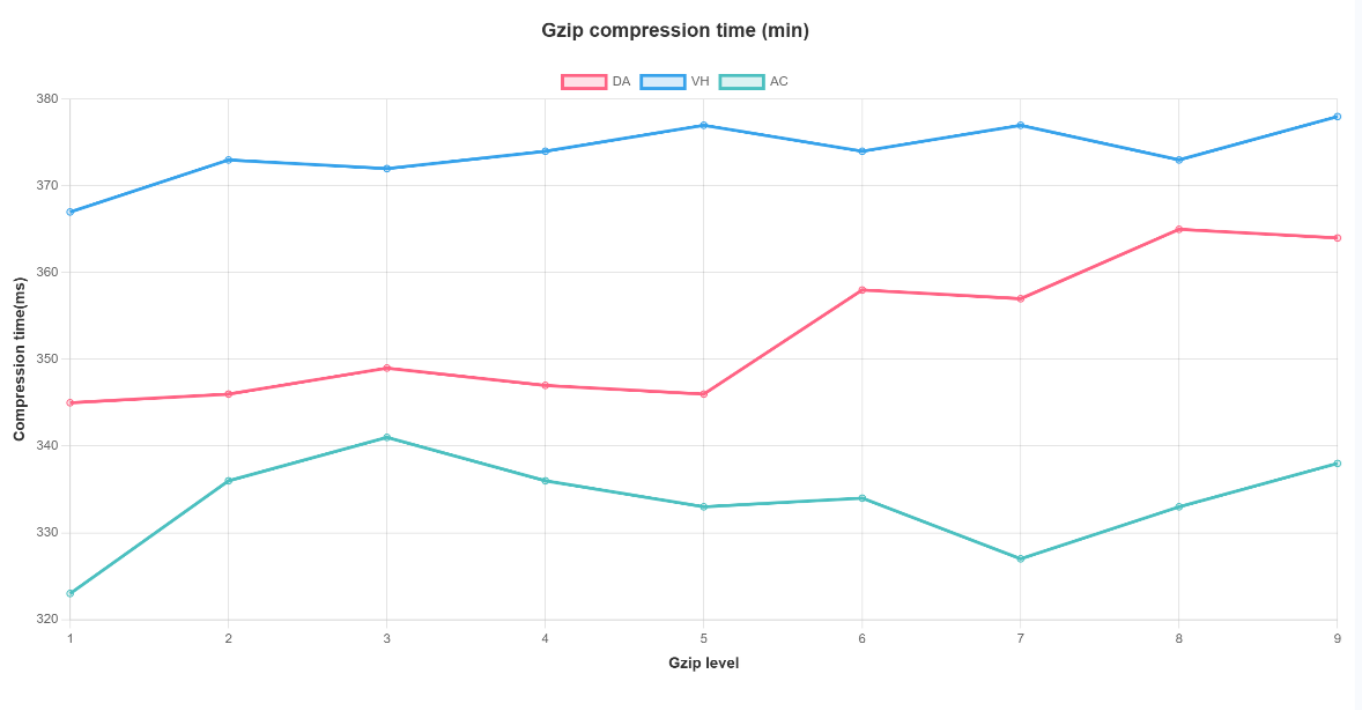


Для Brotli

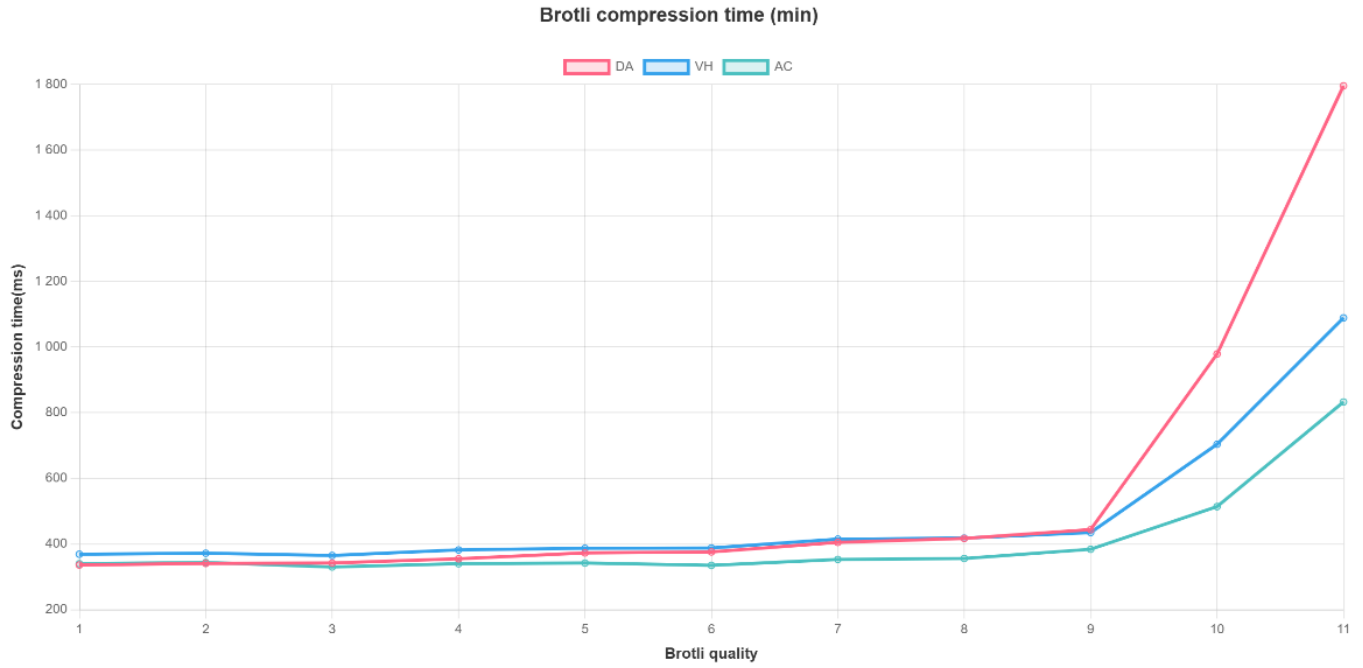


Также построены графики минимального времени. Вероятно, эта интерпретация лучше защищена от помех

Для Gzip

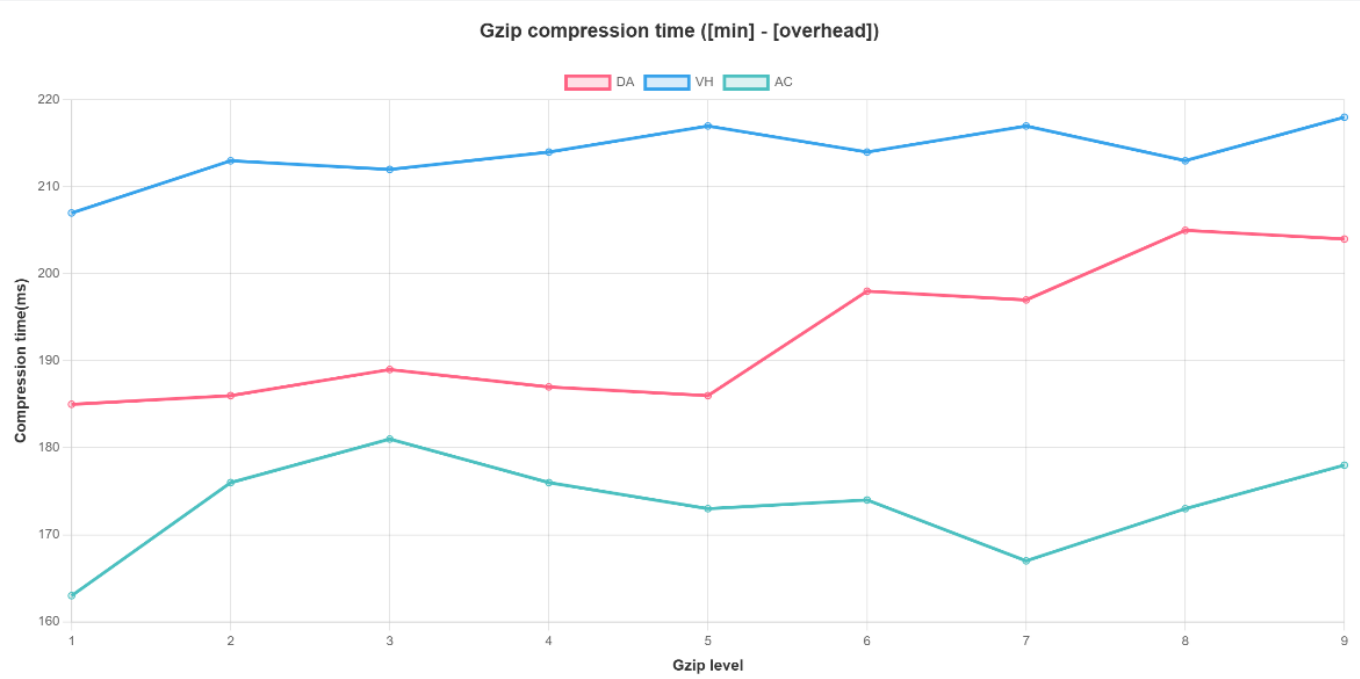


Для Brotli

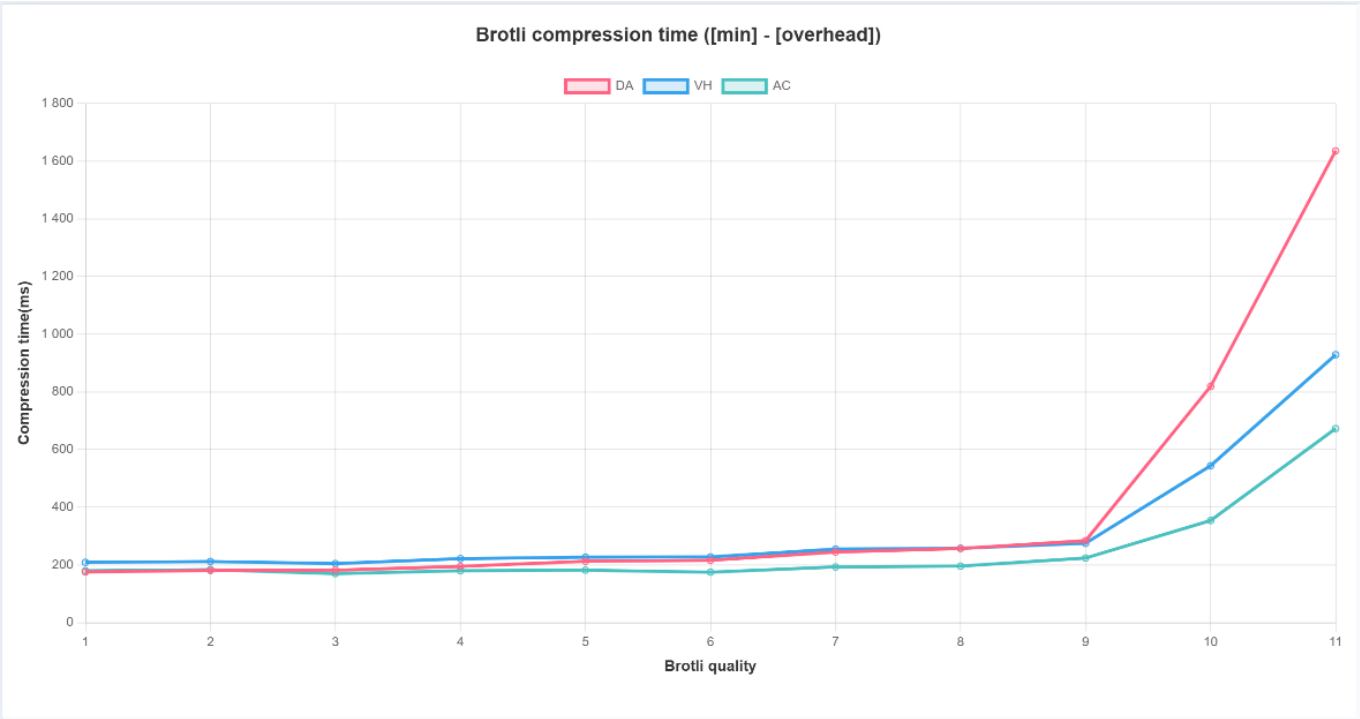


Было замечено, что gzipper тратит 160ms на пустой файл, вероятно это время требуется, чтобы запустить процесс, выделить память. Поэтому также построены графики за вычетом времени на накладные расходы

Для Gzip

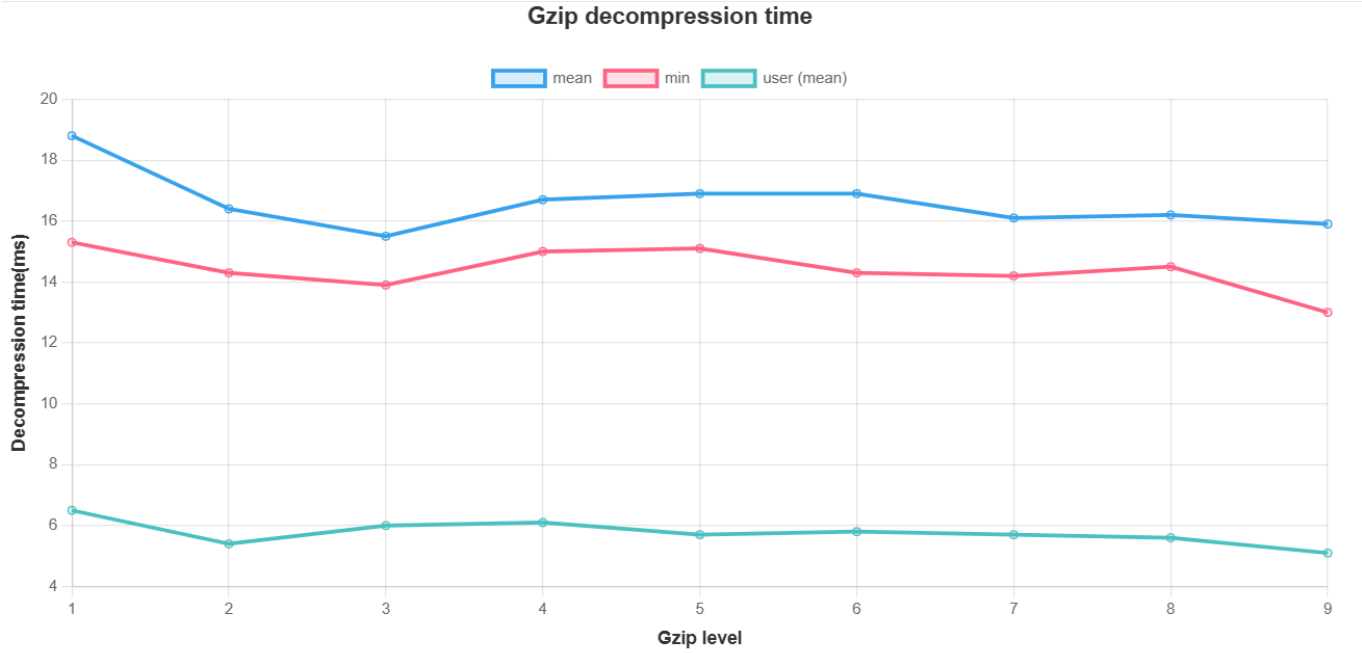


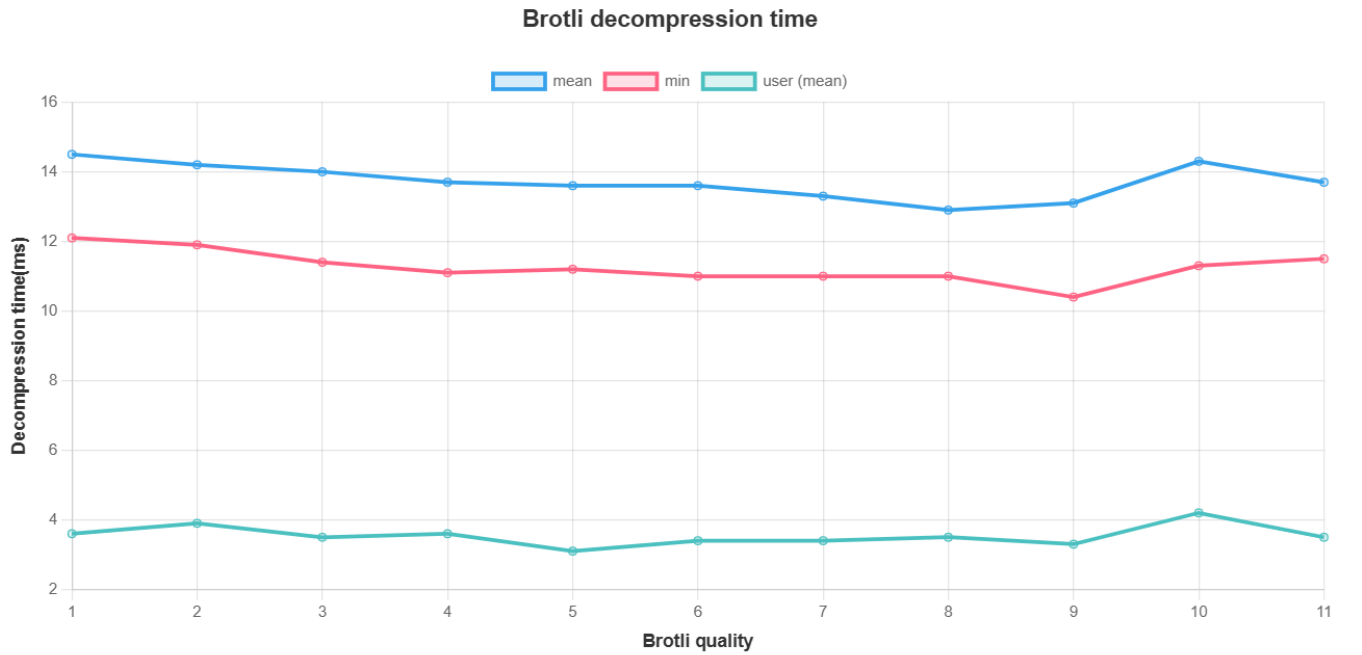
Для Brotli



Измерение времени декомпрессии

Измерения времени декомпрессии программой gunzip произведены на Linux 24.04. Для замеров использована утилита hyperfine.





Как видно, скорость декомпрессии почти не зависит от степени сжатия. Так же видно, что в среднем brotli быстрее разжимает файлы чем gzip (имеются в виду конкретные программы)

Реальные замеры сжатия статических файлов в браузере

Постановка эксперимента

В файле со сравнением gzip и brotli, я сжимал самый большой файл проекта Dating App. 765кб -> 193кб.

Сделаем так: загружаем файл main.js, далее выполняем запрос (fetch). Будем измерять время отправления запроса.

```
<body>
  <script src="http://localhost:88/main.js" />
  <script>
    fetch("http://localhost:88/test");
```

```
    </script>
  </body>
```

Как происходит загрузка сайта в браузере:

1. Загружаем html документ
2. Парсим html документ
3. Встречаем `<script src="http://localhost:88/main.js" />`, что говорит нам: загрузи и исполни скрипт с адреса "http://localhost:88/main.js", дальше не пойдём, пока этот скрипт не загрузится. В это время как раз входит время, необходимое на скачивание декодирование. Скрипт я закомментировал, чтобы испортить измерения сторонние вычисления.
4. Дальше выполняем запрос на "http://localhost:88/test". Это служит в качестве маркера. Время отправки запроса мы и будем измерять.

Буду проводить замеры:

Network throttling, CPU throttling

- No Throttling, No Throttling (Неограниченная скорость интеренета, мощный процессор)
- fast 4g, No Throttling (Быстрый интернет, мощный процессор)
- No Throttling, 6x Throttling

Сравню найду время отправки запроса для сжатия brottli 11x и для файла без сжатия. В теории для 3-го случая может получится, что без сжатия будет быстрее, так как не потребуется тратить время на декодировние.

Но т.к есть подозрения, что CPU throttling не влияет на время декодирования, проведу замеры на своём мобильном телефоне Xiaomi 9T PRO в локальной сети Wi-Fi (300 Mb/s).

Так же дополнительно отключил кеширование на стороне клиента с помощью заголовков "Cache-Control", "Pragma"

Как и в прошлый раз ограничиваю фоновые процессы. 20 замеров на каждую характеристику

Время test запроса в случае с телефоном измерялось на стороне сервера.

Результаты находятся в файле Измерения2.xlsx

Вывод

Результаты измерений показали, что для статических файлов во всех случаях сжатия файлов либо существенно уменьшает время отклика?, либо не увеличивает его. То есть для любых устройств сжатие статических файлов будет предпочтительно.

Нагрузочное тестирование REST запросов (динамические данные)

Введение

В этом эксперименте созданы максимально правдободоные условия. Сервер находится на удалённом сервере в Нидерландах

Технические характеристики сервера:

- CPU 1 vCPU
- RAM 2 GB
- Storage 20 GB
- Speed 1200 Mbps

Технические характеристика моего ноутбука:

- CPU 4 ядра
- RAM 8 GB
- Speed 50 Mbps

В качестве REST сервера используется Node.js express.js, обратный прокси сервер (сжатие http) Nginx

Логика следующая:

- клиент заходит на сайт
- скачивает статические файлы
- когда выполнится js код, выполнится get запрос на получение всех видео /videos/getAll
- сервер сделает запрос в базу данных
- возвращает ответ

Тестрирование проводится с помощью k6. Замеряются такие параметры, как:

- CPU usage
- Memory Usage
- Latency
- Availability

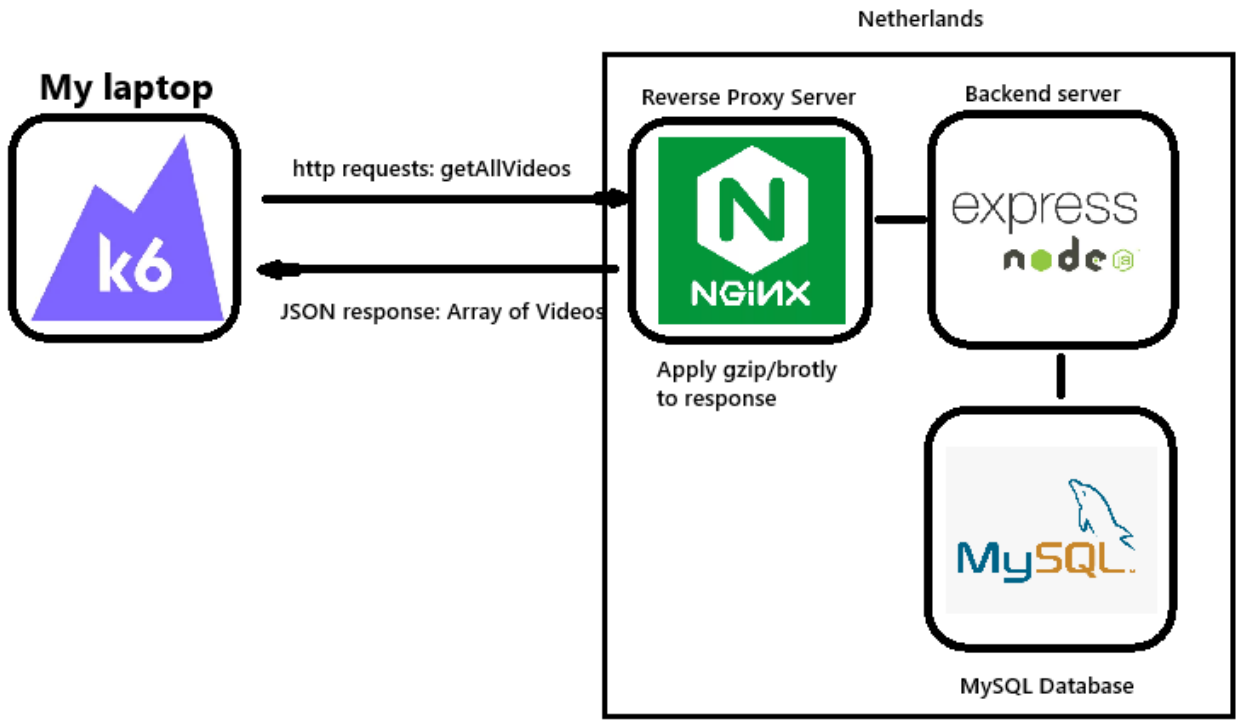
В зависимости от количества запросов в секунду (RPS)

Так выглядит ответ от сервера:

```
[
  {
    title: "Steel Horizon",
    description:
      '"Steel Horizon" is a captivating cinematic journey that explores the
boundaries of imagination and reality. With stunning visuals and a compelling
narrative, it draws viewers into a richly woven tale full of emotion, suspense,
and intrigue. As the characters navigate through complex challenges, deep personal
struggles, and unexpected twists, the story unfolds with intensity and grace.
Crafted by visionary creators, the film blends elements of classic storytelling
with modern cinematic techniques to create an unforgettable experience. Whether
you\'re drawn to heartfelt drama, thrilling action, or thought-provoking ideas,
this film offers a powerful reflection on humanity, resilience, and discovery.',
    number: 19,
    src_url: "https://cdn.example.com/videos/video_19.mp4",
    preview_url: "/previews/bearwolf.mp4",
    image_url: "/images/bearwolf2.jpg",
    studios: ["MegaPix"],
```

```
    tags: ["documentary", "action", "comedy"],
  },
]
```

Схема проекта



Эксперимент первый

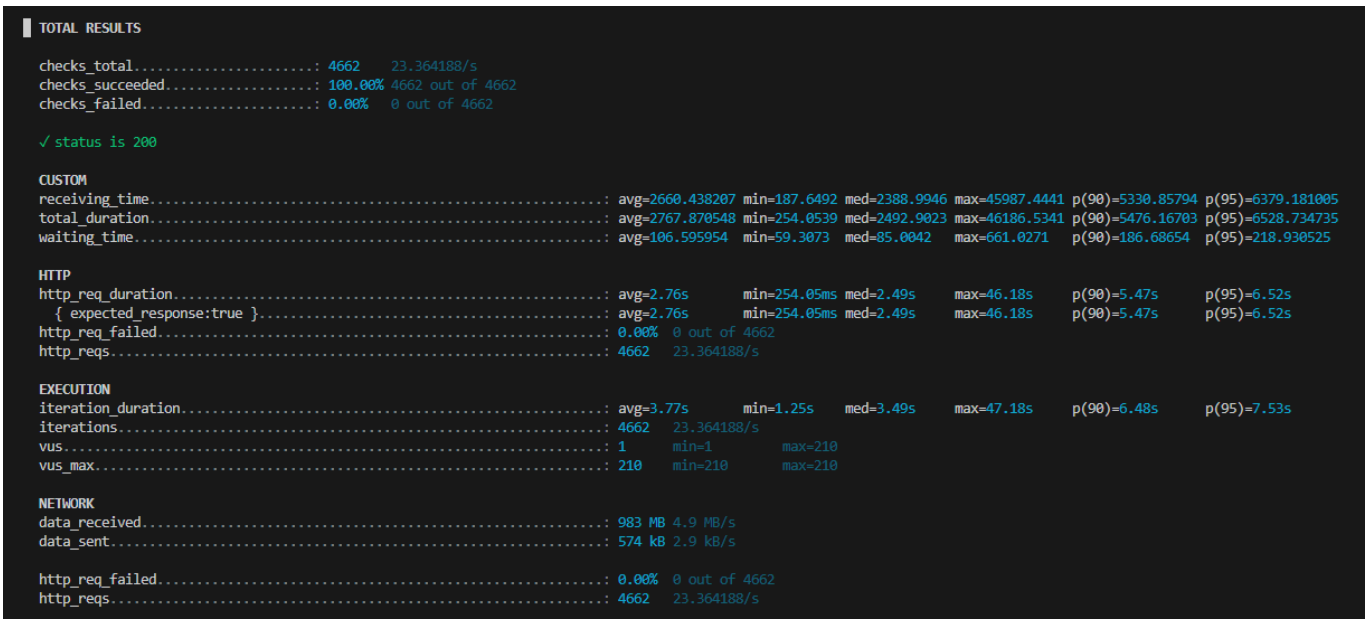
Нагрузка:

```
stages: [
  { duration: "5s", target: 10 },
  { duration: "10s", target: 20 },
  { duration: "10s", target: 30 },
  { duration: "10s", target: 40 },
  { duration: "10s", target: 50 },
  { duration: "10s", target: 60 },
  { duration: "10s", target: 70 },
  { duration: "10s", target: 80 },
  { duration: "10s", target: 90 },
  { duration: "10s", target: 100 },
  { duration: "10s", target: 110 },
  { duration: "10s", target: 120 },
  { duration: "10s", target: 130 },
  { duration: "10s", target: 140 },
  { duration: "10s", target: 150 },
  { duration: "10s", target: 170 },
  { duration: "10s", target: 190 },
  { duration: "10s", target: 210 },
```

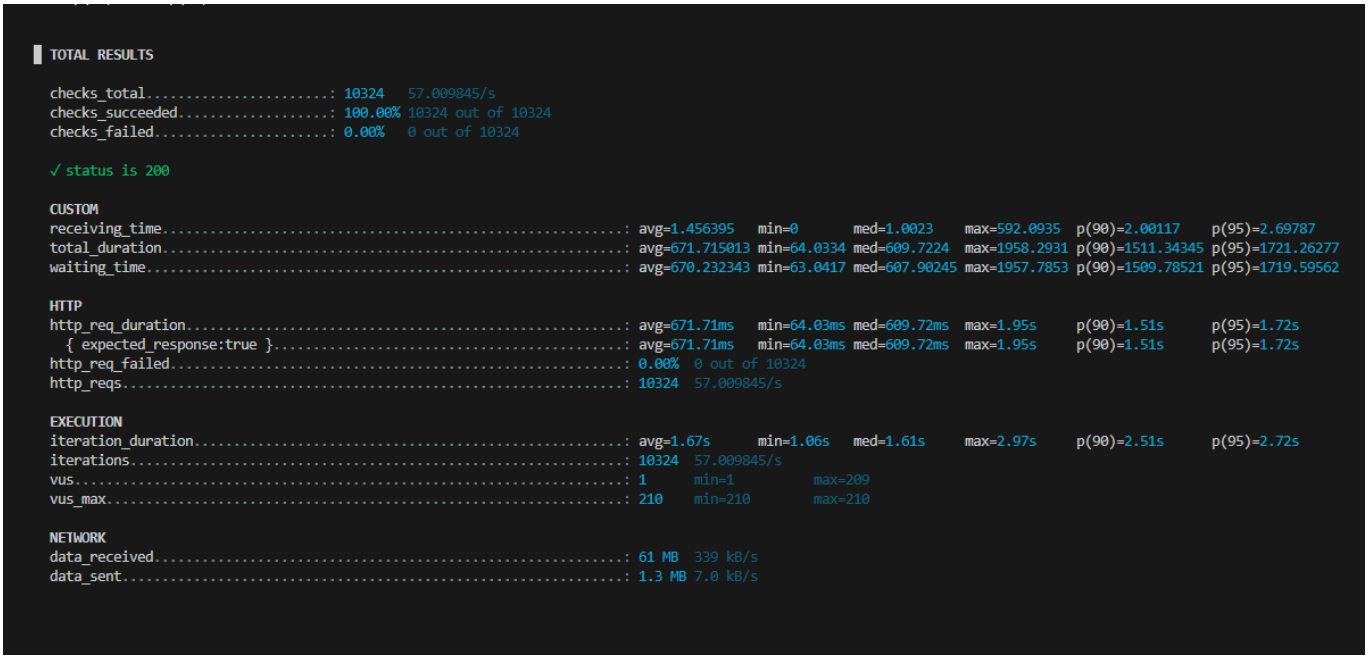
```
    { duration: "5s", target: 5 },
  ]
```

duration - время каждой стадии, target - кол-во одновременно активных пользователей Каждый пользователь отправляет запрос на сервер, ждёт ответа, далее ждёт 1 секунду.

Результаты без сжатия:



Результаты с сжатием (gzip 9):



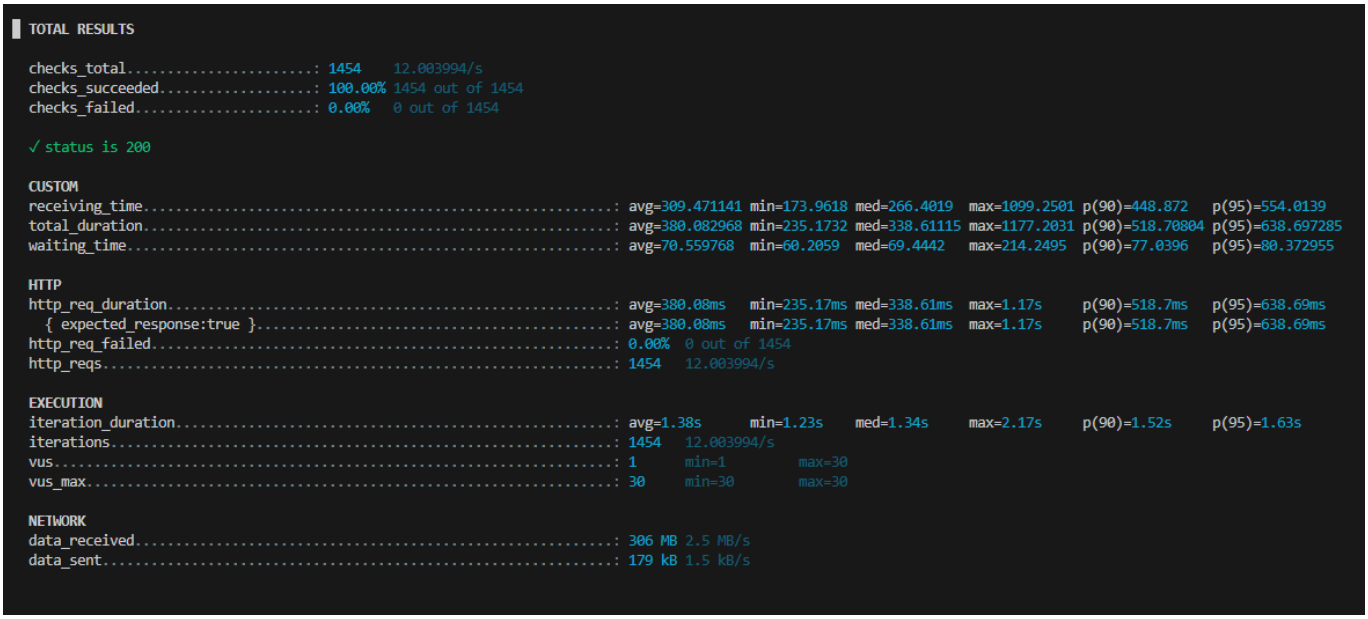
Эксперимент второй

Нагрузка:

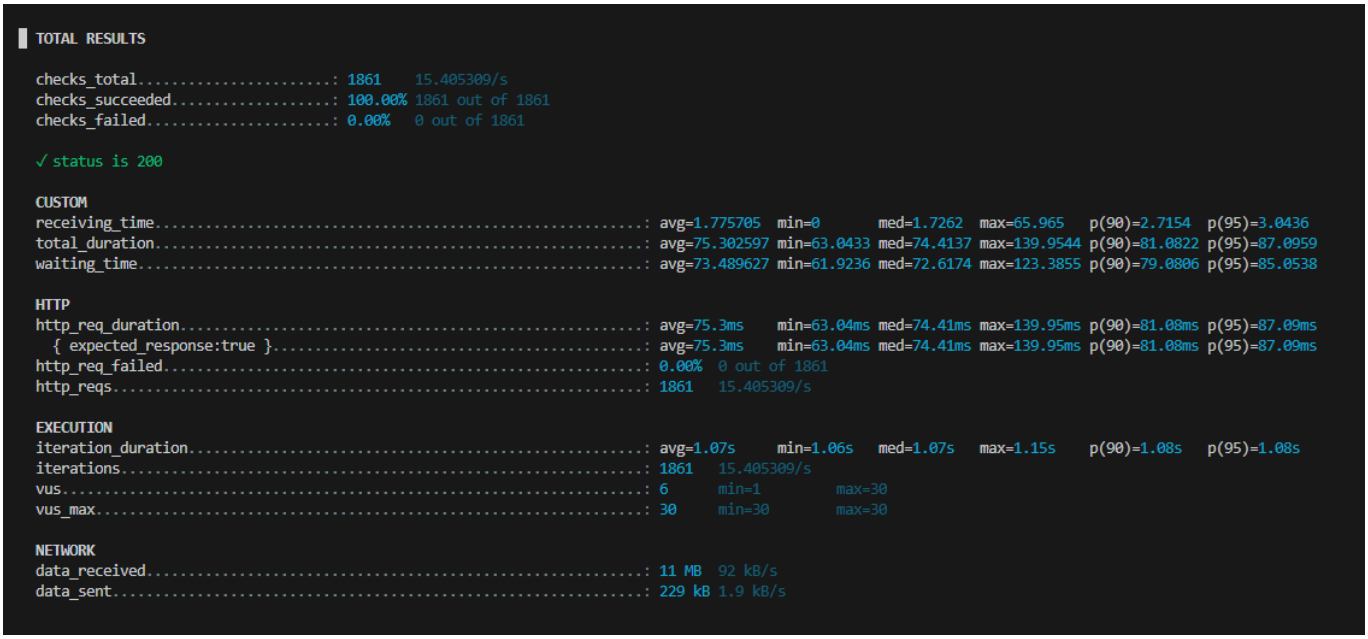
```
  stages: [
    { duration: "10s", target: 10 },
```

```
    { duration: "20s", target: 10 },
    { duration: "20s", target: 15 },
    { duration: "20s", target: 20 },
    { duration: "20s", target: 25 },
    { duration: "20s", target: 30 },
    { duration: "10s", target: 5 },
  ]
```

Результаты без сжатия:



Результаты с сжатием (gzip 9):



Зависимость CPU load, Network load от уровня сжатия (Без switcher)

Технические характеристики сервера:

- CPU 1 vCPU
- RAM 2 GB

- Storage 20 GB
- Speed 1200 Mbps

Технические характеристика моего ноутбука:

- CPU 4 ядра
- RAM 8 GB
- Speed 100 Mbps

Ограничение на скорость создано искусственно с помощью настройки роутера QoS, максимальная скорость около 200 Mb/s Ограничие скорости гарантирует постоянную полосу пропускания На сервере отключены все постороннние процессы. CPU используется только для:

- MySQL Database
- Backend server (Express.js)
- Обратный прокси сервер (Nginx)

Нагрузка создана с помощью k6:

```
stages: [
  { duration: "5s", target: 10 },
  { duration: "10s", target: 20 },
  { duration: "10s", target: 30 },
  { duration: "10s", target: 40 },
  { duration: "10s", target: 50 },
  { duration: "10s", target: 60 },
  { duration: "10s", target: 70 },
  { duration: "10s", target: 80 },
  { duration: "10s", target: 90 },
  { duration: "10s", target: 100 },
  { duration: "10s", target: 110 },
  { duration: "10s", target: 120 },
  { duration: "10s", target: 130 },
  { duration: "10s", target: 140 },
  { duration: "10s", target: 150 },
  { duration: "10s", target: 170 },
  { duration: "10s", target: 190 },
  { duration: "10s", target: 210 },
  { duration: "5s", target: 5 },
]
```

В базе данных находится 400 видео, для тестирования используется Get запрос: `/video/getRecomendations/1`, который возвращает 100 рандомных видео из базы данных

Будет варироваться степень сжатия:

- no compress
- gzip 1
- gzip 5
- gzip 9

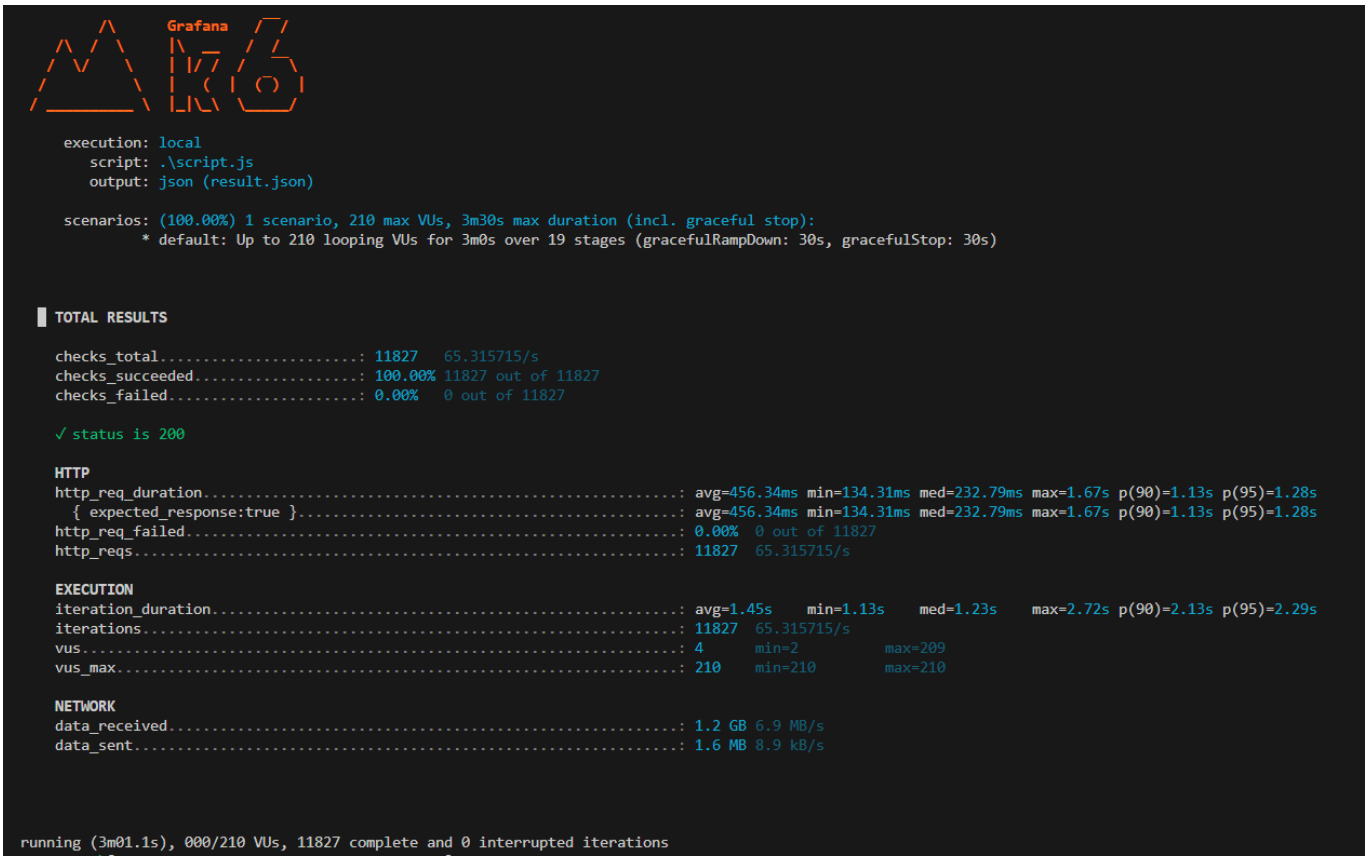
При построении графиков использовалось сглаживание по трём точкам

```
smoothing_x[i] = (x[i-1] + x[i] + x[i+1])/3
```

Построены графики от времени

Результаты измерений

No compress



Gzip level 1

```
PS D:\programing\diploma\Работа\Нагрузочное тестирование\stress_testing> k6 run .\script.js --out json=result.json

     /\      Grafana   /‾‾/
  /\  /  \     |\  __   /  /
 /  \/    \    | |/ /  /   ‾‾\
/          \   |   (  |  (‾)  |
/ _________ \  |_|\_\  \_____/

     execution: local
        script: .\script.js
        output: json (result.json)

     scenarios: (100.00%) 1 scenario, 210 max VUs, 3m30s max duration (incl. graceful stop):
              * default: Up to 210 looping VUs for 3m0s over 19 stages (gracefulRampDown: 30s, gracefulStop: 30s)


  TOTAL RESULTS

    checks_total.....................: 12745   70.397576/s
    checks_succeeded.................: 100.00% 12745 out of 12745
    checks_failed....................: 0.00%   0 out of 12745

    ✓ status is 200

    HTTP
    http_req_duration..............................................: avg=350.55ms min=47.59ms med=146.16ms max=1.27s p(90)=994.46ms p(95)=1.07s
      { expected_response:true }...................................: avg=350.55ms min=47.59ms med=146.16ms max=1.27s p(90)=994.46ms p(95)=1.07s
    http_req_failed................................................: 0.00%  0 out of 12745
    http_reqs......................................................: 12745  70.397576/s

    EXECUTION
    iteration_duration.............................................: avg=1.35s    min=1.04s   med=1.15s    max=2.31s p(90)=1.99s    p(95)=2.07s
    iterations.....................................................: 12745  70.397576/s
    vus............................................................: 1      min=1          max=209
    vus_max........................................................: 210    min=210        max=210

    NETWORK
    data_received..................................................: 57 MB  316 kB/s
    data_sent......................................................: 1.7 MB 9.6 kB/s
```

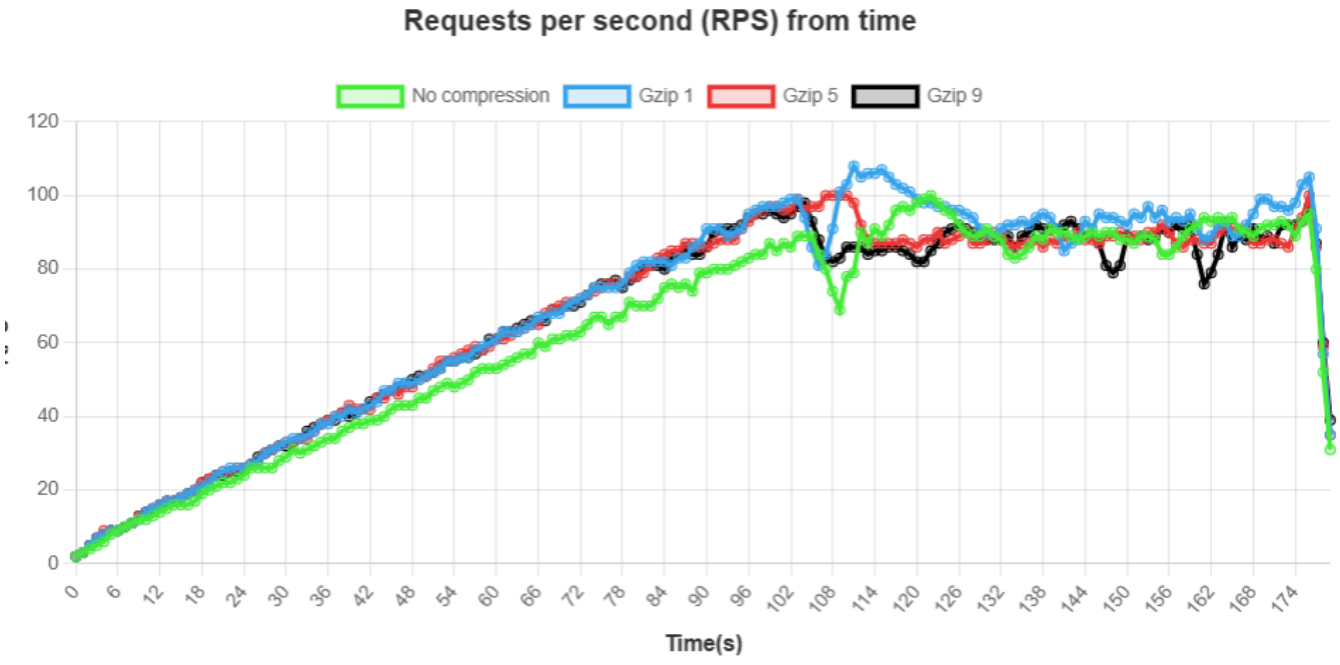
Gzip level 5

```
PS D:\programing\diploma\Работа\Нагрузочное тестирование\stress_testing> k6 run .\script.js --out json=result.json

     /\      Grafana   /‾‾/
  /\  /  \     |\  __   /  /
 /  \/    \    | |/ /  /   ‾‾\
/          \   |   (  |  (‾)  |
/ _________ \  |_|\_\  \_____/

     execution: local
        script: .\script.js
        output: json (result.json)

     scenarios: (100.00%) 1 scenario, 210 max VUs, 3m30s max duration (incl. graceful stop):
              * default: Up to 210 looping VUs for 3m0s over 19 stages (gracefulRampDown: 30s, gracefulStop: 30s)


  TOTAL RESULTS

    checks_total.....................: 12338   68.171548/s
    checks_succeeded.................: 100.00% 12338 out of 12338
    checks_failed....................: 0.00%   0 out of 12338

    ✓ status is 200

    HTTP
    http_req_duration..............................................: avg=396.7ms min=47.79ms med=157ms max=1.57s p(90)=1.05s p(95)=1.21s
      { expected_response:true }...................................: avg=396.7ms min=47.79ms med=157ms max=1.57s p(90)=1.05s p(95)=1.21s
    http_req_failed................................................: 0.00%  0 out of 12338
    http_reqs......................................................: 12338  68.171548/s

    EXECUTION
    iteration_duration.............................................: avg=1.39s   min=1.04s   med=1.17s max=2.57s p(90)=2.06s p(95)=2.21s
    iterations.....................................................: 12338  68.171548/s
    vus............................................................: 2      min=2          max=209
    vus_max........................................................: 210    min=210        max=210

    NETWORK
    data_received..................................................: 48 MB  266 kB/s
    data_sent......................................................: 1.7 MB 9.3 kB/s
```

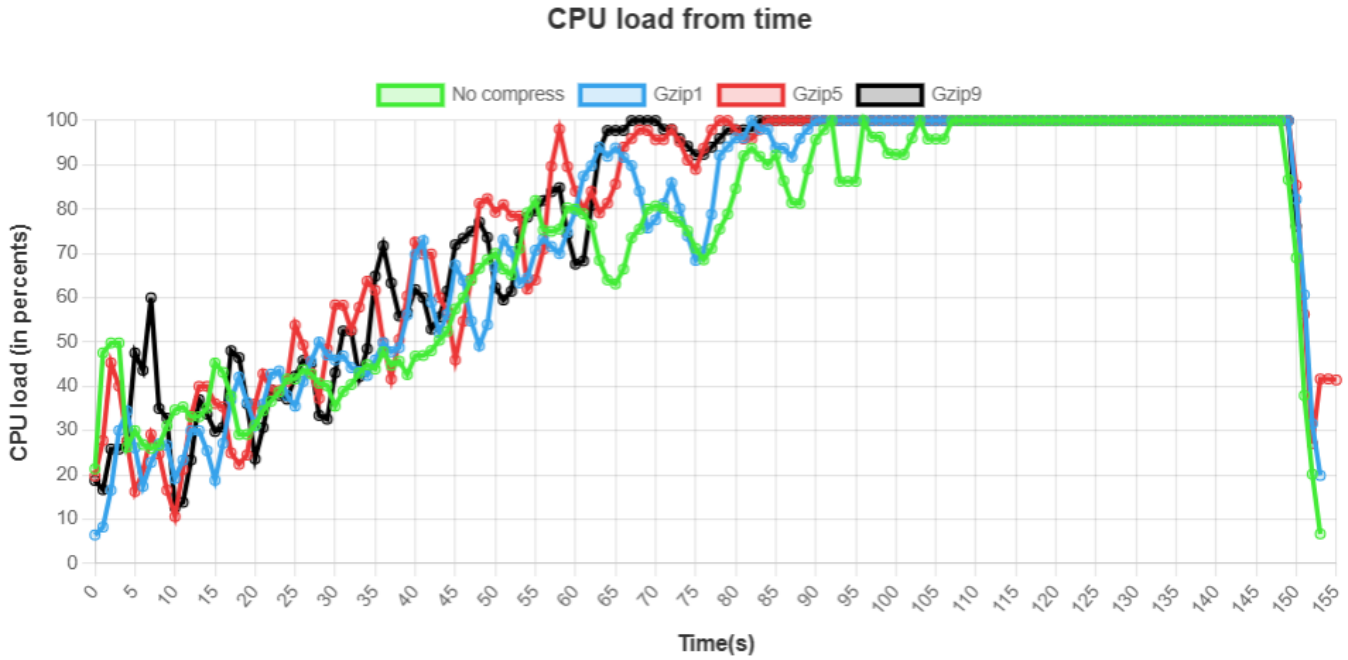
Gzip level 9

```
PS D:\programing\diploma\Работа\Нагрузочное тестирование\stress_testing> k6 run .\script.js --out json=result.json

     /\      Grafana   /‾‾/
  /\  /  \     |\  __   /  /
 /  \/    \    | |/ /  /   ‾‾\
/          \   |   (  |  (‾)  |
/ __________ \  |_|\_\  \_____/

     execution: local
        script: .\script.js
        output: json (result.json)

     scenarios: (100.00%) 1 scenario, 210 max VUs, 3m30s max duration (incl. graceful stop):
              * default: Up to 210 looping VUs for 3m0s over 19 stages (gracefulRampDown: 30s, gracefulStop: 30s)


  TOTAL RESULTS

    checks_total.....................: 12214   67.489938/s
    checks_succeeded.................: 100.00% 12214 out of 12214
    checks_failed....................: 0.00%   0 out of 12214

    ✓ status is 200

    HTTP
    http_req_duration..............................................: avg=411.31ms min=44.38ms med=338.54ms max=1.52s p(90)=1.12s p(95)=1.23s
      { expected_response:true }...................................: avg=411.31ms min=44.38ms med=338.54ms max=1.52s p(90)=1.12s p(95)=1.23s
    http_req_failed................................................: 0.00%  0 out of 12214
    http_reqs......................................................: 12214  67.489938/s

    EXECUTION
    iteration_duration.............................................: avg=1.41s    min=1.04s    med=1.33s    max=2.56s p(90)=2.12s p(95)=2.23s
    iterations.....................................................: 12214  67.489938/s
    vus............................................................: 1      min=1          max=209
    vus_max........................................................: 210    min=210        max=210

    NETWORK
    data_received..................................................: 45 MB  248 kB/s
    data_sent......................................................: 1.7 MB 9.2 kB/s



running (3m01.0s), 000/210 VUs, 12214 complete and 0 interrupted iterations
```

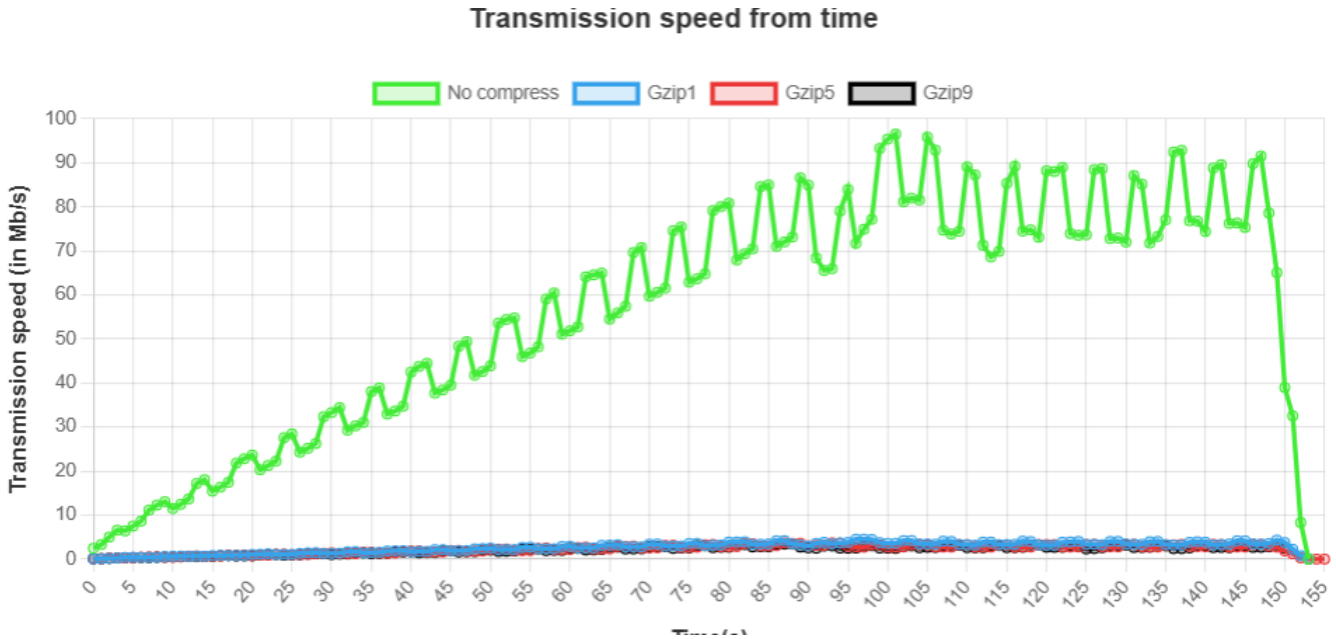
Requests per seconds

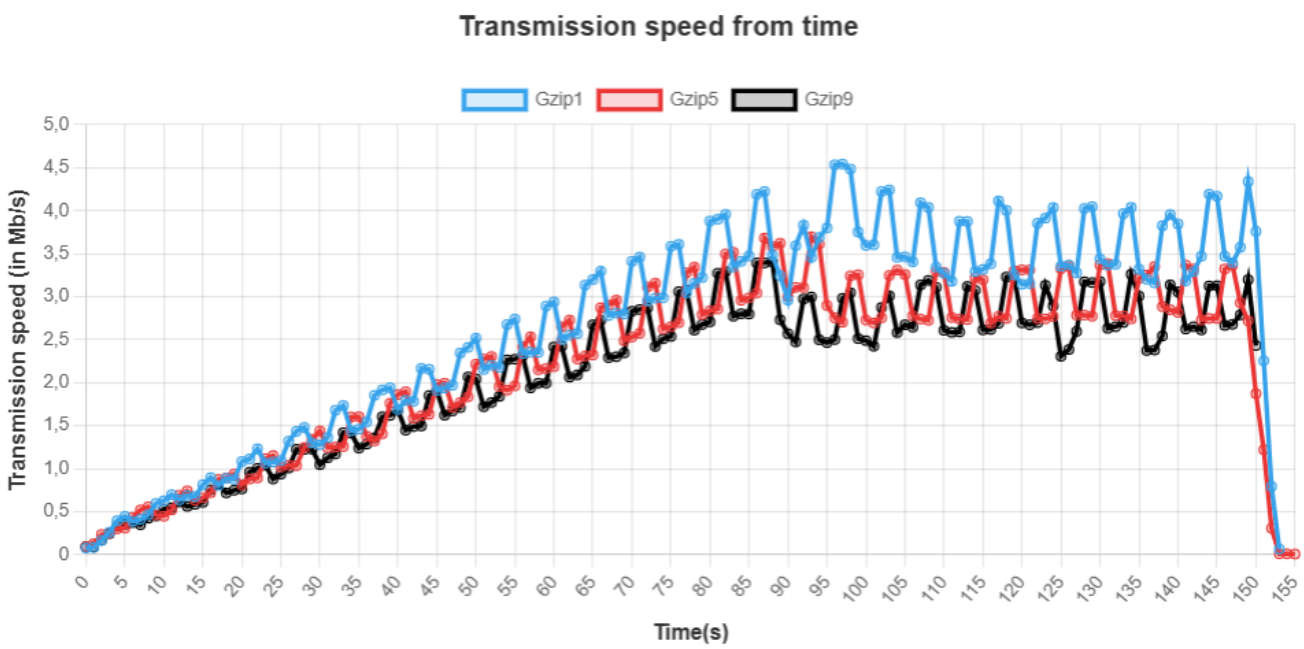


Cpu load



Transmission speed





Анализ полученных результатов

Как видно, использовние gzip заментно снизило нагрузку на сеть примерно в 20-30 раз, при этом даже при 80 МБ/с несжатого трафика, нагрузка на CPU из-за сжатия оказалась крайне малой, что при загрузке CPU на 100%, показатель RPS остался на том же уровне.

4. Вывод (практически показать, что реально сделано)

Приложение

Типы файлов в браузере

Каждый раз, когда браузер получает файл с веб-сервера, сервер добавляет заголовок “Content Type” этому файлу. Ваш браузер соотносит “Content Type” со списком его собственных “MINE Types”, чтобы понять, что дальше делать с этим конкретным файлом. Например, чтобы отобразить PNG изображение на сайте, файл с изображением должен иметь “Content Type” “image/png”, когда браузер его скачивает. Это даёт понимание браузеру, что он получил необходимый файл и может попробовать отобразить его на экране. MIME расшифровывается как "Multipurpose Internet Mail Extensions". Концепция MINE Types” была изначально придумана для использования в электронной почте, поэтому “Mail” присутствует в аббревиатуре. Сейчас “Mine Types” так же используются в веб-приложениях. Браузеры поддерживают широкий спектр MIME-типов. Вот некоторые из наиболее распространенных MIME-типов, которые поддерживаются современными браузерами:

Текстовые данные

- text/html: HTML-документы
- text/css: Таблицы стилей CSS
- text/javascript или application/javascript: JavaScript-файлы
- text/plain: Обычные текстовые файлы

Изображения

- image/jpeg: JPEG-изображения

- image/png: PNG-изображения
- image/gif: GIF-изображения
- image/webp: WebP-изображения
- image/avif: AVIF - изображения
- image/svg+xml: SVG векторная графика

Аудио и видео

- audio/mpeg: MP3 аудио
- audio/ogg: OGG аудио
- audio/wav: WAV аудио
- video/mp4: MP4 видео
- video/webm: WebM видео
- video/ogg: Ogg видео

Документы

- application/pdf: PDF файлы
- application/xml: XML файлы
- application/json: JSON данные

Приложения

- application/octet-stream: Для необработанных двоичных файлов
- application/zip: ZIP архивы

Шрифты

- font/woff: WOFF шрифты
- font/woff2: WOFF2 шрифты
- application/vnd.ms-fontobject: EOT шрифты
- font/ttf: TTF шрифты

Я буду работать со статическими файлами следующих типов:

Текстовые данные

- text/html
- text/css
- text/javascript Изображения
- image/jpeg
- image/png
- image/webp
- image/avif