Compression is one of the most important tools CloudFlare has to accelerate website performance. Compressed content takes less time to transfer, and consequently reduces load times. On expensive mobile data plans, compression even saves money for consumers. However, compression is not free—it comes at a price. It is one of the most compute expensive operations our servers perform, and the better the compression rate we want, the more effort we have to spend.

The most popular compression format on the web is gzip. We put a great deal of effort into improving the performance of the gzip compression, so we can perform compression on the fly with fewer CPU cycles. Recently a potential replacement for gzip, called Brotli, was announced by Google. Being early adopters for many technologies, we at CloudFlare want to see for ourselves if it is as good as claimed.

This post takes a look at a bit of history behind gzip and Brotli, followed by a performance comparison.

**Compression 101**

Many popular lossless compression algorithms rely on LZ77 and Huffman coding, so it’s important to have a basic understanding of these two techniques before getting into gzip or Brotli.

**LZ77**

LZ77 is a simple technique developed by Abraham Lempel and Jacob Ziv in 1977 (hence the original name). Let's call the input to the algorithm a string (a sequence of bytes, not necessarily letters) and each consecutive sequence of bytes in the input a substring. LZ77 compresses the input string by replacing some of its substrings by pointers (or backreferences) to an identical substring previously encountered in the input.

The pointer usually has the form of <length, distance>, where length indicates the number of identical bytes found, and distance indicates how many bytes separate the current occurrence of the substring from the previous one. For example the string abcdeabcdf can be compressed with LZ77 to abcde<4,5>f and aaaaaaaaaa can be compressed to simply a<9, 1>. The decompressor when encountering a backreference will simply copy the required number of bytes from the already decompressed output, which makes it very fast for decompression. This is a nice illustration of LZ77 from my previous blog [Improving compression with a preset DEFLATE dictionary](https://blog.cloudflare.com/improving-compression-with-preset-deflate-dictionary/):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| L | i | t | t | l | e |  | b | u | n | n | y |  | F | o | o |  | F | o | o |
|  | W | e | n | t |  | h | o | p | p | i | n | g |  | t | h | r | o | u | g |
| h |  | t | h | e |  | f | o | r | e | s | t |  | S | c | o | o | p | i | n |
| g |  | u | p |  | t | h | e |  | f | i | e | l | d |  | m | i | c | e |  |
| A | n | d |  | b | o | p | p | i | n | g |  | t | h | e | m |  | o | n |  |
| t | h | e |  | h | e | a | d |  | D | o | w | n |  | c | a | m | e |  | t |
| h | e |  | G | o | o | d |  | F | a | i | r | y | , |  | a | n | d |  | s |
| h | e |  | s | a | i | d |  | " | L | i | t | t | l | e |  | b | u | n | n |
| y |  | F | o | o |  | F | o | o |  | I |  | d | o | n | ' | t |  | w | a |
| n | t |  | t | o |  | s | e | e |  | y | o | u |  | S | c | o | o | p | i |
| n | g |  | u | p |  | t | h | e |  | f | i | e | l | d |  | m | i | c | e |
|  | A | n | d |  | b | o | p | p | i | n | g |  | t | h | e | m |  | o | n |
|  | t | h | e |  | h | e | a | d | . | " |  |  |  |  |  |  |  |  |  |

Output (length tokens are blue, distance tokens are red):

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| L | i | t | t | l | e |  | b | u | n | n | y |  | F | o | o | 5 | 4 | W | e |
| n | t |  | h | o | p | p | i | n | g |  | t | h | r | o | u | g | h | 3 | 8 |
| e |  | f | o | r | e | s | t |  | S | c | o | o | 5 | 28 | u | p | 6 | 23 | i |
| e | l | d |  | m | i | c | e |  | A | n | d |  | b | 9 | 58 | e | m |  | o |
| n | 5 | 35 | h | e | a | d |  | D | o | w | n |  | c | a | m | e | 5 | 19 | G |
| o | o | d |  | F | a | i | r | y | , |  | a | 3 | 55 | s | 3 | 20 | s | a | i |
| d |  | " | L | 20 | 149 | I |  | d | o | n | ' | t |  | w | a | 3 | 157 | t | o |
|  | s | e | e |  | y | o | u | 56 | 141 | . | " |  |  |  |  |  |  |  |  |

The deflate algorithm managed to reduce the original text from 251 characters, to just 152 tokens! Those tokens are later compressed further by Huffman coding.

**Huffman Coding**

Huffman coding is another lossless compression algorithm. Developed by David Huffman back in the 50s, it is used for many compression algorithms, including JPEG. A Huffman code is a type of prefix coding, where, given an alphabet and an input, frequently occurring characters are replaced by shorter bit sequences and rarely occurring characters are replaced with longer sequences.

The code can be expressed as a binary tree, where the leaf nodes are the literals of the alphabet and the two edges from each node are marked with 0 and 1. To decode the next character, the decompressor can parse the tree from the root until it encounters a literal in the tree.

Each compression format uses Huffman coding differently, and for our little example we will create a Huffman code for an alphabet that includes only the literals and the length codes we actually used. To start, we must count the frequency of each letter in the LZ77 compressed text:

(space) - 19, o -14, e - 11, n - 8, t - 7, a - 6, d - 6, i - 6, 3 - 4, 5 - 4, h - 4, s - 4, u - 4, c - 3, m - 3, p - 3, r - 3, y - 3, (") - 2, F - 2, L - 2, b - 2, g - 2, l - 2, w - 2, (') - 1, (,) - 1, (.) - 1, A - 1, D - 1, G - 1, I - 1, 9 - 1, 20 - 1, S - 1, 56 - 1, W - 1, 6 - 1, f - 1.

We can then use the algorithm from [Wikipedia](https://en.wikipedia.org/wiki/Huffman_coding) to build this Huffman code (binary):

(space) - 101, o - 000, e - 1101, n - 0101, t - 0011, a - 11101, d - 11110, i - 11100, 3 - 01100, 5 - 01111, h - 10001, s - 11000, u - 10010, c - 00100, m - 111111, p - 110011, r - 110010, y - 111110, (") - 011010, F - 010000, L - 100000, b - 011101, g - 010011, l - 011100, w - 011011, (') - 0100101, (,) - 1001110, (.) - 1001100, A - 0100011, D - 0100010, G - 1000011, I - 0010110, 9 - 1000010, 20 - 0010111, S - 0010100, 56 - 0100100, W - 0010101, 6 - 1001101, f - 1001111.

Here the most frequent letters - space and 'o' got the shortest codes, only 3 bit long, whereas the letters that occur only once get 7 bit codes. If we were to represent the alphabet of 256 bytes and some length tokens, we would require 9 bits per every letter.

Now apply the code to the LZ77 output, (leaving the distance tokens untouched) and we get:

100000 11100 0011 0011 011100 1101 101 011101 10010 0101 0101 111110 101 010000 000 000 01111 4 0010101 1101 0101 0011 101 10001 000 110011 110011 11100 0101 010011 101 0011 10001 110010 000 10010 010011 01100 8 10001 1101 101 1001111 000 110010 1101 11000 0011 101 0010100 00100 000 000 01111 28 10010 110011 1001101 2311100 1101 011100 11110 101 111111 11100 00100 1101 101 0100011 0101 11110 101 011101 1000010 58 1101 111111 101 000 0101 01111 35 10001 1101 11101 11110 101 0100010 000 011011 0101 101 00100 11101 111111 1101 01111 19 1000011 000 000 11110 101 010000 11101 11100 110010 111110 1001110 101 11101 01100 55 11000 01100 2011000 11101 11100 11110 101 011010 100000 0010111 149 0010110 101 11110 000 0101 0100101 0011 101 011011 11101 01100 157 0011 000 101 11000 1101 1101 101 111110 000 10010 0100100 141 1001100 011010

Assuming all the distance tokens require one byte to store, the total output length is 898 bits. Compared to the 1356 bits we would require to store the LZ77 output, and 2008 bits for the original input. We achieved a compression ratio of 31% which is very good. In reality this is not the case, since we must also encode the Huffman tree we used, otherwise it would be impossible to decompress the text.

**gzip**

The compression algorithm used in gzip is called "deflate," and it’s a combination of the LZ77 and Huffman algorithms discussed above.

On the LZ77 side, gzip has a minimum size of 3 bytes and a maximum of 258 bytes for the length tokens and a maximum of 32768 bytes for the distance tokens. The maximum distance also defines the sliding window size the implementation uses for compression and decompression.

For Huffman coding, deflate has two alphabets. The first alphabet includes the input literals ("letters" 0-255), the end-of-block symbol (the "letter" 256) and the length tokens ("letters" 257-285). There are only 29 letters to encode all the possible lengths and the tokens 265-284 will always have additional bits encoded in the stream to cover the entire range from 3 to 258. For example the letter 257 indicates the minimal length of 3, whereas the letter 265 will indicate the length 11 if followed by the bit 0, and the length 12 if followed by the bit 1.

The second alphabet is for the distance tokens only. Its letters are the codewords 0 through 29. Similar to the length tokens, the codes 4-29 are followed by 1 to 13 additional bits to cover the whole range from 1 to 32768.

Using two distinct alphabets makes a lot of sense, because it allows us to represent the distance code with fewer bits while avoiding any ambiguity, since the distance codes always follow the length codes.

The most common implementation of gzip compression is the zlib library. At CloudFlare, we use a custom version of zlib [optimized for our servers](https://blog.cloudflare.com/cloudflare-fights-cancer/).

zlib has 9 preset quality settings for the deflate algorithm, labeled from 1 to 9. It can also be run with quality set to 0, in which case it does not perform any compression. These quality settings can be divided into two categories:

* Fast compression (levels 1-3): When a sufficiently long backreference is found, it is emitted immediately, and the search moves to the position at the end of the matched string. If the match is longer than a few bytes, the entire matched string will not be hashed, meaning its substrings will never be referenced in the future. Clearly, this reduces the compression ratio.
* Slow compression (levels 4-9): Here, *every* substring is hashed; therefore, any substring can be referenced in the future. In addition slow compression enables "lazy" matches. If at a given position a sufficiently long match is found, it will not be immediately emitted. Instead, the algorithm attempts to find a match at the next position. If a longer match is found, the algorithm will emit a single literal at the current position instead the shorter match, and continue to the next position. As a rule, this results in a better compression ratio than levels 1-3.

Other differences between the quality levels are how far to search for a backreference and how long a match should be before stopping.

A very decent compression rate can be observed at levels 3-4, which are also quite fast. Increasing compression quality from level 4 and up gives incrementally smaller compression gains, while requiring substantially more time. Often, levels 8 and 9 will produce similarly compressed output, but level 9 will require more time to do it.

It is important to understand that the zlib implementation does not guarantee the best compression possible with the format, even when the quality setting is set to 9. Instead, it uses a set of heuristics and optimizations that allow for very good compression at reasonable speed.

Better gzip compression can be achieved by the [zopfli](https://github.com/google/zopfli) library, but it is significantly slower than zlib.

**Brotli**

The Brotli format was developed by Google and has been refined for a while now. Here at CloudFlare, we built an nginx module that performs dynamic Brotli compression, and we deployed it on our [test server](https://http2.cloudflare.com/) that supports HTTP2 and other new features.

To check the Brotli compression on the test server you can use the nightly build of the Firefox browser, which also supports this format.

**Brotli, deflate, and gzip**

Brotli and deflate are very closely related. Brotli also uses the LZ77 and Huffman algorithms for compression. Both algorithms use a sliding window for backreferences. Gzip uses a fixed size, 32KB window, and Brotli can use any window size from 1KB to 16MB, in powers of 2 (minus 16 bytes). This means that the Brotli window can be up to 512 times larger window than the deflate window. This difference is almost irrelevant in web-server context, as text files larger than 32KB are the minority.

Other differences include smaller minimal match length (2 bytes minimum in Brotli, compared to 3 bytes minimum in deflate) and larger maximal match length (16779333 bytes in Broli, compared to 258 bytes in deflate).

**Static dictionary**

Brotli also features a static dictionary. The "dictionary" supported by deflate can greatly improve compression, but has to be supplied independently and can only be addressed as part of the sliding window. The Brotli dictionary is part of the implementation and can be referenced from anywhere in the stream, somewhat increasing its efficiency for larger files. Moreover different transformations can be applied to words of the dictionary effectively increasing its size.

**Context modeling**

Brotli also supports something called context modeling. Context modeling is a feature that allows multiple Huffman trees for the same alphabet in the same block. For example, in deflate each block consists of a series of literals (bytes that could not be compressed by backreferencing) and <length, distance> pairs that define a backreference for copying. Literals and lengths form a single alphabet, while the distances are a different alphabet.

In Brotli, each block is composed of "commands". A command consists of 3 parts. The first part of each command is a word <insert, copy>. "Insert" defines the number of literals that will follow the word, and it may have the value of 0. "Copy" defines the number of literals to copy from a back reference. The word <insert,copy> is followed by a sequence of "insert" literals: <lit> … <lit>. Finally the command ends with a <distance>. Distance defines the backreference from which to copy the previously defined number of bytes. Unlike the distance in deflate, Brotli distance can have additional meanings, such as references to the static dictionary, or references to recently used distances.

There are three alphabets here. One is for <lit>s and it simply covers all the possible byte values from 0 to 255. The other one is for <distance>s, and its size depends on the size of the sliding window and other parameters. The third alphabet is for the <insert,copy>length pairs, with 704 letters. Here <insert, copy> indicates a single letter in the alphabet, as opposed to <length,distance> pairs in deflate where length and distance are letters in distinct alphabets.

So why do we care about context modeling? It means that for any of the alphabets—up to 256 different Huffman trees—can be used in the same block. The switch between different trees is determined by "context". This can be useful when the compressed file consists of different types of characters. For example binary data interleaved with UTF-8 strings, or a multilingual dictionary.

Whereas the basic idea behind Brotli remains identical to that of deflate, the way the data is encoded is very different. Those improvements allow for significantly better compression, but they also require a significant amount of processing. To alleviate the performance cost somewhat, Brotli drops the error detection CRC check present in gzip.

**Benchmarking**

The heaviest operation in both deflate and Brotli is the search for backward references. A higher level in zlib generally means that the backward reference search will attempt to find a better match for longer, but not necessarily succeed. This leads to significant increase in processing time. In contrast, Brotli trades longer searches for lighter operations that give better ROI, such as context modeling, dictionary references and more efficient data encoding in general. For that reason Brotli is, in theory, capable of outperforming zlib at some stage for similar compression rates, as well as giving better compression at its maximal levels.

We decided to put those theories to the test. At CloudFlare, one of the primary use cases for Brotli/gzip is on-the-fly compression of textual web assets like HTML, CSS, and JavaScript, so that’s what we’ll be testing.

There is a tradeoff between compression speed and transfer speed. It is only beneficial to increase your compression ratio if you can reduce the number of bytes you have to transfer faster than you would actually transfer them. Slower compression will actually slow the connection down. CloudFlare currently uses our own zlib implementation with quality set to 8, and that is our benchmarking baseline.

The benchmark set consists of 10,655 HTML, CSS, and JavaScript files. The benchmarks were performed on an Intel 3.5GHz, E3-1241 v3 CPU.

The files were grouped into several size groups, since different websites have different size characteristics, and we must optimize for them all.

Although the quality setting in the Brotli implementation states the possible values are 0 to 11, we didn't see any differences between 0 and 1, or between 10 and 11, therefore only quality settings 1 to 10 are reported.

**Compression quality**

Compression quality is measured as (total size of all files after compression)/(total size of all files before compression)\*100%. The columns represent the size distribution of the HTML, CSS, and JavaScript files used in the test in bytes (the file size ranges are shown using the [x,y) notation).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | [20, 1024) | [1024, 2048) | [2048, 3072) | [3072, 4096) | [4096, 8192) | [8192, 16384) | [16384, 32768) | [32768, 65536) | [65536, +∞) | All files |
| zlib 1 | 65.0% | 46.8% | 42.8% | 38.7% | 34.4% | 32.0% | 29.9% | 31.1% | 31.4% | 31.5% |
| zlib 2 | 64.9% | 46.5% | 42.5% | 38.3% | 33.8% | 31.4% | 29.2% | 30.3% | 30.5% | 30.6% |
| zlib 3 | 64.9% | 46.4% | 42.3% | 38.0% | 33.6% | 31.1% | 28.8% | 29.8% | 29.9% | 30.1% |
| zlib 4 | 64.5% | 45.8% | 41.6% | 37.1% | 32.6% | 30.0% | 27.8% | 28.5% | 28.7% | 28.9% |
| zlib 5 | 64.5% | 45.5% | 41.3% | 36.7% | 32.1% | 29.4% | 27.1% | 27.8% | 27.8% | 28.0% |
| zlib 6 | 64.4% | 45.5% | 41.3% | 36.7% | 32.0% | 29.3% | 27.0% | 27.6% | 27.6% | 27.8% |
| zlib 7 | 64.4% | 45.5% | 41.3% | 36.6% | 32.0% | 29.2% | 26.9% | 27.5% | 27.5% | 27.7% |
| zlib 8 | 64.4% | 45.5% | 41.3% | 36.6% | 32.0% | 29.2% | 26.9% | 27.5% | 27.4% | 27.7% |
| zlib 9 | 64.4% | 45.5% | 41.3% | 36.6% | 32.0% | 29.2% | 26.9% | 27.5% | 27.4% | 27.7% |
| brotli 1 | 61.3% | 46.6% | 42.7% | 38.4% | 33.8% | 30.8% | 28.6% | 29.5% | 28.6% | 29.0% |
| brotli 2 | 61.9% | 46.9% | 42.8% | 38.3% | 33.6% | 30.6% | 28.3% | 29.1% | 28.3% | 28.6% |
| brotli 3 | 61.8% | 46.8% | 42.7% | 38.2% | 33.3% | 30.4% | 28.1% | 28.9% | 28.0% | 28.4% |
| brotli 4 | 53.8% | 40.8% | 38.6% | 34.8% | 30.9% | 28.7% | 27.0% | 28.0% | 27.5% | 27.7% |
| brotli 5 | 49.9% | 37.7% | 35.7% | 32.3% | 28.7% | 26.6% | 25.2% | 26.2% | 26.0% | 26.1% |
| brotli 6 | 50.0% | 37.7% | 35.7% | 32.3% | 28.6% | 26.5% | 25.1% | 26.0% | 25.7% | 25.9% |
| brotli 7 | 50.0% | 37.6% | 35.6% | 32.3% | 28.5% | 26.4% | 25.0% | 25.9% | 25.5% | 25.7% |
| brotli 8 | 50.0% | 37.6% | 35.6% | 32.3% | 28.5% | 26.4% | 25.0% | 25.9% | 25.4% | 25.6% |
| brotli 9 | 50.0% | 37.6% | 35.5% | 32.2% | 28.5% | 26.4% | 25.0% | 25.8% | 25.3% | 25.5% |
| brotli 10 | 46.8% | 33.4% | 32.5% | 29.4% | 26.0% | 23.9% | 22.9% | 23.8% | 23.0% | 23.3% |

Clearly, the compression possible by Brotli is significant. On average, Brotli at the maximal quality setting produces 1.19X smaller results than zlib at the maximal quality. For files smaller than 1KB the result is 1.38X smaller on average, a very impressive improvement, that can probably be attributed to the use of static dictionary.

**Compression speed**

Compression speed is measured as (total size of files before compression)/(total time to compress all files) and is reported in MB/s.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | [20, 1024) | [1024, 2048) | [2048, 3072) | [3072, 4096) | [4096, 8192) | [8192, 16384) | [16384, 32768) | [32768, 65536) | [65536, +∞) | All files |
| zlib 1 | 5.9 | 21.8 | 34.4 | 43.4 | 62.1 | 89.8 | 117.9 | 127.9 | 139.6 | 125.5 |
| zlib 2 | 5.9 | 21.7 | 34.3 | 43.0 | 61.2 | 87.6 | 114.3 | 123.1 | 130.7 | 118.9 |
| zlib 3 | 5.9 | 21.7 | 34.0 | 42.4 | 60.5 | 84.8 | 108.0 | 114.5 | 114.9 | 106.9 |
| zlib 4 | 5.8 | 20.9 | 32.2 | 39.8 | 54.8 | 74.9 | 93.3 | 97.5 | 96.1 | 90.7 |
| zlib 5 | 5.8 | 20.6 | 31.4 | 38.3 | 51.6 | 68.4 | 82.0 | 81.3 | 73.2 | 71.6 |
| zlib 6 | 5.8 | 20.6 | 31.2 | 37.9 | 50.6 | 64.0 | 73.7 | 70.2 | 57.5 | 58.0 |
| zlib 7 | 5.8 | 20.5 | 31.0 | 37.4 | 49.6 | 60.8 | 67.4 | 64.6 | 51.0 | 52.0 |
| zlib 8 | 5.8 | 20.5 | 31.0 | 37.2 | 48.8 | 53.2 | 56.6 | 56.5 | 41.6 | 43.1 |
| zlib 9 | 5.8 | 20.6 | 30.8 | 37.3 | 48.6 | 51.7 | 56.6 | 54.2 | 40.4 | 41.9 |
| brotli 1 | 3.4 | 12.8 | 20.4 | 25.9 | 37.8 | 57.3 | 80.0 | 94.1 | 105.8 | 91.3 |
| brotli 2 | 3.4 | 12.4 | 19.5 | 24.4 | 35.2 | 52.3 | 71.2 | 82.0 | 89.0 | 78.8 |
| brotli 3 | 3.4 | 12.3 | 19.0 | 23.7 | 34.0 | 49.8 | 67.4 | 76.3 | 81.5 | 73.0 |
| brotli 4 | 2.0 | 7.6 | 11.9 | 15.2 | 22.2 | 33.1 | 44.7 | 51.9 | 58.5 | 51.0 |
| brotli 5 | 2.0 | 5.2 | 8.0 | 10.3 | 15.0 | 22.0 | 29.7 | 33.3 | 32.8 | 30.3 |
| brotli 6 | 1.8 | 3.8 | 5.5 | 7.0 | 10.5 | 16.3 | 23.5 | 28.6 | 28.4 | 25.6 |
| brotli 7 | 1.5 | 2.3 | 3.1 | 3.7 | 4.9 | 7.2 | 10.7 | 15.5 | 19.6 | 16.2 |
| brotli 8 | 1.4 | 2.3 | 2.7 | 3.1 | 4.0 | 5.3 | 7.1 | 10.6 | 15.1 | 12.2 |
| brotli 9 | 1.3 | 2.1 | 2.4 | 2.8 | 3.4 | 4.3 | 5.5 | 7.0 | 10.6 | 8.8 |
| brotli 10 | 0.2 | 0.4 | 0.4 | 0.5 | 0.5 | 0.6 | 0.6 | 0.6 | 0.5 | 0.5 |

On average for all files, we can see that Brotli at quality level 4 is slightly faster than zlib at quality level 8 (and 9) while having comparable compression ratio. However that is misleading. Most files are smaller than 64KB, and if we look only at those files then Brotli 4 is actually 1.48X slower than zlib level 8!

**Connection speedup**

For on-the-fly compression, the most important question is how much time to invest in compression to make data transfer faster. Because increasing compression quality only gives incremental improvement over a given level, we need the added compressed bytes to outweigh the additional time spent compressing those bytes.

Again, CloudFlare uses compression quality 8 with zlib, so that’s our baseline. For each quality setting of Brotli starting at 4 (which is somewhat comparable to zlib 8 in terms of both time and compression ratio), we compute the added compression speed as: ((total size for zlib 8) - (total size after compression with Brotli))/((total time for Brotli)-(total time for zlib 8)).

The results are reported in MB/s. Negative numbers indicate lower compression ratio.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | [20, 1024) | [1024, 2048) | [2048, 3072) | [3072, 4096) | [4096, 8192) | [8192, 16384) | [16384, 32768) | [32768, 65536) | [65536, +∞) | All files |
| brotli 4 | 0.33 | 0.56 | 0.52 | 0.47 | 0.42 | 0.44 | -0.24 | -2.86 | 0.02 | 0.00 |
| brotli 5 | 0.44 | 0.55 | 0.60 | 0.61 | 0.71 | 0.97 | 1.01 | 1.08 | 2.24 | 1.58 |
| brotli 6 | 0.36 | 0.36 | 0.37 | 0.37 | 0.45 | 0.63 | 0.69 | 0.86 | 1.52 | 1.12 |
| brotli 7 | 0.28 | 0.20 | 0.19 | 0.18 | 0.19 | 0.23 | 0.24 | 0.34 | 0.72 | 0.52 |
| brotli 8 | 0.26 | 0.20 | 0.17 | 0.15 | 0.15 | 0.17 | 0.15 | 0.21 | 0.48 | 0.35 |
| brotli 9 | 0.25 | 0.19 | 0.15 | 0.13 | 0.13 | 0.13 | 0.11 | 0.13 | 0.30 | 0.24 |
| brotli 10 | 0.03 | 0.04 | 0.04 | 0.04 | 0.03 | 0.03 | 0.02 | 0.02 | 0.02 | 0.02 |

Those numbers are quite low, due to the slow speed of Brotli. To get any speedup we really want to see those numbers being greater than the connection speed. It seems that for files greater than 64KB, Brotli at quality setting 5 can speed up slow connections.

Keep in mind, that on a real server, compression is only one of many tasks that share the CPU, and compression speeds would be slower there.

**Conclusions**

The current state of Brotli gives us some mixed impressions. There is no yes/no answer to the question "Is Brotli better than gzip?". It definitely looks like a big win for static content compression, but on the web where the content is dynamic we also need to consider on-the-fly compression.

The way I see it, Brotli already has an advantage over zlib for large files (larger than 64KB) on slow connections. However, those constitute only 20% of our sampled dataset (and 80% of the total size).

Our Brotli module has a minimal size setting for Brotli compression that allows us to use gzip for smaller files and Brotli only for large ones.

It is important to remember that zlib has the advantage of being the optimization target for years by the entire web community, while Brotli is the development effort of a small but capable and talented team. There is no doubt that the current implementation will only improve with time.