








# Using the Client Cache for Content Encoding: Shared Dictionary Compression for the Web

Benjamin Wollmer<sup>1,3</sup>(✉) , Wolfram Wingerath<sup>2</sup> , Felix Gessert<sup>3</sup> , Florian Bücklers<sup>3</sup>, Hannes Kuhlmann<sup>3</sup>, Erik Witt<sup>3</sup>, Fabian Panse<sup>1</sup> , and Norbert Ritter<sup>1</sup> 

<sup>1</sup> University of Hamburg, Hamburg, Germany  
{benjamin.wollmer, fabian.panse, norbert.ritter}@uni-hamburg.de

<sup>2</sup> University of Oldenburg, Oldenburg, Germany  
wolfram.wingerath@uni-oldenburg.de

<sup>3</sup> Baqend GmbH, Hamburg, Germany  
{fg, bw}@baqend.com

**Abstract.** As different approaches have demonstrated in the past, delta encoding and shared dictionary compression can significantly reduce the payload of websites. However, choosing a good dictionary or delta source is still a challenge and has kept delta encoding from becoming practically relevant for today's web. In this work, we demonstrate that the often prohibitive costs of dictionary generation exhibited by earlier approaches can be avoided by simply using cache entries for content encoding: We divide web pages into different page types and use one actual page of every type as a dictionary to encode pages of the same type. In an experimental evaluation, we show that our approach outperforms current industry standards by a factor of 5 in terms of compression ratio. We discuss optimization and content normalization strategies as well as application scenarios that are possible with our approach, but infeasible with the current state of the art.

**Keywords:** Delta Encoding · Web · Shared Dictionary Compression

## 1 Introduction

*Delta encoding* is a compression method that reduces the size of a file by only describing it as a delta (i.e., relative change) with respect to another file using a dictionary that contains shared content. Our previous work shows how much data can be saved when compressing web pages by finding the optimal dictionary in the client cache and using it to calculate the delta dynamically [12, 13]. However, using the smallest delta for every individual user and file is not feasible in practice because of huge computational costs and a reduced cache hit rate on

the CDN level<sup>1</sup>. Furthermore, client and server communication would suffer by negotiating what the client cache offers. Current approaches therefore use synthetic dictionaries that are created a priori and have to be made known to client and server in advance; for example, Brotli [1] compression uses a shared dictionary that is part of every major browser distribution and contains generic tokens from web content. Approaches to use custom dictionaries for individual websites have been implemented [2,4], but ultimately failed because of the high complexity and computational costs of regenerating and redistributing new dictionaries after website deployments [7].

In this work, we show how the process of choosing dictionaries can be simplified to a degree that allows it to happen at transmission time, while still maintaining a compression ratio comparable to today’s static approaches. Instead of creating a synthetic dictionary or finding the smallest delta possible, we map every web page (i.e., file) to exactly one similar existing page as a dictionary. Dictionaries in our approach are thus predefined and therefore require neither costly computation nor negotiation when requesting new content. This opens up various possibilities for improving infrastructure efficiency and user-perceived performance in the web.

We make the following contributions:

### *C<sub>1</sub> Simplification of Dictionary Selection*

After discussing related work in Sect. 2, we show how the selection of a dictionary can be simplified by simply choosing a random page from a set of the same page type.

### *C<sub>2</sub> Evaluation with Real-World Data*

While shared dictionary compression is not available in any browser yet, we use real-world data of Speed Kit’s caching architecture in Sect. 3 to calculate the expected impact of our approach on the compression ratio.

### *C<sub>3</sub> Practical Considerations & Open Challenges*

In Sect. 4, we discuss how the complexity of shared dictionary compression can be encapsulated in a CDN-like infrastructure to make our approach available as a plugin to existing websites. We also present lines of future research to enable instant page load times through extensions of our work, such as predictive preloading of web content or transformations of the dictionary files to effectively turn them into generic templates (akin to app shells known from progressive web apps).

## 2 Related Work

**Delta Encoding** describes a file as a sequence of copy and delete instructions to build it from another file. While no implementation is used in major web browsers today, there have been efforts to include delta encoding in the HTTP

---

<sup>1</sup> Content delivery networks (CDNs) accelerate content delivery by caching resources that are requested by multiple clients [6]. This is obviously not possible for deltas, if they are computed for individual users.

standard by Mogul et al. [5]. While they showed that calculating the delta from an old version to a new one can significantly reduce the payload, they did not consider deltas between different files or a shared dictionary. VCDIFF, one of the most prominent differentiation algorithms, was proposed by Korn et al., as well as the name for the format of said algorithm [3]. There exists an open-source implementation by Google<sup>2</sup>.

**Shared Dictionary Compression** algorithms use a dictionary for multiple files instead of a dedicated dictionary for each file. Zsdt [2] or FemtoZip<sup>3</sup> can be used with a shared dictionary, but they have yet to be used for web content. Currently, the only exception in the context of the web is Brotli. Despite the other approaches, Brotli uses a static dictionary known to the encoder and decoder [1]. As a result, the dictionary does not need to be transferred. While Brotli is currently the only supported shared dictionary algorithm for all major browsers, its custom dictionary capabilities are currently unavailable in all major implementations. However, there are attempts to bring shared dictionary compression with Brotli to the major web browsers [9]. All of the above mentioned shared dictionary compressors are able to train a custom dictionary from a set of given files. Shared Dictionary Compression over HTTP (SDCH) was developed by Google [4]. The basic idea was to push dictionaries to the browser so that the browser could use those to calculate the delta with VCDIFF to newly requested files. While this feature was available in Chrome, it did not get well adapted by the community: There are reports from LinkedIn claiming they could reduce their payload by up to 81% for certain files [7], but they also stated that the dictionary calculation took longer than their release cycles. Due to the low adoption, SDCH was eventually unshipped [8].

**Speed Kit** is an architecture developed by Baqend that enables caching dynamic content (e.g., HTML files) and various other performance optimizations for websites [10] as well as an extensive real-user monitoring (RUM), processing more than 650 million page loads every month [11]. We use Speed Kit’s RUM data for our evaluation and discuss opportunities for implementing our approach in the Speed Kit architecture in Sect. 4.

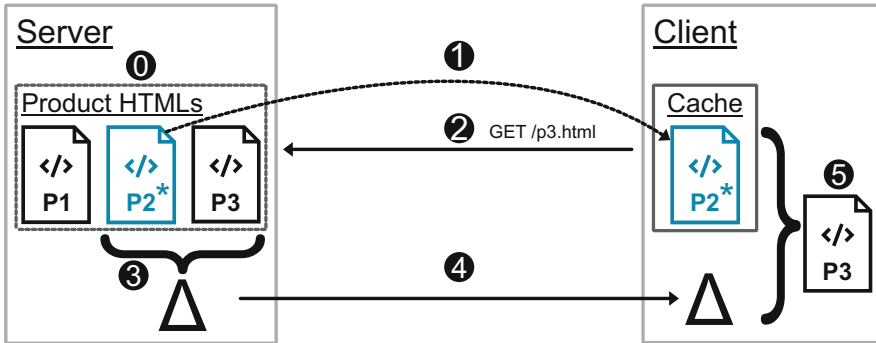
### 3 Selecting Raw Files as Dictionaries

Training a dictionary for shared dictionary compression usually involves an algorithm that analyzes hundreds of files to be compressed with the dictionary to be trained. The main goal is to find common strings (or byte arrays) across different files and to extract them into the dictionary. Web pages are especially well-suited for this kind of compression as they typically share many common strings like div tags or CSS selectors. Also, websites often contain repeating elements which are present on almost every page, such as the navigation header, the logo, search components, or the footer. Pages on a website can further often be categorized by their page type that subsumes a set of pages with the same purpose. Web

<sup>2</sup> <https://github.com/google/open-vcdiff>.

<sup>3</sup> <https://github.com/gtoubassi/femtozip>.

analytics tools like Google Analytics<sup>4</sup> routinely distinguish pages by their type, so that our approach can use the page type information without any overhead as it is readily available in virtually any production environment<sup>5</sup>. For example, e-commerce websites typically define at least the product and listing page types. While a page of type product describes a product on the website, a page of type listing aggregates products of the same category. These pages are often generated with templates on the server side, like handlebars<sup>6</sup> or twig<sup>7</sup>, and therefore share a lot of markup code by design. Transferring these repeating parts with every request is still the standard for server-side rendering. We argue that these similarities are sufficient to use pages<sup>8</sup> of the same page type as a dictionary. This has the additional benefit, that we can just use a visited page as a dictionary, without the need of transferring an actual dictionary.



**Fig. 1.** The server marks one regular HTML of every page type as a dictionary(\*). If a client has already seen this HTML (P2), it can now use it as a dictionary to only request small deltas to the requested file (P3), instead of the whole file.

Figure 1 shows the general idea of our approach. The server chooses one dictionary for every page type (product in this case) and tags it (*product 2* here) (**Step 0**). This can be precomputed once and then be used by every request. Note that this dictionary is just a regular HTML, but with a tag (e.g. HTTP header) to mark it as a dictionary. Each website would have as much dictionaries as it has page types and the dictionaries are not client specific, since every user will use the same set of dictionaries. As a result the server would only handle a couple of dictionaries. In the depicted case, the client cache already has the

<sup>4</sup> <https://analytics.google.com/analytics/web/provision>.

<sup>5</sup> The categorization within those page types is implemented by the website owner, e.g. by URL regex.

<sup>6</sup> <https://handlebarsjs.com/>.

<sup>7</sup> <https://twig.symfony.com/>.

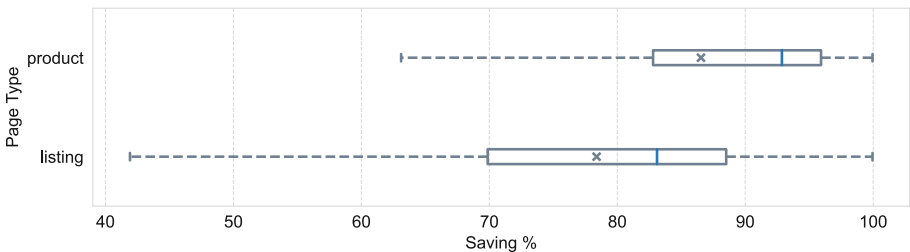
<sup>8</sup> We only use pages within the same website as a dictionary, since browsers would prohibit sharing content across different domains.

dictionary (**Step 1**), this can happen through preloading or by regularly visiting the page (c.f. Section 4.1). The client then requests *product 3*, indicating it has *product 2* in its cache (**Step 2**). The server computes the delta between *product 2* (the dictionary) and *product 3* (**Step 3**) and sends it to the client (**Step 4**). The client then applies the delta to *product 2* and receives *product 3* (**Step 5**).

### 3.1 Evaluation

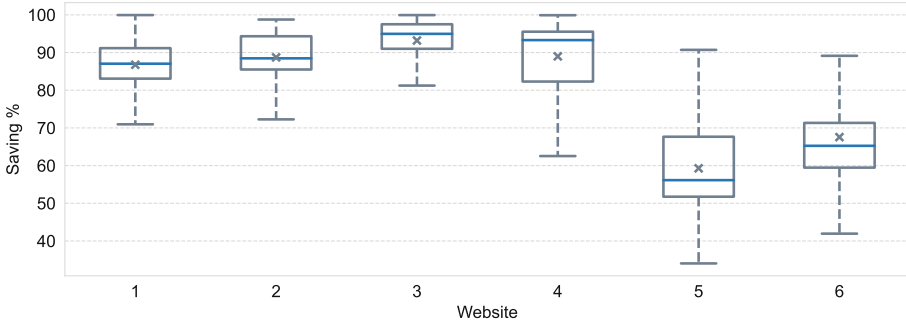
As of today, no major browser distribution supports shared dictionary algorithms with custom dictionaries. Therefore, we evaluated the expected benefits of this approach by compressing real-world HTML files and leave tests using browser implementations as a task for future work, to be conducted after release of the required browser features. We used Brotli as the compressor, since there are efforts to make Brotli’s custom dictionaries available within web browsers (cf. Section 2). As we also have shown in our previous papers [12, 13], Brotli can achieve the highest compression ratio for shared dictionary compression and also excels in decompression speed [1]. We measured our approach’s compression and decompression time with Brotli on levels 6 and 11. Level 6 is essential since it is the default level used for dynamic content. Level 11 can be used for static content, which usually results in better compression ratios but a worse compression time. The hardware we used in this experiment consists of a Ryzen 5950x, 64 GB 3600 MHz RAM and a PCI 4.0 SSD to measure the timings.

*The dataset* is provided by Speed Kit and contains HTML files of the most requested files of the last three days for six different websites. These are all e-commerce platforms. We used the two most common page types for this work: listing and product. We fetched a total of 1420 samples of the most requested HTML pages, evenly distributed across all pages and pages types. For each page type and website combination, we used every page as a dictionary for all other pages of the same page type.



**Fig. 2.** Comparing the page-type dictionary to Brotli’s default dictionary shows that our approach can save around 88% of transferred data.

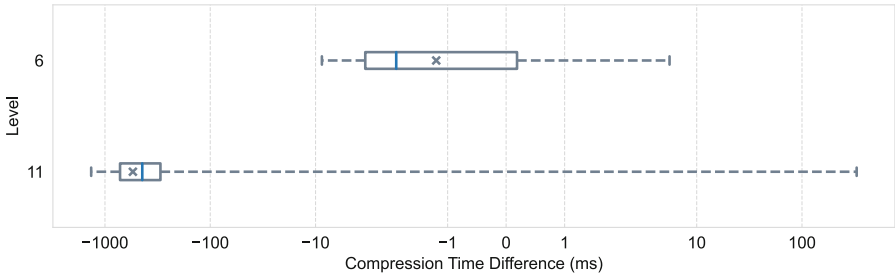
**Size.** Figure 2 shows how the data saving is distributed across the different page types. We chose compression level 11 to show the best possible output, and as shown, we could reduce the payload by 88% of Brotli compression with the standard dictionary – the mean size of those deltas where in the range of 9 kB. As a comparison, the maximum *TCP* package size is 64 kB. The files can be small enough to fit into the initial congestion window, which might improve the download performance. The relative results for Brotli on level 6 look similar and are left out because of space restrictions.



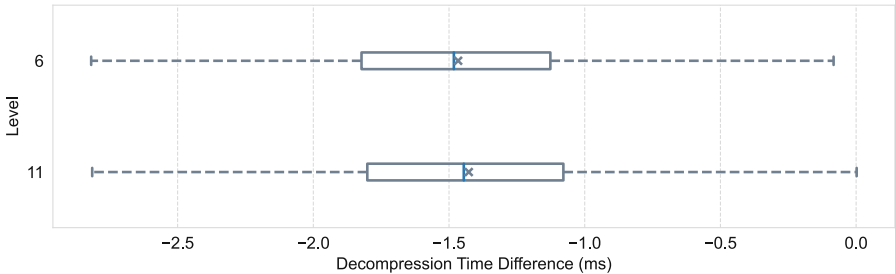
**Fig. 3.** Generally, all analyzed websites benefit from the page-type dictionary; some pages can save up to 93% of the size achieved by Brotli’s default.

The compression ratio was relatively stable within each website. Figure 3 shows the compression ratio for each website. The first four websites were able to save more than 90% of the data, compared to Brotli’s default dictionary. Website 3 performed exceptionally well, with a median saving of 93% of Brotli size with its standard dictionary. Furthermore, while websites 5 and 6 did not perform as well as the first four, they were still far ahead of the standard Brotli compression. There was also no case where the dictionary approach suffered from a worse compression ratio than the standard Brotli compression and is a considerably safe alternative.

**Performance.** As Fig. 4a indicates, the custom dictionary also resulted in a slightly faster compression time for most pages. This was less significant for the default compression level 6, but on level 11, there were time savings for up to a second. As Fig. 4b shows, the decompression time was generally stable through different compression levels. But since the smaller dictionaries resulted in fewer instructions, there was also a slight but negligible improvement (<3 ms).



(a) The high compression level benefits the most of the custom dictionary and, in most cases, saves hundreds of milliseconds.



(b) Decompressing is generally fast, and the provided uplift is neglectable.

**Fig. 4.** Comparing the absolute difference using the custom dictionary to the standard Brotli dictionary shows that the custom dictionary, almost in all cases, outperforms the standard general purpose dictionary.

## 4 Practical Considerations

Using page-type dictionaries is feasible in practice. However, there are still some practical considerations.

### 4.1 Downloading the Dictionary

As with every shared dictionary approach, this approach only works with a given dictionary. Therefore, we cannot optimize the first-page load. But when should we download the dictionary? A naïve approach would be to split the first load of a journey<sup>9</sup> into the dictionary and the delta and then download them simultaneously. This approach has two critical problems: First, we introduce a dependency within the critical rendering path. Normally, the browser can read the HTML as a stream and start its work after receiving the first chunks, e.g., to resolve dependencies. While the decompression is streamable, it can only be started if the dictionary is available. The dictionary itself is likely to be bigger than the actual compressed file. As a result, the compressed file has to wait for the dictionary, and we are essentially disabling the streaming process and therefore

<sup>9</sup> A journey describes multiple consecutive page visits of one user.

slowing down the rendering process. Second, our data showed that doing so increases the total transferred data. This is unsurprising since we transfer all data needed and the decompression operations.

Intuitively, one could also lazy load the dictionary while the browser is idle. While this would not affect our performance, we still only shift the size problem. Because now, the second page load is in total increased and only pays off after loading a third web page. The solutions to this gamble are limited. One solution would be to download the dictionary as a delta from the first page load while idling. In total, this would already payoff with the second page load, without affecting the performance of the first page load. The drawback is that the server cannot precompute this delta since the users can use every page as an entry to the website (e.g., through a google search or a link). This may not be a problem since the calculation is usually in the range of milliseconds and is not time critical because we are preloading this request. However, this approach will only work for static content. Because personalized content usually gets dynamically generated and will not be cached by the server. Caching it on the server side to compute the dictionary delta later will result in some kind of sticky sessions, which are unfavorable in a distributed system due to scalability reasons. So far, we have only talked about new users to a website. This problem does not exist for returning users. They can fully benefit from the first page load of a session and, as described in Sect. 3, are likely to shrink the whole amount of HTML bytes to the size of one HTML file. Of course, longer sessions benefit even more from this approach.

## 4.2 Creating Template Dictionaries

Since server rendered pages are built on templates (cf. Section 3), one could also use this property by simply rendering an "empty" page and using it as the dictionary for this page type. A product page could be rendered without any product information. This template should still be a valid HTML file to be renderable. The dictionary can then act as a proxy once a page of this type is requested. Like single-page applications, the browser can render this proxy template while the actual delta is requested. Dynamically replacing the rendered template has some caveats, like double javascript execution, and may need adjustment, as described in [10], but could improve user-perceived performance. And while there are algorithms to find common strings in a set of files (cf. Section 2: femtozip, zstd), to the best of our knowledge, there is currently no algorithm available to extract a valid HTML subset of a set of given HTML files. While developers could extract said template by modifying their template engine, this approach would not be feasible from a delta infrastructure on top of existing systems. To make delta encoding feasible, this needs to be resolved. To test this approach, one could choose the most straightforward way imaginable: Just opening a random HTML file of a page type for a specific website and removing content that is specific to this page and may change for another one, making sure that we still end up with a valid HTML file. Since no specific domain knowledge is needed and the page types are typically limited, this process could quickly be done. However,



more research for a templating algorithm is needed to automate and scale this templating approach.

However, by deleting the text, the content collapses and will increase in size as soon as the text of the delta arrives. This is generally poorly received by users and should be avoided. Therefore, one can change the template generation process from deleting the content to hiding it via the CSS attribute *visibility='hidden'*. The increase in file size of the additional CSS attribute is neglectable since these additions are just a few bytes after compression. Since this template still has the content, it can again be used as a preloaded HTML and instantly be rendered.

### 4.3 Dictionary Transitions

As described in Sect. 3, most pages share the same header and footer. And even though the main content does not share many similarities, compressing one page type dictionary to another generally results in reasonable compression ratios. This indicates that the other dictionaries can be easily derived after the browser is populated with an arbitrary dictionary. Therefore, it can also improve performance for the first page of an unseen page type. The same also applies to updating deprecated dictionaries to the newest version. Since the dictionaries are regular HTML files, they are invalidated using Speed Kit's approach [10]. If a deprecated dictionary is being used, the server can always just fallback to a regular, non-delta response.

### 4.4 Predictive Preloading

Preload prediction determines which pages a user will likely navigate soon and download them beforehand so that they are already available in the cache. The page can then be instantly served from the cache without downloading it again. This plays well with page-type dictionaries. While using a page-type dictionary for compression entirely discards the dictionary calculation process, it also serves as an actual page and is, therefore, present in the browser cache. With preload prediction data, this dictionary can be chosen wisely to increase the cache hit rate. Alternatively, one could use a highly frequented page, like a product advertised on the home page.

### 4.5 Shared Dictionary Compression at Scale

As shown in Sect. 3, we can save 88% of the HTML payload. Applying these numbers to the statistics provided by Similar Web<sup>10</sup>, this approach could save 700 TB a month, just for the top 50 e-commerce websites. But shared dictionary compression failed in the past due to its high complexity and slow adoption. Adopting shared dictionary compression on an architecture like the one provided by Speed Kit can eliminate this complexity for website providers and make shared dictionary compression available as a plugin. Of course, this is also of

<sup>10</sup> <https://www.similarweb.com/top-websites/e-commerce-and-shopping/>.

interest to the user. According to Similar Web, an average user journey consists of 7.5 page loads. Depending on the website, this approach could download the whole journey of HTML files for the byte "price" of one HTML (cf. Section 3.1).

## 5 Conclusion

In theory, shared dictionary compression (SDC) seems like the perfect fit for transmitting web content as it can result in significantly better compression ratios compared with today's web compression standards such as Brotli. In practice, however, SDC has yet to be adopted in a web context, because negotiating dictionaries between client and server has always turned out prohibitively complex. In this paper, we show that HTML pages from the client cache can be used as dictionaries to reduce the payload of HTML files by up to 88% for single pages, as soon as browser implementations add support for custom dictionaries with Brotli. In evaluating the compression ratio for our approach, we show that the benefit of choosing the smallest delta is negligible when comparing it to using an arbitrary file as a dictionary. We finally discuss how our approach could innovate web content delivery through mechanisms like predictive preloading of web content that are not feasible with the current state of the art.

## References

1. Alakuijala, J., et al.: Brotli: a general-purpose data compressor. ACM TOI **37**(1), 1–30 (2018)
2. Collet, Y., M. Kucherawy, E.: Zstandard Compression and the 'application/zstd' Media Type. RFC 8878, February 2021
3. Korn, D., MacDonald, J., Mogul, J., Vo, K.: The VCDIFF Generic Differencing and Compression Data Format. RFC 3284, June 2002
4. McQuade, B., Mixter, K., Lee, W.H., Butler, J.: A proposal for shared dictionary compression over http (2016)
5. Mogul, J., et al.: Delta Encoding in HTTP. RFC 3229, January 2002
6. Pathan, M., Buyya, R.: A Taxonomy of CDNs, pp. 33–77. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-77887-5\\_2](https://doi.org/10.1007/978-3-540-77887-5_2)
7. Shapira, O.: SDCH at LinkedIn (2015). <https://engineering.linkedin.com/shared-dictionary-compression-http-linkedin>. Accessed Mar 2023
8. Sleevi, R.: Intent to Unship: SDCH (2016). <https://groups.google.com/a/chromium.org/d/msg/blink-dev/nQl0ORHy7sw/HNpR96sqAgAJ>. Accessed Mar 2023
9. Weiss, Y., Meenan, P.: Compression dictionary transport (2023). <https://github.com/WICG/compression-dictionary-transport>. Accessed Mar 2023
10. Wingerath, W., et al.: Speed Kit: A Polyglot & GDPR-Compliant Approach For Caching Personalized Content. In: ICDE, Dallas, Texas (2020)
11. Wingerath, W., et al.: Beaconnect: continuous web performance A/B-testing at scale. In: Proceedings of the 48th International Conference on Very Large Data Bases (2022)

12. Wollmer, B., Wingerath, W., Ferrlein, S., Panse, F., Gessert, F., Ritter, N.: The case for cross-entity delta encoding in web compression. In: Proceedings of the 22nd International Conference on Web Engineering (ICWE) (2022)
13. Wollmer, B., Wingerath, W., Ferrlein, S., Panse, F., Gessert, F., Ritter, N.: The case for cross-entity delta encoding in web compression (extended). *J. Web Eng.* **22**(01), 131–146 (2023)