

# Fast Prefix Search in Little Space, with Applications

Djamal Belazzougui<sup>1</sup>, Paolo Boldi<sup>2</sup>, Rasmus Pagh<sup>3</sup>, and Sebastiano Vigna<sup>1</sup>

<sup>1</sup> Université Paris Diderot—Paris 7, France

<sup>2</sup> Dipartimento di Scienze dell’Informazione, Università degli Studi di Milano, Italy

<sup>3</sup> IT University of Copenhagen, Denmark

**Abstract.** It has been shown in the indexing literature that there is an essential difference between prefix/range searches on the one hand, and predecessor/rank searches on the other hand, in that the former provably allows faster query resolution. Traditionally, prefix search is solved by data structures that are also dictionaries—they actually contain the strings in  $S$ . For very large collections stored in slow-access memory, we propose much more compact data structures that support *weak* prefix searches—they return the ranks of matching strings provided that *some* string in  $S$  starts with the given prefix. In fact, we show that our most space-efficient data structure is asymptotically space-optimal.

Previously, data structures such as String B-trees (and more complicated cache-oblivious string data structures) have implicitly supported weak prefix queries, but they all have query time that grows logarithmically with the size of the string collection. In contrast, our data structures are simple, naturally cache-efficient, and have query time that depends only on the length of the prefix, all the way down to constant query time for strings that fit in one machine word.

We give several applications of weak prefix searches, including exact prefix counting and approximate counting of tuples matching conjunctive prefix conditions.

## 1 Introduction

In this paper we are interested in the following problem (hereafter referred to as *prefix search*): given a collection of strings, find all the strings that start with a given prefix. In particular, we will be interested in the space/time tradeoffs needed to do prefix search in a static context (i.e., when the collection does not change over time).

There is a large literature on indexing of string collections. We refer to Ferragina et al. [14, 4] for state-of-the-art results, with emphasis on the cache-oblivious model. Roughly speaking, results can be divided into two categories based on the power of queries allowed. As shown by Pătraşcu and Thorup [19] any data structure for bit strings that supports predecessor (or rank) queries must either use super-linear space, or use time  $\Omega(\log |p|)$  for a query on a prefix  $p$ . On the other hand, it is known that prefix queries, and more generally range queries, can be answered in constant time using linear space [1].

Another distinction is between data structures where the query time grows with the number of strings in the collection (typically comparison-based), versus those where the query time depends only on the length of the query string (typically some kind of trie)<sup>4</sup>. In this paper we fill a gap in the literature by considering data structures for *weak prefix search*, a relaxation of prefix search, with query time depending only on the length of the query string. In a weak prefix search we have the guarantee that the input  $p$  is a prefix of some string in the set, and we are only requested to output the ranks (in lexicographic order) of the strings that have  $p$  as prefix. Weak prefix searches have previously been implicitly supported by a number of string indexes, most notably the String B-tree [13] and its descendants. In the paper we also present a number of new applications, outlined at the end of the introduction.

Our first result is that weak prefix search can be performed by accessing a data structure that uses just  $O(n \log \ell)$  bits, where  $\ell$  is the average string length. This is much less than the space of  $n\ell$  bits used for the strings themselves. We also show that this is the minimum possible space usage for any such data structure, regardless of query time. We investigate different time/space tradeoffs: At one end of this spectrum we have constant-time queries (for prefixes that fit in  $O(1)$  words), and still asymptotically vanishing space usage for the index. At the other end, space is optimal and the query time grows logarithmically with the length of the prefix. Precise statements can be found in the technical overview below.

*Motivation for smaller indexes.* Traditionally, algorithmic complexity is studied in the so-called RAM model. However, in recent years a discrepancy has been observed between that model and the reality of modern computing hardware. In particular, the RAM model assumes that the cost of memory access is uniform; however, current architectures, including distributed ones, have strongly non-uniform access cost, and this trend seems to go on, see e.g. [17] for recent work in this direction. Modern computer memory is composed of hierarchies where each level in the hierarchy is both faster and smaller than the subsequent level. As a consequence, we expect that reducing the size of the data structure will yield faster query resolution. Our aim in reducing the space occupied by the data structure is to improve the chance that the data structure will fit in the faster levels of the hierarchy. This could have a significant impact on performance, e.g. in cases where the plain storage of the strings does not fit in main memory. For databases containing very long keys this is likely to happen (e.g., static repositories of URLs, that are of utmost importance in the design of search engines, can contain strings as long as one kilobyte). In such cases, reduction of space usage from  $O(n\ell)$  to  $O(n \log \ell)$  bits can be significant.

By studying the weak version of prefix search, we are able to separate clearly the space used by the original data, and the space that is necessary to store an index. Gál and Miltersen [15] classify structures as *systematic* and *non-systematic* depending on whether the original data is stored verbatim or not. Our indices provide a result *without* using the original data, and in this sense our structures

---

<sup>4</sup> Obviously, one can also combine the two in a single data structure.

for weak prefix search are non-systematic. Observe, however, that since those structures gives no guarantee on the result for strings that are not prefixes of some string of the set, standard information-theoretical lower bounds (based on the possibility of reconstructing the original set of strings from the data structure) do not apply.

*Technical overview.* For simplicity we consider strings over a binary alphabet, but our methods generalise to larger alphabets (the interested reader can refer to appendix H for discussion on this point). Our main result is that weak prefix search needs just  $O(|p|/w + \log |p|)$  time and  $O(n \log \ell)$  space in addition to the original data, where  $\ell$  is the average length of the strings,  $p$  is the query string, and  $w$  is the machine word size. For strings of fixed length  $w$ , this reduces to query time  $O(\log w)$  and space  $O(n \log w)$ , and we show that the latter is *optimal* regardless of query time. Throughout the paper we strive to state all space results in terms of  $\ell$ , and time results in terms of the length of the actual query string  $p$ , as in a realistic setting (e.g., term dictionaries of a search engine) string lengths might vary wildly, and queries might be issued that are significantly shorter than the average (let alone maximum) string length. Actually, the data structure size depends on the *hollow trie size* of the set  $S$ —a data-aware measure related to the trie size [16] that is much more precise than the bound  $O(n \log \ell)$ .

Building on ideas from [1], we then give an  $O(1 + |p|/w)$  solution (i.e., constant time for prefixes of length  $O(w)$ ) that uses space  $O(n \ell^{1/c} \log \ell)$ . This structure shows that weak prefix search is possible in constant time using sublinear space. This data structure uses  $O(1 + |p|/B)$  I/Os in the cache-oblivious model.

*Comparison to related results.* If we study the same problem in the I/O model or in the cache-oblivious model, the nearest competitor is the String B-tree [13], and its cache-oblivious version [4]. In the *static* case, the String B-tree can be tuned to use  $O(n \log \ell)$  bits by carefully encoding the string pointers, and it has very good search performance with  $O(\log_B(n) + |p|/B)$  I/Os per query (supporting all query types discussed in this paper). However, a search for  $p$  inside the String B-tree may involve  $\Omega(|p|)$  RAM operations, so it may be too expensive for intensive computations<sup>5</sup>. Our first method, which also achieves the smallest possible space usage of  $O(n \log \ell)$  bits, uses  $O(|p|/w + \log |p|)$  RAM operations and  $O(|p|/B + \log |p|)$  I/Os instead. The number of RAM operations is a strict improvement over String B-trees, while the I/O bound is better for large enough sets. Our second method uses slightly more space ( $O(n \ell^{1/c} \log \ell)$  bits) but features  $O(|p|/w)$  RAM operations and  $O(|p|/B)$  I/Os.

In [14], the authors discuss very succinct static data structures for the same purposes (on a generic alphabet), decreasing the space to a lower bound that is, in the binary case, the trie size. The search time is logarithmic in the number of strings. As in the previous case, we improve on RAM operations and on I/Os for large enough sets.

<sup>5</sup> Actually, the string B-tree can be tuned to work in  $O(|P|/w + \log n)$  time in the RAM model, but this would imply a  $O(|P|/B + \log n)$  I/O cost instead of  $O(|P|/B + \log_B n)$ .

The first cache-oblivious dictionary supporting prefix search was devised by Brodal *et al.* [5] achieving  $O(|p|)$  RAM operations and  $O(|p|/B) + \log_B(n)$  I/Os. We note that the result in [5] is optimal in a comparison-based model, where we have a lower bound of  $\log_B(n)$  I/Os per query. By contrast, our result, like those in [4, 14], assumes an integer alphabet where we do not have such a lower bound.

Implicit in the paper of Alstrup *et al.* [1] on range queries is a linear-space structure for constant-time weak prefix search on fixed-length bit strings. Our constant-time data structure, instead, uses sublinear space and allows for variable-length strings.

*Applications.* Data structures that allow weak prefix search can be used to solve the non-weak version of the problem, provided that the original data is stored (typically, in some slow-access memory): a single probe is sufficient to determine if the result set is empty; if not, access to the string set is needed just to retrieve the strings that match the query. We also show how to solve range queries with two additional probes to the original data (wrt. the output size), improving the results in [1]. We also present other applications of our data structures to other important problems, viz., prefix counting. We finally show that our results extend to the cache-oblivious model, where we provide an alternative to the results in [5, 4, 14] that removes the dependence on the data set size for prefix searches and range queries.

*Our contributions.* The main contribution of this paper is the identification of the weak prefix search problem, and the proposal of an optimal solution based on techniques developed in [2]. Optimality (in space or time) of the solution is also a central result of this research. The second interesting contribution is the description of *range locators* for variable-length strings; they are an essential building block in our weak prefix search algorithms, and can be used whenever it is necessary to recover in little space the range of leaves under a node of a trie.

## 2 Notation and tools

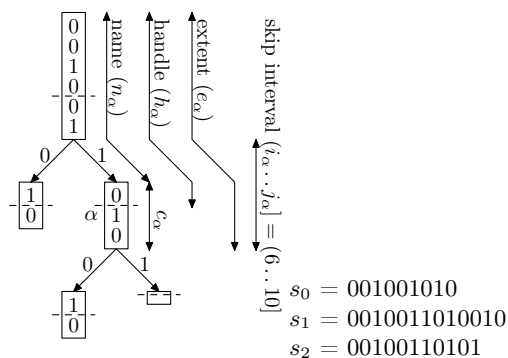
In the following sections, we will use the toy set of strings shown in Figure 1 to display examples of our constructions. In this section, we introduce some terminology and notation adopted throughout the rest of the paper. We use von Neumann’s definition and notation for natural numbers:  $n = \{0, 1, \dots, n - 1\}$ , so  $2 = \{0, 1\}$  and  $2^*$  is the set of all binary strings.

**Weak prefix search.** Given a prefix-free set of strings  $S \subseteq 2^*$ , the *weak prefix search* problem requires, given a prefix  $p$  of some string in  $S$ , to return the range of strings of  $S$  having  $p$  as prefix; this set is returned as the interval of integers that are the ranks (in lexicographic order) of the strings in  $S$  having  $p$  as prefix.

**Model and assumptions.** Our model of computation is a unit-cost word RAM with word size  $w$ . We assume that  $|S| = O(2^{cw})$  for some constant  $c$ , so that constant-time static data structures depending on  $|S|$  can be used.

We also consider bounds in the cache-oblivious model. In this model, we consider that the machine has a two levels memory hierarchy, where the fast level has an unknown size of  $M$  bits (which is actually not used in this paper) and a slower level of unspecified size where our data structures resides. We assume that the slow level plays a role of cache for the fast level with an optimal replacement strategy where the transfers between the two levels is done in blocks of an unknown size of  $B$  bits, with  $B \leq M$ . The cost of an algorithm is the total number of block transfers between the two levels.

**Compacted tries.** Consider the compacted trie built for a prefix-free set of strings  $S \subseteq 2^*$ . For a given node  $\alpha$  of the trie, we define (see Figure 1):



**Fig. 1.** The compacted trie for the set  $S = \{s_0, s_1, s_2\}$ , and the related names.

- $e_\alpha$ , the *extent of node*  $\alpha$ , is the longest common prefix of the strings represented by the leaves that are descendants of  $\alpha$  (this was called the “string represented by  $\alpha$ ” in [2]);
- $c_\alpha$ , the *compacted path of node*  $\alpha$ , is the string stored at  $\alpha$ ;
- $n_\alpha$ , the *name of node*  $\alpha$ , is the string  $e_\alpha$  deprived of its suffix  $c_\alpha$  (this was called the “path leading to  $\alpha$ ” in [2]);
- given a string  $x$ , we let  $\text{exit}(x)$  be the exit node of  $x$ , that is, the only node  $\alpha$  such that  $n_\alpha$  is a prefix of  $x$  and either  $e_\alpha = x$  or  $e_\alpha$  is not a prefix of  $x$ ;
- the *skip interval*  $[i_\alpha \dots j_\alpha]$  associated to  $\alpha$  is  $[0 \dots |c_\alpha|)$  for the root, and  $[|n_\alpha| - 1 \dots |e_\alpha|)$  for all other nodes.

We note the following property, proved in Appendix B:

**Theorem 1.** *The average length of the extents of internal nodes is at most the average string length minus one.*

<table style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px 10px 2px 10px;">0</td><td style="padding: 2px 10px 2px 10px;">→ ∞</td></tr> <tr><td style="padding: 2px 10px 2px 10px;">00</td><td style="padding: 2px 10px 2px 10px;">→ ∞</td></tr> <tr><td style="padding: 2px 10px 2px 10px;">0010</td><td style="padding: 2px 10px 2px 10px;">→ 001001 (6)</td></tr> <tr><td style="padding: 2px 10px 2px 10px;">0010010</td><td style="padding: 2px 10px 2px 10px;">→ ∞</td></tr> <tr><td style="padding: 2px 10px 2px 10px;">00100101</td><td style="padding: 2px 10px 2px 10px;">→ 001001010 (9)</td></tr> <tr><td style="padding: 2px 10px 2px 10px;">0010011</td><td style="padding: 2px 10px 2px 10px;">→ ∞</td></tr> <tr><td style="padding: 2px 10px 2px 10px;">00100110</td><td style="padding: 2px 10px 2px 10px;">→ 0010011010 (10)</td></tr> <tr><td style="padding: 2px 10px 2px 10px;">00100110100</td><td style="padding: 2px 10px 2px 10px;">→ ∞</td></tr> <tr><td style="padding: 2px 10px 2px 10px;">001001101001</td><td style="padding: 2px 10px 2px 10px;">→ 0010011010010 (13)</td></tr> <tr><td style="padding: 2px 10px 2px 10px;">00100110101</td><td style="padding: 2px 10px 2px 10px;">→ 00100110101 (11)</td></tr> </table>	0	→ ∞	00	→ ∞	0010	→ 001001 (6)	0010010	→ ∞	00100101	→ 001001010 (9)	0010011	→ ∞	00100110	→ 0010011010 (10)	00100110100	→ ∞	001001101001	→ 0010011010010 (13)	00100110101	→ 00100110101 (11)	<table style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th style="border-bottom: 1px solid black; padding: 2px 10px 2px 10px;"><i>P</i></th> <th style="border-bottom: 1px solid black; padding: 2px 10px 2px 10px;"><i>b</i></th> </tr> </thead> <tbody> <tr><td style="padding: 2px 10px 2px 10px;"><b>0010010</b></td><td style="padding: 2px 10px 2px 10px;">1</td></tr> <tr><td style="padding: 2px 10px 2px 10px;"><u>0010011</u></td><td style="padding: 2px 10px 2px 10px;">0</td></tr> <tr><td style="padding: 2px 10px 2px 10px;"><u>00100110100</u></td><td style="padding: 2px 10px 2px 10px;">1</td></tr> <tr><td style="padding: 2px 10px 2px 10px;"><u>00100110101</u></td><td style="padding: 2px 10px 2px 10px;">1</td></tr> <tr><td style="padding: 2px 10px 2px 10px;"><u>00100110110</u></td><td style="padding: 2px 10px 2px 10px;">0</td></tr> <tr><td style="padding: 2px 10px 2px 10px;"><u>0010100</u></td><td style="padding: 2px 10px 2px 10px;">0</td></tr> </tbody> </table>	<i>P</i>	<i>b</i>	<b>0010010</b>	1	<u>0010011</u>	0	<u>00100110100</u>	1	<u>00100110101</u>	1	<u>00100110110</u>	0	<u>0010100</u>	0
0	→ ∞																																		
00	→ ∞																																		
0010	→ 001001 (6)																																		
0010010	→ ∞																																		
00100101	→ 001001010 (9)																																		
0010011	→ ∞																																		
00100110	→ 0010011010 (10)																																		
00100110100	→ ∞																																		
001001101001	→ 0010011010010 (13)																																		
00100110101	→ 00100110101 (11)																																		
<i>P</i>	<i>b</i>																																		
<b>0010010</b>	1																																		
<u>0010011</u>	0																																		
<u>00100110100</u>	1																																		
<u>00100110101</u>	1																																		
<u>00100110110</u>	0																																		
<u>0010100</u>	0																																		

**Fig. 2.** The data making up a z-fast prefix trie based on the trie above, and the associated range locator.  $T$  maps handles to extents; the corresponding *hollow* z-fast prefix trie just returns the *lengths* (shown in parentheses) of the extents. In the range locator table, we boldface the zeroes and ones appended to extents, and we underline the actual keys (as trailing zeroes are removed). The last two keys are  $00100110101^+$  and  $0010011^+$ , respectively.

**Data-aware measures.** Consider the compacted trie on a set  $S \subseteq 2^*$ . We define the *trie measure* of  $S$  [16] as

$$T(S) = \sum_{\alpha} (j_{\alpha} - i_{\alpha}) = \sum_{\alpha} (|c_{\alpha}| + 1) - 1 = 2n - 2 + \sum_{\alpha} |c_{\alpha}| = O(n\ell),$$

where the summation ranges over all nodes of the trie. For the purpose of this paper, we will also use the *hollow trie measure*

$$HT(S) = \sum_{\alpha \text{ internal}} (\text{bitlength}(|c_{\alpha}|) + 1) - 1.$$

Since  $\text{bitlength}(x) = \lceil \log(x+1) \rceil$ , we have  $HT(S) = n - 2 + \sum_{\alpha \text{ internal}} \lceil \log(|c_{\alpha}| + 1) \rceil = O(n \log \ell)$ .<sup>6</sup>

**Storing functions.** The problem of storing statically an  $r$ -bit function  $f : A \rightarrow 2^r$  from a given set of keys  $A$  has recently received renewed attention [10, 7, 20]. For the purposes of this paper, we simply recall that these methods allow us to store an  $r$ -bit function on  $n$  keys using  $rn + cn + o(n)$  bits for some constant  $c \geq 0$ , with  $O(|x|/w)$  access time for a query string  $x$ . Practical implementations are described in [3]. In some cases, we will store a *compressed* function using a minimal perfect function ( $O(n)$  bits) followed by a compressed data representation (e.g., an Elias–Fano compressed list [3]). In that case, storing natural numbers  $x_0, x_1, \dots, x_{n-1}$  requires space  $\sum_i \lceil \log(x_i + 1) \rceil + n \log(\sum_i \lceil \log(x_i + 1) \rceil / n) + O(n)$ .

**Relative dictionaries.** A *relative dictionary* stores a set  $E$  relatively to some set  $S \supseteq E$ . That is, the relative dictionary answers questions about membership

<sup>6</sup> A compacted trie is made *hollow* by replacing the compacted path at each node by its length and then discarding all its leaves. A recursive definition of hollow trie appears in [3].

to  $E$ , but its answers are required to be correct only if the query string is in  $S$ . It is possible to store such a dictionary in  $|E| \log(|S|/|E|)$  bits of space with  $O(|x|/w)$  access time [2].

**Rank and select.** We will use two basic blocks of several succinct data structures—rank and select. Given a bit array (or bit string)  $\mathbf{b} \in 2^n$ , whose positions are numbered starting from 0,  $\text{rank}_{\mathbf{b}}(p)$  is the number of ones up to position  $p$ , exclusive ( $0 \leq p \leq n$ ), whereas  $\text{select}_{\mathbf{b}}(r)$  is the position of the  $r$ -th one in  $\mathbf{b}$ , with bits numbered starting from 0 ( $0 \leq r < \text{rank}_{\mathbf{b}}(n)$ ). It is well known that these operations can be performed in constant time on a string of  $n$  bits using additional  $o(n)$  bits, see [18, 8, 6, 21].

### 3 From prefixes to exit nodes

We break the weak prefix search problem into two subproblems. Our first goal is to go from a given a prefix of some string in  $S$  to its exit node.

#### 3.1 Hollow z-fast prefix tries

We start by describing an improvement of the *z-fast trie*, a data structure first defined in [2]. The main idea behind a z-fast trie is that, instead of representing explicitly a binary tree structure containing compacted paths of the trie, we will store a function that maps a certain prefix of each extent to the extent itself. This mapping (which can be stored in linear space) will be sufficient to navigate the trie and obtain, given a string  $x$ , the *name of the exit node of  $x$*  and the exit behaviour (left, right, or possibly equality for leaves). The interesting point about the z-fast trie is that it provides such a name in time  $O(|x|/w + \log |x|)$ , and that it leads easily to a probabilistically relaxed version, or even to blind/hollow variants.

To make the paper self-contained, we recall the main definitions from [2]. The *2-fattest* number in a nonempty interval of positive integers is the number in the interval whose binary representation has the largest number of trailing zeros. Consider the compacted trie on  $S$ , one of its nodes  $\alpha$ , its skip interval  $[i_\alpha \dots j_\alpha)$ , and the 2-fattest number  $f$  in  $(i_\alpha \dots j_\alpha]$  (note the change); if the interval is empty, which can happen only at the root, we set  $f = 0$ . The *handle*  $h_\alpha$  of  $\alpha$  is  $e_\alpha[0 \dots f)$ , where  $e_\alpha[0 \dots f)$  denotes the first  $f$  bits of  $e_\alpha$ . A (*deterministic*) *z-fast trie* is a dictionary  $T$  mapping each handle  $h_\alpha$  to the corresponding extent  $e_\alpha$ . In Figure 2, the part of the mapping  $T$  with non- $\infty$  output is the z-fast trie built on the trie of Figure 1.

We now introduce a more powerful structure, the (*deterministic*) *z-fast prefix trie*. Consider again a node  $\alpha$  of the compacted trie on  $S$  with notation as above. The *pseudohandles* of  $\alpha$  are the strings  $e_\alpha[0 \dots f')$ , where  $f'$  ranges among the 2-fattest numbers of the intervals  $(i_\alpha \dots t]$ , with  $i_\alpha < t < f$ . Essentially, pseudohandles play the same rôle as handles for every *prefix* of the handle that extends the node name. We note immediately that there are at most  $\log(f - i_\alpha) \leq \log |c_\alpha|$  pseudohandles associated with  $\alpha$ , so the overall number of handles and

pseudohandles is bounded by  $\text{HT}(S) + \sum_{x \in S} \log |x| = O(n \log \ell)$ . It is now easy to define a z-fast prefix trie: the dictionary providing the map from handles to extents is enlarged to pseudohandles, which are mapped to the special value  $\infty$ .

We are actually interested in a *hollow* version of a z-fast prefix trie—more precisely, a version implemented by a function  $T$  that maps handles of internal nodes to the length of their extents, and handles of leaves and pseudohandles to  $\infty$ . The function (see again Figure 2) can be stored in a very small amount of space; nonetheless, we will still be able to compute the name of the exit node of any string that is a prefix of some string in  $S$  using Algorithm 2, whose correctness is proved in Appendix D.

**Algorithm 1**

**Input:** a prefix  $p$  of some string in  $S$ .  
 $i \leftarrow \lfloor \log |p| \rfloor$   
 $\ell, r \leftarrow 0, |p|$   
**while**  $r - \ell > 1$  **do**  
  **if**  $\exists b$  such that  $2^i b \in (\ell..r)$  **then**  
    //  $2^i b$  is 2-fattest number in  $(\ell..r)$   
     $g \leftarrow T(p[0..2^i b])$   
    **if**  $g \geq |p|$  **then**  
       $r \leftarrow 2^i b$   
    **else**  
       $\ell \leftarrow g$   
    **end if**  
  **end if**  
   $i \leftarrow i - 1$   
**end while**  
**if**  $\ell = 0$  **then**  
  **return**  $\varepsilon$   
**else**  
  **return**  $p[0..\ell + 1]$   
**end if**

**Fig. 3.** Given a nonempty string  $p$  that is the prefix of at least one string in the set  $S$ , this algorithm returns the name of  $\text{exit}(p)$ .

**3.2 Space and time**

The space needed for a hollow z-fast prefix trie depends on the component chosen for its implementation. The most trivial bound uses a function mapping handles and pseudohandles to one bit that makes it possible to recognise handles of

**Algorithm 2**

**Input:** the name  $x$  of a node  
**if**  $x = \varepsilon$  **then**  
   $i \leftarrow 0, j \leftarrow n$   
**else**  
   $i \leftarrow \text{rank}_b h(x^{\leftarrow})$   
  **if**  $x = 111 \dots 11$  **then**  
     $j \leftarrow n$   
  **else**  
     $j \leftarrow \text{rank}_b h((x^+)^{\leftarrow})$   
  **end if**  
**end if**  
**return**  $[i..j]$

**Fig. 4.** Given the name  $x$  of a node in a trie containing  $n$  strings, compute the interval  $[i..j]$  containing precisely all the (ranks of the) strings prefixed by  $x$  (i.e., the strings in the subtree whose name is  $x$ ).

internal nodes ( $O(n \log \ell)$  bits), and a function mapping handles to extent lengths ( $O(n \log L)$  bits, where  $L$  is the maximum string length).

These results, however, can be significantly improved. First of all, we can store handles of internal nodes in a relative dictionary. The dictionary will store  $n-1$  strings out of  $O(n \log \ell)$  strings, using  $O(n \log((n \log \ell)/n)) = O(n \log \log \ell)$  bits. Then, the mapping from handles to extent lengths  $h_\alpha \mapsto |e_\alpha|$  can actually be recast into a mapping  $h_\alpha \mapsto |e_\alpha| - |h_\alpha|$ . But since  $|e_\alpha| - |h_\alpha| \leq |c_\alpha|$ , by storing this data using a compressed function we will use space

$$\begin{aligned} & \sum_{\alpha} \lceil \log(|e_\alpha| - |h_\alpha| + 1) \rceil + O(n \log \log \ell) + O(n) \\ & \leq \sum_{\alpha} \lceil \log(|c_\alpha| + 1) \rceil + O(n \log \log \ell) \leq \text{HT}(S) + O(n \log \log \ell), \end{aligned}$$

where  $\alpha$  ranges over internal nodes.

Algorithm 1 cannot iterate more than  $\log |p|$  times; at each step, we query constant-time data structures using a prefix of  $p$ : using incremental hashing [9, Section 5], we can preprocess  $p$  in time  $O(|p|/w)$  (and in  $|p|/B$  I/Os) so that hashing prefixes of  $p$  requires constant time afterwards. We conclude that Algorithm 1 requires time  $O(|p|/w + \log |p|)$ .

### 3.3 Faster, faster, faster...

We now describe a data structure mapping prefixes to exit nodes inspired by the techniques used in [1] that needs  $O(n\ell^{1/2} \log \ell)$  bits of space and answers in time  $O(|p|/w)$ , thus providing a different space/time tradeoff. The basic idea is as follows: let  $s = \lceil \ell^{1/2} \rceil$  and, for each node  $\alpha$  of the compacted trie on the set  $S$ , consider the set of prefixes of  $e_\alpha$  with length  $t \in (i_\alpha \dots j_\alpha]$  such that either  $t$  is a multiple of  $s$  or is smaller than the first such multiple. More precisely, we consider prefixes whose length is either of the form  $ks$ , where  $ks \in (i_\alpha \dots j_\alpha]$ , or in  $(i_\alpha \dots \min\{\bar{k}s, j_\alpha\}]$ , where  $\bar{k}$  is the minimum  $k$  such that  $ks > i_\alpha$ .

We store a function  $F$  mapping each prefix  $p$  defined above to the length of the name of the corresponding node  $\alpha$  (actually, we can map  $p$  to  $|p| - |n_\alpha|$ ). Additionally, we store a mapping  $G$  from each node name to the length of its extent (again, we can just map  $n_\alpha \mapsto |c_\alpha|$ ).

To retrieve the exit node of a string  $p$  that is a prefix of some string in  $S$ , we consider the string  $q = p[0 \dots |p| - |p| \bmod s)$  (i.e., the longest prefix of  $p$  whose length is a multiple of  $s$ ). Then, we check whether  $G(p[0 \dots F(q)]) \geq |p|$  (i.e., whether  $p$  is a prefix of the extent of the exit node of  $q$ ). If this is the case, then clearly  $p$  has the same exit node as  $q$  (i.e.,  $p[0 \dots F(q)]$ ). Otherwise, the map  $F$  provides directly the length of the name of the exit node of  $p$ , which is thus  $p[0 \dots F(p)]$ . All operations are completed in time  $O(|p|/w)$ .

The proof that this structure uses space  $O(n\ell^{1/2} \log \ell)$  is deferred to Appendix C.

## 4 Range location

Our next problem is determining the range (of lexicographical ranks) of the leaves that appear under a certain node of a trie. Actually, this problem is pretty common in static data structures, and usually it is solved by associating with each node a pair of integers of  $\log n \leq w$  bits. However, this means that the structure has, in the worst case, a linear ( $O(nw)$ ) dependency on the data.

To work around this issue, we propose to use a *range locator*—an abstraction of a component used in [2]. Here we redefine range locators from scratch, and improve their space usage so that it is dependent on the average string length, rather than on the maximum string length.

A range locator takes as input *the name of a node*, and returns the range of ranks of the leaves that appear under that node. For instance, in our toy example the answer to 0010011 would be [1..3). To build a range locator, we need to introduce *monotone minimal perfect hashing*.

Given a set of  $n$  strings  $T$ , a *monotone minimal perfect hash function* [2] is a bijection  $T \rightarrow n$  that preserves lexicographical ordering. This means that each string of  $T$  is mapped to its rank in  $T$  (but strings not in  $T$  give random results). We use the following results from [3]:<sup>7</sup>

**Theorem 2.** *Let  $T$  be a set of  $n$  strings of average length  $\ell$  and maximum length  $L$ , and  $x \in 2^*$  be a string. Then, there are monotone minimal perfect hashing functions on  $T$  that:*

1. *use space  $O(n \log \ell)$  and answer in time  $O(|x|/w)$ ;*
2. *use space  $O(n \log \log L)$  and answer in time  $O(|x|/w + \log |x|)$ .*

We show how a reduction can relieve us from the dependency on  $L$ ; this is essential to our goals, as we want to depend just on the average length:

**Theorem 3.** *There is a monotone minimal perfect hashing function on  $T$  using space  $O(n \log \log \ell)$  that answers in time  $O(|x|/w + \log |x|)$  on a query string  $x \in 2^*$ .*

*Proof.* We divide  $T$  into the set of strings  $T^-$  shorter than  $\ell \log n$ , and the remaining “long” strings  $T^+$ . Setting up a  $n$ -bit vector  $\mathbf{b}$  that records the elements of  $T^-$  with select-one and select-zero structures ( $n + o(n)$  bits), we can reduce the problem to hashing monotonically  $T^-$  and  $T^+$ . We note, however, that using Theorem 2  $T^-$  can be hashed in space  $O(|T^-| \log \log(\ell \log n)) = O(|T^-| \log \log \ell)$ , as  $2\ell \geq \log n$ , and  $T^+$  can be hashed explicitly using a  $(\log n)$ -bit function; since  $|T^+| \leq n/\log n$  necessarily, the function requires  $O(n)$  bits. Overall, we obtain the required bounds.

We now describe in detail our range locator, using the notation of Section 2. Given a string  $x$ , let  $x^\leftarrow$  be  $x$  with all its trailing zeroes removed. We build a set of

---

<sup>7</sup> Actually, results in [3] are stated for prefix-free sets, but it is trivial to make a set of strings prefix-free at the cost of doubling the average length.

strings  $P$  as follows: for each extent  $e$  of an internal node, we add to  $P$  the strings  $e^{\leftarrow}$ ,  $e1$ , and, if  $e \neq 111 \cdots 11$ , we also add to  $P$  the string  $(e1^+)^{\leftarrow}$ , where  $e1^+$  denotes the successor of length  $|e1|$  of  $e1$  in lexicographical order (numerically, it is  $e1 + 1$ ). We build a monotone minimal perfect hashing function  $h$  on  $P$ , noting the following easily proven fact:

**Proposition 1** *The average length of the strings in  $P$  is at most  $3\ell$ .*

The second component of the range locator is a bit vector  $\mathbf{b}$  of length  $|P|$ , in which bits corresponding to the names of leaves are set to one. The vector is endowed with a ranking structure  $\text{rank}_{\mathbf{b}}$  (see Figure 2).

It is now immediate that given a node name  $x$ , by hashing  $x^{\leftarrow}$  and ranking the bit position thus obtained in  $\mathbf{b}$ , we obtain the left extreme of the range of leaves under  $x$ . Moreover, performing the same operations on  $(x^+)^{\leftarrow}$ , we obtain the right extreme. All these strings are in  $P$  by construction, except for the case of a node name of the form  $111 \cdots 11$ ; however, in that case the right extreme is just the number of leaves (see Algorithm 4 for the details).

A range locator uses at most  $3n + o(n)$  bits for  $\mathbf{b}$  and its selection structures. Thus, space usage is dominated by the monotone hashing component. Using the structures described above, we obtain:

**Theorem 4.** *There are structures implementing range location in time  $O(|x|/w)$  using  $O(n \log \ell)$  bits of space, and in  $O(|x|/w + \log |x|)$  time using  $O(n \log \log \ell)$  bits of space.*

We remark that other combinations of monotone minimal perfect hashing and succinct data structures can lead to similar results. For instance, we could store the trie structure using a preorder standard balanced parentheses representation, use hashing to retrieve the lexicographical rank  $r$  of a node name, select the  $r$ -th open parenthesis, find in constant time the matching closed parenthesis and obtain in this way the number of leaves under the node. Among several such asymptotically equivalent solutions, we believe ours is the most practical.

## 5 Putting It All Together

In this section we gather the main results about prefix search:

**Theorem 5.** *There are structures implementing weak prefix search in space  $\text{HT}(S) + O(n \log \log \ell)$  with query time  $O(|p|/w + \log |p|)$ , and in space  $O(n\ell^{1/2} \log \ell)$  with query time  $O(|p|/w)$ .*

*Proof.* The first structure uses a hollow z-fast prefix trie followed by the range locator of Theorem 3: the first component provides the name  $n_{\alpha}$  of exit node of  $|p|$ ; given  $n_{\alpha}$ , the range locator returns the correct range. For the second structure, we use the structure defined in Section 3.3 followed by the first range locator of Theorem 2.

Actually, the second structure described in Theorem 5 can be made to occupy space  $O(n\ell^{1/c} \log \ell)$  for any constant  $c > 0$ , as shown in Appendix E:

**Theorem 6.** *For any constant  $c > 0$ , there is a structure implementing weak prefix search in space  $O(n\ell^{1/c} \log \ell)$  with query time  $O(|p|/w)$ .*

We note that all our time bounds can be translated into I/O bounds in the *cache-oblivious model* if we replace the  $O(|p|/w)$  terms by  $O(|p|/B)$ . The  $O(|p|/w)$  term represents appears in two places:

- The phase of precalculation of a hash-vector of  $\lceil |p|/w \rceil$  hash words on the prefix  $p$  which is later used to compute all the hash functions on prefixes of  $p$ .
- In the range location phase, where we need to compute  $x^{\leftarrow}$  and  $(x^+)^{\leftarrow}$ , where  $x$  is a prefix of  $p$  and subsequently compute the hash vectors on  $x^{\leftarrow}$  and  $(x^+)^{\leftarrow}$ .

Observe that the above operations can be carried on using arithmetic operations only without any additional I/O (we can use 2-wise independent hashing involving only multiplications and additions for computing the hash vectors and only basic arithmetic operations for computing  $x^{\leftarrow}$  and  $(x^+)^{\leftarrow}$ ) except for the writing the result of the computation which occupies  $O(|p|/w)$  words of space and thus take  $O(|p|/B)$  I/Os. Thus both of the two phases need only  $O(|p|/B)$  I/Os corresponding to the time needed to read the pattern and to write the result.

## 6 A space lower bound

In this section we show that the space usage achieved by the weak prefix search data structure described in Theorem 5 is optimal up to a constant factor. In fact, we show a matching lower bound for the easier problem of prefix counting (i.e., counting how many strings start with a given prefix), and consider the more general case where the answer is only required to be correct up to an additive constant less than  $k$ . We note that any data structure supporting prefix counting can be used to achieve approximate prefix counting, by building the data structure for the set that contains every  $k$ -th element in sorted order. The proof is in Appendix F.

**Theorem 7.** *Consider a data structure (possibly randomised) indexing a set  $S$  of  $n$  strings with average length  $\ell > \log(n) + 1$ , supporting  $k$ -approximate prefix count queries: Given a prefix of some key in  $S$ , the structure returns the number of elements in  $S$  that have this prefix with an additive error of less than  $k$ , where  $k < n/2$ . The data structure may return any number when given a string that is not a prefix of a key in  $S$ . Then the expected space usage on a worst-case set  $S$  is  $\Omega((n/k) \log(\ell - \log n))$  bits. In particular, if no error is allowed and  $\ell > (1 + \varepsilon) \log n$ , for constant  $\varepsilon > 0$ , the expected space usage is  $\Omega(n \log \ell)$  bits.*

Note that the trivial information-theoretical lower bound does not apply, as it is impossible to reconstruct  $S$  from the data structure.

It is interesting to note the connections with the lower and upper bounds presented in [14]. This paper shows a lower bound on the number of bits necessary to represent a set of strings  $S$  that, in the binary case, reduces to  $T(S) + \log \ell$ , and provide a matching data structure. Theorem 5 provides a *hollow* data structure that is sized following the naturally associated measure:  $HT(S) + O(n \log \log \ell)$ . Thus, Theorem 5 and 7 can be seen as the hollow version of the results presented in [14]. Improving Theorem 7 to  $HT(S) + o(HT(S))$  is an interesting open problem.

## 7 Applications

In this section we highlight some applications of weak prefix search. In several cases, we have to access the original data, so we are actually using weak prefix search as a component of a *systematic* (in the sense of [15]) data structure. However, our space bounds consider only the indexing data structure. Note that the pointers to a set of string of overall  $n\ell$  bits need in principle  $O(n \log \ell)$  bits of spaces to be represented; this space can be larger than some of the data structures themselves. Most applications can be turned into cache-oblivious data structures, but this discussion is postponed to the Appendix for the sake of space.

In general, we think that the space used to store and access the original data should not be counted in the space used by weak/blind/hollow structures, as the same data can be shared by many different structures. There is a standard technique, however, that can be used to circumvent this problem: by using  $2n\ell$  bits to store the set  $S$ , we can round the space used by each string to the nearest power of two. As a results, pointers need just  $O(n \log \log \ell)$  bits to be represented.

### 7.1 Prefix search and counting in minimal probes

The structures for weak prefix search described in Section 5 can be adapted to solve the prefix search problem within the same bounds, provided that the actual data are available, although typically in some slow-access memory. Given a prefix  $p$  we get an interval  $[i..j)$ . If there exists some string in the data set prefixed by  $p$ , then it should be at one of the positions in interval  $[i..j)$ , and all strings in that interval are actually prefixed by  $p$ . So we have reduced the search to two alternatives: either all (and only) strings at positions in  $[i..j)$  are prefixed by  $p$ , or the table contains no string prefixed by  $p$ . This implies the two following results:

- We can report all the strings prefixed by a prefix  $p$  in optimal number of probes. If the number of prefixed strings is  $t$ , then we will probe exactly  $t$  positions in the table. If no string is prefixed by  $p$ , then we will probe a single position in the table.
- We can count the number of strings prefixed by a given prefix in just one probe: it suffices to probe the table at any position in the interval  $[i..j)$ : if the returned string is prefixed by  $p$ , we can conclude that the number of strings prefixed by  $p$  is  $j - i$ ; otherwise, we conclude that no string is prefixed by  $p$ .

## 7.2 Range emptiness and search with two additional probes

The structures for weak prefix search described in Section 5 can also be used for range emptiness and search within the same bounds, again if the actual data is available. In the first case, given two strings  $a$  and  $b$  we ask whether any string in the interval  $[a..b]$  belongs to  $S$ ; in the second case we must report all such strings.

Let  $p$  the longest common prefix of  $a$  and  $b$  (which can be computed in time  $O(|p|/w)$ ). Then we have two sub-cases

- The case  $p = a$  ( $a$  is actually a prefix of  $b$ ). We are looking for all strings prefixed by  $a$  which are lexicographically smaller than  $b$ . We perform a prefix query for  $a$ , getting  $[i..j)$ . Then we can report all elements in  $S \cap [a..b]$  by doing a scan strings at positions in  $[i..j)$  until we encounter a string which is not in interval  $[a..b]$ . Clearly the number of probed positions is  $|S \cap [a..b]| + 1$ .
- The case  $p \neq a$ . We perform a prefix query for  $p0$ , getting  $[i_0..j_0)$  and another query for  $p1$ , getting  $[i_1..j_1)$ . Now it is immediate that if  $S \cap [a..b]$  is not empty, then necessarily it is made by a suffix of  $[i_0..j_0)$  and by a prefix of  $[i_1..j_1)$ . We can now report  $S \cap [a..b]$  using at most  $|S \cap [a..b]| + 2$  probes; we start from the end of the first interval and scan backwards until we find an element not in  $[a..b]$ ; then, we start from the beginning of the second interval and scan forwards until we find an element not in  $[a..b]$

We report all elements thus found: clearly, we make at most two additional probes. In particular, we can report whether  $S \cap [a..b]$  is empty in at most two probes. These results improve the space bounds of the index described in [1], provide a new index using just  $\text{HT}(S) + O(n \log \log \ell)$  bits, and give bounds in terms of the average length.

## References

1. S. Alstrup, G. S. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pages 476–482, 2001.
2. D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: Searching a sorted table with  $O(1)$  accesses. In *Proceedings of the 20th Annual Symposium On Discrete Mathematics (SODA)*, pages 785–794. ACM Press, 2009.
3. D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Theory and practise of monotone minimal perfect hashing. In *Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2009.
4. M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string b-trees. In *Proceedings of the 25th ACM symposium on Principles of Database Systems*, pages 233–242, New York, NY, USA, 2006. ACM.
5. G. S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *SODA*, pages 581–590, 2006.
6. A. Brodnik and J. I. Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.

7. D. Charles and K. Chellapilla. Bloomier filters: A second look. In *Proc. ESA 2008*, 2008.
8. D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage (extended abstract). In *SODA*, pages 383–391, 1996.
9. M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, 1992.
10. M. Dietzfelbinger and R. Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In *Proceedings of 35th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5125 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 2008.
11. P. Elias. Efficient storage and retrieval by content and address of static files. *J. Assoc. Comput. Mach.*, 21(2):246–260, 1974.
12. P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.
13. P. Ferragina and R. 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.
14. P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter. On searching compressed string collections cache-obliviously. In *Proceedings of the 27th ACM symposium on Principles of Database Systems*, pages 181–190, 2008.
15. A. Gál and P. Miltersen. The cell probe complexity of succinct data structures. *Theoret. Comput. Sci.*, 379(3):405–417, 2007.
16. A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theor. Comput. Sci.*, 387(3):313–331, 2007.
17. N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In S. W. Keckler and L. A. Barroso, editors, *ISCA*, pages 184–195. ACM, 2009.
18. G. Jacobson. Space-efficient static trees and graphs. In *In Proc 30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989.
19. M. Pătraşcu and M. Thorup. Randomization does not help searching predecessors. In *Proc. 18th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 555–564, 2007.
20. E. Porat. An optimal bloom filter replacement based on matrix solving, 2008. arXiv:0804.1845v1.
21. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding  $k$ -ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*, pages 233–242. ACM Press, 2002.

## A Conclusions

We have presented two data structures for prefix search that provide different space/time tradeoffs. In one case (Theorem 5), we prove a lower bound showing that the structure is space optimal. In the other case (Theorem 6) the structure is time optimal. It is also interesting to note that the space usage of the time-optimal data structure can be made arbitrarily close to the lower bound. Our structures are based on *range locators*, a general building block for static data structures, and on structures that are able to map prefixes to names of the associated exit node. In particular, we discuss a variant of the z-fast trie, the *z-fast prefix trie*, that is suitable for prefix searches. Our variant carries on the good properties of the z-fast trie (truly linear space and logarithmic access time) and significantly widens its usefulness by making it able to retrieve the name of the exit node of prefixes. We have shown several applications in which sublinear indices access very quickly data in slow-access memory, improving some results in the literature.

## B Proof of Theorem 1

Let  $E$  be the sum of the lengths of the extents of internal nodes, and  $M$  the sum of the lengths of the strings in the trie. We show equivalently that  $E \leq M(n-1)/n - n + 1$ . This is obviously true if  $n = 1$ . Otherwise, let  $r$  be the length of the compacted path at the root, and let  $n_0, n_1$  be the number of leaves in the left and right subtrie; correspondingly, let  $E_i$  the sum of lengths of the extents of each subtrie, and  $M_i$  the sum of the lengths of the strings in each subtrie, stripped of their first  $r + 1$  bits. Assuming by induction  $E_i \leq M_i(n_i - 1)/n_i - n_i + 1$ , we have to prove

$$E_0 + (n_0 - 1)(r + 1) + E_1 + (n_1 - 1)(r + 1) + r \leq \frac{n_0 + n_1 - 1}{n_0 + n_1} (M_0 + M_1 + (n_0 + n_1)(r + 1)) - n_0 - n_1 + 1,$$

which can be easily checked to be always true under the assumption above.

## C Proof of the space bound claimed in Section 3.3

First of all, it can easily be proved that the domain of  $F$  is  $O(n\ell^{1/2})$  in size. Each  $\alpha$  contributes at most  $s$  prefixes whose lengths are in interval  $(i_\alpha \dots \min(\bar{k}s, j_\alpha)]$ . It also contributes at most  $(j_\alpha - i_\alpha)/s + 1$  prefixes whose lengths are of the form  $ks$ , where  $ks \in (i_\alpha \dots j_\alpha]$ . Overall the total number of prefixes is no more than:

$$\sum_{\alpha} (s + (j_\alpha - i_\alpha)/s + 1) = (s + 1)(2n - 1) + \sum_{\alpha} ((j_\alpha - i_\alpha)/s)$$

The sum of lengths of skip intervals of all nodes of the trie  $T(S)$  is no larger than sum of lengths of strings  $n\ell$ :

$$T(S) = \sum_{\alpha} (j_{\alpha} - i_{\alpha}) \leq n\ell$$

From that we have:

$$\sum_{\alpha} ((j_{\alpha} - i_{\alpha})/s) = \frac{1}{s} \sum_{\alpha} (j_{\alpha} - i_{\alpha}) \leq \frac{1}{s} n\ell \leq ns$$

Summing up, the total number of prefixes is less than  $(s+1)(2n-1) + ns = O(ns) = O(n\ell^{1/2})$ . Since the output size of the function  $F$  is bounded by  $\max_{\alpha} \log |c_{\alpha}| \leq \log L$ , where  $L$  is the *maximum* string length, we would obtain the space bound  $O(n\ell^{1/2} \log L)$ . To prove the strict bound  $O(n\ell^{1/2} \log \ell)$ , we need to further refine the structure so that the “cutting step”  $s$  is larger in the deeper regions of the trie.

Let  $S^{-}$  be the subset of strings of  $S$  of length less than  $\ell(\log n)^2$ , and  $S^{+}$  the remaining strings. We will change the step  $s$  after depth  $\ell(\log n)^2$ . Let  $s = \lceil \ell^{1/2} \rceil$  and let  $s^{+} = \lceil (\ell(\log n)^2)^{1/2} \rceil = \lceil \ell^{1/2} \log n \rceil$ . We will say that a node is *deep* if its extent is long at least  $\ell(\log n)^2$ . We will split  $F$  into a function  $F^{-}$  with output size  $\log(\ell(\log n)^2) = \log \ell + 2 \log \log n = O(\log \ell)$  that maps prefixes shorter than  $\ell(\log n)^2$  (*short prefixes*), and a function  $F^{+}$  with output size  $\log(\ell n) = \log n + \log \ell$  that maps the remaining *long prefixes*. For every node  $\alpha$  with skip interval  $[i_{\alpha} .. j_{\alpha}]$ , we consider three cases:

1. If  $j_{\alpha} < \ell(\log n)^2$  (a non-deep node), we will store the prefixes of  $e_{\alpha}$  that have lengths either of the form  $ks$ , where  $ks \in (i_{\alpha} .. j_{\alpha}]$ , or in  $(i_{\alpha} .. \min\{\bar{k}s, j_{\alpha}\}]$ , where  $\bar{k}$  is the minimum  $k$  such that  $ks > i_{\alpha}$ . Those prefixes are short, so they will be mapped using  $F^{-}$ .
2. If  $i_{\alpha} < \ell(\log n)^2 \leq j_{\alpha}$  (a deep node with non-deep parent), we store the following prefixes of  $e_{\alpha}$ :
  - (a) Prefixes of lengths  $ks$ , where  $ks \in (i_{\alpha} .. \ell(\log n)^2)$ , or of lengths in  $(i_{\alpha} .. \min\{\bar{k}s, \ell(\log n)^2\}]$ , where  $\bar{k}$  is the minimum  $k$  such that  $ks > i_{\alpha}$ . Those prefixes are short, so they will be mapped using  $F^{-}$ .
  - (b) Prefixes of lengths  $\ell(\log n)^2 + ks^{+}$ , where  $\ell(\log n)^2 + ks^{+} \in [\ell(\log n)^2 .. j_{\alpha}]$ . Those prefixes are long, so they will be mapped using  $F^{+}$ .
3. If  $i_{\alpha} \geq \ell(\log n)^2$  (a deep node with a deep parent), we will store all prefixes that have lengths either of the form  $\ell(\log n)^2 + ks^{+}$ , where  $\ell(\log n)^2 + ks^{+} \in (i_{\alpha} .. j_{\alpha}]$ , or in  $(i_{\alpha} .. \min\{\ell(\log n)^2 + \bar{k}s^{+}, j_{\alpha}\}]$ , where  $\bar{k}$  is the minimum  $k$  such that  $\ell(\log n)^2 + ks^{+} > i_{\alpha}$ . Those prefixes are long, so they will be mapped using  $F^{+}$ .

The function  $F$  is now defined by combining  $F^{-}$  and  $F^{+}$  in the obvious way. To retrieve the exit node of a string  $p$  that is a prefix of some string in  $S$ , we have two cases: if  $|p| < \ell(\log n)^2$ , we consider the string  $q = p[0 .. |p| - |p| \bmod s)$ , otherwise we consider the string  $q = p[0 .. |p| - (|p| - \lceil \ell(\log n)^2 \rceil) \bmod s^{+})$ . Then, we check whether  $G(p[0 .. F(q)]) \geq |p|$  (i.e., whether  $p$  is a prefix of the extent

of the exit node of  $q$ ). If this is the case, then we conclude clearly  $p$  has the same exit node of  $q$  (i.e.,  $p[0..F(q)]$ ). Otherwise, the map  $F$  gives the name of the exit node of  $p : p[0..F(p)]$ .

The space bound holds immediately for  $F^-$ , as we already showed that prefixes (long and short) are overall  $O(n\ell^{1/2})$ , and  $F^-$  has output size  $\log(\ell(\log n)^2) = O(\log \ell)$ .

To bound the size of  $F^+$ , we first bound the number of deep nodes. Clearly either a deep node is a leaf or it has two deep children. If a deep node is a leaf then its extent is long at least  $\ell(\log n)^2$ , so it represent a string from  $S^+$ . Hence, the collection of deep nodes constitute a forest of complete binary trees where the number of leaves is the number of strings in  $S^+$ . As the number of strings in  $S^+$  is at most  $n/(\log n)^2$ , we can conclude that the total number of nodes in the forests (i.e., the number of deep nodes) is at most  $2n/(\log n)^2 - 1$ . For each deep node we have two kinds of long prefixes:

1. Prefixes that have lengths of the form  $\ell(\log n)^2 + ks^+$ . Those prefixes can only be prefixes of long strings, and for each long string  $x \in S^+$ , we can have at most  $|x|/s^+$  such prefixes. As the total length of all strings in  $S^+$  is at most  $n\ell$ , we conclude that the total number of such prefixes is at most  $n\ell/s^+ = O(n\ell/\lceil \ell^{1/2} \log n \rceil) = O(n\ell^{1/2}/(\log n))$ .
2. Prefixes that have lengths in  $(i_\alpha.. \min\{\ell(\log n)^2 + \bar{k}s^+, j_\alpha\})$  or in  $[\ell(\log n)^2.. \min\{\ell(\log n)^2 + \bar{k}s^+, j_\alpha\}]$  for a node  $\alpha$ , where  $\bar{k}$  is the minimum  $k$  such that  $\ell(\log n)^2 + ks^+ > i_\alpha$ . We can have at most  $s^+$  prefix per node: since we have at most  $2n/(\log n)^2 - 1$  nodes, the number of prefixes of that form is  $O(s^+n/(\log n)^2) = O(n\ell^{1/2}/(\log n))$ .

As we have a total of  $O(n\ell^{1/2}/(\log n))$  long prefixes, and the output size of  $F^+$  is  $O(\log \ell + \log n)$ , we can conclude that total space used for  $F^+$  is bounded above by  $O((\log \ell + \log n)n\ell^{1/2}/\log n) = O(n\ell^{1/2} \log \ell)$ .

Finally, we remark that implementing  $G$  by means of a compressed function we need just  $\text{HT}(S) + O(n \log \log \ell) + O(n) = O(n \log \ell)$  bits of space.

## D Correctness proof of Algorithm 1

The correctness of Algorithm 1 is expressed by the following

**Lemma 1.** *Let  $X = \{p_0 = \varepsilon, p_1, \dots, p_t\}$ , where  $p_1, p_2, \dots, p_t$  are the extents of the nodes of the trie that are prefixes of  $p$ , ordered by increasing length. Let  $(\ell..r)$  be the interval maintained by the algorithm. Before and after each iteration the following invariants are satisfied:*

1. *there exists at most a single  $b$  such that  $2^i b \in (\ell..r)$ ;*
2.  *$\ell = |p_j|$  for some  $j$ , and  $\ell \leq |p_t|$ ;*
3. *during the algorithm,  $T$  is only queried with strings that are either the handle of an ancestor of  $\text{exit}(p)$ , or a pseudohandle of  $\text{exit}(p)$ ; so,  $T$  is well-defined on all such strings;*
4.  *$|p_t| \leq r$ ;*

We will use the following property of 2-fattest numbers, proved in [2]:

**Lemma 2.** *Given an interval  $[x..y]$  of strictly positive integers:*

1. *Let  $i$  be the largest number such that there exists an integer  $b$  satisfying  $2^i b \in [x, y]$ . Then  $b$  is unique, and the number  $2^i b$  is the 2-fattest number in  $[x..y]$ .*
2. *If  $y - x < 2^i$ , there exists at most a single value  $b$  such that  $2^i b \in [x..y]$ .*
3. *If  $i$  is such that  $[x..y]$  does not contain any value of the form  $2^i b$ , then  $y - x + 1 \leq 2^i - 1$  and the interval may contain at most one single value of the form  $2^{i-1} b$ .*

Now, the correctness proof of Lemma 1 follows. (1) Initially, when  $i = \lfloor \log |p| \rfloor$  we have  $(\ell..r) = (0..|p|)$ , and this interval contains at most a single value of the form  $2^i b$ , that is  $2^i$ . Now after some iteration suppose that we have at most a single  $b$  such that  $2^i b \in (\ell..r)$ . We have two cases:

- There is no  $b$  such that  $2^i b \in (\ell..r)$ . Then, the interval remains unchanged and, by Lemma 2 (3), it will contain at most a single value of the form  $2^{i-1} b$ .
- There is a single  $b$  such that  $2^i b \in (\ell..r)$ . The interval may be updated in two ways: either we set the interval to  $(g..r)$  for some  $g \geq 2^i b$  or we set the interval to  $(\ell..2^i b)$ . In both cases, the new interval will no longer contain  $2^i b$ . By invariant 3. of Lemma 2, the new interval will contain at most a single value of the form  $2^{i-1} b$ .

(2) The fact that  $\ell = |p_j|$  for some  $j$  is true at the beginning, and when  $\ell$  is reassigned it remains true: indeed, if  $T(p[0..2^i b]) = g < |p|$  this means that  $p[0..2^i b]$  is the handle of a node  $\alpha$  found on the path to  $\text{exit}(p)$ , and  $g = |e_\alpha|$ ; but since  $p$  is a prefix of some string,  $p[0..g] = e_\alpha$  and the latter is  $p_j$  for some  $j$ . This fact also implies  $\ell \leq |p_t|$ , since the  $p_i$ 's have decreasing lengths.

(3) By (2),  $\ell$  is always the length of the extent of some  $p_j$ , whereas  $r = |p|$  at the beginning, and then it can only decrease; so  $(\ell..r)$  is a union of some skip intervals of the ancestors of  $\text{exit}(p)$  and of an initial part of the skip interval of the node  $\text{exit}(p)$  itself. Hence, its 2-fattest number is either the handle of some of the ancestors (possibly, of  $\text{exit}(p)$  itself) or a pseudohandle of  $\text{exit}(p)$  (this can only happen if  $r$  is not larger than the 2-fattest number of the skip interval of  $\text{exit}(p)$ ).

(4) The property is true at the beginning. Then,  $r$  is reduced only in two cases: either  $2^i b$  is the 2-fattest number of the skip interval of  $\text{exit}(p)$  (in this case,  $g$  is assigned  $|e_{\text{exit}(p)}| \geq |p|$ ); or we are querying with a pseudohandle of  $\text{exit}(p)$  or with the handle of a leaf (in this case,  $g$  is assigned the value  $\infty$ ). In both cases,  $r$  is reduced to  $2^i b$ , which is still not smaller than the extent of the parent of  $\text{exit}(p)$ .

## E Proof of Theorem 6

Rather than describing the proof from scratch, we describe the changes necessary to the proof given in Appendix C.

The main idea is that of setting  $t = \lceil \ell^{1/c} \rceil$ ,  $t^+ = \lceil \ell^{1/c} \log n \rceil$ , and let  $s = t^{c-1}$  and  $s^+ = t^{+c-1}$ . In this way, since  $n\ell = O(nt^c)$  clearly the prefixes of the form  $ks$  and  $\ell(\log n)^c + ks^+$  are  $O(n\ell^{1/c})$  and  $O(n\ell^{1/c}/\log n)$ . The problem is that now the prefixes at the start of each node (i.e., those of length  $i_\alpha, i_\alpha + 1, \dots$ ) are too many.

To obviate to this problem, we record significantly less prefixes. More precisely we record sequences of prefixes of increasing lengths:

- For non deep nodes we first store prefixes of lengths  $i_\alpha, i_\alpha + 1, \dots$  until we hit a multiple of  $t$ , say  $k_0t$ . Then we record prefixes of lengths  $(k_0 + 1)t, (k_0 + 2)t, \dots$  until we hit a multiple of  $t^2$ , and so on, until we hit a multiple of  $s$ . Then we finally terminate by recording prefixes of lengths multiple of  $s$ .
- We work analogously with  $t^+$  and  $s^+$  for deep nodes whose parents are also deep nodes. That is we store all prefixes of lengths  $i_\alpha, i_\alpha + 1, \dots$  until we hit a length of the form  $\ell(\log n)^c + k_0t^+$ , then record all prefixes of lengths  $\ell(\log n)^c + (k_0 + 1)t^+, \ell(\log n)^c + (k_0 + 2)t^+$  until we hit a length of the form  $\ell(\log n)^c + k_1t^{+2}$ , and so on until we record a length of the form  $\ell(\log n)^c + k_{c-2}s^+$ . Then we finally store all prefixes of lengths  $\ell(\log n)^c + (k_{c-2} + 1)s^+, \ell(\log n)^c + (k_{c-2} + 2)s^+, \dots$ .
- For deep nodes with non deep parents, we do the following two things:
  - We first record short prefixes. We record all strings of lengths  $i_\alpha, i_\alpha + 1, \dots$  until we either hit a length multiple of  $t$  or length  $\ell(\log n)^c$ . If we have hit length  $\ell(\log n)^c$  we stop recording short prefixes. Otherwise, we continue in the same way, with prefixes of lengths multiple of  $t^u$  for increasing  $u = 1 \dots c - 1$  each time terminating the step if we hit a multiple of  $t^{u+1}$  or completely halt recording short prefixes if we hit length  $\ell(\log n)^c$ .
  - Secondly, we record long prefixes. That is all prefixes of lengths of the form  $\ell(\log n)^c + ks^+$

Clearly each node contributes at most  $O(ct)$  short prefixes ( $O(ct^+)$  long prefixes respectively). In the first case, there are obviously  $O(n\ell^{1/c})$  short prefixes. In the second case, since the number of deep nodes is at most  $2n/(\log n)^2 - 1$  there are at most  $O(n\ell^{1/c}/\log n)$  long prefixes. Overall,  $F^-$  requires  $O(n\ell^{1/c} \log(\ell(\log n)^2)) = O(n\ell^{1/c} \log \ell)$  bits, whereas  $F^+$  requires  $O((n\ell^{1/c}/\log n) \log(\ell n)) = O(n\ell^{1/c} \log \ell)$  bits.

The query procedure is modified as follows: for  $i = c - 1, c - 2, \dots, 0$ , if  $p$  is short we consider the string  $q = p[0..|p| - |p| \bmod t^i]$  and check whether  $G(p[0..F(q)]) \geq |p|$ . If this happen, we know that  $p[0..F(q)]$  is the name of the exit node of  $p$  and we return it. Note that if  $p$  is a prefix of some string in  $S$ , this must happen at some point (eventually at  $i = 0$ ). If  $p$  is long, we do the same using  $t^+$  and  $s^+$ . That is we consider the string  $q = p[0..|p| - (|p| - \lceil \ell(\log n)^2 \rceil) \bmod t^{+i}]$  and check whether  $G(p[0..F(q)]) \geq |p|$ . If this happen, we know that  $p[0..F(q)]$  is the name of the exit node of  $p$  and we return it.

This procedure finds the name of the exit node of  $|p|$  in time  $O(|p|/w)$ .

## F Proof of Theorem 7

Let  $u = 2^\ell$  be the number of possible keys of length  $\ell$ . We show that there exists a probability distribution on key sets  $S$  such that the expected space usage is  $\Omega((n/k) \log \log(u/n))$  bits. By the “easy directions of Yao’s lemma,” this implies that the expected space usage of any (possibly randomised) data structure on a worst case input is at least  $\Omega((n/k) \log \log(u/n))$  bits. The bound for  $\ell > (1 + \varepsilon) \log n$  and  $k = 1$  follows immediately.

Assume without loss of generality that  $n/(k+1)$  and  $k$  are powers of 2. All strings in  $S$  will be of the form  $abc$ , where  $a \in 2^{\log_2(n/(k+1))}$ ,  $b \in 2^{\ell - \log_2(n/(k+1)) - \log_2 k}$ , and  $c \in 2^{\log_2 k}$ . Let  $t = \ell - \log_2(n/(k+1)) - \log_2 k$  denote the length of  $b$ . For every value of  $a$  the set will contain exactly  $k+1$  elements: One where  $b$  and  $c$  are strings of 0s, and for  $b$  chosen uniformly at random among strings of Hamming weight 1 we have  $k$  strings for  $c \in 2^{\log_2 k}$ . Notice that the entropy of the set  $S$  is  $n/(k+1) \log_2 t$ , as we choose  $n/(k+1)$  values of  $b$  independently from a set of  $t$  strings. To finish the argument we will need to show that any two such sets require different data structures, which means that the entropy of the bit string representing the data structure for  $S$  must also be at least  $n/(k+1) \log_2 t$ , and in particular this is a lower bound on the expected length of the bit string.

Consider two different sets  $S'$  and  $S''$ . There exists a value of  $a$ , and distinct values  $b'$ ,  $b''$  of Hamming weight 1 such that  $S'$  contains all  $k$   $\ell$ -bits strings prefixed by  $ab'$ , and  $S''$  contains all  $k$   $\ell$ -bits strings prefixed by  $ab''$ . Assume without loss of generality that  $b'$  is lexicographically before  $b''$ . Now consider the query for a string of the form  $a0^\ell$ , which is a prefix of  $ab'$  but not  $ab''$  – such a string exists since  $b'$  and  $b''$  have Hamming weight 1. The number of keys with this prefix is  $k+1$  and 1, respectively, for  $S'$  and  $S''$ , so the answers to the queries must be different (both in the multiplicative and additive case). Hence, different data structures are needed for  $S'$  and  $S''$ .

## G Applications in the cache-oblivious model

*Prefix search and counting.* For prefix search our constant time results imply an improvement in the cache-oblivious model compared to results in [5, 4, 14]. Suppose that the strings are stored contiguously in increasing lexicographic order. Using Elias–Fano encoding [11] we can store pointers to strings such that starting position of a string can be located in constant time. Using our fastest result, a weak prefix search takes just  $O(|p|/B)$  for the weak prefix search and then just  $O(1)$  time to locate the pointers to beginning of first string in the range (which we note by  $R_s$ ) and to the ending of the last string in the range (which we note by  $R_e$ ). Then the prefix search can be carried in optimal  $O(K/B)$  I/Os, where  $K = R_e - R_s + 1$  is the total length of the strings satisfying the query or in optimal  $O(|p|/B)$  I/Os if no string satisfies the query. That is we read the first  $|p|$  bits of first string in the range (or any other string in the range), just to check whether those  $|p|$  bits are identical to  $p$ , in which case, we read the remaining  $K - |p|$  bits satisfying the search query.

For counting queries we also clearly have the optimal  $O(|p|/B)$  I/Os bound as we can just do a weak prefix search in  $O(|p|/B)$  I/Os, locating a range of strings and then retrieving the first  $|p|$  bits of any key in the range.

*Range emptiness.* In the cache oblivious model a range query for an interval  $[a..b]$  of variable length strings can be done in optimal  $O(K/B + |a|/B + |b|/B)$  I/Os where  $K$  is size of output. For that, we can slightly modify the way strings are stored in memory. As before, we store strings in increasing lexicographic order. but this time we store the length of each string at the end and at the beginning of each string. We note that storing the string lengths using Elias  $\delta$  or  $\gamma$  coding [12] never increases the lengths of any individual string by more than a constant factor and that they occupy no more than constant number of memory words each (moreover total space wasted by stored lengths is  $O(n\ell)$  bits). Thus we can scan the strings backward and forward in optimal time in cache-oblivious model. That is the number of blocks we read is no more than a constant factor than necessary.

The algorithm becomes very clear now : given the interval  $[a..b]$ , as before, we either perform a single prefix query for prefix  $a$  in case  $a$  is a prefix of  $b$  or otherwise perform two prefix queries for  $p0$  and  $p1$  (where  $p$  is the longest common prefix). Then using Elias-Fano we can locate the pointers to first string in first case or locate the two pointers to last string in first interval and first string in second interval in the second case. Then we only need to scan forward and backward checking each time whether the strings are in the interval. Each checking needs only to read  $\max(|a|, |b|)$  bits. Thus we do not perform more than  $O(K/B + |a|/B + |b|/B)$  I/Os. In particular checking for the two boundary strings does not take more than  $O(|a|/B + |b|/B)$  I/Os. Even reading the length of the two boundary strings takes no more than  $O(1)$  I/Os as the lengths are encoded in no more than a constant number of words and we have  $w < B$ .

## H Extensions to larger alphabets

Our data structures for for weak prefix search can easily be extended to work for any integer alphabet of size  $\sigma$ .

### H.1 First extension approach

The most straightforward approach consists in considering each character in the alphabet as a sequence of  $\log(\sigma)$  bits. The length of a string of length  $p$  becomes  $|p| \log \sigma$ , and the total length of the strings in the collection becomes  $O(n\ell \log \sigma)$ . In this context query times for weak prefix search become:

- The query time for the space optimal solution becomes  $O((|p| \log \sigma)/w + \log(|p| \log \sigma)) = O((|p| \log \sigma)/w + \log |p| + \log \log \sigma)$ , while the space usage becomes  $\text{HT} + O(n(\log \log \ell + \log \log \log \sigma)) = \text{HT} + O(n \log \log \ell)$ , where HT is the hollow trie size of the compacted binary trie built on the transformed set of strings.

- The query time of the time optimal solution becomes  $O((|p| \log \sigma)/w)$ , while space usage becomes  $O(n(\ell \log \sigma)^{1/c} \log(\ell \log \sigma))$ . Note that query time is still optimal in this case.

*Cache oblivious model* The query times for prefix range and general range queries in the cache oblivious model become :

- For the the space optimal solution, query time for a prefix range query becomes  $(|p| \log \sigma)/B + \log |p| + \log \log \sigma + K \log(\sigma)/B$  for a prefix search on a prefix  $p$ , where  $K$  is length of the output in number of characters. For a range query on an interval  $[a, b]$  the query time becomes  $((|a|+|b|) \log \sigma)/B + \log |p| + \log \log \sigma + K \log(\sigma)/B$ .
- The query time of the time optimal solution becomes  $O(|p| \log \sigma)/B + K \log(\sigma)/B$  for a prefix query and  $((|a| + |b|) \log \sigma)/B + \log |p| + \log \log \sigma + K \log(\sigma)/B$  for range queries.

## H.2 Second extension approach

We now describe the second extension approach which is almost as simple as the first one. We do a small modification of our scheme which will permit us to gain time or space compared to the naive solution above. We first notice that the procedure which identifies exit nodes does not depend at all on the alphabet being binary. Our solution will be built on a top of a compacted trie (which we note by  $T_\sigma$ ) built on the set of strings over original alphabet (we do not convert the strings to binary alphabet). Thus any node in the internal node in the trie will have between 2 and  $\sigma$  children where each children is labeled with a character in  $[0, \sigma - 1]$ . All the data structures defined in section 4 can be reused without any modification on an alphabet of size  $\sigma$  instead of an alphabet of size 2. The range locator in section 5 also extends directly to larger alphabets. A more detailed description of the range locator extension to larger alphabets is given below. Before that we will first present an analysis of space usage of our data structures. For that we first redefine the hollow z-fast trie size for larger alphabets (which we not by  $HT_\sigma(S)$ ) as :

$$HT_\sigma(S) = \sum_{\alpha} (\text{bitlength}(|c_\alpha|) + 1) - 1.$$

where the summation is done over internal nodes only. Note that for a given data set on alphabet  $\sigma$ ,  $HT_\sigma$  could be much smaller than  $HT$ , for two reasons :

- The first reason is that the number of internal nodes in  $T_\sigma$  could be as small as  $O(n/\sigma)$ . In contrast the trie  $T$  has exactly  $n$  internal nodes.
- The second reason is that the length  $|c_\alpha|$  is expressed in number of characters instead of number of bits. That is  $\text{bitlength}(|c_\alpha|)$  uses  $\log \log \sigma$  bits.

Thus space used by the functions which maps prefixes to their exit nodes can be restated as  $HT_\sigma + O(n \log \log \ell)$  bits for the z-fast hollow trie and  $O(n\ell^{1/c}(\log \ell +$

$\log \log n$ )) for the time optimal solution. The difference between space usage in this variant and the variant for binary alphabet is the  $(\log \log n)$  term. This term comes from the function  $F^-$  which dominates space usage and uses  $O(\log \ell + \log \log n)$  bits per inserted prefix. This term was previously absorbed by the  $\log \ell$  term as for binary alphabet where we had  $\ell \geq \log n$ .

*Range locator for larger alphabets* The adaptation of range locator for large alphabets is also straightforward. The main difference between range locator for binary alphabet and the range locator for an alphabet of size  $\sigma$  is that the names of exit nodes are now of the form  $pc$ , where  $p$  is the extent of some node  $\alpha$  in the trie and  $c$  is a character from  $[0 \dots \sigma)$  instead of  $\{0, 1\}$ . For each node whose name is  $pc$  we will store in the mmphf the names  $pc$  and  $(pc)^+$ , where  $(pc)^+$  is define recursively in the following way: if  $c$  is the last character in the alphabet then  $(pc)^+ = p^+$ , otherwise  $(pc)^+ = pc^+$ , where  $c^+$  is the successor of  $c$  in the lexicographic order of the alphabet. The number of elements inserted in the mmphf is at most  $2(2n - 1)$ . This is the case because we have  $2n - 1$  for each node contributes at most two prefixes to the range locator. Note that because the two prefixes associated with two different nodes may overlap, we could have less  $2(2n - 1)$  prefixes.

Depending on the mmphf we used we get two different tradeoffs:

- Space usage  $O(n(\log \log \ell + \log \log \log \sigma))$  bits with query time  $O((|P| \log \sigma)/w + \log(|P| + \log \log(\sigma)))$  if we use logarithmic time mmphf.
- Space usage  $O(n(\log \ell + \log \log \sigma))$  bits with query time  $O((|P| \log \sigma)/w)$  if we use constant time mmphf.

*Improved space usage for constant time exit node mapping* The space usage of the constant time exit node mapping functions (functions  $F$  and  $G$ ) can be reduced from  $O(n\ell^{1/c}(\log \ell + \log \log n))$  to  $O(n\ell^{1/c}(\log \ell + \log \log \log n))$ . For that we will use three function  $F^1$ ,  $F^2$  and  $F^3$  instead of just two function  $F^-$  and  $F^+$ . That is  $F^1$  stores prefixes of lengths less  $\ell \log \log n$ ,  $F^2$  stores prefixes of lengths between  $\ell \log \log n$  and  $\ell \log n$  and finally  $F^3$  stores prefixes of lengths above  $\ell \log n$ . It can easily be seen that space usage is dominated by the function  $F^1$  which will thus use  $\log \ell + \log \log \log n$  bits of space per stored prefix. We could further reduce space usage by using four or more function, but we will see in next section that this does not give any asymptotic improvement.

### H.3 Putting things together

The second extension makes it possible to improve the space/time tradeoff for the logarithmic and constant time weak prefix search solutions.

*Constant time queries* Using the second extension, we note that space usage of constant time solution can be improved to  $O(n(\log \ell + \log \log \sigma))$  in case  $\ell$  is constant. Note that because we have  $\ell \log \sigma \geq \log n$ , we have that  $\log \sigma = \Omega(\log n)$ . The space usage for the first extension method is at least

$\Omega((\ell \log \sigma)^{1/c}(\log \ell + \log \log \sigma))$ . Thus with the second extension we have a logarithmic improvement in space usage compared to the first extension. However, we can do even better. That is even if we have  $\ell = O((\log \log n)^k)$  for any constant  $k$  (note that this implies that  $\log \sigma = \Omega(\log n / (\log \log n)^k)$ ), we can still obtain  $O(n(\log \ell + \log \log \sigma))$  bits of space. That is by choosing  $c = k + 1$ , the space usage of exit node mapping functions in second extension (the functions  $F^1$ ,  $F^2$ ,  $F^3$  and  $G$ ) method becomes  $O(\ell^{1/c}(\log \ell + \log \log \log n)) = O((\log \log n)^{k/(k+1)} \log \log \log n)$  bits and thus the space usage becomes dominated by the range locator which uses  $O(n(\log \ell + \log \log \sigma))$  bits of space.

*Logarithmic time queries* For logarithmic weak prefix searches, we can have two kinds of improvements. Either space usage improvement or query time improvement. That is by combining the second approach method with the range locator suitable for large alphabets, we get the following:

- By using a constant time mmphf as a sub-component of the range locator we get query time  $O((|P| \log \sigma)/w + \log(|P|))$  with  $O(n(\log \ell + \log \log \sigma))$  of space usage.
- By using the logarithmic time mmphs as a sub-component of the range locator, we get query time  $O((|P| \log \sigma)/w + \log(|P|) + \log \log \sigma)$  with improved  $HT_\sigma + O(n \log \log \ell)$  bits of space usage.

#### H.4 Cache oblivious model

The performance of prefix search naturally improves in the cache-oblivious model for the logarithmic time prefix search. Using the second extension combined with range locator based constant time mmphf, we get query time  $O(|P|/B + \log |P|)$  against  $O(|P|/B + \log |P| + \log \log \sigma)$  for the first extension method. With the first extension the used space is  $HT + O(n \log \ell)$  against  $O(n(\log \ell + \log \log \sigma))$  for the second extension. Note that the space usage of the first extension is better than that of second extension as we have  $n(\log \ell + \log \log \sigma) \geq HT$ . Note that the improvement in query time can be very significant in case  $\sigma$  is very large.

For constant time weak prefix queries, the query time is already optimal. However the space/time tradeoff can be improved using the second extension method. For a given constant  $c$ , the query time is  $O(|P|/B + c)$  for both the first and second extension, but space is for second extension is  $O(n(\ell^{1/c}(\log \ell + \log \log \log n) + \log \log \sigma))$  against  $O(n(\ell \log \sigma)^{1/c} \log(\ell \log \sigma))$  for the first extension method. It can be seen that for large  $\log \sigma$  the space saving can be significant. In the particular case where  $\log \sigma = \Omega(\log n / (\log \log n)^k)$  for some constant  $k$  and the  $\ell = (\log \log n)^k$  we can obtain  $O(|P|/B + k)$  using only  $O(n(\log \ell + \log \log \sigma))$  bits of space.