Compressed String Dictionaries

Nieves R. Brisaboa*[1], Rodrigo Cánovas†[2], Francisco Claude‡[3], Miguel A. Martínez-Prieto§†[2,4], and Gonzalo Navarro†[2]

[1] Database Lab, Universidade da Coruña, Spain
[2] Department of Computer Science, University of Chile, Chile
[3] School of Computer Science, University of Waterloo, Canada
[4] Department of Computer Science, Universidad de Valladolid, Spain

Abstract. The problem of storing a set of strings – a *string dictionary* – in compact form appears naturally in many cases. While classically it has represented a small part of the whole data to be processed (e.g., for Natural Language processing or for indexing text collections), recent applications in Web engines, RDF graphs, Bioinformatics, and many others, handle very large string dictionaries, whose size is a significant fraction of the whole data. Thus efficient approaches to compress them are necessary. In this paper we empirically compare time and space performance of some existing alternatives, as well as new ones we propose. We show that space reductions of up to 20% of the original size of the strings is possible while supporting dictionary searches within a few microseconds, and up to 10% within a few tens or hundreds of microseconds.

1 Introduction

String dictionaries arise naturally in a large number of applications. We associate them classically to Natural Language (NL) processing: finding the *lexicon* of a text corpus is the first step in analyzing it [25]. They also arise, together with *inverted indexes*, when indexing text collections formed by NL [2, 33].

In those NL applications, there has not been much concern about the size of the dictionary. This is because, in classical NL collections, the dictionary grows sublinearly with the text size: Heaps' law [19] establishes that in a text of length n, the dictionary size is $O(n^\beta)$, for some $0 < \beta < 1$ depending on the type of text. This β value is usually in the range 0.4–0.6 [2], and thus the dictionary of terabyte-size collections should occupy just a few megabytes and would easily fit in the main memory of a commodity PC.

Heaps' law, however, does not model well the reality of Web search engines. Web collections are much less "clean" than text collections whose content quality is carefully controlled. Dictionaries of Web crawls easily exceed

*Funded by Ministry of Science and Innovation of Spain (PGE and FEDER) TIN2009-14560-C03-02 and Xunta de Galicia ref. 09TIC060E.

†Funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

‡Funded by the David R. Cheriton scholarships program.

§Funded by Ministry of Science and Innovation of Spain, TIN2009-14009-C02-02.

the gigabytes, due to typos and unique identifiers that are taken as “words”, but also for “regular words” from multiple languages. The *ClueWeb09* dataset (`http://boston.lti.cs.cmu.edu/Data/clueweb09`; thanks to Leonid Boystov) is a real example which comprises close to 200 million different words obtained from 1 billion web pages on 10 languages. This results in a large dictionary of far more than 1GB.

Web graphs are another application where the size of the URL names, classically neglected, is becoming very relevant with the advances of the techniques that compress the graph topology. The nodes of a Web graph are typically the pages of a crawl, and the edges are the hyperlinks. Typically there are 15 to 30 links per page. Compressing Web graphs has been an area of intense study, as it permits caching larger graphs in main memory, for tasks like Web mining, Web spam detection, finding communities of interest, etc. [21, 9]. URL names are used to improve the mining quality [34, 27].

In an uncompressed graph, 15 to 30 links per page would require 60 to 120 bytes if represented as a 4-byte integer. This posed a more serious memory problem compared to the name of the URL itself once some simple compression procedure was applied to those names (such as Front-Coding, see Section 2). For example, Broder et al. [5] reports 27.2 bits per edge (bpe) and 80 bits per node (bpn). This means that each node takes around 400–800 bits to represent its links, compared to just 80 bits used for storing its URL. In the same way, an *Internet Archive* graph of 115M nodes and 1.47 billion edges required [31] 13.92 bpe plus around 50 bpn, so 200–400 bits are used to encode the links and only 50 for the URL. In both cases, the space required to encode the URLs was just 10%-25% of that required to encode the links. However, the advances in compressing the edges have been impressive in recent years, achieving compression ratios around 1–2 bits per edge [3, 1]. At this rate, the edges leaving a node require on average 2 to 8 bytes, compared to which the name of the URL certainly becomes an important part of the overall space.

Another application is Bioinformatics. Popular alignment software like BLAST [18] indexes all the different substrings of length q of a text, storing the positions where they occur in the sequence database. For DNA sequences $q = 11, 12$ is common, whereas for proteins they use $q = 3, 4$. Over a DNA alphabet of size 4, or a protein alphabet of size 20, this amounts to up to 200 million characters. Using a larger q would certainly allow one improve the quality in searching for conserved regions, but this is infeasible for memory constraints.

The emergent *Linked Data Project* (`http://linkeddata.org`) focuses on the publication of RDF (`http://www.w3.org/TR/rdf-syntax-grammar`) data and their connection between different data sources in the “Web of Data”. This movement results in huge and heterogeneous RDF datasets from diverse fields.

The dictionary is an essential component in the logical division of an RDF database [10]. However, its effective representation has not been studied in depth. Our experience with the tool *HDT-It!* (`http://code.google.com/p/hdt-it`) shows that the dictionary for dataset *DBpedia-en* (`http://downloads.dbpedia.org/3.5.1/en`) takes about 80% of the total size.

Finally, Internet routing poses another interesting problem on dictionary strings. Domain name servers map domain names to IP addresses, and routers map IP addresses to physical addresses. They may handle large dictionaries of domain names or IP addresses, and serve many requests per second.

This short tour over various example applications shows that handling very large string dictionaries is an important and pervasive problem. Curiously, we have not seen much research on compressing them, perhaps because a few years ago the space of these dictionaries was not a serious problem, and at most Front-Coding was sufficient. In this paper we study Front-Coding and other solutions we propose for compressing large string dictionaries, so that two basic operations are supported: (1) given a string, return its position in the dictionary or tell it is not in the dictionary; (2) given a position, retrieve its string content.

Our study over various application scenarios spots a number of known and novel alternatives that dominate different niches of the space/time tradeoff map. The least space-consuming variants perform efficiently while compressing the dictionary to 9%–22% of its original size, depending on the type of dictionary.

2 Basic Concepts and Related Work

Rank and select on bitmaps. Let $B[1,n]$ be a $0,1$ string (*bitmap*) of length n and assume there are m ones in the sequence. We define $rank_b(B,i)$ as the number of occurrences of bit b in $B[1,i]$ and $select_b(B,i)$ as the position of the *i-th* occurrence of b in B.

In this paper we will use two different succinct data structures (implementations available at `http://libcds.recoded.cl`) that answer *rank* and *select* queries. The first one, that we will refer to as *RG* [16], uses $(1+x)n$ bits to represent B. It supports *rank* using two random accesses to memory plus $4/x$ contiguous (i.e., cached) accesses. *Select* requires and additional binary search.

The second data structure, that we will call *RRR* [29], is a compressed bitmap that uses in practice about $\log\binom{n}{m}+(\frac{4}{15}+x)n$ bits (our logarithms are in base 2), answering *rank* within two random accesses plus $3+8/x$ accesses to contiguous memory, and *select* with an extra binary search. In practice this compresses the bitmap when $m < 0.2\,n$.

Huffman and Hu-Tucker codes. For compressing sequences, statistical methods assign shorter codes (i.e., bit streams) to more frequent symbols. Huffman coding [20] is the optimal code (i.e., it achieves the minimum length of encoded data) that is uniquely decodable. In this paper we use canonical Huffman codes [26], which have various advantages.

Hu-Tucker codes [22] are optimum among those that maintain the lexicographical order of the symbols. Two sequences encoded using Hu-Tucker can be lexicographically compared bytewise directly in encoded form. We use both codes in this paper, in some cases padding them (with zeros) to the next byte in order to simplify alignment and bytewise comparisons.

Hashing. Hashing [8] is a folklore method to store a dictionary of any kind. A *hash function* transforms the elements into indexes in a *hash table*, where the corresponding value is to be inserted or sought. A *collision* arises when two different elements are mapped to the same array cell. In this paper we use *closed hashing*: If the cell where an element is to be found is occupied, one successively probes other cells until finding a free cell (insertions and unsuccessful searches) or until finding the element (successful searches).

We will consider two policies to determine the next cells to probe when a collision is detected at cell x. *Double hashing* computes another hash function y that depends on the key and probes $x + y$, $x + 2y$, etc. modulo the table size. *Linear probing* is a simpler policy. It tries the successive cells of the hash table, $x + 1$, $x + 2$, etc. modulo the table size.

The *load factor* is the fraction of occupied cells, and it influences space usage and time performance. Using good hash functions, insertions and unsuccessful searches require on average $1/(1 - \alpha)$ probes with double hashing, whereas successful searches require $\ln(1/(1 - \alpha))/\alpha$ probes. Linear probing requires more probes on average: $(1 + 1/(1 - \alpha)^2)/2$ for insertions and unsuccessful searches, and $(1 + 1/(1 - \alpha))/2$ for successful searches. Despite its poorer complexities, we consider also linear probing because it has advantages on some compressed representations we try.

Front-coding. Front-coding [33] is the folklore compression technique for lexicographically sorted dictionaries. It is based on the fact that consecutive entries are likely to share a common prefix. Each entry in the dictionary is be differentially encoded with respect to the preceding one. Two values are stored: an integer that encodes the length of their common prefix, and the remaining suffix of the current entry.

To allow searches, Front-Coding partitions the dictionary into buckets, where the first element is explicitly stored and the rest are differentially encoded. This allows the dictionary to be efficiently searched using a two-step process: first, a binary search on the first entry of the buckets locates the candidate bucket, and second a sequential scan of this candidate bucket rebuilds each element on the fly and compares it with the query. The bucket size yields a time/space tradeoff.

Front-coding has been sucessfully used in many applications. We emphasize its use in *WebGraph* (`http://webgraph.dsi.unimi.it`) to encode URL dictionaries from Web graphs.

Compressed text self-indexes. A compressed text *self-index* takes advantage of the compressibility of a text $T[1, N]$ to represent it in space close to that of the compressed text, while supporting random access and search operations. More precisely, a self-index supports at least operations $extract(i, j)$, which returns $T[i, j]$, and $locate(p)$, which returns the positions in T where pattern p occurs.

There are several self-indexes [28, 11]. For this paper we are interested in particular in the *FM-index* family [12, 13], which is based on the *Burrows-Wheeler transform (BWT)* [6]. FM-indexes achieve the best compression among self-indexes and are very fast to determine whether p occurs in T. Many self-indexes are implemented in the *PizzaChili* site (`http://pizzachili.dcc.uchile.cl`).

The BWT of $T[1,N]$, $T^{bwt}[1,N]$, is a permutation of its symbols. If the *suffixes* $T[i,N]$ of T are sorted lexicographically, then $T^{bwt}[j]$ is the character preceding the jth smallest suffix. We use the BWT properties in this paper to represent a dictionary as the FM-index of a text T.

FM-indexes support two basic operations on T^{bwt}. One is the *LF-step*, which moves from $T^{bwt}[j]$ that corresponds to the suffix $T[i,N]$ to $T^{bwt}[j']$ that corresponds to the suffix $T[i-1,N]$ (or $T[N,N]$ if $i=1$), that is $j'=LF(j)$. The second is the *backward step*, which moves from the lexicographical interval $T^{bwt}[sp,ep]$ of all the suffixes of T that start with string x to the interval $T^{bwt}[sp',ep']$ of all the suffixes that start with cx, for a character c.

Grammar-based compression. Grammar-based compresson is about finding a small grammar that generates a given text [7]. These methods exploit repetitions in the text to derive good grammar rules, so they are particularly suitable for texts containing many identical substrings. Finding the smallest grammar for a given text is NP-hard [7], so grammar-based compressors look for good heuristics. We use Re-Pair [23] as a concrete compressor, as it runs in linear time and yields good results in practice.

Re-Pair finds the most-repeated pair xy in the text and replaces all its ocurrences by a new symbol R. This adds a new rule $R \rightarrow xy$ to the grammar. The process iterates until all remaining pairs are unique in the text. Then Re-Pair outputs the set of r rules and the compressed text, $\mathcal{C}$. We use a public implementation (`http://www.dcc.uchile.cl/gnavarro/software`) for the compressor; each value (elements of a rule and symbols in $\mathcal{C}$) is stored in $\log(\sigma+r)$ bits.

Variable-length and direct-access codes. Brisaboa et al. [4] introduce a symbol reordering technique called directly addressable variable-length codes (*DACs*). Given a concatenated sequence of variable-length codes, DACs reorder the target symbols so that direct access to any code is possible. The overhead is at most one bit per target symbol, which is not too much if the target alphabet is large.

All the first symbols of the codes are concatenated in a first array A_1. A bitmap B_1 stores one bit per code in A_1, marking with a 1 the codes of length more than 1. The second symbols of the codes of length more than one are concatenated in a second array A_2, with B_2 marking which are longer than two, and so on. To extract the ith code, one finds its first symbol in $A_1[i]$. If $B_1[i]=0$, we are done. Otherwise we continue in $A_2[rank_1(B_1,i)]$, and so on.

A variable-length coding we use in this paper (albeit not in combination with DACs) is Vbyte [32]. It is used to represent numbers of distinct magnitudes, where most are small. Vbyte partitions the bits into 7-bit chunks and reserves the last bit of each byte to signal whether the number continues or not.

Tries and the XBW. A trie is an edge-labeled tree where each path from the root to a leaf represents a string. Strings that share a common prefix share a corresponding common path from the root.

A trie can represent a dictionary in a natural way. Searching for a string in the dictionary corresponds to following the labeled edges according to the string characters. The number of the leaf would correspond to the *id* of the string (if the leaf exists), and it usually matches with the rank of that string in the set.

The main problem in practice is that tries tend to use much space; even when the space is linear, the constants are not negligible. To overcome this limitation, Ferragina et al. [15] proposed a compressed representation for trees that supports navigational operations, and subpath searching, using *rank* and *select* data structures for sequences. The representation, called *XBW*, corresponds to an extension of the BWT to trees.

3 Compressed Dictionary Representations

We describe now various approches for representing a dictionary within compressed space while solving two operations on it. The first operation, ***locate(p)***, gives a unique nonnegative identifier for the string p, if it appears in the dictionary; otherwise it returns -1. The second operation, ***extract(i)***, returns the string with identifier i in the dictionary, if it exists; otherwise returns $NULL$.

3.1 Hashing and Compression

We explore several combinations of hashing and compression. We Huffman-encode each string and the codes are concatenated in byte-aligned form. We insert the (byte-)offsets of the encoded strings in a hash table. The hash function operates over the encoded strings (seen as a sequence of bytes, that is, we compare them bytewise). This lowers the time to compute the function and to compare search keys (as the string is shorter). For searching we first Huffman-encode the search string and pad its bits to an integral number of bytes.

Our main hash function is a modified Bernstein's hash[5]. The second function for double hashing is the "rotating hash" proposed by Knuth [22, Sec. 6.4][6].

We concatenate the strings in the same order they are finally stored in the hash table. This improves locality of reference for linear probing, and gives other benefits, as seen later (in particular we easily know the length in bytes of each encoded string). We consider three variants to represent the hash table, and combine each of them with linear probing *(lp)* or double hashing *(dh)*.

The first variant, *Hash*, stores the hash table in classical form, as an array $H[1, m]$ pointing to the byte offset of the encoded strings. To answer *locate(p)* we proceed as usual, returning the offset of H where the answer was found, or -1 if not. To answer *extract(i)*, we simply decompress the string pointed from $H[i]$. Then with load factor $\alpha = n/m$ (n being the number of strings in the dictionary), the structure requires m integers in addition to the Huffman-compressed strings.

[5]`http://www.burtleburtle.net/bob/hash/doobs.html`. We initialize h as a large prime and replace the 33 by $2^{15}+1$, taking modulo the table size at each iteration.

[6]Precisely, the variant at `http://burtleburtle.net/bob/hash/examhash.html`. We also initialize h as a large prime.

The second variant, *HashB*, stores $H[1,m]$ in compact form, that is, removing the empty cells, in an array $M[1,n]$. It also stores an *RG*-encoded bitmap $B[1,m]$ that marks with a 1 the nonempty cells of H. Then $H[i]$ is empty if $B[i] = 0$, and if it is nonempty then its value is $H[i] = M[rank_1(B,i)]$. Now *locate(p)* returns positions in M, so our identifiers become contiguous in the range $[1,n]$, which is desirable. For *extract(i)* we simply decompress the string pointed from $M[i]$. The space of this representation is n integers plus $(1+x)m$ bits, where x is the parameter of bitmap representation *RG*. The n integers require $n \log N$ bits, where N is the total byte length of the encoded strings.

The price is in time, as each new probe requires an additional $rank$ on B. However, with linear probing, $rank$ needs to be computed only once, as the successive cells are also successive in M. We only need to access the bits of B to determine where is the next empty cell.

The third variant, *HashBB*, also stores M and B instead of H, but M is replaced by a second bitmap. Note that since we have reordered the codes according to where they appear in H (or M), the values in these arrays are increasing. Thus instead of M we store a second bitmap $Y[1,N]$, where a 1 marks the beginning of the codes. Then $M[i] = select_1(Y,i)$. Bitmap Y is encoded in compressed form (*RRR*). Now the $n \log N$ bits of M are reduced to $\log \binom{N}{n} + (\frac{4}{15} + x)N$ bits, which is smaller unless the encoded strings are long.

The price is, again, in time. Each access to M requires a *select* operation. Note that linear probing does not save us from successive *select* operations, despite the involved string being contiguous, because we have no way to know where a code ends and the next starts.

3.2 Front-Coding and Compression

We consider two variants of Front-Coding. *Plain Front-Coding* implements the original technique by using Vbyte to encode the length of the common prefix. The remaining suffix is terminated with a zero-byte. Only bytewise operations are needed to search. The block sizes are measured in number of strings, so *extract(i)* determines the appropriate block with a simple division, and then scans the block to find the corresponding string.

Hu-Tucker Front-Coding is similar, but all the strings and Vbyte codes are encoded together using a single Hu-Tucker code. The bucket starts with the Hu-Tucker code of the first string, which is padded to the next byte boundary and preceded by the byte length of the encoded string, in Vbyte form. This prelude enables binary searching the first strings without decompressing them. The rest of the bucket is Hu-Tucker-compressed and bit-aligned, and is sequentially decompressed when scanning the bucket, both for locating and for extracting. We use a pointer-based Hu-Tucker tree implementation.

3.3 FM-Index Based Representation

We use two FM-indexes from *PizzaChili*. They represent the BWT using a *wavelet tree* [17], whose bitmaps are represented using *RG* (version `SSA_v3.1`) or

RRR (version `SSA_RRR`). The former corresponds to the "succinct suffix array" [13], which achieves zero-order compression of T, and the second to the "implicit compression boosting" idea [24], which reaches higher-order compression. Both FM-index implementations support functions LF and BWS, as well as obtaining $T^{bwt}[j]$ given j, in time $O(\log \sigma)$, where σ is the alphabet size of T. We use the indexes with no extra sampling because we need only limited functionality.

We concatenate all the strings in lexicographic order, terminating each one with a special character, \$, that is lexicographically smaller than all the symbols in T (in practice \$ is the ASCII code zero, which is the natural string terminator). We also add \$ at the beginning of the sequence. Thus we can speak of the ith string in lexicographical or positional order, indistinctly.

Note that, when the suffixes of T are sorted lexicographically, the first corresponds to the final \$, and the next n correspond to the \$s that precede each dictionary string. Thus $T^{bwt}[1]$ is the final character of the nth dictionary string, and $T^{bwt}[i+2]$ is the final character of the ith string, for $1 \leq i < n$. Therefore *extract*(i) can be carried out by starting at the corresponding position of T^{bwt} and using LF-steps until reaching a \$. The $T^{bwt}[j]$ characters traversed spell out the desired dictionary string in reverse order.

To answer *locate(p)* we just need to determine whether $\$p\$$ occurs in T. Thus we start with $(sp, ep) = (1, n+1)$ and use $|p|+1$ backward steps until finding the lexicographical interval (sp', ep') of the suffixes that start with $\$p\$$. If $p$ exists in the dictionary and is the ith string, then $sp' = ep' = i+1$ and we simply return i; otherwise $sp' > ep'$ holds at some point in the process and we return -1.

3.4 Re-Pair Based Representation

We concatenate all the dictionary strings in lexicographic order and apply Re-Pair compression to the concatenation. However, we avoid forming rules that contain the string terminator. This ensures that each string is encoded with an integral number of symbols in $\mathcal{C}$ and thus decompression is fast.

Locating is done via binary search, where each dictionary string to compare must be decompressed first. We decompress the string only up to the point where the lexicographical comparison can be decided. For extraction we simply decompress the desired string.

For both operations we need direct access to the first symbol of the ith string in $\mathcal{C}$. Each compressed string can be seen as a variable-length sequence of symbols in $\mathcal{C}$, where they are concatenated. Thus we use the DAC representation on those sequences. This gives fast direct access to the ith string, at the price of 1.25 bits per symbol: we use *RG* representation with 25% overhead.

3.5 XBW Trie Representation

If the trie has N nodes, the XBW consists of a sequence $S_\alpha[1, N+n]$ of labels (each leaf is identified with a label \$ leading to it) plus a bitmap $S_{last}[1, N+n]$ with n bits set. We represent S_α using wavelet trees and, as for the FM-Index, represent their bitmaps (and S_{last}) using *RG* or *RRR*.

For locating, we use operation `GetChildren` [15] to find the leaf. Then we map the leaf x to an identifier in the range $[1, n]$ with $rank_{\$}(S, x)$. For extracting, we start from the leaf and use `GetParent` [15] to obtain all the string characters.

4 Experimental Results

We consider four dictionaries that are representative of relevant applications:

Words comprises all the different words with at least 3 ocurrences in the `ClueWeb09` dataset It contains 25,609,784 words and occupies 256.36 MB.

DNA all substrings of 12 nucleotides found in S. Paradoxus, known as the `para` dataset[7]. It contains 9,202,863 subsequences and occupies 114.09 MB.

URLs corresponds to a 2002 crawl of the `.uk` domain from the *WebGraph* framework. It contains 18,520,486 URLs and occupies 1.34 GB.

URIs contains all different URIs used in the *DBpedia-en* RDF dataset (blank nodes and literals excluded). It contains 30,176,012 URIs and takes 1.52 GB.

We use an Intel Core2 Duo processor at 3.16 GHz, with 8 GB of main memory and 6 MB of cache, running Linux kernel 2:6:24-28. We ran *locate* experiments for successful and unsuccessful searches. For the former we chose 10,000 dictionary strings at random. For the latter we chose other 1,000 strings at random and excluded them from the indexing. For *extract* we queried 10,000 random numbers between 1 and n. Each data point is the average user time over 10 repetitions.

Figure 4 shows our results. Most methods are drawn as a line that corresponds to their main space/time tuning parameter. On the left we show locate time for successful searches; the plots for unsuccessful searches are very similar and omitted for lack of space. On the right we show extraction times. Time is shown in microseconds and space as a percentage of the space required by concatenating the plain strings. Since, despite the advantages of linear probing in this scenario, double hashing was always better, we only plot the latter.

Front-Coding with Hu-Tucker compression shows to be an excellent choice in all cases, achieving good time performance and the least space usage (only beaten by *XBW* and, on URLs, by *Re-Pair*). The folklore *Front-Coding*, without compression, is almost everywhere dominated by the compressed variant.

The least space is always achieved by *XBW+RRR*, yet the time it achieves is significantly higher than the other approaches. The next best space, on URLs, is achieved by *Re-Pair*, which is much faster than *XBW* but still noticeably slower than compressed *Front-Coding*. On the shorter-string dictionaries (Words and DNA), *Re-Pair* does not compress well and compressed *Front-Coding* achieves the second-best space (with much better time than *XBW* variants).

HashBB performs better in space than *HashB* when the strings are short, otherwise the bitmap becomes too long. It is never, however, clearly the best alternative. *HashB* and *Hash* excell in time with short strings when much space is used (nearly 100%), yet *HashB* is never much better than *Hash*.

[7] `http://www.sanger.ac.uk/Teams/Team71/durbin/sgrp`

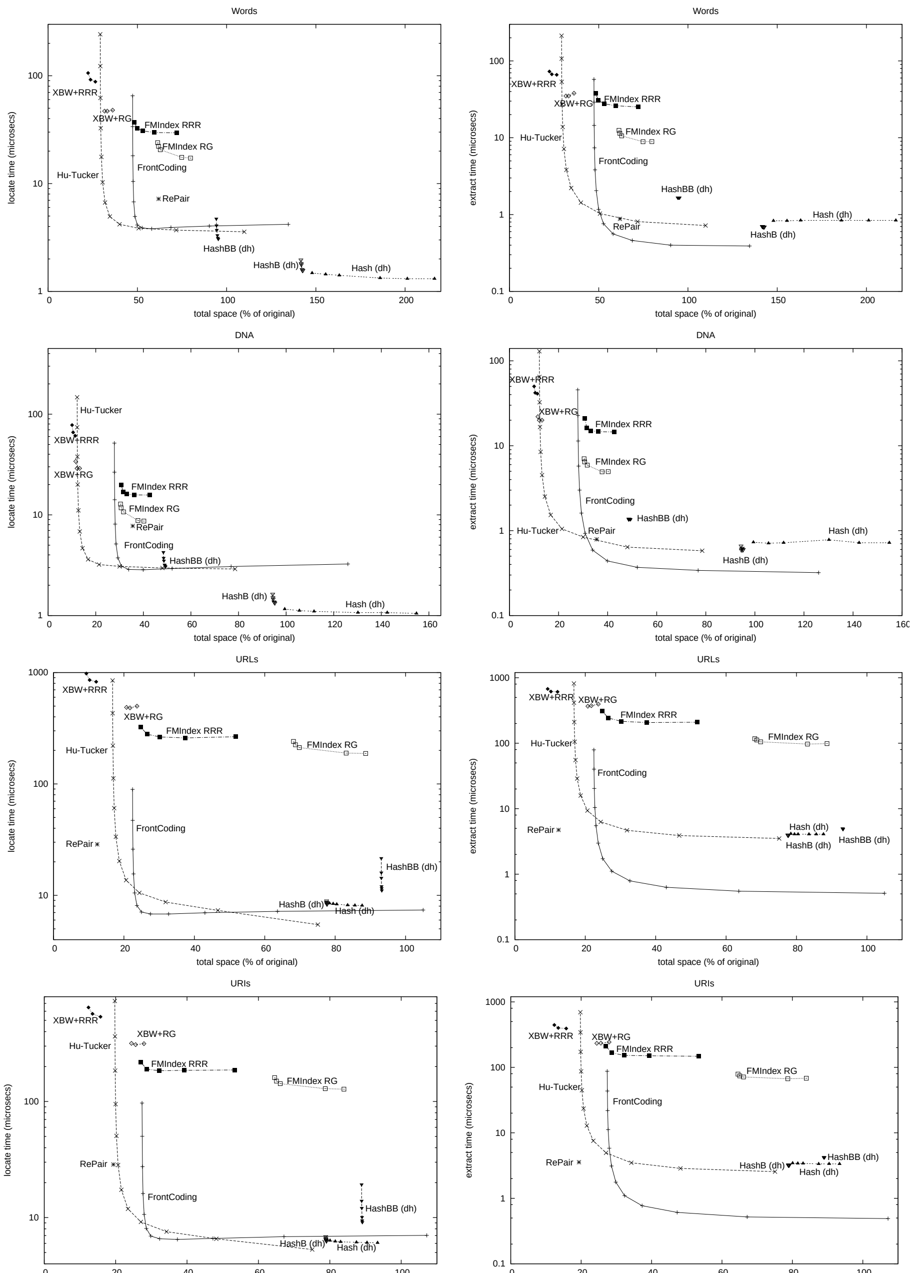

Fig. 1. Locate times (left) and extract times (right) for the different methods as a function of their space consumption.

For extracting, the map is dominated by *Front-Coding*, in plain or compressed form (the plain folklore variant is more relevant in this case). Still *Re-Pair* achieves less space on URLs, and *XBW* always requires the minimum space but the highest times.

5 Final Remarks

Prefix search, that is, finding the dictionary strings that start with a given pattern, is easily supported by the methods we have explored, except hashing. Other variants that can likewise be supported are of interest for Internet routing tables, e.g., find the dictionary string that is the longest prefix of the pattern.

Despite the *FM-index* and the *XBW* being the slowest solutions, they support other searches of interest, such as finding the dictionary strings that contain a substring, or that have a given prefix and a given suffix [14, 15]. They also support approximate searches [30].

We have reordered the strings at our convenience, but sometimes the order must be fixed. Hashing is easily adapted to any order (except the variant *HashBB*), but others would need an explicit permutation that would significantly increase the space. The *FM-index* and the *XBW* can use the LF-step mechanism to trade space for time and store just a sample permutation.

References

1. A. Apostolico and G. Drovandi. Graph compression by BFS. *Algorithms*, 2:1031–1044, 2009.
2. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, 1999.
3. P. Boldi and S. Vigna. The Webgraph framework i: Compression techniques. In *Proc. WWW*, pages 595–2562, 2004.
4. N. Brisaboa, S. Ladra, and G. Navarro. Directly addressable variable-length codes. In *Proc. SPIRE*, LNCS 5721, pages 122–130, 2009.
5. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A.Tomkins, and J. Wiener. Graph structure in the Web. *Comput. Netw.*, 33:309–320, 2000.
6. M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.
7. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005.
8. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2nd edition, 2001.
9. D. Donato, L. Laura, S. Leonardi, U. Meyer, S. Millozzi, and J. Sibeyn. Algorithms and experiments for the Webgraph. *J. Graph Algor. App.*, 10(2):219–236, 2006.
10. J.D. Fernández, M.A. Martínez-Prieto, and C. Gutierrez. Compact representation of large RDF data sets for publishing and exchange. In *Proc. ISWC*, LNCS 6496, pages 193–208 (part I), 2010.

11. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM JEA*, 13:article 12, 2009.
12. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.
13. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Alg.*, 3(2):article 20, 2007.
14. P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Trans. Alg.*, 7(1):article 10, 2010.
15. Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. FOCS*, pages 184–196, 2005.
16. R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Posters WEA*, pages 27–38, 2005.
17. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.
18. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge Univ. Press, 2007.
19. H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects.* Academic Press, 1978.
20. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.
21. J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The Web as a graph: Measurements, models, and methods. In *Proc. COCOON*, LNCS 1627, pages 1–17, 1999.
22. D.E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching.* Addison Wesley, 2007.
23. N.J. Larsson and J. A. Moffat. Offline dictionary-based compression. *Proc. of the IEEE*, 88:1722–1732, 2000.
24. V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. SPIRE*, LNCS 4726, pages 214–226, 2007.
25. C.D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing.* MIT Press, 1999.
26. A. Moffat and J. Katajainen. In-place calculation of minimum-redundancy codes. In *Proc. WADS*, LNCS 955, pages 393–402, 1995.
27. N. Nagwani. Clustering based URL normalization technique for Web mining. In *Proc. ACE*, pages 349–351, 2010.
28. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):article 2, 2007.
29. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA*, pages 233–242, 2002.
30. L. Russo, G. Navarro, A. Oliveira, and P. Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009.
31. T. Suel and J. Yuan. Compressing the graph structure of the Web. In *Proc. DCC*, pages 213–222, 2001.
32. H. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42:193–201, 1999.
33. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images.* Morgan Kaufmann, 1999.
34. M. Yin, D. Goh, E.-P. Lim, and A. Sun. Discovery of concept entities from Web sites using web unit mining. *Intl. J. of Web Inf. Sys.*, 1(3):123–135, 2005.