

# On Compressing the Textual Web

Paolo Ferragina<sup>\*</sup>  
Univ. Pisa, Italy  
ferragina@di.unipi.it

Giovanni Manzini  
Univ. Piemonte Orientale, Italy  
manzini@mfn.unipmn.it

## ABSTRACT

Nowadays we know how to effectively compress most basic components of any modern search engine, such as, the graphs arising from the Web structure and/or its usage, the posting lists, and the dictionary of terms. But we are not aware of any study which has *deeply* addressed the issue of compressing the raw Web pages. Many Web applications use simple compression algorithms— e.g. `gzip`, or word-based Move-to-Front or Huffman coders— and conclude that, even compressed, raw data take more space than Inverted Lists.

In this paper we investigate two typical scenarios of use of data compression for large Web collections. In the first scenario, the compressed pages are stored on disk and we only need to support the *fast scanning* of large parts of the compressed collection (such as for map-reduce paradigms). In the second scenario, we consider the fast access to *individual* pages of the compressed collection that is distributed among the RAMs of many PCs (such as for search engines and miners). For the first scenario, we provide a thorough experimental comparison among state-of-the-art compressors thus indicating pros and cons of the available solutions. For the second scenario, we compare compressed-storage solutions with the new technology of *compressed self-indexes* [45].

Our results show that Web pages are more compressible than expected and, consequently, that some common beliefs in this area should be reconsidered. Our results are novel for the large spectrum of tested approaches and the size of datasets, and provide a threefold contribution: a non-trivial *baseline* for designing new compressed-storage solutions, a *guide* for software developers faced with Web-page storage, and a natural *complement* to the recent figures on InvertedList-compression achieved by [57, 58].

## Categories and Subject Descriptors

E.4 [Coding and Information Theory]: Data com-

<sup>\*</sup>Supported in part by a Yahoo! Research grant and by MIUR-FIRB Linguistica 2006.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WSDM'10, February 4–6, 2010, New York City, New York, USA.  
Copyright 2010 ACM 978-1-60558-889-6/10/02 ...\$10.00.

paction and compression; H.3 [Information Storage and Retrieval]: Content Analysis and Indexing, Information Storage, Information Search and Retrieval.

## General Terms

Algorithms, Experimentation.

## 1. INTRODUCTION

The textual content of the Web is growing at a such staggering rate that compressing it has become mandatory. In fact, although ongoing advancements in technology lead to ever increasing storage capacities, the reduction of storage usage can still provide rich dividends because of its impact on the number of machines/disks required for a given computation (hence the cost of their maintenance and their energy consumption!), on the amount of data that can be cached in the faster memory levels closer to the CPU (i.e. DRAMs and L1/L2 caches), etc.. These issues are well known to researchers and developers, and have motivated the investigation of compressed formats for various data types— e.g. sequences, trees, graphs, etc.— which are space succinct and support fast random access and searches over the compressed data. The net result is a rich literature of techniques that may be used to build efficient indexing and mining tools for large-scale applications which deal with the many facets of Web data: content (text and multimedia in general), structure (links) and usage (navigation and query logs). Nowadays we know how to effectively compress most basic components of a search engines, such as, the graphs arising from the Web structure or its usage (see [10, 22, 16] and refs therein), the posting lists (see [18, 59, 58, 57] and refs therein) and the dictionary of terms (see [7, 35, 56]).

However, most Web IR tools also need to store the html pages for retrieval, mining, and post-processing tasks. As an example, any modern search engine—like Google, Yahoo!, Live/bing, Ask—offers two additional precious functions which need the original collection of indexed Web pages: *snippet retrieval* (showing the context of a user query within the result pages), and *cached-link* (showing the page as it was crawled by the search engine). The challenge in storing the original html-data is that its size is by far larger than the *digested* information indexed by the above (compression) techniques. Just to have an idea, the internal memory of a commodity PC may host a graph of few million nodes (pages) and billion edges (hyper-links) [10, 22], as well as it may store the search engine built on the textual content of those (few millions) pages [58]. The same seems

to be hardly true for the storage of the original `html`-pages, because their overall size can reach hundreds of Gigabytes!

Despite the massive literature produced in this field (see Sect. 1.1), we are not aware of any experimental study which has *deeply* addressed the issue of compressing Terabytes of Web-data by comparing all best known approaches and by giving *full details* about them. Clearly, this is a key concern if one wants to choose in a principled way his/her own compressed-storage solution which best fits the specialties of the (Web) application that will use it.

Known results for the compression of Web collections are scattered over multiple papers which either do not consider large datasets [29, 41], or refer to old compressors (e.g. [24, 46, 40, 44]), or do not provide the fine details of the proposed compressor (e.g. [20]). We think that it is properly for these reasons that many *real* Web applications use relatively simple compression algorithms. For example, LUCENE [23] and MG4J [11] use `gzip` on individual pages; the MG system [56] uses a word-based Huffman coder; [54] uses `Move-to-Front` to assign (sub-optimal) codewords to input words. In all these cases the compression performance is far from what we could achieve with state-of-the-art compressors. In fact, it is well known that using `gzip` on each individual page does not capture the repetitiveness among different pages, and that Huffman and `Move-to-Front` do not take advantage of the fact that each word is often accurately predicted by the immediately preceding words. Nevertheless these simple compressors are the typical choice for Web applications because they “fast” access single pages and achieve “reasonable” compression ratios. However, apart from some folklore results (such as 20% ratio for `gzip` on single pages, or 10% for Google’s approach [20]), no complete and fully detailed experimental study has yet investigated the compressed storage and retrieval of very large Web collections.

Let us first review the literature of this field in order to address then this issue in a more principled way.

## 1.1 Problem definition and related results

We consider the problem of compressing a *large* collection of `html`-pages. We assume that the pages in the Web collection are joined into a *single long file* which is then compressed using a *lossless* data compression algorithm. This approach offers the potential of detecting and exploiting redundancy both within and across pages. Note that how much inter-page redundancy can be detected and exploited by a compressor depends on its ability to “look back” at the previously seen data. However, such ability has a cost in terms of memory usage and running time.

The classic tools `gzip` and `bzip` have been designed to have a small memory footprint. For this reason they look at the input file in small blocks (less than 1MB) and compress each block separately. These tools can thus detect redundancy only if it is relatively close within or among pages. More recent and sophisticated compressors, like `ppmd` [49, 56] and the family of `bwt`-based compressors [29, 56], have been designed to use up to a *few hundreds* MBs of memory and thus have the ability to take advantage of repetitions that occur at much longer distances. However, since we are dealing with very large collections it is clear that even these tools cannot detect and exploit all their redundancy. These are known problems [36], and for their solution we have nowadays the following three approaches.

**Use of a “global” compressor.** A natural approach is to

modify an existing compressor giving it unlimited resources so that it can work, and search for regularities, on arbitrarily large files. We tested this approach by using an algorithm based on the Burrows-Wheeler Transform (shortly `bwt` [56]), and applied it on the whole collection. This algorithm is able to compress the collection up to its  $k$ th order entropy, for any  $k \geq 0$  [30]. However, because of the collection sheer size, the `bwt` cannot be computed entirely in internal memory. We therefore resorted a disk-based `bwt`-construction algorithm [28] and used it to compress one of our test collections (namely UK50 of 50GB), achieving the impressive compression ratio of 4.07%. Unfortunately the computation took roughly 400 hours of CPU time, so this approach is certainly not practical but it will be used in Sect. 2.1 to provide a reasonable estimate of the compressibility of UK50 via modern compressors with *unbounded* memory.

**Delta compression.** This technique was proposed to efficiently encode a given target file with respect to one, or more, reference files. This can be viewed as compressing only the difference between the two files. This scheme is pretty natural in the context of (low bandwidth) remote file synchronization and/or in (HTML) caching/proxy scenario [51, 53]. When used on file collections the problem boils down to find the best pairwise encodings, i.e. to find a maximum branching of a *complete* directed graph in which the nodes are the files and the edge-weights are the benefit of compressing the target vertex wrt the source vertex. Over large collections this approach is unaffordable because of its quadratic complexity. Therefore heuristic approaches for graph pruning have been proposed to scale it to larger data sizes [46, 27, 24]. Overall, these heuristics are still very sophisticated, depend on many difficult-to-trade parameters, and eventually achieve the negligible improvement of 10% wrt `bzip` (see Tables 1-3 in [24]) or 20% wrt `gzip` (see Table 6 in [46]). Conversely, the compressors we consider in this paper achieve an improvement of up to 75% wrt `bzip` and 86% wrt `gzip`. Furthermore, since in this paper we test `lzma`, a `gzip`-like compressor able to detect redundancies far back in the input file, we are implicitly using a sort of *delta*-encoding with a very large number of *reference files* (shown in [19] to achieve significant improvements).

Recent work (see e.g. [40] and refs therein) has improved the *delta*-encoding scheme over long files by using sophisticated techniques—like chunk-level duplicate [44, 52, 53] or resemblance detection [42, 14]. The main idea consists of partitioning the input file into *chunks*, which are either fixed-size blocks (identified with possibly rolling checksums as in `rsync`), or content-defined variable-sized blocks (see e.g. SHA hashes or [48, 53]). These chunks are then compared to *logically* remove the *duplicate* chunks, whereas the *near-duplicate* chunks are *delta* encoded. The efficient identification of (near-)duplicate chunks is done via sophisticated (super-)fingerprinting techniques [42, 14]. Remaining chunks are compressed by any known data compressor. `Rebl` [40] is one of the best systems to date that implements these ideas: however, its performance depends on many parameters and experiments on a Web collection showed that it is worse than the combination of `tar + gzip` (Table 2 in [40]).

Another related tool is the one described in [20] where the Bentley-McIlroy algorithm [6] is used to find, and squeeze out, long repeated substrings at long distances. Then, a variant of `gzip` is used to compress the remaining data taking advantage of “local” similarities (this is the same idea

underlying the Vcdiff standard<sup>1</sup>). The authors of [20] use this simple, but effective, approach to store large Web collections sorted by URL-addresses, and report the remarkable compression ratio of 10% (cfr. 20% of `gzip`). Fine details about this algorithm are missing in [20], their Web collection is not accessible, and no comparison with other compressors is provided at all. In the next sections, we propose an implementation of this promising approach and compare it with all best known compressors over our two Web collections: WebUK and GOV2 (see Sect. 2).

**Page reordering.** Instead of trying to capture repetitions which are far away in the input, we can rearrange the Web pages so that similar ones end up close one another in the input file. The most well known re-ordering heuristics for Web pages are the `url`-based one and the `similarity`-based one. The former has been applied successfully to the compression of Posting Lists [50, 58], Web Graphs [10] and, recently, also to the compressibility of Web-page collections [20]. It exploits the fact that pages within the same domain typically share a lot of intra-links, content, and possibly the same *page-template*. The second approach is IR-inspired and it is used by many Web-clustering tools. The main idea is to compute a page clustering based on various features: syntactic [15, 38, 46] (e.g. shingles and their derivatives [42, 14]), query-log [47], Web-link [43], or TF-IDF [56]. In all cases the goal is to detect pages which are similar enough that it is worth to compress them together. Although some progress has been done to speed up the clustering process, see e.g. [5], the `url`-based approach is so simple, time efficient and compression effective, that it is a natural choice in the Web-compression setting. Our experiments will quantify and validate this common belief, and they will pave the way for the investigation of other approaches (such as [9, 21]).

## 1.2 Our contribution

The main goal of this paper is to consider two main scenarios for the compressed data-storage of large Web-collections. In the first scenario the compressed pages are stored on disk and we only need to support the *fast scanning* of large parts of the compressed collection (such as for map-reduce paradigms [26, 1]). In the second scenario we assume that we also need to *fast access* individual pages of the compressed collection in a random way (such as for Web search engines and miners), so we assume that the compressed collection is distributed among the RAMs of (possibly) many PCs.

In order to address both issues in a principled way we proceed in this way. For the first scenario we test the effectiveness of *all* state-of-the-art compressors over two Web collections: one drawn from the UK domain in 2006-07 [8] (of about 2.1TB) and the classic GOV2 collection (of about 440GB). We take into account various issues such as parameter settings, working memory, `html` vs `txt` compression, etc., and we investigate the impact on compressibility performance of the *re-ordering* of the Web pages— i.e. random, crawl-based, `url`-based, and *similarity*-based. Compression (ratio/speed) figures are reported in Sect. 2: the overall conclusion is that UK-collection can be compressed to *less than 3%* of its original size and this corresponds to roughly *one seventh* of the space used by `gzip` (a figure of 3.84% holds for GOV2). This significant space reduction (without a significant decompression slowdown wrt `gzip`) cannot be neglected

given its impact on the energy/maintenance costs required for a Web-storage system [2, 4].

As for InvertedList [58] and graph-compression [10], we will show that `url`-based page reordering is effective and may reduce the compression ratio of a factor up to 2.7, especially for small working memories (i.e. 10MB). Conversely, similarity-based orderings are not advantageous in terms of time and compression performance (as observed by [50] for InvertedList-compression). Algorithmically we can conclude that, whenever the sequential scanning of large portions of Web data is requested, dictionary-based compressors (like `lzma` and other variations) seem unbeatable because of their cache-aware access to the internal memory of the PC, thus confirming the choice made in [20].

As a cross-check for our UK-results we tested our best compressors over the GOV2 collection, and found that GOV2 is less compressible than UK-full. More importantly, we compare the compressed storage of InvertedLists built on GOV2 *vs* the compressed storage of its raw pages, drawing some surprising conclusions. The most interesting one is that the storage of *term*-positions in ILs, which is useful to implement phrase-queries or proximity-aware ranking functions (see e.g. [57]), is larger than the space taken by the compressed raw pages! This suggests the need of improved integer-compression techniques for IL-postings which use more information in addition to term frequencies and their distributions within individual documents.

For the second scenario (for which we also need fast access to compressed data), in addition to the data compression tools, we consider the *compressed self-indexes* which are a recent algorithmic technology supporting fast searches and random accesses over highly compressed data (see [45] and refs therein). The key idea underlying these indexes is to combine `bwt`-compressed files with a sublinear amount of additional information that allows to support fast searches and accesses over the compressed data. These relevant theoretical achievements have been validated by several experimental results (see [31] and refs therein) which, however, have been confined to small data collections (up to 200MB), and involved only *well-formed* textual datasets (e.g. Wikipedia and DBLP) that are very distant from the *noisy* but possibly *highly repetitive* data available on the Web. We compared (the fast) dictionary-based compressors, built over small blocks of pages (less than 1MB), against the best compressed (self-)index to date (namely CSA [31]) and found that the combination between the `bmi`-preprocessor [6] and `gzip`, suggested in [20], offers the best space/time trade-off. Compressed self-indexes turn to be not yet competitive on Web collections, but new engineering and theoretical results [12, 13] could change this scenario. In Sect. 3 we fully comment on this issue and show that there are other contexts, such as the storage of emails or logs, in which the *records* to be individually extracted are shorter (typically few KBs in size, *vs* 20KB of Web pages) and thus the algorithmic features of compressed self-indexes could turn them to be competitive.

We believe that our wide set of experiments provides a thorough comparison among state-of-the-art compressors and compressed self-indexes indicating pros/cons of advanced solutions for disk-based and memory-based compressed storage of large Web-collections. These contributions are novel in terms of the large spectrum of tested approaches and the size of processed datasets, and suggest a non-trivial *baseline* for estimating the performance of newly

<sup>1</sup>See <http://code.google.com/p/open-vcdiff/>

designed compressed-storage solutions for Web collections. In addition, our results constitute a natural complement to the recent figures drawn on Inverted-lists compression by [58, 57] (see next sections for further comments). Overall, we can conclude that Web collections are more compressible than expected, and it has become necessary to reconsider whether the choice of compressing them by using simple tools, such as `gzip`, is the best one: before implementing your next Web-application, consider also the open-source tools `lzma`, `bmi` and `CSA`.

## 2. COMPRESSION FOR DISK STORAGE

In this section we describe our experimental study for the first scenario depicted in the Introduction, namely the one in which the compressed pages are stored on disk and we only need to support the scanning/decoding of large parts of the compressed collection (such as for map-reduce paradigms [26, 1]). We thus assume that the collection is compressed once and decompressed many times, so we will favor algorithms with a higher decompression speed. In this scenario we will address three main questions: How much compressible is the textual content of the Web? Which are the best algorithms for compressing a large collection of Web pages and how much their performance depend on their parameters setting? How the ordering of the Web pages does impact on their compression ratio?

**Dataset.** For our experiments we used a Web collection crawled from the UK-domain in 2006-07 [8] consisting of about 127 million pages for a total of 2.1TB.<sup>2</sup> We call this collection UK-full. In order to test the largest possible set of compressors and compression options, we extracted a sample of this collection consisting of the initial 50GB (about 3 million pages), this is called UK50.<sup>3</sup> We also tested our best compressors over the classic GOV2 collection, consisting of about 440GB and 25 million pages.

We run our experiments on six dedicated Linux machines; the machine used for the timings is a dual-P4 at 3GHz, with 1MB cache and 2GB RAM.

**Algorithms.** Our experimental results refer to the following suite of compressors:

- `gzip` and `bzip`. The classic compression tools with option `-9` for maximum compression.
- `gzip-sp`. The algorithm `gzip` (again with option `-9`) applied to the single pages of the collection.
- `lzma`<sup>4</sup>. This is a dictionary-based algorithm, like `gzip`, that can use a very large dictionary (up to 4GB) and compactly stores pointers to previous strings using a Markov-chain range-encoder. We tested the compression level “Ultra” (option `-mx=9`) with a dictionary size of 128MB (option `-md=128m`).
- `bzip*`. This algorithm is analogous to `bzip` except that it operates on the *whole input* instead of splitting it in blocks of size 900KB [29]. `bzip*` computes the `bwt` of

its entire input and compresses it by applying Move-to-Front, `Rle`, and Multi-table Huffman coding. This algorithm is called `MtfRleMth` in [29]. We tested also other `bwt`-based algorithms from [29]: `bzip*` was the one with the best compression ratio/speed tradeoff.

- `ppmd`. This is the `ppm` encoder from [49], in which we used the maximum-compression setting: 16th-order model with cut-off and 256MB of working memory (options `-r1 -o16 -m256`).
- `bmi+gzip`, `bmi+lzma`. These compressors combine Bentley-McIlroy’s algorithm [6] (shortly, `bmi`), for finding long repeated substrings at large distances, with `gzip` and `lzma`. We run the `bmi` tool<sup>5</sup> with a chunk of 50 chars, which was the setting providing the best compression ratio on our datasets.

We point out that we tested many other compressors and compression options in addition to the ones reported above, but we obtained worse performance or slight variations that do not justify their reporting here. For example we tested the powerful `Paq8` and `Durilca` compressors, but we found that they achieve *very small* improvements in the compression ratio, at the prize of an unacceptably slow compression/decompression speed (in accordance with the results reported in [41]). We also tested the bit-optimal LZ-compressor, recently proposed in [34], but its compression ratio resulted slightly worse than `lzma`, the main reason being that the current implementation of this compressor uses a poor encoder for the LZ-phrases. Hence improvement is expected on this powerful tool by engineering its encoding functions, this is left as a future research issue.

We also tested some compressors based on the *differencing* approach (e.g. the `Vcdiff` standard): `open-vcdiff`<sup>6</sup>, `xdelta`<sup>7</sup>, and `vcodex`<sup>8</sup>. Since these are *differencing* algorithms which encode an input (target) file given a (reference) dictionary file, we used them by setting the dictionary to be empty, and by enabling the option to *copy* from the input file itself. Overall, we experienced on UK50 a compression ratio and a (de)compression speed which was inferior to `bmi+gzip`.

A comment is in order on `bmi+gzip`. It tries to mimic the approach suggested in [20] where the authors report that using `bmi` and a variant of `gzip` they were able to reach the impressive decompression speed of 400MB-1GB/secs. Our `bmi+gzip` algorithm is not as fast; we do not know if this depends on our machine (given that the classic `gzip` is also significantly slower than those figures) or on the specialties of their decompressor: in any case, the missing details of [20] do not allow us to go deeper in this issue. We also tested the `rzip` compressor<sup>9</sup> (version 2.1, option `-9` for maximum compression), but we found it to be inferior than our `bmi`-based algorithms, probably because ours use larger windows. So we tested also the combination `bmi + bzip*` (`bzip` with unbounded window), but it resulted worse than `bmi+lzma` both in decompression speed and compression ratio.

Overall, our selection of algorithms provides a significant sample of the options nowadays available for the compression of Web collections: we have the top performers for the three

<sup>2</sup>We merged the files `law0-7` of [8], discarding the WARC header and the pages whose `gzipped-file` was corrupted.

<sup>3</sup>We repeated some experiments on a sample of 300GB, without obtaining significant differences in compression time/ratio performance. Even using the smaller UK50 collection our experiments took more than 8 months!

<sup>4</sup>Available from <http://www.7-zip.org/>

<sup>5</sup>Available from [www.cs.dartmouth.edu/~doug/source.html](http://www.cs.dartmouth.edu/~doug/source.html)

<sup>6</sup>Available from <http://code.google.com/p/open-vcdiff/>.

<sup>7</sup>Available from <http://code.google.com/p/xdelta/>

<sup>8</sup>Thanks to P. Vo for his code

<sup>9</sup>Available from <http://rzip.samba.org>.











