# Compressed Static Functions with Applications*

Djamal Belazzougui† Rossano Venturini‡

**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. In this paper we provide time and space optimal solutions for three well-established relaxations of this basic problem: *(Compressed) Static functions, Approximate membership* and *Relative membership*.

## 1 Introduction

The dictionary problem is one of the most fundamental problem in algorithmics which 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 $\Sigma = [\sigma]$. 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 symbols) from an alphabet $\Sigma$ of size $\sigma$. 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, 17, 12, 11, 5, 7, 14]. Solutions have to carefully organize the 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 unavoidable 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 w.h.p. and $O(\log n)$ query time. A subsequent result by Porat [17] is asymptotically superior: it requires $n \log \sigma(1 + o(1))$ 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 *compressed static functions* asks to represent $F$ with constant evalua-

*This research has been partially funded by MADALGO (Center for Massive Data Algorithmics, funded by the Danish National Research Association), the French ANR-2010-COSI-004 MAPPI project, the Midas EU Project, Grant Agreement no. 318786, the InGeoCloudS EU Project, Grant Agreement no. 297300, the MIUR PRIN, and the FIRB Linguistica 2006.

†LIAFA, Univ. Paris Diderot - Paris 7. Email: dbelaz@liafa.jussieu.fr.

‡Dept. of Computer Science, University of Pisa and ISTI-CNR, Pisa. Email: rossano@di.unipi.it.

[1] To be precise, this space complexity is achieved by combining the solution of [17] with results in [9] and [12].

tion 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 terms of the entropy, the unavoidable 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 $\delta$ 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, and sublinear overhead.

**Approximate membership.** The problem asks to solve a relaxed membership query which 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 $\epsilon$ 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 and 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 $\epsilon$. 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 unavoidable 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, 17]. In those solutions however the space is optimal (up to lower order terms) 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 terms) 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 $m = |R|$. The space overhead in the solution is $\Omega(n)$ bits since $\Omega(\log \binom{m}{n})$ bits a lower bound for this problem.

In Section 4 we use our compressed static functions to obtain a constant time solution achieving an optimal space complexity (up to lower order terms).

## 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 with 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 $\Sigma = [\sigma]$. [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 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 $U \setminus S$.

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 in a plain fashion (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 juxtaposing, 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

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

where $n_\alpha$ is the number of occurrences of symbol $\alpha$ 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 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 2.1. *A static function $F$ defined over a subset $S = \{x_1, \ldots, x_n\} \subseteq U = [2^w]$ into symbols drawn from an alphabet $\Sigma = [\sigma]$ 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 with $H_0$. 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

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

[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.

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

THEOREM 2.1. *A static function $F$ defined over a subset $S = \{x_1, \ldots, x_n\} \subseteq U = [2^w]$ into symbols drawn from an alphabet $\Sigma = [\sigma]$ 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 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 2.1 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 $\sigma = O(\log^2 n)$.

Crucial for our scheme is the initial step that partitions the keys of $S$ into $m = \Theta(\frac{n \log \log n}{\log n})$ buckets containing at most $b = c\frac{\log n}{\log \log n}$ keys each, where $c$ is an arbitrary constant smaller than 1. The keys of the same bucket are injectively mapped to the range $[b^2]$. 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.

LEMMA 2.2. *Given a set $S$ of $n$ keys each of length $w$ bits, and fixed any parameter $b$ such that $b = O(\log^{\Theta(1)} n)$, 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 $[b^2]$;*
- *The data structure is described within $O(n\frac{\log \log n}{b} + \log w)$ bits of space;*
- *Both $G$ and $Q$ can be evaluated in constant time.*

*Proof.* This lemma is obtained by using a simple variant of the partitioning strategy proposed in [11]. This solution partitions the keys in $S$ obtaining $\Theta(\frac{n}{b})$ buckets of size $O(b)$ each, where $b$ could be any value in $\Theta(\log^{\Theta(1)} n)$. The keys in the same bucket are injectively mapped into a range $R$ of size $O(\log^{O(1)} n)$ (see [11] for more details). This construction matches our requests with the exception that $b$ cannot be $o(\log n)$. We modify this solution to deal also with smaller values of $b$. First, we use their partitioning by choosing buckets of size $b'$ in $\Theta(\log^{\Theta(1)} n)$ obtaining $m' = \Theta(n/b')$ buckets with at most $b'$ keys each and with keys in the same bucket injectively mapped to $R = [\log^{\Theta(1)} n]$. Our final partition is obtained by dividing the above buckets into sub-buckets of size $b$ each (possibly with the exception of the last sub-bucket of each bucket). The keys in each of the $\Theta(n/b)$ sub-bucket are injectively mapped to $[b^2]$ by using an universal hash function which is representable by using $O(\log |R| + \log b') = O(\log \log n)$ bits. Finally, we use an atomic heap [18] for each bucket to store the first key of each of its sub-buckets. Since each atomic heap has to manage a polylogarithmic number of keys, predecessor queries are solved in constant time. The space required by these atomic heaps is $\Theta(n \log \log n/b)$ bits. At query time, given a key $x \in S$, we first identify its bucket by using the solution in [11] and, then, we compute its sub-bucket via a predecessor query on the atomic heap of this bucket. ∎

Once we have partitioned the keys into buckets, our goal is to represent space efficiently all these buckets. We start by observing that 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 $k \leq c\frac{\log n}{\log \log n}$, $o_j < o_{j+1}$, $o_k \leq b^2$, 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 at most $n^\epsilon$ rows and $b^2$ columns, with $\epsilon$ a parameter smaller than 1 that will be set later. Generating a suitable table $T$ is a delicate task that will be detailed later. 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. This representation is neither uniquely decodable nor random accessible. We overcome these issues by augmenting the representation with a standard prefix sum data structure which stores the starting position in $V$ of each table index encoding. The cost of this additional data structure is $O(\frac{(\log \log n)^2}{\log n})$ bits.

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 analysis and efficient table generation.** In the remaining of this section we prove that the length of the binary vector $V$ can be bounded in term of $nH_0$ by resorting to an appropriately chosen table $T$. The steps of our analysis are the following. First, we show that the number of possible bucket configurations is $n^\epsilon$, where $\epsilon$ is a parameter smaller than 1. Then, we prove that the size of $|V|$ can be bounded in terms of $nH_0$ for a sufficiently high portion of the randomly generated tables $T$ having $b^2$ columns but an unbounded number rows. Finally, we show that the process of random generation of tables can be driven so that a suitable table with $n^\epsilon$ rows can be found in expected linear time.

**Number of bucket configurations.** We 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 $[b^2]$ and $s_i \in \Sigma$ is a symbol. Thus, we have at most $b^2 \cdot \sigma$ different values for each pair $(o_i, s_i)$ and, consequently, at most $(b^2 \cdot \sigma)^b = (b^2 \cdot \log^2 n)^{c \frac{\log n}{\log \log n}} \leq 2^{4 \cdot c \log n}$ different bucket configurations. Therefore, it suffices to choose $c$ so that $4 \cdot c < \epsilon$ in order to conclude that the total number of possible bucket configurations is at most $n^\epsilon$, for any constant $\epsilon < 1$. This implies that a table with at most $n^\epsilon$ rows suffices to be able to encode all the possible bucket configurations.

**Expected size of $V$.** Assume to have randomly generated table $T$ storing $b^2$ symbols in each row. Each cell of $T$ is randomly set equal to any symbol $\alpha \in \Sigma$ with probability $f_\alpha$, where $f_\alpha$ denotes the frequency of the symbol $\alpha$. Given any bucket $B_i$, we say that 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 $j \in [k]$. Let $P_i$ denote the probability that a generic row of $T$ matches $B_i$. We have $P_i = \prod_{1 \leq j < k} f_{s_j}$, since $f_{s_j}$ is the probability that the row contains the symbol $s_j$ in position $o_j$. Let $X_i$ be the random variable corresponding to the position of the first matching row in $T$ for bucket $B_i$. Observe that $X_i$ follows a geometric distribution with $P_i$ as probability of success. Thus, the expected size of $V$ can be bounded in terms of 0-th order entropy:

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

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

We can 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 $|V| \leq (nH_0 + m)\left(1 + \frac{1}{\log^3 n}\right)$ with probability $\Omega(\frac{1}{\log^3 n})$.

**Finding a suitable table.** Notice that the analysis above does not limit in any way the number of rows that we have to generate until all the blocks find their first match. Thus, we may be forced to generate a table with $\Omega(n)$ rows. This is problematic for two reasons: 1) storing such a large table would affect the space complexity of our scheme; 2) the time complexity of the construction of our scheme would be superlinear. A simple way to overcome the first issue is to observe that at most $n^\epsilon$ rows of a table are used to encode all the possible bucket configurations. Therefore, any table can be contracted by removing all the unused rows without increasing the size of $V$.

The second problem is more subtle and requires

to modify the generating process of $T$. The process goes as described above but continues deterministically as soon as $T$ reaches $n^\epsilon$ rows. Let $W$ be the set of bucket configurations that do not find their match in the first $n^\epsilon$ randomly generated rows. We contract the table $T$ by removing its unused rows. In this way, we have space to accommodate $|W|$ new rows. We use these rows to match in an arbitrary fashion all the bucket configurations in $W$. The interesting fact is that the encoding with this new table is at least as space efficient as the one that uses the best table obtainable by continuing the random generation. The proof of this fact is as follows. The bucket configurations not in $W$ are encoded with (at least) the same efficiency because the contraction cannot increase the space. Indeed, the contraction moves the row matched by a bucket from the, say, $h$th position of the table to the $j$th position, with $j \leq h$. The bucket configurations in $W$ are encoded with at most $\log n^\epsilon$ bits. Conversely, they would be encoded with at least $\log n^\epsilon$ bits with any table obtained by continuing the random generation having $T$ already reached $n^\epsilon$ rows.

Clearly, this table has $n^\epsilon$ rows and it is obtained in $O(n^{2\epsilon} \cdot b)$ time. [4] Thus, since the random process generates a suitable table with probability $\Omega(\frac{1}{\log^3 n})$, we expect $\log^3 n$ attempts before succeed. Thus, the construction of our scheme requires expected linear time by choosing any $\epsilon < 1/2$.

**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 2.1 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 this quantity is required to be $O(n^\epsilon)$ for $\epsilon < 1$, the 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 2.2 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 2.1 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 2.3. *A static function $F$ defined over a subset $S = \{x_1, \ldots, x_n\} \subseteq U = [2^w]$ into symbols drawn from an alphabet $\Sigma = [\sigma]$ 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. A function $F_i$ is responsible only for keys associated with a portion of the symbols in $\Sigma$ 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 the keys of $S$ in groups by using the alphabet partitioning strategy presented in [2]. We sort symbols by their frequencies and 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 integer in $[0, \log^2 n/\log\log n - 1]$. Each group has assigned an unique identifier in $[\log^2 n]$. Let $\sigma_i$ 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 $F_i$s suffices, i.e., each $F_i$ can be represented by using $\log \sigma_i$ bits per key. We refer to [2] for a formal proof of this fact. 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

[4] We obtain this time complexity because we have to search the match of each of the $n^\epsilon$ bucket configurations on the $n^\epsilon$ rows of $T$.

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 $F_i$ 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 $F_i$'s symbols. The overhead of $\Theta(1)$ bits is within the lower order term of Theorem 2.1. Indeed, the contribution of each these symbols to original $F$'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 2.3. 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 2.1. 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 $i/\log n$ bits, while the lower order term is $O(n'\frac{n'\log\sigma_i\log\log n'}{\log n'} = O(n'\frac{(i+1)/\log n\log\log n}{\log n'}) = O(n'(i/\log n)\frac{\log\log n'}{\log n'})$.

The space bound claimed in Theorem 2.1 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 2.1.

The main drawback of Theorem 2.1 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 2.1 and an approximate membership data structure. The following theorem summarizes the obtained result. The proof of this theorem is deferred to Appendix A because it requires results provided in the next section.

THEOREM 2.2. *A static function $F$ defined over a subset $S = \{x_1, \ldots, x_n\} \subseteq U = [2^w]$ into symbols drawn from an alphabet $\Sigma = [\sigma]$ 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) $\epsilon$ is a power of two [5, 8, 15, 17]. 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 $\epsilon$ such that $\log\frac{1}{\epsilon} = o(\log n/\log\log n)$. More formally we prove the following theorem.

THEOREM 3.1. *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 $\epsilon$. The scheme can be built in $O(n)$ expected time.*

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

LEMMA 3.1. *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 chosen to be respectively $1-\epsilon$ and $\epsilon$. We would like to obtain a representation in which elements of $S$ have always assigned symbol 1, while elements in

$U \setminus S$ have assigned symbol 1 with probability at most $\epsilon$ and symbol 0 with probability at least $1-\epsilon$. Even in this case, we partition the keys of $S$ into $\Theta(n/b)$ buckets of size at most $b = c\frac{\log n}{\log\log n}$. 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 $K = [b^2]$. 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 $(b^2)^{c\frac{\log n}{\log\log n}} = 2^{2c\log n} = n^{2c}$ distinct sets of size $b = c\frac{\log n}{\log\log n}$ out of universe of size $k = b^2$, with $c < 1/2$ chosen so that $2 \cdot c < 1$.

As in the previous section, we randomly generate a binary table $T$ with $k$ columns and $n^{2c} = o(n)$ 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. [5] Notice that the value $k_0$ forces $\epsilon$ to be at least $\frac{2}{b}$ which is not problematic because we are restricting ourself to the case $\epsilon \in (1/2, 1)$. 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). 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

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

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 since $k = b^2$, we have that the final size of the encoding is $t\log\frac{1}{\epsilon} + O(1)$. By summing up the encoding sizes for all the $\Theta(n/b)$ buckets, we obtain the (expected) 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

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

This concludes the proof of Lemma 3.1.

We obtain a scheme for the general case with $\epsilon \leq 1/2$ by combining Lemma 3.1 with a compressed function of Theorem 2.1. We first observe that the false positive probability $\epsilon$ can be seen as $\epsilon = 2^{-i} \cdot d$, with $d \in (1/2, 1)$ and $i$ a positive integer. We store the keys in a first AM implemented according to Lemma 3.1 with fpp $d$. 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. [6] 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 $d$. 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 $1/2^i$. [7]

[5] Notice that when in the construction of the table the rows are deterministically chosen for the set $W$ (i.e., the bucket configurations that did not find their match in the first $n^\epsilon$ randomly generated rows), we are free to force all the $t$ cells corresponding to keys of the bucket configuration to 1 while the remaining cells are set to 0.

[6] Actually, a strictly pairwise independent hash function needs $O(w)$ bits which is too much for very small sets $S$ (namely, $|S| = O(w)$). In these cases, we use a $(1+\delta)$-universal hash function [16] by setting $\delta$ to $(1/n^2)$ which requires $O(\log n + \log w)$ bits. Such a hash function generates a fpp larger by a factor $1+\delta$ which can be compensated by using a smaller $\epsilon$ inducing a a negligible space overhead.

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

## 4 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 $x \in U \setminus R$.

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 [3] is based on a combination of an AM and a static function on binary alphabet and provides a constant time solution which requires $B(m,n) + O(n + \log w)$ bits of space. A simple way to obtain an improved result is to directly apply Theorem 2.2 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 2.2 and by observing that $nH_0 \approx B(m,n)$:

LEMMA 4.1. *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 4.1. *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 $n \leq m/2$ [8]) we use Lemma 4.1 which has overhead $O(n)$ 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.*.

[8] 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 $S$.

## 5 Conclusion

In this paper we presented both time and space optimal solutions for three well-established relaxations of this basic problem: *(Compressed) Static functions, Approximate membership* and *Relative membership*.

The best solutions for compressed static functions were by Porat [17] and Hreinsson *et al.* [12]. The first one requires $n \log \sigma(1 + o(1))$ 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 $\delta$ 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, and sublinear overhead. We strongly believe that these characteristics make the use of static functions significantly more appealing for many applications.

We use compressed static functions to solve the approximate membership problem. Previously known 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, 17], or require $O(n)$ extra bits overhead to deal with an arbitrary $\epsilon$ [12]. Our optimal scheme is the first known solution having $o(n)$ overhead for any value of $\epsilon$ such that $\log \frac{1}{\epsilon} = o(\log n / \log\log n)$. Essentially, for any reasonable choice of $\epsilon$ in practice.

Finally, we use compressed static functions to obtain a constant time solution for the relative membership problem achieving an optimal space complexity (up to lower order terms).

The most interesting open problem asks to design a deterministic construction algorithm for compressed static functions having a reasonable (construction) time complexity still retaining optimal space complexity. Finding an efficient algorithm for generating deterministically our encoding tables is another interesting open problem.

## 6 Acknowledgements

We would like to thank Giuseppe Ottaviano for insightful discussions and helpful comments on a preliminary version of this work.

## 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.

[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] Rasmus Pagh. Dispersing hash functions. *Random Struct. Algorithms*, 35(1):70–82, 2009.

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

[18] 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 Theorem 2.2

As we already observed, a drawback of Theorem 2.1 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 2.1 and the approximate membership data structure of Theorem 3.1 in order to prove the following theorem.

**Theorem 2.2.** *A static function $F$ defined over a subset $S = \{x_1, \ldots, x_n\} \subseteq U = [2^w]$ into symbols drawn from an alphabet $\Sigma = [\sigma]$ 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 $\alpha$ 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 indentifying most of $\alpha$'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 2.1. This suffices since, if all symbols have frequency at most $1 - \frac{\log\log n}{\sqrt{\log n}}$, the bounds of Theorem 2.1 and Theorem 2.2 already match. Indeed, in this case $H_0 = \Omega(\frac{(\log\log n)^2}{\sqrt{\log(nH_0)}})$ and Theorem 2.1 has an overhead of $O(\frac{n(\log\log n)^2}{\log n}) = O(\frac{nH_0}{\sqrt{\log(nH_0)}})$ bits.

Thus, in the following we will assume that there is a symbol $\alpha$ having frequency at least $f > 1 - \frac{\log\log n}{\sqrt{\log n}}$. We will also assume that $f < 1 - \frac{1}{\sqrt{n}}$. The extreme case $f > 1 - \frac{1}{\sqrt{n}}$ is left for the end of the section. The latter assumption means that each symbol appears with frequency at most $1 - \frac{1}{\sqrt{n}}$ and, thus, needs at least $\Omega(\frac{1}{\sqrt{n}})$ bits to be encoded. Thus, we have $nH_0 \geq \frac{n}{\sqrt{n}} = \sqrt{n}$. Since it also holds $nH_0 < n \log n$, we have $\log(nH_0) = \Theta(\log n)$ and we can use the two terms interchangeably.

Let $S_\alpha$ with $|S_\alpha| = nf$ denote the set of keys mapped to symbol $\alpha$ and let $q = \frac{\sqrt{\log n}}{2 \log\log n}$. Let $S_{\bar{\alpha}} = S \setminus S_\alpha$ be the set of keys mapped to a symbol different from $\alpha$. Observe that each of the $n(1-f)$ elements of $S_{\bar{\alpha}}$ needs at least $\log \frac{1}{1-f} \geq \log\log n$ bits to be encoded. This implies that $nH_0 \geq |S_{\bar{\alpha}}| \log\log n$.

We use the AM of Theorem 3.1 with false positive probability equal to $(1-f)q/f$ built on the set $S_{\bar{\alpha}}$ as a (pre-)filter. The keys that are recognized by the filter to do not belong to $S_{\bar{\alpha}}$ have automatically assigned the symbol $\alpha$. However, there exist keys in $S_\alpha$ that are erroneously considered in $S_{\bar{\alpha}}$ due to the approximation of the membership query. Let $E$ denote the set of these keys. We repeat the construction of the AM until the size of $E$ is at most its expected size $nf\frac{(1-f)q}{f} = n(1-f)q$. These wrong assignments are solved by using a compressed function of Theorem 2.1 on the set $R = E \cup S_{\bar{\alpha}}$, where we associate the symbol $\alpha$ to keys in $E$ and the 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 $x$ in this scheme is as follows: if $x$ is considered in $S_{\bar{\alpha}}$ by the AM, we return the symbol stored in the compressed function, otherwise we return $\alpha$.

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 need less space to be encoded. We consider separately the entropy of symbols in $S_{\bar{\alpha}}$ and in $S_\alpha$.

The AM stores symbols of $S_{\bar{\alpha}}$ with false positive probability $(1-f)q/f$. Thus, it uses space

$$|S_{\bar{\alpha}}| \log \tfrac{f}{(1-f)q} + O(\tfrac{(|S_{\bar{\alpha}}| \log \frac{f}{(1-f)q} + \log\log|S_{\bar{\alpha}}|)\log\log|S_{\bar{\alpha}}|}{\log|S_{\bar{\alpha}}|})$$

Since we have that $|S_{\bar{\alpha}}| \log \frac{f}{(1-f)q} \leq nH_0$, the lower order term above is within the lower order term of Theorem 2.2. The dominant term $|S_{\bar{\alpha}}| \log \frac{f}{(1-f)q}$ will be added later to the cost of encoding symbols in $S_{\bar{\alpha}}$.

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. Since $n' = n(1-f)(1+q) \leq nH_0(1+q) = nH_0(1 + \frac{\sqrt{\log n}}{2\log\log n})$ and $n'H_0' \leq nH_0$, the above 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 2.2.

We now turn our attention to the dominant terms of both AM and compressed function by analyzing separately the contributions of elements in $S_{\bar{\alpha}}$ and elements in $E \subset S_\alpha$. We know that the compressed function stores all elements of $S_{\bar{\alpha}}$ but only a subset $E$ of $S_\alpha$. Thus, we 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 $|E| < |S_\alpha|$. Thus, we deduce that

$$\begin{aligned} |E| \log \tfrac{n'}{|E|} &= |E| \log \tfrac{n+|E|-|S_\alpha|}{|E|} \\ &= |E| \log(1 + \tfrac{n-|S_\alpha|}{|E|}) \\ &< |S_\alpha| \log(1 + \tfrac{n-|S_\alpha|}{|S_\alpha|}) \\ &= |S_\alpha| \log \tfrac{n}{|S_\alpha|}. \end{aligned}$$

The last inequality derives from the fact that the function $x \log(1+\frac{b}{x})$ is monotonically increasing when $x > 0$ (we replace $x$ by $|E|$ and $|S_\alpha|$ and replace $b$ by $n-|S_\alpha|$). 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 $S_{\bar{\alpha}}$. Let $S_c \subset S_{\bar{\alpha}}$ be the set of elements associated with symbol $c$. The contribution of $S_c$ to $nH_0$ is

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

while its contribution to $n'H'_0$ is

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

For each element in $S_{\bar{\alpha}}$ we have also to add the $\log \frac{f}{(1-f)q}$ bits used by the AM. Thus, the overall space used by an element of $S_c$ is

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

Thus, the space used for an element in $S_{\bar{\alpha}}$ is within $\Theta(\frac{1}{q}) = \Theta(\frac{\log\log(nH_0)}{\sqrt{\log(nH_0)}})$ from optimal. We conclude the proof of Theorem 2.2 (for the case $f \leq 1 - \frac{1}{\sqrt{n}}$) by observing that the space used to encode all the elements in $S_{\bar{\alpha}}$ is within $O(|S_{\bar{\alpha}}| \cdot \frac{1}{q}) = O(\frac{nH_0}{q}) = O(nH_0 \frac{\log\log(nH_0)}{\sqrt{\log(nH_0)}})$ from optimal since $|S_{\bar{\alpha}}| \leq nH_0$.

It remains to take care of the extreme case where $f > 1 - \frac{1}{\sqrt{n}}$. In this case we use the approach above with the only difference of using different data structures and a false positive probability equal to $\epsilon = \frac{1-f}{f}$. We use simple strategy mentioned in the introduction to implement the AM with minimal perfect hash function. This AM requires $m \log \frac{1}{\epsilon} + O(m)$ to store $m$ elements with fpp equal to $\epsilon$. In our case, it translates in paying $n\frac{1-f}{f} \log \frac{f}{1-f} + O(n\frac{1-f}{f})$ bits of space. We resort to Lemma 2.1 for the compressed function on the set $R$. Since $|E| = \epsilon|S_\alpha| = nf\frac{1-f}{f} = n(1-f) = |S_{\bar{\alpha}}|$, the compressed function uses $\log \frac{|E|+|S_\alpha|}{|E|} + O(1) = O(1)$ bits for each element of $E$ and $|S_c| \log \frac{S_{\bar{\alpha}}|+|E|}{|S_c|} + O(1) = |S_c| \log \frac{S_{\bar{\alpha}}|}{|S_c|} + O(1)$ for each element of $S_c \subset |S_{\bar{\alpha}}|$. The AM adds $\log \frac{n}{|S_{\bar{\alpha}}|} + O(1)$ for each element of $|S_{\bar{\alpha}}|$. Thus, each element of $S_c$ needs $\log \frac{n}{|S_{\bar{\alpha}}|} + \log \frac{S_{\bar{\alpha}}|}{|S_c|} + O(1) = \log \frac{n}{|S_c|} + O(1)$ bits overall. This gives a total of $nH_0 + O(|S_{\bar{\alpha}}|)$ bits for all the elements in $S_{\bar{\alpha}}$. The bound does not increase if we add the $O(|E|) = O(|S_{\bar{\alpha}}|)$ bits required for all the elements in $E$. We conclude by observing that each element of $S_{\bar{\alpha}}$ has entropy at least $\log \frac{n}{\sqrt{n}} = \Omega(\log n)$ and, thus, we have $nH_0 = \Omega(|S_{\bar{\alpha}}| \log n)$. Therefore, the term $O(|S_{\bar{\alpha}}|)$ is $O(\frac{nH_0}{\log n}) = O(\frac{nH_0}{\log(nH_0)})$ is within the lower order term of Theorem 2.2.