Improved Compressed String Dictionaries

Anonymous Author(s)

ABSTRACT

We introduce a new family of compressed data structures to efficiently store and query large string dictionaries in main memory. Our main technique is a combination of hierarchical Front-coding with ideas from longest-common-prefix computation in suffix arrays. Our data structures yield relevant space-time tradeoffs in real-world dictionaries. We focus on two domains where string dictionaries are extensively used and efficient compression is required: URL collections, a key element in Web graphs and applications such as Web mining; and collections of URIs and literals, the basic components of RDF datasets. Our experiments show that our data structures achieve better compression than the state-of-the-art alternatives while providing very competitive query times.

CCS CONCEPTS

• **Information systems → Data compression**; **Dictionaries**; *Web indexing*; *Resource Description Framework (RDF)*.

KEYWORDS

compression, data structures, string dictionaries

ACM Reference Format:

Anonymous Author(s). 2019. Improved Compressed String Dictionaries. In *CIKM '19: ACM International Conference on Information and Knowledge Management, November 03–07, 2019, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/xxxx

1 INTRODUCTION

A string dictionary is essentially a bidirectional mapping between strings and identifiers. Those identifiers are usually consecutive integer numbers that can be interpreted as the position of the string in the dictionary. By using string dictionaries, applications no longer need to store multiple references to large collections of strings. Replacing those strings, which can be long and have different lengths, with simple integer values simplifies the management of this kind of data.

Many applications need to make use of large collections of strings. The most immediate ones are text collections and full-text indexes, but several other applications, not specifically related to text processing, still require an efficient representation of string collections. Some relevant examples include those handling Web graphs, ontologies and RDF datasets, natural language collections and biological sequences. Web graphs, for example, store a graph representing hyperlinks between Web pages, so the node identifiers are URLs. Most representations transform those strings into numeric identifiers (ids), and then store a graph referring to those ids. Compact Web graph representations can store the graphs within just a few bits per edge [3, 5]. Since the average node arities are typically 15–30, storing the URL of the node becomes in practice a large fraction of the overall space. In RDF datasets, information is stored as a labeled graph where nodes are either blank, URIs, or literal values; labels are also URIs. The usual approach to store RDF data is also to use string dictionaries to obtain numeric identifiers for each element, in order to save space and speed up queries [17]. The classical technique of storing a string dictionary is extended in some proposals by keeping separate dictionaries for URIs and literal values [15].

In this paper we consider the problem of efficiently storing large static string dictionaries in compressed space in main memory, providing efficient support for two basic operations: *lookup(s)* receives a string and returns the string identifier, an integer value representing its position in the dictionary; *access(i)* receives a string identifier and returns the string in the dictionary corresponding to that identifier.

We focus on two types of dictionaries that are widely used in practical applications: URL dictionaries used in Web graphs, which are of special interest for many Web analysis and retrieval tasks; and URIs and literals dictionaries for RDF collections, which are a key component of the Web of Data and the Linked Data initiative and have experienced a sharp growth in recent years.

Our techniques achieve compression by exploiting repetitiveness among the strings, so they are especially well suited to URL and URI datasets where individual strings are relatively long and very similar to other strings close to them in lexicographical order. In particular, we build on Front-coding, which exploits long common prefixes between consecutive strings, and design a hierarchical version that enables binary searches without using any sampling. We enhance this binary search with techniques inherited from suffix array construction algorithms, which boost the computation of longest common prefixes along lexicographic ranges of strings. These main ideas are then composed with other compression techniques.

Experimental results on real-world datasets show that our data structures achieve better compression than the state-of-the-art alternatives, and we are much faster than the few alternatives that can reach similar compression. Even if faster solutions exist, our techniques are still competitive in query times and significantly smaller than them.

The remaining of this paper is organized as follows: in Section 2 we introduce some concepts and refer to previous work in string dictionary compression. Section 3 presents our proposal, describing the structure and query algorithms and explaining the main variants implemented. Section 4 contains the experimental evaluation of our structures. Finally, Section 5 summarizes the results and shows some lines for future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIKM '19, November 03–07, 2019, Beijing, China

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN xxxx…$15.00
https://doi.org/xxxx

2 RELATED WORK

2.1 Previous concepts and basic compression techniques

In this section we introduce some preliminary concepts, presenting existing data structures and compression techniques that are used in the paper.

2.1.1 Bit sequences. A bit sequence or bitmap is a sequence $B[1, n]$ of n bits. Bit sequences are widely used in many compact data structures. Usually, bit sequences provide the following three basic operations: $access(B, i)$ obtains the value of the bit at position i, $rank_v(B, i)$ counts the number of bits set to v up to position i, and $select_v(B, j)$ obtains the position in B of the j-th bit set to v. All the operations can be answered in constant time using $n + o(n)$ bits [16, Ch. 4]. Additionally, compressed bit sequence representations have been proposed to further reduce the space requirements [19]. In this paper we use an implementation of the SDArray compressed bitmap [18] provided by the Compact Data Structures Library `libcds`[1]. This solution can achieve compression when the sequence is sparse and still supports *select* queries in constant time.

2.1.2 Integer compression techniques. In this paper we use Variable-byte (Vbyte) encoding [21], a simple integer compression technique that essentially splits an integer in 7-bit chunks, and stores them in consecutive bytes, using the most significant bit of each byte to mark whether the number has more chunks or not. It is simple to implement and fast to decode.

A technique of special relevance is Directly Addressable Codes (DACs) [4]. This technique aims at storing a sequence of integers in compressed space while providing direct access to any position. Given the Vbyte encoding of the integers, DACs store the first chunk of each integer consecutively, and use a bitmap B_1 to mark the entries with a second chunk. The process is repeated with the second chunks and its corresponding bitmap B_2, and so on. DACs support decompressing entries accessing the first chunk directly and using $rank_1$ operations on the B_is to locate the corresponding position of the next chunk.

DACs can work with Vbyte encoding but they are actually a general chunk-reordering technique. In this paper we make use of a variant that is designed to store a collection of variable-length integer sequences, instead of a sequence of integers. In this variant, that we call DAC-VLS, integers are not divided in chunks; instead, the first integer in each sequence is stored in the first level, and a bitmap is used to mark whether the current sequence has more elements. This technique does not reduce the space of the original integers, but provides direct access to any sequence in the collection.

2.1.3 String compression: Front-coding and Re-Pair. Front-coding is a folklore compression technique that is used as a building block in many well-known compression algorithms. Front-coding compresses a string s relative to another s_0 by computing their longest common prefix (lcp) and removing the first lcp characters from the encoded string. Hence, Front-coding represents s as a tuple containing the lcp and the substring after it $\langle lcp, s[lcp..len(s)]\rangle$. Despite its simplicity, it is a very useful technique for many applications, providing a simple way to compress collections of similar strings. URLs, for instance, tend to have relatively long common prefixes, so Front-coding compression is very effective on them, even if the string portions remaining after Front-coding, or string tails, are still relatively long.

Re-Pair [12] is a grammar compression technique that achieves good compression in practice for different kinds of texts. Given a text T, Re-Pair finds the most repeated pair of consecutive symbols ab and replaces each occurrence of ab by a new symbol R, adding to the grammar a new rule $R \rightarrow ab$. The process is repeated until no repeated pairs appear in the text. The output of Re-Pair is a list of r rules and the resulting reduced text T^C, represented as a sequence of integers in the range $(1, \sigma + r)$, where σ is the number of different symbols in the original text.

2.2 String dictionary compression

Simple techniques for storing collections of strings have been used in many applications. Hash tables and tries [11] are just some examples of classical representations that can be used in main memory for small dictionaries.

As the dictionary size increases, those classical data structures no longer fit in main memory, so a compressed representation has to be used or the dictionary must be stored in secondary memory. A simple approach to reduce space is to compress individual strings using general or domain-specific compression techniques, before adding them to the dictionary structure. Modern techniques for dictionary compression are based on specific compact data structures usually combined with custom compression techniques applied to the strings. Several theoretical solutions have been proposed for static dictionaries [2], and solutions also exist for the dynamic dictionary problem [9, 10, 20]. In this section we will focus on practical solutions for a static dictionary, outlining the most relevant existing implementations.

Martinez-Prieto et al. [14] have proposed a collection of compressed string dictionary representations that provide a choice for different space/time tradeoffs. In their survey, they show advantages against proposals based on compressed tries and similar compression techniques. Their representations are based on well-known compression techniques that are combined to build space-efficient versions of data structures like tries and hash tables. The most relevant proposal in this survey is a collection of differentially encoded dictionaries. The authors sort the strings and split them into fixed-size buckets. Then, they store the first string of each bucket, or bucket header, in full, and the remaining strings of the bucket are compressed relative to the previous one using Front-coding. To answer *lookup* queries, a binary search in the bucket headers is used to locate the bucket containing the string, and a sequential search in the bucket is performed; *access* queries just traverse sequentially the bucket containing the query identifier. The authors propose several variants of this idea in the original paper that combine the previous idea with additional compression techniques like Huffman [8], Hu-Tucker [6] or Re-Pair applied to the strings in each bucket or to the bucket headers to reduce the overall space usage.

[1] https://github.com/fclaude/libcds

In the previous work several other alternatives are proposed that share similarities with our proposal. Binary-searchable Re-Pair compresses the strings with Re-Pair and uses DACs to provide direct access to each one, supporting *lookup* queries through binary search. An improvement on the same idea uses a hash table to provide direct access to the location of a string, instead of resorting to binary search, improving *lookup* queries significantly at the cost of additional space.

Grossi and Ottaviano propose a structure based on path decomposed tries (PDT) [7]. The authors create a path decomposition of the trie representing the dictionary strings, and build a compact representation of the tree generated by the path decomposition. They explore different techniques for the representation of the trie (lexicographical and centroid-based path decomposition). They also propose compressed variants in which the path labels are compressed using Re-Pair. Their solution has shown good results in different kinds of string dictionaries. Their compressed tries are competitive in space with previous techniques, but more importantly provide fast and very consistent query times.

Arz and Fischer [1] have recently proposed a solution based on Lempel-Ziv-78 (LZ-78) compression on top of PDT. This technique has been shown to slightly improve the compression of PDT in some datasets, but improvement is small in most cases and the LZ-78-compressed structures have much higher query times, especially in *lookup* queries.

3 OUR PROPOSAL

3.1 Data structure and algorithms

We propose a family of compression techniques for string collections that aim at providing good compression with efficient query times. Our techniques follow some of the ideas of differential compression described in Section 2.2 and aim at improving their weak points.

To build our representation, the strings are sorted in lexicographic order. This order is frequently used in most string dictionary representations, so that entries that are close to each other should also be similar to each other. For convenience, we also add two marker strings at the beginning and at the end of the collection: the former is the empty string, and the latter is a single-character string lexicographically larger than any string in the original collection.

Our goal is to use Front-coding to reduce the common prefix of common entries. However, instead of compressing each string relative to the previous one, we use a different scheme for comparisons that constitutes the basis of our proposal. Our technique is based on a binary decomposition of the list of strings, following similar ideas to the binary search algorithms over suffix arrays proposed by Manber and Myers [13].

Assume we have a collection C of n strings, including our initial and last string, and let $C[p_i]$ be the string at position p_i in the collection. Our structure is built as follows:

- We initialize two markers $p_\ell = 0$ and $p_r = n - 1$, set to the limits of the collection.
- We select the middle point $p_m = (p_\ell + p_r)/2$ and compute $llcp[m] = lcp(C[p_m], C[p_\ell])$ and $rlcp[m] = lcp(C[p_m], C[p_r])$, the longest common prefixes between the string at position p_m and the strings at both limits of the interval.
- Let $maxlcp$ be the maximum between $llcp[p_m]$ and $rlcp[p_m]$. $C[p_m]$ is compressed using Front-coding, by removing the $maxlcp$ initial bytes. In practice, Front-coding is applied relative to the most similar of the entries at each limit of the interval. We will refer to these as the "parents" of a given entry.
- We recurse on both halves of the collection ($[0, p_m]$ and $p_m, n-1]$), repeating the previous steps to compare the middle element with the limits of the interval and apply Front-coding accordingly.

After this procedure, our conceptual representation consists of two integer sequences $llcp$ and $rlcp$, and the remaining of each string after Front-coding is applied to them. Let us call this $S[n]$. In practice we use different techniques to store the strings, but for simplicity we will write $S[i]$ to refer to the string stored at position i.

Figure 1 shows an example of our dictionary structure for a small set of strings. We use `$` to denote a string terminator. Our marker strings are denoted as `$` and `~$` respectively. The original strings at each position are displayed below the arrays, with the prefix that would be removed after Front-coding compression grayed out. Arrows identify the position of the left and right "parent" of each entry. For instance, $C[8]$ is compared with positions 0 and 16 (our marker strings), and it is stored in full. $C[12]$ (`climate $`) is compared with $C[8]$ ($llcp = 2$) and $C[16]$ ($rlcp[8] = 0$), and after Front-coding is applied it becomes `imate$`, removing the longest common prefix. Note that the marker strings we use will never share a common prefix with any string in the collection, so both marker strings and the string in the middle position will always have $llcp$ and $rlcp$ values of 0 and will be stored in full. The final representation needs to store the $llcp$ and $rlcp$ arrays and the collection of string tails.

Our construction technique is expected to yield worst compression results than the usual Front-coding approach that would be applied sequentially to the collection of strings. We will describe later our implementation strategies to improve the space utilization. However, as we will see next, our binary decomposition allows us to provide an efficient method to answer queries without resorting to sampling or partitioning of the collection, as is necessary in other techniques.

Next we outline the algorithms for *lookup* and *access* operations. In both cases we perform a binary-search-like traversal of the collection.

3.1.1 Lookup operation. To obtain the identifier of a string in the dictionary (*lookup*), a trivial algorithm would involve a binary search, checking the midpoint at each step and comparing the resulting string with the target. However, our scheme is able to improve the performance of *lookup* operations by avoiding some string comparisons.

The pseudo-code used for *lookup* searches is described in Algorithm 1. Let s_q be the search string. The values p_ℓ and p_r are the limits of our interval, initially $p_\ell = 0$ and $p_r = n - 1$. The variables ℓ and r store the longest common prefix of the left- and right-hand strings in the dictionary with the search string and are initially set to 0. Hence, a *lookup(s_q)* is translated into *doLookup*$(s_q, 0, n - 1, 0, 0)$.

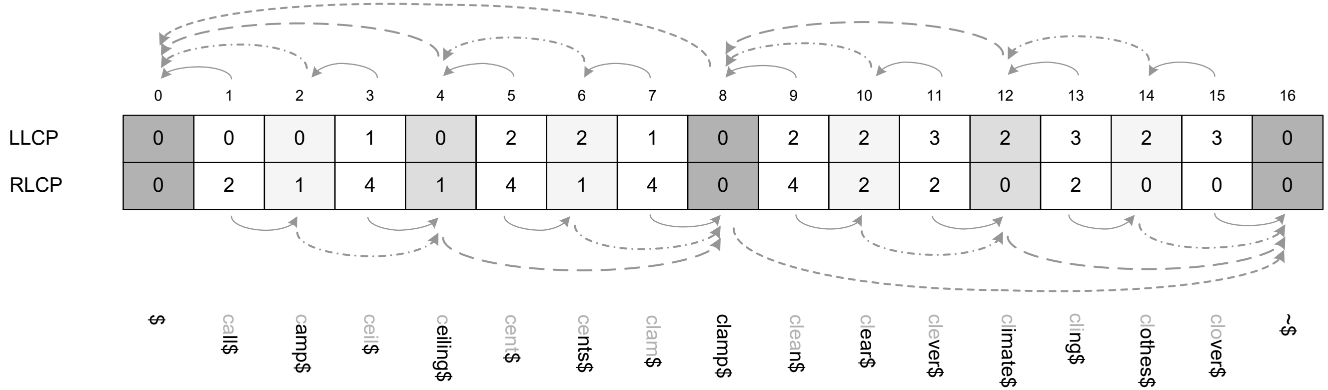

Figure 1: Conceptual dictionary structure.

At any step of search, we first compare ℓ and r. We will focus on the case $\ell >= r$ (i.e., the string at p_ℓ is more similar to s_q than the string at p_r) covered in lines 3-18 of the algorithm [2], since the other case is symmetric. We obtain the midpoint p_m and the value of $llcp[p_m]$ and then compare it with ℓ:

- If $llcp[p_m] > \ell$, entry p_m has a longer prefix in common with p_ℓ than p_ℓ with s_q. Hence, the result cannot be to the left of p_m. We recurse on the right half of the range ($[p_m, p_r]$) without comparing strings.
- If $llcp[p_m] < \ell$, we are on the symmetric case: entry p_ℓ is more similar to the pattern than to entry p_m. We recurse on the left half of the interval, and we set the new lower bound $r = llcp[p_m]$, since our current string must have $llcp[p_m]$ characters in common with the search string.
- If $llcp[p_m] = \ell$, we need to compare entry p_m with s_q. Our comparison method in Algorithm 1 gives us the two relevant pieces of information: the comparison value *cmp*, and the offset o of the last equal character. If both strings are equal, we return immediately. Otherwise, we recurse on the appropriate half, setting the value of ℓ or r to o.

Following the example in Figure 1, assume we are searching for string `clam`. First we compare with `clamp` (due to our markers, in the first iteration a comparison is always performed). The query string is smaller than $S[8]$, and they share the first 4 characters. Therefore, we recurse on the left half $[0, 8]$, setting $r = 4$. In the next step ($p_m = 4$), $r > \ell$ and $rlcp[4] = 1 < r$, so we do not need to compare strings: we just recurse on the right-side interval $[4, 8]$, and we set $\ell = 1$, since $rlcp[4] = 1$, meaning that it shares also a prefix of length 1 with s_q. In the next step ($p_m = 6$), again $r > \ell$, and $rlcp[6] = 1 < r$, so we recurse on the interval $[6, 8]$. At the last step, $rlcp[7] = 4 = r$, so we compare strings to find that both strings are equal.

3.1.2 Access operation. The second main operation, *access(i)* , is the opposite of the previous one, retrieving the string for a given identifier. It follows a bottom-up approach, starting at the position

[2]In practice, when $\ell = r$ we have to check the values of $llcp$ and $rlcp$ to choose the branch for traversal. Algorithm 1 shows the actual comparison.

Algorithm 1 Algorithm for *lookup*

```
    function doLookup(s_q, p_ℓ, p_r, ℓ, r)
        p_m ← (p_ℓ + p_r)/2
        if ℓ > r or ((ℓ = r) & llcp[p_m] >= rlcp[p_m]) then
            lval ← llcp[p_m]
 5:         if lval > ℓ then
                return doLookup(s_q, p_m, p_r, ℓ, r)
            else if lval < ℓ then
                return doLookup(s_q, p_ℓ, p_m, ℓ, lval)
            else
10:             (cmp, o) ← compare(s + ℓ, S[p_m])
                if cmp > 0 then
                    return doLookup(s_q, p_m, p_r, o, r)
                else if cmp < 0 then
                    return doLookup(s_q, p_ℓ, p_m, ℓ, o)
15:             else
                    return p_m
                end if
            end if
        else
20:         rval ← rlcp[p_m]
            if rval > r then
                return doLookup(s_q, p_ℓ, p_m, ℓ, r)
            else if lval < ℓ then
                return doLookup(s_q, p_m, p_r, rval, r)
25:         else
                (cmp, o) ← compare(s + r, S[p_m])
                if cmp > 0 then
                    return doLookup(s_q, p_m, p_r, o, r)
                else if cmp < 0 then
30:                 return doLookup(s_q, p_ℓ, p_m, ℓ, o)
                else
                    return m
                end if
            end if
35:     end if
    end function
```

p_i and traversing up to the parent position until we have recovered the full string. The procedure is described in Algorithm 2. The string is decoded from the end, prepending new characters at each new step until we reach the beginning of the string. Given an identifier i, we read $llcp[p_i]$ and $rlcp[p_i]$ and compute their maximum as o. Then, we can extract all the characters from $S[p_i]$, that will correspond to the result string from position o onwards. Since we have already decoded the result from position o, in the next iterations we set a $limit$ to mark that we only need to extract characters up to that position.

After extracting the required characters, we move to the appropriate parent[3], the one corresponding to the maximum lcp, and repeat the procedure. Whenever $o \leq limit$, we prepend the first $limit - o$ characters of the current $S[i]$ to the result. When we reach $o = 0$ the result has been decoded and the procedure ends.

Note that in the worst case we may have to traverse up until we reach one of the positions that are always stored in full: 0, $(n-1)/2$ or $n-1$, hence running $\log(n)$ string comparisons. However, in many instances we can reach $o = 0$ earlier in the traversal. Additionally, in iterations where $o > limit$ comparisons are skipped so we do not even need to access the text. This will be relevant in some implementation variants that apply compression to the string tails, since in those solutions string comparisons are relatively expensive.

Algorithm 2 Algorithm for *access*

```
    function ACCESS(p_i)
        s ← ''
        limit ← max(llcp[p_i], rlcp[p_i]) + len(S[p_i])
        o ← ∞
5:      while limit > 0 do
            if llcp[p_i] ≥ rlcp[p_i] then
                (o, n) ← (llcp[p_i], left(p_i))
            else
                (o, n) ← (rlcp[p_i], right(p_i))
10:         end if
            if o ≤ limit then
                s[o..limit] ← S[p_i][0..limit − o]
                limit ← o − 1
            end if
15:         p_i ← n
        end while
        return s
    end function
```

Following again the example in Figure 1, assume we want to obtain the string for identifier 9 (`clean`). At the first iteration, the maximum common prefix is $rlcp[9] = 4$. This means that $S[4]$ is stored from position 4, so we can recover the characters from position 4 until the end of string (`____n`). We set $limit = 4$ for future iterations, and since the $rlcp$ value was higher we move to the right-side parent, i.e. to position 10. Now $llcp[10] = rlcp[10] = 2$, so $o = 2$ and we can extract the first two characters of $S[10]$ to fill positions 2-3 of the result string, getting `__ean`. In this step we could move to either side, assume we simply move left by convention. We reach position 8, and we get $llcp[8] = rlcp[8] = 0$, so we copy the first two characters of $S[8]$ to fill the remaining positions of our result and then return.

[3]In practice, the parent positions are not computed bottom-up in our implementations. Instead, the list of search positions is obtained in a top-bottom fashion before the *access* algorithm starts. These details are omitted for simplicity in Algorithm 2.

3.2 Implementation variants

Our conceptual representation stores two integer sequences $llcp$ and $rlcp$ and a set of string tails S. Several alternatives exist for the representation of both structures, hence originating a family of structures that provide a space/time tradeoff. In this section we introduce implementation details for the different variants of our proposal:

IBiS is the simplest proposal. In this approach we store $llcp$ and $rlcp$ as sequences of fixed-length integers. This solution is simple and efficient, but in datasets where the maximum lcp value is high it is space-inefficient. The string tails S are concatenated in a single sequence Str. A bitmap B is added to indicate the position in Str where each string begins marking with 1 those positions and setting the remaining positions to 0. We store the bit array using an SDArray compressed bitmap representation, to provide *select* support. In this representation, $S[i]$ is obtained by selecting the position of the i-th 1 in B, and extracting $Str[select_1(B, i)..select_1(B, i+1) - 1]$.

IBiS^{RP} differs from the previous one on the representation of the strings. All the string tails are again concatenated in a single sequence Str, including the end-of-string markers, or string terminators. After this, a variant of Re-Pair compression is applied to the sequence, generating a grammar-compressed sequence where symbols never overlap two dictionary strings. This transforms the original byte string into a grammar and a sequence of integers. The sequence of integers is encoded using Vbyte. We also use a bitmap B that marks with 1 the first byte of each dictionary string. $S[i]$ can be obtained by extracting the sequence in the same way as before, and then decoding the corresponding Re-Pair sequence.

IBiS^{RP+DAC} is similar to IBiS^{RP} but it uses DACs to store the sequences $llcp$ and $rlcp$. This is expected to achieve much better space in many real-world collections, and especially in collections with long strings where the maximum lcp is much higher than the average.

IBiS$^{RP+DAC-VLS}$ is again similar to IBiS^{RP} but uses the variant of DACs designed for variable-length integer sequences (DAC-VLS) to store the Re-Pair-compressed strings, instead of storing them using Vbyte. Since the DAC-VLS structure provides direct access to any string, the bitmap B is not necessary, and a string $S[i]$ is just decoded by extracting symbols from the DAC-VLS structure and decompressing them using the Re-Pair grammar.

IBiS$^{RP+DAC+DAC-VLS}$ just combines the two previous ones: $llcp$ and $rlcp$ are stored using DACs, and Str is stored using DAC-VLS.

3.2.1 End-of-string symbols. All our implementations use a bitmap B or a DAC-VLS structure to provide direct access to any string tail, so, unlike alternatives based on sequential search, our representation does not need to physically store end of string markers. The string terminators are used as markers when applying Re-Pair compression, so that no Re-Pair symbol overlaps two dictionary strings.

However, after compression, we can remove these string terminators to save a byte per string in Str. Nevertheless, we still tested, as well, the version with string terminators since having them we can decode until we reach the terminator instead of performing a second $select_1$ operation on B. Even though *select* operations are constant-time, they are relatively costly and avoiding them we can speed up string decoding.

Notice that, when a string is compressed relative to a larger string in lexicographical order, a zero-length tail may appear (see for example the string at position 5 in Figure 1). We handle these empty strings as a special case, storing them as an end-of-string symbol even if our implementation would remove these symbols in any other case. This is necessary for *select* operations in B to work, so that each $S[i]$ is associated with a different offset in Str; the DAC-VLS structure also requires this adjustment since it is not designed to support zero-length sequences. Note also that the DAC-VLS implementation, due to its construction, would not benefit from extra end-of-string symbols, so for those implementations we only use variants with no string terminators.

3.2.2 Single-lcp implementations. Our main proposal stores two integer sequences, $llcp$ and $rlcp$, to optimize *lookup* operations. Similar algorithms can be designed to work with only one array, saving half the space of these arrays at the cost of worst Front-coding compression.

Single-lcp implementations of any of our proposals can also be built in order to reduce the space utilization. The same idea of the general construction applies to these variants, but now we always compare with the left parent (LLCP-only variants) or with the right parent (RLCP-only variants). Compression of the strings is expected to be worse since we are no longer using the maximum lcp, but these variants can still achieve better overall compression by removing one of the integer sequences.

Regarding query algorithms, *lookup* operations can still save some string comparisons using a similar algorithm to the one we proposed: essentially, LLCP-only variants use lines 4-18 of the original algorithm, and RLCP-only variants lines 20-34. On *access* operations, the algorithm is also essentially the same, but we always move to the left (right) parent. When the lcp arrays are compressed, removing one access to them will have a positive effect on performance, since a single-lcp implementation only needs one DAC access per step.

4 EXPERIMENTAL EVALUATION

In this section we test the performance of our proposal in comparison with several alternatives in the state of the art. We perform tests with real-world datasets, focusing on two main application domains: representation of URLs, obtained from Web graph crawls, and representation of URIs and literal values extracted from RDF datasets. First we show an empirical evaluation of the implementation variants described in Section 3, in order to display their strengths. Then, we perform an experimental evaluation of our best implementation variants, comparing them with existing solutions for string dictionaries. Our comparison focuses on compression capabilities and query performance, and shows that our solutions obtain a better trade-off than state-of-the-art alternatives.

Table 1: Description of the datasets

Dataset	Size(MB)	#strings	Avg. length
UK	1372.06	18,520,486	77.68
Arabic	1774.42	22,744,080	81.81
URIs	1553.46	30,137,450	54.05
Literals	2048.00	331,253,572	7.48

4.1 Experimental setup

We use in our tests a collection of datasets including URLs from real Web graphs and also URIs and literal values from an RDF dataset. Table 1 shows a summary of the datasets used. For each one, we display its size in plain, the number of strings it stores and the average string length (note that the average length displayed is computed as total size divided by number of strings, so it includes an extra character per string corresponding to the string terminator in the input).

UK and *Arabic* are datasets containing URLs of two different Web graph crawls. UK[4] has been obtained from a 2002 crawl of `.uk` domains, whereas Arabic[5] is a 2005 crawl that includes pages from countries whose content is potentially written in Arabic. Both datasets have been obtained from the Webgraph framework [3]. The UK dataset has been used in previous work as a baseline for URL compression [1, 7, 14]. The Arabic dataset is included for better confirmation of the performance of each solution in different Web graphs. Both datasets are similar in number of strings and average string length.

URIs contains all the different URIs in the English version of the DBpedia RDF dataset, in its 3.5.1 version[6].

Literals is a subset of the literals existing in the same DBpedia 3.5.1 dataset. Our input was generated from the original data by extracting all the literal values of the collection and obtaining the raw value from the RDF literal. To do this we remove language tags and type information, as well as the enclosing quotes of the original string. For instance, the RDF literal `"100 AD"@en` becomes `100 AD` after removing the language tag, whereas the numeric value `"57805"^^<http://www.w3.org/2001/XMLSchema#int>` is converted to `57805`. We sorted the values lexicographically, discarding duplicates and taking the entries in the first 2 GB. We limit the input size to 2 GB since it is the maximum supported by most of the state-of-the art alternatives that will be used for comparison. Notice that using raw literal values the strings are significantly shorter, but keeping the full strings would have little effect on our techniques: since only one language tag and a small number of different types are used, Re-Pair compression would be able to efficiently represent the extra characters at small cost. Our choice of raw values aims at highlighting the fundamental differences between Literals and the other datasets used, as Literals has much shorter strings on average, and much more different from each other.

The space shown for each structure is computed precisely from the size of the corresponding components. To measure query times,

[4] http://law..dsi.unimi.it/webdata/uk-2002
[5] http://law.di.unimi.it/webdata/arabic-2005/
[6] http://downloads.dbpedia.org/3.5.1/all_languages.tar

we build a set of 10,000 queries for each dataset by selecting random positions from the collection. The same positions are used for *access* and for the corresponding *lookup* queries. Query times are measured as the average over 100 iterations of the query set.

We implemented all our proposals in C++. We use an implementation of compressed bitmaps and Re-Pair based on the `libcds` library, the same used by Martinez-Prieto et al. [14]. All our implementations are compiled with g++ 4.8 with full optimizations enabled.

We compare our results with the following techniques:

- PFC, RPFC and RPHTFC are some of the differential encoding techniques based on Front-coding [14] described in Section 2. PFC is the plain solution, RPFC uses Re-Pair to compress buckets. RPHTFC is similar to the previous one, but it also applies Hu-Tucker compression to the bucket headers. We include PFC because it is the simplest solution, and RPFC and RPHTFC because they achieved the best results among their Front-coding-based solutions. We used bucket sizes 4, 8, 16 and 32.
- RPDAC and HASHRPDAC are the binary searchable Re-Pair techniques also introduced in Section 2. The first one uses binary search, and the second one adds a hash table to speed up queries. Both of them are used with the default configuration parameters.
- PDT is the the centroid-based compressed implementation of path-decomposed tries variants [7], the best-performing alternative of this family.

All the alternatives are compiled with g++ with full optimizations enabled, using the default settings as provided by the authors apart from the parameters described above.

Note that we do not include a comparison with the implementation of LZ-78-compressed tries also described in Section 2, since their publicly-available code could not be compiled. Nevertheless, previous results have shown that their proposals are dominated in most cases by PDT, and they are also less efficient than HASHRPDAC and RPHTFC.

4.2 Comparison of our variants

Due to the relatively large number of variants that we propose, we first outline some of the general characteristics of our implementation variants to display their relative strengths. After that, in the following sections we will only show experimental results corresponding to those of our techniques that provide the better tradeoff.

Figure 2 shows the space/time tradeoff provided by some of our proposals, considering both uncompressed (IBiS) and Re-Pair-compressed strings (IBiS^{RP}, IBiS^{RP+DAC}). For each approach we show the space/time tradeoff achieved in the dataset UK for the basic implementation (two lcp arrays) and both possible single-lcp implementations, labeled `-L` and `-R` respectively. For each of those, we show results for the basic techniques that keep string terminators and also for *no-term* implementations (labeled with `-nt`). The plot also shows a few of the differential encoding techniques described in Section 2, since they share similarities with our approach: PFC is similar to IBiS, whereas the rest of our variants are similar to RPFC or RPHTFC, improving compression through the use of Re-Pair and other techniques.

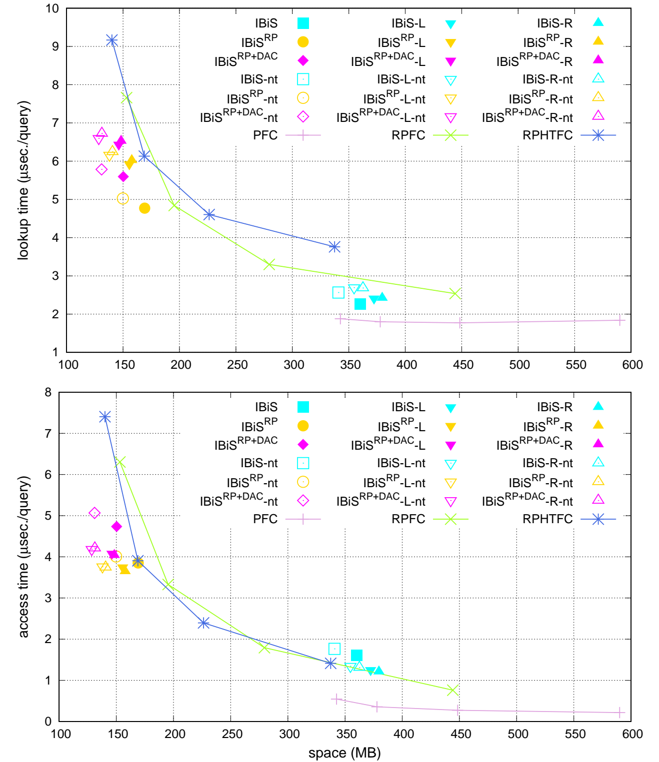

Figure 2: Comparison of implementation variants on dataset UK

As shown in Figure 2, our plain implementations are not very competitive with PFC, since our techniques cannot compress each string with Front-coding as efficiently as the sequential encoding. Plain approaches will be omitted in the next sections, focusing on the more space-efficient alternatives. Figure 2 also shows some trends among our variants that are mostly the same in all the datasets used in our experiments:

- Single-lcp implementations are, in general, a bit more space-efficient than double-lcp implementations in all the variants that use Re-Pair. A single-lcp variant may have significantly more characters in the string tails than a double-lcp variant. However, due to the efficiency of Re-Pair to compress the resulting strings, the actual increase in size of the compressed text is much smaller, and removing one of the lcp arrays easily compensates for this additional space. Plain single-lcp implementations, on the other hand, are much less efficient in space, since the extra bytes in the string tails are not compressed in any way. Regarding query times, single-lcp implementations are slower on *lookup* queries, due to the potentially larger cost of searches, but faster on *access* queries, thanks to the simpler bottom-up traversal that only needs to access a single lcp array. We will show experimental results for both single-lcp and double-lcp variants, since they

can be useful in different scenarios depending on whether *lookup* or *access* queries are more relevant.

- llcp-only and rlcp-only implementations achieve almost identical query times, as expected, but llcp-only achieves better compression in all cases. It is also slightly simpler, since in llcp-only variants we always perform Front-coding compression respective to lexicographically smaller string, so we are guaranteed to have non-empty string tails in every position. In view of these results, we will omit rlcp-only variants from the remaining test results, noting that in all our experiments they were consistently slightly larger than their llcp-only counterparts and query times are similar.
- *no-term* implementations achieve much better compression in most variants and in all datasets. This is expected since after Re-Pair compression is applied to *Str* the average length of a string tail is usually much shorter, so removing a byte per word yields a significant reduction in the overall space. As expected, *no-term* variants are also slightly slower, both in *lookup* and *access* queries, but we consider the effect on compression much more relevant. In the remaining test results we will focus mostly on *no-term* variants.

4.3 Comparison with the state of the art

Next we compare our implementations with the most significant state-of-the-art alternatives to the best of our knowledge. Note that, as stated earlier, we omit some of our implementation alternatives to provide clearer plots, and focus our comparison on the best-performing techniques from previous work.

Figures 3 and 4 show the space/time tradeoff on the Web graph datasets UK and Arabic. Both datasets are similar and the results obtained by the different techniques are also similar. Our proposals achieve the best compression results among all the tested implementations. Particularly, our techniques improve the space-time tradeoff provided by RPFC and RPHTFC. The llcp-only variant of $\text{IBiS}^{RP+DAC+DAC-VLS}$ obtains the best overall space results, but the equivalent IBiS^{RP+DAC} is very close. Regarding query times, the most efficient techniques are HASHRPDAC and PDT; RPDAC is also the fastest technique on *access* queries, but since it is very slow on *lookup* it is much less competitive. Among our variants, query times for *lookup* are very similar, but in *access* queries the DAC-VLS solutions are much slower. The query times of our best solutions are roughly two times slower than the fastest solutions, but we are also significantly smaller than those, becoming the best alternative to optimize compression with competitive query times.

Figure 5 shows the results for the URIs dataset. Overall compression of all the tested representations is slightly worse when compared with the URL collections, but our compressed representations achieve again the best space results, around 15% compression. The best compression is achieved by $\text{IBiS}^{RP+DAC+DAC-VLS}$, particularly its single-lcp variant, but the corresponding IBiS^{RP+DAC} is very close and much more efficient in *access* queries. Like in the previous datasets, most of our proposals improve the tradeoff provided by RPFC and RPHTFC. The exception is $\text{IBiS}^{RP+DAC-VLS}$, that is dominated by RPHTFC on *access* queries. Our best proposals are also significantly smaller than HASHRPDAC, that achieves the best query times. PDT is also very space-efficient, and reaches

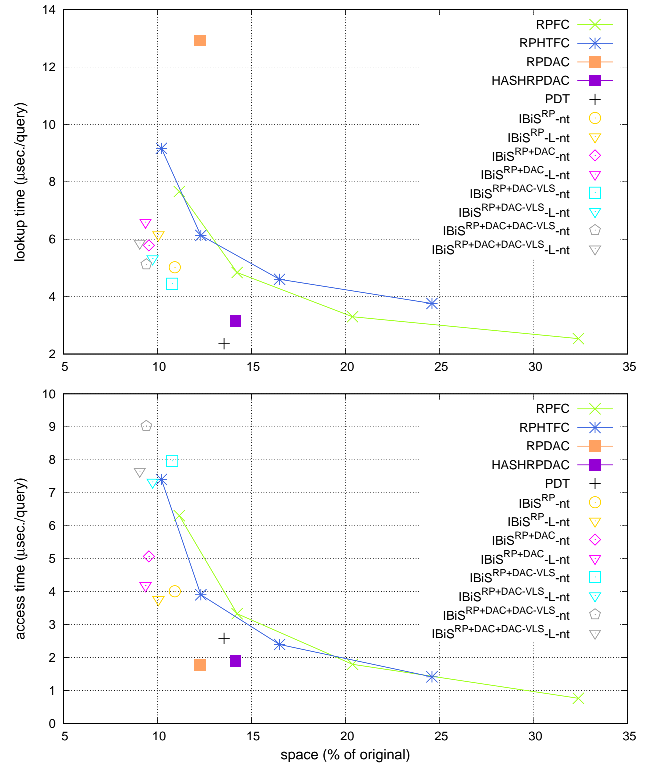

Figure 3: Space and query times on dataset UK

results close to our proposals, while achieving better query times on *lookup* queries. Nevertheless, on *access* queries our best structures are still competitive with PDT, achieving similar query times in less space. We consider this result on *access* queries more relevant than the result on *lookup* queries since, in practice, the former are usually more relevant than the latter, because they are more frequently used. In an RDF engine, for instance, a SPARQL query just requires a few *lookup* operations to encode the URIs/literals used in the query into numeric identifiers; then, after the query is executed, each result has to be translated back into the corresponding URIs/literals, which means a potentially very large number of *access* operations to answer a single query. Hence, even though good performance is required on both operations, performance on *access* queries may be more important in many applications.

Figure 6 shows the results obtained for the Literals dataset. In this dataset, the different nature of the string leads to significantly different results: PDT, RPDAC and HASHRPDAC are much less efficient to compress the collection. Also, among our variants, the DAC-VLS techniques become much less efficient, since they are not well-suited to handle this kind of collection, with very short average string length but a few very long strings. Nevertheless, we still show in the plot the results for the best performing DAC-VLS variants, namely the $\text{IBiS}^{RP+DAC+DAC-VLS}$ approaches. Note also that, in this dataset, llcp-only implementations are not as efficient, and the smallest representation is the double-lcp IBiS^{RP+DAC}. In spite of all these differences, our best solutions (both double-lcp and single-lcp) are much smaller than PDT and HASHRPDAC, while

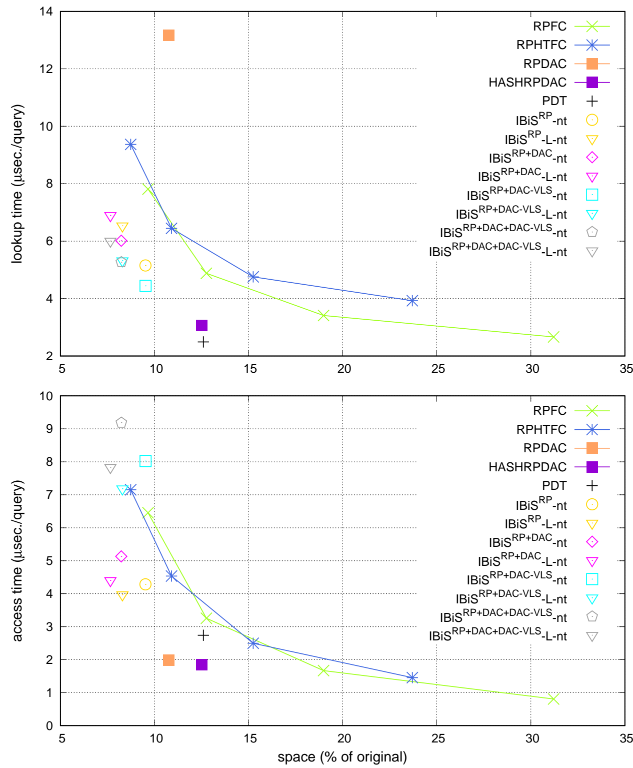

Figure 4: Space and query times on dataset Arabic

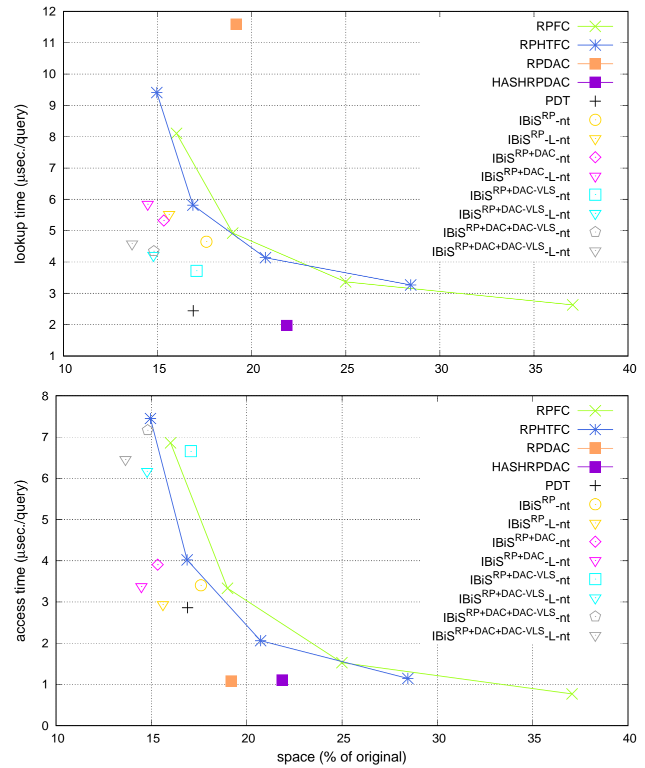

Figure 5: Space and query times on dataset URIs

obtaining query times competitive with them. RPFC and RPHTFC, for larger bucket sizes, can achieve compression similar to us, but at the cost of much larger query times. Notice that, due to the characteristics of this dataset, the overall compression of all the solutions for this collection is much worse than in the previous ones, but still IBiS^{RP+DAC} reaches 25% compression whereas PDT and HASHRPDAC are above 35%.

Taking into account the combined results from Figures 5 and 6, our techniques clearly obtain the best compression results for both URIs and literal values, constituting a very efficient basis for string dictionary compression of RDF data. Our query times are competitive with those of existing data structures, especially on *access* queries, and the space-time tradeoff provided overcomes the tested alternatives.

5 CONCLUSIONS AND FUTURE WORK

We have introduced a new family of compressed data structures for the efficient in-memory representation of string dictionaries. Our solutions can be regarded as an enhanced binary search that combines a hierarchical variant of Front-coding with suffix-array-based techniques to speed up longest-common-prefix computations. Those ideas are then composed with other techniques to derive a family of variants.

We perform a complete experimental evaluation of our proposals, comparing them with the best-performing state-of-the-art solutions and applying them to real-world datasets. We focus on two of the most active application domains for string dictionaries: Web graph and RDF data. Our results show that our representations achieve better compression than existing solutions, for similar query times, and are significantly smaller than any other alternative that is able to outperform our query times. Overall, our representation, in its several implementation variants, provides a relevant improvement in compression relative to previous proposals within very efficient query times.

We plan to explore the possibilities to extend our ideas to the dynamic scenario, where insertions and deletions are supported. A direct application of our techniques is not feasible in a dynamic environment, since we use a static decomposition of the collection and compression techniques that are also of static nature. However, we believe that simple adaptations based on the same compression techniques introduced here would still yield sufficiently compact dynamic dictionaries. Dynamic string dictionaries in compressed space are useful, for instance, for better handling large datasets in RDF engines in main memory.

6 ACKNOWLEDGEMENTS

(Omitted for anonymity purposes)

REFERENCES

[1] Julian Arz and Johannes Fischer. 2018. Lempel—Ziv-78 Compressed String Dictionaries. *Algorithmica* 80, 7 (July 2018), 2012–2047. https://doi.org/10.1007/s00453-017-0348-7
[2] Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. 2017. Deterministic Indexing for Packed Strings. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland.* 6:1–6:11.

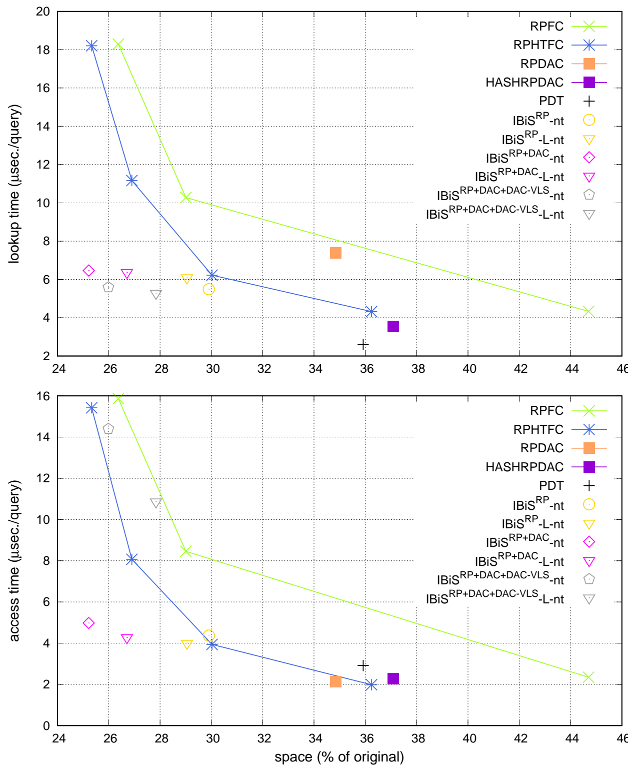

Figure 6: Space and query times on dataset Literals

https://doi.org/10.4230/LIPIcs.CPM.2017.6

[3] Paolo Boldi and Sebastiano Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW ’04)*. ACM, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752

[4] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. 2013. DACs: Bringing Direct Access to Variable-length Codes. *Information Processing and Management* 49, 1 (Jan. 2013), 392–404. https://doi.org/10.1016/j.ipm.2012.08.003

[5] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. 2014. Compact Representation of Web Graphs with Extended Functionality. *Information Systems* 39 (Jan. 2014), 152–174. https://doi.org/10.1016/j.is.2013.08.003

[6] T C. Hu and A C. Tucker. 1979. Optimal Computer Search Trees and Variable-Length Alphabetical Codes. *Siam Journal on Applied Mathematics - SIAMAM* 21 (Jan. 1979). https://doi.org/10.1137/0121057

[7] Roberto Grossi and Giuseppe Ottaviano. 2015. Fast Compressed Tries Through Path Decompositions. *Journal of Experimental Algorithmics* 19, Article 3.4 (Jan. 2015), 11 pages. https://doi.org/10.1145/2656332

[8] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers* 40, 9 (Sept. 1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[9] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. 2017. Compressed double-array tries for string dictionaries supporting fast lookup. *Knowledge and Information Systems* 51, 3 (2017), 1023–1042. https://doi.org/10.1007/s10115-016-0999-8

[10] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. 2017. Practical Implementation of Space-Efficient Dynamic Keyword Dictionaries. In *Proceedings of the 24th International Symposium on String Processing and Information Retrieval (SPIRE ’17)*, Vol. 10508. Springer, 221–233. https://doi.org/10.1007/978-3-319-67428-5_19

[11] Donald E. Knuth. 1998. *The Art of Computer Programming, volume 3: Sorting and searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[12] N. Jesper Larsson and Alistair Moffat. 1999. Offline Dictionary-Based Compression. In *Proceedings of the Conference on Data Compression (DCC ’99)*. IEEE Computer Society, Washington, DC, USA, 296–. https://doi.org/10.1109/DCC.1999.755679

[13] Udi Manber and Eugene W. Myers. 1993. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.* 22 (Jan. 1993), 935–948. https://doi.org/10.1145/320176.320218

[14] Miguel A. Martínez-Prieto, Nieves R. Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. 2016. Practical Compressed String Dictionaries. *Information Systems* 56, C (Mar. 2016), 73–108. https://doi.org/10.1016/j.is.2015.08.008

[15] Miguel A. Martínez-Prieto, Javier D. Fernández, and Rodrigo Cánovas. 2012. Querying RDF Dictionaries in Compressed Space. *ACM SIGAPP Applied Computing Review* 12, 2 (June 2012), 64–77. https://doi.org/10.1145/2340416.2340422

[16] Gonzalo Navarro. 2016. *Compact Data Structures: A Practical Approach* (1st ed.). Cambridge University Press, New York, NY, USA. https://doi.org/10.1017/CBO9781316588284

[17] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal* 19, 1 (Feb. 2010), 91–113. https://doi.org/10.1007/s00778-009-0165-y

[18] Daisuke Okanohara and Kunihiko Sadakane. 2007. Practical Entropy-compressed Rank/Select Dictionary. In *Proceedings of the Meeting on Algorithm Engineering & Experiments*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 60–70. https://doi.org/10.1137/1.9781611972870.6

[19] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. 2002. Succinct Indexable Dictionaries with Applications to Encoding K-ary Trees and Multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA ’02)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 233–242. https://doi.org/10.1145/1290672.1290680

[20] Kazuya Tsuruta, Dominik Köppl, Shunsuke Kanda, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. 2019. Dynamic Packed Compact Tries Revisited. *CoRR* abs/1904.07467 (2019). arXiv:1904.07467

[21] Hugh E. Williams and Justin Zobel. 1999. Compressing Integers for Fast File Access. *Comput. J.* 42, 3 (Jan. 1999), 193–201. https://doi.org/10.1093/comjnl/42.3.193