Practical Compressed String Dictionaries[☆]

Miguel A. Martínez-Prieto[*,a,1], Nieves Brisaboa[b,1], Rodrigo Cánovas[c], Francisco Claude[d,3], Gonzalo Navarro[e,2]

[a] *DataWeb Research, Department of Computer Science, University of Valladolid, Spain.*
[b] *Database Laboratory, University of A Coruña, Spain.*
[c] *NICTA Victoria Research Laboratory, Department of Computing and Information Systems (CIS), The Univerity of Melbourne, Australia.*
[d] *Escuela de Informática y Telecomunicaciones, Universidad Diego Portales, Chile.*
[e] *CeBiB — Center of Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile.*

Abstract

The need to store and query a set of strings – a *string dictionary* – arises in many kinds of applications. While classically these string dictionaries have accounted for a small share of the total space budget (e.g., in Natural Language Processing or when indexing text collections), recent applications in Web engines, Semantic Web (RDF) graphs, Bioinformatics, and many others, handle very large string dictionaries, whose size is a significant fraction of the whole data. In these cases, string dictionary management is a scalability issue by itself. This paper focuses on the problem of managing large static string dictionaries in compressed main memory space. We revisit classical solutions for string dictionaries like hashing, tries, and front-coding, and improve them by using compression techniques. We also introduce some novel string dictionary representations built on top of recent advances in succinct data structures and full-text indexes. All these structures are empirically compared on a heterogeneous testbed formed by real-world string dictionaries. We show that the compressed representations may use as little as 5% of the original dictionary size, while supporting lookup operations within a few microseconds. These numbers outperform the state-of-the-art space/time tradeoffs in many cases. Furthermore, we enhance some representations to provide prefix- and substring-based searches, which also perform competitively. The results show that compressed string dictionaries are a useful building block for various data-intensive applications in different domains.

Keywords: *Compressed string dictionaries, text processing, text databases, compressed data structures.*

[☆] A preliminary version of this paper appeared in *Proc. 10th International Symposium on Experimental Algorithms (SEA), pages 136–147, 2011.*

[*] Corresponding author

Email addresses: migumar2@infor.uva.es (Miguel A. Martínez-Prieto), brisaboa@udc.es (Nieves Brisaboa), rcanovas@student.unimelb.edu.au (Rodrigo Cánovas), fclaude@recoded.cl (Francisco Claude), gnavarro@dcc.uchile.cl (Gonzalo Navarro)

[1] Funded by the Spanish Ministry of Economy and Competitiveness: TIN2013-46238-C4-3-R, and ICT COST Action KEYSTONE (IC1302)

[2] Funded with basal funds FB0001, Conicyt, Chile.

[3] Funded in part by Fondecyt Iniciación 11130104.

1. Introduction

A *string dictionary* is a data structure that maintains a set of strings. It arises in classical scenarios like Natural Language (NL) processing, where finding the *lexicon* of a text corpus is the first step in analyzing it [56]. They also arise as a component of *inverted indexes*, when indexing NL text collections [79, 19, 6]. In both cases, the dictionary comprises all the different words used in the text collection. The dictionary implements a bijective function that maps strings to identifiers (IDs, generally integer values) and back. Thus, a string dictionary must provide, at least, two complementary operations: (i) `string-to-ID` *locates* the ID for a given string, and (ii) `ID-to-string` *extracts* the string identified by a given ID.

String dictionaries are a simple and effective tool for managing string data in a wide range of applications. Using dictionaries enables replacing (long, variable-length) strings by simple numbers (their IDs), which are more compact to represent and easier and more efficient to handle. A compact dictionary providing efficient mapping between strings and IDs saves storage space, processing and transmission costs, in data-intensive applications. The growing volume of the datasets, however, has led to increasingly large dictionaries, whose management is becoming a scalability issue by itself. Their size is of particular importance to attain the optimal performance under restrictions of main memory.

This paper focuses on techniques to compress string dictionaries and the space/time tradeoffs they offer. We focus on *static* dictionaries, which do not change along the execution. These are appropriate in the many applications using dictionaries that either are static or are rebuilt only sparingly. We revisit traditional techniques for managing string dictionaries, and enhance them with data compression tools. We also design new structures that take advantage of more sophisticated compression methods, succinct data structures, and full-text indexes [62]. The resulting techniques enable large string dictionaries to be managed within compressed space in main memory. Different techniques excel on different application niches. The least space-consuming variants operate within microseconds while compressing the dictionary to as little as 5% of its original size.

The main contributions of this paper can be summarized as follows:

1. We present, as far as we know, the most exhaustive study to date of the space/time efficiency of compressed string dictionary representations. This is not only a survey of traditional techniques, but we also design novel variants based on combinations of existing techniques with more sophisticated compression methods and data structures.

2. We perform an exhaustive experimental tuning and comparison of all the variants we study, on a variety of real-world scenarios, providing a global picture of the current state of the art for string dictionaries. This results in clear recommendations on which structures to use depending on the application.

3. Most of the techniques outstanding in the space/time tradeoff turn out to be combinations we designed and engineered, between classical methods and more sophisticated compression techniques and data

structures. These include combinations of binary search, hashing, and Front-Coding with grammar-based and optimized Hu-Tucker compression. In particular, uncovering the advantages of the use of grammar compression for string dictionaries is an important finding.

4. We create a C++ library, `libCSD` (*Compressed String Dictionaries*), implementing all the studied techniques. It is publicly available at `https://github.com/migumar2/libCSD` under GNU LGPL license.

5. We go beyond the basic `string-to-ID` and `ID-to-string` functionality and implement advanced searches for some of our techniques. These enable *prefix*-based searching for most methods (except Hash ones) and *substring* searches for the FM-Index and XBW dictionaries.

The paper is organized as follows. Section 2 provides a general view of string dictionaries. We start describing various real-world applications where large dictionaries must be efficiently handled, then define the notation used in the paper, and finally describe classical and modern techniques used to support string dictionaries, particularly in compressed space. Section 3 provides the minimal background in data compression necessary to understand the various families of compressed string dictionaries studied in this paper. Section 4 describes how we have applied those compression methods so that they perform efficiently for the dictionary operations. Sections 5 to 9 focus on each of the families of compressed string dictionaries. Section 10 provides a full experimental study of the performance of the described techniques on dictionaries coming from various real-world applications. The best performing variants are then compared with the state of the art. We find several niches in which the new techniques dominate the space/time tradeoffs of classical methods. Finally, Section 11 concludes and describes some future work directions.

2. String Dictionaries

2.1. Applications

This section takes a short tour over various example applications where handling very large string dictionaries is a serious issue and compression could lead to considerable improvements.

NL Applications. It is the most classic application area of string dictionaries. Traditionally, the size of these dictionaries has not been a concern because classical NL collections were carefully polished to avoid typos and other errors. On those collections, Heaps [44] formulated an empirical law establishing that, in a text of length n, the dictionary grows sublinearly as $O(n^\beta)$, for some $0 < \beta < 1$ depending on the type of text. Value β value is usually in the range 0.4–0.6 [6], so the dictionary of a terabyte-size collection would occupy just a few megabytes and easily fit in any main memory. Heaps' law, however, does not model well the dictionaries used in other NL applications. The use of string dictionaries in Web search engines or in Machine Translation (MT) systems are two well-known examples:

- Web collections are much less "clean" than text collections whose content quality is carefully controlled. Dictionaries of Web crawls easily exceed the gigabytes, due to typos and unique identifiers that are taken as "words", but also due to "regular words" from multiple languages. The *ClueWeb09* dataset[4] is a real example that comprises close to 200 million different words obtained from 1 billion Web pages on 10 languages. Such a dictionary uses well above a gigabyte of memory.

- The success of a statistical MT system depends on the information stored in its "translation table". This table stores the translation data for two given languages: Each entry records pairs of word sequences conveying the same meaning in each language (and also some statistical information). Using longer sequences leads to better translation quality, but the combination of large collections and long sequences quickly renders the table unwieldy [20]. Therefore, in practice the dictionary is limited to storing segments of up to q words, for some small value q. The work of Pauls and Klein [65], about compression and query resolution in N-gram language models, is a good example of the need to compress string dictionaries in MT applications.

Web graphs. It is another application area where the size of the URL names, traditionally neglected, is becoming very relevant thanks to the improvements in the compression of 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. [49, 24, 72]. In several cases, the URL names are used to improve the mining quality [80, 61].

In an uncompressed graph, 15 to 30 links per page would require 60 to 120 bytes if represented as 4-byte integers. This posed a more serious memory problem than the name of the URL itself once some simple compression procedure was applied to those names (such as Front-Coding, see Section 6). For example, Broder et al. [17] report 27.2 bits per edge (bpe) and 80 bits per node (bpn), which means that each node takes around 400–800 bits to represent its links, compared to just 80 bits used for storing its URL. Similarly, an *Internet Archive* graph of 115M nodes and 1.47 billion edges required 13.92 bpe plus around 50 bpn [76], 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 edge compression have been impressive in recent years, achieving around 1–2 bits per edge [12, 5, 2, 11, 40]. 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.

Semantic Web. The so-called Web of Data is the modern materialization of the basic principles of the Semantic Web [10]. It interconnects RDF [57] datasets from diverse fields of knowledge into a cloud of data-to-data hyperlinks. As the Web of Data grows in popularity, more data are linked together and larger

[4] `http://boston.lti.cs.cmu.edu/Data/clueweb09`

datasets emerge. String dictionaries are massively used in this scenario for reducing storage and exchange costs [28], but also to simplify query processing [63]. Semantic data management involves handling three specific dictionaries, one for each term class in RDF: *URIs*, *blank nodes*, and *literal* values. A recent paper [58] analyzes the impact of RDF dictionaries, reporting that their plain representation takes up to 3 times more space than the inner dataset graph structure.

Bioinformatics. Another application of string dictionaries is Bioinformatics. Popular alignment softwares like BLAST [43] index all the different substrings of length k (called k-mers) of a text, storing the positions where they occur in the sequence database. The information on all the k-mers is also used for genome assembly. Common values of k are 11 or 12 for DNA sequences, whereas for proteins they use $k = 3$ or 4. Over a DNA alphabet of size 4, or a protein alphabet of size 20, this amounts to up to 200 million characters. Managing such dictionaries within limited space is challenging [66, 70], and prevents the use of larger k values.

NoSQL Databases. The relational model has proven inadequate to address the requirements posed by Big Data management, and NoSQL (*Not only SQL*) databases have gained momentum in recent years. NoSQL encompasses a wide range of architectures and technologies, most of which use distributed computing. Therefore, query resolution depends much on transmission time. To reduce such time, data is returned as IDs instead of strings to reduce delays, which requires a centralized string dictionary that translates the final ID-based results to strings. Very large dictionaries are required for managing Big Data in NoSQL. Urbani et al. [77] study this problem in a MapReduce scenario managing Big Semantic Data, reporting significative scalability improvements on large scale RDF management by applying compression techniques.

Column-oriented databases use independent tables to store each different attribute, so very similar data records tend to be put together. This arrangement enables effective compression techniques for integers when data are represented as ID-sequences. Abadi et al. [1] report significant performance gains in C-Store by implementing lightweight compression schemes and operators that work directly on compressed data.

Internet routing. It poses another interesting problem on dictionary strings. Domain Name Servers map domain names to IP addresses. They may handle large dictionaries of domain names or IP addresses, and must serve request very fast. Another case is that of routers, which map IP addresses to physical addresses using extremely limited configurations in storage and processing power. Thus, space optimizations have a significant impact. For instance, mask-based operations could be resolved through specific prefix-based lookup within a compressed dictionary of IP addresses. Rétvári et al. [69] address this scenario by introducing a couple of compressed variants for managing the IP Forwarding Information Base (FIB). They report that FIBs are highly compressible, encoding 440K prefixes in 100-400 KBytes of memory, while lookup performance remains competitive.

Geographic Information Systems (GIS). Finally, GIS are another application managing a large number of strings. Managing, for example, the set of street names of a region for searching and displaying purposes, is a complex task within a limited-resource navigation system such as a smartphone or a GPS device, which in addition must download large amounts of geographic data through wireless connections.

2.2. Basic Definitions

A *string dictionary* is a data structure that represents a sequence of n distinct strings, $\mathcal{D} = \langle s_1, s_2, \ldots, s_n \rangle$ and provides a mapping between numbers i and strings s_i. More precisely, string dictionaries provide two primitive operations:

- `string-to-ID` transformation: `locate(`p`)` returns i if $p = s_i$ for some $i \in [1, n]$; otherwise it returns 0.
- `ID-to-string` transformation: `extract(`i`)` returns the string s_i, for $i \in [1, n]$.

In addition to these primitives, some other operations can be useful in specific applications. When possible, we will enhance our dictionaries with location/extraction by *prefix* and by *substring*. Prefix-based operations are useful, for example, to handle stemmed searches [6] and autocompletions [7] in NL dictionaries, or to find the text sequences starting with a given sequence of words in statistical machine translation systems [51]. Substring searches arise, for example, in SPARQL `regex` queries [67], mainly used for full-text purposes in Semantic Web applications [3]. They are also useful on GIS, when searching entities by name. The operations can be formalized as follows:

- `locatePrefix(`p`)` returns $\{i,\ \exists y,\ s_i = py\}$, that is, the IDs of the strings starting with p. Note that this set is a contiguous ID range for lexicographically sorted dictionaries, which are particularly convenient for this query.
- `extractPrefix(`p`)` returns $\{s_i,\ \exists y,\ s_i = py\}$, that is, returns the strings instead of the IDs. It is equivalent to composing `locatePrefix(`p`)` with individual `extract(`i`)` operations, but it can be carried out more efficiently on lexicographically sorted dictionaries.
- `locateSubstring(`p`)` returns $\{i,\ \exists x, y,\ s_i = xpy\}$, that is, the IDs of strings that contain p. It is very similar to the problem solved by full-text indexes.
- `extractSubstring(`p`)` returns $\{s_i,\ \exists x, y,\ s_i = xpy\}$, and is equivalent to running `locateSubstring(`p`)` followed by individual `extract(`i`)` operations.

Substring-based operations can be generalized to more complex ones, such as regular expression searching and approximate searching [14]. Other related search problems arise in Internet routing, where we want to find the longest s_i in the dictionary that is a prefix of a given address p.

We conclude with a few technical remarks. We will assume the strings s_i are drawn from a finite alphabet Σ of size σ. We serialize $\mathcal{D}$ as a text $\mathcal{T}_{dict}$, which concatenates all the strings appending a special symbol

\$ to them (\$ is, in practice, the ASCII zero code, the natural string terminator), that is $\mathcal{T}_{dict}[1, N] = s_1\$s_2\$ \ldots s_n\$$. Since the ID values are usually unimportant, $\mathcal{T}_{dict}$ is assumed to be in lexicographic order unless otherwise indicated. Thus, we can speak of the ith string in lexicographical or positional order, indistinctly, and this arrangement is convenient in many cases.

The previous concepts are illustrated using the set of strings $\{\texttt{alabar}, \texttt{a}, \texttt{la}, \texttt{alabada}, \texttt{alabarda}\}$, with $n = 5$ words. These strings are reordered into $\mathcal{D} = \{s_1 = \texttt{a}, s_2 = \texttt{alabada}, s_3 = \texttt{alabar}, s_4 = \texttt{alabarda}, s_5 = \texttt{la}\}$, serialized into the text $\mathcal{T}_{dict} = \texttt{a\$alabada\$alabar\$alabarda\$la\$}$, of length $N = 29$.

Finally, all the logarithms used in this paper are in base 2.

2.3. Related Work

The most basic approach to handle a string dictionary of n strings of total length N over an alphabet of size σ is to store $\mathcal{T}_{dict}$ plus an array of n pointers to the beginnings of the strings. This arrangement requires $N \log \sigma + n \log N$ bits of space and supports `locate(`p`)` in $O(p \log n)$ time, whereas `extract(`i`)` is done in optimal time $O(|s_i|)$. Classical hashing schemes increase the space to $N \log \sigma + O(n \log N)$ bits, and in exchange reduce locating time to $O(p)$ on average. Perfect hashing makes that time worst-case. Another classical structure, using $O(N \log \sigma + n \log N)$ bits, is the *trie* [50]. The trie is a digital tree where each string s_i can be read in a root-to-leaf path, and therefore one can locate p by following its symbols downwards from the root.

Those classical structures, all designed for use in main memory, use too much space when the dictionary becomes very large. A solution is to resort to secondary memory, where the best data structure is the String B-tree [30]. While it searches in optimal I/O time $O(p/B + \log_B n)$, where B is the disk block size, any access to secondary memory multiplies main memory access times by orders of magnitude. In this paper we explore the alternative path of compressing the dictionary, so that much larger dictionaries can be maintained in main memory and the access times remain competitive.

One relatively obvious approach to reducing space is to compress the strings. To be useful for implementing string dictionaries, such compression must allow for fast decompression of the individual strings. An appropriate compressor for this purpose is Huffman coding [46]; another is a grammar compressor like Re-Pair [52]. When the strings are sorted in lexicographic order, another powerful compression technique is Front-Coding [79], in which each string omits the prefix it shares with the previous one.

Throughout the paper, we combine, engineer and tune several variants of those ideas. In Section 5 we explore the combination of hashing with Huffman or grammar compression of the strings. In Section 6 we combine Front-Coding with binary-searchable Huffman or grammar compression. In Section 7 we combine plain binary search with grammar compression. In Section 8 we adapt a compressed full-text index [62] for dictionary searches. Finally, in Section 9 we use a compressed data structure [32] for representing tries.

There has been some recent research on the specific problem of compressing string dictionaries for main memory, mostly related to compressing the trie. Grossi and Ottaviano [42] introduce a new succinct data

structure inspired in the *path decomposition* approach [31]. In short, it transforms the trie into a new tree-shaped structure in which each node represents a *path* in the original trie. This solution excels in space, while remaining highly competitive for `locate` and `extract`. Arz and Fischer [4] adapt the LZ78 parsing [81] to operate on string dictionaries, and use the resulting LZ-trie as a basis for building a compressed structure. The basic idea is to re-parse the strings for obtaining a more compact trie in which phrases can be used multiple times. The structure includes an additional trie for the phrases. This proposal is implemented in two complementary ways, one using path decomposition and another performing Front-Coding compression. Both techniques are also implemented on an *inverted* parsing, where they run LZ78 from right to left and then perform a left-to-right parsing with a trie built on the inverted phrases. These techniques display better space/time tradeoffs on highly repetitive dictionaries. In these cases, they often achieve better compression ratios and, in general, report competitive times.

3. Data Compression and Coding

Data compression [74] studies the way to encode data in less space than that originally required. We consider compression of sequences and focus on *lossless* compression, which allows reconstructing the exact original sequence. We only cover the elements needed to follow the paper.

Statistical Compression. A way to compress a sequence is to exploit the variable frequencies of its symbols. By assigning shorter codewords to the most frequent symbols and replacing each symbol by its codeword, compression is achieved (more with increasingly biased symbol distributions). To be useful, it must be possible to distinguish the codewords from their concatenation, and to be efficient, it must be possible to tell where the first codeword ends as soon as we read its last bit. Such codes are called *instantaneous*. To be instantaneous, it is necessary and sufficient that the code is a *prefix code*, that is, no code is a prefix of another. *Huffman* [46] gave an algorithm for obtaining *optimal* (i.e., minimizing the average code length) prefix codes given a frequency distribution. There are many possible Huffman codes for a given distribution, all of which are optimal. One of those, the Canonical Huffman codes [75], can be decoded particularly efficiently [53]. We use such codes in this paper.

Huffman coding does not retain the lexicographic order of the symbols in the resulting codes. The *Hu-Tucker* code [45, 50] is the optimal among those that do. That is, if symbol x precedes y, the binary codeword for x must be lexicographically smaller than that for y. This feature allows two Hu-Tucker encoded sequences to be efficiently compared bytewise in compressed form.

Variable-Length and Direct-Access Codes. Variable-length codes, as explained above, are key for statistical data compression. Albeit using bit-sequences for the codes yields the minimum space, using byte sequences is a competitive choice on large alphabets. Such byte-codes are faster to handle because they avoid expensive bit manipulations.

Variable-length byte sequences are also used to encode integers of varying sizes, so as to use fewer bytes for the smaller numbers. *Variable byte* (Vbyte) coding [78] is a folklore byte-oriented technique used in information retrieval applications. In this paper we use byte-sized chunks ($b = 8$ bits per chunk) in which the highest bit (called the flag bit) indicates if the chunk is the last in the represented number, and the remaining $b - 1$ bits encode the binary representation of the number. For instance, the binary encoding of 824, takes $\lceil \log 824 \rceil = 10$ bits (1100111000). Its Vbyte representation uses 2 chunks: the first one starts with **1** (because it is not the final chunk) and stores the most significant bits (**1**0000110), whereas the second chunk (starting with **0** since it is the last chunk) stores the least significant bits (**0**0111000). Vbyte can be generalized to use an arbitrary number of bits b, to best fit the distribution of the numbers.

A problem with variable-length representations is how to access the code of the ith symbol directly (i.e., without decoding the previous $i - 1$ symbols). Brisaboa et al. [16] introduce a chunk reordering technique called *Directly Addressable Codes (DACs)*, which allows such direct access. DACs use a tiered representation of the chunks. The first level concatenates the first chunks of all the codes into a sequence A_1, concatenating separately the flag bits into a bit sequence B_1. The second level stores A_2 and B_2 for the codes that have two or more chunks, and so on. To retrieve the i-th code, one finds its first part in $A_1[i]$. If $B_1[i] = 0$, we are done. Otherwise, the process continues accessing the second level, and so on. To navigate across levels one needs to perform `rank` operations (see below) on the bit sequences B_k.

BITSEQUENCES. Binary sequences (bitsequences) are the basic block of many succinct data structures and indexes. A bitsequence $B[1, n]$ stores a sequence of n bits and provides two basic operations:

- $\texttt{rank}_a(B,\texttt{i})$ counts the occurrences of the bit a in $B[1, i]$.
- $\texttt{select}_a(B,\texttt{i})$ locates the position of the i-th occurrence of a in B.

Bitsequence representations must also provide direct access to any bit; $\texttt{access}(B,i)$ returns $B[i]$.

In this paper we will use three different bitsequence representations (their implementations are available in the Compact Data Structures Library `libcds`[5]). The first one, that we refer to as RG [38], pays an additional overhead x on top of the original bitsequence size, so its total space is $(1 + x)\, n$ bits. It performs `rank` using two random accesses to memory plus $4/x$ contiguous (i.e., cached) accesses, whereas `select` requires an additional binary search. The second one, referred to as RRR [68], compresses the bitsequence to about $\log \binom{n}{m} + (\frac{4}{15} + x)n$ bits, where m is the number of 1s in B. It answers `rank` within two random accesses plus $3 + 8/x$ accesses to contiguous memory, and `select` with an extra binary search. In practice, RRR achieves compression when the proportion of 1s in the bitsequence is below 20% or above 80%. Finally, we consider the SDArray from Okanohara and Sadakane [64]. It needs $nH_0(B) + 2m + o(m)$ bits, and supports

[5] `https://github.com/fclaude/libcds`

`select` queries very efficiently, in constant time, and `rank` queries in time $O(\log(n/m))$. The SDArray achieves compression when the proportion of 1s in B is below 10%.

Compressed Text Self-Indexes. A compressed text *self-index* takes advantage of the compressibility of a text $\mathcal{T}[1, N]$ in order to represent it in space close to that of the compressed text. Self-indexes support, at least, two basic operations:

- `locate(p)` returns all the positions in $\mathcal{T}$ where pattern p occurs.
- `extract(i,j)` retrieves the substring $\mathcal{T}[i, j]$.

Therefore, a self-index stores the text and supports indexed searches on it, within a space proportional to its statistical entropy. Although the are several self-indexes [62, 29], in this paper we focus on the *FM-index* family [33, 34]. As described in Section 8, it takes advantage of the *Burrows-Wheeler transform (BWT)* [18] to build a highly compressed self-index.

Grammar-Based Compression. Grammar compression is a non-statistical method to compress sequences. The idea is to find a small context-free grammar that generates the text to compress [21]. 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 [21], but there exist several grammar-based compressors that achieve $O(\log N)$ approximation factors or less [71, 73, 59, 47], where N is the text length. We use Re-Pair [52] as our grammar compressor. Despite offering only weak approximation guarantees [21], Re-Pair achieves very good compression ratios in practice and builds the grammar in linear time. Like many other grammars-compression algorithms, Re-Pair guarantees convergence to the statistical entropy of the text [48].

Re-Pair finds the most repeated pair of symbols xy in the text, adds a new rule $R \to xy$ to the grammar, and replaces all of the occurrences of xy in the text by the nonterminal R. The process iterates (nonterminals can in turn form pairs) until all the pairs that remain in the text are unique. Then Re-Pair outputs the set of r rules and the reduced text, $\mathcal{C}$. Each value (an element of a rule or a symbol in $\mathcal{C}$) is represented using $\lceil \log(\sigma + r) \rceil$ bits.

4. Compressing the Dictionary Strings

To reduce space, we represent the strings of the dictionary, $\mathcal{T}_{dict}$, in compressed form. We cannot use any compression method, however, but have to choose one that enables fast decompression and comparison of individual strings. We describe three methods we will use in combination with the dictionary data structures. Their basics are described in Section 3. An issue is how to know where a compressed string $s_i\$$ ends in the compressed $\mathcal{T}_{dict}$. If we decompress s_i, we simply stop when we decompress the terminator \$. In the sequel we consider other cases, such as when comparing strings without decompressing them.

HUFFMAN COMPRESSION. After gathering the frequencies of the characters to be represented, we assign each character an optimal variable-length bit code. To simplify the operations we need on the dictionary structures, we make sure that the encoding of each new string starts at a byte-aligned boundary (padding with 0-bits), so that each string uses an integral number of bytes. When we compress a string s_i, we include its terminator symbol, so we compress $s_i\$$.

Although the zero-padding wastes some space, it allows pointers to the compressed strings to be byte-aligned, which in some cases recovers much of the space lost. It also permits faster processing. In particular, if we have compressed the search pattern $p\$$ into a sequence of bytes p' (using zero-padding as well), we only need to compare the strings $p'[1..|p'|]$ with $s'_i[1..|p'|]$ bytewise. If they are equal, this means that $s'_i[1..|p'|]$ encodes a string that starts with $p\$$, since the underlying bit-wise Huffman code is prefix free. Thus, the terminator indicates that the string encoded is precisely p. If, on the other hand, $p'[1..|p'|] \neq s'_i[1..|p'|]$, this means that s'_i encodes a string that does not start with p, due to the zero-padding. Such a bytewise comparison is much faster than decompressing s'_i and comparing s_i with p.

HU-TUCKER COMPRESSION. This compression will be used similarly to Huffman, including the zero-padding. Although slightly less space-efficient, Hu-Tucker compression has the advantage of permitting a bytewise lexicographical comparison, determining whether $p < s_i$, $p = s_i$, or $p > s_i$.

If the strings p' and s'_i coincide in their first $|p'|$ bytes, then they are equal, just as for Huffman coding. Otherwise a difference occurs before and we can use the lexicographic comparison. Note that, in the Hu-Tucker coding, the symbol \$ is encoded as a sequence of 0-bits (because \$ is the smallest character and thus it is assigned the lexicographically smallest code), thus a byte-based comparison works correctly even when one string is a prefix of the other.

Both Huffman and Hu-Tucker compressors require additional structures for fast encoding and decoding. For encoding, a simple symbol-codeword mapping table $\mathcal{M}$ is used. For decoding, we use two structures: (1) a pointer-based tree (i.e., a binary trie where each root-to-leaf path is a codeword and the leaf stores the corresponding symbol), that supports bit-wise decompression, and (2) a table H that supports chunk-based decompression [53]. Table H enables highly optimized decoding, by processing k bits at a time (we use $k = 16$ in practice). The table has 2^k rows, so that row x stores the result of Huffman or Hu-Tucker decoding binary string x: a sequence of decoded symbols, $H[x].dec$, and the number of unused bits at the end, $H[x].u$. The table allows decoding by reading k consecutive bits into a k-bit number x, outputting $H[x].dec$, and advancing the reading pointer by $k - H[k].u$. The tree is used when $H[x].u = k$, indicating that the first symbol to decode is already longer than k. In this case $H[x].dec$ points to the part of the decoding tree where decoding should continue at depth k. In fact, only those parts of the tree are stored.

RE-PAIR COMPRESSION. In the case of Re-Pair, we make sure that each string spans an integral number of symbols in $\mathcal{C}$ (the sequence of terminals and nonterminals into which $\mathcal{T}_{dict}$ is compressed). To do so, we add

unique separators after each terminator \$, to prevent Re-Pair from forming pairs that include them. The special symbols are removed after compression finishes.

We use a public implementation of Re-Pair[6] to obtain the set of r rules and the compressed sequence $\mathcal{C}$. The grammar is encoded in plain form in an array $\mathcal{R}[1, 2r]$, in which each cell uses $\lceil \log(\sigma + r) \rceil$ bits. More precisely, nonterminals will be identified with numbers in $[\sigma + 1, \sigma + r]$. Thus, a rule $X \to YZ$ will be stored as $\mathcal{R}[2(X - \sigma) - 1] = Y$ and $\mathcal{R}[2(X - \sigma)] = Z$. Sequence $\mathcal{C}$ will be regarded as a sequence of integers in $[1, \sigma + r]$ comprising n variable-length subsequences (i.e., the encodings of $s_1, s_2, \ldots, s_n$).

Compressing $p\$$ in order to compare it directly with a string is not practical with Re-Pair. We have to ensure that the rules are applied in the order they were created; otherwise a differently compressed string may result. Doing this requires a complicated preprocessing of p, so we instead decompress s_i before comparing it with p. Re-Pair is very fast at decompressing, so this is affordable.

5. Compressed Hashing Dictionaries (Hash)

Hashing [23] is a folklore method to store a dictionary of any kind (not only strings). In our case, a *hash function* transforms a given string into an index in a *hash table*, where the corresponding value is to be inserted or sought. A *collision* arises when two different strings are mapped to the same array cell.

In this paper, we use *closed hashing*: if the cell corresponding to an element is occupied by another, one successively probes other cells until finding a free cell (for insertions and unsuccessful searches) or until finding the element (for successful searches). We use *double hashing*[7] 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. Our main hash function is a modified Bernstein's hash[8]. The second function for double hashing is the "rotating hash" proposed by Knuth[9].

Let n be the number of elements stored and m the table size. The *load factor* $\alpha = n/m$ 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 $\frac{1}{1-\alpha}$ probes with double hashing, whereas successful searches require $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ probes.

Another alternative on a static set of strings is *perfect hashing* [37], which guarantees no collisions. In particular, it is possible to achieve *minimum perfect hashing*, which uses a table of size $m = n$ to store the n strings. Representing a minimum perfect hash function requires at least $n/\ln 2 \approx 1.44n$ bits [37]. There are practical implementations of minimal perfect hash functions achieving at most $2.7n$ bits [8, 13]. For our dictionaries, a problem of perfect hashing is that strings that do not belong to the set are hashed to

[6] `http://www.dcc.uchile.cl/gnavarro/software`

[7] We also considered *linear probing*, but it was outperformed by double hashing in our experiments [15].

[8] `http://burtleburtle.net/bob/hash/doobs.html`. We replace the value 33 by 63 to reduce hashing degradation on long strings, see `https://gist.github.com/hmic/1676398`.

[9] The variant at `http://burtleburtle.net/bob/hash/examhash.html`. We also initialize h as a large prime.

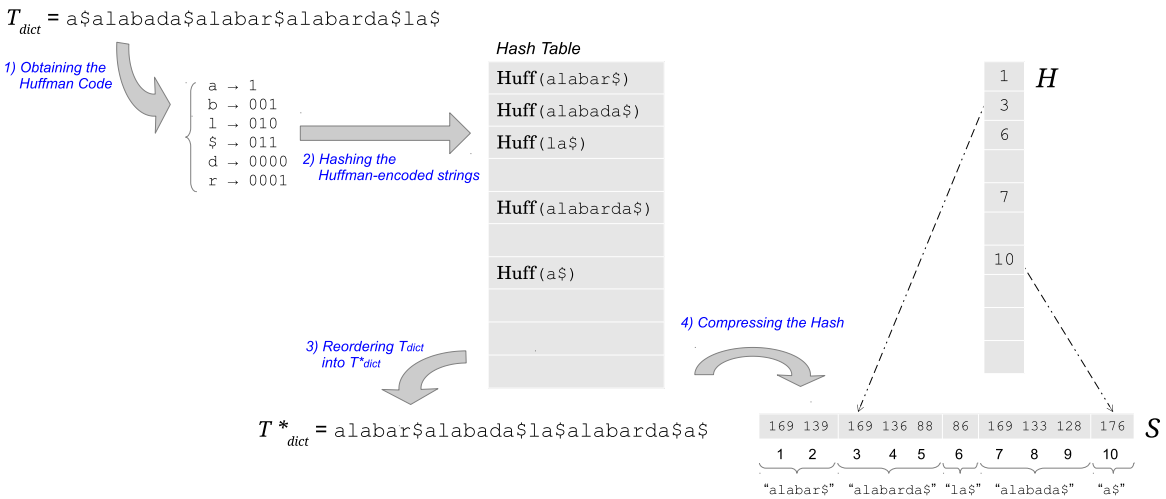

Figure 1: $\mathcal{T}_{dict}$ encoding based on hashing and Huffman compression.

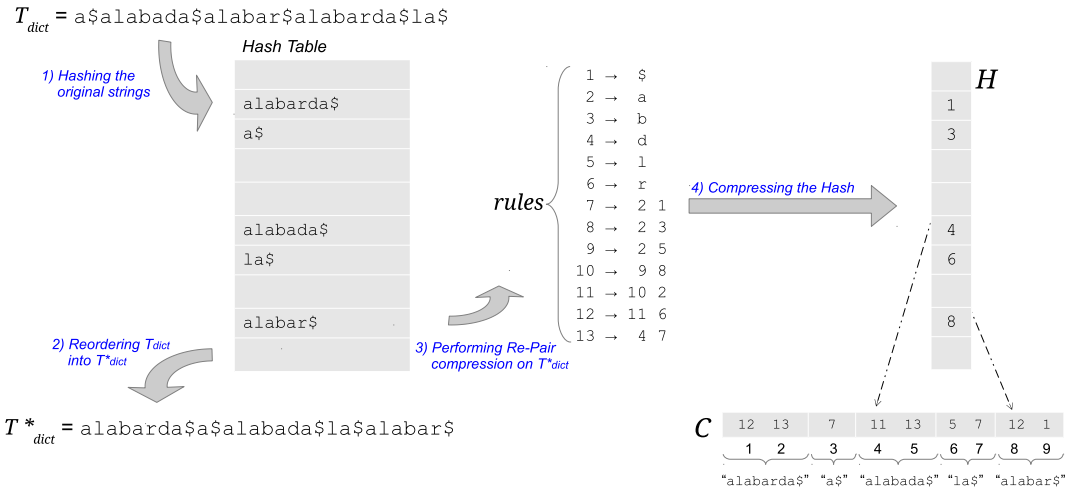

Figure 2: $\mathcal{T}_{dict}$ encoding based on hashing and Re-Pair compression.

arbitrary positions, and therefore we cannot avoid performing one string comparison to determine if the string p is present in the set or not. In Section 10 we show that our engineered double-hashing structures achieve basically the same performance of state-of-the-art perfect hashing implementations.

We propose four different hash-based techniques for managing string dictionaries, each of which can be combined with Huffman or with Re-Pair compression of the strings. First, $\mathcal{T}_{dict}$ is scanned string-by-string, and each string is stored in its corresponding cell in the hash table, $\mathcal{H}$. Now we reorder the original text $\mathcal{T}_{dict}$ into a new text $\mathcal{T}^*_{dict}$, in which the strings are concatenated in the same order they are stored in the hash table. The IDs are then assigned following this new ordering instead of the lexicographic one.

The process for Huffman compression is illustrated in Figure 1. Note that Huffman encoding of the strings is applied before hashing. For instance, the Huffman code of "`alabar$`" (referred to as `Huff(alabar$)` in the figure) is hashed to the position 1, "`alabada$`" to the position 2, and so on. This same order holds in

$\mathcal{T}^*_{dict}$ so "`alabar`" is now identified as 1, "`alabada`" as 2, etc. Figure 2 illustrates the process when Re-Pair is used for string compression. In this case, the hash function is applied to the original strings. For instance, "`alabar$`" is now hashed to the position 9, and "`alabada$`" to the position 6. $\mathcal{T}^*_{dict}$ is always built according to the hash ordering.

Finally, string $\mathcal{T}^*_{dict}$ is Huffman- or Re-Pair compressed, as described in Section 4. The resulting compressed sequence is called $\mathcal{S}$, of $|\mathcal{S}|$ bytes in case of Huffman or $|\mathcal{S}| = |\mathcal{C}|$ symbols in case of Re-Pair. What we encode in $\mathcal{H}$ is the offset in $\mathcal{S}$ of the corresponding strings. In Figure 1, $\mathcal{H}[2] = 3$, because the Huffman-compressed representation of "`alabada`" starts from $\mathcal{S}[3]$. In Figure 2, $\mathcal{H}[6] = 4$ because the Re-Pair compressed representation of "`alabada`" starts from $\mathcal{S}[4] = \mathcal{C}[4]$.

The search algorithm for `locate(`p`)` depends on the way we compress the strings. In the case of Huffman, we first compress the search key p\$ into p', padding it with 0-bits so that it occupies an integral number of bytes, $|p'|$. Then we use the hash functions to compute the corresponding positions to look for in $\mathcal{H}$. When $\mathcal{H}$ points to offset k in $\mathcal{S}$, we perform a direct byte-wise comparison between p' and $\mathcal{S}[k, k + |p'| - 1]$, as described in Section 4. In case of Re-Pair, we decompress from $\mathcal{S}[k \ldots]$ the string we need to compare p with. We can stop decompression as soon as the comparison with p is defined. In most cases, just probing one cell of $\mathcal{H}$ suffices to complete the search.

For `extract(`i`)` we simply decompress the string pointed from some cell of $\mathcal{H}$, with either method, until we decompress the terminator \$. The techniques relying on Huffman use the decoding table (see Section 4) to speed up extraction. None of these hash-based techniques provide prefix nor substring based searches.

The main difference between our four hash-based dictionaries is the way $\mathcal{H}$ is actually encoded. The simplest one, referred to as `Hash` (Section 5.1), stores $\mathcal{H}$ as is. The second technique, `HashB` (Section 5.2), removes the empty cells and stores $\mathcal{H}$ compactly. The third, `HashBB` (Section 5.3) introduces additional compression on the pointers stored in table $\mathcal{H}$. Finally, `HashDAC` (Section 5.4) uses DACs (Section 3) provide a directly-addressable representation of $\mathcal{S}$ and get rid of the pointers. Variants of the ideas behind `Hash` and `HashB` can be found in the literature [8], whereas `HashBB` is implicit in the encoding of Elias and Fano [25, 26]. Instead, our use of DACs in the variant `HashDAC` is novel.

5.1. Plain-table encoding (Hash)

The first technique stores the table in classical form, as an array $\mathcal{H}[1, m]$ in which each cell uses $\lceil \log |\mathcal{S}| \rceil$ bits. For `locate(`p`)` we proceed as above, until we find that $k = \mathcal{H}[j]$ points to the compressed string $s_i = p$. To complete the operation, we need a way to obtain the desired identifier i. Since we have reordered the IDs to match the order of the strings in $\mathcal{H}$, i is the number of nonempty cells of $\mathcal{H}$ up to position j.

To obtain i fast, we store a bitsequence $\mathcal{B}[1, m]$ in which $\mathcal{B}[i] = 1$ iff $\mathcal{H}[i]$ is nonempty. We compute $i = \texttt{rank}_1(\mathcal{B}, j)$ to complete the operation. This bitsequence is also useful for operation `extract(`i`)`: we decompress the sequence starting at position $k = \mathcal{H}[\texttt{select}_1(\mathcal{B}, i)]$ in $\mathcal{S}$.

The `Hash` dictionary requires, in addition to the compressed strings in $\mathcal{S}$, $m\lceil \log |\mathcal{S}| \rceil$ bits for $\mathcal{H}$, and $m(1+x)$ additional bits for the bitsequence $\mathcal{B}$ (using RG with $x = 0.05$ in our implementation).

5.2. Compressing the table (HashB)

The technique `HashB` stores the table in compact form (i.e., removing the empty cells) in a new table $\mathcal{H}'[1,n]$. The necessary mapping is provided by the same bitsequence $\mathcal{B}$. Now each access to $\mathcal{H}[j]$ during the execution of `locate(`p`)` must be remapped to $\mathcal{H}'[\texttt{rank}_1(\mathcal{B}, j)]$. To be precise, we first have to check whether $\mathcal{H}[j]$ is empty: if $\mathcal{B}[j] = 0$ we immediately know that p is not in the set. At the end, we simply return i when we find p pointed from $\mathcal{H}'[i]$. For `extract(`i`)` we just decompress from $\mathcal{S}[\mathcal{H}'[i] \ldots]$.

The space requirements are reduced with respect to those of `Hash`. In this case, table $\mathcal{H}$ is implemented in $n\lceil \log |\mathcal{S}| \rceil$ bits, instead of $m\lceil \log |\mathcal{S}| \rceil$.

5.3. Further compression (HashBB)

`HashBB` further reduces the space used by `HashB`. It exploits that offset values, within $\mathcal{H}$, are increasing. `HashBB` replaces the array $\mathcal{H}'[1,n]$ by a bitsequence $\mathcal{Y} = [1, |\mathcal{S}|]$, where $\mathcal{Y}[k] = 1$ iff $\mathcal{S}[k]$ stores the beginning of a compressed string (i.e., if $\mathcal{H}'[i] = k$ for some i). For instance, in Figure 1 the values are $1, 3, 6, 7, 10$, and thus $\mathcal{Y} = [1010011001]$, whereas in Figure 2 the values are $1, 3, 4, 6, 8$ and $\mathcal{Y} = [101101010]$.

For `locate(`p`)` we proceed as before, simulating the access to $\mathcal{H}'[i] = \texttt{select}_1(\mathcal{Y}, i)$. Similarly, for `extract(`$i$`)` we start decoding from $\mathcal{S}[\texttt{select}_1(\mathcal{Y}, i) \ldots]$.

Ths bitsequence $\mathcal{Y}$ is implemented differently when `HashBB` is combined with Huffman or RePair compression. In the first case, $\mathcal{Y}$ turns out to be sparser, because the the compressed strings are still long enough. In this case, SDArray turns out to be a good encoding for $\mathcal{Y}$, and in addition it is fast for the required `select` operation. In the second case, RePair reduces each string to a very short sequence of terminals and nonterminals, thus the resulting bitvectors are much denser. We choose RG for this case, ensuring a limited overhead of 0.05 bits per element in $\mathcal{Y}$.

5.4. Using direct access (HashDAC)

Bitsequence $\mathcal{Y}$ is used to mark the positions in $\mathcal{S}$ where the encoded strings s_i begin. We can get rid of this bitsequence by regarding the encoding s_i' of each $s_i\$$ as a variable-length sequence (of bytes in case of Huffman, of symbols in case of Re-Pair), and use DACs (Section 3) to provide access to those variable-length sequences. In exchange for the space reduction, DACs introduce some redundancy in the compression. Some of the redundancy can be removed thanks to the fact that, since DACs indicate where the encoded string ends, we do not need to compress $s_i\$$, but just s_i. This fact is exploited by the `HashDAC` variant using Re-Pair compression, but we keep the \$ for the one using Huffman because the terminator is necessary for efficient use of the decoding table.

A note on minimal perfect hashing. With such a hash function we can have table $\mathcal{H}'$ directly, without the use of bitsequence $\mathcal{B}$. On the other hand, a table similar to $\mathcal{B}$ is nevertheless stored internally in most implementations of minimal perfect hashing [8].

6. Front-Coding: Differentially Encoded Dictionaries

Front-Coding [79] is a folklore compression technique for lexicographically sorted dictionaries, for example it is used to compress the set of URLs in the WebGraph framework [12]. Front-Coding exploits the fact that consecutive entries are likely to share a common prefix, so each entry in the dictionary can be differentially encoded with respect to the preceding one. More precisely, each entry is represented using two values: an integer that encodes the length of the prefix it shares with the previous entry, and the remaining characters of the current entry. A plain Front-Coding representation, although useful for compression purposes, does not provide random access to arbitrary strings in the dictionary: we might have to decode the entire dictionary from the beginning in order to recover a given string.

To allow for direct access, we use a *bucketed* Front-Coding scheme. We divide the dictionary into buckets encoding b strings each. A bucket is represented as follows:

- The first string (referred to as *header*) is explicitly stored.
- The remaining $b-1$ strings (referred to as *internal strings*) are differentially encoded, each with respect to the previous one.

Now operation `extract(`i`)` is carried out as follows. First, we initialize the answer with the header of bucket $t = \lceil i/b \rceil$. Second, we sequentially decode the internal strings of the bucket, until obtaining the $((i-1) \bmod b)$th internal string (the 0th string is the header). The decoding effort can be made proportional to the size of the differentially-encoded bucket, not to its uncompressed size: if the current entry shares m characters with the previous one, we just rewrite its explicit characters starting at position $m+1$ of the string where we are computing the answer.

Operation `locate(`p`)` is carried out as follows. First, we binary search for p in the set of headers, obtaining the bucket where the answer must lie. Second, we sequentially decode the internal strings of the bucket, comparing each with p. A practical speedup is obtained as follows [60]. After having processed string s_i, we remember the length $0 \leq \ell < |p|$ of the longest common prefix between p and s_i (so they differ at $p[\ell+1] \neq s_i[\ell+1]$). Now, if the encoding of s_{i+1} indicates that it shares m characters with s_i, we do as follows: (i) if $m > \ell$ we simply skip s_{i+1}, as it is equal to s_i in the area of interest; (ii) if $m < \ell$ we return that p is not in the dictionary, as the strings s_i are sorted and we now have $p[1,m] = s_{i-1}[1,m] = s_i[1,m]$ and $p[m+1] = s_{i-1}[m+1] < s_i[m+1]$; (iii) if $m = \ell$, we compare $p[m+1\ldots]$ with $s_{i+1}[m+1\ldots]$, which are the characters of s_{i+1} that are explicitly coded. We compute the new value of ℓ, and also return that p is not in the dictionary if $p[\ell+1] < s_i[\ell+1]$.

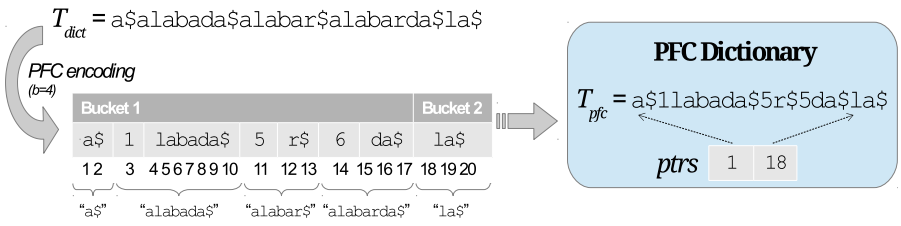

Figure 3: $\mathcal{T}_{dict}$ encoding with PFC ($b = 4$) and the resulting dictionary.

We propose two different Front-Coding based techniques for managing string dictionaries in compressed space: *Plain Front Coding* (PFC, Section 6.1) is an efficient byte-oriented implementation of the original technique, and *Hu-Tucker Front Coding* (HTFC, Section 6.2) uses Hu-Tucker coding on the headers and Huffman or Re-Pair compression on the buckets, in order to reduce the space requirements of PFC. The variant HTFC is novel, as far as we know.

6.1. Plain Front Coding (PFC)

PFC is a straightforward byte-oriented Front-Coding implementation. It encodes the data as follows.

- It uses VByte [78] to encode the length of the common prefix.
- The remaining string is encoded with one byte per character, plus the terminator \$.
- The header string is followed by the internal strings (each concatenating the VByte-coded length of the shared prefix and the remaining string), consecutively in memory.
- The buckets are laid consecutively in memory, and an array *ptrs* stores pointers to the beginning of each bucket.

Figure 3 shows how our example $\mathcal{T}_{dict}$ is encoded using PFC with a bucket size of $b = 4$ strings. The resulting encoded sequence (renamed $\mathcal{T}_{pfc}$) comprises two buckets: the first contains the first four words and the second contains only the fifth word. In this case, $\mathcal{T}_{pfc}$ takes $N' = 20$ bytes, whereas the original $\mathcal{T}_{dict}$ took $N = 29$, so the compression ratio is $N'/N \approx 69\%$. The entries of *ptr* use $\lceil \log N' \rceil$ bits.

Prefix-based Operations. The PFC representation enables prefix-based operations to be easily solved. All the strings prefixed by a given pattern hold a contiguous range of positions (and IDs) in the dictionary, so we only need to determine the first and last strings prefixed by the pattern p.

Operation `locatePrefix(p)` begins by determining the range of buckets $[c_1, c_2]$ containing the p-prefixed strings. This process involves a binary search (similar to that performed in `locate`) that, at some moment, may split into the search for c_1 and the search for c_2. The process finishes with a sequential scan of c_1 and c_2.

Operation `extractPrefix(p)` first locates the corresponding range using `locatePrefix`, and then scans the range extracting the strings one by one. Extraction is speeded up thanks to the shared prefix information.

6.2. Hu-Tucker Front Coding (HTFC)

`HTFC` is algorithmically similar to `PFC`, but it takes advantage of the redundancy of $\mathcal{T}_{pfc}$ to achieve a more compressed representation at the price of slightly slower operations. We obtain the Hu-Tucker (HT) code for the set of bucket headers. For the rest of the contents of the buckets (which include the VByte representations of the lengths used in `PFC`) we either build a Huffman code (a single one for the whole dictionary) or a Re-Pair set of rules (a single one for the whole dictionary). Then we encode $\mathcal{T}_{pfc}$ into a new string $\mathcal{T}_{htfc}$, which is also divided into buckets of b strings. Each original bucket of $\mathcal{T}_{pfc}$ is encoded as follows.

- The original header string is compressed with Hu-Tucker code, and the last encoded byte is padded with 0-bits in order to pack the header representation in an integral number of bytes.
- The rest of the bucket is compressed using Huffman or Re-Pair. In this case, it is convenient to avoid the zero-padding of the Huffman codes, as well as allowing Re-Pair rules spanning more than one string in the bucket. Only the last encoded byte of the last internal string is zero-padded, so that the resulting encoded bucket is also byte-aligned.
- As for `PFC`, the encoded buckets are concatenated (into string $\mathcal{T}_{htfc}$) and array *ptrs* points to the bucket beginnings.

Basic Operations. Both `locate` and `extract` follow the same algorithms described for `PFC`, but their implementation performs additional encoding/decoding operations to deal with the compressed representation.

For `locate(`p`)` we Hu-Tucker encode $p\$$ into p' (using $\mathcal{M}$) and pad it with 0-bits to use an integral number of bytes. Thus p' is directly binary searched for among the Hu-Tucker encoded headers of the buckets, $\mathcal{T}_{htfc}[ptrs[i]\ldots]$.

Once the candidate bucket c is determined, it is sequentially scanned as in `PFC` (unless the header was the string p). Each internal string is decompressed in turn. The decompressed data include the VByte representation of the length of the shared prefix with the previous entry and the remaining characters. Once decompressed, these data are used exactly as in `PFC`.

Operation `extract`(i) also performs as in `PFC`: the bucket $\lceil i/b \rceil$ is identified, and i mod b strings are then decompressed to obtain the desired answer.

Prefix-based Operations. These operations implement the same `PFC` algorithms, but are tuned for dealing with the compressed representation.

7. Binary Searchable Re-Pair (RPDAC)

If we remove the bitsequence $\mathcal{B}$ in Section 5, and instead sort $\mathcal{T}^*_{dict}$ in lexicographic order, we can still binary search $\mathcal{S}$ for p, using either bitsequence $\mathcal{Y}$ (Section 5.3) or DAC codes (Section 5.4). In this case, it is

better to replace Huffman by Hu-Tucker compression, so that the strings can be lexicographically compared bytewise, without decompressing them (as done in Section 6).

This arrangement corresponds to applying compression on the possibly simplest data organization for a dictionary: binary searching an array of strings. While this usually saves much space compared to a classical hash-based dictionary, the difference with our compressed hashing schemes is only the size of $\mathcal{B}$. As we will see in the experiments, this yields an almost negligible space gain, whereas in exchange the time of a binary search is much higher than when using hashing. Therefore, we anticipate that a binary searchable array will not be competitive with hashing in the compressed scenario.

However, as a proof of concept of this simple dictionary organization, we will develop its most promising variant, `RPDAC`, and include it in the experiments. `RPDAC` uses a lexicographically sorted $\mathcal{T}_{dict}$, which is compressed with Re-Pair ensuring that each string comprises an integral number of symbols in $\mathcal{C}$. In this way, the Re-Pair encoding of each string s_i can be seen as a variable-length substring of $\mathcal{C}$. We use DACs to represent each such variable-length string, so that the Re-Pair encoding of each string s_i can be directly accessed and binary search is possible. In addition, we do not need to represent the terminators \$, as explained in Section 5.4 .

Basic Operations. The `RPDAC` representation operates essentially as a binary searchable concatenation of the strings. For `locate(`p`)` we binary search the n strings, using DACs to extract the consecutive Re-Pair nonterminals that represent any string s_i, then we use $\mathcal{R}$ to expand those nonterminals, and finally compare the decompressed string s_i with p. In practice, the nonterminals are only extracted and expanded up to the point where the lexicographical comparison with p can be decided. The cost is linear in the sum of the lengths of the extracted strings. For `extract(`i`)` we access the i-th element in the DAC structure and decompresses it using $\mathcal{R}$. The cost is proportional to the output size.

Prefix-based Operations. Since strings are lexicographically sorted in `RPDAC`, we can again carry out prefix-based operations by determining the left and right range of the strings prefixed by the pattern. For `locatePrefix(p)` we the binary searches for the strings prefixed by p splits into two at some point, one for the first and one of the last such strings. For `extractPrefix(p)` we first locate the corresponding range using `locatePrefix(`p`)`, and then scan the range to extract the strings.

8. Full-Text Dictionaries (FM-Index)

A *full-text index* is a data structure that, built on a text $T[1, N]$ over an alphabet of size σ, supports fast search for patterns p in T, computing all the positions where p occurs. A *self-index* is a compressed full-text index that, in addition, contains enough information to efficiently reproduce any text substring [62]. A self-index can therefore replace the text.

Most self-indexes emulate a *suffix array* [55]. This structure is an array of integers $A[1, N]$, so that $A[i]$ represents the text suffix $T[A[i], N]$ and the suffixes are lexicographically sorted in A. Therefore, the positions

of all the occurrences of p in T, which correspond to the suffixes starting with p, form a lexicographic interval in the set of suffixes of T, and thus an interval in the suffix array, $A[sp, ep]$. The limits sp and ep can be found with two binary searches in $O(|p| \log N)$ time [55].

In order to use a suffix array for our dictionary problem, we consider a slight variant of $\mathcal{T}_{dict}$, where we prepend a symbol \$, that is, $\mathcal{T}_{dict}[1, N] = \$s_1\$s_2\$ \ldots \$s_n\$$. Since the strings s_i are concatenated in lexicographic order in $\mathcal{T}_{dict}$, and symbol \$ is smaller than all the others, we have an important property in the suffix array: $A[1] = N$, pointing to the final \$, and for all $1 \leq i \leq n$, $A[i+1]$ points to the suffix $\$s_i\$s_{i+1}\$ \ldots \$s_n\$$. Now, if we search for pattern $\$p\$$, we will find an occurrence iff $p = s_i \in \mathcal{D}$, and moreover it will hold $A[sp, ep] = A[i+1, i+1]$, so we just return $sp - 1$ to solve a `locate(`p`)` query.

A self-index emulating a suffix array can find the interval $A[sp, ep]$ given the pattern $\$p\$$, thus we solve the `locate(`p`)` query with it. Most self-indexes can also extract any text segment $T[l, r]$ provided one knows the suffix array cell k such that $A[k] = l$ (or $A[k] = r$, depending on the self-index). In our case, we can easily perform `extract(`i`)` because we know that the first character of $\$s_i\$$ is pointed to by $A[i+1]$, and the last character is pointed to by $A[i+2]$.

The self-index we will use is the FM-Index [33, 34], as it was found to be the most space-efficient in practice [29]. The FM-index computes sp and ep in time $O(|p| \log \sigma)$, and extracts s_i in time $O(|s_i| \log \sigma)$ (it starts from $A[i+2]$, which points to the end of $\$s_i\$$). We use two variants of the FM-Index, available at *PizzaChili*[10]. The one we call *RG* (version `SSA_v3.1` in *PizzaChili*) is faster but uses more space, and the one we call *RRR* (version `SSA_RRR` in *PizzaChili*) is slower but uses less space. Variant *RG* corresponds to the so-called *succinct suffix array* [34], which achieves zero-order compression of T, whereas variant *RRR* uses the *implicit compression boosting* idea [54], which reaches higher-order compression. We note that the use of the FM-index to handle dictionaries is not new, and it has indeed been extended to more powerful searches, where one looks for strings starting with a pattern p and simultaneously ending with a pattern s [35].

Prefix-based Operations. If, instead of searching for $\$p\$$, we search for $\$p$, we find the area $A[sp, ep]$ of all the strings s_i that start with p, and can output the range of IDs $[sp - 1, ep - 1]$ as the result of query `locatePrefix(`p`)`. For operation `extractPrefix(p)` we apply `extract(`i`)` to each $sp - 1 \leq i \leq ep - 1$.

Substring-based Operations. If we search for p, we will find all the occurrences of p within any string s_i of the dictionary. In order to find the ID i corresponding to an occurrence, we use the ability of self-indexes to extract $T[l \ldots]$ if one knows the k such that $A[k] = l$. In the case of the FM-index, we can extract $T[\ldots r]$ in reverse order if one knows the k such that $A[k] = r$. Moreover, at any time during this text extraction, the FM-index knows which cell of A points to each symbol $T[j]$ it displays. In the first case, let $A[sp, ep]$ be the interval that results from searching for p, and let $sp \leq k \leq ep$ be any cell in the range. Then we know that p is inside some s_i in $\mathcal{T}_{dict}$, and that $A[k] = r$ points to the position where p starts. Then we extract the

[10] `http://pizzachili.dcc.uchile.cl`

area $T[l, r] = \$ \ldots p[1]$, one by one until extracting the symbol \$. At this point we know that this symbol is pointed from $A[i + 1]$ and hence reveal i.

Thus, the mechanism to solve query `locateSubstring(`p`)` is to find $A[sp, ep]$ for p and apply the process described for each $k \in [sp, ep]$. A further complication is that p could occur several times within the same string s_i, thus we have to remove duplicates before reporting the resulting set of IDs. For `extractSubstring(`p`)`, we apply `extract(`i`)` on each ID reported by `locateSubstring(`p`)` (this can be slightly optimized because a part of s_i has been already recovered in order to reveal each ID).

As described, a problem is that operation `locateSubstring(`p`)` may take time proportional to the sum of the lengths of the located strings. Compressed suffix arrays provide a worst-case guarantee by choosing a sampling step s, regularly sampling T at all positions $j \cdot s$, marking the corresponding positions $A[i] = j \cdot s$ by setting $B[i] = 1$ in a bitsequence $B[1, N]$, and recording the sampled values $A[i]$ at another array $S[\texttt{rank}_1(B, i)] = j$. Then the location of any occurrence $A[k] = r$ is obtained in at most s steps by traversing, with the FM-index, the text positions $r, r - 1, r - 2, \ldots$ while knowing the suffix array position $A[k_i]$ from where the position $r - i$ is pointed. As soon as it holds $B[k_i] = 1$, we have the answer $r = S[\texttt{rank}_1(B, k_i)] + i$.

We use that scheme, with the only differences that (1) we store the ID of the string, instead of the position, in S, and (2) we make sure that the symbols \$ of $\mathcal{T}_{dict}$ are also sampled, so that we do not confuse one string ID with another. Therefore, using $(N/s) \log n$ bits for S, we ensure a locating time of $O(s \log \sigma)$ per located string using an FM-index.

9. Compressed Trie Dictionaries (XBW)

A trie (or digital tree) [36, 50] is an edge-labeled tree that represents a set of strings, and thus a natural choice to represent a string dictionary. Each path in the trie, from the root to a leaf, represents a particular string, so those strings sharing a common prefix also share a common subpath from the root. The leaves are marked with the corresponding string IDs.

Our basic operations are easily solved on tries. For `locate(`p`)` we traverse the trie from the root, descending by the edges labeled with the successive characters of p. If we end in a leaf, its stored ID is the answer. For `extract(`i`)`, we start from the leaf labeled i (so we need some way to find it directly) and traverse the trie upwards to the root, finding s_i in reverse order at the labels of the traversed edges. Tries also naturally support prefix-based searches: if we descend from the root following the characters of p and end in an internal trie node, then the IDs stored at all the leaves descending from that node are the answer to query `locatePrefix(`p`)`, and for `extractPrefix(`p`)` we traverse the trie upwards from each of those leaves.

The main problem of tries is that, in practice, they use much space, even if such space is linear. While there are several compressed trie representations [9, 42, 4] (some of which we compare in our experiments), we focus on representing a compressed trie using the so-called `XBW` [32], because this will support substring searches as well. The `XBW` is an extension of the FM-index to handle a labeled tree instead of a linear string.

Let τ be a trie with N nodes, I of which are internal. By analogy with the string case, call a *suffix* of τ any string formed by reading the labels from an internal node to the root. Now assume we sort all those I suffixes into an array $A[1, I]$. Then, given a pattern p, two binary searches on A (for p read backwards) are sufficient to identify the range $A[sp, ep]$ of all the internal nodes that are reached by following a path labeled with p. This is the basic idea behind the powerful *subpath search* operation of the `XBW`.

The `XBW` structure consists of two elements: (1) a sequence $S_\alpha[1, N]$ storing the labels of the edges that lead to the children of each internal node, considering the internal nodes in the order of A, and (2) a bitsequence $S_{last}[1, N]$ marking the last child of each of those I internal nodes in S_α. Ferragina et al. [32] show that this is sufficient to simulate downward and upward traversals on τ, and to support subpath searches. The space required is, at most, $(1 + \log \sigma)N$ bits, where we note that here N is the number of nodes in the trie, usually significantly less than the length of the string $\mathcal{T}_{dict}$.

To use the `XBW` for our purposes, we insert the strings $\$s_i\$$ into τ, instead of just s_i. Further, we renumber the IDs so that they coincide with the positions of the \$ labels in S_α: the node corresponding to the ith occurrence of \$ in S_α (i.e., the leaf that is the target of such edge labeled \$) will correspond to the string called s_i. In addition, we use a *wavelet tree* structure [41] to represent S_α. It uses at most $N \log \sigma$ bits of space (and less if $\mathcal{D}$ is compressible) and supports operations `rank` and `select` on S_α in $O(\log \sigma)$ time. The subpath search operation is carried out in $O(|p| \log \sigma)$ time, and it identifies the area $S_\alpha[sp, ep]$ of all the children of the resulting nodes. The bitsequences, both those of the wavelet tree and S_{last}, can be represented in uncompressed or compressed form (variants *RG* or *RRR*, respectively). While there is little novelty in the use of the `XBW` to represent a set of strings, our implementation of the data structure is new, as we could not find it publicly available.

Basic Operations. For `locate(`p`)`, instead of traversing the trie from the root to a leaf (which is possible, but slow on the `XBW` representation), we use the subpath search operation for pattern $\$p\$$. As a result, a single position $S_\alpha[k]$ is obtained if $p = s_i \in \mathcal{D}$. The corresponding ID is obtained as $i = \texttt{rank}_\$(S_\alpha, k)$. For `extract(`$i$`)`, we find the corresponding leaf $k = \texttt{select}_\$(S_\alpha, i)$, and traverse the trie upwards from $S_\alpha[k]$.

Prefix-based Operations. For `locatePrefix(`p`)` we search as above, this time for $\$p$, and end up in a range $S_\alpha[sp, ep]$ corresponding to (the children of) the internal node $v \in \tau$ whose path from the root spells out p. Now we perform a downward traversal from v towards every possible leaf descendant. Unfortunately this is relatively slow and the resulting leaves (and their IDs) are not consecutive. We can recall the labels followed in this recursive traversal so that, when we arrive at each leaf, we can output the corresponding string (prepending p), in order to solve operation `extractPrefix(`p`)`.

Substring-based Operations. Although prefix-based operations are not so fast, they are easily generalized to the powerful substring-based operations. For `locateSubstring(`p`)` we search as above, this time just for p, then proceed as for `locatePrefix(`p`)` (now range $S_\alpha[sp, ep]$ may include the children of many

different internal nodes v). For `extractSubstring(`p`)`, we must in addition recover the symbols that label the edges in the path from the root to each corresponding node v.

10. Experimental Evaluation

This section analyzes the empirical performance of our techniques, in space and time, over dictionaries coming from various real-world scenarios. We first consider the basic operations of `locate` and `extract`, comparing our techniques in order to choose the most prominent ones, and then comparing those with other relevant approaches from the literature. Then, we consider the prefix and substring based operations on those dictionaries where those operations are useful in practice. At the end, we discuss the construction costs of our techniques.

10.1. Experimental Setup

Our experiments were performed on two different computational configurations, which differ mainly in the RAM size. The lookup and extraction tests were performed on an Intel-Core i7 3820 @3.6 GHz, 16GB RAM, running Debian 7.1. The construction, instead, was carried out on a more powerful configuration: Intel Xeon X5675 @3.07 GHz, 48GB RAM, running Ubuntu 14.04.2 LTS.

Datasets. We consider a variety of dictionaries from different application domains.

Geographic names comprises all different names for the geographic points in the *geonames* dump[11]. We choose the *"asciiname"* column and delete all duplicates. The dictionary contains $5,455,164$ geographic names and occupies 81.62 MB.

Words comprises all the different words with at least 3 occurrences in the *ClueWeb09* dataset[12]. It contains $25,609,784$ words and occupies 256.36 MB.

Word sequences is obtained from the phrase table of a parallel English-Spanish corpus[13] of $1,353,454$ pairs of sentences. It results in two word sequence dictionaries:

(en) It comprises $36,677,283$ different *English* word sequences, and occupies 983.32 MB.

(sp) It comprises $39,180,899$ different *Spanish* word sequences, and occupies 1127.87 MB.

[11] `http://download.geonames.org/export/dump/allCountries.zip`

[12] `http://lemurproject.org/clueweb09`

[13] This corpus was obtained by combining Europarl, `http://www.statmt.org/europarl/v7/es-en.tgz`, and News Commentary corpus from the WMT Workshop 2010, `http://www.statmt.org/wmt10/`. The resulting bitext was tokenized and bilingual phrases were discarded if the phrase in a language contained 9 times more words than its counterpart in the other language, or if the phrase was longer than 40 words.

Dictionary	Size (MB)	# strings	Avg. length	σ	H_0	Trie Nodes	Front-Coding	Re-Pair
Geographic names	81.62	5,455,163	15.69	123	4.96	45.74%	51.82%	44.87%
Words	257.07	25,671,285	10.50	38	4.75	37.78%	47.33%	60.50%
Word sequences (en)	983.32	36,677,283	28.11	136	4.32	24.86%	28.41%	25.39%
Word sequences (sp)	1127.87	39,180,899	30.18	138	4.35	24.59%	27.90%	24.16%
URIs	1311.91	26,948,638	51.04	81	5.07	5.45%	7.41%	10.96%
URLs	1372.06	18,520,486	77.68	101	5.29	21.08%	22.40%	11.61%
Literals	1590.62	27,592,059	60.45	206	5.27	×	84.45%	15.10%
DNA	114.09	9,202,863	13.00	6	2.27	20.08%	27.51%	35.50%

Table 1: Description of the datasets.

URIs comprises all different URIs used in the *Uniprot* RDF dataset[14]. It contains $26,948,638$ different URIs taking 1311.91 MB of space.

URLs corresponds to a 2002 crawl of the `.uk` domain from the WebGraph framework[15]. It contains $18,520,486$ different URLs and 1372.06 MB.

Literals comprises an excerpt of $27,592,013$ different literals from the *DBpedia 3.9* RDF dataset[16]. It takes 1590.62 MB of space.

DNA contains all subsequences of 12 nucleotides found in the sequences of S. Paradoxus published in the *para* dataset[17]. It contains $9,202,863$ subsequences and occupies 114.09 MB.

Table 1 summarizes the most relevant statistics for each dictionary: the original $\mathcal{T}_{dict}$ size (in MB), number of different strings, average number of characters per string (including the special \$ terminator), number of different characters used in the dictionary (σ), and the zero-order entropy (H_0) in bits per character. In addition, the three last columns provide basic details about potential sizes of a trie-based representation (expressed as the number of nodes in the trie as a percentage of n), a front-coded one (expressed as the number of characters required for a Front-Coding with infinite bucket size, as a percentage of n), and a Re-Pair one (expressed as the number of bytes needed by a plain representation of the rules and $\mathcal{C}$ array, as a percentage of n). For some dictionaries, we were unable to build the corresponding tries in our computational setup, due to excessive memory usage during construction.

Prototypes. All our structures are implemented in C++, and use facilities (when necessary) from the `libcds` library[18]. Prototypes are compiled using `g++` (version 4.7.2) with optimization `-O9`. Below, we describe the different parameterizations studied for each technique:

[14] `ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/rdf`
[15] `http://law.dsi.unimi.it/webdata/uk-2002`
[16] `http://downloads.dbpedia.org/3.9/en`
[17] `http://www.sanger.ac.uk/Teams/Team71/durbin/sgrp`
[18] `https://github.com/fclaude/libcds`

Hash : The four hash-based techniques (`Hash`, `HashB`, `HashBB`, and `HashDAC`) are combined with Huffman (referred to as `huff`) and Re-Pair (referred to as `rp`) compression. In all cases, they implement their respective bitsequences using RG with 5% of overhead (parameter 20). Space/time tradeoffs are obtained by varying the load factor $\alpha = n/m$. We consider $m = 1.1n$ (i.e., the hash table has 10% more cells than strings in the dictionary), $m = 1.25n$, $m = 1.5n$, $m = 1.75n$, and $m = 2n$.

Front-Coding : We consider several variants of the two Front-Coding based techniques (`PFC` and `HTFC`). On the one hand, we analyze `PFC` as described in Section 6.1, but also consider the use of Re-Pair for compressing the internal strings (referred to as `PFC-rp`). On the other hand, we test `HTFC` in combination with Huffman (`HTFC-huff`) and Re-Pair (`HTFC-rp`).

RPDAC : We implement the technique following its description. We also tested how the dictionary performs when the Re-Pair grammar is also compressed [39], but this was never competitive with the basic technique in our case.

FM-Index : Two FM-indexes prototypes are tested. `FMI-rg` uses RG bitsequences for implementing the aforementioned `SSA_v3.1`, and `FMI-rrr` uses compressed RRR bitsequences for building `SSA_RRR`. We parameterize RG using sample values of 20 (5% of overhead), 5 (20% of overhead), and 2 (50% of overhead), and RRR using sample values 16, 64, and 128. The additional sampling structure, required for substring lookups, is built according to the specific dictionary features and is described for each particular test.

XBW : Two variants are tested, using RG or RRR bitsequences (`XBW-rg` and `XBW-rrr`, respectively). Their parameters are as for the FM-index.

10.2. Basic Operations

The first test analyzes `locate` and `extract` performance. For `locate`, we choose 1 million strings at random from each dataset in order to measure response times. In addition, we tested unsuccessful searches, that is, for strings not in the dataset. These results are not shown because they gave times similar to those obtained for successful searches. For `extract`, we look for the IDs corresponding to the strings located before, running 1 million operations. All the results reported for each experiment are averaged *user times* over 10 independent runs.

The results for these experiments are presented through pairs of plots reporting space/time tradeoffs for `locate` and `extract`. Each plot represents dictionary sizes in the x-axis and query times in the y-axis (in logscale). Space is reported as the percentage of the size of the dictionary encoding with respect to the size of the original $\mathcal{T}_{dict}$ string using one byte per character. Times are expressed in microseconds per operation.

We first identify the most relevant alternatives of compressed hashing and of Front-Coding. These will then be compared with our other proposed techniques. For succinctness, in this stage we only show two

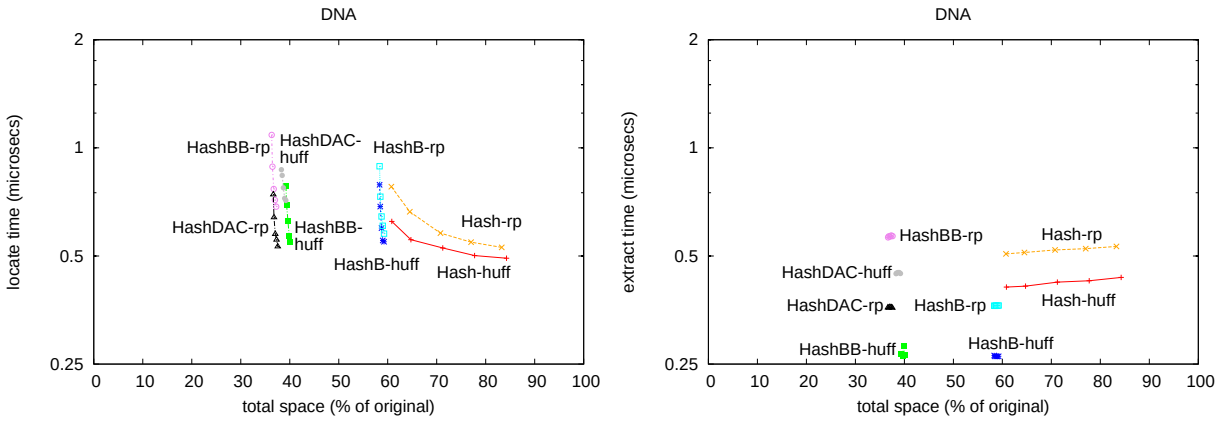

Figure 4: `locate` and `extract` performance comparison for DNA using hash-based techniques.

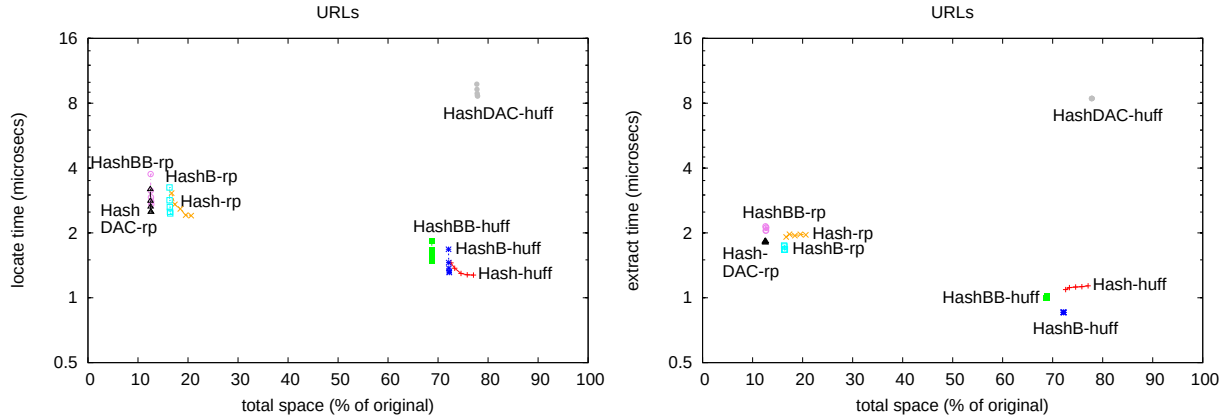

Figure 5: `locate` and `extract` performance comparison for URLs using hash-based techniques.

dictionaries to draw conclusions about each family of techniques, choosing the plots where the conclusions show up most clearly. All the other plots for all the datasets in the setup are shown in the Appendix.

Compressed hash dictionaries. Regardless of the specific hash-based technique, the use of Re-Pair for string compression clearly outperforms Huffman, except for `DNA`, which has the lowest zero-order entropy among the datasets. This result shows that string repetitiveness in dictionaries generally offers better compression opportunities than bias in the symbol frequencies. Figures 4 and 5 show the results on `DNA` and `URLs`.

Huffman effectiveness is lower-bounded by the zero-order entropy of the dictionary strings, which is generally over 4 bits per character (see Table 1). On top of this space, `Hash-`*huff* adds a table of pointers with a percentage of empty cells, `HashB-`*huff* replaces the empty cells by a bitsequence marking which are empty, `HashBB-`*huff* replaces the nonempty cells by another bitsequence, and finally `HashDAC-`*huff* changes

this last bitsequence for a DAC encoding of the compressed strings. It is not surprising that the space of `HashB-`*huff* is very close to that of `Hash-`*huff* when the latter uses the minimum number of empty cells (only 10%). In turn, `HashBB-`*huff* sharply improves on that space, using between 6% (`URIs`) and 30% (`DNA`) of space on top of the zero-order entropy. Moreover, `HashDAC-`*huff* demands more space than `HashBB-`*huff*, and the difference increases for longer strings. With respect to `locate` time, `Hash-`*huff* is slightly faster than `HashB-`*huff*, as it saves a `rank` operation in the intermediate steps of the search (i.e., those where the value is rehashed to a new cell, which may happen zero times); `HashB-`*huff* is in turn faster than `HashBB-`*huff*, as it saves a `select` operation on a longer bitsequence. However, this difference is minimal for dictionaries with shorter strings like `DNA`. `HashDAC-`*huff* competes with `HashBB-`*huff* on dictionaries with shorter strings (for instance, `DNA`), but the space/time performance of DAC degrades for longer strings (as for `URLs`). For `extract`, it is now `Hash-`*huff* the one needing a `select` operation that is unnecessary on `HashB-`*huff*, which is the fastest choice. `HashBB-`*huff* is close to `HashB-`*huff*, but only outperforms it on `DNA`, while `HashDAC-`*huff* never competes. The comparison among them is as for `locate`.

The use of Re-Pair compression shows to be an excellent choice. The comparison among the different hash techniques conveys to the same conclusions reported for Huffman compression, except for `HashDAC-`*rp*. Since it uses symbols wider than bytes, the space overhead is lower and fewer `rank` operations are needed to extract the compressed strings, compared to the byte-aligned Huffman codes. This variant always achieves the most compressed dictionaries and reports competitive performance for both `locate` and `extract`. In each case, it performs close to the fastest variant: `Hash-`*rp* for `locate` and `HashB-`*rp* for `extract`.

We conclude that `HashDAC-`*rp* is the best positioned technique among the hash-based ones. It achieves compression ratios around 12%–60%, and requires 0.5–3.2 μs to `locate` and 0.4–2 μs to `extract`. We will also promote `HashB-`*huff* for the next experiments. Although its compression effectiveness is not competitive (60%–100%), it reports the best overall time performances: 0.5–1.7 μs for `locate` and 0.2–1 μs for `extract`.

Perfect hashing. An interesting experiment is to compare our double-hashing technique with minimum perfect hashing, considering space and `locate` speed (the extraction process is the same for both schemes). Minimum perfect hashing maps all the strings to the interval $\mathcal{H}'[1, n]$ without collisions, and thus saves the space for bitsequence $\mathcal{B}$ used in our hashing schemes. In exchange, it needs in practice about $2.7n$ bits of extra space, which is similar to that of double hashing with a table of $m \approx 2.57n$ entries and a load factor of $\alpha = n/m \approx 0.39$. Even with perfect hashing, since we cannot ensure that the string p is actually in the dictionary, operation `locate(`p`)` must also extract the string found and compare it with p.

For the comparison between double and perfect hashing, we choose the representation `HashDAC-`*rp*, which has emerged as generally the best choice in our experiments. This means that we hash the uncompressed string p, and that for each string found s_i we must decompress its DAC + Re-Pair representation and then compare it with p.

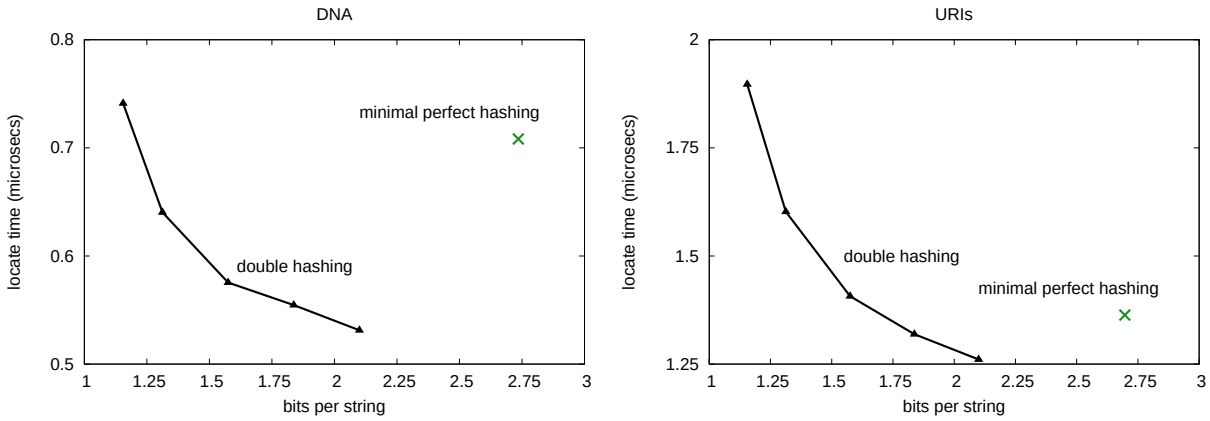

Figure 6: Space and space and `locate` time of our double hashing and minimal perfect hashing, for technique `HashDAC-`*rp*, on `DNA` and `URIs`.

We choose an efficient minimal perfect hash implementation from the Sux4J[19] library. We add up the time to compute the hash function on p with Sux4J and the time to extract the string s_i and compare it with p in our implementation.

Figure 6 shows the results achieved on `DNA` and `URIs`. Each plot represents, in the x-axis, the amount of additional bits used on top of the compressed strings. The y-axis shows the query times in μs. We remind that `HashDAC-`*rp* adds 1.05 bits per cell in the hash table, that is, if the hash table has 25% more cells than strings, it adds $1.05 \cdot 1.25 = 1.3125$ bits per string. We consider table sizes up to $m = 2n$ for double hashing, which requires 2.1 bits per string. This is still well below the ≈ 2.7 bits of minimal perfect hashing. As shown in our experiments, the impact of these bits in the total space used by the index is low anyway; we only want to emphasize that the use of perfect hashing does not involve a space reduction.

The differences in time slightly favor double hashing over perfect hashing in the figures, and the difference decreases on longer strings. These results are not conclusive, however, because the perfect hashing implementation is in Java and that of double hashing is in C++, and there is much discussion about up to what extent one can compare implementations in those languages.

As a platform-independent comparison, double hashing produces about 30% collisions when using about 2.7 bits per string (theory predicts 27% with an ideal hash function). Each such collision involves 2 cache misses to compute `rank` and 2–5 to extract the string and compare it (only up to the point where one can see that there is a difference). This amounts on average to about $130\% \times 2 + 30\% \times 2{-}5 \approx 3.2{-}4.1$ cache misses on top of the cost of comparing the key with the right string. Inspection of the perfect hash in Sux4J shows that 4 cache misses are to be expected in function `MinimalPerfectHashFunction.getLong`. The time spent

[19] `http://sux.di.unimi.it/`

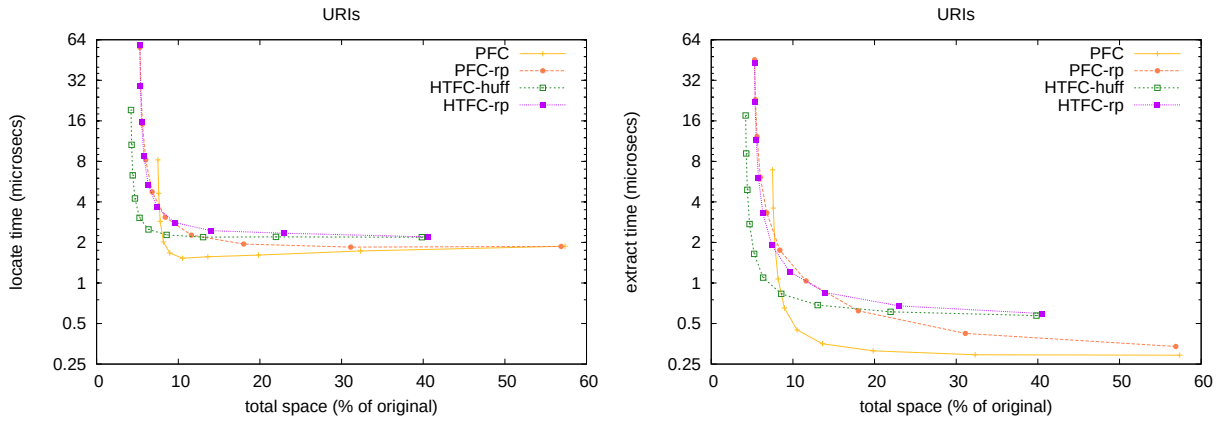

Figure 7: `locate` and `extract` performance comparison for URIs using Front-Coding.

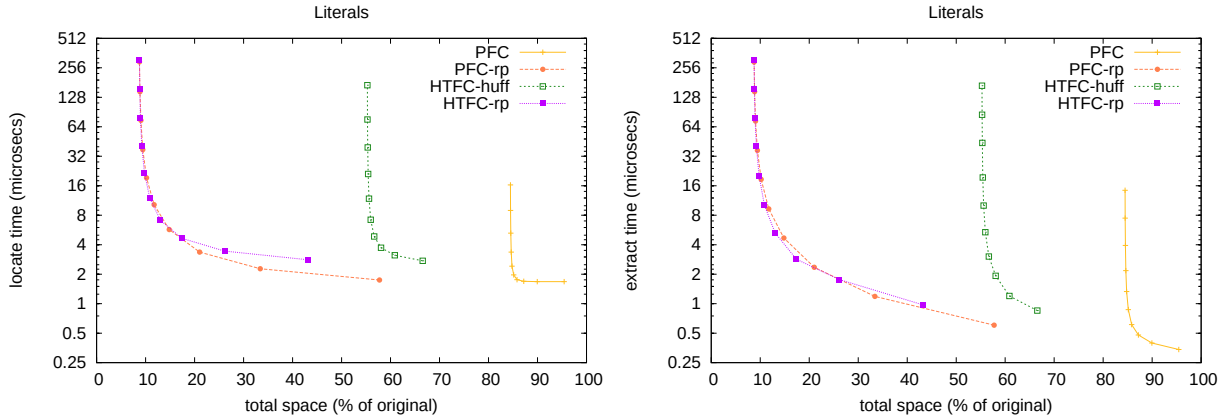

Figure 8: `locate` and `extract` performance comparison for Literals using Front-Coding.

by both schemes to extract and decompress the final string further blurs those differences. This explains why no noticeable differences should be expected between double and perfect hashing in our application.

Front-Coding dictionaries. This family of techniques draws comparable patterns for all datasets in our setup. Lookup times increase gradually from small ($b = 2$) to medium-size bucket sizes ($b = 32$), but from these to larger sizes ($b = 1024$) their performance degrades sharply. Thus, we consider as competitive configurations those ranging from $b = 2$ to $b = 32$: the former achieve better time and the latter obtain better space. This can be seen in Figures 7 and 8, where results for `URIs` and `Literals` are shown. They are, respectively, the best and the worst datasets for Front-Coding, as shown in Table 1.

`PFC` is the fastest choice in all cases, both for `locate` and `extract`, at the price of being the least effective compression technique. Extraction is always faster because it only needs to traverse a bucket, whereas

`locate` firstly locates the corresponding bucket with a binary search and then traverses it. The variant compressing the buckets with Re-Pair, `PFC-`*rp*, achieves some improvement for `URIs`, but its improvement is huge on `Literals`, where Re-Pair exploits repeated substrings within the buckets. Obviously, `PFC-`*rp* is slower than `PFC` because it must perform Re-Pair decompression within the buckets, but this difference is acceptable for bucket sizes up to $b = 32$.

`PFC-`*rp* performs similarly to `HTFC-`*rp*. Their lookup times are almost equal from buckets of 32 strings and their differences in space are negligible. Only for small bucket sizes is `HTFC-`*rp* more space-effective, although it is also slightly slower than `PFC-`*rp* (mainly for string location), and the latter takes over in the space/time tradeoff. `HTFC-`*rp* is also the most effective choice for the dictionaries `Geographic names`, `Word sequences` (English and Spanish), `URLs`, and `Literals`. However, `HTFC-`*huff* leads on `Words`, `URIs`, and `DNA`. Figure 7 shows this last case. `HTFC-`*huff* reports compression ratios as low as 4.2%, compared to 5.3% achieved by `HTFC-`*rp*, and it also offers better time performance on the interesting range of space usage. The comparison changes completely on `Literals` (Figure 8), where the space usage of `HTFC-`*huff* makes it uninteresting.

Therefore, we use `HTFC-`*huff* for the upcoming experiments on `DNA`, `Words`, and `URIs`, and `HTFC-`*rp* for the remaining datasets. We will also include `PFC`, as it reaches the maximum speed.

Overall comparison. Finally, we compare in Figures 9 and 10 the best performing members from the hashing family (`HashB-`*huff* and `HashDAC-`*rp*), the best performing members of the Front-Coding family (`PFC`, and `HTFC-`*rp* or `HTFC-`*huff*), and our remaining techniques: `RPDAC`, `FM-Index`, and `XBW`. A number of general facts can be concluded from the performance figures:

- As anticipated, `RPDAC` and `HashDAC-`*rp* reach similar compression performance for all datasets, and also show similar `extract` times. However, `HashDAC-`*rp* outperforms `RPDAC` by far for `locate` because hashing is always faster than binary string searching.

- The `FM-Index` variants reach 20% to 50% of compression, which is never competitive with the leading techniques. They are also slower than the most efficient variants by an order of magnitude or more, for both operations.

- The `XBW` variants have not been built for `Literals` because their construction complexity exceeds the memory resources of our computational configuration. For the remaining datasets, their time performance is even worse than that of `FM-Index`, but `XBW-`*rrr* achieves the best compression of all the techniques, reaching 3% to 20% of space. It takes 20–200 μs for `locate` and 50–500 μs for `extract`.

- The variant of `HTFC` we chose for each dictionary achieves the best space after `XBW` (4% to 30%), but much better time: Using 5%–35% of space it solves `locate` in 1–6 μs and `extract` in 0.4–2 μs. It is the dominant technique, in general, unless one spends significatively more space.

- `HashDAC-`*rp* is faster than `HTFC` for `locate`, and performs similarly for `extract`. It compresses to 12%–65%, much worse than `HTFC`, but solves `locate` in 0.5–3.2 μs and `extract` in 0.4–2 μs.

- PFC also takes over HTFC for both operations when sufficient space is used: 8% to 55% (except on Literals, where PFC uses more than 80%). PFC obtains 1–2 μs for locate and 0.2–0.4 μs for extract. The relation between PFC and HashDAC-*rp* varies depending on the dataset: sometimes one completely dominates the other, sometimes each has its own niche.
- Finally, HashB-*huff* obtains the best locate times (albeit sometimes by a small margin) but not the best extract times, at a high price in space: 60%–100%. Its locate times are in the range 0.5–2 μs.

Let us analyze the particularities of the collections:

- The best compression performance is obtained on URIs: up to 3%, while obtaining competitive performance with just 5%. This dataset contains very long shared prefixes (Table 1), which is exploited by front-coded representations, and also by Re-Pair. However, the fact that HTFC-*huff* is preferred over HTFC-*rp* indicates that most of the redundancy is indeed in the shared prefixes. As a consequence, HashDAC-*rp* is completely dominated by PFC in this dataset.
- URLs and both Word sequences dictionaries are the next most compressible datasets: HTFC-*rp* reaches around 10% of space. They also contain long shared prefixes, yet not as long as URIs (see Table 1). In this case, the number of repeated substrings is a more important source of compressibility than the shared prefixes, as witnessed by the fact that HashDAC-*rp* (which applies only Re-Pair) outperforms PFC (which applies only Front-Coding) in space. The effect is most pronounced in URLs, where HashDAC-*rp* achieves almost the same space as HTFC-*rp*. In both Word sequences datasets, HashDAC-*rp* completely dominates PFC regarding locate times.
- In DNA, a low entropy is combined with fairly long shared prefixes. No further significant string repetitiveness arises, as witnessed by the fact that HashDAC-*rp* does not dominate PFC in space (yet it is faster). Added to the low entropy, it is not surprising that HTFC-*huff* is the preferred variant of HTFC, reaching almost 10% of space.
- It may be surprising that Literals also reaches around 10% of space, given that Table 1 shows that very short consecutive prefixes are shared, and thus PFC fails to compress this dataset. As in URLs, however, there is a large degree of substring repetitiveness, which makes Re-Pair based approaches succeed in compressing. As expected, HashDAC-*rp* gets very close in space to HTFC-*rp*.
- Finally, Geographic names and Words achieve much worse compression, close to 30% with HTFC. This results owes to the fact that they do not share long prefixes nor have much string repetitiveness (see Table 1). Shared prefixes are a better source of compressibility in Words, and string repetitiveness is in Geographic names, as witnessed by the relation between the space of PFC and HashDAC-*rp*.

To summarize, we have that, depending on the degree of shared prefixes and repeated substrings in the dictionaries, compression ratios of 5%, 10%, and 30% can be achieved. Within those spaces, operation

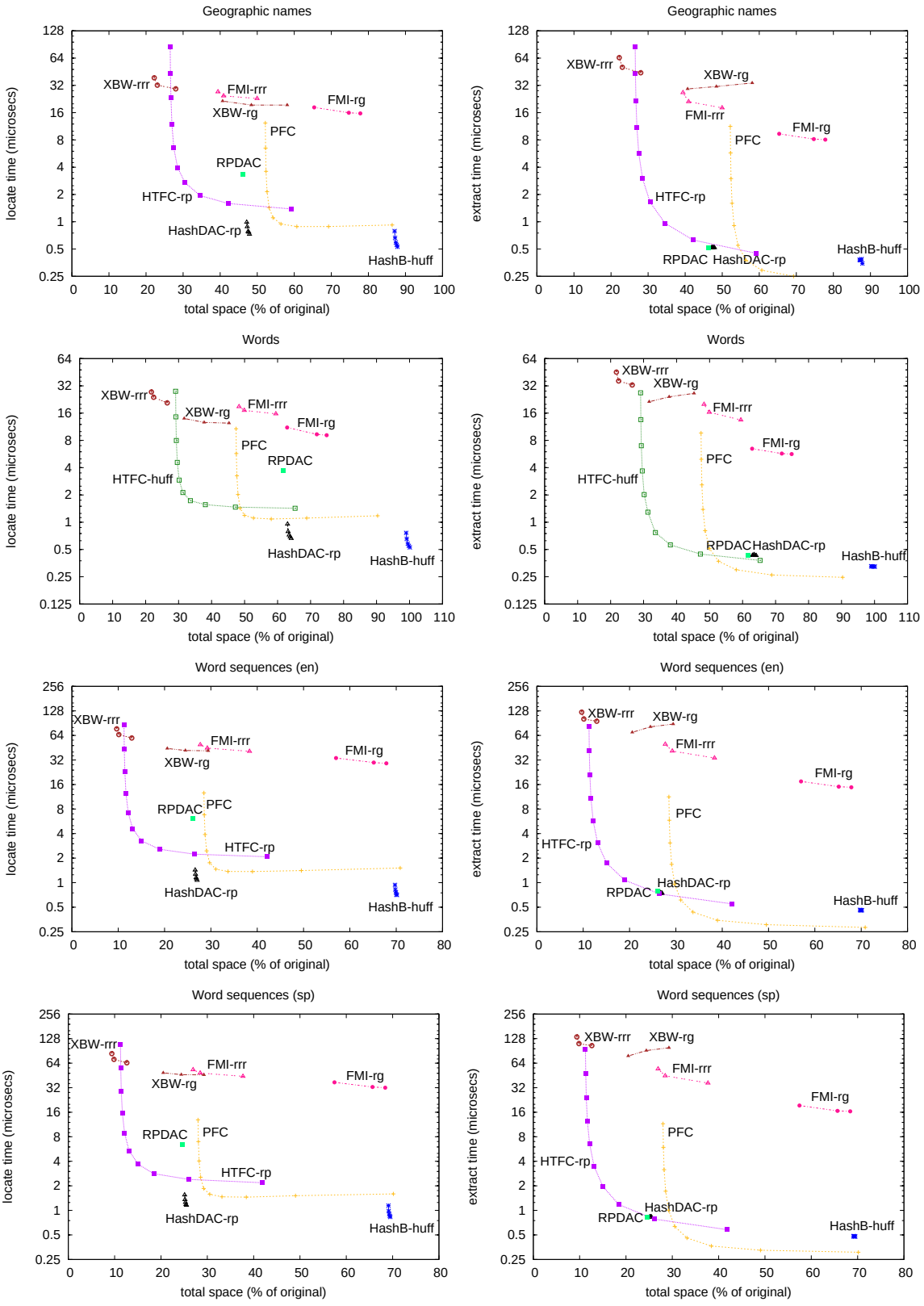

Figure 9: `locate` and `extract` performance comparison for `Geographic names`, `Words`, and `Word sequences`.

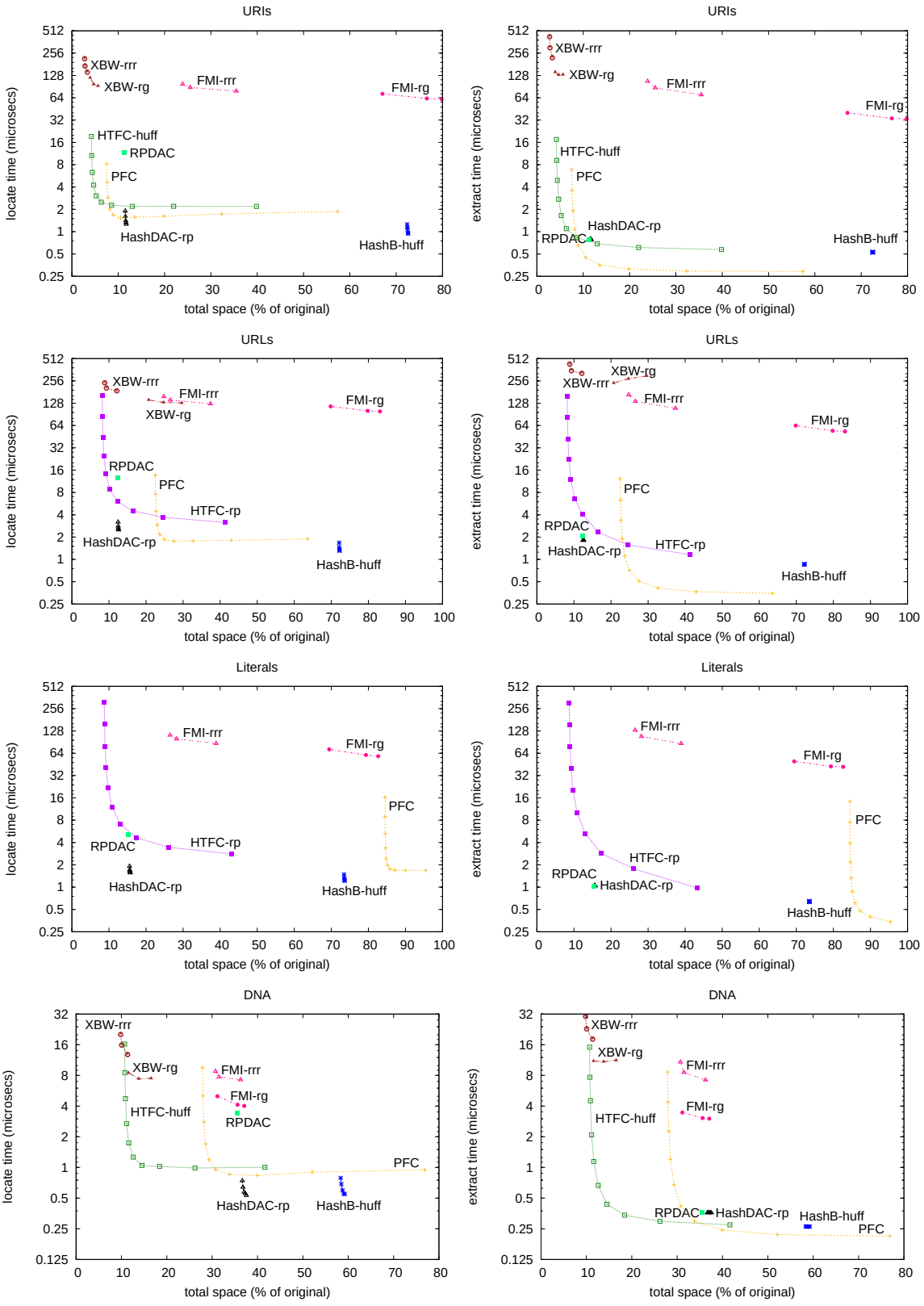

Figure 10: `locate` and `extract` performance comparison for `URIs`, `URLs`, `Literals`, and `DNA`.

`locate` can be solved in 1–6 μs, and *extract* in 0.4–2 μs, basically depending on the average string length in the dictionaries. Those performances correspond to the `HTFC` data structure. Faster operations and larger spaces can be obtained by using other structures like `HashDAC-`*rp*, `PFC`, and `HashB-`*huff*.

10.3. Comparison with the State of the Art

Now we compare our most prominent approaches with the most relevant techniques in the literature. We test the *centroid* (`Cent`) path-decomposed trie and the path decomposition with *lexicographic* order (`Lex`) [42]. Both techniques are compared with and without label compression (these are referred to as `CentRP` and `LexRP`, where the labels are Re-Pair compressed). We also compare the LZ-dictionaries [4]: one using path decomposition (`LZ-pd`) and other based on Front-Coding compression (`LZ-fc`) with bucket size 16. Additionally, we study their variants performing on the inverted dictionary parsing: $\texttt{LZ}^{-1}\texttt{T-pd}$ and $\texttt{LZ}^{-1}\texttt{T-fc}$. From our techniques, we include `PFC`, `HTFC`, and `HashDAC-`*rp* for all datasets. As before, we use `HTFC-`*huff* on `Words`, `URIs` and `DNA`, and `HTFC-`*rp* on the other datasets.

Figures 11 and 12 summarize the results obtained for the basic operations. In general, LZ-dictionaries report competitive tradeoffs, but all their variants are dominated by the centroid-based approaches that use Re-Pair. The only exception is on `URLs`, where `LZ-pd` achieves less space (but is slower) than the centroid-based schemes. Nevertheless, all the LZ-dictionaries are systematically dominated by the corresponding variant of `HTFC`. This is not surprising, since the LZ-dictionaries are based on variants of LZ78 compression, and this is weaker than Re-Pair compression.

From the centroid-based approaches, which can be seen as representatives of data structures based on tries (as our `XBW` approaches), `CentRP` clearly dominates the others. It is, however, dominated by `HTFC` when operation `extract` is considered, in almost all cases. The exceptions are `URIs` (where `CentRP` outperforms `HTFC` only marginally), `URLs` (where `CentRP` is anyway dominated by `HashDAC-`*rp*), and `Literals`. On the other hand, `CentRP` does dominate on a niche of the space/time map of operation `locate`. Generally, `CentRP` cannot achieve as little space as `HTFC`, but it achieves more speed for the same space, and then `HashDAC-`*rp* outperforms it by using more space. Some exceptions are `URLs` (where `HashDAC-`*rp* needs less space and similar time than `CentRP`), and `DNA` (where `HTFC` dominates `CentRP`). `CentRP` achieves 4%–35% compression and 1–3 μs for both operations.

The fact that our techniques dominate in almost all cases for operation `extract` is very relevant, as in many scenarios one carries out many more `extract` than `locate` operations. For example, when a dictionary is used to tokenize a NL text [19, 6], query words are converted into IDs using `locate`, once per query word. Most queries contain just 1–5 words. However, if a text snippet or a whole document is displayed, tens to thousands of `extract` operations are necessary, one per displayed word. Similarly, RDF engines like RDF3X [63] or Virtuoso[20] use a dictionary transformation to rewrite the original data as IDs, and these are used for

[20] `http://www.openlinksw.com`

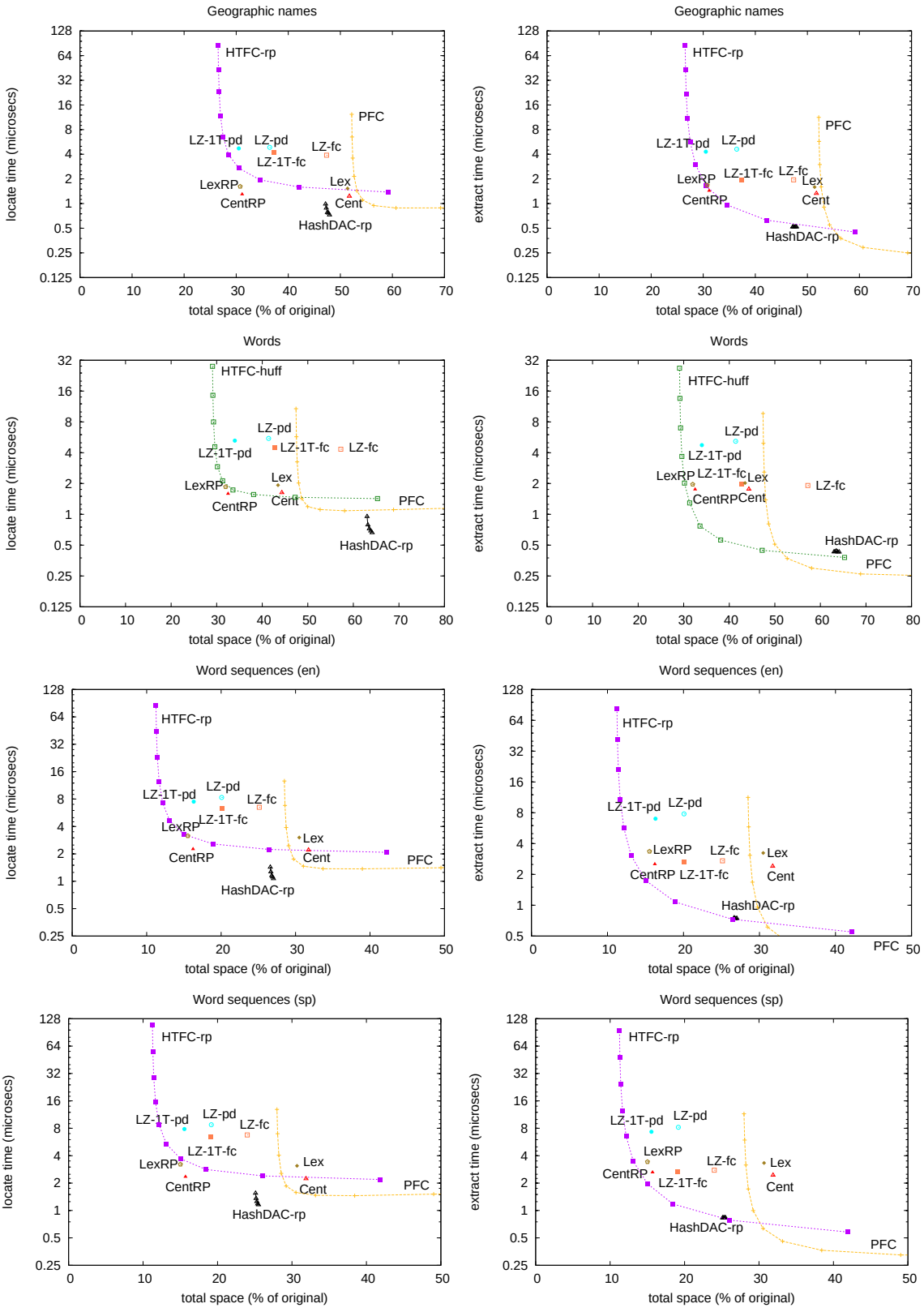

Figure 11: `locate` and `extract` performance comparison for `Geographic names`, `Words`, and `Word sequences`.

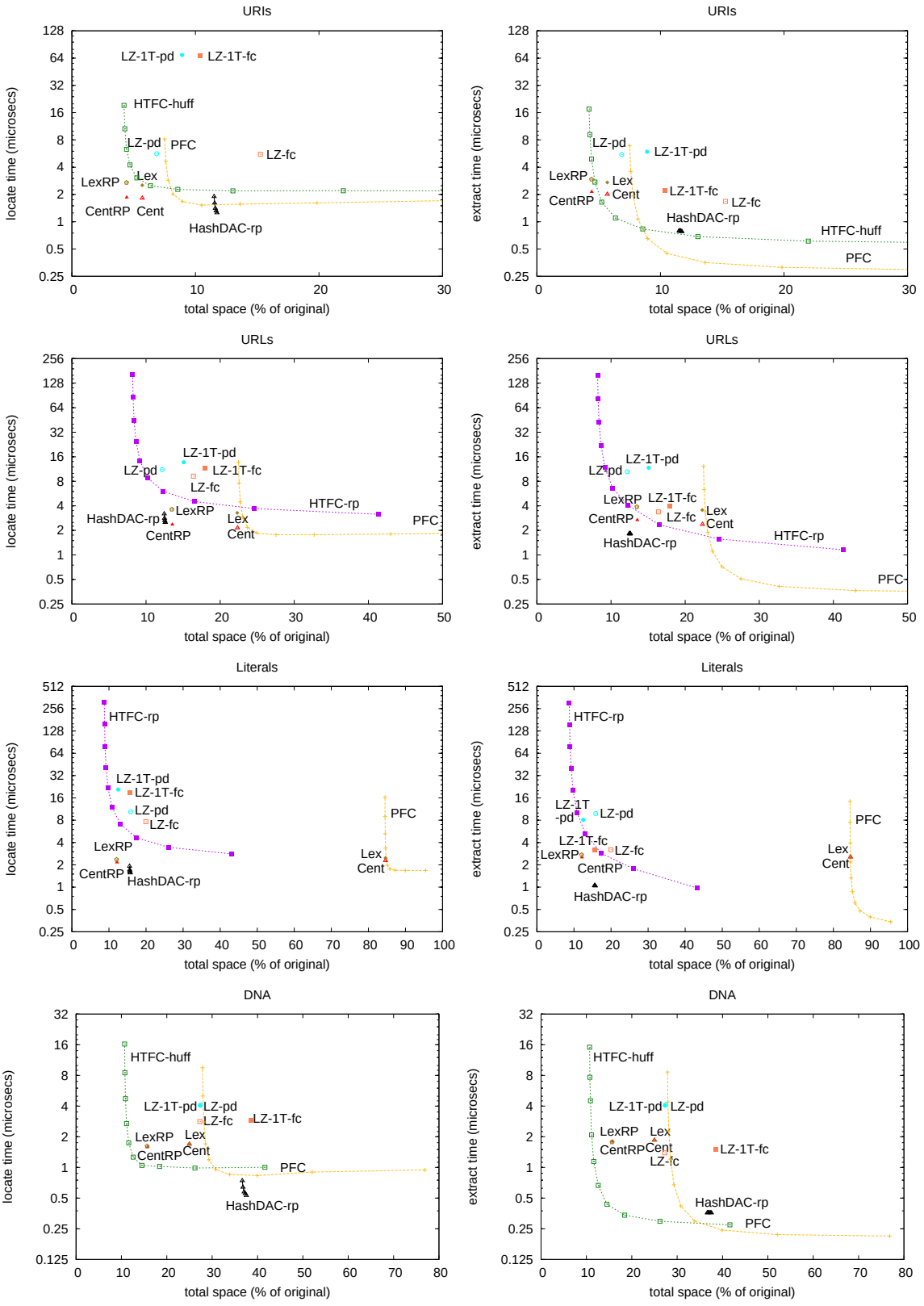

Figure 12: `locate` and `extract` performance comparison for `URIs`, `URLs`, `Literals`, and `DNA`.

indexing purposes. The words within SPARQL queries [67] are then converted into IDs using `locate`. In practice, the most complex queries (rarely used) involve at most 15 different patterns [3]. Once the engine retrieves a set of IDs, these must be translated to their corresponding strings in order to present them to the user. Although highly-restrictive queries can return a few results, the most common ones obtain hundreds or thousands of results. The situation is also similar in most applications of geographic names and URLs, whereas the weight of both operations is likely to be more balanced in word sequences for translation systems and in DNA k-mers.

10.4. Prefix-based Operations

Except possibly for `Literals`, prefix-based searches are useful in the applications where our dictionary datasets are used. For example, prefix searches are common on geographic names and words for autocompletion, they are used on MT systems (word sequences) to find the best translation starting at a given point of a text, to retrieve RDF descriptions for all URIs published under a given namespace, to retrieve all URLs under a specific domain or subdomain, and to retrieve all k-mers with a given prefix.

For each dataset except `Literals`, we obtain five sets of $100,000$ valid prefixes (returning, at least, one result per prefix) of different lengths: prefixes lengths are 60%, 70%, 80%, 90%, and 100%, of the average string length. The exception is `URIs`, where these prefixes are not sufficiently selective, so for `extractPrefix` we use prefix lengths of 90%, 100%, 110%, 120%, and 130% of the average string length.[21]

Hashing-based techniques do not support prefix-based searches, so we use `HTFC` (with the variant chosen as before) and `PFC` in these experiments. In both cases we use bucket size $b = 8$, which is generally the turning point in space vs time of both techniques. We also include `RPDAC` in these tests, but discard `FM-Index` and `XBW` because their performance is far from competitive in these tests.

The time measured for operation `locatePrefix` consists of the time required to determine the range of contiguous IDs $[a, b]$ in which the strings prefixed by the pattern are encoded. On the other hand, the time for `extractPrefix` comprises both the time required to determine the range $[a, b]$, and the time required for extracting those $b - a + 1$ strings in the range.

Figures 13 and 14 illustrate the prefix-based experiments for `Word sequences (en)` and `URIs`, showing `locatePrefix` (left) and `extractPrefix` (right). The times, on the y-axis, are expressed as μs per query in the case of `locatePrefix`, whereas for `extractPrefix` they are expressed in *nanoseconds* (ns) per extracted string. The x-axis represents the prefix length for `locatePrefix`, while for `extractPrefix` it represents the number of elements retrieved, on average, from each pattern (it is logarithmic on `URIs`). Obviously, longer patterns are more selective than shorter ones, thus they retrieve fewer results.

[21] Precisely, the prefix lengths used are 9, 10, 12, 13 and 15 on `Geographic names`; 6, 7, 8, 9 and 11 on `Words`; 16, 19, 22, 25 and 28 on `Word sequences (en)`,; 18, 21, 24, 27 and 30 on `Word sequences (es)`; 30, 35, 40, 45, and 51 on `URIs` (for `extractPrefix`, 45, 51, 56, 61 and 66); 46, 54, 62, 70 and 78 on `URLs`; and 7, 8, 9, 10 and 12 on `DNA`.

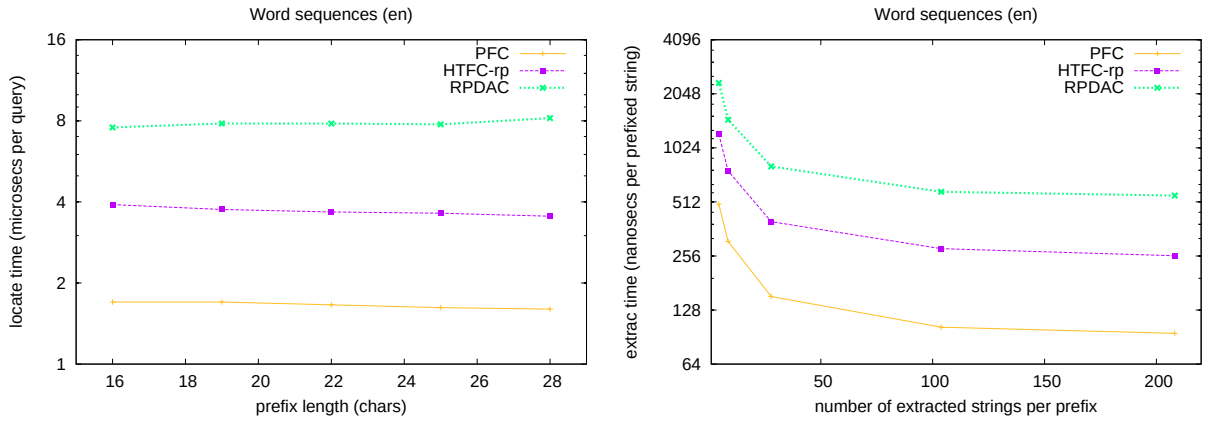

Figure 13: `locatePrefix` and `extractPrefix` performance comparison for `Word sequences (en)`.

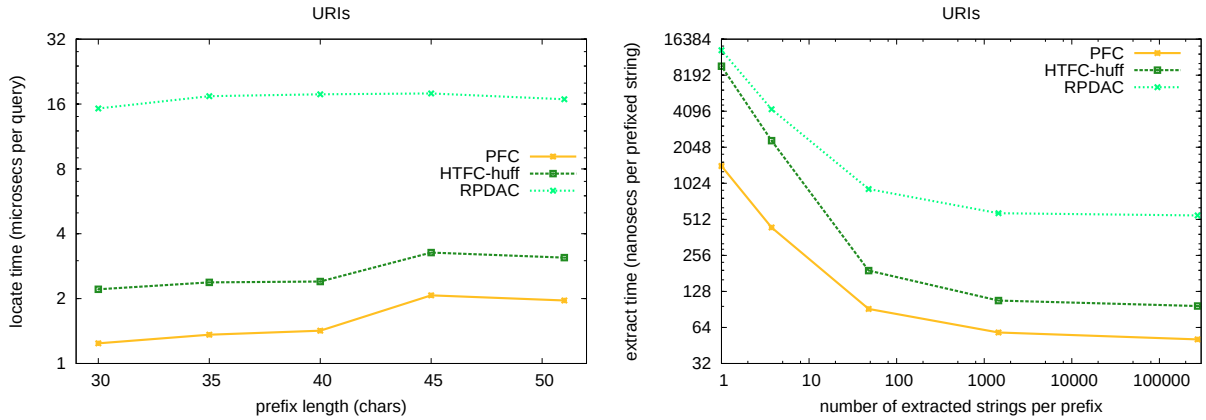

Figure 14: `locatePrefix` and `extractPrefix` performance comparison for `URIs`.

Two opposite effects arise when increasing the pattern length in `locatePrefix`. On the one hand, string comparisons may be more expensive, especially when long prefixes are shared among the strings. On the other hand, the search is more selective and the resulting range is shorter, which reduces the number of binary comparisons needed to find it. Different dictionaries are affected differently by these two effects. For `Geographic names`, `Words` and both `Word sequences` datasets, times remain stable as the patterns grow. The times slightly increase with the prefix length on `URLs` and `URIs`, as the strings sought are longer and long prefixes are shared. Finally, the times decrease with longer prefixes on `DNA`. This decrease owes to the fact that the strings are short and compressible, so the increase in selectivity is more relevant than the increase in length. In all cases, `PFC` is always the fastest choice (yet in several cases it uses more space), followed by `HTFC` and finally `RPDAC`. `PFC` times are around 1-2.5 μs per query, being `URLs` and `URIs` the worst case.

On the other hand, `extractPrefix` times (per retrieved result) decrease with prefix selectivity, reaching a stable value at about 50 extracted strings (for `URIs`, up to 1000 results must be retrieved to reach stability). This means that the cost of prefix location is quickly amortized, and from then on the time of `extractPrefix` is mainly due to string extraction. This time is roughly 50–100 ns per string for `PFC`, which is again the fastest choice, followed by `HTFC` and then by `RPDAC`, which is the least competitive choice.

10.5. Substring-based Operations

We have chosen two datasets where substring searches are most clearly used. On the one hand, it is common to use substrings when searching a collection of `Geographic names`, as many official names are longer than the names actually used, or in order to write, say "museum", and get suggestions. On the other hand, substring-based searching is the most frequently used form of the SPARQL `regex` query over RDF `Literals` [3]. For each dataset, we obtain five sets of 100,000 substrings of different lengths, returning at least one result. We consider substrings lengths from 25% to 60% of the average string lengths[22].

`FM-Index` and `XBW` are the only structures that support substring searching. Nevertheless, `XBW` dictionaries are only analyzed for `Geographic names` because we could not build them on `Literals`. Considering the reported tradeoffs for `locate` and `extract`, we build `FMI-rg` and `XBR-rg` with sampling value 20, while `FMI-rrr` and `XBR-rrr` are built with sampling value 64.

`FM-Index` *sampling.* We remind that the `FM-Index` uses a sampling step s to efficiently locate the strings. Before comparing it with others, we study how this sampling impacts on the `FM-Index` tradeoffs. We consider five different sampling values, $s = 8, 16, 32, 64, 128$.

Figure 15 compares space requirements for each sampling value and also without sampling (bar "original"). For `FMI-rg`, the space used with $s = 8$ doubles the original requirements in both datasets. This overhead progressively decreases with larger samplings, requiring 25%–60% for $s \geq 16$. On `FMI-rrr`, the original space is doubled already for $s = 16$, but the overhead also reaches reasonable values for larger values of s.

We study `locateSubstring` times, since the performance of `extractSubstring` is independent on how the strings were found. Figures 16 and 17 show the performance for `Geographic names` and `Literals`, respectively, showing the results for `FMI-rg` (left) and for `FMI-rrr` (right). The x-axis represents the length of the substring sought, and the y-axis (logarithmic) is the time required per located ID, in μs. This time includes finding the range $A[sp, ep]$ in which the substrings are represented, obtaining the corresponding string IDs (this requires at most s steps per ID), and removing duplicate IDs.

On `Geographic names`, the times are relatively stable as the pattern lengths grow, because the strings are not too long. As expected, the sampling step has a significant impact on times. For instance, `FMI-rg` takes about 1 μs per located ID with $s = 8$, 3 μs with $s = 32$, and more than 4 μs for $s = 128$. `FMI-rrr`

[22] Precisely, the substring lengths used are 4, 5, 6, 8 and 10 on `Geographic names`; and 18, 21, 24, 30 and 36 on `Literals`.

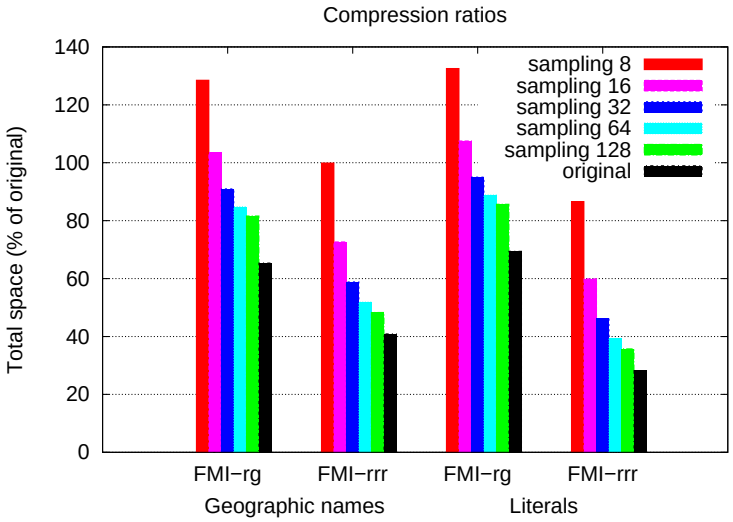

Figure 15: Space comparison for different `FM-Index` samplings.

reports higher times: 4 μs for $s = 8$, 10 μs for $s = 32$, and 13 μs for $s = 128$. However, `FMI-rg` uses much more space than `FMI-rrr` to achieve these times.

On `Literals`, times clearly worsen for longer search patterns, as the strings sought are longer. This effect is blurred for larger s values, where the locating cost becomes more relevant than the cost of finding $A[sp, ep]$. Comparisons between values of s are similar as before, but in this case `FMI-rrr` is closer to `FMI-rg`.

Comparing `FM-Index` *and* `XBW`. Figure 18 compares these structures on `Geographic names`. For clarity, we only include the `FM-index` structures with the extreme samplings $s = 8$ and $s = 128$. When using little space (i.e., sampling steps $s \geq 128$), the `XBW` variants are clearly better than the `FM-Index`, using much less space and the same time or less. By decreasing the sampling step, `FM-Index` structures can become several times faster, however, but this comes at a steep price in space. As a matter of fact, `FMI-rrr` is not really attractive compared to `XBW-rg`: it needs to reach $s = 8$ in order to reduce the time by 25%, but at the cost of increasing the space 2.5 times. On the other hand, `FMI-rg` uses twice the space of `XBW-rg` already with $s = 128$, and to become significantly faster (5 times with $s = 8$) it uses 130% of space. These numbers endorse `XBW-rg` as a good alternative for substring lookups in a general scenario, but `FMI-rg` with a small sampling step is the choice when space requirements are more flexible.

10.6. Construction costs

Our techniques focus on compressing and querying *static* dictionaries. Thus, their contents do not change along time, or changes are sufficiently infrequent to allow a reconstruction from scratch. Although we have not engineered the construction process of our dictionaries, it is important to have a rough idea of the construction time and space of each structure, because large variations may favor one over another.

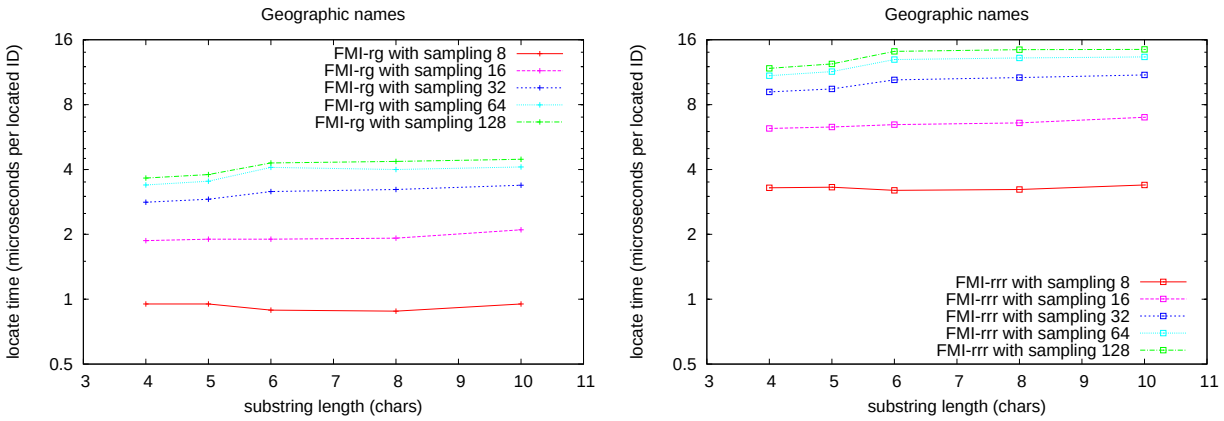

Figure 16: `locateSubstring` comparison for `Geographic names` (`FMI-rg` and `FMI-rrr`).

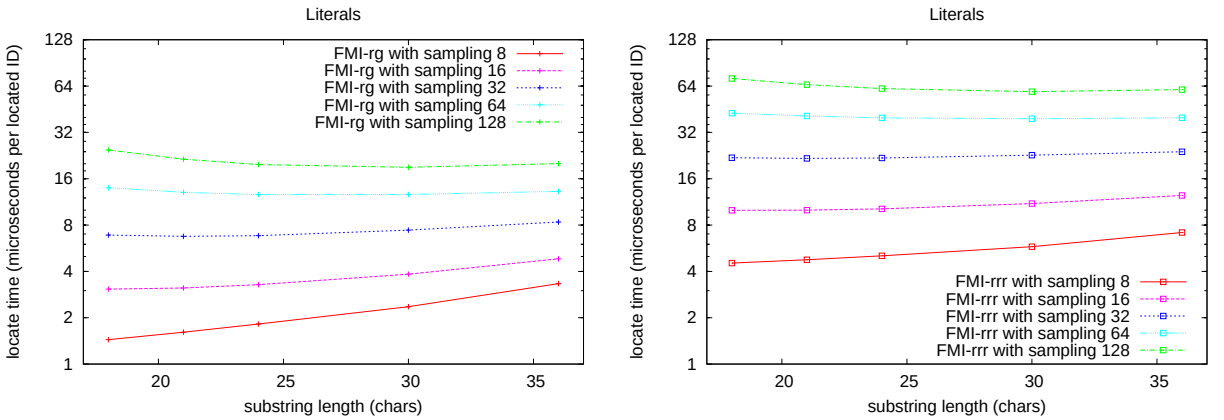

Figure 17: `locateSubstring` comparison for `Literals` (`FMI-rg` and `FMI-rrr`).

Figure 19 compares construction space and time on `DNA` and `URIs` (similar conclusions are drawn from the other datasets). Each plot represents, in the x-axis, the peak memory used for construction (in MB). The y-axis shows construction times in seconds. The most prominent choice is `PFC`, which is so fast that it could be used for online dictionary building. It processes more than 9 million `DNA` sequences in just 0.5 seconds, and almost 27 million `URIs` in just 2.5 seconds. `PFC` uses an amount of memory proportional to the dictionary size because it is fully loaded in memory (by default) before processing. Combining `HTFC` with Huffman could also be considered for online dictionaries, but it requires 2–4 times more time than `PFC` for building competitive dictionary configurations. Regarding memory, it uses similar space than `PFC` because we previously build the *plain front coding* representation and then perform Hu-Tucker and Huffman compression. `PFC` and `HTFC` on Re-Pair compression obtain moderate times, similar to those reported by

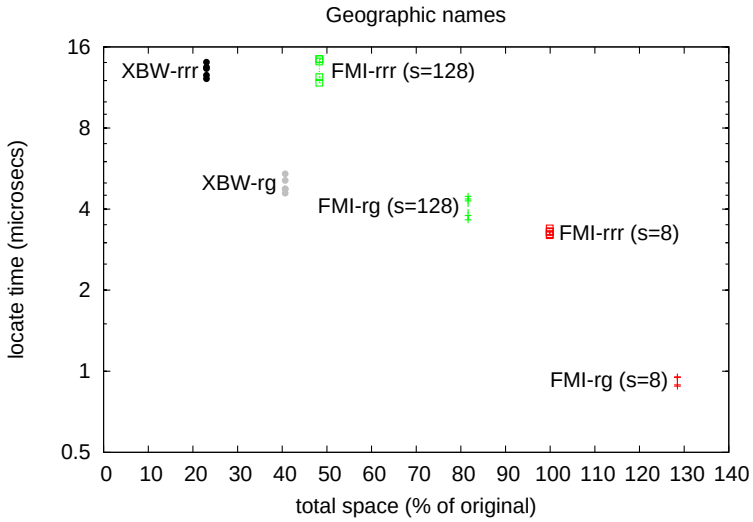

Figure 18: `locateSubstring` comparisons for `Geographic names`, with different pattern lengths.

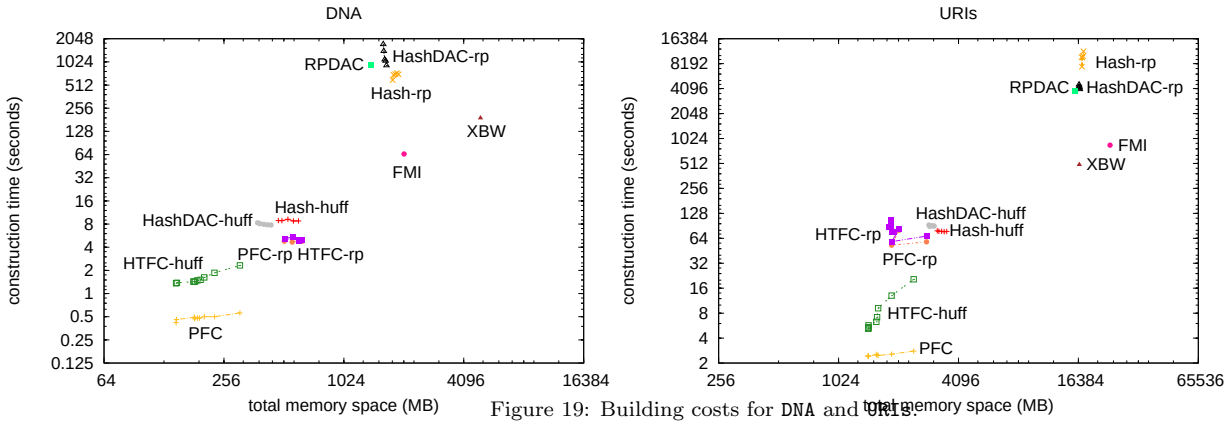

Figure 19: Building costs for `DNA` and `URIs`.

Hash-based techniques using Huffman compression. For `DNA`, building times range from 4 to 10 seconds, while they need $1-2$ minutes for `URIs`. These techniques also demand more memory because of RePair requirements and the need of reserving space for managing the (uncompressed) hash table, respectively.

Finally, the least efficient techniques are self-indexes and Hash-based dictionaries using Re-Pair compression. The latter need between 10 and 24 minutes for `DNA`, and up to 3 hours for `URIs`. Such a result may discourage the use of Re-Pair compressed hashing schemes in applications where the time for construction is limited. The responsible for this high construction cost, in the first case, is the Re-Pair compression algorithm, which is linear-time but still rather slow. Note that its high construction time loses importance when combined with Front-Coding, as it has to work on the much shorter sequences that are output by this

encoder. On the other hand, the amount of memory used in self-indexes increases due to the need of building the suffix array of the dictionary.

11. Conclusions and Future Work

String dictionaries have been traditionally implemented using classical data structures such as sorted arrays, hashing or tries. However, these solutions are falling short in facing the new scalability challenges brought up by modern data-intensive applications. Managing string dictionaries in compressed storage is becoming a key technique to handle the large datasets that are emerging within fast main memory.

This paper studies the problem of representing and managing string dictionaries from a practical perspective. By combining in various ways classical techniques (sorted arrays, hashing, tries) with various compression methods (such as Huffman, Hu-Tucker, Front-Coding and Re-Pair) and compressed data structures (such as bit and symbol sequences, directly addressable codes, full-text indexes, and compressed labeled trees), we derive five families of compressed dictionaries, each with several variants. These approaches are studied with thorough experiments on a heterogeneous testbed that comprises dictionaries arising in real-life applications, including natural language, URLs, RDF data, and biological sequences.

The results display a broad range of space/time tradeoffs, enabling applications to choose the technique that best suits their needs. Depending on the type of dictionary, our experiments show that it is possible to compress them up to 5%, 10%, or 30% of their original space, while supporting within a few microseconds the most basic operations of locating a string in the dictionary and extracting the string corresponding to a given ID. The best techniques, dominating the space/time tradeoff map, turn out to be variants of binary searching that compress the dictionary using combinations of Hu-Tucker, Front-Coding, and/or Re-Pair. A variant combining hashing with directly addressable codes and Re-Pair generally achieves better times while using more space. We also compared our techniques with the few compressed dictionary data structures available in the literature [42, 4], showing that a compressed variant of the trie data structure combined with Re-Pair [42] is also competitive and shows up in the map of the dominant techniques.

We have also studied more sophisticated prefix and substring-based searches, which are supported only by some of the proposed techniques. These operations open the door to more complex uses of dictionaries in applications. For instance, substring-based lookups (within the dictionary) have been proposed for pushing-up filter evaluation within SPARQL query processors [58], reducing the amount of data to be explored in the query and thereby improving the overall query performance. While prefix-based searches only exclude hashing-based techniques, only full-text indexing data structures (on strings and trees) are able to cope with substring-based searches. While these structures achieve good space usage, they are an order of magnitude slower than our best approaches that handle the basic operations and prefix searches. Finding more efficient data structures for these more complex operations is an interesting open problem. It is also interesting to study other complex searches that can be supported. For example, full-text indexes can be modified to allow for simultaneous prefix- and suffix-searching [35].

We plan to incorporate the proposed techniques in different types of applications. Currently, our results have been successfully used for word indexes [27] and RDF-based solutions [28], as well as for speeding up biological indexes [22]. Besides these, compressed string dictionaries could be a powerful tool for restricted computational configurations such as mobile devices. We are currently considering their use for applications running on smartphones or GPS devices.

References

[1] Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. of the 33th International Conference on Management of Data (SIGMOD)*, pages 671–682, 2006.

[2] Alberto Apostolico and Guido Drovandi. Graph compression by BFS. *Algorithms*, 2:1031–1044, 2009.

[3] Mario Arias, Javier D. Fernández, and Miguel A. Martínez-Prieto. An empirical study of real-world SPARQL queries. In *Proc. of the 1st International Workshop on Usage Analysis and the Web of Data (USEWOD)*, 2011. Available at `http://arxiv.org/abs/1103.5043`.

[4] Julian Arz and Johannes Fischer. LZ-compressed string dictionaries. In *Proc. of the Data Compression Conference (DCC)*, pages 322–331, 2014.

[5] Yasuhito Asano, Yuya Miyawaki, and Takao Nishizeki. Efficient compression of Web graphs. In *Proc. of the 14th Annual International Conference on Computing and Combinatorics (COCOON)*, pages 1–11, 2008.

[6] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 2nd edition, 2011.

[7] Hannah Bast, Christian Worm Mortensen, and Ingmar Weber. Output-sensitive autocompletion search. *Information Retrieval*, 11(4):269–286, 2008.

[8] Djamal Belazzougui, Fabiano C. Botelho, and Martin. Dietzfelbinger. Hash, displace, and compress. In *17th Annual European Symposium on Algorithms (ESA)*, LNCS 5757, pages 682–693, 2009.

[9] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[10] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 2001.

[11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proc. of the 20th International Conference on the World Wide Web (WWW)*, pages 587–596, 2011.

[12] Paolo Boldi and Sebastiano Vigna. The Webgraph framework I: Compression techniques. In *Proc. of the 13th International World Wide Web Conference (WWW)*, pages 595–2562, 2004.

[13] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, 38(1):108–131, 2013.

[14] Leonid Boytsov. Indexing methods for approximate dictionary searching: Comparative analysis. *ACM Journal of Experimental Algorithmics*, 16(1):article 1, 2011.

[15] Nieves R. Brisaboa, Rodrigo Cánovas, Francisco Claude, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Compressed string dictionaries. In *Proc. of the 10th International Symposium on Experimental Algorithms (SEA)*, pages 136–147, 2011.

[16] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing and Management*, 49(1):392–404, 2013.

[17] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the Web. *Computer Networks*, 33:309–320, 2000.

[18] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[19] Stefan Büttcher, Charles L.A. Clarke, and Gordon Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[20] Chris Callison-Burch, Collin Bannard, and Josh Schroeder. Scaling phrase-based statistical machine translation to larger corpora and longer phrases. In *Proc. of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 255–262, 2005.

[21] Moses Charikar, E. Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[22] Francisco Claude, Antonio Fariña, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proc. of the 10th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 86–91, 2010.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[24] Debora Donato, Luigi. Laura, Stefano Leonardi, and Stefano Millozzi. Algorithms and experiments for the Webgraph. *Journal of Graph Algorithms and Applications*, 10(2):219–236, 2006.

[25] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21:246–260, 1974.

[26] Robert Fano. On the number of bits required to implement an associative memory. Memo 61, Computer Structures Group, Project MAC, Massachusetts, 1971.

[27] Antonio Fariña, Nieves Brisaboa, Gonzalo Navarro, Francisco Claude, Ángeles Places, and Eduardo Rodríguez. Word-based self-indexes for natural language text. *ACM Transactions on Information Systems*, 30(1):article 1, 2012.

[28] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary RDF representation for publication and exchange. *Journal of Web Semantics*, 19:22–41, 2013.

[29] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rosano Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics*, 13:article 12, 2009.

[30] Paolo Ferragina and Roberto Grossi. The String B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[31] Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey S. Vitter. On searching compressed string collections cache-obliviously. In *Proc. of the 27th Symposium on Principles of Database Systems (PODS)*, pages 181–190, 2008.

[32] Paolo Ferragina, Fabrizio. Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–196, 2005.

[33] Paolo Ferragina and Giovanni Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[34] Paolo Ferragina, Giovanni Manzini, Velli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.

[35] Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1):article 10, 2010.

[36] Edward Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[37] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[38] Rodrigo González, Szymon Grabowski, Velli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proc. of the 4th Workshop on Experimental Algorithms (WEA)*, pages 27–38, 2005.

[39] Rodrigo González and Gonzalo Navarro. Compressed text indexes with fast locate. In *Proc. of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.

[40] Szymon Grabowski and Wojciech Bieniecki. Merging adjacency lists for efficient Web graph compression. *Advances in Intelligent and Soft Computing*, 103(1):385–392, 2011.

[41] Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. High-order entropy-compressed text indexes. In *Proc. of the 14th Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[42] Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. In *Proc. of the 14th Meeting on Algorithm Engineering & Experiments (ALENEX)*, pages 65–74, 2012.

[43] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 2007.

[44] Harold S. Heaps. *Information Retrieval: Computational and Theoretical Aspects.* Academic Press, 1978.

[45] T.C. Hu and A.C. Tucker. Optimal computer-search trees and variable-length alphabetic codes. *SIAM Journal of Applied Mathematics*, 21:514–532, 1971.

[46] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[47] Artur Jez. A really simple approximation of smallest grammar. In *Proc. 25th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 8486, pages 182–191, 2014.

[48] John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[49] Jon M. Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S. Tomkins. The Web as a graph: Measurements, models, and methods. In *Proc. of the 5th Annual International Conference on Computing and Combinatorics (COCOON)*, pages 1–17, 1999.

[50] Donald E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching.* Addison Wesley, 1973.

[51] Philipp Koehn, Franz Josef Och, and Daniel Marcu. Statistical phrase-based translation. In *Proc. of the Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology (NAACL)*, pages 48–54, 2003.

[52] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. *Proceedings of the IEEE*, 88:1722–1732, 2000.

[53] Mike Liddell and Alistair Moffat. Decoding prefix codes. *Software Practice and Experience*, 36(15):1687–1710, 2006.

[54] Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 32, 2008.

[55] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[56] Christopher D. Manning and Hinrich Schtze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[57] Frank Manola and Eric Miller, editors. *RDF Primer*. W3C Recommendation, 2004. `www.w3.org/TR/rdf-primer/`.

[58] Miguel A. Martínez-Prieto, Javier D. Fernández, and Rodrigo Cánovas. Querying RDF dictionaries in compressed space. *ACM SIGAPP Applied Computing Review*, 12(2):64–77, 2012.

[59] Shirou Maruyama, Hiroshi Sakamoto, and Masayuki Takeda. An online algorithm for lightweight grammar-based compression. *Algorithms*, 5(2):214–235, 2012.

[60] Joong C. Na and Kunsoo Park. Simple implementation of String B-Trees. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 3246, pages 214–215, 2004.

[61] Naresh Kumar Nagwani. Clustering based URL normalization technique for Web mining. In *Proc. of the International Conference Advances in Computer Engineering (ACE)*, pages 349–351, 2010.

[62] Gonzalo Navarro and Velli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[63] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.

[64] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. of ALENEX*, pages 60–70, 2007.

[65] Adam Pauls and Dan Klein. Faster and smaller n-gram language models. In *Proc. of HLT*, pages 258–267, 2011.

[66] Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M. Tiedje, and C. Titus Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[67] Eric Prud'hommeaux and Andy Seaborne, editors. *SPARQL Query Language for RDF*. `http://www.w3.org/TR/rdf-sparql-query/`. W3C Recommendation, 2008.

[68] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of the 13th Annual ACM-SIAM Aymposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[69] Gábor Rétvári, János Tapolcai, Attila Kőrösi, András Majdán, and Zalán Heszberger. Compressing IP forwarding tables: towards entropy bounds and beyond. In *Proc. of the ACM SIGCOMM Conference*, pages 111–122, 2013.

[70] Einar Andreas Rødland. Compact representation of k-mer de Bruijn graphs for genome read assembly. *BMC Bioinformatics*, 14(1):1–19, 2013.

[71] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[72] Hiroo Saito, Masashi Toyoda, Masaru Kitsuregawa, and Kazuyuki Aihara. A large-scale study of link spam detection by graph algorithms. In *Proc. of the 3rd International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, 2007.

[73] Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(24):416 – 430, 2005.

[74] David Salomon. *A Concise Introduction to Data Compression*. Springer, 2008.

[75] Eugene S. Schwartz and Bruce Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.

[76] Torsten Suel and Jun Yuan. Compressing the graph structure of the Web. In *Proc. of the Data Compression Conference (DCC)*, pages 213–222, 2001.

[77] Jacopo Urbani, Jason Maassen, and Henri Bal. Massive semantic web data compression with mapreduce. In *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, pages 795–802, 2010.

[78] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, 42:193–201, 1999.

[79] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[80] Ming Yin Yin, Dion Hoe-lian Goh, Ee-Peng Lim, and Aixin Sun. Discovery of concept entities from Web sites using web unit mining. *International Journal of Web Information Systems*, 1(3):123–135, 2005.

[81] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

A. Experimental Results

The following subsections comprise `locate` and `extract` graphs (*i*) for compressed hash dictionaries, (*ii*) for Front-Coding dictionaries, (*iii*) `locatePrefix` and `extractPrefix` (except for `Literals`), and (*iv*) `locatePrefix` and `extractPrefix` (only for `Geographic names` and `Literals`).

A.1. Geographic Names

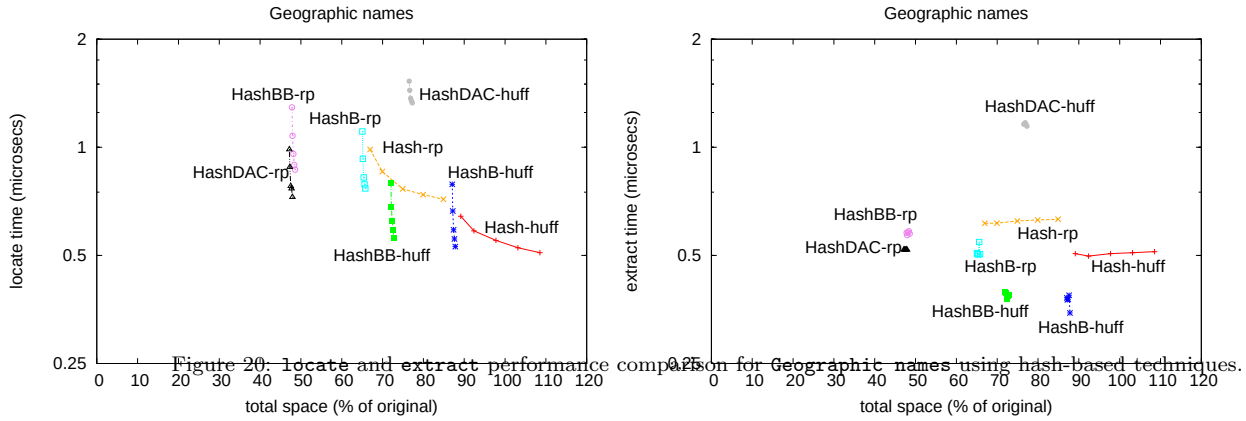

Figure 20: `locate` and `extract` performance comparison for `Geographic names` using hash-based techniques.

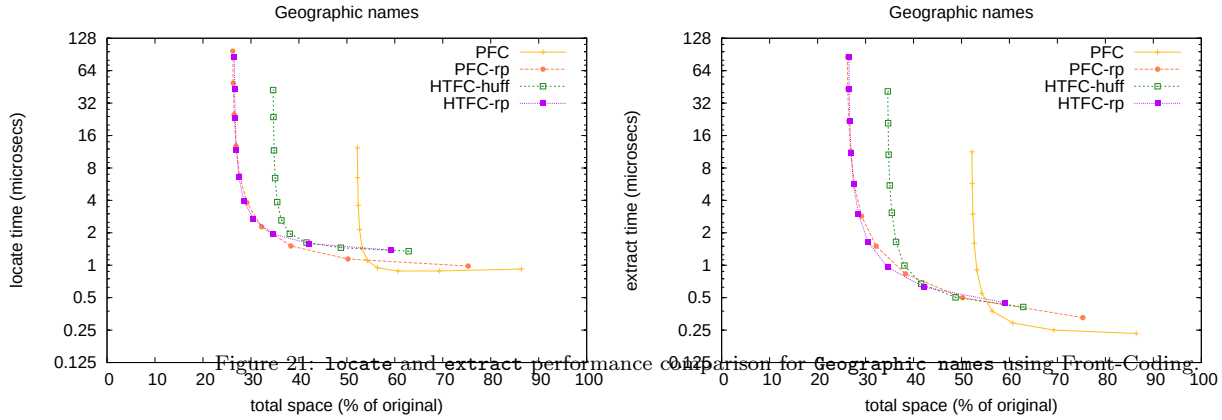

Figure 21: `locate` and `extract` performance comparison for `Geographic names` using Front-Coding.

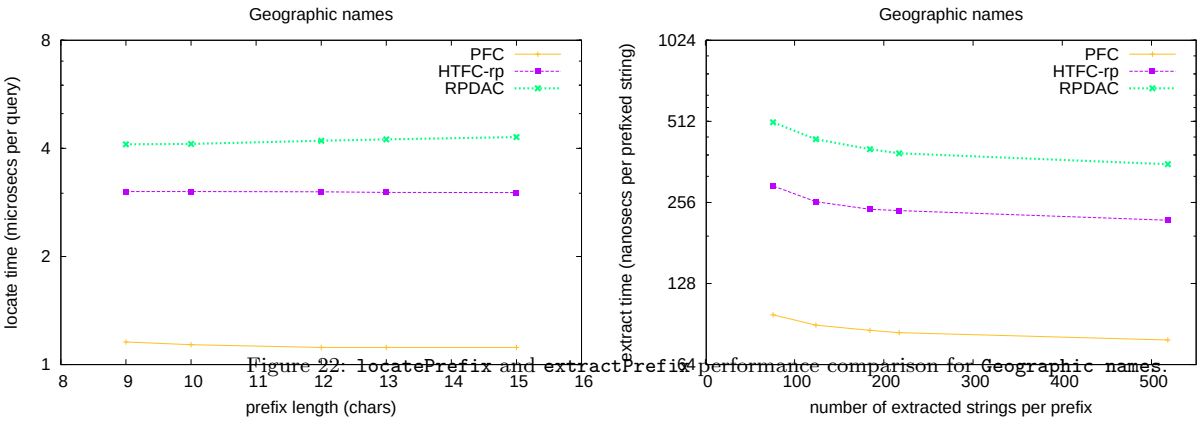

Figure 22: `locatePrefix` and `extractPrefix` performance comparison for `Geographic names`.

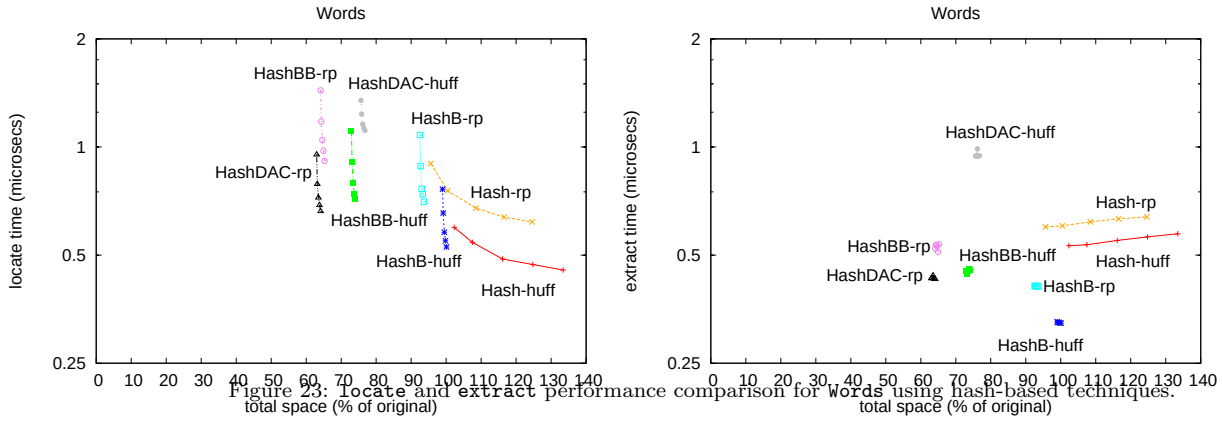

Figure 23: `locate` and `extract` performance comparison for `Words` using hash-based techniques.

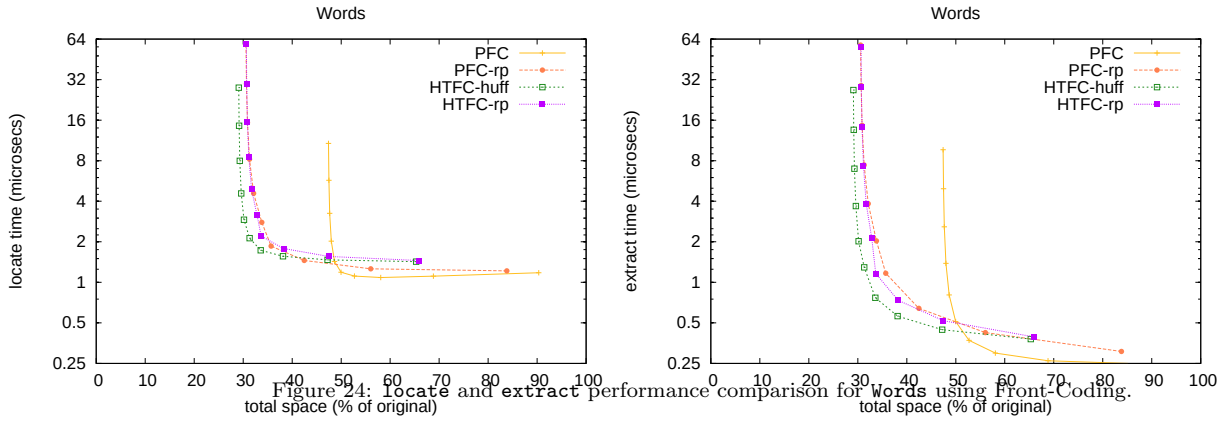

Figure 24: `locate` and `extract` performance comparison for `Words` using Front-Coding.

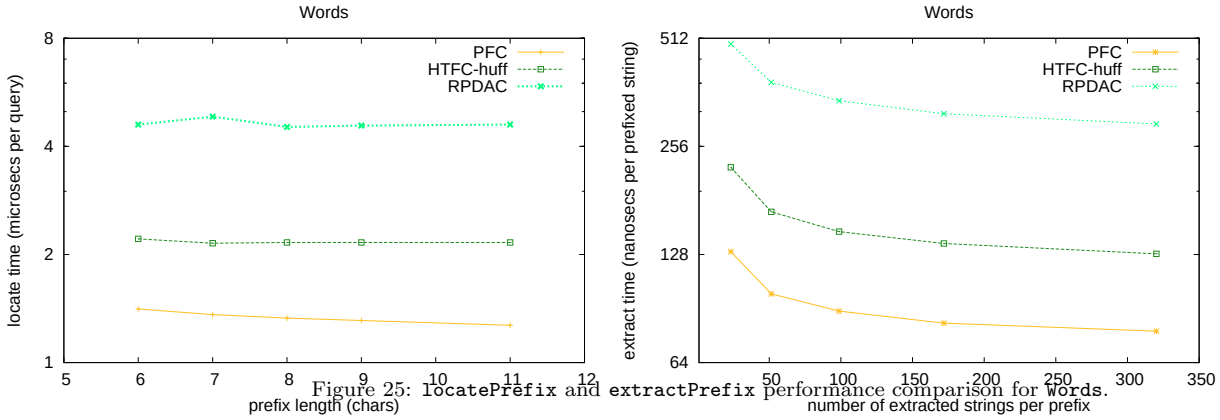

Figure 25: `locatePrefix` and `extractPrefix` performance comparison for `Words`.

A.3. Word sequences (English)

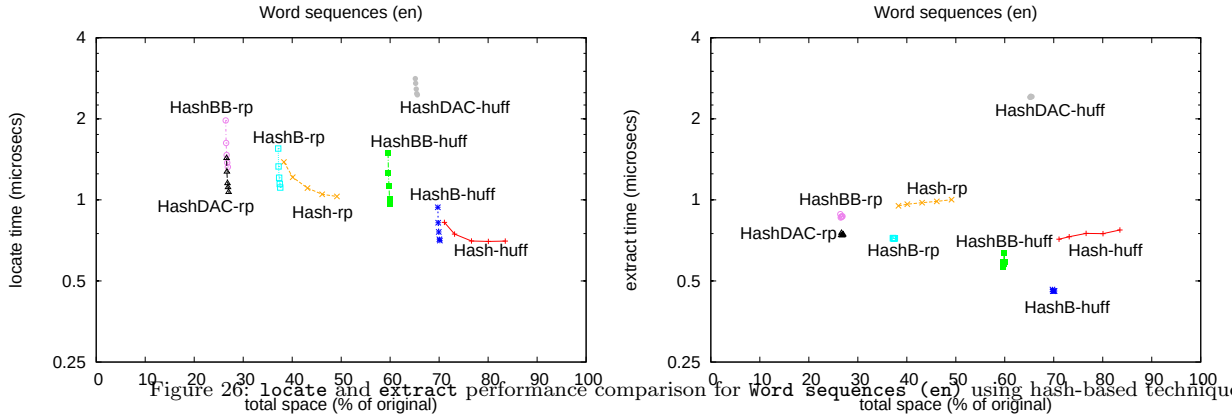

Figure 26: `locate` and `extract` performance comparison for `Word sequences (en)` using hash-based techniques.

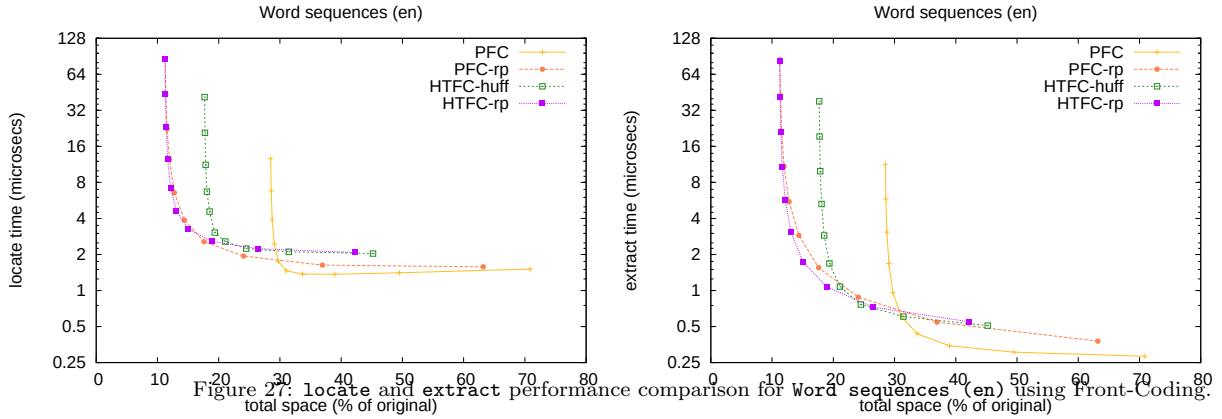

Figure 27: `locate` and `extract` performance comparison for `Word sequences (en)` using Front-Coding.

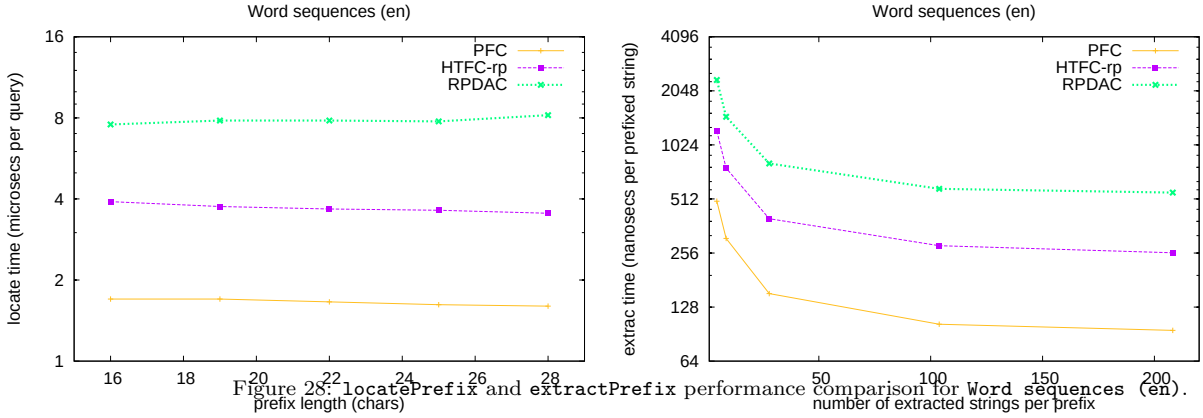

Figure 28: `locatePrefix` and `extractPrefix` performance comparison for `Word sequences (en)`.

A.4. Word sequences (Spanish)

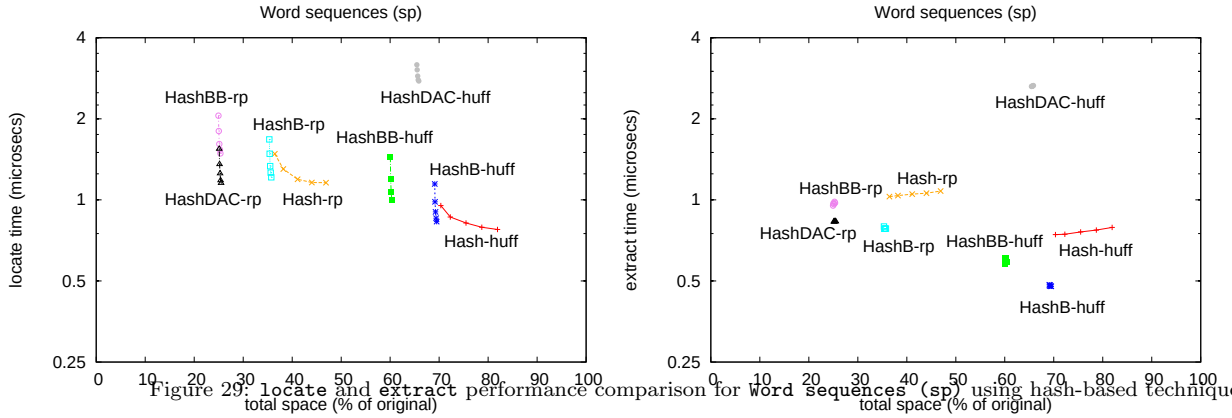

Figure 29: `locate` and `extract` performance comparison for `Word sequences (sp)` using hash-based techniques.

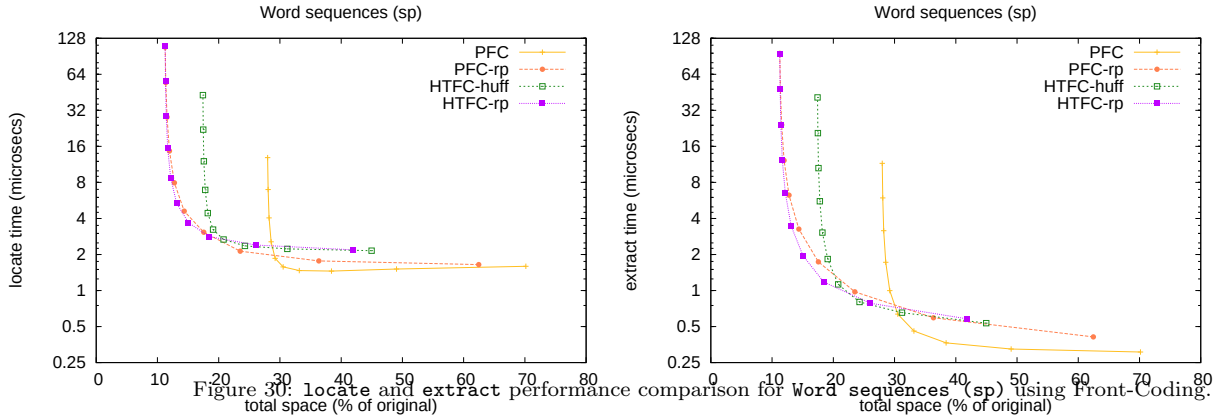

Figure 30: `locate` and `extract` performance comparison for `Word sequences (sp)` using Front-Coding.

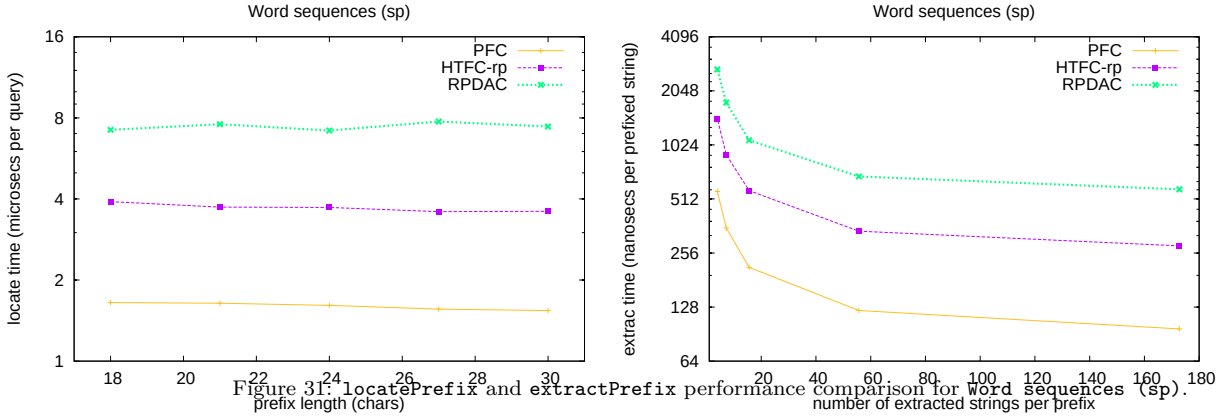

Figure 31: `locatePrefix` and `extractPrefix` performance comparison for `Word sequences (sp)`.

A.5. URIs

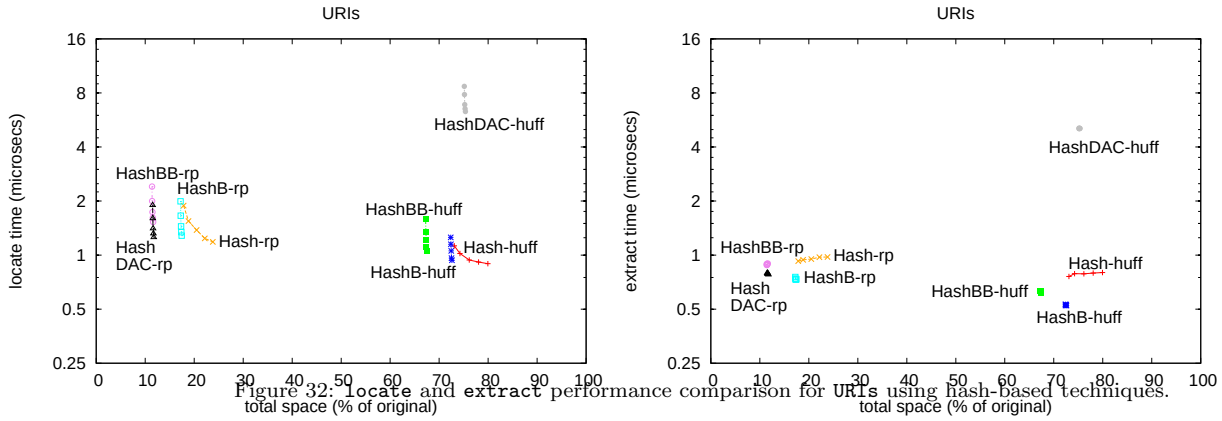

Figure 32: `locate` and `extract` performance comparison for `URIs` using hash-based techniques.

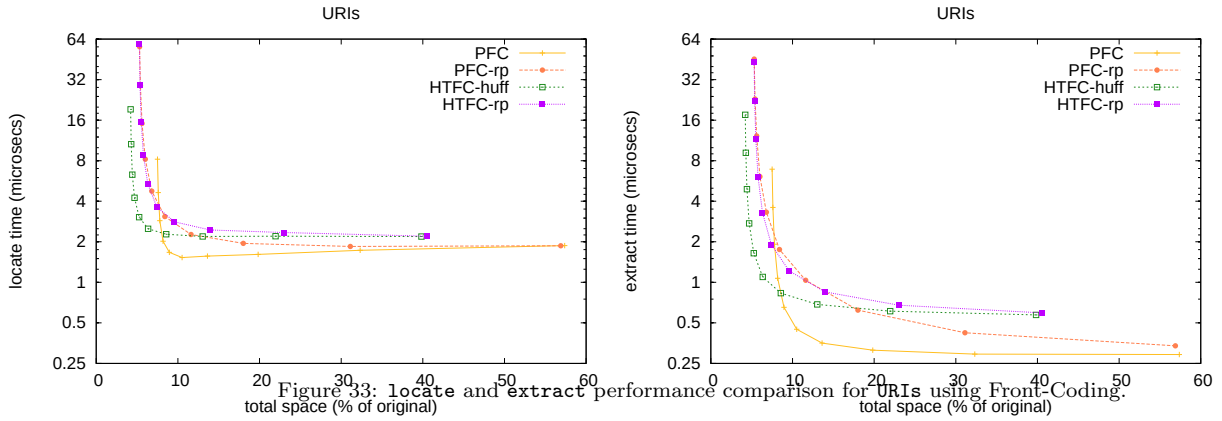

Figure 33: `locate` and `extract` performance comparison for `URIs` using Front-Coding.

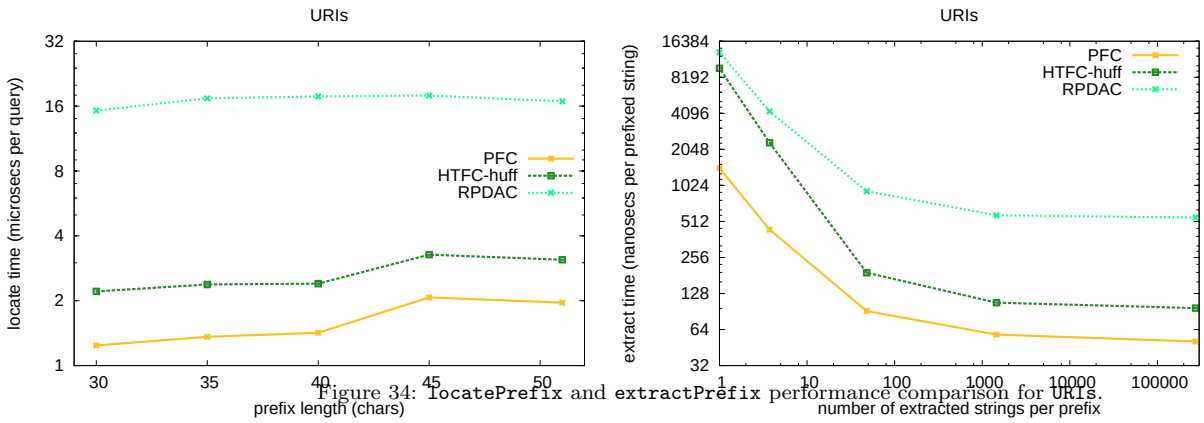

Figure 34: `locatePrefix` and `extractPrefix` performance comparison for `URIs`.

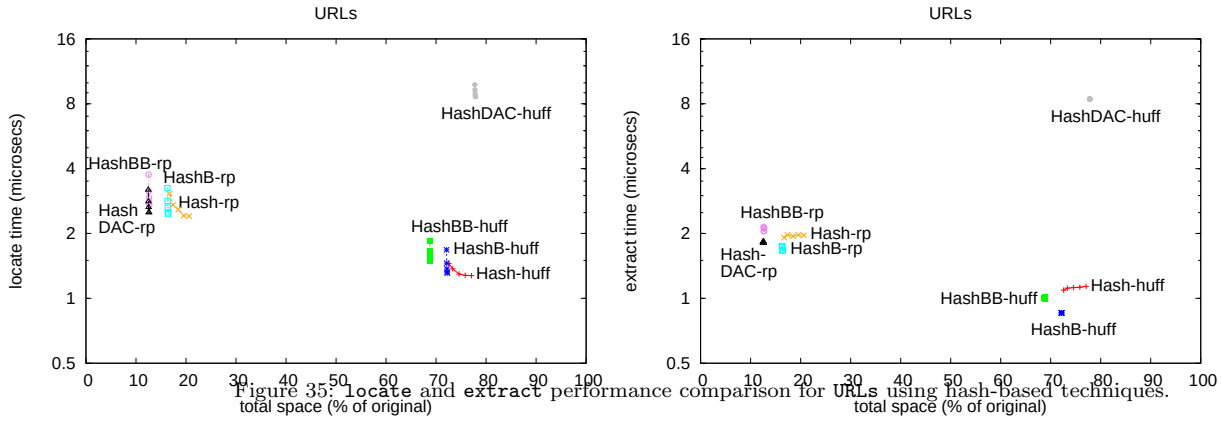

Figure 35: `locate` and `extract` performance comparison for `URLs` using hash-based techniques.

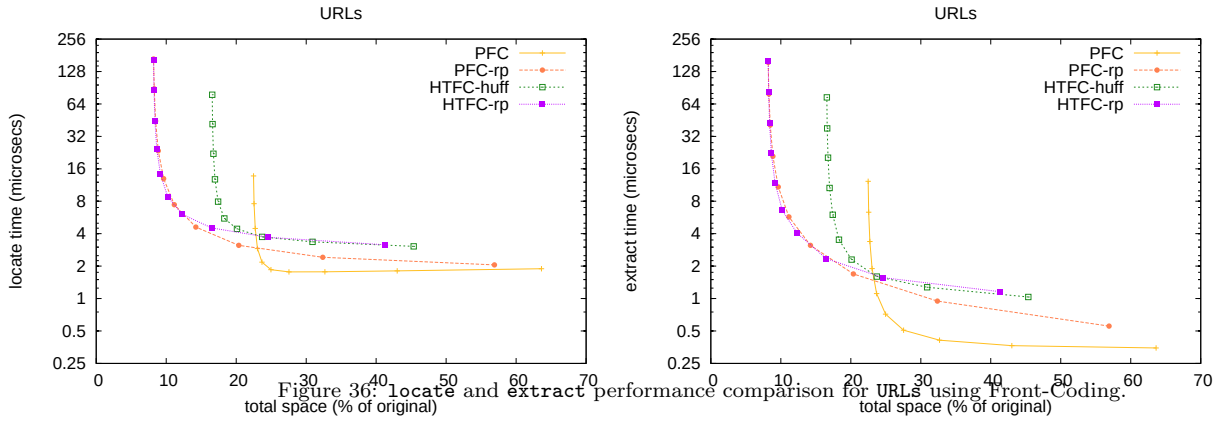

Figure 36: `locate` and `extract` performance comparison for `URLs` using Front-Coding.

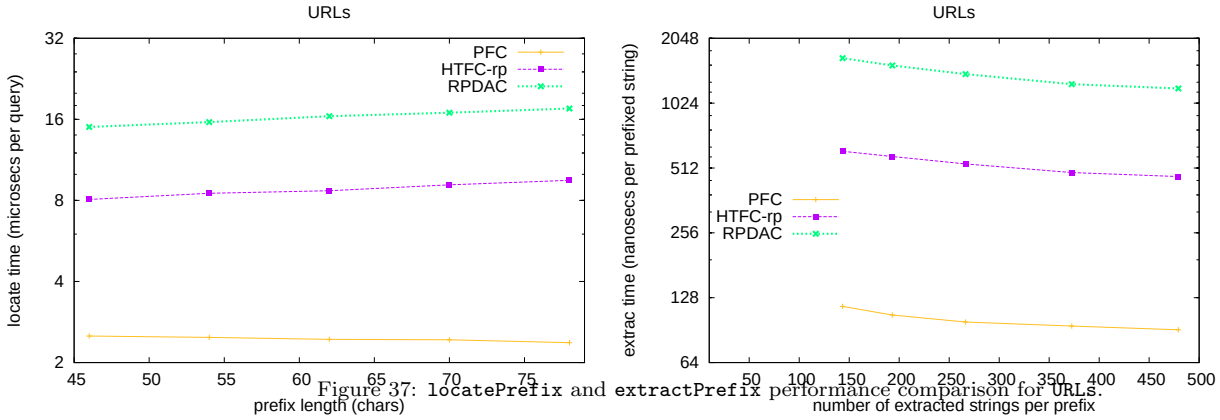

Figure 37: `locatePrefix` and `extractPrefix` performance comparison for `URLs`.

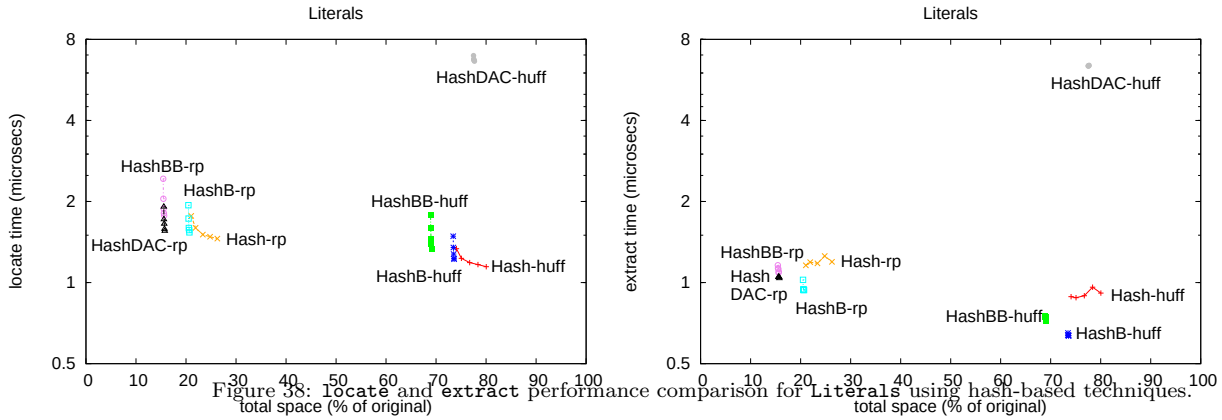

Figure 38: `locate` and `extract` performance comparison for `Literals` using hash-based techniques.

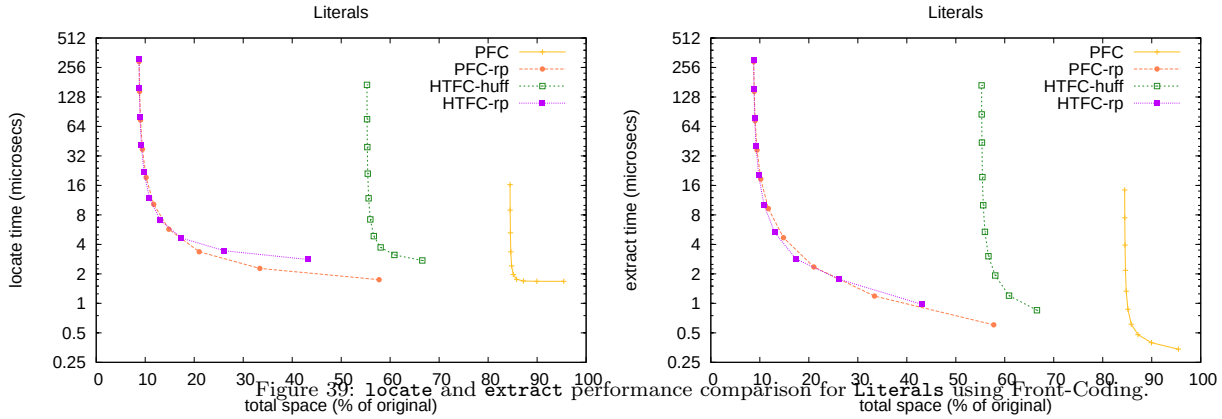

Figure 39: `locate` and `extract` performance comparison for `Literals` using Front-Coding.

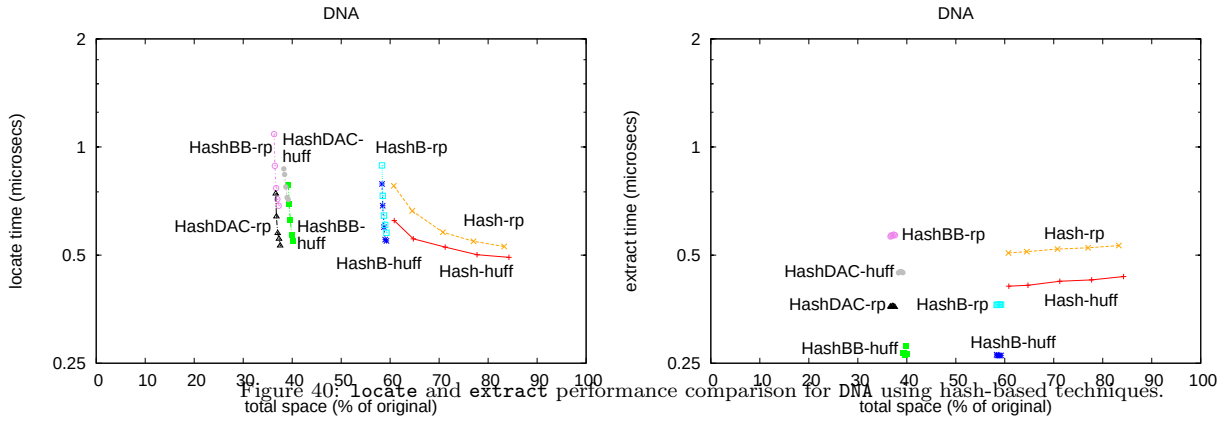

Figure 40: `locate` and `extract` performance comparison for `DNA` using hash-based techniques.

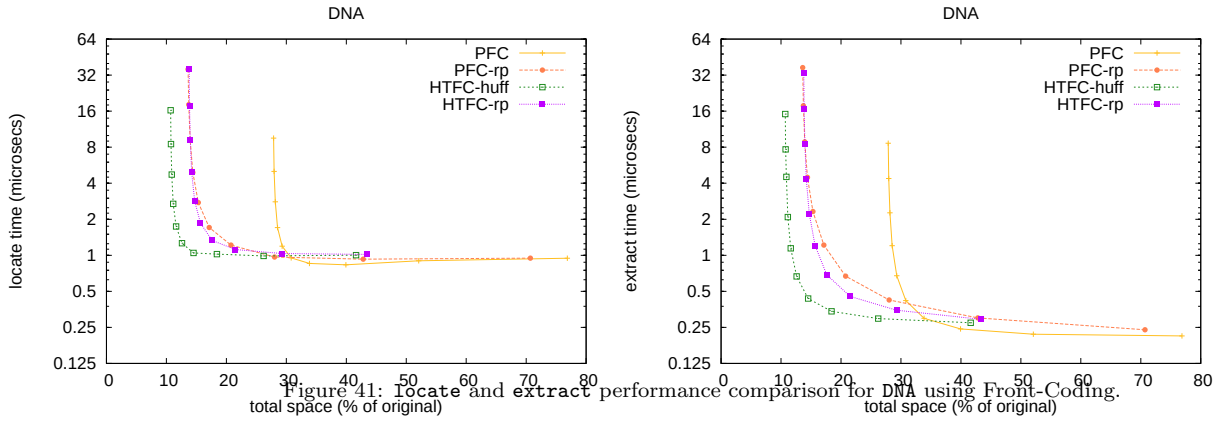

Figure 41: `locate` and `extract` performance comparison for `DNA` using Front-Coding.

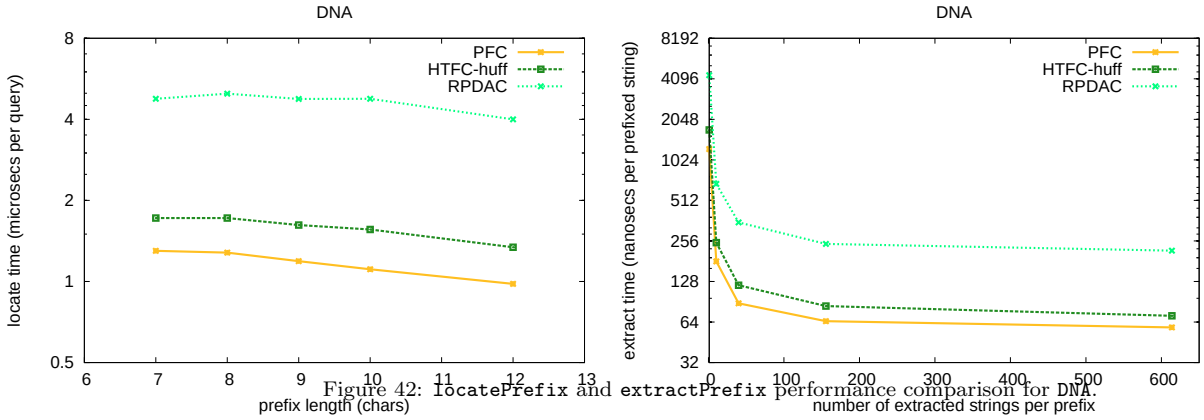

Figure 42: `locatePrefix` and `extractPrefix` performance comparison for `DNA`.