Learned Static Function Data Structures

Stefan Hermann
Karlsruhe Institute of Technology
Karlsruhe, Germany
hermann@kit.edu

Hans-Peter Lehmann
Karlsruhe Institute of Technology
Karlsruhe, Germany
hans-peter.lehmann@kit.edu

Giorgio Vinciguerra
University of Pisa
Pisa, Italy
giorgio.vinciguerra@unipi.it

Stefan Walzer
Karlsruhe Institute of Technology
Karlsruhe, Germany
stefan.walzer@kit.edu

ABSTRACT

We consider the task of constructing a data structure for associating a static set of keys with values, while allowing arbitrary output values for queries involving keys outside the set. Compared to hash tables, these so-called *static function data structures* do not need to store the key set and thus use significantly less memory. Several techniques are known, with *compressed* static functions approaching the zero-order empirical entropy of the value sequence. In this paper, we introduce *learned* static functions, which use machine learning to capture correlations between keys and values. For each key, a model predicts a probability distribution over the values, from which we derive a key-specific prefix code to compactly encode the true value. The resulting codeword is stored in a classic static function data structure. This design allows learned static functions to break the zero-order entropy barrier while still supporting point queries. Our experiments show substantial space savings: up to one order of magnitude on real data, and up to three orders of magnitude on synthetic data.

PVLDB Reference Format:
Stefan Hermann, Hans-Peter Lehmann, Giorgio Vinciguerra, and Stefan Walzer. Learned Static Function Data Structures. PVLDB, 19(5): 917-930, 2026.
doi:10.14778/3796195.3796205

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gvinciguerra/LearnedStaticFunction.

1 INTRODUCTION

The standard way for storing a map $f : K \to V$ from keys to values is a hash table [8, 47, 48, 62]. However, when f is static and queries are restricted to keys in K (with arbitrary outputs allowed otherwise), specialised data structures called *static functions* or *retrieval data structures* can be used instead [13, 14, 25, 26, 33, 36, 37, 65, 79]. Their main advantage is that the keys do not have to be stored, and the mapping can thus be represented in a little more than $|K| \log_2 |V|$ bits.[1] Consider, for instance, a set K of n URLs annotated with labels from $V = \{\text{GOOD}, \text{AI-SLOP}, \text{PHISHING}\}$. A static function could represent this mapping using $n \log_2 3 < 2n$ bits, regardless of the URL lengths. In contrast, any encoding of the set of all key-value pairs (e.g. using a hash table) would need orders of magnitude more space.

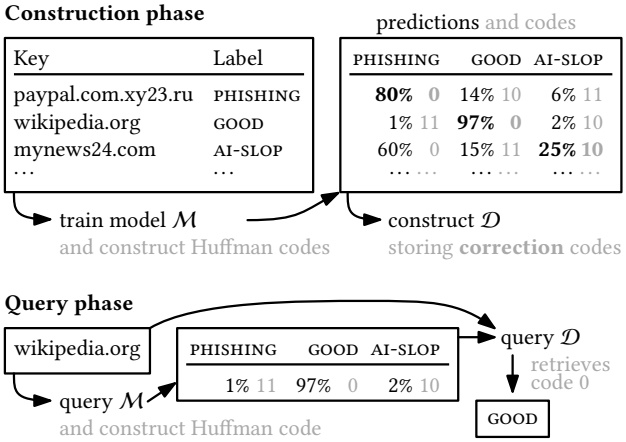


Figure 1: Architecture of learned static functions. In gray, a simple implementation that stores Huffman codes in the auxiliary data structure $\mathcal{D}$.

For random functions with domain K and range V, we cannot do better than $|K| \log_2 |V|$ bits on average. However, when some values from V appear much more frequently than others, the space usage can be further reduced using *compressed static functions* (CSFs) to approximately nH_0 bits, where $H_0 = \sum_{v \in V} p_v \log_2 (1/p_v)$ denotes the zero-order empirical entropy of the value sequence, and $p_v \in [0, 1]$ is the relative frequency of v [7, 33, 41]. In the example, if 95% of the URLs are GOOD, 4% are AI-SLOP and 1% are PHISHING, a CSF could achieve as little as $0.32n$ bits.

If f exhibits low H_0 but is otherwise random, we have again hit an information-theoretic barrier. However, if f stores real-world data, then knowing a key may allow us to make an informed guess about its associated value. In this paper, we leverage such key-specific predictions to achieve significant compression, even when each value appears equally often overall, i.e., even when $H_0 \approx \log_2 |V|$. Going back to our example, we might observe that most `.edu` URLs are GOOD, while URLs containing sneaky misspellings are more likely to be PHISHING. Such correlations are often subtle, domain-specific, and difficult to capture with handcrafted rules, making learning-based approaches a natural fit. We hence propose *learned static functions* (LSFs), which involve two components, as illustrated in Figure 1.

- First, a model $\mathcal{M}$ that is specific to the use case and trained to predict $f(k)$ given $k \in K$. Technically, $\mathcal{M}$ must output a probability distribution on V reflecting what it believes

Proceedings of the VLDB Endowment, Vol. 19, No. 5 ISSN 2150-8097.
doi:10.14778/3796195.3796205

[1] How precisely an SF can achieve this using systems of linear equations is beside the point of this introduction. Some details are unpacked in Section 4 as needed.

$f(k)$ to be. Additional data related to k can inform the prediction, provided it is also available at query time.[2]
- Second, an auxiliary data structure $\mathcal{D}$, much like a CSF, intuitively responsible for correcting all the mistakes made by $\mathcal{M}$. This suggests that if $\mathcal{M}$ is right about most keys, then $\mathcal{D}$ needs only to store a small set of exceptions. Note, however, that technically the "rightness" of $\mathcal{M}$ about a given key is not categorical but a matter of degree (explained more precisely in Section 2).

When LSFs Might be Useful. Like all SFs, an LSF faithfully returns $f(k)$ whenever $k \in K$, but returns an arbitrary value for $k \notin K$. Among contexts where this is acceptable, LSFs are particularly well-suited for scenarios where space efficiency is paramount, such as when the key set is so large that traditional data structures like CSFs no longer fit in fast, local, or cheap memory. LSFs are especially effective when the mapping from keys to values exhibits structure that can be learned, in the sense that the combined encoding of the model $\mathcal{M}$ and the data structure $\mathcal{D}$ meaningfully undercuts H_0 that would be achievable with a CSF.

Applications. SFs and CSFs have applications in areas such as bioinformatics [18, 75], log management [68], systems for ML [18], data structures for range queries [1, 59], perfect hashing [6, 14, 27, 33, 40, 53], prefix search [5], and approximate membership [25, 26, 36, 37] (Bloom filters), which themselves have widespread use.

LSFs open new opportunities in contexts where keys and values exhibit correlations. For example, we experiment with a dataset where URLs are labelled with {PHISHING, GOOD}. Such a labelling might be generated by a web-crawler at an early stage to later also detect clusters of sites linked to phishing. Both labels appear similarly often. Our LSF represents the labelling using 0.053 bits per key, while any CSF requires at least 0.99 bits per key — a compression by a factor of 18.7. Our experiments encompass additional real-world and synthetic datasets. In the following, we outline possibilities which exceed those we experimented with.

- *Tabular Data.* In many structured datasets, such as relational databases, the value of one column can often be inferred from others. For example, in a product catalogue, the category of an item (e.g., books, clothing, electronics) can often be inferred from fields like the title or brand. LSFs offer a natural solution to compress and efficiently retrieve these predictable values. Previous work exploited such learnability in tabular data in a lossy way [4, 43], without point query support [22, 32], or assuming linear correlation only [55]. Additional opportunities arise from exploiting correlations across tables. For example, customer attributes such as demographics can help predict order-related columns such as purchased product, payment method, or delivery time. When tables are joined and materialised (e.g., in views), LSF can help reduce storage space.
- *Hyphenation.* Imagine f maps English words to their correct hy-phen-ation, i.e., a set of positions where a word break is allowed. It is not hard to imagine cases where an app could use a corresponding data structure but would only want to use negligible memory to store it and would not require large throughput. Given that hyphenation sounds highly learnable, an LSF might be useful.
- *Map Annotations.* Imagine a road network where some information is useful but unlikely to be requested frequently, such as whether some footpath is wheelchair accessible or whether some road is winter maintained. Geographic regularities might make such data highly learnable. We experiment with a related (though different) dataset regarding forest types.
- *Endgame Tablebases.* When analysing games with complete information such as chess, it can be useful to tabulate information for vast sets of game states (especially states with few remaining pieces), storing for each the best possible move or simply the outcome from {DRAW, WIN, LOSS} assuming optimal play [34]. Such labels are clearly learnable, and thus LSFs could help compress tablebases. It seems, however, that the LSF would also have to be made aware of complicated game-specific symmetries that are well-known but beyond the scope of this paper.
- *k-mer Annotations.* Over the past 20 years, the cost of genome sequencing has dropped by roughly four orders of magnitude, leading to a growth of bioinformatics datasets that outpace the rate at which computational hardware becomes cheaper. We believe that LSFs can play a role in conserving memory, e.g., when storing annotations of large sets of DNA snippets, called k-mers. The annotations might reflect taxonomic origin (think "mouse or rat"), whether the snippet is coding or non-coding, its function (related GO [17] term or KEGG [45] pathway), or whether it is unique in the reference genome. All of these are learnable in some contexts.

Our Contributions. We achieve the following:

- We formally introduce the concept of learned static functions (LSFs) and give the first practical implementation.
- As an intermediate problem, we extend the applicability of BuRR [26], a highly space-efficient practical SF, by adapting it to implement a variable-length static function (VL-SF) storing a map $f : K \to \{0,1\}^*$ from keys to variable-length bit strings.
- We address a key inefficiency in the design of LSFs and existing CSFs [33, 41] based on VL-SF by introducing a "generalised filter trick", which saves up to $\frac{1}{2}$ bits per key.[3]
- We evaluate several machine learning (ML) models to minimise the space of our LSF, showing how model choice impacts performance and storage.
- We demonstrate the effectiveness of our overall approach on real-world datasets spanning diverse domains such as geographic information systems, network security, phishing detection, and music, as well as synthetic datasets. Our experiments show space savings of up to one order of magnitude on real data, and up to three orders of magnitude on synthetic data.

[2] This makes no conceptual difference as any such information can be considered to be part of the key. An example could be the key being a file name and queries reading the file to extract additional information.

[3] The case where up to $\frac{1}{2}$ bits per key can be saved is the second example in Table 2.

Table 1: Types of static functions (SFs).

Type	Context	Ideal space (bits)	References
SF	$f : K \to V$	$n \log_2 \lvert V \rvert$	[13, 14, 25, 26, 33, 36, 37, 65]
r-bit SF	$f : K \to \{0,1\}^r$	nr	
VL-SF	$f : K \to \{0,1\}^*$	$\sum_{k \in K} \lvert f(k) \rvert$	[7, 18, 33, 41]
CSF	$f : K \to V$	nH_0	
LSF	$f : K \to V$, $\mathcal{M} = (\mu_k)_{k \in K}$	$S(f, \mathcal{M})$ + $\lvert \text{enc}(\mathcal{M}) \rvert$	NEW

Outline. We start with preliminaries in Section 2. We then introduce our new type of data structures, learned static functions, in Section 3. In Section 4, we build a VL-SF based on BuRR [26]. In Section 5, we explain the implementation of an LSF based on VL-SFs and look at engineering challenges. We perform an evaluation using different ML models in Section 6. In Section 7, we explain related work from the literature and compare it to our new data structure. Finally, we conclude the paper in Section 8.

2 PRELIMINARIES

A *static function* (SF), also called a *static retrieval data structure*, is a data structure constructed for a function $f : K \to V$ where K is a set of n *keys* from a universe U and V is a (typically small) set of *values* or *labels*. After construction, it cannot be modified but just queried with $k \in U$. If $k \in K$, then the answer must be $f(k)$. If $k \in U \setminus K$, then the answer is arbitrary. In Table 1 we summarise variants of SFs discussed below.

Related: Filter Data Structures. In a sense, an SF must "forget" parts of the data for which it was constructed to be space-efficient. It shares this property with Bloom filters [11] (filters for short, also known as approximate membership query data structures). A filter represents a set $M \subseteq U$ and answers membership queries "$k \in M$?" with one-sided errors, i.e., any $k \notin M$ may be erroneously reported to be present in M with a small false-positive probability ε. Any SF can be used as a filter [25, 26, 36, 37], while filters are helpful when constructing CSFs [7, 18, 33, 41].

r-bit Static Functions. Most uncompressed SFs assume $V = \{0,1\}^r$ for some $r \in \mathbb{N}$; we call them r-bit SFs.[4] The case of $r = 1$, i.e. $V = \{0,1\}$, has also been called the relative membership problem [7] and is closely related to Bloom filters with a false positive free zone [46]. Succinct r-bit SFs using $(1 + o(1))rn$ bits are known; we will build upon one such SF called *BuRR* [26] in Section 4.

Variable-Length Static Functions (VL-SF). Given a space-efficient 1-bit SF, it is not hard to construct an SF for the case where values are variable-length bit strings, i.e. $V = \{0,1\}^*$, while using close to $\sum_{k \in K} |f(k)|$ bits. Conceptually, we treat a key k associated with an l-bit string $b_1 \dots b_l$ as l independent keys $(k,1), \dots, (k,l)$, where (k,i) is associated with b_i. If we know a key k and l, then we can recover $f(k)$ by internally querying $(k,1), \dots, (k,l)$. A challenge is to use this idea without losing a factor of l in running time. We comment on these challenges in the case of BuRR in Section 4.2.

[4] An exception is an SF using $V = \{0,1,2\}$ [33, Section 7.2].

Note that if l is not known when querying k, we get a stream of bits that begins with $f(k)$ and continues with garbage bits. The full VL-SF needs to detect when the meaningful part of the stream ends. This is easy if we are storing prefix codes, but otherwise additional space might be required.

We remark that storing variable-length bit strings efficiently is non-trivial even if $K = \{1, \dots, n\}$, i.e., when storing an array of variable-length bit strings with random access, see e.g., [50, 51] and [61, Section 3.2]. See also [64] for compressed arrays.

Compressed Static Functions (CSFs). It is easy to build CSFs from VL-SFs. Given $f : K \to V$, we consider the relative frequencies $p_v := |f^{-1}(v)|/|K|$ with which v occurs and construct a corresponding optimal prefix code for V (e.g. a Huffman code [42]). We then use a VL-SF to store the codeword of $f(k)$ for each $k \in K$. The combined codeword length is then $|K|(H_0 + \delta)$ bits, where $\delta \in [0,1]$ is a term accounting for the redundancy of the prefix code [31]. The full CSF must also store the prefix code (e.g., as a Huffman tree), which takes negligible space. GOV [33] is the current state-of-the-art practical CSF. [5] A noteworthy corner case for classical CSFs is when H_0 is close to zero, which may happen if the same element of V is assigned to most keys from K. In that case, the CSF as described above still requires at least 1 bit per key and thus is not space efficient. The problem can be fixed by using a Bloom filter that represents the small set of keys not mapping to the majority element [7, 18, 33, 41, 75]. We explain and expand upon this *filter trick* in Section 5.

Surprisal. Given a probability distribution μ and an outcome x with probability $p = \mu(x)$, then $\log_2(1/p)$ is called the *surprisal* (or *information content*) of x given μ. Note that surprisal can arise even when x is not random: If Alice, when asked about the capital of Canada, assigns a credence of 75% to Toronto and 25% to Ottawa, then her surprisal when learning the truth (Ottawa) is $\log_2(4) = 2$ bits. A different observer may experience different surprisal.

3 LEARNED STATIC FUNCTIONS

We are now ready to introduce the concept of learned static functions (LSFs). An LSF represents a mapping $f : K \to V$ in a context where we can construct an ML model $\mathcal{M}$ that, given any $k \in K$, outputs a probability distribution μ_k on V, intuitively reflecting what $\mathcal{M}$ believes $f(k)$ to be.[6] From each distribution μ_k, we derive a prefix code and store the codeword corresponding to $f(k)$ in an auxiliary data structure $\mathcal{D}$. This step is analogous in spirit to a CSF but with a crucial difference: the prefix code is determined by μ_k and is specific to each key, while a CSF uses the same prefix code for every key. Importantly, the prefix code need not be stored: a query for k will consult $\mathcal{M}$ again, obtain the same probability distribution μ_k and compute the same prefix code, which allows decoding $f(k)$ from the bits stored in $\mathcal{D}$.

[5] CARAMEL [18] implements a CSF that supports the storage of tuples rather than single values. Since it builds directly on GOV's implementation and, in our setting without tuples, would perform equivalently to GOV, we do not discuss it further.

[6] We could also work with a weaker model that merely outputs a single value $v_k \in V$ and a confidence $p \in (0,1)$ that $v_k = f(k)$. To obtain a distribution μ_k on all of V, we could simply allocate the remaining probability mass $1 - p$ to $V \setminus \{v_k\}$ according to the relative frequencies or in some other ad-hoc way. However, this is undoubtedly less powerful in some contexts.

Ideally, the storage cost for each key k is $\log_2(1/\mu_k(f(k)))$ bits, which is the surprisal of the outcome $f(k)$ under the probability distribution μ_k.[7] Summing over all keys yields the total space cost

$$S(\mathcal{M}, f) := \sum_{k \in K} \log_2(1/\mu_k(f(k))). \tag{1}$$

While this suggests that $S(\mathcal{M}, f)$ can serve as a loss function when training $\mathcal{M}$, minimising it alone does not fully capture the actual optimisation goal in the LSF setting, which differs from classical supervised learning in two important ways. First, the model does not have to generalise well beyond K, i.e., overfitting to K is not a problem but actually desirable to some extent. Second, the size $|\text{enc}(\mathcal{M})|$ of the model $\mathcal{M}$ itself contributes to the total space budget. The actual optimisation goal is therefore minimising

$$\text{space} \approx \Sigma := S(\mathcal{M}, f) + |\text{enc}(\mathcal{M})|. \tag{2}$$

Therefore, for small- and medium-sized datasets, the most effective choice is typically a simple model or heuristic rather than a deep neural network, even if the latter yields a smaller value for $S(\mathcal{M}, f)$. That said, in scenarios where a suitable model is already available (e.g., trained for a related upstream task), it can be reused for LSF at no additional storage cost, making the use of complex models more attractive in such cases.

Defining LSFs conceptually is only the first step. To make LSFs practical, we must now design the auxiliary data structure $\mathcal{D}$. To this end, we develop the following:

- A space-efficient VL-SF called VL-BuRR (Section 4).
- Based on VL-BuRR, a *weighted filter data structure* where a weight $r(k)$ controls the false-positive probability $2^{-r(k)}$, individually for each key $k \in U$. (Section 5.1)
- Based on the weighted filter, a *weighted relative membership* (WRM) data structure, or equivalently, a 1-bit SF where each key k has a weight $p(k)$ that indicates the probability that k is assigned a value of 1. If the weights are provided by an ML model, we have a 1-bit LSF. (Section 5.2)
- A general LSF that conceptually reconstructs a prefix code of a key's value by using an adaptive sequence of 1-bit LSF queries. (Section 5.3)

An advantage of our method is that prefix codes are not stored directly, and any inefficiencies in the prefix code are not reflected in the final space consumption.[8] By exploiting this flexibility:

- We adopt prefix codes of suboptimal length that are much faster to construct than optimal prefix codes such as Huffman. (Section 5.4)

In practice, the space required by $\mathcal{D}$ exceeds $S(\mathcal{M}, f)$ by at most 10.1%, provided that $S(\mathcal{M}, f)$ is large enough to render lower-order overheads negligible.

4 SPACE-EFFICIENT VL-SFs

In this section, we present a new design for VL-SFs, i.e. data structures that represent a mapping $f : K \to \{0,1\}^*$ where the values are variable-length bit strings. Previous practical approaches incur a multiplicative space overhead of at least 2.4% over the ideal cost of storing $\sum_{k \in K} |f(k)|$ bits [33]. Our approach builds upon BuRR [26] and extends it to efficiently handle variable-length values, achieving an overhead of at most 1.2%.

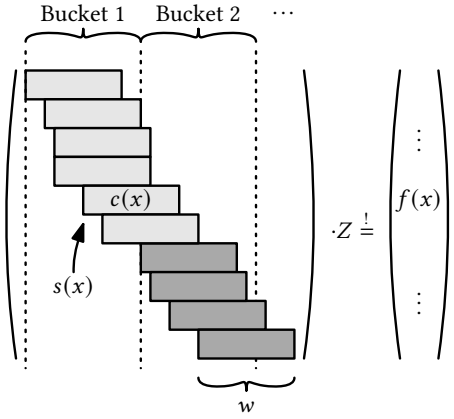


Figure 2: Structure of the BuRR equation system [26].

We begin by briefly recalling BuRR in Section 4.1, and then we describe our new VL-SF construction in Section 4.2.

4.1 1-bit BuRR

BuRR [26] is an r-bit SF. For our purposes, we recall its 1-bit variant, which stores a mapping $f : K \to \{0,1\}$ for a given set K of n keys. The construction of BuRR is based on linear algebra over the field $\mathbb{F}_2 = \{0,1\}$. The mapping itself is represented by a binary vector $Z \in \{0,1\}^m$, with $m \geq n$ rows. To query a key k, we first determine a hash row vector $h(k) \in \{0,1\}^{1 \times m}$, and then we compute the dot product $h(k)Z$ (modulo 2) to obtain $f(k)$. To construct Z, we must ensure that $h(k)Z = f(k)$ for all keys k. This corresponds to solving the linear system $HZ = F$, where $H \in \{0,1\}^{n \times m}$ is the matrix with a row $h(k)$ for each $k \in K$, and $F \in \{0,1\}^n$ is the vector of all values. Even if $m = n$, there is a chance of ≈ 0.289 that the system is solvable if h is fully random [19]. We then obtain an SF that requires just m bits to store Z. However, the query time, i.e., the cost of computing $h(k)Z$, is linear in m. BuRR achieves $O(1)$ query time by choosing $h(k)$ such that its non-zeroes are within a small window. More specifically $h(k) = 0^{s(k)-1}c(k)0^{m-s(k)-w+1}$, where $w \in \mathbb{N}$ is called the *ribbon width* (e.g. $w = 64$), $s(k) \in \{1, \ldots, m-w+1\}$ is a random starting position, and $c(k) \in \{0,1\}^w$ is a random binary string. Hence, to compute $h(k)Z$, BuRR has to consider at most w consecutive entries of Z, beginning with entry $s(k)$. In addition to improving cache efficiency during queries, this locality (and sorting the rows by $s(k)$) transforms the system of linear equations we have to solve during construction into a form that is almost a diagonal matrix: the only 1-bits form a narrow *ribbon* along the diagonal, as we illustrate in Figure 2. However, this locality comes at the cost of the solvability of the system $h(k)Z = f(k)$. If too many keys have similar starting positions $s(k)$, the system becomes unsolvable. BuRR then *bumps* keys recursively to another BuRR data structure to reduce the number of keys in overloaded regions and therefore ensure solvability. A fixed number of consecutive starting positions form a *bucket* (see Figure 2) and share bumping

[7] This is sometimes called the categorical cross entropy, i.e. the cross entropy of μ_k relative to the distribution that assigns probability 1 to $f(k)$.

[8] For example, with a confident model prediction such as $\mu_k(f(k)) = 0.9$, the ideal cost for key k is only $\log_2(1/\mu_k(f(k))) \approx 0.05$ bits, which we get close to, while a prefix code still requires at least 1 full bit.

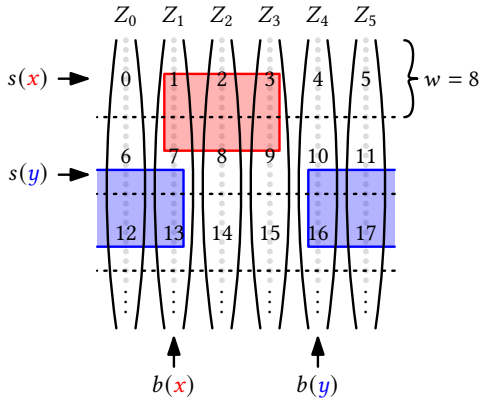


Figure 3: A VL-BuRR data structure using $\ell_{\max}$ separate 1-bit SFs, given by column vectors $Z_0, \ldots, Z_{\ell_{\max}-1}$ (here $\ell_{\max} = 6$). Shaded areas are two examples of the bits accessed by a single query. Grey dots indicate individual bits, dotted lines indicate how the bits are grouped into words of length w, and numbers indicate the order in which these words are stored.

metadata. The metadata stores a threshold that splits the keys in the bucket into those that remain in the equation system and those that are bumped. In the presence of bumping, it is advantageous to choose m slightly *smaller* than n. This ensures that every part of the data structure is at full capacity. See [26] for details.

4.2 VL-BuRR

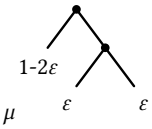
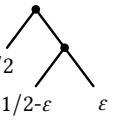
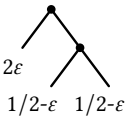
In Section 2, we explained how VL-SFs can be constructed by encoding each bit as an individual key. An alternative technique is to use several independent 1-bit SFs, one for each bit position of the values, and to query them bit by bit. However, these straightforward approaches suffer from poor cache locality and thus high query latency. We address this by adopting an interleaved storage layout of $\ell_{\max}$ instances of 1-bit BuRR, where $\ell_{\max}$ is the length of the longest value. There are three ingredients needed to accomplish this.

(1) *Same Load.* There are a total of $B = \sum_{k \in K} |f(k)|$ bits that our VL-SF has to represent. We ensure that all of our $\ell_{\max}$ BuRR instances represent roughly equal numbers of bits ($\approx B/\ell_{\max}$) as follows. The first bit of each value is represented by the $b(k)$-th SF, where $b : K \to \{0, 1, \ldots, \ell_{\max} - 1\}$ is a hash function. The second bit is represented by the $((b(k) + 1) \bmod \ell_{\max})$-th SF and so on. We therefore view the individual BuRRs arranged in a cyclic list.
(2) *Same Dimension.* To ensure that the solution vectors Z_i of all instances have equal size, we set their sizes to the maximum $m_{\max}$ of what the individual sizes would naturally be.
(3) *Same Hash Function.* We choose the same hash function $s(k)$ in each structure to map keys to starting positions.

Let $Z_i \in \{0, 1\}^{m_{\max}}$ be the solution vector of the i-th BuRR data structure, where $i \in \{0, 1, \ldots, \ell_{\max} - 1\}$. With the ingredients above, the bits that a query reads are in the same positions within all Z_i. To make the queries cache efficient, we only have to interleave the storage layout of all Z_i. We do this by dividing each Z_i into groups of w bits (where w is the ribbon width). We then store the groups from all Z_i in a round-robin fashion. Usually, all the bits that a query needs to read are therefore in the same cache line. We give an illustration in Figure 3.

BuRR ensures that the resulting system of linear equations is solvable by bumping keys to a fallback structure. Note that when used in a variable-length context, bumping information across the BuRR structures is positively correlated: if the i-th bucket of the k-th ribbon requires bumping, it is quite likely that the i-th bucket of the $(k+1)$-th ribbon also requires bumping, because of the codes that extend across both structures. Given this observation, it is wasteful to store bumping information for each BuRR structure individually. Instead, the i-th buckets of all structures share common bumping information. Like in r-bit BuRR, bumped keys are handled recursively using a second instance of our data structure.

Table 2: Code trees (leaves annotated with probabilities) where the expected codeword length L significantly exceeds the entropy $H(\mu)$ of the underlying distribution.

Code tree and distribution μ			
L	$1 + 2\varepsilon$	$\frac{3}{2}$	$2 - 2\varepsilon$
$H(\mu)$	$O(\varepsilon \log 1/\varepsilon)$	$1 + O(\varepsilon \log 1/\varepsilon)$	$1 + O(\varepsilon \log 1/\varepsilon)$
$\lim_{\varepsilon \to 0} \frac{L}{H(\mu)}$	∞	50%	∞

5 OUR LSF BASED ON VL-BuRR

In this section, we present our construction of learned static functions. Recall the plan to rely on an ML model to output a distribution μ_k on V for each $k \in K$ and to use μ_k as a prior when storing and retrieving $f(k)$. In the following, we describe this auxiliary data structure $\mathcal{D}$ that stores all $f(k)$. Roughly speaking, we construct a binary prefix-free code based on μ_k and store the codeword for $f(k)$ in the VL-SF from Section 4.

An issue with prefix codes generally is that for $v \sim \mu$ the length ℓ_v of its codeword may have an expectation $L = \mathbb{E}[\ell_v]$ that significantly exceeds the entropy $H(\mu)$. Table 2 illustrates three such cases. The first case shows that $H(\mu)$ could be close to 0 while L is always at least 1. This can happen if a value has a probability close to 1. Previous work on CSFs has addressed this problem with a filter that stores all keys *not* assigned the most common value [18, 41, 75]. Most keys that *do* use the most common value can then be recognised by being negative elements of the filter. We use this "filter trick" and expand it to cases where a skewed binary decision occurs in places other than the code tree's root node, which can save up to $\frac{1}{2}$ bits per key (second case in Table 2). In our experiments, this expansion saved up to 0.18 bits per key. Our techniques actually allow us to work with suboptimal prefix codes (third case in Table 2) without suffering from worse space efficiency.

We begin in Section 5.1 by explaining how VL-BuRR can serve as a *weighted filter*. We then use the weighted filter to build in Section 5.2 what we call a *weighted relative membership* data structure. This data structure can efficiently store the skewed binary choices

made when descending the code tree. By suitably concatenating the information related to the same key, we obtain our LSF implementation in Section 5.3. Finally, in Section 5.4 we discuss our use of suboptimal prefix codes, which are much faster to construct on the fly than optimal prefix codes (e.g. Huffman) and significantly improve query time with minimal sacrifices in space efficiency.

5.1 Weighted Filters from VL-SF

Recall that a filter is constructed for a set M and answers queries of the form "$k \in M$?" with no false negatives and a false positive probability of ε. It has long been known [25] that a filter with $\varepsilon = 2^{-r}$ can be obtained from an r-bit SF as follows. We use a hash function $g : U \to \{0,1\}^r$ (sometimes called *fingerprint function*) and build an SF for the restriction $f = g|_M$ of g to the domain M. A query simply checks whether $\text{SF.query}(x) = g(x)$, which is true for all $x \in M$ by construction, and true for $x \in U \setminus M$ only with probability 2^{-r} since $g(x)$ is a random bitstring independent from the SF.

The idea naturally generalises to *weighted filters*. These are constructed for a set M and a function $r : U \to \mathbb{R}_{\geq 0}$ to which we assume oracle access. A query must satisfy

$$\Pr[\text{query}(x) = 1] \begin{cases} = 1 & \text{if } x \in M, \\ \leq 2^{-r(x)} & \text{if } x \notin M. \end{cases}$$

If r only attains values in $\mathbb{N}_0$ then we can implement such a filter by storing a fingerprint of length $r(x)$ in a VL-SF.

Previous work has studied weighted Bloom filters and variations in contexts where the elements of U differ in their probabilities of being in M and differ in their probabilities of being queried [9, 15].

5.2 Weighted Relative Membership from Weighted Filters

A relative membership data structure (RM) [7] is constructed for two sets M, X with $m = |M|$, $n = |X|$, and $M \subseteq X \subseteq U$. For a given $x \in X$, it must answer whether $x \in M$. Equivalently, an RM is a 1-bit SF with domain X that indicates membership in M.

A *weighted relative membership* data structure (WRM) is given, in addition to M and X, a function $p : U \to [0, \frac{1}{2}]$, which we interpret as probabilistic information indicating $p(x) = \Pr[x \in M \mid x \in X]$.[9] For now, let us assume that M is indeed obtained by including each $x \in X$ independently with probability $p(x)$. The information-theoretic space lower bound for the WRM is then $\sum_{x \in X} H(p(x))$ bits, where $H(p) = p \log_2(1/p) + (1-p)\log_2(1/(1-p))$ is the binary entropy function. The following WRM gets close to this bound.

We construct a weighted filter F for M (with weight function $r : U \to \mathbb{N}_0$ defined below) as well as a 1-bit SF that stores for all positives of F whether they are true positives or false positives. We give pseudocode in Algorithms 1 and 2.

We now discuss which choice of r minimises space. We assume for simplicity that the 1-bit SF requires 1 bit per key and the weighted filter requires $r(x)$ bits for each $x \in M$ (with 1-bit BuRR and VL-BuRR we get within 1.2% of this ideal on sufficiently large inputs where lower order terms are negligible).

[9] The assumption that $p(x) \leq \frac{1}{2}$ simplifies further discussion. It is made without loss of generality as the role of "$\in$" and "$\notin$" can be swapped for keys x with $p(x) > \frac{1}{2}$.

Algorithm 1: Construction of our WRM data structure.

Input: $M, X, p : U \to [0, \frac{1}{2}]$
Output: a WRM data structure

```
choose r : U → ℕ_0 // see discussion
F ← constructWF(M, r)
truePos ← {x ↦ 1 | x ∈ M}
falsePos ← {x ↦ 0 | x ∈ U \ M, F.query(x) = 1}
SF ← constructSF(truePos ∪ falsePos)
return (F,SF)
```

Algorithm 2: Querying our WRM data structure.

Input: (F,SF), x

```
return if F.query(x) then SF.query(x) else 0
```

Any $x \in X$ contributes $p(x) \cdot r(x)$ bits to the expected space usage of F (it requires $r(x)$ bits if it is stored, which happens with probability $p(x)$) and it contributes $p(x) + (1-p(x))2^{-r(x)}$ bits to the static function (with probability $p(x)$ it is a true positive of F, with probability $(1-p(x))2^{-r(x)}$ it is a false positive of F). The optimal choice of $r(x)$ depends on $p(x)$. We define

$$\text{space}(p, r) = pr + p + (1-p)2^{-r}$$
$$r^*_{\mathbb{R}_{\geq 0}}(p) = \arg\min_{r \geq 0} \text{space}(p, r)$$
$$r^*_{\mathbb{N}_0}(p) = \arg\min_{r \in \mathbb{N}_0} \text{space}(p, r)$$

When given a (hypothetical) space-optimal weighted filter that supports non-integer r, we should choose $r(x) = r^*_{\mathbb{R}_{\geq 0}}(p(x))$.[10] Given that our BuRR-based weighted filter only supports integer r, we choose $r(x) = r^*_{\mathbb{N}_0}(p(x))$. In Figure 4, we plot the resulting space overhead compared to $H(p)$ in both cases. Under the idealising assumptions we made, the space overhead of our approach compared to $H(p)$ is at most $\approx 10.8\%$.

Learned Relative Membership and Calibration. Now assume $p : U \to [0, \frac{1}{2}]$ arises from an ML model. We obtain a data structure for *learned relative membership* or simply a 1-bit LSF (to be generalised shortly).

The arguments just given, including the bound of $\approx 10.8\%$ on space overhead, still apply in this case if the underlying model is *calibrated*. Calibration means that out of all keys from X for which the model outputs a probability of roughly p, roughly a p-fraction are in fact in M. The average contribution of many keys to overall space consumption then behaves as predicted by our probabilistic analysis, even though M is not random. Our space goal, previously the entropy $\sum_{x \in X} H(p(x))$, is now formally the surprisal $\sum_{x \in M} \log_2(1/p(x)) + \sum_{x \in X \setminus M} \log_2(1/(1-p(x))$ of M given p. If p is calibrated, these quantities are (roughly) the same.

When using non-calibrated models, the space overhead of our data structure can be higher than predicted. The problem can be mostly mitigated by choosing $r(x)$ randomly (e.g. rounding $r^*_{\mathbb{R}_{\geq 0}}(p(x))$ up or down while preserving expectation). But given

[10] A (non-weighted) filter supporting non-integer r is described in [41, §4]. It is not quite space-optimal, though.

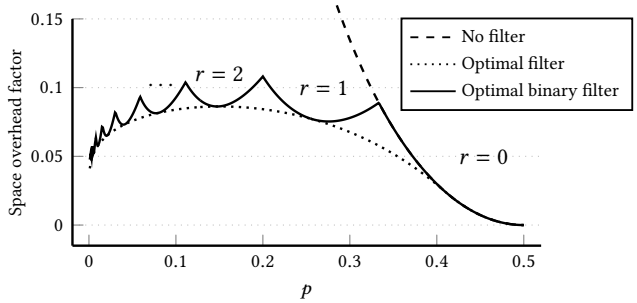


Figure 4: Idealised space overhead of filter-based WRM for keys with weight $p \in [0, \frac{1}{2}]$. *Optimal filter* shows the overhead $(\text{space}(p, r^*_{\mathbb{R}_{\geq 0}}(p)) - H(p))/H(p)$ when using a weighted filter that supports real-valued weights. The maximum is ≈ 0.086 at $p \approx 0.15$. *Optimal binary filter* shows the overhead $(\text{space}(p, r^*_{\mathbb{N}_0}(p)) - H(p))/H(p)$ when using a weighted filter that supports integer weights only. The maximum is ≈ 0.108 at $p = 0.2$.

that our loss function incentivises calibration and since we observe good calibration in our experiments, we have not investigated this issue in detail.

5.3 LSFs via Weighted Relative Membership: The Generalised Filter Trick

We now describe our construction of LSFs, first focusing on the conceptual level and then on efficient implementation.

Conceptual Level. Recall that for each key $k \in K$, we obtain a probability distribution $\mu_k = \mathcal{M}(k)$ on V from the ML model $\mathcal{M}$. For each k, we construct a code tree $T = T(k)$ and a corresponding codeword $c_1 \dots c_\ell \in \{0,1\}^*$ with $\ell = \ell(k)$ that describes how the leaf with label $f(k)$ is reached in T. Instead of storing $c_1, \dots, c_\ell$ in a VL-SF in plain form, we effectively apply the filter trick (explained in Section 2) at every node of T to save space whenever a binary decision is unbalanced.

Formally, for each node u of T, we define $u.p$ to be the combined probability (under μ_k) of all leaf labels in the subtree of u. If u is an inner node with children u_0 and u_1, this also gives probabilities $u_0.p/u.p$ of "going left" at u and $u_1.p/u.p$ of "going right" at u. In this sense, the binary decisions $c_1, \dots, c_\ell$ leading to $f(k)$ are associated with probabilities $p_1, \dots, p_\ell$. Note that the product $p = p_1 \cdot \ldots \cdot p_\ell$ is simply the probability of $f(k)$ under μ_k.

Let X be the set of $\sum_{k \in K} \ell(k)$ binary decisions relating to the keys from K. Let $M \subseteq X$ be the subset of those decisions that are made in the less likely way (breaking ties consistently). In principle, we can use the WRM data structure described in Section 5.2 to store M and recover the prefix code of any $k \in K$ using $\ell(k)$ queries. The amortised contribution of $k \in K$ to overall space is (under the assumptions we have made, including calibration) at most 10.8% more than the sum $\sum_{i=1}^{\ell(k)} \log_2(1/p_i)$ of the surprisal of the individual binary decisions. This sum is equal to $\log_2(1/p)$, which is the surprisal of $f(k)$ given μ_k. We have hence reached the goal we have set in the introduction (see Equation (1)).

Implementation. A remaining challenge is to implement this idea while interleaving the ℓ requests to the WRM to improve query efficiency. Recall that a WRM query for $x \in X$ first requests some number $r = r(x)$ of filter bits from a weighted filter and, if the filter bits match, another bit from a 1-bit SF to correct for false positives. An LSF query for $k \in K$ therefore involves a number $F = r_1 + \dots + r_\ell$ filter bits (r_i bits for the ith binary decision) and up to ℓ correction bits. The correction bits can simply be concatenated and stored using VL-BuRR. For the filter bits, we modify VL-BuRR to aggregate many weighted filter queries into a single query as follows.

At construction time, we specify for each $k \in K$ a bit string $s \in \{?,1\}^F$ where F depends on k. A query for $k \in K$ returns a random bitstring $t \in \{0,1\}^{F_{\max}}$ (for a fixed $F_{\max} \geq F$) obtained from s by replacing each ? with a random bit and appending $F_{\max} - F$ random bits. Recall that VL-BuRR distributes the task of storing a bitstring $f(k) \in \{0,1\}^F$ to F independent 1-bit SFs with interleaved storage. We can distribute $s \in \{?,1\}^F$ in the same way except that a "?" causes the corresponding SF to be skipped and not store anything. To ensure that a random bit is returned from a skipped SF, we use the standard trick of masking the stored bits with random bits.[11]

In Algorithm 3 and Algorithm 4, we show how our LSF can be built from these components. We suppress $w_{\max}$ and assume that VL-BuRR returns streams of output bits from which we read as needed. Note that at query time, the length ℓ of the prefix code of $f(k)$, the numbers $r_1, \dots, r_\ell$ of filter bits, and the number C of correction bits are only revealed gradually as we descend the code tree.

5.4 A Fast Prefix Code

When querying a key k, we construct a prefix code using the probability distribution μ_k given by the model. Preliminary experiments show that using optimal Huffman codes is expensive because they have to fully sort the value probabilities [42].

Shannon Codes. We identified Shannon codes [74] as a good compromise between construction time and prefix code lengths. One reason why Shannon is faster than Huffman is that we only sort the values by their code length $l(p) := \lceil -\log(p) \rceil$. The code length is directly determined by the exponent $e(p)$ of the floating point representation of the probabilities using $l(p) = -e(p)$. Similar to bucket sort, the values are directly assigned to buckets, where each bucket is responsible for one code length. Starting with the values of shortest code length, we assign the next available codeword of its desired length to each value as follows. Let $p_1, \dots, p_{|V|}$ be the probability of the values under μ_k, ordered by ascending code lengths. To compute the codeword for the i-th key, we first calculate $\sum_{j=1}^{i-1} 2^{-l(p_j)}$ and then use the leading $l(p_i)$ bits in the fractional part of this value as the code.

We have previously argued that our bound of roughly 10.8% on the space overhead holds regardless of the choice of prefix code. In a preliminary experiment using the datasets and models described in Section 6, we compare Shannon with Huffman codes and find that

[11] We use an additional hash function $g : U \to \{0,1\}^{F_{\max}}$. Before construction, we replace $s \in \{?,1\}^F$ with $s' \in \{?,0,1\}^F$ by replacing each 1 with the corresponding bit from $g(k)$. When a query would return $t' \in \{0,1\}^{F_{\max}}$, it instead returns $t \in \{0,1\}^{F_{\max}}$ that indicates bitwise equality of t' with $g(k)$ (with C-style operators: $t = t' \ \hat{} \ \sim g(k)$).

Algorithm 3: Construction of our LSF.

Input: K, $f : K \to V$
Output: LSF

Train $\mathcal{M}$ on f with loss function S
filterData $\leftarrow$ {} // empty dictionary
for $k \in K$ **do**
  $T \leftarrow$ code tree for $\mu_k = \mathcal{M}(k)$
  $c_1 \dots c_\ell \leftarrow$ codeword for $f(k)$ in T
  $(u_0, \dots, u_\ell) \leftarrow$ path to leaf with label $f(k)$ in T
  filterBits $\leftarrow$ ( )
  for $i = 1$ **to** ℓ **do**
    // $u_i.p$ is sum of the probabilities of values in u_i's subtree
    $p \leftarrow u_i.p / u_{i-1}.p$ // probability for c_i
    $r \leftarrow$ optimalBitLength$(p) \in \mathbb{N}_0$
    // add to filter if c_i is the less probable value
    filterBits.append(**if** $p < 0.5$ **then** 1^r **else** $?^r$)
  end
  filterData$[k] \leftarrow$ filterBits
end
// data structure aggregating weighted filter queries (see Sec. 5.3):
filterVL-SF $\leftarrow$ constructVL-BuRR*(filterData) // *specialised
correctionData $\leftarrow$ {} // empty dictionary
for $k \in K$ **do**
  $T \leftarrow$ code tree for $\mu_k = \mathcal{M}(k)$
  $c_1 \dots c_\ell \leftarrow$ codeword for $f(k)$ in T
  $(u_0, \dots, u_\ell) \leftarrow$ path to leaf with label $f(k)$ in T
  correctionBits $\leftarrow$ ( )
  filterStream $\leftarrow$ filterVL-SF.query(k)
  for $i = 1$ **to** ℓ **do**
    $p \leftarrow u_i.p / u_{i-1}.p$ // probability for c_i
    $r \leftarrow$ optimalBitLength$(p) \in \mathbb{N}_0$
    if filterStream.read$(r) = 1^r$ **then**
      correctionBits.append(c_i)
    end
  end
  correctionData$[k] \leftarrow$ correctionBits
end
correctionVL-SF $\leftarrow$ constructVL-BuRR(correctionData)
return $\mathcal{M}$, filterVL-SF, correctionVL-SF

Shannon codes increase the space by at most 1.03% while improving query time by 77% on average. Although our primary objective is space reduction, we include this optimisation because it achieves a substantial speedup at only a marginal space cost.

Descending an Implicit Code Tree. At query time, we descend the code tree and have to determine the probability of the next bit, i.e. the probability of going left or right (see Algorithm 4). All we need is the sorted sequence of values with associated codewords corresponding to the leaves of the code tree. A node u in the code tree is implicitly given by its depth d and two indices $1 \le i \le j \le |V|$ indicating a range of leaves. The probability of going left or right at u, as well as the index $s \in [i, j]$ separating the two subtrees, can

Algorithm 4: Query of our LSF.

Input: $k \in K$, LSF($\mathcal{M}$, filterVL-SF, correctionVL-SF)
Output: value $\in V$

$T \leftarrow$code tree for $\mu_k = \mathcal{M}(k)$
filterStream $\leftarrow$ filterVL-SF.query(k)
correctionStream $\leftarrow$ correctionVL-SF.query(k)
$u \leftarrow$ root of T
while $\neg u.isLeaf$ **do**
  $p \leftarrow u$.rightChild.$p / u.p$
  $r \leftarrow$ optimalBitLength$(p) \in \mathbb{N}_0$
  if filterStream.read$(r) = 1^r$ **then**
    $c \leftarrow$ correctionStream.read(1)
  else
    $c \leftarrow$ (**if** $p > 0.5$ **then** 1 **else** 0)
  end
  $u \leftarrow$ (**if** $c = 1$ **then** u.rightChild **else** u.leftChild)
end
return u.label

be computed with a linear scan of the range $[i, j]$. It is easy to see that this process takes $O(|V|)$ time.

Avoiding Code Construction. If one value has a probability $p >$ 50%, we can apply an optimisation during queries. In this case, we can try not to construct a prefix code initially. Because the probability is more than 50%, we immediately know that the code of the most likely value is "0" and that the probability that the first bit is "0" is simply p, and we therefore know how many bits of the variable-length filter code to look at. Only if we run into the unlikely case where the first bit turns out to be a "1", we do have to actually construct the prefix code. This optimisation is particularly helpful if the model often confidently predicts the correct symbol. Preliminary experiments show that this trick improves the query time by up to 54% on our datasets.

6 EXPERIMENTS

In this section, we evaluate the performance of our LSF on real-world and synthetic datasets. Our experiments show that:

- LSFs can leverage key-value correlations in the data to break the zero-order entropy barrier, achieving a space reduction between 37% and 94% (corresponding to 19.5× less space) compared to the best existing compressed static functions (CSF) on real-world datasets.
- Our auxiliary data structure $\mathcal{D}$ is highly space efficient, offering a multiplicative space overhead below 10.1% over the ideal space $S(\mathcal{M}, f)$.
- Query performance of LSFs remains practical, with at most 10.1 times slower queries compared to the best CSF.
- Even when using our LSF without learning, i.e. when using it as a CSF, it remains more space efficient compared to the best CSF competitor, which shows the effectiveness of our VL-BuRR structure and the generalised filter trick.

The code to reproduce our experiments is publicly available at https://github.com/gvinciguerra/LearnedStaticFunction.

Table 3: Dataset statistics.

Dataset	n	# Features	# Classes	H_0
covertype	581012	54	7	1.739
nids	1379274	10	10	2.376
songs	1159764	26	82	6.249
urls	235795	15	2	0.985
gauss	100 M	1	8	3.000

6.1 Experimental Setup

Hardware and Software. We use a Rocky Linux 9.5 machine with an Intel Core i7-11700 CPU with 64 GiB of DDR4 RAM. The CPU is set to a fixed clock frequency of 2.5 GHz. Each core has 48 KiB L1 and 512 KiB L2 data cache. We compile using clang 17 and compiler options `-march=native` and `-O3`.

We compare our LSF with the state-of-the-art compressed static function GOV [33] and the uncompressed static function BuRR [26]. The GOV and BuRR implementations are from the original authors. Note that the GOV construction is implemented in Java, but the queries are executed in C++. For BuRR, we use the configuration recommended for fast queries. Query time is measured on 10 million randomly chosen keys, and taking the average of 10 runs.

Datasets. We use 4 real-world datasets from various domains, differing in number of input features, output classes, and entropy of the values, along with 4 synthetic datasets. Table 3 summarises them.

- *covertype.* The task is to map a land to one of 7 forest cover types (such as pine, spruce, etc.) using 54 cartographic variables (elevation, soil type, distance to water, etc.) [10].
- *nids.* The task is to map a network flow to one of 10 cyber attack types (such as DDoS, ransomware, etc.) using 10 features extracted from packet captures (protocol, incoming and outgoing number of bytes or packets, etc.) [71].
- *songs.* The task is to map a song to one of 82 song genres using numerical audio features (such as tempo, loudness, instrumentalness, etc.) [44].
- *urls.* The task is to store 1 bit indicating whether a URL is legitimate or phishing using 15 features extracted from the URL itself (such as use of HTTPS, number of letters, digits, special characters, etc.) [66].
- *gauss.* The task in this synthetic dataset is to map a scalar value to one of 8 classes, where each class corresponds to a Gaussian distribution centred at a location along the real line. The class means are equally spaced with a distance of 2. The standard deviation σ controls the degree of overlap between the distributions, thus the hardness of the task. We vary $\sigma \in \{0.25, 0.5, 0.75, 1\}$. The number of samples is equal across all classes.

We keep manual preprocessing to a minimum, applying only one-hot encoding for categorical variables and standard scaling for numerical ones. We do not modify the real dataset distribution with class balancing techniques commonly used in ML, as they would trivially maximise the entropy and make CSFs ineffective.

Model Choice. For our LSF implementations, we employ lightweight models, namely logistic regression (LR), Gaussian Naive Bayes (GNB), and compact multi-layer perceptrons (MLPs), which, as we will show, offer both time and space efficiency. For MLP, we consider three configurations by varying the number of hidden layers L and hidden units U per layer as $(L, U) \in \{(1, 50), (1, 100), (2, 50)\}$. We use the softmax activation function for the output layer and ReLU for the hidden layers. We train the LR and MLP models in Keras using the Adam optimiser and minimise the surprisal $S(\mathcal{M}, f)$ as defined in Equation (1). Training stops if the surprisal did not improve by at least 1% within 3 epochs on the validation set.

The LR and MLP models are then exported to the TensorFlow Lite format and invoked from our LSF implementation in C++. In addition to experimenting with `float32` weights, we apply post-training quantisation to compress them to `float16` and `int8`.

For GNB, we use our own training and inference implementation in C++. We apply GNB only to the synthetic gauss dataset, as its generation process aligns directly with the inductive bias of GNB.

For all models, we report both accuracy and top-3 accuracy (for the multi-class datasets) on a hold-out test set comprising 20% of the data selected via stratified sampling. From the remaining training data, we use 10% as a validation set to enable early stopping during the training of LR and MLP models.

In our experiments, the model parameters and the training time are always accounted for in the performance of the LSF. However, in scenarios where a model is already available (e.g., for an upstream task), we note that it could be reused without incurring additional storage cost. Conversely, we observe that our training setup encourages model generalisation. This makes the resulting model potentially reusable across multiple LSFs on similar data, thus amortising both model training and storage costs. Interestingly, in additional experiments with early stopping based on the training loss (i.e., deliberately overfitting the data), we observed almost identical results, most likely because the small models do not have enough capacity to memorise the training data.

6.2 Internal Evaluation

As a first step of our evaluation, we identify the best-performing model of each dataset. We then integrate the model into our LSF and evaluate it as a whole.

Generally, all models captured the relation between keys and values, as the remaining surprisals $S(\mathcal{M}, f)$ are significantly below the zero-order entropy of the values of each dataset.

Weight Quantisation. We found that quantising the model weights to `float16` results in at most just 0.1% higher surprisal $S(\mathcal{M}, f)$ than `float32` for all datasets and models. At the same time, the model requires about half the size $|enc(\mathcal{M})|$. The `int8` quantisation results in an even smaller model but has a drastically worse performance in terms of overall space $\Sigma := S(\mathcal{M}, f) + |enc(\mathcal{M})|$, which is up to 149% higher. Regarding inference time, `float16` quantisation is 9% faster than `float32` on average; instead, `int8` is 43% slower than `float32`, likely because our models are shallow and any potential speedup is offset by type conversion overhead. We therefore use the `float16` quantisation henceforth.

Model Performance. Table 4 shows the training time, accuracy, space consumption, and inference time of the `float16` models in detail. On the real-world covertype, nids, and songs datasets, the

Table 4: Model performance, quantised to `float16`. Training and inference time are measured in µs per key, and space consumption in bits per key. For each dataset, models are ordered by total space $\Sigma = S(\mathcal{M}, f) + |enc(\mathcal{M})|$.

Dataset (H_0)	Model			Training	Accuracy		Space			Inference
	Layers	Units	# Param		Top	Top-3	$S(\mathcal{M}, f)/n$	$\lvert enc(\mathcal{M})\rvert/n$	Σ/n	
covertype (1.74)	2	50	5657	238	87.4	99.9	0.4361	0.1558	0.5919	1.958
	1	100	6207	245	85.2	99.9	0.5133	0.1709	0.6842	1.788
	1	50	3107	182	81.8	99.8	0.6202	0.0856	0.7057	1.414
	LR		385	57	72.2	99.3	0.9172	0.0106	0.9278	0.779
nids (2.38)	2	50	3610	63	70.2	94.2	1.0578	0.0419	1.0997	1.955
	1	50	1060	59	69.4	93.3	1.1020	0.0123	1.1143	1.406
	1	100	2110	59	69.5	93.6	1.0931	0.0245	1.1176	1.742
	LR		110	43	66.4	91.4	1.2762	0.0013	1.2775	0.773
songs (6.25)	2	50	8082	62	29.4	50.4	3.8438	0.1115	3.9553	2.483
	1	50	5532	60	28.5	49.1	3.9243	0.0763	4.0006	1.883
	1	100	10982	62	29.4	50.3	3.8526	0.1515	4.0041	2.601
	LR		2214	35	23.8	43.0	4.3365	0.0305	4.3671	1.346
urls (0.99)	LR		16	205	99.4	–	0.0448	0.0011	0.0459	0.403
	1	50	851	93	99.7	–	0.0243	0.0577	0.0820	1.114
	1	100	1701	75	99.6	–	0.0244	0.1154	0.1398	1.526
	2	50	3401	38	99.6	–	0.0253	0.2308	0.2561	1.671
gauss σ=0.25 (3.0)	GNB		16	<1	100.0	–	0.0002	<0.0001	0.0002	0.190
gauss σ=0.5 (3.0)	GNB		16	<1	96.0	–	0.1526	<0.0001	0.1526	0.208
gauss σ=0.75 (3.0)	GNB		16	<1	84.0	–	0.5579	<0.0001	0.5579	0.215
gauss σ=1 (3.0)	GNB		16	<1	72.2	–	0.9139	<0.0001	0.9139	0.192

MLP model with 2 hidden layers and 50 hidden units performed best in overall space consumption Σ. Interestingly, the songs dataset is the most challenging in terms of learnability, as evident from its lower model accuracies and higher surprisal, and yet Σ is up to 34.4% lower than H_0, which highlights the effectiveness of LSFs even in scenarios where the model predictions are less confident. In contrast, the real-world urls dataset is highly learnable: all models predict the correct values with high confidence, as evidenced by their high accuracy and the low values of $S(\mathcal{M}, f)$. In such cases, the model size $|enc(\mathcal{M})|$ could become the dominant term in the total space Σ. Accordingly, the simple LR model performed best here, despite having a value $S(\mathcal{M}, f)$ nearly twice as high as that of more complex models. On all datasets, we observe differences in training and inference times across models. Hence, selecting a model entails a trade-off between space and time that directly affects the performance of the overall LSF. In the subsequent evaluation of the overall LSF (Section 6.3 and Table 5), we select for each dataset both the model that minimises Σ and the one that minimises the inference time, thereby illustrating both ends of the trade-off.

Model Calibration. In Section 5.2, we discussed that the space efficiency of WRM (underlying our LSFs) relies on a well-calibrated model. A model is well-calibrated if events of predicted probability p actually occur a p fraction of the time. A common metric to measure the calibration of a model is the expected calibration error (ECE) [23, 60]. An ECE of 0% indicates a perfectly calibrated model, while an ECE of 100% indicates full miscalibration. We measured an ECE of at most 0.6% for the model of each dataset (using 50 equally-sized bins, see [60] for details).

Structure Overheads. Besides the model $\mathcal{M}$, another part of our LSF is an auxiliary data structure $\mathcal{D}$ that intuitively makes up for what $\mathcal{M}$ does not know about $f : K \to V$ (see Section 5). Ideally, this internal data structure would require about $S(\mathcal{M}, f)$ bits of space. However, as Table 5 shows, it requires slightly more space due to two main factors: (i) the WRM data structure underlying our LSF has an overhead of up to 10.8% (as determined in Section 5.2, assuming $\mathcal{M}$ is calibrated), and (ii) VL-BuRR has additional overhead, e.g. for storing bumping data, amounting to at most 1.2% on large inputs. To measure the overhead introduced by the internal data structure, we compare its space with the ideal space $S(\mathcal{M}, f)$. On most datasets, the overhead is below 10.1%, indicating that the worst-case overhead of 10.8% rarely occurs in practice. Exceptions to the overhead of 10.1% occur for the gauss $\sigma = 0.25$ and the urls dataset, in which the overhead is 133% and 14.8%, respectively. In both cases, the LSFs are so effective that their overall size is less than 56 KB, making modest constant overheads of the data structure (such as internal pointers and counters) appear large, while in fact remaining negligible in absolute terms (see also Figure 5).

Construction. The construction time strongly depends on the training complexity of the model. For simple models like GNB, training times are below 0.004 µs per key, which is negligible in the overall construction of the LSFs. Complex models like MLP (L2 U50) on the covertype dataset result in training times as high as

Table 5: Comparison with competitors. LSF construction time includes training, space usage includes the ML model (quantised to `float16`), and query time includes model inference. From Table 4, we select models minimising space Σ or inference time. L and U denote the number of hidden layers and hidden units. "Ours (CSF)" refers to our LSF without learning, i.e. the model outputs the value frequencies. On the gauss dataset, non-learned approaches perform independently of σ and are thus reported once for "any σ".

Dataset (H_0)	Competitor	Constr. µs/key	Space bits/key	Query µs/key
covertype (1.74)	**Ours** (L2 U50)	241.980	0.6360	2.212
	Ours (LR)	58.857	1.0050	1.023
	Ours (CSF)	0.658	1.8020	0.125
	GOV	1.613	1.9404	0.342
	BuRR	0.137	3.0471	0.021
nids (2.38)	**Ours** (L2 U50)	67.449	1.1573	2.290
	Ours (LR)	45.085	1.3564	1.084
	Ours (CSF)	0.795	2.4451	0.149
	GOV	1.801	2.5643	0.244
	BuRR	0.138	4.0541	0.029
songs (6.25)	**Ours** (L2 U50)	71.538	4.1005	4.569
	Ours (LR)	42.595	4.5231	3.522
	Ours (CSF)	1.751	6.3102	0.156
	GOV	13.683	6.5181	0.269
	BuRR	0.136	7.0823	0.040
urls (0.99)	**Ours** (LR)	206.440	0.0526	0.561
	Ours (CSF)	0.536	1.0180	0.052
	GOV	2.042	1.0941	0.186
	BuRR	0.128	1.0266	0.011
gauss σ=0.25 (3.0)	**Ours** (GNB)	0.448	0.0006	0.352
gauss σ=0.5 (3.0)	**Ours** (GNB)	0.618	0.1665	0.436
gauss σ=0.75 (3.0)	**Ours** (GNB)	0.840	0.6066	0.500
gauss σ=1 (3.0)	**Ours** (GNB)	0.936	0.9845	0.513
gauss any σ (3.0)	**Ours** (CSF)	1.374	3.0159	0.141
	GOV	2.180	3.1485	0.202
	BuRR	0.136	3.0454	0.053

238 µs per key and dominate the overall LSF construction. Training time can be significantly reduced through standard ML techniques like smaller training sets, increased batch sizes, early stopping, etc., and is of secondary concern for our algorithmic evaluation of LSFs.

Queries. Model inference time makes up between 38% and 89% of the total query time on the most space efficient models and at most 72% on the fastest models. Overall, the query time of our LSF ranges from 0.352 to 4.569 µs per key across all datasets (see Table 5). The query time positively correlates with the number of classes and $S(\mathcal{M}, f)/n$ (see Table 5 and Figure 5). This is due to the increased inference time required to compute probabilities for more classes and the higher cost of constructing the prefix code in each query. Finally, the traversal time of the code tree positively correlates with the average code length which is roughly $S(\mathcal{M}, f)/n$.

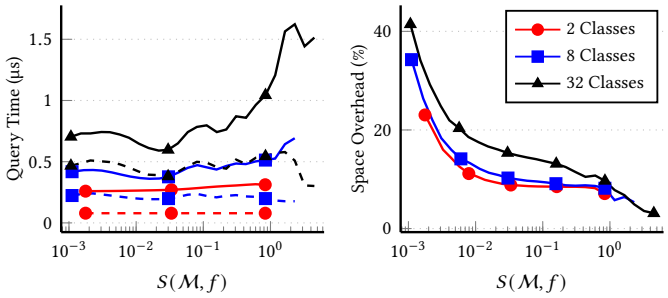


Figure 5: Query time, inference time (dashed) and space overhead on the gauss dataset varying # classes and $S(\mathcal{M}, f)$.

6.3 Comparison With Competitors

In Table 5, we compare the performance of our LSF with the CSF GOV [33] and the r-bit SF BuRR [26], which are both introduced in Section 2. The table also includes a new CSF based on our LSF, which we compare against GOV at the end of this section.

Qualitatively, we find that our LSF significantly outperforms GOV and BuRR in terms of space consumption. This is not surprising, because the space of our LSF undercuts the zero-order empirical entropy H_0 of the value sequence on all datasets, while the space of GOV is lower bounded by H_0. The r-bit SF BuRR has an even higher lower bound of $r = \lceil \log(|V|) \rceil$ bits per key. Looking at specific datasets, our LSF is particularly effective on the urls and the synthetic gauss datasets. Our LSF achieves a space reduction by at least 94.9% compared to BuRR and GOV on the urls dataset. Using the most space efficient model, space reduction is 67.2% and 54.9% compared to the closest competitor on the covertype and nids dataset, respectively. The songs dataset has the largest number of classes (82) and is the hardest to learn in the sense that $S(\mathcal{M}, f)$ is the closest to the value entropy. Still, our LSF results in 37.1% less space compared to the closest competitor.

On the synthetic gauss dataset, we varied the variance σ of the 8 classes. Since the class means are fixed at a distance of 2, lower variances σ allow the model to be more confident about the value of a key. For $\sigma = 1$, our LSF requires 67.7% less space compared to the closest competitor. Pushing σ down to 0.25, we achieve a space reduction of 99.98% compared to the closest competitor, i.e. the closest competitor requires 5448× as much space as our LSF.

Construction. In terms of construction time, our LSF is at most 151 times slower compared to GOV and 1766 times slower than r-bit BuRR across all datasets. Training by far dominates the construction time of our LSF. However, we remark that on larger datasets or in tasks that can reuse an existing model (e.g. upstream tasks), training times might not be an issue. Some scenarios might also allow for simpler models. One example of this is the gauss dataset, where we use GNB. GNB has a fast training time because it only has to determine the mean and variance of each value. On all gauss datasets, we are even faster in construction than GOV with a speedup of at least 1.37×. This is also because of the high compression that we can achieve, resulting in only a small amount of data that the LSF has to encode.

Queries. In terms of queries, our LSF has to perform model inference and constructs a prefix code for each query. GOV has no model and constructs the prefix code only once. BuRR even uses fixed-length codes. The slower query speed of our LSF is therefore expected. On all datasets, query times are at most 17.0× slower than GOV and 114× slower than BuRR. The query time of our LSF on the gauss dataset $\sigma = 0.25$ is close to the GOV competitor. Arguably, we need to store much less data, resulting in better cache efficiency.

Our CSF. We replaced the machine learning model of our LSF with one that outputs the value frequencies, independently of the key. Our LSF then becomes a CSF, allowing us to evaluate the performance of our generalised filter trick and our VL-BuRR data structure without the learning part. Instead of constructing per-key prefix codes on demand, we build a global Huffman code shared by all keys, improving query performance. The space of our data structure, like any CSF, is now lower bounded by the value entropy. The resulting performance is reported in Table 5 under the name "Ours (CSF)". On all datasets, our CSF achieves space usage within 3.6% of the value entropy, with the VL-BuRR component contributing no more than 1.8% to this overhead. The remaining space overhead is due to sub-optimal coding, which we further reduce compared to GOV using our generalised filter trick. Our CSF outperforms the state-of-the-art GOV competitor in all three metrics simultaneously on all datasets. We achieve a space reduction as high as 7.1% compared to GOV on the covertype dataset.

Note that our CSF outperforms the BuRR competitor in space on the urls and gauss datasets, where $H_0 \approx \lceil \log_2(|V|) \rceil$, even though we also use BuRR internally. This is because we use a space-efficient BuRR configuration internally, while the competitor uses a faster configuration, as speed is its main selling point in our comparison.

7 RELATED WORK

Several learning-based approaches have been proposed for data structure design and lossless data compression. Here, we mention approaches that are conceptually related to LSF.

DeepMapping [80] memorises a key-value map with a neural network and stores the misclassified pairs in key-sorted partitions, which are then compressed with a standard general-purpose compressor. Hence, unlike an LSF, it explicitly stores the keys and does not exploit the model's output probabilities for compression.

Sequential neural compressors [24, 35, 73] compress sequences such as texts by combining arithmetic coding with a neural network (e.g. a language model) that predicts a probability distribution for the next symbol given the preceding context. As such, they do not provide a randomly-accessible key-value map.

Some integer compressors approximate the input sequence with a regression model and encode the residuals, either supporting random access [2, 12, 28, 38, 56] or not [76, 77]. Unlike LSFs, they neither use a probabilistic classifier nor provide a key-value map.

Learned Bloom filters (LBFs) [20, 21, 49, 57, 58, 63, 67, 69, 72, 78] train a binary classifier to accept keys from a given set $K \subseteq U$ and reject keys from $U \setminus K$, and use a backup Bloom filter to eliminate false negatives, i.e. keys from K that the classifier incorrectly rejects. LSFs are more general than LBFs in that they map keys to arbitrary values rather than just a binary membership. Even in the binary case, LSFs differ from LBF in that they guarantee no false positives on a given set of non-keys from $U \setminus K$. Learned Functional Bloom filters (LFBF) [16] extend LBFs to return a value associated with each key, rather than only approximate membership information. However, they introduce indeterminables, i.e. queries for which the data structure cannot determine a value at all. Thus, LFBFs offer different guarantees than LSFs.

Perfect hash functions [47, 54] (PHFs) are related to SFs as they also do not need to store the keys. A PHF for a set K of size n maps the keys from K to a range $\{1, \ldots, m\}$ for some $m \geq n$ without collisions. An SF can be derived directly from a PHF [30, 39]. Conversely, SFs have been used to construct PHFs [14, 52, 53]. Monotone Minimal PHFs (MMPHFs) specialise PHFs by mapping each key in K to its rank in $\{1, \ldots, n\}$ [3, 6, 27]. MMPHFs have also been studied in the learned setting [27], where a learned index [29] is used as a rank estimator. A different line of work replaces traditional hash functions with learned models in standard hash tables [49, 70]. However, hash-based indexing techniques do not address the compression of values associated with keys.

8 CONCLUSION

We have introduced the concept of a *learned static function* (LSF). Like a compressed static function (CSF), it is meant to store key-value assignments in contexts where some values are more likely than others. *Unlike* CSFs, however, the prediction can be based on the requested key itself rather than merely on the global value frequencies in the data set. While the space of CSFs cannot beat the zero-order entropy of the values, our LSFs can be much smaller if there is a learnable correlation between keys and values.

In principle, there is no limit on how much more space efficient an LSF can be compared to a CSF (for the synthetic Gauss dataset, the LSF is smaller by a factor of ≈5000). We show experimentally that this promise is borne out *in practice*: in a real-world dataset with annotated URLs, we save a factor of ≈19.

We pay for this reduced space with increased query times, mostly due to the inference time of the used machine learning model. Whether or not this is acceptable depends on the application context, as does the question of how much the problem can be mitigated, say by relying on simpler heuristics with smaller inference times.

Technically, our LSF construction relies on a combination of machine learning, fast per-key prefix codes, efficient variable-length static functions (VL-SF), and the clever use of weighted Bloom filters. The VL-SF in particular may be of independent interest.

We have suggested potential applications of LSFs in databases, natural language processing, endgame tablebases and bioinformatics. We are curious to see which of these application areas bear fruit and which further directions we may not have anticipated.

ACKNOWLEDGMENTS

This work was supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF). GV was supported by the NextGenerationEU – National Recovery and Resilience Plan (Piano Nazionale di Ripresa e Resilienza, PNRR) – Project: "SoBigData.it - Strengthening the Italian RI for Social Mining and Big Data Analytics" – Prot. IR0000013 – Avviso n. 3264 del 28/12/2021. We thank Matthias Becht for working on the implementation of VL-BuRR.

REFERENCES

[1] Stephen Alstrup, Gerth Brodal, and Theis Rauhe. 2001. Optimal static range reporting in one dimension. In *Proc. 33rd Annual ACM Symposium on Theory of Computing (STOC)*. 476–482.

[2] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. 2011. Efficient Parallel Lists Intersection and Index Compression Algorithms Using Graphics Processing Units. *PVLDB* 4, 8 (2011), 470–481. https://doi.org/10.14778/2002974.2002975

[3] Sepehr Assadi, Martin Farach-Colton, and William Kuszmaul. 2023. Tight Bounds for Monotone Minimal Perfect Hashing. In *Proc. 34th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 456–476. https://doi.org/10.1137/1.9781611977554.CH20

[4] Shivnath Babu, Minos N. Garofalakis, and Rajeev Rastogi. 2001. SPARTAN: A Model-Based Semantic Compression System for Massive Data Tables. In *Proc. 28th ACM International Conference on Management of Data (SIGMOD)*. 283–294. https://doi.org/10.1145/375663.375693

[5] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2010. Fast prefix search in little space, with applications. In *Proc. 18th Annual European Symposium on Algorithms (ESA)*. 427–438.

[6] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2011. Theory and practice of monotone minimal perfect hashing. *ACM J. Exp. Algorithmics* 16, Article 3.2 (2011), 26 pages. https://doi.org/10.1145/1963190.2025378

[7] Djamal Belazzougui and Rossano Venturini. 2013. Compressed static functions with applications. In *Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 229–240. https://doi.org/10.1137/1.9781611973105.17

[8] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Iceberg Hashing: Optimizing Many Hash-Table Criteria at Once. *J. ACM* 70, 6 (2023), 40:1–40:51. https://doi.org/10.1145/3625817

[9] Ioana O. Bercea, Jakob Bæk Tejs Houen, and Rasmus Pagh. 2024. Daisy Bloom Filters. In *Proc. 19th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*. 9:1–9:19. https://doi.org/10.4230/LIPICS.SWAT.2024.9

[10] Jock Blackard. 1998. Covertype. UCI Machine Learning Repository. https://doi.org/10.24432/C50K5N

[11] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[12] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2022. A Learned Approach to Design Compressed Rank/Select Data Structures. *ACM Trans. on Algorithms* 18, 3, Article 24 (oct 2022), 28 pages. https://doi.org/10.1145/3524060

[13] Fabiano C. Botelho, Yoshiharu Kohayakawa, and Nivio Ziviani. 2005. A Practical Minimal Perfect Hashing Method. In *Proc. 4th International Workshop on Experimental and Efficient Algorithms (WEA)*. 488–500. https://doi.org/10.1007/11427186_42

[14] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2013. Practical Perfect Hashing in Nearly Optimal Space. *Inf. Syst.* 38, 1 (2013), 108–131. https://doi.org/10.1016/j.is.2012.06.002

[15] Jehoshua Bruck, Jie Gao, and Anxiao Jiang. 2006. Weighted Bloom filter. In *Proc. 39th IEEE International Symposium on Information Theory (ISIT)*. 2304–2308. https://doi.org/10.1109/ISIT.2006.261978

[16] Hayoung Byun and Hyesook Lim. 2022. Learned FBF: Learning-Based Functional Bloom Filter for Key-Value Storage. *IEEE Trans. Computers* 71, 8 (2022), 1928–1938. https://doi.org/10.1109/TC.2021.3112079

[17] Seth Carbon et al. 2021. The Gene Ontology resource: enriching a GOld mine. *Nucleic acids research* 49, D1 (2021), D325–D334. https://doi.org/10.1093/NAR/GKAA1113

[18] Benjamin Coleman, David Torres Ramos, Vihan Lakshman, Chen Luo, and Anshumali Shrivastava. 2023. CARAMEL: A Succinct Read-Only Lookup Table via Compressed Static Functions. arXiv:2305.16545 [cs.DS] https://arxiv.org/abs/2305.16545

[19] Colin Cooper. 2000. On the rank of random matrices. *Random Struct. Algorithms* 16, 2 (2000), 209–232. https://doi.org/10.1002/(SICI)1098-2418(200003)16:2%3C209::AID-RSA6%3E3.0.CO;2-1

[20] Zhenwei Dai and Anshumali Shrivastava. 2020. Adaptive Learned Bloom Filter (Ada-BF): Efficient Utilization of the Classifier with Application to Real-Time Information Filtering on the Web. In *Proc. 33rd Annual Conference on Neural Information Processing Systems (NeurIPS)*. https://proceedings.neurips.cc/paper/2020/hash/86b94dae7c6517ec1ac767fd2c136580-Abstract.html

[21] Zhenwei Dai, Anshumali Shrivastava, Pedro Reviriego, and José Alberto Hernández. 2022. Optimizing Learned Bloom Filters: How Much Should Be Learned? *IEEE Embed. Syst. Lett.* 14, 3 (2022), 123–126. https://doi.org/10.1109/LES.2022.3156019

[22] Scott Davies and Andrew W. Moore. 1999. Bayesian Networks for Lossless Dataset Compression. In *Proc. 5th ACM International Conference on Knowledge Discovery and Data Mining, (KDD)*. 387–391. https://doi.org/10.1145/312129.312289

[23] Morris H DeGroot and Stephen E Fienberg. 1982. Assessing Probability Assessors: Calibration and Refinement. In *Statistical Decision Theory and Related Topics III*, Vol. 1. 291–314.

[24] Grégoire Delétang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness. 2024. Language Modeling Is Compression. In *Proc. 12th International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=jznbgiynus

[25] Martin Dietzfelbinger and Rasmus Pagh. 2008. Succinct Data Structures for Retrieval and Approximate Membership (Extended Abstract). In *Proc. 35th International Colloquium on Automata, Languages, and Programming (ICALP)*. 385–396. https://doi.org/10.1007/978-3-540-70575-8_32

[26] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. 2022. Fast Succinct Retrieval and Approximate Membership Using Ribbon. In *Proc. 20th International Symposium on Experimental Algorithms (SEA)*. 4:1–4:20. https://doi.org/10.4230/LIPICS.SEA.2022.4

[27] Paolo Ferragina, Hans-Peter Lehmann, Peter Sanders, and Giorgio Vinciguerra. 2023. Learned monotone minimal perfect hashing. In *Proc. 31st Annual European Symposium on Algorithms (ESA)*. 46:1–46:17. https://doi.org/10.4230/LIPIcs.ESA.2023.46

[28] Paolo Ferragina, Giovanni Manzini, and Giorgio Vinciguerra. 2022. Compressing and Querying Integer Dictionaries Under Linearities and Repetitions. *IEEE Access* 10 (2022), 118831–118848. https://doi.org/10.1109/ACCESS.2022.3221520

[29] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB* 13, 8 (2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[30] Michael L. Fredman, János Komlós, and Endre Szemerédi. 1984. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *J. ACM* 31, 3 (1984), 538–544. https://doi.org/10.1145/828.1884

[31] Robert G. Gallager. 1978. Variations on a theme by Huffman. *IEEE Trans. Inf. Theory* 24, 6 (1978), 668–674. https://doi.org/10.1109/TIT.1978.1055959

[32] Yihan Gao and Aditya G. Parameswaran. 2016. Squish: Near-Optimal Compression for Archival of Relational Datasets. In *Proc. 22nd ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 1575–1584. https://doi.org/10.1145/2939672.2939867

[33] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. 2020. Fast scalable construction of ([compressed] static | minimal perfect hash) functions. *Inf. Comput.* 273 (2020), 104517. https://doi.org/10.1016/J.IC.2020.104517

[34] Dave Gomboc, Christian R. Shelton, Andrew S. Miner, and Gianfranco Ciardo. 2024. Comparing Lossless Compression Methods for Chess Endgame Data. In *Proc. 27th European Conference on Artificial Intelligence (ECAI)*, Vol. 392. IOS Press, 4116–4123. https://doi.org/10.3233/FAIA240982

[35] Mohit Goyal, Kedar Tatwawadi, Shubham Chandak, and Idoia Ochoa. 2019. DeepZip: Lossless Data Compression Using Recurrent Neural Networks. In *Proc. 29th Data Compression Conference (DCC)*. 575. https://doi.org/10.1109/DCC.2019.00087

[36] Thomas Mueller Graf and Daniel Lemire. 2020. Xor Filters. *ACM J. Exp. Algorithmics* 25 (2020), 1–16. https://doi.org/10.1145/3376122

[37] Thomas Mueller Graf and Daniel Lemire. 2022. Binary Fuse Filters: Fast and Smaller Than Xor Filters. *ACM J. Exp. Algorithmics* 27 (2022), 1.5:1–1.5:15. https://doi.org/10.1145/3510449

[38] Andrea Guerra, Giorgio Vinciguerra, Antonio Boffa, and Paolo Ferragina. 2025. Learned Compression of Nonlinear Time Series With Random Access. In *Proc. 41st IEEE International Conference on Data Engineering (ICDE)*. 1579–1592. https://doi.org/10.1109/ICDE65448.2025.00122

[39] Torben Hagerup and Torsten Tholey. 2001. Efficient Minimal Perfect Hashing in Nearly Minimal Space. In *Proc. 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, Vol. 2010. 317–326. https://doi.org/10.1007/3-540-44693-1_28

[40] Stefan Hermann. 2025. MorphisHash: Improving Space Efficiency of ShockHash for Minimal Perfect Hashing. In *Proc. 33rd Annual European Symposium on Algorithms (ESA)*, Vol. 351. 9:1–9:16. https://doi.org/10.4230/LIPIcs.ESA.2025.9

[41] Jóhannes B. Hreinsson, Morten Krøyer, and Rasmus Pagh. 2009. Storing a Compressed Function with Constant Time Access. In *Proc. 17th Annual European Symposium on Algorithms (ESA)*. 730–741. https://doi.org/10.1007/978-3-642-04128-0_65

[42] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[43] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Cetintemel. 2020. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *Proc. 46th ACM International Conference on Management of Data (SIGMOD)*. 1733–1746. https://doi.org/10.1145/3318464.3389734

[44] Amitansh Joshi, Amit Parolkar, and Vedant Das. 2023. Spotify_1Million_Tracks. https://doi.org/10.34740/KAGGLE/DSV/5987852

[45] Minoru Kanehisa, Miho Furumichi, Yoko Sato, Mari Ishiguro-Watanabe, and Mao Tanabe. 2021. KEGG: integrating viruses and cellular organisms. *Nucleic acids research* 49, D1 (2021), D545–D551. https://doi.org/10.1093/NAR/GKAA970

[46] Sándor Z. Kiss, Éva Hosszu, János Tapolcai, Lajos Rónyai, and Ori Rottenstreich. 2021. Bloom Filter With a False Positive Free Zone. *IEEE Trans. Netw. Serv.*

Manag. 18, 2 (2021), 2334–2349. https://doi.org/10.1109/TNSM.2021.3059075

[47] Donald E Knuth. 1998. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional.

[48] Dominik Köppl, Simon J. Puglisi, and Rajeev Raman. 2022. Fast and Simple Compact Hashing via Bucketing. *Algorithmica* 84, 9 (2022), 2735–2766. https://doi.org/10.1007/S00453-022-00996-Y

[49] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proc. 44th ACM International Conference on Management of Data (SIGMOD)*. 489–504. https://doi.org/10.1145/3183713.3196909

[50] M. Oguzhan Külekci. 2014. Enhanced Variable-Length Codes: Improved Compression with Efficient Random Access. In *Proc. 24th Data Compression Conference (DCC)*. 362–371. https://doi.org/10.1109/DCC.2014.74

[51] Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. 2023. PaCHash: Packed and compressed hash tables. In *Proc. 25th Symposium on Algorithm Engineering and Experiments (ALENEX)*. 162–175. https://doi.org/10.1137/1.9781611977561.ch14

[52] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. 2023. SicHash - Small Irregular Cuckoo Tables for Perfect Hashing. In *Proc. 25th Symposium on Algorithm Engineering and Experiments (ALENEX)*. 176–189. https://doi.org/10.1137/1.9781611977561.ch15

[53] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. 2024. ShockHash: Towards Optimal-Space Minimal Perfect Hashing Beyond Brute-Force. In *Proc. 26th Symposium on Algorithm Engineering and Experiments (ALENEX)*. 194–206. https://doi.org/10.1137/1.9781611977929.15

[54] Hans-Peter Lehmann, Thomas Mueller, Rasmus Pagh, Giulio Ermanno Pibiri, Peter Sanders, Sebastiano Vigna, and Stefan Walzer. 2025. Modern Minimal Perfect Hashing: A Survey. *CoRR* abs/2506.06536 (2025). https://doi.org/10.48550/ARXIV.2506.06536

[55] Hanwen Liu, Mihail Stoian, Alexander van Renen, and Andreas Kipf. 2024. Corra: Correlation-Aware Column Compression. In *Proc. 2nd Workshop on Cloud Databases (CloudDB)*. https://vldb.org/workshops/2024/proceedings/CloudDB/clouddb-2.pdf

[56] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proc. ACM Manag. Data* 2, 1, Article 65 (mar 2024), 28 pages. https://doi.org/10.1145/3639320

[57] Dario Malchiodi, Davide Raimondi, Giacomo Fumagalli, Raffaele Giancarlo, and Marco Frasca. 2024. The role of classifiers and data complexity in learned Bloom filters: insights and recommendations. *J. Big Data* 11, 1 (2024), 45. https://doi.org/10.1186/S40537-024-00906-9

[58] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *Proc. 31st Annual Conference on Neural Information Processing Systems (NeurIPS)*. 462–471. https://proceedings.neurips.cc/paper/2018/hash/0f49c89d1e7298bb9930789c8ed59d48-Abstract.html

[59] Christian Worm Mortensen, Rasmus Pagh, and Mihai Pătraşcu. 2005. On dynamic range reporting in one dimension. In *Proc. 37th annual ACM Symposium on Theory of Computing (STOC)*. 104–111.

[60] Mahdi Pakdaman Naeini, Gregory F. Cooper, and Milos Hauskrecht. 2015. Obtaining Well Calibrated Probabilities Using Bayesian Binning. In *Proc. 29th AAAI Conference on Artificial Intelligence*. AAAI Press, 2901–2907. https://doi.org/10.1609/AAAI.V29I1.9602

[61] Gonzalo Navarro. 2016. *Compact Data Structures: A Practical Approach*. Cambridge University Press.

[62] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* 51, 2 (2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[63] Ripon Patgiri, Anupam Biswas, and Sabuzima Nayak. 2023. deepBF: Malicious URL detection using learned Bloom Filter and evolutionary deep learning. *Comput. Commun.* 200 (2023), 30–41. https://doi.org/10.1016/J.COMCOM.2022.12.027

[64] Mihai Pătraşcu. 2008. Succincter. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 305–313. https://doi.org/10.1109/FOCS.2008.83

[65] Ely Porat. 2009. An Optimal Bloom Filter Replacement Based on Matrix Solving. In *Proc. 4th International Computer Science Symposium in Russia (CSR)*. 263–273. https://doi.org/10.1007/978-3-642-03351-3_25

[66] Arvind Prasad and Shalini Chandra. 2024. PhiUSIIL: A diverse security profile empowered phishing URL detection framework based on similarity index and incremental learning. *Comput. Secur.* 136 (2024), 103545. https://doi.org/10.1016/J.COSE.2023.103545 Dataset available at https://archive.ics.uci.edu/dataset/967/phiusiil+phishing+url+dataset.

[67] Jack W. Rae, Sergey Bartunov, and Timothy P. Lillicrap. 2019. Meta-Learning Neural Bloom Filters. In *Proc. 36th International Conference on Machine Learning (ICML)*. 5271–5280. http://proceedings.mlr.press/v97/rae19a.html

[68] Julian Reichinger, Thomas Krismayer, and Jan Rellermeyer. 2024. COPR – Efficient, large-scale log storage and retrieval. arXiv:2402.18355 [cs.IR] https://arxiv.org/abs/2402.18355

[69] Pedro Reviriego, José Alberto Hernández, Zhenwei Dai, and Anshumali Shrivastava. 2021. Learned Bloom Filters in Adversarial Environments: A Malicious URL Detection Use-Case. In *Proc. 22nd IEEE International Conference on High Performance Switching and Routing (HPSR)*. 1–6. https://doi.org/10.1109/HPSR52026.2021.9481857

[70] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. 2022. Can Learned Models Replace Hash Functions? *PVLDB* 16, 3 (2022), 532–545. https://doi.org/10.14778/3570690.3570702

[71] Mohanad Sarhan, Siamak Layeghy, Nour Moustafa, and Marius Portmann. 2020. NetFlow Datasets for Machine Learning-Based Network Intrusion Detection Systems. In *Proc. 10th EAI International Conference on Big Data Technologies and Applications (LNICST)*, Vol. 371. 117–135. https://doi.org/10.1007/978-3-030-72802-1_9 Dataset available at https://staff.itee.uq.edu.au/marius/NIDS_datasets/.

[72] Atsuki Sato and Yusuke Matsui. 2023. Fast Partitioned Learned Bloom Filter. In *Proc. 36th Annual Conference on Neural Information Processing Systems (NeurIPS)*. http://papers.nips.cc/paper_files/paper/2023/hash/7b2e844c52349134268e819a9b56b9e8-Abstract-Conference.html

[73] Jürgen Schmidhuber and Stefan Heil. 1996. Sequential neural text compression. *IEEE Trans. Neural Networks* 7, 1 (1996), 142–146. https://doi.org/10.1109/72.478398

[74] Claude E Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

[75] Yoshihiro Shibuya, Djamal Belazzougui, and Gregory Kucherov. 2022. Space-efficient representation of genomic k-mer count tables. *Algorithms Mol. Biol.* 17, 1 (2022), 5. https://doi.org/10.1186/S13015-022-00212-0

[76] Samuel D. Stearns. 1995. Arithmetic coding in lossless waveform compression. *IEEE Trans. Signal Process.* 43, 8 (1995), 1874–1879. https://doi.org/10.1109/78.403346

[77] Samuel D. Stearns, Li Zhe Tan, and Neeraj Magotra. 1993. Lossless compression of waveform data for efficient storage and transmission. *IEEE Trans. Geosci. Remote. Sens.* 31, 3 (1993), 645–654. https://doi.org/10.1109/36.225531

[78] Kapil Vaidya, Eric Knorr, Michael Mitzenmacher, and Tim Kraska. 2021. Partitioned Learned Bloom Filters. In *Proc. 9th International Conference on Learning Representations, (ICLR)*. https://openreview.net/forum?id=6BRLOfrMhW

[79] Sebastiano Vigna. 2025. ϵ-Cost Sharding: Scaling Hypergraph-Based Static Functions and Filters to Trillions of Keys. *CoRR* abs/2503.18397 (2025). https://doi.org/10.48550/ARXIV.2503.18397

[80] Lixi Zhou, K. Selçuk Candan, and Jia Zou. 2024. DeepMapping: Learned Data Mapping for Lossless Compression and Efficient Lookup. In *Proc. 40th IEEE International Conference on Data Engineering (ICDE)*. 1–14. https://doi.org/10.1109/ICDE60146.2024.00008