Learned Monotone Minimal Perfect Hashing

Paolo Ferragina
University of Pisa, Italy

Hans-Peter Lehmann
Karlsruhe Institute of Technology, Germany

Peter Sanders
Karlsruhe Institute of Technology, Germany

Giorgio Vinciguerra
University of Pisa, Italy

Abstract

A Monotone Minimal Perfect Hash Function (MMPHF) constructed on a set S of keys is a function that maps each key in S to its rank. On keys not in S, the function returns an arbitrary value. Applications range from databases, search engines, data encryption, to pattern-matching algorithms.

In this paper, we describe LeMonHash, a new technique for constructing MMPHFs for integers. The core idea of LeMonHash is surprisingly simple and effective: we learn a monotone mapping from keys to their rank via an error-bounded piecewise linear model (the PGM-index), and then we solve the collisions that might arise among keys mapping to the same rank estimate by associating small integers with them in a retrieval data structure (BuRR). On synthetic random datasets, LeMonHash needs 34% less space than the next larger competitor, while achieving about 16 times faster queries. On real-world datasets, the space usage is very close to or much better than the best competitors, while achieving up to 19 times faster queries than the next *larger* competitor. As far as the construction of LeMonHash is concerned, we get an improvement by a factor of up to 2, compared to the competitor with the next best space usage.

We also investigate the case of keys being variable-length strings, introducing the so-called LeMonHash-VL: it needs space within 13% of the best competitors while achieving up to 3 times faster queries than the next larger competitor.

2012 ACM Subject Classification Theory of computation → Data compression; Information systems → Point lookups

Keywords and phrases compressed data structure, monotone minimal perfect hashing, retrieval

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.46

Related Version *Extended Version*: `https://arxiv.org/abs/2304.11012` [19]

Supplementary Material *Software (Source Code)*: `https://github.com/ByteHamster/LeMonHash`
archived at `swh:1:dir:af11424a42cdc2b2c4d8c62d041314c37075e8d9`

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500). PF and GV have been supported by the European Union – Horizon 2020 Program under the scheme "INFRAIA-01-2018-2019 – Integrating Activities for Advanced Communities", Grant Agreement n. 871042, "SoBigData++: European Integrated Infrastructure for Social Mining and Big Data Analytics" (`http://www.sobigdata.eu`), 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, by the spoke "FutureHPC & BigData" of the ICSC – Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing funded by European Union – NextGenerationEU – PNRR, by the Italian Ministry of University and Research "Progetti di Rilevante Interesse Nazionale" project: "Multicriteria data structures and algorithms" (grant n. 2017WR7SHH).

Acknowledgements We thank Stefan Walzer for early discussions leading to this paper.

31st Annual European Symposium on Algorithms (ESA 2023).
Editors: Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman; Article No. 46; pp. 46:1–46:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Given a set S of n keys drawn from a universe $[u] = \{0, \dots, u-1\}$, a *Monotone Minimal Perfect Hash Function* (MMPHF) is a hash function that maps keys from S to their rank, and returns an arbitrary value for keys not in S. As the name suggests, such a function is both *perfect* because it has no collisions on S, and *minimal* because its output range is $[n]$. Differently from a *Minimal Perfect Hash Function* (MPHF) [4, 12, 16, 26, 40, 42, 44, 50], which maps keys from S bijectively to $[n]$ in any order, and from an *Order-Preserving MPHF* (OPMPHF) [25], which retains a given (arbitrary) order on the keys, an MMPHF takes advantage of the natural order of the universe to rank the keys in S in small space, i.e. without encoding them. Indeed, encoding S needs $\log \binom{u}{n}/n = \Omega(\log \frac{u}{n})$ bits per key, and encoding the ranks via an OPMPHF needs $\log(n!)/n = \Omega(\log n)$ bits per key, whilst an MMPHF may use as few as $\mathcal{O}(\log \log \log u)$ bits per key [2], which was recently proven to be optimal [1]. Throughout this paper, $\log x$ stands for $\log_2 x$, and we use the w-bit word RAM model.

MMPHFs have numerous applications [1]. They enable efficient queries both in encrypted data [11] and databases [39, 41]. Further applications can be found in information retrieval, where MMPHFs can be used to index the lexicon [54] or to compute term frequencies [6, 46], and in pattern matching [5, 27, 32], where MMPHFs are applied mostly to integer sequences representing the occurrences of certain characters in a text.

Despite the widespread use of MMPHFs and recent advancements on their asymptotic bounds [1], the practical implementations have not made significant progress in terms of new designs and improved space-time performance since their introduction more than a decade ago [3], with only some exceptions targeting query time [33]. As a matter of fact, the solutions in [3] are very sophisticated and well-optimised, and they offer a vast number of efficient space-time trade-offs that were hard to beat.

In this paper, we offer a fresh new perspective on MMPHFs that departs from existing approaches, which are mostly based on a trie-like data structure on the keys. We build upon recent advances in (learning-based) indexing data structures, namely the PGM-index [20, 24], and in retrieval data structures (or static functions), namely BuRR [14]. The former learns a piecewise linear approximation mapping keys in S to their rank estimate. The latter allows associating a small fixed-width integer to each key in S, without storing S. We combine these two seemingly unrelated data structures in a surprisingly simple and effective way. First, we use the PGM to monotonically map keys to buckets according to their rank estimate, and we store the global rank of each bucket's first key in a compressed data structure. Second, since the rank estimate of some keys might coincide, we solve such bucket collisions by storing the local ranks of these keys using BuRR. We call our proposal *LeMonHash*, because it *learns* and *leverages* the smoothness of the input data to build a space-time efficient *monotone* MPHF. On the theoretical side, this achieves $\mathcal{O}(1)$ bits per key for inputs which are sufficiently random within buckets – breaking the superlinear lower bound. Practically, on various integer datasets tried, it needs about one-third less space than previous approaches and is an order of magnitude faster. We also extend LeMonHash to support variable-length string keys. This approach needs space within 13% of the best competitors while being up to 3× faster.

Outline. We first describe the basic building blocks of LeMonHash in Section 2 and discuss related work in Section 3. In Section 4, we describe LeMonHash for integers and then extend it to variable-length strings in Section 5. In Section 6, we discuss variants and refinements, before proving the space-time guarantees of LeMonHash in Section 7. In Section 8, we present our experiments. In Section 9, we summarise the paper and give an outlook for future work.

2 Preliminaries

In this section, we describe the basic building blocks of LeMonHash.

Bit Vectors. Given a bit vector of size n and $b \in \{0, 1\}$, the $rank_b(x)$ operation returns the number of b-bits before position x, and the $select_b(i)$ operation returns the position of the ith b-bit. These operations can be executed in constant time using as little as $o(n)$ bits on top of the bit vector [13, 34], and they have very space-time efficient implementations [30, 38, 52].

Elias-Fano. Elias-Fano Coding [15, 17] is a way to efficiently store a non-decreasing sequence of n integers over a universe of size u. An integer at position i is split into two parts. The $\log n$ upper bits x are stored in a bit vector H as a 1-bit in $H[i + x]$. The remaining lower bits are directly stored in an array L. Integers can be accessed in constant time by finding the ith 1-bit in H using a $select_1$ data structure and by looking up the lower bits in L. Predecessor queries are possible by determining the range of integers that share the same upper bits of the query key using two $select_0$ queries, and then performing a binary search on that range. If there are no duplicates, this binary search takes $\mathcal{O}\big(\min\{\log n, \log \frac{u}{n}\}\big)$ time. The space usage of an Elias-Fano coded sequence is $n\lceil \log \frac{u}{n} \rceil + 2n + o(n)$ bits (see [47, §4.4]). Partitioned Elias-Fano [49] is an extension that uses dynamic programming to partition the input into multiple independent Elias-Fano sequences to minimise the overall space usage.

PGM-index. The PGM-index [24] is a space-efficient data structure for predecessor and rank queries on a sorted set of n keys from an integer universe $[u]$. Given a query $q \in [u]$, it computes a rank estimate that is guaranteed to be close to the correct rank by a given integer parameter ε. If one stores the input keys, then the correct rank can be recovered via an $\mathcal{O}(\log \varepsilon)$-time binary search on $2\varepsilon + 1$ keys around the rank estimate. The PGM is constructed in $\mathcal{O}(n)$ time by first mapping the sorted integers $x_1, \ldots, x_n$ in S to points $(x_1, 1), \ldots, (x_n, n)$ in a key-position Cartesian plane, and then learning a piecewise linear ε-approximation of these points, i.e. a sequence of m linear models each approximating the rank of the keys in a certain sub-range of $[u]$ with a maximum absolute error ε. The value m, which impacts on the space of the PGM, can range between 1 and $m \leq n/(2\varepsilon)$ [24, Lemma 2] depending on the "approximate linearity" of the points. In practice, it is very low and can be proven to be $m = \mathcal{O}\big(n/\varepsilon^2\big)$ when the gaps between keys are random variables from a proper distribution [20]. The time complexity to compute the rank estimate with a PGM is given by the time to search for the linear model that contains the searched key q, which boils down to a predecessor search on m integers from a universe of size u. For this, there exist many trade-offs in various models of computations [24, 48].

Retrieval Data Structures. A *retrieval data structure* or *static function* on a set S of n keys denotes a function $f : S \to \{0, 1\}^r$ that returns a specific r-bit value for each key. Applying the function on a key not in S returns an arbitrary value. Retrieval data structures take $(1 + \eta)rn$ bits, where $\eta \geq 0$ is the *space overhead* over the space lower bound of rn bits.

MWHC [43] is a retrieval data structure based on hypergraph peeling, has an overhead $\eta = 0.23$ and can be evaluated in constant time. 2-step MWHC [3] can have a smaller overhead than MWHC by using two MWHC functions of different widths.

The more recently proposed *Bumped Ribbon Retrieval* (BuRR) data structure [14] basically consists of a matrix. The output value for a key can be obtained by multiplying the hash of the key with that matrix. The matrix can be calculated by solving a linear equation system. Because BuRR uses hash functions with *spacial coupling* [53], the equation system is almost a diagonal matrix, which makes it very efficient to solve. When some rows of the equation system would prevent successful solving, BuRR *bumps* these rows (and the corresponding keys) to the next layer of the same data structure. BuRR has an overhead $\eta = \mathcal{O}\big(\log W/(rW^2)\big)$ and can be evaluated in $\mathcal{O}(1 + rW/\log n)$ time, where $W = \mathcal{O}(\log n)$ is a parameter called ribbon width. In practice, BuRR achieves space overheads well below $\eta = 1\%$ while being faster than widely used data structures with much larger overhead [14].

3 Related Work

Non-monotone perfect hash functions are a related and very active area of research [4, 7, 12, 16, 26, 40, 42, 44, 50]. Due to space constraints, we do not review them in detail. For a more detailed list, refer to Ref. [40]. We also do not describe order-preserving minimal perfect hash functions [25] because their theoretical lower bound can trivially be reached by using a retrieval data structure taking $\log n$ bits per key (plus a small overhead). Another loosely related result is using learned models as a replacement for hash functions in traditional hash tables [37, 51], but it generally has a negative impact on the probe/insert throughput (and most likely on the space too, due to the storage of the models' parameters, which these studies do not evaluate). We now look at monotone minimal perfect hash functions, first describing the idea of bucketing before then continuing with specific MMPHF constructions.

Bucketing. Bucketing [3] is a general technique to break down MMPHF construction into smaller sub-problems. The idea is to store a simple monotone, but not necessarily minimal or perfect *distributor* function that maps input keys to buckets. Each bucket receives a smaller number of keys that can then be handled using some (smaller) MMPHF data structure. To determine the global rank of a key, we need the prefix sum of the bucket sizes. For equally-sized buckets, this is trivial. Otherwise, this sequence can be stored with Elias-Fano coding. In the paper by Belazzougui et al. [3], where many of the following techniques are described, the authors use MWHC [43] to explicitly store the ranks within each bucket. LeMonHash uses a learned distributor and buckets of expected size 1 (see Section 4).

Longest Common Prefix. Bucketing with Longest Common Prefixes (LCP) [2] maps keys to equally sized buckets. A first retrieval data structure maps all keys to the *length* of the LCP among all keys in its bucket. A second one then maps the *value* of the LCP to the bucket index. Overall, it uses $\mathcal{O}(\log \log u)$ bits per key and query time $\mathcal{O}((\log u)/w)$, and in practice it has been shown to be the fastest but the most space-inefficient MMPHF [3].

Partial Compacted Trie. First map the keys to equally sized buckets and consider the last key of each bucket as a *router* indexed by a *compacted trie*, e.g., a binary tree where every node contains a bit string denoting the common prefix of its descending keys. During queries, the trie is traversed by comparing the bit string of the traversed nodes with the key to decide whether to stop the search operation at some node (if the prefix does not match), or descend into the left or right subtree based on the next bit of the key. A *Partial Compacted Trie* (PaCo Trie) [3] compresses the compacted trie above by 30–50% by exploiting the fact that, in an MMPHF, the trie needs to correctly rank only the keys from the input set. Therefore, each node can store a shorter bit string just long enough to correctly route all input keys.

Hollow Trie. A *Hollow Trie* [3] only stores the *position* of the next bit to look at. Hollow tries can be represented succinctly using balanced parentheses [45]. To use hollow tries for bucketing, and thus allow the routing of not-indexed keys, we need a modification to the data structure. The *Hollow Trie Distributor* [3] uses a retrieval data structure that maps the compacted substrings of each key in each tree node to the behaviour of that key in the node (stopping at the left or right of the node, or following the trie using the next bit of the key). Overall, it uses $\mathcal{O}(\log\log\log u)$ bits per key and query time $\mathcal{O}(\log u)$.

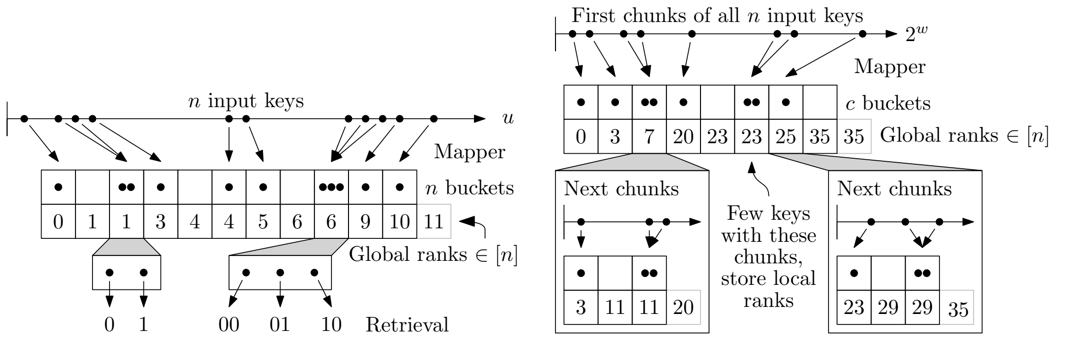
ZFast Trie. To construct a *ZFast Trie* [2], we first generate a path-compacted trie. Then, for prefixes of a specific length (*2-fattest number*) of all input keys, a dictionary stores the trie node that represents that prefix. A query can then perform a binary search over the length of the queried key. If there is no node in the dictionary for a given prefix, the search can continue with the pivot as its upper bound. If there is a node, the lower bound of the search can be set to the length of the longest common prefix of all keys represented by that node. The ZFast trie uses $\mathcal{O}(\log\log\log u)$ bits per key and query time $\mathcal{O}((\log u)/w + \log\log u)$.

Path Decomposed Trie. In the previous paragraphs, we described binary tries with a rather high height. However, those tries are inefficient to query because of the pointer chasing to non-local memory areas. The main idea behind *Path Decomposed Tries* [18], which can be used as an MMPHF [33], is to reduce the height of the tries. We first select one path all the way from the root node to a leaf. This path is now contracted to a single node, which becomes the root node in our new path decomposed trie. The remaining nodes in the original trie form subtries branching from every node in that path. We take all of these subtries, make them children of the root node, and annotate them by their branching character with respect to the selected path. The subtries are then converted to path decomposed tries recursively. In *centroid* path decomposition, the path to be contracted is always the one that descends to the node with the most leaves in its subtree.

4 LeMonHash

We now introduce the main contribution of this paper – the MMPHF LeMonHash. The core idea of LeMonHash is surprisingly simple. We take all the n input integers and map them to n buckets using some *monotone* mapping function, that we will describe later. We store an Elias-Fano coded sequence with the *global ranks* of the first key in each bucket using $2n + o(n)$ bits. Given a bucket of size b, we use a $\lceil \log b \rceil$-bit retrieval data structure (see Section 2) to store the *local ranks* of all its keys. Note that we do not need to store local ranks if the bucket has only 0 or 1 keys. For squeezing space, instead of storing one retrieval data structure per bucket, we store a collection of retrieval data structures so that the ith one stores the local ranks of all keys mapped to buckets whose size b is such that $i = \lceil \log b \rceil$. An illustration of the overall data structure is given in Figure 1a.

Bucket Mapping Function. The space efficiency of LeMonHash is directly related to the quality of the monotone mapping function. For uniform random integers, a linear mapping from input keys to n buckets, i.e. a mapping from a key x to the bucket number $\lfloor xn/u \rfloor$, leads to an MMPHF with a space usage of just 2.915 bits per key (see Theorem 1). Intuitively, such a linear mapping returns a rank estimate in $[n]$ for a given key. However, for skewed distributions, the rank estimate can be far away which can create large buckets whose local ranks are expensive to store. For example, if the majority of the keys are such that $x < u/n$,



(a) LeMonHash. Keys are mapped to buckets. Ranks within buckets are stored in (a collection of) retrieval data structures.

(b) LeMonHash-VL. Global ranks in each level are stored together. Buckets that are not handled recursively use retrieval data structures like before.

Figure 1 Illustration of the LeMonHash and LeMonHash-VL data structures.

then the first bucket will be large enough to require $\Theta(\log n)$ bits per key, i.e. our MMPHF degenerates to a trivial OPMPHF. To tackle this problem, we implement the mapping function with a *PGM-index* [24]. As we observed in Section 2, the PGM was originally designed as a predecessor-search data structure. Here, we use the PGM as a *rank estimator* that, for a given key, returns an ε-bounded estimate of its rank. To achieve this result in LeMonHash, we do not store the list of indexed keys and simply use the PGM's rank estimate as the bucket index. The PGM internally adapts to the input data by learning the smoothness in the distribution via a piecewise linear ε-approximation model, thus it can be thought of as a "local" approximation of the linear mapping above. Real-world data sets can often be approximated using piecewise linear models, as discussed in the literature [20] and also demonstrated by the good space efficiency of our experiments (see Section 8). There is a trade-off between the amount of space needed to represent the PGM and the quality of the mapping, which depends on both the input data distribution and the given integer parameter ε. In Section 8, we test both a version with a constant ε value and a version that auto-tunes its value by constructing multiple PGMs and then selecting the optimal ε. Finally, we observe that with the PGM mapper, unlike for the linear mapping and other non error-bounded learning-based approaches [23, 36], the number of retrieval data structures we need to keep is bounded by $\mathcal{O}(\log \varepsilon)$ regardless of the input key distribution (see Theorem 2).

Queries. Given a key q, we obtain its bucket i using the mapping function. The global rank of the (first key in the) bucket is the ith integer in the Elias-Fano coded sequence of global ranks, which can be accessed in constant time, and the bucket size is computed by subtraction from the next integer in that sequence. The bucket size b directly tells us which retrieval data structure to query, i.e. the $\lceil \log b \rceil$th one. Evaluating the retrieval data structure with q gives us its local rank in the bucket. Adding this to the global rank of the bucket gives us the rank of q. As we show in Section 7, for uniform data, the linear bucket mapper gives constant time queries, while for other inputs we use the PGM mapper and the query time is $\mathcal{O}(\log \log u)$.

Comparison to Known Solutions. Known MMPHFs in the literature typically divide the keys into equal-size buckets and build a compact trie-based distributor. Unlike them, LeMonHash learns the data linearities and leverages them to distribute keys to buckets close to their rank. Whenever some keys collide into a bucket, LeMonHash handles these keys via a (small) collection of succinct retrieval structures. In contrast to known solutions, whenever a key is the only one mapped to its bucket, no information needs to be stored in (and no query is issued on) a retrieval data structure. These features allow LeMonHash to possibly achieve reduced space occupancy compared to classic MMPHFs, which are oblivious to data linearities. Also, LeMonHash can reduce the query time by replacing the cache-inefficient traversal of a trie with the PGM mapper, which in practice is fast to evaluate.

5 LeMonHash-VL

Of course, the idea of LeMonHash can be immediately applied to keys whose maximum longest common prefix (LCP) is less than w bits. In this case, each string prefix and the following bit (which are sufficient to distinguish every string from each other) fit into one machine word and thus can be handled efficiently in time and in space by the PGM mapper. For strings with longer LCPs, we introduce a tree data structure that we call *LeMonHash-VL* (since it handles Variable-Length strings). The main idea is to simply compute the bucket mapping on a length-w substring of each string, which we call a *chunk*. Buckets that receive many keys using this procedure are then handled recursively. Details follow.

Overview. We start with a root node representing all the string keys in S and consider the set of chunks extracted from each key starting from position $|p|$ (which we store), where p is the LCP among the keys in S. Given these c distinct chunks, we construct a PGM mapper to distribute the keys to buckets in $[c]$, and we store an Elias-Fano coded sequence with the global ranks of the first key in each bucket. Clearly, different keys can be mapped to the same bucket because the PGM mapper is not perfect (as in the integer case) and because they share the same chunk value (unlike in the integer case). For example, for the strings $S = \{\texttt{cherry}, \texttt{cocoa}, \texttt{coconut}\}$ with $p = \texttt{c}$ and chunks composed of 3 characters, the keys `cocoa` and `coconut` share the chunk value `oco` and will be mapped to the same bucket.

If a bucket of size b contains fewer input strings than a specific threshold t, we store the local ranks of the strings in the bucket in a $\lceil \log b \rceil$-bit retrieval data structure. Once again, we do not need to store local ranks if the bucket has only 0 or 1 keys. If instead the bucket is large (i.e. $b \geq t$), we create a *child node* in the tree data structure by applying the same idea recursively on the strings S' of that bucket. This means that we compute a PGM mapper on the chunks extracted from each string in S' starting from position $|p'|$, where p' is the LCP among the bucket strings S'. Notice that $|p'| \geq |p|$ but we always guarantee that $S' \subsetneq S$, so the recursion is bounded. In practice, we set the threshold $t = 128$ (see Section 8.1).

At query time, we can use the sequence of global ranks to calculate the bucket size b, which allows determining whether we need to continue recursively on a child (because $b \geq t$) or directly return the global rank of the bucket plus the local rank stored in the $\lceil \log b \rceil$-bit retrieval data structure. Figure 1b gives an overview of the data structure.

We observe that the global ranks of each node increase monotonically from left to right in each level of the overall tree. Therefore, we merge all these global ranks in a level into one Elias-Fano sequence, thereby avoiding the space overhead of storing many small sequences.

Of course, each inner node of the tree needs some extra metadata, like the encoding of its bucket mapper, the value of $|p|$, and an offset to its first global rank in the per-level Elias-Fano sequence. We associate a node to its metadata via a minimal perfect hash function, where the identifier of a node is given by the path of the buckets' indices leading to it.

Given the overall idea, there is a wide range of optimisations that we use. In the following, we outline the main algorithmic ones and refer the interested reader to our implementation [28] and the extended version [19] for the many other small-and-tricky optimisations, such as the use of specialised instructions like `popcount` and `bextr`, or lookup tables.

Alphabet Reduction. The number of nodes and the depth of LeMonHash-VL depend on both the length and distribution of the input strings, and on how well the PGM mapper at each node can map strings to distinct buckets given their w-bit chunks. Therefore, we should aim to fit as much information as possible in the w-bit chunks. We do so by exploiting the fact that, in real-world data sets, often only a very small alphabet Σ of branching characters distinguish the strings in each bucket, and that we do not care about the other characters. We extract chunks from the suffix of each string starting from the position following the LCP p, as before, but interpret the suffix as a number in radix $\sigma = |\Sigma|$ where each character is replaced by its 0-based index in Σ if present, or by 0 if not present. For example, for a node on the strings $\{\texttt{shoppers}, \texttt{shopping}, \texttt{shops}\}$ whose LCP is $p = \texttt{shop}$, we would store the alphabet $\Sigma = \{\texttt{e}, \texttt{i}, \texttt{p}, \texttt{s}\}$ and map the suffix "`pers`" of "`shoppers`" to $\text{index}(\texttt{p})\sigma^3 + \text{index}(\texttt{e})\sigma^2 + \text{index}(\texttt{r})\sigma^1 + \text{index}(\texttt{s})\sigma^0 = 2\sigma^3 + 0\sigma^2 + 0\sigma^1 + 3\sigma^0$. Observe that the chunks computed in this way still preserve the lexicographic order of the strings. The number of characters we extract is computed to fit as many characters as possible in a w-bit word, i.e. $\lfloor w/\log\sigma \rfloor$ characters. In our implementation over bytes, we store Σ via a bitmap of size 128 or 256, depending on whether its characters are a subset of ASCII or not. Finally, we mention that a mapping from strings to numbers in radix σ has also been used to build compressed string dictionaries [8], but the twist here is that we are considering only the alphabet of the branching characters since we do not need to store the keys.

Elias-Fano Sequences. The large per-level Elias-Fano sequences of global ranks have a very irregular structure. For example, if many of the strings in a node share the same chunks, there is a large gap between two of the stored ranks. We can deal with these irregularities and reduce the overall space usage by using partitioned Elias-Fano [49]. Furthermore, the PGM mappers do not always provide a very uniform mapping, which thus results in empty buckets. An empty bucket corresponds to a duplicate offset value being stored in the Elias-Fano sequences (see e.g. the duplicate offset 23 in Figure 1b). To optimise the space usage of such duplicates, we filter them out before constructing the partitioned Elias-Fano sequence. We do this by grouping the stored numbers in groups of 3 numbers. If all 3 numbers are duplicates of the number before that group, we do not need to store the group. A bit vector with rank support indicates which groups were removed.

Perfect Chunk Mapping. In many datasets, there might be only a small number of different chunks, even if the number of strings they represent is large. For instance, chunks computed on the first bytes of a set of URLs might be a few due to the scarcity of hostnames, but each host may contain many distinct pages. In these cases, instead of a PGM, it might be more space-efficient to build a (perfect) map from chunks to buckets in $[c]$ via a retrieval data structure taking $c\lceil\log c\rceil$ bits overall (plus a small overhead), where c is the number of distinct chunks. In practice, we apply this optimisation whenever $c < 128$ (see Section 8.1).

Comparison to Known Solutions. In essence, LeMonHash-VL applies the idea of LeMonHash recursively to handle variable-length strings. Therefore, unlike known solutions, it can leverage data linearities to distribute w-bit chunks from the input strings to buckets using small space, and use additional child nodes only whenever a bucket contains many strings that thus require inspecting the following chunks to be distinguished. Additionally, it performs an adaptive alphabet reduction within the buckets to fit more information in the w-bit chunks, thus leveraging the presence of more regularities in the input data. Overall, these features result in a data structure that has a small height and is efficient to be traversed.

6 Variants and Refinements

LeMonHash can be refined in numerous ways, which we only mention briefly due to space constraints. Looking at a possible external memory implementation, LeMonHash can be constructed trivially by a linear sweep and queries are possible using a suitable representation of the predecessor and bucket-size data structures. LeMonHash can also be constructed in parallel without affecting the queries, in contrast to the trivial parallelisation by partitioning the input. In LeMonHash-VL, extracting chunks from non-contiguous bytes reduces the height of the trees but has worse trade-offs in practice. Finally, we present an alternative to storing the local ranks explicitly. The idea is to recursively split the universe size of that bucket and record the number of keys smaller than that midpoint. Despite its query overhead, this technique might be of general interest for MMPHFs. Refer to the extended version [19] for details.

7 Analysis

We now prove some properties of our LeMonHash data structure for integers. In our analysis, we use succinct retrieval data structures taking $rn + o(n)$ bits per stored value and answering queries in constant time (see Section 2 and [14]). Furthermore, since our bucket mappers need multiplications and divisions, we make the simplifying assumption $u = 2^w$ to avoid dealing with the increased complexity of these arithmetic operations over large integers.

▶ **Theorem 1.** *A LeMonHash data structure with a bucket mapper that simply performs a linear interpolation of the universe on a list of n uniform random keys needs $\approx n(2.91536 + o(1))$ bits on average*[1] *and answers queries in constant time.*

Proof. We approximate the number of keys per bucket using a Poisson distribution which results in $0.91536n + o(n)$ bits of space for the retrieval data structures. On top of that, an Elias-Fano coding of the global bucket ranks gives $2n + o(n)$ bits. Refer to the extended version [19] for the full proof. ◀

While this result is formally only valid for a global uniform distribution, for use in LeMonHash it suffices if each segment computed by the PGM-index is sufficiently smooth. It need not even be uniformly random as long as each local bucket has a constant expected size. As long as the space for encoding the segments is in $\mathcal{O}(n)$ bits, we retain the linear space bound of Theorem 1. Moreover, the following worst-case analysis gives us a fallback position that holds regardless of any assumptions.

▶ **Theorem 2.** *A LeMonHash data structure with the PGM mapper takes $n(\lceil \log(2\varepsilon+1)\rceil + 2 + o(1)) + \mathcal{O}\big(m \log \frac{u}{m}\big)$ bits of space in the worst case and answers queries in $\mathcal{O}\big(\log\log_w \frac{u}{m}\big)$ time, where m is the number of linear models in a PGM with an integer parameter $\varepsilon \geq 0$ constructed on the n input keys.*

---

[1] Numerically, we find that a better space usage of $\approx 2.902n$ bits can be achieved by mapping the n keys to only $\approx 0.909n$ buckets, but this difference is irrelevant in practice. It is also interesting to note that this is close to the space requirements of most of the practical non-monotone MPHFs [4,7,12,16,26,40,42,44,50]. Using an MMPHF can be useful when indexing an array through an MPHF, because sorting the hash values can be more cache efficient than a large number of random accesses to the array.

Proof. The basic idea is that the rank estimate returned by the PGM is guaranteed to be far from the correct rank by ε, which limits the space of the retrieval data structures. The $\mathcal{O}\big(m \log \frac{u}{m}\big)$-term in the space bound is given by a compressed encoding of the linear models in the PGM, and the query time is given by a predecessor search structure on the linear models' keys. Refer to the extended version [19] for the full proof. ◀

The worst-case bounds obtained in Theorem 2 are hard to compare with the ones of classic MMPHF (see Section 3) due to the presence of m (and ε), which depends on (and must be tuned according to) the approximate linearity of the input data, which classic MMPHFs are oblivious to.[2] Refer to Section 2 for bounds on m. Our experiments show that we obtain better space or space close to the best classic MMPHFs, while being much faster (we use a weaker but practical predecessor search structure than the one in Theorem 2). Refer to Section 8 for details.

8 Experiments

In the following section, we first compare different configurations of LeMonHash and LeMonHash-VL before comparing them with competitors from the literature.

Experimental Setup. We perform our experiments on an Intel Xeon E5-2670 v3 with a base clock speed of 2.3 GHz running Ubuntu 20.04 with Linux 5.10.0. We use the GNU C++ compiler version 11.1.0 with optimisation flags `-O3 -march=native`. As a retrieval data structure, we use BuRR [14] with 64-bit ribbon width and 2-bit bumping info. To store the bucket sizes, we use the select data structure by Kurpicz [38] in LeMonHash and Partitioned Elias-Fano [49] in LeMonHash-VL. To map tree paths to the node metadata, we use the MPHF PTHash [50]. For the PGM implementation in LeMonHash, we use the encoding from Theorem 2 and use a predecessor search on the Elias-Fano sequence (Section 2). In LeMonHash-VL, since the number of linear models in a node is typically small, we encode them explicitly as fixed-width triples $(\mathit{key}, \mathit{slope}, \mathit{intercept})$ and find the predecessor via a binary search on the keys. All our experiments are executed on a single thread. Because the variation is very small, we run each experiment only twice and report the average. We run the Java competitors on OpenJDK 17.0.4 and perform one warm-up run for the just-in-time compiler that is not measured. With this, the Java performance is expected to be close to C++ [3]. Because Java does not have an unsigned 64-bit integer type, we subtract 2^{63} from each input key to keep their relative order.

The code and scripts needed to reproduce our experiments are available on GitHub under the General Public License [28, 29].

Datasets. Our datasets, as in previous evaluations [3, 33], are a *text* dataset that contains terms appearing in the text of web pages [3] and *urls* crawled from .uk domains in 2007 [10]. Additionally, we also test with *dna* sequences consisting of 32-mers [22]. Regarding real-world integer datasets, *5gram* contains positions of the most frequent letter in the BWT of a text file containing 5-grams found in books indexed by Google [9, 31]. The *fb* dataset contains Facebook user IDs [35] and *osm* contains OpenStreetMap locations [35]. As synthetic integer datasets, we use 64-bit *uniform*, *normal*, and *exponential* distributions. Refer to Table 1 for details.

[2] This happens also in other problems in which data is encoded with linear models [9, 21].

Table 1 Datasets used for the experiments, together with their length or average (ø) length. Top: real-world string datasets. Middle: real-world integer datasets. Bottom: synthetic integer datasets.

Dataset	n	Length	Description
text	35M	ø 11 bytes	Terms appearing in the text of web pages, GOV2 corpus [3]
dna	367M	32 bytes	32-mer from a DNA sequence, Pizza&Chili corpus [22]
urls	106M	ø 105 bytes	Web URLs crawled from .uk domains in 2007 [10]
5gram	145M	32 bits	Positions of the most frequent letter in the BWT of a text file containing 5-grams found in books indexed by Google [9, 31]
fb	200M	64 bits	Facebook user IDs [35]
osm	800M	64 bits	OpenStreetMap locations [35]
uniform	100M	64 bits	Uniform random
normal	100M	64 bits	Normal distribution ($\mu = 10^{15}$, $\sigma^2 = 10^{10}$)
exponential	100M	64 bits	Exponential distribution ($\lambda = 1$, scaled with 10^{15})

8.1 Tuning Parameters

In the following section, we compare several configuration parameters of LeMonHash and show how they provide a trade-off between space usage and performance.

LeMonHash. Different ways of mapping the keys to buckets have their own advantages and disadvantages. Table 2 gives measurements of the construction and query throughput, as well as the space consumption of different bucket mappers. Our implementation of LeMonHash with a linear bucket mapper achieves a space usage of $2.94n$ bits, which is remarkably close to the theoretical space usage of $2.91n$ bits (see Theorem 1). Of course, a global, linear mapping does not work for all datasets. A bucket mapper that creates equal-width segments by interpolating between sampled keys (denoted as "Segmented" in the table) is fast to construct and query, and it achieves good space usage. But, as for the global linear mapping, this approach is not robust enough to manage arbitrary input distributions. In particular, for this heuristic mapper, it is easy to come up with a worst-case input that degenerates the space usage. Conversely, with the PGM mapper, LeMonHash still achieves $2.96n$ and $2.98n$ bits on uniform random integers but it is more performant and robust on other datasets (except on osm, where the heuristic mapper obtains a good enough mapping with only its equal-width segments, which are inexpensive to store). In fact, we explicitly avoided heuristic design choices in our PGM mapper (such as sampling input keys, removing outliers, or using linear regression) to not inflate our performance on the tested datasets at the expense of robustness on unknown ones (see Ref. [36]). Finally, on most input distributions, auto-tuning the value of $\varepsilon \in \{15, 31, 63\}$ does not have a large effect on the space usage.

LeMonHash-VL. Table 3 lists the effect of alphabet reduction on the query and construction performance. In general, alphabet reduction enables noticeable space improvements with only a small impact on the construction time. For the dna dataset, which uses only 15 different characters, the alphabet reduction has the largest effect, saving 1.3 bits per key and simultaneously making the queries 40% faster. The faster queries can be explained by the reduced tree height. Note that alphabet reduction makes the queries slightly slower for the other datasets. The reason is that instead of one single `bswap` instruction for chunk extraction, it needs multiple arithmetic operations (including `popcount`) for each input character. The

Table 2 Comparison of different bucket mappers. The space usage is given in bits per key, the query throughput in kQueries/second, and the construction throughput (c.t.) in MKeys/second.

Dataset	Linear mapper			PGM ε = auto			PGM $\varepsilon = 31$			Segmented		
	bpk	kq/s	c.t.	bpk	kq/s	c.t.	bpk	kq/s	c.t.	bpk	kq/s	c.t.
5gram	5.60	1833.5	6.2	2.62	1747.0	3.8	2.63	1779.4	8.5	2.64	2145.9	14.5
fb	34.35	0.8	5.1	4.91	1156.1	2.8	4.91	1150.7	5.1	4.93	1441.3	7.2
osm	12.92	1525.3	5.5	4.42	999.6	2.8	4.42	998.6	5.0	4.33	1272.9	6.8
uniform	2.94	3244.6	8.7	2.96	1903.3	3.5	2.98	1850.5	6.5	3.03	2192.0	8.7
normal	34.27	105.3	4.8	2.95	1935.0	3.6	2.97	1858.0	6.6	3.00	1727.7	8.7
exponential	5.42	2715.9	6.0	2.95	1876.9	3.6	2.98	1791.5	6.6	3.01	2085.1	8.8

Table 3 Comparison of different variants of LeMonHash-VL. The space usage is given in bits per key, the query throughput in kQueries/second, and the construction throughput (c.t.) in MKeys/second. Variants with and without alphabet reduction (AR), a special indexed variant (Idx, see the extended version [19]), and a variant with fixed instead of auto-tuned parameter ε for the bucket mapper.

Dataset	ε = auto, no AR			ε = auto, AR			$\varepsilon = 63$, AR			Idx, ε = auto, AR		
	bpk	kq/s	c.t.	bpk	kq/s	c.t.	bpk	kq/s	c.t.	bpk	kq/s	c.t.
text	6.52	1062.9	1.7	6.03	1005.8	1.6	6.08	1001.8	2.5	6.10	933.2	2.3
dna	7.66	452.8	2.0	6.32	631.3	1.7	6.25	644.8	2.7	6.27	601.1	2.4
urls	7.14	282.7	2.3	6.37	298.8	1.8	6.46	295.1	2.3	6.63	298.1	1.6

indexed variant that builds chunks from the distinguishing bytes instead of a contiguous byte range (see the extended version [19]) is slower to construct but does not show clear space savings, which can be explained by larger per-node metadata. We also experimented with different thresholds for when to stop recursion, as well as the perfect chunk mapping (see Section 5). Given that the space overhead from each bucket mapper is the same for all data sets, it is not surprising that the same threshold (128 keys) works well for all datasets (see the extended version [19]). Finally, making the ε value of the PGM mapper constant instead of auto-tuned, we naturally get faster construction. As in the integer case, one would expect a fixed ε value to always produce results that are the same or worse than the auto-tuned version. This is not the case because, in the recursive setting, it is hard to estimate the effect of a mapper on the overall space usage. Therefore, an ε value that needs more space locally can lead to a mapping that proves useful on a later level of the tree. This is why $\varepsilon = 63$ can achieve better space usage than the auto-tuned version on the dna dataset.

8.2 Comparison with Competitors

In this section, we compare the performance of LeMonHash and LeMonHash-VL with competitors from the literature. Competitors include the C++ implementation by Grossi and Ottaviano [33] of the Centroid Hollow Trie, Hollow Trie, and Path Decomposed Trie. Because that implementation only supports string inputs, we convert the integers to a list of fixed-length strings. We point out that the Path Decomposed Trie crashes at an internal assertion when being run on integer datasets. For the Hollow Trie, we encode the skips with either Gamma or Elias-Fano coding, whatever is better on the dataset. We also include the Java

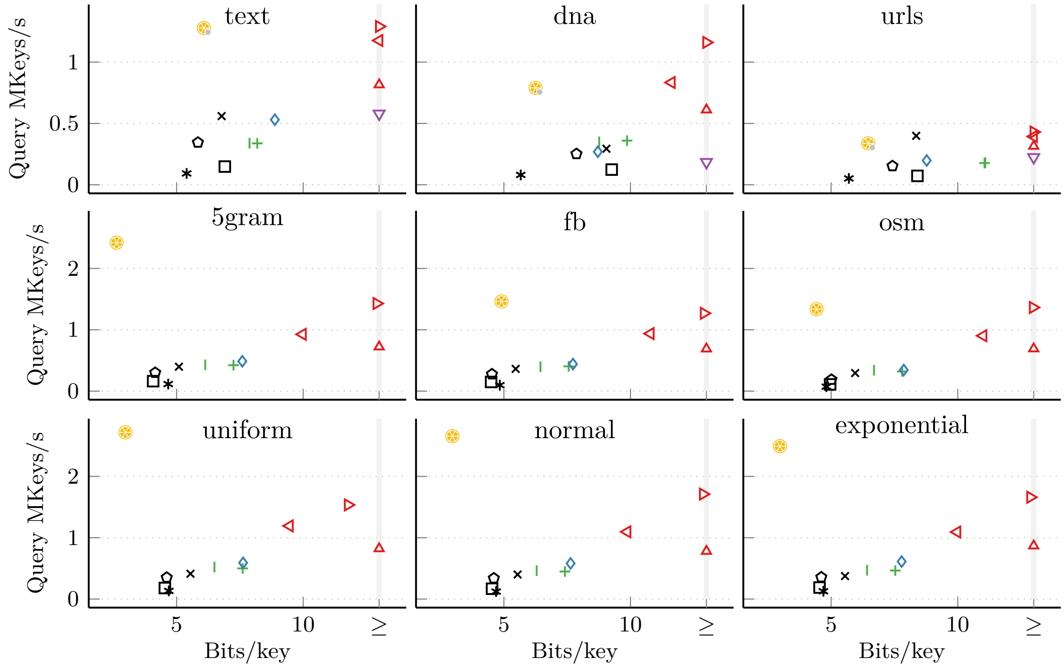


Figure 2 Query throughput for string, integer, and synthetic integer datasets vs space usage. The top-left corner of every plot shows the top-performing solutions in terms of space-time efficiency.

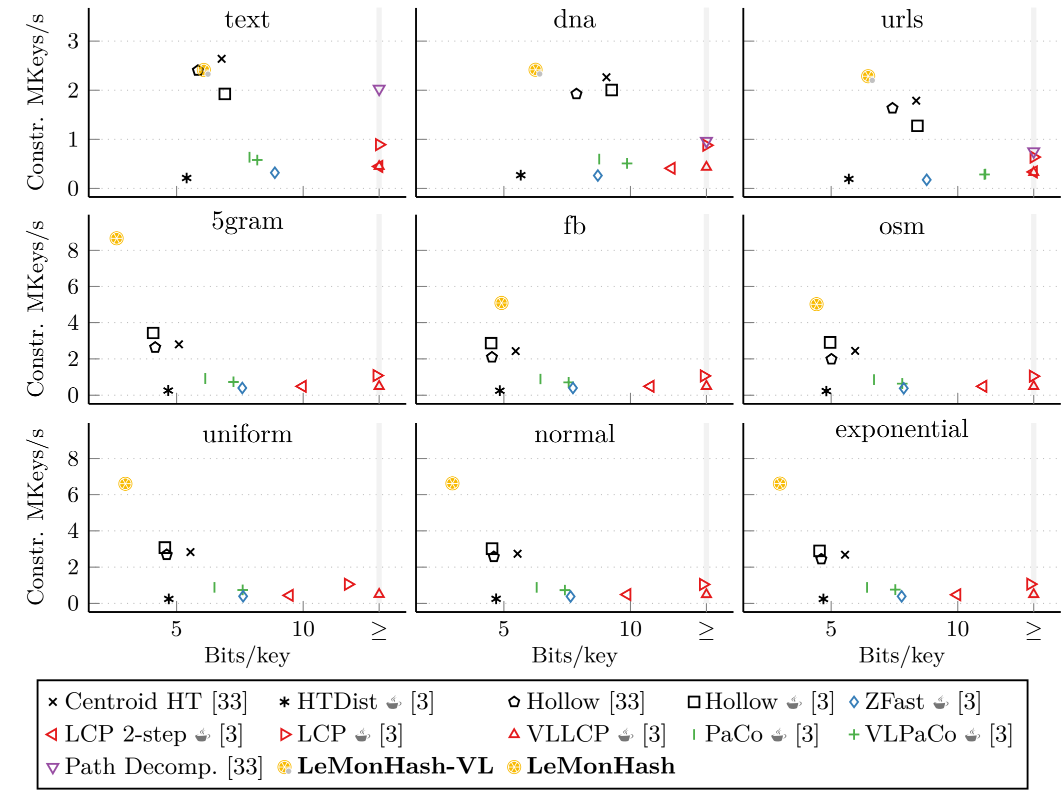


Figure 3 Construction throughput for string, integer, and synthetic integer datasets. Competitors with the ☕ symbol in the legend are implemented in Java.

implementations by Belazzougui et al. [3] of a range of techniques (see Section 3). We use either the FixedLong or PrefixFreeUtf16 transformation, depending on the data type of the input. For LeMonHash, we use the PGM mapper with $\varepsilon = 31$. For LeMonHash-VL, we use the PGM mapper with $\varepsilon = 63$, alphabet reduction and a recursion threshold $t = 128$.

Queries. Figure 2 plots the query throughput against the achieved storage space. In the extended version [19], we additionally detail the numbers in tabular format. The LCP-based methods (see Section 3) have very fast queries but also need the most space (in fact, they appear to the top-right of the plots). At the same time, LeMonHash matches or even outperforms the query throughput of LCP-based methods, while being significantly more space efficient (in fact, it appears towards the top-left of the plots). Compared to competitors with similar space usage, LeMonHash offers significantly higher query throughput.

Construction. Figure 3 plots the construction throughput against the space needed. On most synthetic integer datasets, LeMonHash provides a significant improvement to the state-of-the-art approaches, whereas it matches or outperforms the competitors on real-world datasets. LeMonHash improves the construction throughput by up to a factor of 2, compared to the competitor with the next best space usage (typically, variants of the Hollow Trie). While LeMonHash-VL does not achieve the same space usage as the Hollow Trie Distributor, its construction is significantly faster, and still it is the second best in space usage.

9 Conclusion and Future Work

In this paper, we have introduced the monotone minimal perfect hash function LeMonHash. LeMonHash, unlike previous solutions, learns and leverages data smoothness to obtain a small space usage and significantly faster queries. On most synthetic and real-world datasets, LeMonHash dominates all competitors – simultaneously – on space usage, construction and query throughput. Our extension to variable-length strings, LeMonHash-VL, consists of trees that are significantly more flat and efficient to traverse than competitors. This enables extremely fast queries with space consumption similar to competitors.

Future Work. Many MMPHF construction algorithms are based on the idea of explicitly storing ranks of keys within a small bucket. The idea to split small buckets recursively that we mention in Section 6 can help to reduce the space usage. It remains an open problem whether the idea works in practice, especially when the distribution of keys inside the bucket is skewed. It is also worth investigating a different construction of the piecewise linear approximation in the PGM that minimises the overall space given by the segments *and* the local ranks stored in retrieval data structures, rather than the current approach that maximises the length of the segment (thus minimising just the segments space). Applying non-linear transformations like low-degree polynomials within each segment would also be interesting future work. Finally, it would be interesting to apply smoothed analysis to formally show that many real-world distributions locally behave as if they were uniform random, therefore leading to tighter space bounds.

References

1 Sepehr Assadi, Martin Farach-Colton, and William Kuszmaul. Tight bounds for monotone minimal perfect hashing. In *Proc. 34th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 456–476, 2023. doi:10.1137/1.9781611977554.CH20.

2 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 785–794, 2009. doi:10.1137/1.9781611973068.86.

3 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Theory and practice of monotone minimal perfect hashing. *ACM J. Exp. Algorithmics*, 16, 2011. doi:10.1145/1963190.2025378.

4 Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *Proc. 17th Annual European Symposium on Algorithms (ESA)*, pages 682–693, 2009. doi:10.1007/978-3-642-04128-0_61.

5 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Trans. Algorithms*, 16(2):17:1–17:54, 2020. doi:10.1145/3381417.

6 Djamal Belazzougui, Gonzalo Navarro, and Daniel Valenzuela. Improved compressed indexes for full-text document retrieval. *J. Discrete Algorithms*, 18:3–13, 2013. doi:10.1016/j.jda.2012.07.005.

7 Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. High Performance Construction of RecSplit Based Minimal Perfect Hash Functions. In *31st Annual European Symposium on Algorithms (ESA 2023)*, volume 274 of *LIPIcs*, pages 19:1–19:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2023.19.

8 Antonio Boffa, Paolo Ferragina, Francesco Tosoni, and Giorgio Vinciguerra. Compressed string dictionaries via data-aware subtrie compaction. In *Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 233–249, 2022. doi:10.1007/978-3-031-20643-6_17.

9 Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. A learned approach to design compressed rank/select data structures. *ACM Trans. Algorithms*, 18(3):24:1–24:28, 2022. doi:10.1145/3524060.

10 Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2):33–38, 2008. doi:10.1145/1480506.1480511.

11 Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Proc. 31st Annual International Cryptology Conference (CRYPTO)*, pages 578–595, 2011. doi:10.1007/978-3-642-22792-9_33.

12 Jarrod A. Chapman, Isaac Ho, Sirisha Sunkara, Shujun Luo, Gary P. Schroth, and Daniel S. Rokhsar. Meraculous: De novo genome assembly with short paired-end reads. *PLOS ONE*, 6(8):1–13, August 2011. doi:10.1371/journal.pone.0023501.

13 David Richard Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.

14 Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership using ribbon. In *Proc. 20th International Symposium on Experimental Algorithms (SEA)*, pages 4:1–4:20, 2022. doi:10.4230/LIPICS.SEA.2022.4.

15 Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974. doi:10.1145/321812.321820.

16 Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. RecSplit: Minimal perfect hashing via recursive splitting. In *Proc. 22nd Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 175–185, 2020. doi:10.1137/1.9781611976007.14.

17 Robert Mario Fano. On the number of bits required to implement an associative memory. Technical report, MIT, Computer Structures Group, 1971. Project MAC, Memorandum 61".

18 Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. On searching compressed string collections cache-obliviously. In *Proc. 27th ACM Symposium on Principles of Database Systems (PODS)*, pages 181–190, 2008. doi:10.1145/1376916.1376943.

19 Paolo Ferragina, Hans-Peter Lehmann, Peter Sanders, and Giorgio Vinciguerra. Learned monotone minimal perfect hashing, 2023. doi:10.48550/arXiv.2304.11012.

20 Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. On the performance of learned data structures. *Theor. Comput. Sci.*, 871:107–120, 2021. doi:10.1016/J.TCS.2021.04.015.

21 Paolo Ferragina, Giovanni Manzini, and Giorgio Vinciguerra. Compressing and querying integer dictionaries under linearities and repetitions. *IEEE Access*, 10:118831–118848, 2022. doi:10.1109/ACCESS.2022.3221520.

22 Paolo Ferragina and Gonzalo Navarro. Pizza&Chili corpus. Accessed: February 2023. URL: http://pizzachili.dcc.uchile.cl/texts.html.

23 Paolo Ferragina and Giorgio Vinciguerra. Learned data structures. In Luca Oneto, Nicolò Navarin, Alessandro Sperduti, and Davide Anguita, editors, *Recent Trends in Learning From Data*, pages 5–41. Springer International Publishing, 2020. doi:10.1007/978-3-030-43883-8_2.

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

25 Edward A. Fox, Qi Fan Chen, Amjad M. Daoud, and Lenwood S. Heath. Order-preserving minimal perfect hash functions and information retrieval. *ACM Trans. Inf. Syst.*, 9(3):281–308, 1991. doi:10.1145/125187.125200.

26 Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proc. 15th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 266–273, 1992. doi:10.1145/133160.133209.

27 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.

28 LeMonHash - GitHub. https://github.com/ByteHamster/LeMonHash, 2023.

29 MMPHF-Experiments - GitHub. https://github.com/ByteHamster/MMPHF-Experiments, 2023.

30 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.

31 Google. Google ngram exports. Accessed: March 2023. URL: https://storage.googleapis.com/books/ngrams/books/datasetsv3.html.

32 Roberto Grossi, Alessio Orlandi, and Rajeev Raman. Optimal trade-offs for succinct string indexes. In *Proc. 37th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 678–689, 2010. doi:10.1007/978-3-642-14165-2_57.

33 Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. *ACM J. Exp. Algorithmics*, 19(1), 2014. doi:10.1145/2656332.

34 Guy Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.

35 Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. SOSD: A benchmark for learned indexes. *CoRR*, abs/1911.13014, 2019.

36 Evgenios M. Kornaropoulos, Silei Ren, and Roberto Tamassia. The price of tailoring the index to your data: Poisoning attacks on learned index structures. In *Proc. 48th International Conference on Management of Data (SIGMOD)*, pages 1331–1344, 2022. doi:10.1145/3514221.3517867.

37 Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proc. 44th International Conference on Management of Data (SIGMOD)*, pages 489–504, 2018. doi:10.1145/3183713.3196909.

38 Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 257–272, 2022. doi:10.1007/978-3-031-20643-6_19.

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

40 Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. SicHash - small irregular cuckoo tables for perfect hashing. In *Proc. 25th Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 176–189, 2022. doi:10.1137/1.9781611977561.ch15.

41 Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–13. ACM, 2011. doi:10.1145/2043556.2043558.

42 Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. In *Proc. 16th International Symposium on Experimental Algorithms (SEA)*, pages 25:1–25:16, 2017. doi:10.4230/LIPICS.SEA.2017.25.

43 Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. A family of perfect hashing methods. *Comput. J.*, 39(6):547–554, 1996. doi:10.1093/COMJNL/39.6.547.

44 Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and perfect hashing using fingerprinting. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 138–149, 2014. doi:10.1007/978-3-319-07959-2_12.

45 J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

46 Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):1–47, 2014. doi:10.1145/2535933.

47 Gonzalo Navarro. *Compact data structures: a practical approach.* Cambridge University Press, 2016.

48 Gonzalo Navarro and Javiel Rojas-Ledesma. Predecessor search. *ACM Comput. Surv.*, 53(5), 2020. doi:10.1145/3409371.

49 Giuseppe Ottaviano and Rossano Venturini. Partitioned Elias-Fano indexes. In *Proc. 37th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 273–282, 2014. doi:10.1145/2600428.2609615.

50 Giulio E. Pibiri and Roberto Trani. PTHash: Revisiting FCH minimal perfect hashing. In *Proc. 44th International ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 1339–1348, 2021. doi:10.1145/3404835.3462849.

51 Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. Can learned models replace hash functions? *PVLDB*, 16(3):532–545, 2022. doi:10.14778/3570690.3570702.

52 Sebastiano Vigna. Broadword implementation of rank/select queries. In *Proc. 7th International Workshop on Experimental Algorithms (WEA)*, pages 154–168. Springer, 2008. doi:10.1007/978-3-540-68552-4_12.

53 Stefan Walzer. Peeling close to the orientability threshold - spatial coupling in hashing-based data structures. In *Proc. 32nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2194–2211, 2021. doi:10.1137/1.9781611976465.131.

54 Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, 2nd edition, 1999.