On Compressing the Textual Web

Paolo Ferragina*
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 compaction 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

*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.

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 kth 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 sophisticate (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[1]). 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

[1]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.[2] 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.[3] 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[4]. 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[5] 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[6], xdelta[7], and vcodex [8]. 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[9] (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

[2]We merged the files law0-7 of [8], discarding the WARC header and the pages whose gziped-file was corrupted.

[3]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!

[4]Available from `http://www.7-zip.org/`

[5]Available from www.cs.dartmouth.edu/∼doug/source.html

[6]Available from http://code.google.com/p/open-vcdiff/.

[7]Available from http://code.google.com/p/xdelta/

[8]Thanks to P. Vo for his code

[9]Available from http://rzip.samba.org.

Compressor	c/ratio	c/speed	d/speed
gzip-sp	21.75	9.70	34.08
gzip	20.35	8.26	69.15
bzip	17.64	2.90	10.38
bzip*-10	11.53	1.86	4.36
bzip*-50	7.30	1.45	3.57
bzip*-100	6.30	1.21	3.31
bzip*-200	5.62	1.03	2.95
ppmd-10	9.83	2.99	2.60
ppmd-50	7.21	1.79	1.67
ppmd-100	6.82	1.58	1.46
ppmd-200	6.65	1.42	1.35
ppmd-400	6.58	1.42	1.35
ppmd-1500	6.57	1.41	1.29
lzma-50	6.31	1.29	52.58
lzma-100	5.64	1.27	56.11
lzma-200	5.22	1.22	56.53
lzma-400	4.96	1.19	51.94
lzma-1500	4.85	1.18	38.36
bmi+gzip-50	8.62	4.01	23.41
bmi+gzip-400	7.14	2.62	23.30
bmi+lzma-50	6.31	1.28	19.05
bmi+lzma-400	4.91	1.63	19.31

Table 1: Compression of the UK50 collection. Compression ratios are expressed as percentages with respect to the original size. Compression and decompression speed are in MBs per second. Times include the cost of reading/writing from/to disk.

families of compressors, namely dictionary-based (gzip, lzma and their bmi-combinations), PPM-based (ppmd) and bwt-based (bzip and bzip*).

Since many of the compression algorithms are not able to handle GB-size files, we tested them by splitting the input in blocks of different sizes. Therefore, in the following, we will use the notation bzip*-N, ppmd-N, etc., to denote that the input is split into blocks of (about) N MB each (individual pages are not split), which are then compressed separately by the given compressor. The maximum value of N was chosen to give each compressor roughly the same amount of working memory.

2.1 Crawl-based page ordering

The collection UK50 consists of a single file of about 50GB in which pages have been appended by UbiCrawler [8] during its BFS-visit of the UK-domain, starting from a seed set of hosts. As soon as a new host is met, it is entirely BFS-visited (possibly with bounds on the depth reached or on the overall number of pages). For this reason we say that pages are in *crawl-based ordering*. Compression results for UK50 are summarized in Table 1.

We see that gzip-sp achieves about 20% compression, thus confirming the folklore figure (mentioned at the beginning of the paper) and showing that there is a significant *intra*-page repetitiveness. gzip and bzip are only slightly better than gzip-sp: hence, the *inter*-page repetitiveness is negligible at *small* distances. Our choice of gzip as a reference compressor is motivated by the fact that most open-source search engines (like Lucene [23] and MG4J[8]) use the zlib library which offers the same compression as gzip (note that Lucene and MG4J do not have document compression as their main objective). Hence, gzip ratios provide us with a simple, heavily used, baseline for our following experiments.

Algorithms able to work on large blocks (10MB and more) are significantly better than gzip and bzip. Blocks of 10MB reduce the output size of gzip-sp by half; blocks of size 200MB or more get a saving of a factor from 3 to 4.5! This proves that there is indeed also a significant inter-page repetitiveness in the crawl-based order at medium distances (which are within the capabilities of modern compressors). In the next sections we will investigate to what extent the inter-page repetitiveness is due to the page ordering. Here we concentrate on the crawl-based page ordering. From Table 1 we see that bzip* is better than ppmd for increasing block sizes, both in terms of compression ratio and decompression speed. This confirms on larger datasets the theoretical figures of [30]. However ppmd and bzip* are both relatively slow: they are up to 6 times slower than gzip in compression, and more than 30 times slower in decompression. It is particularly impressive the performance of the dictionary-based compressor lzma, which is slightly better than bzip* in compression ratio and speed, and it is significantly better in decompression speed (up to 25 times). Indeed it is almost as fast as gzip! Thus we have the first important experimental result of this paper:

> Dictionary-based compressors using sophisticated phrase-encoders (like lzma) achieve a compression ratio and a decompression speed which is superior to the (theoretical) state-of-the-art compressors based on bwt- or ppmd-approches.

This shows that dictionary-based compressors are very competitive if well engineered in their phrase encoders, as theoretically shown in [34]. At the same time, the appealing entropy-based results of [30] stress the need of further engineering and research about bwt-based compressors.

As far as the preprocessor bmi is concerned, we notice that it significantly improves the compression performance of gzip, but not the one of lzma. Experiments show that bmi alone reduces the input up to 19.31% of its original length, suggesting that repetitions of chunks of 50-chars are surprisingly frequent in UK50. Thus bmi+gzip achieves roughly the compression ratio of bzip* and ppmd, but it is much faster in decompression and much easier to code. Decompression speed is actually much slower than the one claimed in [20] (see our previous comments). Conversely, the combination bmi+lzma seems not to pay off, because lzma already uses a large window and dictionary.

> bmi-preprocessing is an easy-to-code and effective compression booster for gzip. Its current implementation [6] is slow, but further tuning could probably lead it to the performance claimed in [20].

At this point it is natural to ask how far we can get by providing a compressor with unlimited memory resources, or equivalently, how much repetitiveness does it exist at larger and larger distances. Recall that applying the bwt to the entire UK50 yields a compression ratio of 4.07% (see Sect. 1.1), this is roughly 1% better than lzma's performance. Hence state-of-the-art compressors working over blocks of few hundreds MB do a very good job by using reasonable time and space resources; nonetheless, there is still some room for improving their compression ratio.

Finally, we have tested the best compressors over the total UK-full collection by taking the largest block sizes that can be dealt efficiently with our PC. Table 2 reports that

	gzip	bzip*-200	lzma-400	bmi+lzma-400
UK50	20.35	5.62	4.96	4.91
UK-full	17.51	3.62	3.06	2.83
UK50 txt	31.80	8.03	7.68	7.28
UK-full txt	25.97	6.36	5.63	5.37

Table 2: Testing the sample of 50GB and the full UK-crawl, with or without the `html`-markup. Size is given in GBs, and compression ratio in %. `txt`-only collections are about 30% of the corresponding `html`-ones.

the whole collection is more compressible than UK50 (about 2% better), reaching the impressive figure of 2.83%. Table 2 also measures the influence of the html-structure of the Web pages over their compressibility. This is interesting because `html`-pages are needed by search engines to return the *cached*-version of Web pages; whereas `txt`-only pages are useful in the context of snippet retrieval or for some recently proposed proximity-aware ranking functions (see [57] and Sect. 3.1). Given that the `txt`-only collections are about 30% of the original ones, it is easy to calculate the compression ratio obtained by bmi+lzma-400 over UK50 txt and UK-full txt with respect to the total `html`-collections by dividing the figures reported in Table 2 by 3.3: so we get, respectively, 2.21% and 1.63%. This means that the (compressed) html-markup constitutes less than 40% of the total compressed data.

> The UK-full collection (`html`) can be compressed in less than 3% of its original size, whereas its `txt`-only version can be compressed up to 1.63%. Here, bmi becomes an useful compression booster for lzma.

2.2 URL-based page ordering

Inspired by papers on compressing PostingLists [50, 58] and WebGraphs [10], and following the approach proposed in [20], we have permuted the (about) 3 million pages of UK50 according to their url with the host-name reversed. Results reported in Table 3 suggest three main considerations: (1) compression ratios are impressive, better than the collection-wide bwt (Sect. 1.1), and robust with respect to the block size; (2) bzip* turns out to be surprisingly better than lzma on small block sizes, but worse for larger blocks; (3) in terms of speed/compression trade-off, lzma is still the best choice. The effectiveness of the `url`-based reordering strategy in bringing similar pages close one another is confirmed by the fact that even using relatively small blocks (10MB) we get the impressive compression ratio of 4.24% (bzip*). Interesting is also the increased decompression throughput of dictionary-based algorithms (i.e. lzma and bmi-combinations), probably due to a reduction in the number of phrases to be decoded (because of an increased compression ratio).

> The url-based ordering is fast to be computed, robust over all compressors, and achieves over UK50 the result of 3.78% with lzma-400.

We also tested the formation of *variable-sized* blocks defined as groups of pages coming *only* from the *same* host. With this blocking, bzip*-200 got less than 3.80%. Hence we think that the so called *Permuting-Partitioning-Compressing* paradigm investigated in [33] is worth of further theoretical/experimental study (cfr [52]).

Compressor	c/ratio	c/speed	d/speed
ppmd-10	4.53	6.46	5.61
ppmd-400	4.28	3.02	4.03
bzip*-10	4.24	0.78	3.59
bzip*-200	3.82	0.48	2.71
lzma-10	4.25	1.94	60.65
lzma-100	3.86	1.79	66.32
lzma-200	3.80	1.78	68.90
lzma-400	3.78	1.76	71.46
bmi+gzip-10	5.32	2.81	24.96
bmi+lzma-10	4.26	2.12	21.17
bmi+gzip-200	5.02	2.36	25.23
bmi+lzma-200	3.83	1.85	22.03
bmi+lzma-400	3.79	1.76	21.56

Table 3: Compression of the UK50 collection when `URL`-sorted. Compression and decompression speed are given in MB per second.

We finally observe that the impact of url-based ordering is more significant for smaller blocks, reaching an improvement of a factor 2.7 (bzip*-10), than on larger blocks (cfr Table 1). So one might argue that the UK-collection has a quite fortunate ordering of the Web pages due to the BFS-strategy of UbiCrawler [8]. To investigate this issue we have *randomly* permuted the pages of UK50, and compressed them via bzip*. It achieved a ratio between 9.14% (bzip*-100) and 13.65% (bzip*-10). Since in the randomly permuted file there is likely not much repetitiveness in the content of nearby pages, we can estimate that this compression ratio is mainly due to the html markup.

> The best compressors on the randomly-permuted pages of UK50 yet achieve a compression ratio of less than 10%, more than a factor 2 better than gzip.

2.3 Other page orderings

The next logical step is to consider other types of orderings of the Web pages, namely the ones induced by their *content*. Following a standard IR-practice [56], we have mapped the (about) 3 million pages of UK50 to high-dimensional vectors, considering three possible kinds of features for their terms– bm25, tfidf, and bool (presence or not of a term).[10] Then, these high-dim points have been clustered via a distributed k-means algorithm implemented over the Hadoop system [1].[11] We considered two values for k, namely $k = 50$ (coarse clustering) and $k = 500$ (finer clustering). The larger value of k ensures to fit every cluster into few blocks of 100MB, and thus deploy the large compression windows of our compressors. Finally, we have permuted the Web pages by taking into account their *cluster-ID*, and in case of equality, their url (host reversed). We found that bm25-features are better than bool-features (both around 3.95% with bmi+lzma-400), tfidf being the worst. The difference in performance is quite small and worse than the result achievable with the url-based ordering, but taking much longer time.

[10] We have one feature per (stemmed and alphanumeric) term of a page. Stop words, too short terms (≤ 2 chars), and too long terms (> 40 chars) are removed.

[11] We warmly thank Vassilis Plachouras for running the clustering experiments on the Yahoo's Hadoop system.

In Sect. 1.1 we commented about other approaches to cluster-based compression, such as [46, 24]. Our analysis and experiments subsume those approaches because they deploy a shingle-based similarity measure among pages and then apply an LSH-based clustering scheme [38]. The authors of [46] admit the noisy behavior of their approach and finally take shingle-size equal to 4 bytes/chars, saying that larger shingles get similar (or worse) results. In our case we have parsed the textual part of a page into words (of avg length $5 \geq$ shingle-length 4). Our k-means clustering is also more flexible than the LSH-clustering suggested in [46] because we have properly set k in order to fit all pages of a cluster into the internal memory of the PC, and thus compress all of them together.[12] Our approach can be looked at as a *differencing*-algorithm with a large number of *source*-files, which is exactly what [46, 24] suggested as "future research" given that they used no more than 2 reference files and argued that more than that would have induced a better compression.

2.4 Some further comments

It is clear that our results are far to establish the ultimate compression for large Web collections. They have simply shown what can be done by using state-of-the-art compressors. Nevertheless, the figures reported in this paper show that the *easy-to-code* approach using gzip which is at the base of many open-source search engines and data miners (such as Lucene and MG4J) may be more than *seven times worse* in compression ratio than what state-of-the-art solutions can achieve. Recent open source compressors like lzma cannot be neglected: they are well tested and engineered, and they have better performance than bwt-based and ppmd-based compressors that were considered the best ones in practice. This suggests that further study is needed for dictionary-based compressors about the combination between bit-optimal parsers (like [34]) and better phrase-encoding strategies (like the ones used in lzma).

Our results validate the choices made in [20] (see also Sect. 3). Our current implementation of bmi+gzip seems to be slower but more compression effective than what they report in their paper (10% on an unknown Web-collection of few TB). Given that [20] does not report the details of their implementation, and their dataset and code are unavailable, we cannot draw any conclusion. On our dataset the more sophisticated lzma resulted more effective and efficient than bmi+gzip, thus turning to be a valid sparring partner for the design of Web-page storage systems (see Sect. 3).

We notice that our figures can be taken as a non-trivial *baseline* for estimating the performance of new compressors which will be proposed in the future. These compressors could either aim at better encodings for LZ-phrases (see e.g. [34], and lzma) and/or propose novel *clustering approaches* for the pages based on other features– such as query-log [47], Web-links [43, 9, 21], `html`-structure [39], or still the raw content by using more sophisticated fingerprinting and content-based chunking methods (see e.g. [53, 48, 37]). It goes without saying that the trade-off between compression ratio and (de)compression speed has to be taken into account here, because Web miners and search engines, sooner or later, have to access those compressed data. Last, we point out that our experimental figures might be also used for evaluating the efficacy of clustering methods at the Web-scale in an Information-Theoretic sense: take the compression ratio that the clustering method allows to achieve!

[12] We have also tried an LSH-ordering, derived by taking the projections onto a random line in k-dim of the points having as coordinates the distances of the pages from the pivots of the k clusters. The random line was "drawn" according to a p-stable distribution [25]. The results were worse and thus are not reported here.

Compressor	GOV2	GOV2-sort	GOV2 `txt`-only	GOV2-sort `txt`-only
gzip	18.69	10.41	23.93	13.46
bzip*-200	6.45	3.96	10.49	7.07
lzma-400	6.15	3.84	9.90	6.54
bmi+lzma-400	6.05	3.89	9.70	5.28
IL [58] docIDs + freq	1.52–1.90	0.85–1.30	—	—

Table 4: Compression ratios and indexing space for the GOV2 collection. Since the `txt`-only collection is about 50% the original GOV2, one can compute the compression ratio wrt the whole `html`-based collection dividing by 2. Of course, there is no difference in the IL-space of `html` and `txt` collections.

As a cross-check for our UK-results we tested our best compressors onto the GOV2 collection. Table 4 reports our experimental results for four cases: the original dataset and the `URL`-sorted dataset, with or without the `html`-markup. We notice that GOV2 is less compressible than UK-full with the gap shrinking when GOV2 is `url`-sorted (probably this depends on its crawl-based ordering). The `url`-based ordering reduces by roughly 1.6 times the compression ratio of every algorithm; and the best compressors are up to 3 times better than gzip both on crawl-based and sort-based GOV2.

In addition, the last line of Table 4 reports the space-usage of the best known compressed InvertedList [58]. This allows us to compare compressed IL-storage *vs* compressed document storage, drawing some interesting conclusions. First, we notice that `url`-based page ordering is similarly effective on IL-compression as it is on document-compression: in both cases the improvement is roughly a factor 1.6. More importantly, if we compare compressed-pages vs compressed-ILs we find that the former are a factor 3.54 larger than the latter over the crawl-based ordering, and a factor 3.62 larger over the `url`-based ordering. These factors are not as big as *common belief* typically assumes! A more surprising result comes out if we consider the storage of *term*-positions in ILs, which is useful to implement phrase-queries or proximity-aware ranking functions [57]. Term-positions are usually assumed to take a factor 3-4 more than docIDs (cfr. [3, 17]); in accordance with this estimate, the MG4J search engine takes 7.21% for encoding the term positions of GOV2 [55]. These figures are larger than the space taken by the compressed raw pages (6.05% on GOV2 and 3.84% on GOV2-sort), so contradicting another common belief!

> Although documents are more informative than the corresponding IL storing `term-pos + docIDs + freq` info, they may be stored compressed in less space!

This suggests the need of improved compression techniques for IL-postings which use more information in addition to term frequencies and their distributions within documents.

Approach	block size	c/ratio	avg d/speed (min/max)
gzip	0.2	13.31	178.95 (43/358)
gzip	0.5	12.70	185.36 (54/347)
gzip	1	12.39	187.07 (61/324)
lzma	0.2	8.14	82.29 (7/299)
lzma	0.5	6.25	110.76 (7/314)
lzma	1	5.47	130.27 (7/313)
lzma	2	4.98	144.52 (7/313)
bmi+gzip	0.2	8.69	127.61 (41/555)
bmi+gzip	0.5	7.01	131.48 (42/262)
bmi+gzip	1	6.52	140.90 (40/210)
bmi+gzip	2	6.04	145.38 (44/202)
bmi+gzip	4	5.68	148.37 (46/200)
CSA-32	200	42.66	2.12 (1.50/2.42)
CSA-64	200	30.16	1.93 (1.14/2.37)
CSA-256	200	20.60	1.50 (0.5/2.1)
CSA-1024	200	23.46	0.69 (0.5/2.1)

Table 5: Performance on page decoding in internal memory over UK50 collection when URL-sorted. Decompression speed is given in MB per second. Min/max decompression speed are rounded, for readability, to the closest integer except for CSA where two decimal digits are preserved for details.

3. FAST ACCESS TO COMPRESSED DATA

Let us now turn our attention to the second scenario: a Web-storage system distributed among the internal memory of (possibly) many PCs [2, 4] that must guarantee not only compression efficacy, but also support the fast decompression of *individual* Web pages. We can devise two solutions.

The classic solution (e.g. Lucene [23] and MG4J [11]) consists of using a compressor over *small* blocks of pages so that, once a page is requested, its block (residing in-memory) is *fully* decompressed and the requested page is then extracted. This induces an obvious trade-off between the size of the block and the time to decompress one single page belonging to that block: the larger is the block, the better should be the compression ratio, but the slower is the throughput in terms of (retrieved) *pages/sec*. Given our previous results, we choose as compressors: gzip (Lucene and MG4J), lzma (our top performer), and bmi+gzip (because of [20]).

A modern solution, mostly unexplored at the Web-scale, consists of using the *compressed self-indexes* as a storage-format that achieves (in theory) space-occupancy close to the kth order entropy of the indexed data and cost of extracting arbitrary portions of compressed data *independent* of the overall size of the index [45]. So we assume to build one compressed self-index per PC, and fit it in its internal memory. This way we may use *much larger* blocks, thus possibly catching more repetitiveness in the data than the classic scheme. Compressed self-indexes need to set some *parameters* that control the trade-off between the space occupancy of the index and its decompression throughput. Some recent experimental studies [31] have investigated this trade-off on *small* and *well-formed* textual datasets (e.g. Wikipedia and DBLP), that are hence very distant from the *noisy* but possibly *highly repetitive* data available on the Web. This is exactly what we aim at testing in this section by considering the best known compressed-index over English and XML files, namely CSA (according to [31]), and compare it against the compressed-storage schemes offering the best time/space trade-offs in Tables 1–3.

In Tables 5–6 we report the results of our experiments where we estimated the extraction throughput of all four approaches over two collections: one is URL-sorted, and the other is *randomly* permuted. The former collection aims at investigating a scenario in which *repetitiveness* is forced to occur at *small* distances, thus giving an advantage to the dictionary-based approaches that need to use small blocks; the latter collection aims at addressing a scenario in which repetitiveness is not expected to occur at small distances thus penalizing the dictionary-based approaches and highlighting the *robustness* of compressed self-indexes which use larger blocks.

Let us start from Table 5. For dictionary-based compressors we consider small blocks (up to few MBs), because the retrieval of a single Web-page needs the decoding of its entire block. We stop the testing when increasing the block size is not advantageous: namely, there is an un-significant change in decompression throughput and compression ratio. We note that (in-memory) decoding is very fast for dictionary-based compressors, being gzip about 0.77-2 times faster than lzma, but at the cost of a significantly worse compression performance. It is interesting to notice a large variance in decompression speed, which actually depends on the compressibility of the underlying block: the better is the block compression, the smaller is the number of produced phrases, and thus the smaller is the computational cost incurred in decoding them (these are mainly cache misses). Importantly, bmi+gzip turns out to be slightly worse than lzma in storage, but faster and more robust in decompression performance. The URL-based sorting of the Web pages allows to use very-small blocks still guaranteeing an effective compression ratio. This validates [20] as a good choice also when data has to fit into the internal-memory of the PC. As in Sect. 2.1, we argue that a careful engineering of dictionary-phrase encodings and cache deployment in gzip- and bmi-implementations may be crucial to achieve the throughput of few GB/sec claimed in [20].[13]

> On url-ordered pages stored in memory, grouped in small blocks, bmi+gzip offers the best space/time trade-off. There is large room for performance improvement, in the light of [20]'s results.

As far as CSA is concerned, we considered blocks of 200MB because this is the largest indexable block according to the software distributed in [32], and we experienced no significant variations in decompression speed for smaller blocks. We tested the CSA-performance by varying the parameter that controls the trade-off between the extraction speed from compressed data and index's space-occupancy (option -p). The larger is this parameter, the smaller is CSA's space occupancy, but the slower is its decoding speed. To deploy at the best the *random*-access to compressed data offered by CSA, we stored an explicit pointer to the starting byte of each page (i.e. a row number in the BWT-matrix [45]).

> On url-ordered pages, the compression ratio of CSA may be up to 8 times worse than lzma; its decoding speed is stable but much slower than dictionary-based compressors— i.e. up to two order of magnitudes.

[13] bmi+gzip-0.2 achieves peaks of 555 MB/sec in decompression on our PC. bmi+gzip-0.4 is not able to achieve the same throughput probably because of cache-misses.

Approach	block size	c/ratio	avg d/speed (min/max)
gzip	0.2	20.87	126.48 (56/357)
gzip	0.5	20.69	126.02 (60/329)
lzma	0.2	18.28	33.32 (7/500)
lzma	0.5	17.28	34.87 (6/313)
lzma	1	16.50	36.68 (8/208)
bmi+gzip	0.2	20.43	73.28 (31/202)
bmi+gzip	0.5	20.09	73.74 (35/179)
bmi+gzip	1	19.71	74.41 (35/177)
CSA-32	200	47.68	2.25 (1.6/2.5)
CSA-64	200	35.18	2.02 (1.6/2.2)
CSA-128	200	28.93	1.69 (1.6/1.9)
CSA-256	200	25.80	1.29 (1.1/1.4)
CSA-1024	200	18.43	0.81 (0.2/1.4)

Table 6: Performance on page decoding in internal memory over UK50 collection when `randomly` permuted. Decompression speed is given in MB per second. Min/max decompression speed are rounded, for readability, to the closest integer except for CSA where two decimal digits are preserved for details.

This seems to be a very negative result for this algorithmic technology! But, let us compare the space occupancy of bzip*-200 in Table 3 with CSA in Table 5: there is indeed a large difference. Given that compressed-indexes are bwt-based and still in their infancy, we expect a large room for space/time improvement. Our experimental results should stimulate further research in this setting, starting from [31] and/or the promising *word-based* approaches in [12, 13].

We repeated our experiments on UK50 with its pages *randomly* permuted. Table 6 reports the main figures of our experiments where we notice a few crucial things. Dictionary-based compressors get significantly worse both in compression ratio and decompression speed. As expected, bmi is no longer effective on small blocks because of the random page-permuting. This impacts also on the speed because of the increased number of parsed phrases to be decoded, and thus an increased number of cache misses in the decompression process. Conversely CSA performance is robust, and its compression ratio approaches the one of the other tools.

> On `randomly`-permuted pages, bmi is no longer an effective compression booster. Dictionary-based compressors get significantly worse both in compression ratio and decompression speed. Conversely, the compression ratio of CSA is stable and its gap in decompression-speed with the other approaches is reduced, but still large.

3.1 Some further comments

A superficial analysis of these experimental results could lead a reader to drop the use of compressed self-indexes from the Web-scale scenario. We rise two objections to this conclusion: (1) Compressed self-indexes are still in their infancy [12, 13, 31]: theoretical/engineering improvements might reduce the performance gap significantly; (2) Dictionary-based compressors need to decompress the *entire* block that includes the page to be decoded, whereas CSA can decode just that page. To see the consequences of (2) we proceed with a "back of the envelop" calculation relating decompression speed and block size.

Compressor	block size	GOV2	GOV2-sort
gzip	1	18.72	10.63
lzma	1	12.45	5.48
bmi+gzip	1	15.38	6.97
CSA-256	200	23.27	20.20
IL: docIDs + freq [58]		1.52–1.90	0.85–1.30

Table 7: Compression ratios and indexing space for the GOV2 collection, when considering random access to individual pages in internal memory.

Let B be the block-size of a dictionary-based compressor, P be the average page size in the Web collection[14], and d_A the decompression throughput of compressor A in bytes/sec. A dictionary-based compressor is faster than CSA whenever its decoding speed is larger than $\frac{B*d_{\mathsf{CSA}}}{P}$. Since currently $d_{\mathsf{CSA}} \approx 2$MB/sec this is indeed the case for our Web-collections, but new engineered solutions for compressed self-indexes could change this conclusion. More importantly, the previous formula suggests that for a collection of *small records*, such as email and log-collections, for which $P \ll B$, CSA could be the winning choice in terms of speed.

It is interesting to consider also the point of view of a search engine that needs to sustain q queries/sec, each returning s snippets (typically $s = 10$). From Tables 5–6 (where we take $B = 1$MB), we derive that dictionary-based compressors could sustain up to 15 query/sec and CSA up to 12.5 query/sec (per server). So, pretty much the same given our approximations! However, this would be one order of magnitude less than the throughput guaranteed by compressed-IL [58] (namely, hundreds queries/sec). To sustain that throughput one would need a page-storage scheme with an order of magnitude faster decompression (actually, what it is claimed in [20], but without giving details on their implementation). Hence, more algorithmic engineering is required on the open-source softwares available all around the net to achieve this performance.

Finally, as we did at the end of Sect. 2.4, we repeated some experiments on the GOV2 collection. Table 7 shows that compression ratios are similar to what we got for the `url`-sorted UK50 (cfr. Table 5). Decompression speeds agree with the ones reported in Table 5 and thus are not reported. As in Sect. 2.4, we compare the compressed storage of raw pages with lzma against the compressed storage of ILs [58], now assuming that everything fits in the internal memory of (possibly many) PCs. Because of the smaller block, we find a larger gap than before: about 7.28 for GOV2 and 5.10 for GOV2-sort. As in Sect. 2.4 we note that for GOV2-sort the space used for the compressed pages (5.48%) is less than the space required for the *term*-positions (7.21%) by MG4J [55].

For space reasons we cannot provide the results of many other experiments we conducted over the two Web-collections UK-full and GOV2. It goes without saying that changes to the options of the tested compressors, or other combinations of them, might (slightly) change some results, but the overall picture would not be very different. So we can safely conclude that it has become necessary to reconsider whether the choice of compressing Web collections using simple tools, such as gzip, is the best one: for future Web applications the open-source tools lzma and bmi and compressed self-indexes [31] are certainly worth considering.

[14]It is $P = 16$ KB in UK-collection and $P = 18$KB in GOV2.

4. REFERENCES

[1] Apache Foundation. Hadoop. http://hadoop.apache.org/.

[2] R. Baeza-Yates and R. Ramakrishnan. Data challenges at Yahoo! *EDBT*, 652–655, 2008.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*, Addison Wesley, 1999.

[4] L. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2), 2003.

[5] R. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. *WWW*, 131–140, 2007.

[6] J. Bentley and M. McIlroy. Data compression with long repeated strings. *Information Sciences*, 135(1-2):1–11, 2001.

[7] D. Blandford and G. Blelloch. Dictionaries using variable-length keys and data, with applications. *ACM-SIAM SODA*, 1–10, 2005.

[8] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[9] P. Boldi, M. Santini, and S. Vigna. Permuting Web Graphs. *WAW*, 116-126, 2009.

[10] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. *WWW*, 595–602, 2004.

[11] P. Boldi and S. Vigna. MG4J at TREC 2005. http://mg4j.dsi.unimi.it/.

[12] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. *ACM SIGIR*, 139–146, 2008.

[13] N. Brisaboa, A. Fariña, G. Navarro, A. Places, and E. Rodríguez. Self-indexing natural language. *SPIRE*, LNCS 5280, 2008.

[14] A. Broder. Identifying and filtering near-duplicate documents. *CPM*, LNCS 1848, 2000.

[15] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13), 1997.

[16] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. *WSDM*, 95–106, 2008.

[17] S. Büttcher and C. Clarke. Efficiency vs. Effectiveness in terabyte-scale information retrieval. *TREC*, 2005.

[18] S. Büttcher and C. Clarke. Index compression is good, especially for random access. *CIKM*, 761–770, 2007.

[19] M. Chan and T. Woo. Cache-based compaction: A new technique for optimizing web-transfer. *INFOCOM*, 1999.

[20] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[21] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, P. Raghavan. On compressing social networks. *KDD*, 219-228, 2009.

[22] F. Claude and G. Navarro. A fast and compact web graph representation. *SPIRE*, LNCS 4726, 118–129, 2007.

[23] D. Cutting. Apache Lucene. http://lucene.apache.org/.

[24] T. S. D. Trendafilov, N. Memon. Compressing file collections with a TSP-based approach. TR-CIS-2004-02, Polytechnic University, 2004.

[25] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. *ACM SoCG*, 253–262, 2004.

[26] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI*, 2004.

[27] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. *USENIX*, 113–126, 2003.

[28] P. Ferragina, T. Gagie, and G. Manzini. Lightweight Data Indexing and Compression in External Memory. arXiv:0909.4341, 2009.

[29] P. Ferragina, R. Giancarlo, and G. Manzini. The engineering of a compression boosting library: Theory vs practice in BWT compression. *ESA*, LNCS 4168, 756–767, 2006.

[30] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compressionoptimal linear time. *Journal of the ACM*, 52:688–713, 2005.

[31] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[32] P. Ferragina and G. Navarro. Pizza&Chili corpus home page. http://pizzachili.di.unipi.it/.

[33] P. Ferragina, I. Nitto, and R. Venturini. On optimally partitioning a text to improve its compression. *ESA*, LNCS 5757, 420–431, 2009.

[34] P. Ferragina, I. Nitto, and R. Venturini. On the bit-complexity of Lempel-Ziv compression. *ACM-SIAM SODA*, 2009.

[35] P. Ferragina and R. Venturini. Compressed permuterm index. *ACM SIGIR*, 535–542, 2007.

[36] D. Geer. Reducing the storage burden via data deduplication. *Computer*, 41(12):15–17, 2008.

[37] O.A. Hamid, B. Behzadi, S. Christoph, and M.R. Henzinger. Detecting the origin of text segments efficiently. *WWW*, 61–70, 2009.

[38] T. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. *WebDB*, 129–134, 2000.

[39] J. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Info. Theory*, 46(3):737–754, 2000.

[40] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey. Redundancy elimination within large collections of files. *USENIX*, 2004.

[41] M. Mahoney. Large text compression benchmark. http://www.cs.fit.edu/ mmahoney/compression/text.html.

[42] U. Manber. Finding similar files in a large file system. *USENIX*, 1–10, 1994.

[43] F. Menczer. Lexical and semantic clustering by web links. *Journal of the American Society for Information Science and Technology*, 55(14):1261–1269, 2004.

[44] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. *ACM SIGCOMM*, 181–194, 1997.

[45] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[46] Z. Ouyang, N. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of a collection of files. *WISE*, 257–268, 2002.

[47] M. Sahami and T. Heilman. A web-based kernel function for measuring the similarity of short text snippets. *WWW*, 377–386, 2006.

[48] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. *SIGMOD*, 76–85, 2003.

[49] D. Shkarin. PPM: One step to practicality. *IEEE Data Compression Conference*, 202–211, 2002.

[50] F. Silvestri. Sorting out the document identifier assignment problem. *ECIR*, LNCS 4425, 101–112, 2007.

[51] T. Suel and N. Memon. *Lossless Compression Handbook*, chapter "Algorithms for delta compression and remote file synchronization", Academic Press, 2002.

[52] T. Suel, P. Noel, and D. Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. *IEEE ICDE*, 153–164, 2004.

[53] D. Teodosiu, N. Bjorner, J. Porkka, M. Manasse, and Y. Gurevich. Optimizing File Replication over Limited-Bandwidth Networks using Remote Differential Compression. Microsoft Research TR-2006-157, 2006.

[54] A. Turpin, Y. Tsegay, D. Hawking, and H. Williams. Fast generation of result snippets in web search. *ACM SIGIR*, 127–134, 2007.

[55] S. Vigna. Personal Communication.

[56] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann Publishers, 1999.

[57] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. *ACM SIGIR*, 2009.

[58] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. *WWW*, pages 401–410, 2009.

[59] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. *WWW*, 387–396, 2008.