Università di Pisa
Dipartimento di Informatica

Technical Report: TR-12-08

Compressed static functions with applications to other dictionary problems

Djamal Belazzougui Rossano Venturini

July 12, 2012

Address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Compressed static functions with applications to other dictionary problems

Djamal Belazzougui * Rossano Venturini †

July 12, 2012

Abstract

Given a set of integer keys from a bounded universe along with associated data, the dictionary problem asks to answer two queries: membership and retrieval. Membership has to tell whether a given element is in the dictionary or not; retrieval has to return the data associated with the searched key. This paper studies three well-established relaxations of this basic problem: *(Compressed) Static functions, Approximate membership* and *Relative membership*.

(Compressed) Static functions. In this relaxation, also known as *retrieval-only dictionaries*, we are given a set $S \subseteq U$ of n integers and each of them has associated data from an alphabet of size σ. The problem asks to build a dictionary that, given a key $x \in S$, returns its associate data. Notice that, whenever $x \notin S$, arbitrary data is returned. This problem has been widely studied in the past [6, 8, 1, 16, 12, 11, 5, 7, 14]. Solutions to this problem have to carefully organize associated data, so that, they can be retrieved in constant time without the need of storing keys in S. Compressed static functions move a step forward: not only do not store the keys, but also achieve space complexities bounded in term of the entropy H_0 of the associated data. Such kind of solutions are very interesting mainly for two reasons. Firstly, being nH_0 at most $n \log \sigma$, these results are always at least as good as the uncompressed ones. Moreover, since the associated data often follow a skewed distribution, nH_0 could even become sublinear in n. The current best solutions are by Porat [16] and Hreinsson *et al.* [12]. The first one requires $n \log \sigma + o(n)$ bits of space, while the second one uses $(1+\delta)nH_0 + n \cdot \min(p_0 + 0.086, 1.82(1-p_0))$ bits of space, where p_0 is the probability of the most frequent symbol and δ is a constant greater than 0. Thus, the space complexities of these solutions are incomparable: the former has a sublinear overhead but is not compressed, while the latter is suboptimal due to the factor $(1+\delta)$ to multiply H_0 and has an overhead that may be $\Theta(n)$ depending on p_0. Our optimal scheme achieves the best of the two being the first known solution obtaining simultaneously constant query time, compressed space (nH_0), and sublinear overhead ($o(n)$). We strongly believe that these characteristics makes the use of static functions significantly more appealing for many applications.

Approximate membership. The approximate membership problem has been studied for decades and the Bloom filter data structure [4] is probably the most popular and widely used technique solving it. With Bloom filters we can represent a set of n integers by using $n \log \frac{1}{\epsilon} \log e$ bits of space with false positive probability (fpp) ϵ. Both its space and time complexities are non-optimal: space is a constant factor away from optimal, and query time is logarithmic in $\frac{1}{\epsilon}$. Constant time approximate membership data structures are able to achieve optimal $n \log \frac{1}{\epsilon} + o(n)$ bits of space only if $\frac{1}{\epsilon}$ is a power of two [5, 8, 15, 16]. The current best solution for general ϵ has been presented in [12]. This solution requires an $O(n)$ bits overhead in addition to the

*LIAFA, Univ. Paris Diderot - Paris 7. dbelaz@liafa.jussieu.fr

†Dept. of Computer Science, University of Pisa. rossano@di.unipi.it

optimal space, for an arbitrary fpp. Our optimal scheme is the first known solution having $o(n)$ overhead, for any value of ϵ such that $\log \frac{1}{\epsilon} = o(\log n / \log \log n)$, namely, any reasonable choice of ϵ in practice.

Relative membership. This problem asks to solve a further relaxation of the membership query [3]. Given two sets S and R with $S \subset R$, we must be able to distinguish whether a key belongs to S or $R \setminus S$. Our compressed static functions are used to obtain a constant time solution for the problem achieving an optimal space complexity (up to lower order term).

1 Introduction

The dictionary problem is one of the most fundamental problem in algorithmics that consists in storing and answering queries on a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements. Usually, elements of S are integers from a large universe $U = [2^w]$, where w is the size of a machine word. Each element may have associated satellite data which are integers chosen from an alphabet Σ of size σ. The problem asks to answer two kind of queries: given a key $x \in U$, *membership* has to decide whether x belongs to S and *retrieval* has to return the satellite data associated with x whenever it belongs to S. This paper studies three well-established relaxations of the dictionary problem: *(Compressed) Static functions, Approximate membership* and *Relative membership.*

(Compressed) Static functions. A well-known relaxation of the retrieval query, also known as *retrieval-only dictionaries*, allows to report arbitrary data whenever the searched key x does not belong to S. The problem can be formally stated as follows.

Given a set $S = \{x_1, \ldots, x_n\} \subset U = [2^w]$ of n integers, where w is the size of a machine word. Elements of S have associated data (called symbol*) from an alphabet Σ of size σ. The problem asks to build a dictionary that, given a key $x \in S$, returns its associate symbol. An arbitrary symbol is returned whenever $x \notin S$.*

Essentially, we are defining a static function F whose domain is the set S and whose codomain is formed by the symbols associated with those keys (i.e., $\Sigma = \{F(x_1), \ldots, F(x_n)\}$). The problem asks to evaluate F on its domain with the possibility of returning any symbol for keys in $U \setminus S$.

This problem has been widely studied in the past [6, 8, 1, 16, 12, 11, 5, 7, 14]. Solutions to this problem have to carefully organize associated symbols, so that, they can be retrieved in constant time without the need of storing keys in S. A simple solution solving the problem resorts to a minimal perfect hash function (e.g., see [11]) with the aim of serializing F's symbols in a array of length n which is, then, represented plain. This solution uses $n \log \sigma + O(n)$ bits of space allowing $O(1)$ evaluation time. However, this approach has to pay an ineliminable overhead of at least $n \log e \approx 1.44n$ bits due to the presence of the minimal perfect hash function. The goal of the subsequent solutions is that of reducing this overhead as much as possible. Dietzfelbinger and Pagh [8] provides a solution achieving $(1+\delta)n \log \sigma$ bits of space and $O(1 + \log \frac{1}{\delta})$ query time, for any $\delta > 0$. An alternative instance of their technique achieves $n \log \sigma + O(\log \log n)$ bits of space whp and $O(\log n)$ query time. A subsequent result by Porat [16] is asymptotically superior: it requires $n \log \sigma + o(n)$ bits of space and $O(1)$ query time. [1]

Often the symbols associated with the keys follow a skewed distribution: few symbols are considerably more frequent than others. In these scenarios, it is desirable to achieve space that depends on the entropy of the data rather than on the number of possible symbols. Thus, designing

[1] To be precise, this space complexity is achieved by combining solution of [16] with results in [9] and [12].

compressed static functions asks to represent F with constant evaluation time and by using space close to nH_0 as much as possible, where H_0 denotes the 0-th order empirical entropy of the sequence $F(x_1), F(x_2), \ldots, F(x_n)$. Even in this case, there exists a simple solution that serializes F's symbols in an array via a minimal perfect hash function. Here, the array is represented in a compressed form by using a scheme that guarantees $O(1)$ access time to any of its values (e.g., see [10]). Even if it achieves space bounded in term of the entropy, the ineliminable overhead of at least $n \log e$ bits still persists. This overhead is even more critical in the compressed scenario: it may become the dominant term for highly compressible functions. A recent result by Hreinsson *et al.* [12] proposes a solution with a smaller overhead. Indeed, their solution represents any function F within $(1+\delta)nH_0 + n \min(p_0 + 0.086, 1.82(1-p_0))$ bits of space guaranteeing constant evaluation time of F, where p_0 is the (empirical) probability of the most frequent symbol and δ is a constant greater than 0. This space complexity has two drawbacks: 1) the solution is suboptimal due to the factor $(1+\delta)$ to multiply H_0; and 2) the overhead may become $\Theta(n)$ depending on p_0, even if it has a small constant to multiply n, thus, there may be no asymptotical improvement over the solution based on perfect hashing.

In Section 2 we describe our optimal scheme which is the first known solution obtaining simultaneously constant query time, compressed space (nH_0), and sublinear overhead ($o(n)$). We strongly believe that these characteristics makes the use of static functions significantly more appealing for many applications.

Approximate membership. The problem asks to solve a relaxed membership query and is defined as follows.

Given a set $S = \{x_1, \ldots, x_n\} \subset U = [2^w]$, we want to build an approximate membership data structure AM *with false positive probability ϵ such that:*

- *Given any element $x \in S$, it always returns* true*;*
- *Given any element $x \in U \setminus S$, it returns* false *with probability at least $1 - \epsilon$.*

The approximate membership problem has been studied for decades. The Bloom filter data structure [4] is probably the most popular technique solving this problem. With Bloom filters a space usage of $n \log \frac{1}{\epsilon} \log e$ bits suffices for a false positive probability ϵ. However, its space and time complexities are both non-optimal: space is a constant factor away from optimal and query time is logarithmic in $\frac{1}{\epsilon}$. For achieving constant query time we can resort to a minimal perfect hash function to injectively map the keys of S to the cells of an array of size n. Each cell contains a $\log \frac{1}{\epsilon}$-bit signature of its assigned key. A query is solved by mapping the searched key to a cell and comparing its signature with the one contained in the cell. In this case the space occupancy can be bounded with $n \log \frac{1}{\epsilon} + n \log e + o(n)$ bits. This space usage differs from the best possible by the term $n \log e$ which is ineliminable by resorting to this strategy. Another constant query time solution has been presented in [6] which still has non-optimal space complexity. More recently, some other constant time solutions with almost optimal space have been proposed [5, 8, 15, 16]. In those solutions however the space is optimal (up to lower order term) only when the false positive probability is a negative power of two. Recently, Hreinsson *et al.* [12] achieves a smaller space complexity for an arbitrary false positive probability. However, their space occupancy uses $0.086n$ bits of space more than the best possible.

Section 3 describes the first known approximate membership solution achieving optimal space (up to lower order term) for any false positive probability (not necessarily a power of two).

Relative membership problem. This problem has been introduced in [3] and asks to solve a further relaxation of the membership query. We have two sets S and R with $S \subset R$, given any $x \in U$, the data structure has to establish if x belongs to S or $R \setminus S$. More formally, the problem is defined as follows.

Given two sets of integers S, R such that $S \subset R \subseteq U = [2^w]$, we want to build a data structure such that:

- *Given any element $x \in S$ and $x \notin R$, it always returns* true*;*
- *Given any element $x \in R \setminus S$, it always returns* false*;*
- *Given any element $x \in U \setminus R$, it is allowed to return an arbitrary answer.*

The constant time solution presented in Belazzougui *et al.* [3] uses $\log \binom{m}{n} + O(n)$ bits of space, where $n = |S|$ and $m = |R|$. Being $\Omega(\log \binom{m}{n})$ bits a lower bound for this problem, the space overhead in the solution is $\Omega(n)$ bits.

In Appendix C we use our compressed static functions to obtain a constant time solution achieving an optimal space complexity (up to lower order term).

2 Compressed Static Functions

In this section we address the problem of designing a data structure that stores and retrieves values that have been associated to the keys of a set $S = \{x_1, \ldots, x_n\} \subset U = [2^w]$ of n integers. Values can be seen as symbols drawn from an alphabet Σ of size σ. [2] The data structure is allowed to return an arbitrary symbol whenever the searched key x does not belong to S. This corresponds to the definition of a function F whose domain is the set S and whose codomain is formed by the symbols associated to those keys (i.e., $\Sigma = \{F(x_1), \ldots, F(x_n)\}$). The problem asks to evaluate F on its domain with the possibility of returning any symbol for keys in $U \setminus S$.

The simple known solution that we mentioned in the Introduction resorts to a minimal perfect hash function to map S's keys to the first $[n]$ integers. This mapping can be done with a data structure (see e.g., [11]) requiring (optimal) $n \log e + o(n) + O(\log w)$ bits of space while supporting a constant time evaluation of the hash functions. The symbols associated with elements of S are stored in a contiguous (bit)array in the order induced by the minimal perfect hash function. The overall space complexity of this simple scheme is, thus, $n\lceil \log \sigma \rceil + O(n + \log w)$ bits.

One could expect that in real applications the codomain of F is highly compressible due to the presence of many repetitions of the same symbols. Interestingly, we can exploit the presence of these repetitions to reduce the space usage up to achieving, depending on the data, even space sublinear in the number of bits required to represent F's codomain plain (i.e., $n \log \sigma$ bits). To clarify this aspect, it is convenient to introduce the definition of the 0th order empirical entropy of a sequence of symbols. Let T be a text obtained by concatenating, in arbitrary order, the symbols assigned by F (i.e., $\{F(x_1), \ldots, F(x_n)\}$). The zero-th order *empirical entropy* of T is defined as

$$nH_0(T) = \sum_{\alpha \in \Sigma} n_\alpha \log \frac{n}{n_\alpha},$$

[2] In the paper we will assume that $\sigma \leq n$.

where n_α is the number of occurrences of symbol α in T and it is assumed that all logarithms are taken to the base 2 and $0 \log 0 = 0$. It is well known that H_0 is the maximum compression one can achieve using a uniquely decodable code in which a fixed codeword is assigned to each alphabet symbol. In particular, the so-called *zero-th order statistical compressors* (such as Huffman or Arithmetic) achieve an output size which is very close to this bound. Notice that, by definition, H_0 is independent on the order of symbols in T.

A simple known approach to represent F in compressed space uses three main components: minimal perfect hash functions, Huffman coding and prefix sum data structures. We use a minimal perfect hash function to map S's keys to the first $[n]$ integers. This mapping can be done with a data structure (see e.g., [11]) requiring (optimal) $n \log e + o(n) + O(\log w)$ bits of space. The symbols associated with elements of S are encoded using Huffman coding and stored in a contiguous (bit)array in the order induced by the minimal perfect hash function. It is well-known that the Huffman coding achieves redundancy of at most 1 bit per stored symbol which overall adds up to at most n bits of extra space. Finally, we store the starting position of the encoding of each Huffman encoded symbol using a prefix sum data structure which occupies at most $n(\lceil \log(H_0+1) \rceil + 2)$ bits of space. Notice that we also require a data structure to decode each codeword in constant time. For this aim, a table requiring $O(\sigma \log \sigma)$ bits of space suffices. [3]

Given a query for an element $x \in S$, we first evaluate the perfect hash function on element x. Then, we probe the prefix sum data structure in order to get the position of the Huffman encoded symbol which is finally decoded. We observe that this evaluation algorithm takes constant time.

The following Lemma follows from this discussion.

Lemma 1 *A static function F defined over a subset $S = \{x_1, \ldots, x_n\} \subseteq U = [2^w]$ into symbols drawn from an alphabet Σ of size σ can be represented in $nH_0 + n \log(H_0+1) + O(n+\sigma+\log w)$ bits of space so that computing $F(x)$ for any $x \in S$ takes constant time, where H_0 denotes the empirical zero-th order entropy of the sequence $F(x_1), \ldots, F(x_n)$.*

The main advantage of this scheme is that, for large H_0, the overhead in its space complexity is relatively small when compared to H_0. In particular the overhead is sublinear whenever H_0 is superconstant. However, its space occupancy is quite far from being optimal for small values of H_0. In particular, for constant H_0 the used space is a constant factor away from optimal while for smaller entropy the overhead becomes dominant. Moreover, we observe that the scheme does not achieve any compression whenever the alphabet is binary. Indeed, its $O(n)$ additive term results in being always dominant. As we will see in the subsequent sections, the binary alphabet case is very interesting due to its important applications. We also observe that no approach based on minimal perfect hashing could expect to reduce this overhead due to the $n \log e$ bits lower bound on their space occupancy [13].

In this section we will prove that achieving sub linear overhead for static functions is possible. Our main result states that the overhead for compressed functions is $o(n)$ bits worst case for all but very large alphabets (namely, $\log \sigma = \Omega(\log n / \log \log n)$). Indeed, the lower order term is $o(n)$ as long as $H_0 = o(\frac{\log n}{\log \log n})$, which is always the case whenever $\log \sigma = o(\frac{\log n}{\log \log n})$ being $H_0 \leq \log \sigma$. Formally, we obtain the following theorem

[3] As minor technical detail, we notice that $O(\sigma \log \sigma)$ may be $\Omega(n)$. In this case we replace Huffman encoding with Hu-Tucker encoding. The latter requires smaller table of size $O(\sigma)$ bits at the cost of increasing the redundancy from 1 to 2 bits per symbol.

Theorem 1 *A static function F defined over a subset $S = \{x_1, \ldots, x_n\} \subseteq U = [2^w]$ into symbols drawn from an alphabet Σ of size σ can be represented in $nH_0 + O(\frac{n(H_0+\log\log n)\log\log n}{\log n} + \sigma + \log w)$ bits of space so that computing $F(x)$ for any $x \in S$ takes constant time. H_0 denotes the empirical zero-th order entropy of the sequence $F(x_1), \ldots, F(x_n)$. The scheme can be built in $O(n)$ expected time.*

The proof of this theorem is divided in the next two subsections. In Subsection 2.1 we start by presenting proving the result for alphabets of size $O(\log^2 n)$. Then, in Subsection 2.2, we combine this idea with a proper alphabet partitioning step in order to deal also with general alphabet sizes. In Subsection 2.2 we also show how to further reduce the lower order term in Theorem 1 so that the space complexity vanishes as the underlying function is more compressible.

2.1 Compressed static functions for $\sigma = O(\log^2 n)$

In this subsection we prove the main theorem for alphabets of size $\sigma = O(\log^2 \sigma)$. In the following we will use f_α to denote the frequency of symbol $\alpha \in \Sigma$.

In our discussion we assume that any symbol $\alpha \in \Sigma$ is sufficiently frequent (namely, $f_\alpha > \frac{1}{\log^4 n}$). This assumption is not problematic. We can use a simple scheme to manage rare symbols requiring constant time and $O(n/\log n)$ bits of space. We say that a symbol α is *rare* if and only if $f_\alpha \leq \frac{1}{\log^4 n}$). Let $S' \subset S$ be the subset of keys associated with these rare symbols. Notice that

$$|S'| = \sum_{\alpha \in \Sigma, f_\alpha < 1/\log^4 n} n \cdot f_\alpha \leq \log^2 n \cdot \frac{n}{\log^4 n} = O(n/\log^2 n)$$

Being rare symbols few, we are allowed to represent them with a simple and explicit scheme. First, we use a universal hash function h that injectively maps keys of S to $[n^2]$. It is well know that such a function can be found in linear expected time and its representation requires $O(\log w + \log n)$ bits. Then, we use a minimal perfect hash function m to assign an unique identifier to keys in S'. Finally, for each $x \in S'$, we store explicitly the pair $\langle h(x), F(x)\rangle$ in the $m(x)$-th entry of an array with $|S'|$ entries. Notice that such entry requires $O(\log n)$ bits: $O(\log n)$ bits for storing $h(x)$ and $O(\log\log n)$ bits for storing $F(x)$. This simple data structure allows us to distinguish in $O(1)$ time between keys that belong to S' and keys in $S \setminus S'$. For keys in S' we immediately know their associated symbols. The overall space required by this data structure is $O(n \cdot |S'| \log n + \log w) = O(\frac{n \log\log n}{\log n} + \log w)$ bits.

After having addressed the extreme case of too rare symbols we can present our scheme by safely assuming that any f_α is at least $\frac{1}{\log^4 n}$. Crucial for our scheme is the intial step that partitions the keys of S into $m = \Theta(\frac{n \log\log n}{\log n})$ buckets containing at most $b = c_0 \frac{\log n}{\log\log n}$ keys each, where c_0 is an arbitrary constant smaller than 1. The keys of the same bucket are injectively mapped to the range $[\log^{c_1} n]$, for some constant $c_1 > 0$. Given $x \in S$, we use $G(x)$ and $Q(x)$ to denote respectively the bucket in which x has been mapped and its offset within this bucket. We resort to a partitioning strategy having these characteristics as stated by the following lemma (whose proof is reported in Appendix A).

Lemma 2 *Given a set S of n keys each of length w bits, and fixed any parameter b such that $b = O(\log^{\Theta(1)} n)$, and any range R such that $R = [\log^{\Theta(1)} n)]$, we can build in linear time a data structure that partitions the set S into $m = \Theta(n/b)$ buckets. There exists a data structure consisting of two functions G and Q that has the following properties:*

- *The number of keys mapped by G to the same bucket B_i is $O(b)$;*
- *The keys of the same bucket are injectively mapped by Q to the range R;*
- *The data structure is described by using $O(n\frac{\log\log n}{b} + \log w)$ bits of space;*
- *Both G and Q can be evaluated in constant time.*

For our problem, each bucket B_i has its own configuration of offsets and symbols. This configuration can be seen as a tuple $\langle(o_1, s_1), \ldots, (o_k, s_k)\rangle$ with $k \leq c_0\frac{\log n}{\log\log n}$, $o_j < o_{j+1}$, $o_k \leq \log^{c_1} n$, and $s_j \in \Sigma$. Essentially, we have the pair (o_j, s_j) if and only if there exists a key x in S such that $G(x) = i$, $Q(x) = o_j$ and $F(x) = s_j$.

Our objective is to encode all bucket configurations by using space close to nH_0 bits. The idea is to use a (fully) randomly generated table T that has $n^{3\epsilon}$ rows and $\log^{c_1} n$ columns, with ϵ a parameter smaller than $1/3$ that will be set later. Each cell of T is randomly set equal to any symbol $\alpha \in \Sigma$ with probability f_α ($\geq \frac{1}{\log^4 n}$). The *table index* r of bucket B_i is the index of the first row of table T whose cells match the configuration of B_i (namely, for any $j \leq k$, we must have that $T[r, o_j]$ is equal to s_j). We obtain a binary vector V by concatenating, in order, the table indexes of all the buckets. Each table index is encoded in binary by using $\lceil \log r \rceil$ bits, where r is the index of the row that matches the bucket. We augment this representation with a standard prefix sum data structure which stores the starting position of each bucket by using overall $O(\frac{n(1+H_0)\log\log n}{\log n})$ bits of additional space.

The constant time evaluation of $F(x)$ is as follows. Given the key x, we first determine its bucket $G(x)$ and its offset $Q(x)$. Then, we use the prefix sum data structure to retrieve the table index r of the bucket $B_{G(x)}$. By construction, $Q(x)$-th cell of r-th row of table T is the value of $F(x)$. Obviously, an arbitrary value is returned whenever x does not belong to S.

Space/Time analysis. It remains to show that the length of the binary vector V can be bounded in term of nH_0 if we use suitable tables T which are sufficiently easy to be found. We evaluate the number of possible bucket configurations. Recall that a bucket configuration is a tuple $\langle(o_1, s_1), \ldots, (o_k, s_k)\rangle$, where each o_i is an offset in R and $s_i \in [\Sigma]$ is a symbol. Thus, we can have at most $|R| \cdot \sigma$ different values for each pair (o_i, s_i) and consequently at most $(\log^{c_1} n \cdot \sigma)^b = (\log^{c_1} n \cdot \log^2 n)^{c_0\frac{\log n}{\log\log n}} = 2^{(c_1+2)c_0 \log n}$ different bucket configurations. Therefore, in order to conclude that the total number of possible bucket configurations is at most n^ϵ, it suffices to choose c_0 and c_1 so that $(c_1 + 2)c_0 < \epsilon$, for any constant $\epsilon < 1$.

As said above, the table T is built by randomly generating $n^{3\epsilon}$ rows with $\log^{c_1} n$ symbols each. For each bucket B_i, we identify the first row that matches its configuration as explained above. If at least one bucket configuration fails in finding its match among the $n^{3\epsilon}$ rows of T, we generate a new random table T and we start everything from scratch. Now we show that the probability that this happens is at most $\frac{1}{n^{2\epsilon}}$. We denote with P_i the probability that a generic row of T matches B_i. A row of T matches the configuration $\langle(o_1, s_1), \ldots, (o_k, s_k)\rangle$ of bucket B_i if and only if the row contains the symbol s_j at position o_j, for every $j \in [k]$. Being f_{s_j} the probability that the row contains the symbol s_j in position o_j, we conclude that $P_i = \prod_{1\leq j<k} f_{s_j}$.

Let the random variable X_i be the position of this row in the T. Observe that X_i follows a geometric distribution with P_i as probability of success. Since we are assuming that the frequencies f_{s_j} are larger that $\frac{1}{\log^4 n}$, it follows that $P_i > (\frac{1}{\log^4 n})^k = 2^{-4c_0 \log n}$. Thus, it suffices to set $c_0 < \frac{\epsilon}{4}$ to ensure that $P_i > \frac{1}{n^\epsilon}$. By Markov's inequality, the probability that X_i is larger than $n^{3\epsilon}$ (i.e.,

$n^{2\epsilon}$ times more than the maximum expected number of trials n^ϵ) is $1/n^{2\epsilon}$. By the union bound, a failure over the (at most) n^ϵ configurations occurs with probability at most $1/n^\epsilon$. Thus, the expected number of attempts needed to find a suitable table T for all the configurations is upper bounded by $O(n^\epsilon)$.

If each bucket configuration has found its match in T, we check that the size of V is good enough. More precisely, we check that

$$|V| = \sum_{1\le i\le m} \lceil \log X_i \rceil \le \sum_{1\le i\le m} \log X_i + m \le nH_0\left(1+\frac{1}{\log^3 n}\right) + m.$$

First, we observe that the expected size of V can be bounded in term of 0-th order entropy. Indeed,

$$\begin{aligned} E[|V|] = & \; E[\textstyle\sum_{1\le i\le m}\lceil \log X_i\rceil] \\ \le & \; E[\textstyle\sum_{1\le i\le m}\log X_i] + m \\ = & \; \textstyle\sum_{1\le i\le m} E[\log X_i] + m \\ \le & \; \textstyle\sum_{1\le i\le m} \log E[X_i] + m \\ = & \; nH_0 + m \end{aligned}$$

where the last inequality follows by Jensen's inequality (since logarithm is concave).

To conclude, we use Markov's inequality to bound the probability that $|V|$ is larger than the above bound by a factor $1+\frac{1}{\log^3 n}$. Indeed, by Markov's inequality the probability that $|V| > E[|V|](1+\frac{1}{\log^3 n})$ is at most $1/(1+\frac{1}{\log^3 n})$. This means that $|V| \le nH_0(1+\frac{1}{\log^3 n})$ with probability $\Omega(\frac{1}{\log^3 n})$. If table T is not good enough for achieving that space bound, we retry everything from scratch by using a new randomly generated table. Notice that a suitable table is found with $O(\log^3 n)$ expected trials. Since building and checking the size of V induced by each table requires $O(n^{4\epsilon}) = o(n)$ time (by choosing $\epsilon < \frac{1}{4}$), a suitable table can be found in $O(n^{4\epsilon}\log^3 n)$ expected time.

2.2 Compressed static functions for general alphabets

There is only one issue that prevent us from using directly the scheme of the previous section to prove Theorem 1 also for larger alphabet sizes. The problem is due to the fact that the number of possible bucket configurations is at least $2^{b\cdot\log\sigma}$, where b is the maximum number of keys in a bucket. Since it is required this quantity to be $O(n^\epsilon)$, larger is the alphabet size, the smaller we have to choose the value b (namely, we have to choose $b = \epsilon\frac{\log n}{\log\sigma}$ to match the required condition). However, partitioning accordingly to Lemma 2 requires $O(\frac{n\log\log n}{b}) = O(\frac{n\log\sigma\log\log n}{\log n})$ bits for obtaining buckets of size at most b. By plugging these arguments into our solution, we obtain a scheme that requires $nH_0 + O(\frac{n\log\sigma\log\log n}{\log n}) + O(\sigma+\log w)$ bits of space. Thus, the lower order term of this approach is larger than the bound claimed in Theorem 1 for large alphabets (namely, $\log\sigma = \omega(\log\log n)$). Thus, the best we can achieve in this way is the following weaker lemma which, nevertheless, will be useful later.

Lemma 3 *A static function F defined over a subset $S = \{x_1,\ldots,x_n\} \subseteq U = [2^w]$ into symbols drawn from an alphabet Σ of size σ can be represented in $nH_0 + O(\frac{n(\log\sigma+\log\log n)\log\log n}{\log n} + \sigma + \log w)$ bits of space so that computing $F(x)$ for any $x \in S$ takes constant time. H_0 denotes the empirical zero-th order entropy of the sequence $F(x_1),\ldots,F(x_n)$. The scheme can be built in $O(n)$ expected time.*

We improve this lemma by appropriately dividing the original function F into several static functions F_i. Each of these functions F_i is responsible only for keys associated with a portion of the symbols in Σ having approximately the same frequencies. We resort to a global function F_g over an alphabet of size $O(\log^2 n)$ which is used as a dispatcher to choose the function F_i which is responsible for the queried key. More precisely, we divide keys of S in groups by using the alphabet partitioning strategy presented in [2]. We start by sorting symbols by their frequencies and we form at most $\log^2 n$ groups of symbols. The group G_i is formed by the symbols with frequencies in the range $[2^{-(i/\log n)}, 2^{-(i+1)/\log n})$, for i an integral in $[0, \log^2 / \log\log n - 1]$. Each group has assigned an unique identifier in $[\log^2]$. Let σ_i denote the number of distinct symbols in group G_i, each symbol of group G_i has assigned an unique identifier in $[\sigma_i]$.

Function F_g is obtained from F by replacing each symbol with its group identifier. We have a function F_i for each group G_i which is defined only for the keys S which are associated with symbols in G_i. Thus, F_i is obtained from F by replacing occurrences of symbols in G_i with their identifiers. In order to achieve space bounded by $nH_0 + O(n/\log n)$ bits, it is enough to represent only F_g in compressed form, namely, by using close to nH'_0 bits, where H'_0 denotes the empirical zero-th order entropy of the sequence $F_g(x_1), \ldots, F_g(x_n)$. Indeed, since symbols represented by the same function F_i have almost the same frequencies, the use of succinct representations for F_is suffices, i.e., each F_i can be represented by using $\log \sigma_i$ bits per key. For a formal proof of this intuitive fact we refer to [2]. Our scheme resorts to different representations for the above functions depending on the frequencies and the number of distinct symbols.

- Function F_g is represented by resorting to the scheme described in the previous subsection. We notice, indeed, that F_g is defined over an alphabet of size at most $\log^2 n$. The space required by this representation is $nH'_0 + (\frac{n(H'_0+\log\log n)\log\log n}{\log n})$ bits of space, where H'_0 denotes the empirical zero-th order entropy of the sequence $F_g(x_1), \ldots, F_g(x_n)$.

- Function F_i for a group G_i where the sum of the frequencies of its symbols is at most $\frac{1}{\log^4 n}$ is represented with a dedicated plain dictionary as we have done for rare symbols in Subsection 2.1. Notice that this dictionary stores at most $O(n' \log n / \log^4 n)$ keys, where n' is the size of the domain of F_i, and, thus, uses negligeble space.

- Function F_i with $i \geq \log^2 n / \log\log n$ is represented with a simple scheme that costs no more than $\log \sigma_i + \Theta(1)$ bits per symbol. The scheme is obtained by using a minimal perfect hash function and by writing an array with its symbols. The overhead of $\Theta(1)$ bits is within the lower order term of Theorem 1. Indeed, the contribution of infrequent symbols to F's entropy is at least $\Theta(\frac{\log n}{\log\log n})$ bits.

- Function F_i with $i \leq \log n \log\log n$ is represented with the scheme of the previous subsection. Indeed, the number of distinct symbols having frequencies in $[2^{-(i/\log n)}, 2^{-(i+1)/\log n})$ cannot be more than $\log^2 n$.

- Function F_i with $\log n \log\log n < i < \log^2 n / \log\log n$ is represented by the scheme of Lemma 3. The space complexity is, thus, $n' \log \sigma_i + O(\frac{n' \log \sigma_i \log\log n'}{\log n'}) + O(\sigma_i + \log w)$ bits, where n' is the size of the domain of F_i. We observe that even in this case the lower order term is within the lower order term of Theorem 1. Indeed, with these choices of i we have that $\log \sigma_i \leq \log 2^{(i+1)/\log n} = (i+1)/\log n$ being $2^{(i+1)/\log n}$ the maximum number of symbols

having their frequencies in $[2^{-(i/\log n)}, 2^{-(i+1)/\log n})$. Notice that the contribution of these symbols to F's entropy is at least i bits.

The space bound claimed in Theorem 1 easily follows by summing up the above space complexities. The query algorithm is as follows. Given x, we query $F_g(x)$ to identify the group of the symbol associated with key x. Then, we query the function with index $F_g(x)$ which is responsible for this group (i.e., we query $F_{F_g(x)}(x)$). This concludes the proof of Theorem 1.

The main drawback of Theorem 1 is that its space occupancy does not vanish when H_0 tends to zero. Following an idea in [12], it is possible to overcome this limitation by carefully combining the result of Theorem 1 and an approximate membership data structure. The following theorem summarizes the obtained result. Due to space limitations, its proof is deferred to Appendix B.

Theorem 2 *A static function F defined over a subset $S = \{x_1, \ldots, x_n\} \subseteq U = [2^w]$ into symbols drawn from an alphabet Σ of size σ can be represented in $nH_0 + O(\frac{nH_0 \log\log(nH_0)}{\sqrt{\log(nH_0)}}) + O(\sigma + \log w)$ bits of space so that computing $F(x)$ for any $x \in S$ takes constant time, where H_0 denotes the empirical zero-th order entropy of the sequence $F(x_1), \ldots, F(x_n)$. The scheme can be built in $O(n)$ expected time.*

3 Optimal solution for the Approximate membership problem

As reported in the introduction (constant time) approximate membership data structures (AM) are able to achieve optimal $n \log \frac{1}{\epsilon} + o(n)$ bits of space only if the false positive probability (fpp) ϵ is a power of two [5, 8, 15, 16]. The current best solution for general fpp has been presented in [12]. This solution requires an overhead of $0.086 \cdot n$ bits over the optimal space, for an arbitrary fpp. In this section we show a solution whose overhead is reduced to $o(n)$ bits, for values of ϵ such that $\log \frac{1}{\epsilon} = o(\log n / \log\log n)$. More formally we prove the following theorem.

Theorem 3 *There exists an approximate membership data structure that, given a set $S \subset U = [2^w]$ of n elements, has $O(1)$ query time and requires $n \log \frac{1}{\epsilon} + O(\frac{n(\log\log n + \log \frac{1}{\epsilon}) \log\log n}{\log n} + \log w)$ bits of space, for any false positive probability ϵ. The scheme can be built in $O(n)$ expected time.*

We start by showing a space optimal solution for the simpler case in which ϵ is any constant in $(1/2, 1)$. More precisely, we prove the following lemma.

Lemma 4 *There exists an approximate membership data structure that, given a set $S \subset U = [2^w]$ of n elements, has $O(1)$ query time and requires $n \log \frac{1}{\epsilon} + O(\frac{n(\log\log n)^2}{\log n} + \log w)$ bits of space, for any false positive probability $\epsilon \in (1/2, 1)$. The scheme can be built in $O(n)$ expected time.*

The idea is to use an approach similar to the one in the previous section. In this context, we want to encode only two values, 0 and 1, whose probabilities are respectively $1 - \epsilon$ and ϵ. We have that the elements of S have assigned symbol 1, while elements in $U \setminus S$ have assigned symbol 1 with probability at most ϵ and symbol 0 with probability at least $1 - \epsilon$. In this case we partition the keys of S into $\Theta(n/b)$ buckets of size at most $b = c_0 \frac{\log n}{\log\log n}$. The offset of each key inside its bucket is assigned with a global pairwise independent hash function Q that maps a key to a value in $K = [\log^{c_1} n]$. We notice that for this problem collisions induced by Q among keys of S are not

problematic: keys in S are always associated with symbol 1. By this construction, there are at most $(\log^{c_1} n)^{c_0 \frac{\log n}{\log\log n}} = 2^{c_0 c_1 \log n} = n^{c_0 c_1}$ distinct sets of size $b = c_0 \frac{\log n}{\log\log n}$ out of universe of size $k = |K| = \log^{c_1} n$, with $c_0 < 1/2$ and $c_1 \geq 2$ chosen so that $c_0 c_1 < 1$.

As in the previous section, we randomly generate a binary table T with k columns and $n^{c_0 c_1} = o(n)$ rows. In this case, we force each row to have exactly $k_1 = \lfloor \epsilon k \rfloor - b$ cells set to 1 and exactly $k_0 = k - k_1$ cells set to 0. A bucket is encoded by writing the index of the first row of T that matches its configuration (namely, all the cells corresponding to the keys of S in the bucket must be set to 1). We have that the probability p that a cell matches a bucket with $t \leq b$ keys of S (i.e., all its t keys are mapped to a 1) is at least $(\frac{k_1}{k}(1 - \frac{t}{k_1}))^t$.

Thus, the expected size of the encoding of the bucket is at most $\lfloor \log(1/p) \rfloor \leq \log(1/p) + 1$ bits. We have

$$\begin{aligned}
\log(1/p) + 1 = &\ t \log(\tfrac{k}{k_1}(\tfrac{1}{1-\frac{t}{k_1}})) + 1 \\
= &\ t(\log \tfrac{k}{k_1} - \log(1 - \tfrac{t}{k_1})) + 1 \\
= &\ t(\log(\tfrac{1}{\epsilon}(1 + O(\tfrac{t}{k}))) - \log(1 - \tfrac{t}{k_1})) + 1 \\
\leq &\ t \log \tfrac{1}{\epsilon} + O(\tfrac{b^2}{k}) + 1
\end{aligned}$$

where we use $\frac{k}{k_1} = \frac{1}{\epsilon}(1 + O(\frac{b}{k}))$ and $t \leq b$, and the fact that $|\log(1+x)| = \Theta(x)$ when $|x| < 1$. Now we know that $k = \log^{c_1} n$ and that $b < \log n$ which means that the final size of the encoding is $t \log \frac{1}{\epsilon} + O(\log^{2-c_1} n + 1)$. By setting $c_1 \geq 2$, we ensure that the additive term $O(\log^{2-c_1} n + 1)$ remains $O(1)$. By summing up the encoding sizes for all the $\Theta(n/b)$ buckets, we obtain the space usage of $n \log \frac{1}{\epsilon} + O(\frac{n(\log\log n)^2}{\log n})$ bits.

Finally, we analyze the false positive probability of our scheme. We consider a key $x \notin S$ and we assume that x has been mapped to a certain bucket with $t \leq b$ keys of S on it. We want to bound the probability that the scheme erroneously reports that x belongs to S. This could happen in two cases: 1) x collides with a key of S in its bucket; 2) x has been mapped to a cell with a 1 in the row of T that matches its bucket configuration. The probability of case 1 is at most t/k. Indeed, by the pairwise independence of the function Q, we know that x collides with each of the t keys with probability at most $1/k$. Thus, x collides with any of the t keys with probability at most t/k (notice that those t keys themselves could collide and be mapped to only $t' \leq t$ distinct positions without affecting the false positive probability of the scheme). In case 2, since x does not collide with any of the t keys, it is mapped to one of the other $k - t'$ possible cells. We have a false positive iff the cell has been set to 1. By construction of the row, $k_1 - t'$ cells are set to 1 and $k - k_1$ cells are set to 0. Therefore, the global probability of a false positive for x is at most

$$\begin{aligned}
\tfrac{t}{k} + \tfrac{k_1 - t'}{k - t'} \leq &\ \tfrac{t}{k} + \tfrac{k_1}{k} \\
= &\ \tfrac{t + \lfloor \epsilon k \rfloor - b}{k} \leq \epsilon
\end{aligned}$$

This concludes the proof of Lemma 4.

We obtain a scheme for the general case with $\epsilon \leq 1/2$ by combining Lemma 4 with a compressed function of Theorem 1. We first observe that the false positive probability ϵ can be seen as $\epsilon = c2^{-i}$, with $c \in (1/2, 1)$ and i a positive integer. We store the keys in a first AM implemented according to Lemma 4 with fpp c. We also use a pairwise independent hash function Q to map keys of S to signatures in $[2^i]$ (i.e., of length i bits) which are stored as their associated symbols in the compressed function. A query is solved by first querying the AM and, if the key passes the filter, by comparing its signature with the one returned by the compressed function.

The false positive probability of this second scheme is bounded as follows. Consider a key $x \notin S$. The filter is passed with probability at most c. The signature returned by the compressed function could be of two types: it is the signature of a key, say y, in S or it is a completely random symbol stored in a cell of the table used in our solution to represent the function (see Section 2). In both cases the probability that the two signatures are equal is $1/2^i$. [4]

References

[1] Martin Aumüller, Martin Dietzfelbinger, and Michael Rink. Experimental variations of a theoretically good retrieval data structure. In *ESA*, pages 742–751, 2009.

[2] Jérémy Barbay, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Alphabet partitioning for compressed rank/select and applications. In *ISAAC (2)*, pages 315–326, 2010.

[3] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $o(1)$ accesses. In *SODA*, pages 785–794, 2009.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[5] Denis Charles and Kumar Chellapilla. Bloomier filters: A second look. In *ESA*, 2008.

[6] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA*, pages 30–39, 2004.

[7] Erik D. Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pătraşcu. De dictionariis dynamicis pauco spatio utentibus. In *Proc. 7th Latin American Theoretical Informatics (LATIN)*, pages 349–361, 2006. See also arXiv:cs/0512081.

[8] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership. In *ICALP (1)*, pages 385–396, 2008.

[9] Yevgeniy Dodis, Mihai Pătraşcu, and Mikkel Thorup. Changing base without losing space. In *STOC*, pages 593–602, 2010.

[10] Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. *Theor. Comput. Sci.*, 372(1):115–121, 2007.

[11] Torben Hagerup and Torsten Tholey. Efficient minimal perfect hashing in nearly minimal space. In *STACS*, pages 317–326, 2001.

[12] Jóhannes B. Hreinsson, Morten Krøyer, and Rasmus Pagh. Storing a compressed function with constant time access. In *ESA*, pages 730–741, 2009.

[13] Kurt Mehlhorn. Data structures and algorithms, volume 1. sorting and searching. *Springer-Verlag Berlin*, 263:264, 1984.

[4] In the first case, by pairwise independence of Q, the probability that $Q(x) = Q(y)$ when $x \neq y$ is precisely $1/2^i$.

[14] Christian Worm Mortensen, Rasmus Pagh, and Mihai Pătraşcu. On dynamic range reporting in one dimension. In *Proc. 37th ACM Symposium on Theory of Computing (STOC)*, pages 104–111, 2005. See also arXiv:cs/0502032.

[15] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal Bloom filter replacement. In *SODA*, pages 823–829. ACM Press, 2005.

[16] Ely Porat. An optimal bloom filter replacement based on matrix solving. In *CSR*, pages 263–273, 2009.

[17] Dan E. Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, December 1999.

A Proof of Lemma 2

Lemma 2. *Given a set S of n keys each of length w bits, and fixed any parameter b such that $b = O(\log^{\Theta(1)} n)$, and any range R such that $R = [\log^{\Theta(1)} n)]$, we can build in linear time a data structure that partitions the set S into $m = \Theta(n/b)$ buckets. There exists a data structure consisting of two functions G and Q that has the following properties:*

- *The number of keys mapped by G to the same bucket B_i is $O(b)$;*
- *The keys of the same bucket are injectively mapped by Q to the range R;*
- *The data structure is described by using $O(n\frac{\log\log n}{b} + \log w)$ bits of space;*
- *Both G and Q can be evaluated in constant time.*

The partitioning strategy we use to obtain this lemma is a variation of the one by Hagerup and Thoely [11] which proves a similar bound with the exception that b must be $\Theta(\log^{\Theta(1)} n)$. We modify their partitioning in the following way. First, we use their partitioning by choosing buckets of size b' in $\Theta(\log^{\Theta(1)} n)$ which is multiple of our target b. Their partitioning gives us $m' = \Theta(n/b')$ buckets with at most b' keys each. Keys in each bucket are injectively mapped to $R = [\log^{\Theta(1)} n)]$. Then, we process each of the above buckets separately dividing them into subbuckets of size b each (possible with the exception of the last one). In this way we obtain overall $\Theta(n/b')$ subbuckets which is our final partition. We use an atomic heap [17] for each bucket to store the first key of each of its subbuckets. The atomic heap is used to solve predecessor queries in constant time (notice that the number of keys to manage in each atomic heap is polylogarithmic). At query time, the atomic heap of a bucket is used to understand which subbucket contains the queried key $x \in S$. The space required by these fusion trees is $\Theta(n \log\log n/b)$ bits as required.

B Proof of Theorem 2

As we already observed, a drawback of Theorem 1 is that the space occupancy does not vanish when H_0 tends to zero. Following an idea in [12], it is possible to overcome this limitation by carefully combining the solution of Theorem 1 and the approximate membership data structure of Theorem 3 in order to prove the following theorem.

Theorem 2. *A static function F defined over a subset $S = \{x_1, \ldots, x_n\} \subseteq U = [2^w]$ into symbols drawn from an alphabet Σ of size σ can be represented in $nH_0 + O(\frac{nH_0 \log\log(nH_0)}{\sqrt{\log(nH_0)}}) + O(\sigma + \log w)$ bits of space so that computing $F(x)$ for any $x \in S$ takes constant time, where H_0 denotes the empirical zero-th order entropy of the sequence $F(x_1), \ldots, F(x_n)$. The scheme can be built in $O(n)$ expected time.*

The idea is that of using an AM as (pre-)filter for the compressed function whenever the frequency f of the most frequent symbol α is sufficiently high (namely, $f > 1 - \frac{\log\log n}{\sqrt{\log(n)}}$). In this case, indeed, the task of the AM is that of individuating most of α's occurrences. A small compressed function is used to disambiguate the remaining uncovered keys in S.

If such a frequent symbol does not exist, we simply use the solution of Theorem 1. This suffices since the bound of Theorem 1 already matches the bound of Theorem 2, if all symbols

have frequency at most $1 - \frac{\log\log n}{\sqrt{\log(n)}}$. Indeed, in this case $H_0 = \Omega(\frac{(\log\log n)^2}{\sqrt{\log(nH_0)}})$ and Theorem 1 has an overhead of $O(n\frac{(\log\log n)^2}{\log n}) = O(\frac{nH_0}{\sqrt{\log(nH_0)}})$ bits.

Thus, in the following we will assume that there is a symbol α which has frequency at least $f > 1 - \frac{\log\log n}{\sqrt{\log n}}$. Let S_α with $|S_\alpha| = nf$ denote the set of keys mapped to symbol α and let $q = \frac{\sqrt{\log(nH_0)}}{\log\log(nH_0)}$. As filter we use an AM with false positive probability equal to $(1-f)q/f$ built on the set $S_{\bar{\alpha}} = S \setminus S_\alpha$ of keys mapped to a symbol different from α. For the AM we use the solution of Theorem 3. Any key that passes the filter has automatically assigned the symbol α since we know that it does not belong to $S_{\bar{\alpha}}$. However, due to the approximation of the membership query, there are keys in S_α that are erroneously considered in $S_{\bar{\alpha}}$. We use E to denote the set of these keys and we observe that its size is $n(1-f)q$. This wrong assignment is solved by using a compressed function of Theorem 1 on the set $R = E \cup S_{\bar{\alpha}}$, where we associate the symbol α to keys in E and their original symbols to keys in $S_{\bar{\alpha}}$. Notice that the set R has size $n' = |E| + |S_{\bar{\alpha}}| = n(1-f)q + n(1-f) = n(1-f)(1+q)$.

Querying for a key x is simple. We query the AM: If x is considered in $S_{\bar{\alpha}}$, we return the symbol stored in the compressed function, otherwise we return α.

We now analyze the space usage of this scheme.

Let use H_0' to denote the entropy of the symbols stored in the compressed function (i.e., the symbols associated with elements of R). Notice that $n'H_0' \leq nH_0$ as the compressed function stores only part of the original set S and, thus, it must use less space. We consider separately the entropy of symbols in $S_{\bar{\alpha}}$ and in S_α.

The AM stores symbols of $S_{\bar{\alpha}}$ with false positive rate $(1-f)q/f$. Thus, it uses space $|S_{\bar{\alpha}}|\log(\frac{f}{(1-f)q}) + O(\frac{(|S_{\bar{\alpha}}|+1)(\log\log|S_{\bar{\alpha}}|)^2}{\log|S_{\bar{\alpha}}|})$ bits. Now, we know that $|S_{\bar{\alpha}}| \leq nH_0$ which means that this bound is within the lower order term of Theorem 2.

The compressed function uses $n'H_0' + O(\frac{n'(H_0'+\log\log n')\log\log n'}{\log n'}) + O(\sigma + \log w)$ bits of space. We notice that $n' = n(1-f)(1+q) \leq nH_0(1+q) \leq nH_0(1 + \frac{\sqrt{\log(nH_0)}}{\log\log(nH_0)})$ and $n'H_0' \leq nH_0$, to obtain that its lower order term is actually $O(\frac{nH_0\log\log(nH_0)}{\sqrt{\log(nH_0))}})$ bits and, thus, as in the claimed Theorem 2.

We now turn our attention to the dominant terms in the AM and in the compressed function. For that we analyse separately the contributions of elements of $S_{\bar{\alpha}}$ and elements of $E \subset S_\alpha$ to the term $n'H_0'$.

The dominant term in the compressed function is $n'H_0' \leq nH_0$.

We know that the compressed function stores all elements of $S_{\bar{\alpha}}$ but only the part $E \subset S_\alpha$. We thus have

$$nH_0 = |S_\alpha|\log(\frac{n}{|S_\alpha|}) + \sum_{c\in(\Sigma\setminus\{\alpha\})} |S_c|\log(\frac{n}{|S_c|})$$

$$n'H_0' = |E|\log(\frac{n'}{|E|}) + \sum_{c\in(\Sigma\setminus\{\alpha\})} |S_c|\log(\frac{n'}{|S_c|})$$

We know that $n' = n + |E| - |S_\alpha|$ and that $|E| < |S_\alpha|$. We thus deduce that

$$\begin{aligned} |E| \log(\tfrac{n'}{|E|}) &= |E| \log(\tfrac{n+|E|-|S_\alpha|}{|E|}) \\ &= (|S_c| + |E| - |S_c|) \log(\tfrac{n+|E|-|S_\alpha|}{|S_c|+|E|-|S_c|}) \\ &< |S_\alpha| \log(\tfrac{n}{|S_\alpha|}) \end{aligned}$$

(The inequality derives from the fact that the function $(x+a)\log(1+\frac{b}{x+a})$ is monotonically increasing)

Thus, elements of E contribute less to $n'H_0'$ than to nH_0.

It remains to bound the space used for encoding elements of $S_{\bar{\alpha}}$. For an element in $S_c \subset S_{\bar{\alpha}}$ the contribution to nH_0 is

$$\begin{aligned} \log(\tfrac{n'}{|S_c|}) &= \log((\tfrac{(1-f)n}{|S_c|})(\tfrac{n}{(1-f)n})) \\ &= \log(\tfrac{(1-f)n}{|S_c|}) + \log(\tfrac{n}{(1-f)n}) \\ &= \log(\tfrac{(1-f)n}{|S_c|}) + \log(\tfrac{1}{1-f}) \end{aligned}$$

Its contribution to $n'H_0'$ is

$$\begin{aligned} \log(\tfrac{n'}{|S_c|}) &= \log((\tfrac{(1-f)n}{|S_c|})(\tfrac{\quad}{(1-f)n})) \\ &= \log(\tfrac{(1-f)n}{|S_c|}) + \log(\tfrac{n'}{(1-f)n}) \\ &= \log(\tfrac{(1-f)n}{|S_c|}) + \log(\tfrac{n(1-f)(1+q)}{(1-f)n}) \\ &= \log(\tfrac{(1-f)n}{|S_c|}) + \log(1+q) \end{aligned}$$

On the other hand for each element of in $S_{\bar{\alpha}}$, the AM uses space $\log(\frac{f}{(1-f)q})$. Thus, when summing up the space used by the AM and the compressed static function, we have that, for the element of $S_c \subset S_{\bar{\alpha}}$, we use space

$$\begin{aligned} \log(\tfrac{(1-f)n}{|S_c|}) + \log(1+q) + \log(\tfrac{f}{(1-f)q}) &= \log(\tfrac{(1-f)n}{|S_c|}) + \log(1+q) - \log(q) + \log(\tfrac{f}{1-f}) \\ &\leq \log(\tfrac{(1-f)n}{|S_c|}) + \log\tfrac{1}{1-f} + \Theta(\tfrac{1}{q}) \end{aligned}$$

Thus, the space used for an element in $S_{\bar{\alpha}}$ is within $\Theta(\frac{1}{q})$ from optimal. This concludes the proof of Theorem 2.

C Optimal solution for the Relative membership problem

In the Relative membership problem we have two sets S and R such that $S \subset R \subseteq U$ and we want to answer to the following query: given a key $x \in U$, it has to establish whether $x \in S$ or not. The query is allowed to return any answer whenever $x \in U \setminus R$.

It can be proven that the number of bits required by any solution to this problem is $B(m,n) = \log\binom{m}{n} = n\log\frac{m}{n} + (m-n)\log\frac{m}{m-n} - O(\log n)$, where $m = |R|$ and $n = |S|$. A previous result in [3] provides a constant time solution requiring $B(m,n) + O(n + \log w)$ bits of space which is based on a combination of an AM and a static function on binary alphabet. A simple way to obtain an improved result is to directly apply Theorem 2 on a binary function defined on the key R by assigning symbol 1 to keys in S and symbol 0 to keys in $R \setminus S$. A query for a key $x \in U$ is solved by querying the compressed function and by deducing that $x \in S$ if and only if the compressed function returns the value 1.

We derive the following lemma by using Theorem 2 and by observing that $nH_0 = B(m,n)$,

Lemma 5 *There exists a data structure solving in $O(1)$ time the relative membership problem on two sets $S \subset R \subseteq [2^w]$ which requires $B(m,n) + O(\frac{B(m,n)\log\log B(m,n)}{\sqrt{\log(B(m,n))}} + \log w)$ bits of space, where $n = |S|$ and $m = |R|$. The scheme can be built in $O(m)$ expected time.*

By combining this lemma with the result of Belazzougui *et al* [3] we get the following theorem:

Theorem 4 *There exists a data structure solving in $O(1)$ time the relative membership problem on two sets $S \subset R \subseteq [2^w]$ which requires $B(m,n) + O(\min(n, \frac{B(m,n)\log\log B(m,n)}{\sqrt{\log(B(m,n))}}, m-n) + \log w)$ bits of space, where $n = |S|$ and $m = |R|$. The scheme can be built in $O(m)$ expected time.*

The combination is done in the following way: in case $B(n,m) \leq \frac{\sqrt{\log n}}{\log\log n}$ (assuming $n \leq m/2$ [5]) we use Lemma 5 which has overhead at most $O(n)$ bits which is at least as good as the result of Belazzougui *et al* but may become even $o(n)$. Otherwise, we use the result of Belazzougui *et al*.

[5]The case $n > m/2$ is handled in the same way by replacing the set S with its complement $R \setminus S$ and associating 0 instead of 1 with the set $R \setminus S$ and 1 instead of 0 with the set S.