

# Modern Minimal Perfect Hashing: A Survey

HANS-PETER LEHMANN, Karlsruhe Institute of Technology, Germany

THOMAS MUELLER, Independent researcher, Switzerland

RASMUS PAGH, BARC, University of Copenhagen, Denmark

GIULIO ERMANNO PIBIRI, Ca' Foscari University of Venice, Italy

PETER SANDERS, Karlsruhe Institute of Technology, Germany

SEBASTIANO VIGNA, Università degli Studi di Milano, Italy

STEFAN WALZER, Karlsruhe Institute of Technology, Germany

Given a set  $S$  of  $n$  keys, a perfect hash function for  $S$  maps the keys in  $S$  to the first  $m \geq n$  integers without collisions. It may return an arbitrary result for any key not in  $S$  and is called *minimal* if  $m = n$ . The most important parameters are its space consumption, construction time, and query time. Years of research now enable modern perfect hash functions to be extremely fast to query, very space-efficient, and scale to billions of keys. Different approaches give different trade-offs between these aspects. For example, the smallest constructions get within 0.1% of the space lower bound of  $\log_2 e$  bits per key. Others are particularly fast to query, requiring only one memory access. Perfect hashing has many applications, for example to avoid collision resolution in static hash tables, and is used in databases, bioinformatics, and stringology.

Since the last comprehensive survey in 1997, significant progress has been made. This survey covers the latest developments and provides a starting point for getting familiar with the topic. Additionally, our extensive experimental evaluation can serve as a guide to select a perfect hash function for use in applications.

**CCS Concepts:** • Theory of computation → Data compression; Bloom filters and hashing; • Information systems → Point lookups; • General and reference → Surveys and overviews.

**Additional Key Words and Phrases:** Compressed Data Structures, Minimal Perfect Hashing, Survey

## 1 Introduction

Given a set  $S \subseteq U$  of  $n$  keys drawn from a universe set  $U$  of size  $u$ , a perfect hash function (PHF) for  $S$  maps the keys in  $S$  to the first  $m \geq n$  integers  $[m] = \{0, \dots, m-1\}$  without collisions. For any key not in  $S$ , it may return an arbitrary result. Therefore, it does not need to be able to distinguish between keys in  $S$  and keys that it was not constructed with. As we will see later, this enables a constant amount of space per key regardless of the nature of keys. Most practical implementations of such functions need, in fact, less than 2 bits/key, depending on the load factor  $n/m$ . In contrast, a data structure storing a set of  $n$  keys must use at least  $\log_2(eu/n) - o(1)$  bits/key, which could be much larger than 2 when  $u \gg n$ . A perfect hash function is called *minimal* (MPHF) if  $m = n$ , in which case it represents a bijection between the input set and the first  $n$  integers. Intuitively, the smaller output range reduces the flexibility of how keys can be placed, thus reducing the number of functions that are collision-free. This increases the space consumption compared to the non-minimal case (details in a later paragraph). For an MHPF, the space lower bound is about  $\log_2 e \approx 1.443$  bits per key, and practical applications can already achieve 1.444 bits per key [110]. In contrast to cryptographic hashing, properties of the inverse function are irrelevant. We follow the practice in the literature and focus on *minimal* perfect hash functions only. Note that most of the techniques we describe have configurations to construct non-minimal perfect hash functions as well. In fact, some techniques construct a PHF first and later convert it to an MHPF (see Section 3.1).

---

Authors' Contact Information: Hans-Peter Lehmann, hans-peter.lehmann@kit.edu, Karlsruhe Institute of Technology, Germany; Thomas Mueller, Independent researcher, Switzerland; Rasmus Pagh, pagh@di.ku.dk, BARC, University of Copenhagen, Denmark; Giulio Ermanno Pibiri, giulioermanno.pibiri@unive.it, Ca' Foscari University of Venice, Italy; Peter Sanders, sanders@kit.edu, Karlsruhe Institute of Technology, Germany; Sebastiano Vigna, sebastiano.vigna@unimi.it, Università degli Studi di Milano, Italy; Stefan Walzer, stefan.walzer@kit.edu, Karlsruhe Institute of Technology, Germany.

Perfect hashing has many applications in various fields of computing, from bioinformatics to stringology. Over the past years, there has been significant progress in this research area. Many different approaches have been proposed, showing a wide variety of ideas on how to solve the perfect hashing problem.

*This Survey in Context.* Lewis and Cook [116] review early letter-based approaches [40, 44–46, 61, 74, 91, 101, 154, 159]. The last survey about minimal perfect hashing by Czech et al. [48] gives a comprehensive review of approaches until the year 1997 [4, 29, 30, 35–37, 39, 49, 54, 55, 69, 71, 72, 84, 92, 93, 98, 100, 116, 122, 128, 140, 150, 153, 156, 158, 161, 163, 168, 169, 171]. We do not go into detail about these here and instead focus on the *modern* approaches that have outperformed the former ones on modern hardware. In particular, our focus is on practical constructions, i.e., on those that have been implemented and shown to perform well in practice. As such, we also give an extensive evaluation that shows their performance in practice and helps to choose a fitting PHF for a given application. We nonetheless point out theoretical constructions when necessary.

*Space Lower Bounds.* The space lower bound for representing an MPHFs is  $n \log_2 e - O(\log n) \approx 1.443n$  bits for large  $n$  and  $u \rightarrow \infty$  [73, 122, 127]. This bound is quite simple to explain. Take a function  $f : S \rightarrow [n]$  that is minimal perfect on some set  $S$ . Because  $f$  can be evaluated with any input key from the universe and outputs only values from  $[n]$ , there can be additional input sets for which  $f$  is minimal perfect. More precisely,  $f$  is minimal perfect for all sets where exactly one input key is in each preimage  $f^{-1}(1), f^{-1}(2), \dots, f^{-1}(n)$ . The number of input sets on which  $f$  is minimal perfect is therefore  $|f^{-1}(1)| \cdot |f^{-1}(2)| \cdot \dots \cdot |f^{-1}(n)|$ . This expression is maximized if all preimages have the same size  $u/n$ . Therefore, a single function can be minimal perfect for at most  $(u/n)^n$  different input sets. There are  $\binom{u}{n}$  different possible input sets. Therefore, we need to be able to differentiate between at least  $\binom{u}{n}/(u/n)^n$  different behaviors of our MPHFs to be able to cover every possible input set. A minimal perfect hash function then needs to store which of these behaviors has to be used on the respective input set. Therefore, the number of bits needed is  $\log_2 \left( \binom{u}{n} / \left( \frac{u}{n} \right)^n \right)$  which, by using Stirling's approximation of the factorial, is  $\approx \log_2 \left( \left( \frac{ue}{n} \right)^n / \left( \frac{u}{n} \right)^n \right) - O(\log n) = n \log_2 e - O(\log n) \approx 1.443n$  for large  $n$  and  $u \rightarrow \infty$ . Using a similar combinatorial argument, one can show that the space lower bound to represent non-minimal PHFs with load factor  $\alpha = n/m$  is  $n (\log_2 e + (\frac{1}{\alpha} - 1) \log_2 (1 - \alpha)) - O(\log_2 n)$  [12, 73, 112, 128]. For example, a PHF with a load factor of  $\alpha = 0.5$  already has a lower bound of  $\approx 0.443n$  bits.

These lower bounds also hold for the *expected* space requirement of randomized constructions of MPHFs and even if the input set is chosen uniformly at random. For randomized MPHFs the lower bounds are tight as we will see for  $n = m$  in Section 6: a randomized brute force construction [128] requires  $n \log_2 e - \Omega(\log n)$  bits in expectation, even for infinite universes.

A completely independent lower bound of  $\Omega(\log \log u)$  bits in total [73] holds for *deterministic* constructions of MPHFs in the *worst case* even for  $n = 2$  because  $\log_2 u$  functions  $f : U \rightarrow [2]$  are required for each pair of elements from the universe to be separated by at least one such function. Since this lower bound does not apply to randomized constructions and exceeds the more general lower bound only if  $n = O(\log \log u)$  we leave it aside for the rest of the paper.

*Variants.* There are several variants of perfect hashing. Because these use independent techniques, we only mention them briefly but do not go into detail. *k-perfect hash functions* are a generalization that allow for up to  $k$  collisions on each output value. *Order-preserving* MPHFs retain any arbitrary order of the keys, which means that they need at least  $\log n$  bits per key due to storing one of the  $n!$  permutations. *Monotone* MPHFs, in contrast, use the *intrinsic* order of the keys as implied by an order of the universe. Surprisingly, MMPHFs achieve a much smaller space consumption of just  $O(\log \log \log u)$  bits per key [6, 9, 106]. We refer to Appendix A for details.

*Overview.* We begin our journey with an overview of different applications of perfect hashing in Section 2. In Section 3, we give an introduction to common techniques. We then categorize the modern approaches into three categories based on their main working principle in Section 4. In Sections 5 to 7 we then discuss the approaches in those categories: Perfect hashing through retrieval in Section 5, brute-force in Section 6, and fingerprinting in Section 7. In Section 8, we give theoretical details about the space consumption and construction time of different approaches. After this overview of the literature, we give a detailed practical evaluation in Section 9. There, Figure 7 can serve as a guide to select a perfect hash function in practice, given specific space and time budgets. We conclude this paper in Section 10.

## 2 Applications

In the following section, we outline some of the wide range of applications of perfect hashing.

*Hash Tables and Retrieval.* Hash tables are one of the most fundamental data structures used today. When the key set is static, a perfect hash function can be used to directly index the hash table without the need for collision resolution. This guarantees very fast access times [74] with as few cache faults as well-engineered hash tables (like, for example, Abseil’s “Swiss” hash tables [82, 83]), and fewer external memory access operations [107]. If we do not need the ability to query whether some key is stored in the table, we can store only the values and not the keys in the table cells. We thus obtain a space-efficient dictionary with a static key set and updateable values known as a *value-dynamic* retrieval data structure. As in the static version (see Section 3.2), its behavior is undefined for any key not in the original set. The best known lower and upper bounds for updateable retrieval [109] suggest that constructions based on perfect hashing are optimal. We remark that there is also work on retrieval data structures with an incremental or dynamic key set not based on perfect hashing [51, 108, 131].

*AMQ Data Structures.* An Approximate Membership Query (AMQ) data structure (also called *filter*) can answer membership queries, “Is  $x \in S?$ ”, allowing false-positive answers. The space lower bound for an AMQ data structure for  $S$  with false positive rate  $\epsilon$  is  $n \log_2(1/\epsilon)$  bits [34]. The most widely known example is the Bloom filter [22]. With the optimal number of hash functions, it needs at least  $n(\log_2(e) \cdot \log_2(1/\epsilon))$  bits to be represented [32], thus approximately 44.3% more space than the space lower bound for an AMQ data structure. Assuming a PHF  $f$  taking  $c$  bits per key, we can easily achieve an AMQ data structure with  $m(c + \lceil \log_2(1/\epsilon) \rceil)$  bits by storing  $\lceil \log_2(1/\epsilon) \rceil$ -bit fingerprints of the input keys in a packed array with  $m$  cells indexed by  $f$  [18, 32, 66, 86, 125]. For example, with an MPHFs we can get close to  $n(\log_2(e) + \log_2(1/\epsilon))$  bits, which is  $\frac{\log_2(e)}{\log_2(1/\epsilon)} \cdot 100\%$  extra space over the lower bound. Since MPHFs can have very fast construction and query time, this can be practically valuable because even more space efficient AMQ constructions based on retrieval [60] have slower queries.

*Databases.* Perfect hashing also finds use in static databases (or *stores*) to map variable-length keys (e.g., strings) from a known universe to smaller identifiers that can be stored with less space [25, 132]. For example, this is the case for Indeed Engineering [64]. The COPR database [151] uses perfect hashing for efficient search in log files. Perfect hashing can also be used to speed up finding association rules [2] in databases [38, 96]. SNIPS [135] uses perfect hashing to synchronize the content of a distributed database without communicating the full list of files that each peer stores. Perfect hashing can also be used to enable faster joins and aggregates in OLAP databases [76]. Finally, *monotone* minimal perfect hash functions can be useful as part of an algorithm for range queries in databases [107, 119].

*Bioinformatics.* The (node-centric) De Bruijn graph is a directed graph where nodes correspond to strings of length  $k$ , called  $k$ -mers, and edges model  $(k-1)$ -length overlaps between the nodes [50, 81]. Note that there is a one-to-one correspondence between a De Bruijn graph and a set of  $k$ -mers. This combinatorial object is vital to many applications in bioinformatics, such as *de novo* genome assembly [118, 141], transcriptome assembly from RNA [85], and variant calling [97]. Representing such graphs in compact space is therefore very important. Perfect hashing can be used to enable fast navigation in such graphs [41] and to support membership queries for  $k$ -mers [3, 126, 142, 143, 157]. Pibiri et al. [144] also introduce the first MPHFs specifically designed for  $k$ -mer sets. They exploit the fact that two consecutive  $k$ -mers overlap by  $k - 1$  symbols to beat the space lower bound that applies to general-purpose minimal perfect hashing, and to even achieve locality of the hash values for overlapping  $k$ -mers. Perfect hashing is also used to build a De Bruijn graph in external memory; for example to associate some metadata to each  $k$ -mer to aid construction [103], and to efficiently implement union-find data structures for  $k$ -mers [43].

*Text Indexing.* Perfect hashing can be used to index children of a node of a suffix tree [167] in constant time [14]. In general, perfect hashing can be used to store trees, for example in prefix search [10]. In a large external memory lexicon, perfect hashing can be used to ensure that the number of memory accesses is low when accessing specific words [170]. Monotone minimal perfect hashing can be used for constant-time rank queries on an input set. As a special case, one can build a monotone MPHFs on the positions of 1-bits of a bit vector [134], possibly undercutting the space consumption of storing the bit vector itself. The data structure then only supports rank queries for positions that we know have a 1-bit, which can be a common use-case. Running rank queries on an alphabet is useful to decode the Burrows-Wheeler Transform (BWT) [33] using a perfect hash function [14]. For reporting the number of occurrences of each character in a substring, Belazzougui et al. [15] store the rank of each character occurrence in an array indexed by a monotone MPHFs. From a list of documents, this then enables efficiently finding the  $k$  documents with the most occurrences of a pattern (top- $k$  retrieval) [15, 133]. Finally, in pattern matching [13, 77, 89], monotone MPHFs are applied mostly to integer sequences representing the occurrences of certain characters in a text.

*Natural Language Processing.*  $N$ -gram language models are a crucial ingredient of natural language processing, machine learning, and spell checking [124]. Pibiri and Venturini [147, 148] introduce a compressed trie representation of  $N$ -gram language models and use perfect hashing to achieve faster lookups. For real-time speech recognition at Amazon, Strimel et al. [160] introduce the compressed  $N$ -gram data structure DashHashLM, which is particularly focused on fast lookups. It computes IDs of the  $N$ -gram context using a minimal perfect hash function. In contrast to storing the IDs explicitly, this avoids the need for a more complex lookup table.

*Further Applications.* In flow lookup tables in routers [121], partially dynamic PHFs get rebuilt only if they change too much (see also Appendix A). Perfect hashing helps to perform depth-first search in large implicitly defined graphs using a value-dynamic retrieval data structure that stores one bit per node while all other data structures can reside in external memory [62]. For example, this can be used for formal verification of software via model-checking. Finally, monotone MPHFs can be used for efficient queries in encrypted data [23].

### 3 Preliminaries

In this section, we explain general tools and techniques that we need later in the paper. We also highlight basic building blocks that are used in almost every MPHFs construction.

### 3.1 Common Perfect Hashing Tools

*Fully Random Hash Functions.* In the course of this paper, we regularly need access to a fully random hash function  $h(x), x \in U$ . In fact, we often need a number of different hash functions, which we here model by adding a second parameter to  $h$ : we assume that  $h(x, s)$  is a fully random function for all values  $s \in \mathbb{N}_0 = \{0, 1, 2, \dots\}$  and for any  $x \in U$ . We also assume that  $h(x, s)$  can be evaluated in constant time for any  $x \in U$  and  $s \in \mathbb{N}_0$ . This is known as the “simple uniform hashing assumption” which is common in the literature about hashing [54, 107, 113, 115, 136, 137]. The assumption is an adequate model for practical hash functions and can be justified in theory [1, 20, 58, 162].

*Master Hashes.* Most perfect hash functions first generate an *initial, “master”, hash* using a high quality random hash function with large output range (e.g., with range  $[2^{64}]$  or  $[2^{128}]$ ).<sup>1</sup> This makes the construction mostly independent of the input distribution. Another important advantage is that the construction works with fixed-size keys (i.e., the hashes themselves) instead of the original, possibly variable-length, keys. Clearly, handling fixed-length keys aids implementation and improves performance. The idea is introduced in one of the first perfect hashing papers by Sprugnoli [159] (although initially just with a modulo operation).

*Partitioning.* A second fundamental idea introduced by Sprugnoli [159] is to apply *partitioning* to perfect hashing. In order to determine a perfect hash function for a larger key set, it is possible to first divide the input keys into smaller subsets of approximately the same size, for example using a random hash function. After determining perfect hash functions on all subsets independently, adding a prefix sum over the subset sizes to the hash values gives a perfect hash function of the overall set. While this causes some space and time overhead due to the need to store and lookup the prefix-sum values, it can help with cache locality and therefore construction performance. Furthermore, each partition can be built independently in parallel using multiple threads.

Another way to partition PHFs that avoids storing prefix sums is to construct a  $k$ -perfect hash function. PtrHash [88] takes a different approach, to avoid prefix sums, by constructing a non-minimal PHF. In particular, each partition is allocated the same output range  $r$ . If the number of partitions is  $P$ , then the function is made minimal by *remapping*, i.e., by ignoring the extra  $rP - n$  output values as explained below.

*Remapping.* Any perfect hash function with output range  $m > n$  can be converted to an MPHF. Folklore methods include marking the actual outputs in a vector of  $m$  bits and then performing a rank query using the output of the perfect hash function. An alternative is to store the unused output values in a predecessor data structure and then discount the output of the perfect hash function by the number of unused output values that precede it. Sometimes, in the first case one does not even need an additional bit vector because a special encoding of the output can be used to distinguish between used and unused entries [27]. Pibiri and Trani [145] remap the values greater than  $n$  to the unused output values in  $[n]$  instead. Therefore, only a small fraction of the keys need to query the remapping data structure. Constructing a non-minimal PHF first might be faster but might also increase space consumption and query time.

### 3.2 Retrieval Data Structures

A (*static function* [78]) *retrieval data structure* represents a function  $f : S \rightarrow \{0, 1\}^r$  that maps each key in  $S \subseteq U$  to an  $r$ -bit integer. Similarly to an MPHF, the behavior of a retrieval data structure is

---

<sup>1</sup>Examples include MurmurHash, xxHash, CityHash, and GxHash. The SMHasher test suite [5] offers a comprehensive comparison.

undefined for any key not in  $S$  and, because it does not need to differentiate between keys in  $S$  and keys not in  $S$ , it can represent  $f$  in close to  $rn$  bits.

Demaine et al. [51] point out that given an MPHF  $p$  for  $S$  it is trivial to construct a retrieval data structure: simply index a packed array of  $r$ -bit values using  $p$ . This results in a space consumption of at least  $n \cdot (r + \log_2 e)$  bits. For small  $r$  (such as  $r = 1$ ), the space overhead of at least  $\log_2 e$  bits/key is large percentagewise, while for large  $r$  (such as  $r = 16$ ), it may be acceptable. However, the packed array approach issues at least two dependent memory accesses for each query whereas many kinds of retrieval data structures issue independent memory accesses [57], which are parallelized efficiently by modern processors. Classically, three accesses for a good balance between query efficiency and space effectiveness as discussed below. The converse can be efficient: in Section 5 we use retrieval data structures to build perfect hash functions.

One of the main approaches to the construction of retrieval data structures is due to Czech et al. [47, 123], and has been the source of most development in the area. The idea is that given the function  $f$ , one fixes an integer  $k > 0$  and chooses  $k$  hash functions  $h : U \times [k] \rightarrow [m]$ ,  $m \geq n$ , and uses them to write a system of  $n$  equations in  $k$  variables  $\{w_i\}$

$$w_{h(x,0)} + w_{h(x,1)} + \cdots + w_{h(x,k-1)} = f(x) \quad \text{for all } x \in S. \quad (1)$$

Note that ‘+’ denotes the sum operator in an Abelian group. In the original proposal, it was modular arithmetic, but is now more commonly a bitwise XOR operation between bit vectors. Clearly, storing a solution  $w_0, \dots, w_{m-1}$  for this system is sufficient to compute  $f$  on all keys in  $S$ , but elements in  $U \setminus S$  will be mapped to arbitrary values. The criterion originally used to make the system solvable was to make the ratio  $n/m$  small enough that the system can be almost always put in echelon form in linear time, instead of the typical cubic time of Gaussian elimination. For example, for  $k = 2$  and any  $\varepsilon > 0$  one can obtain a structure with a space of  $\approx (2 + \varepsilon)rn$ , i.e., more than twice the optimum, whereas for  $k = 3$  the overhead decreases to  $\approx 23\%$ . These overheads come from *thresholds* for the *peelability* problem discussed in Section 3.3. Arbitrarily lower overheads can be obtained by increasing  $k$  and solving the linear system [78] – for example, for  $k = 3$  (10% overhead) or  $k = 4$  (3% overhead), as the threshold for solvability is lower [149], and tends to 1 as  $k$  grows, as we discuss in the Section 3.3. Recent retrieval data structures achieve very low space consumption: for example, Bumped Ribbon Retrieval (BuRR) [60] is also based on solving a system of linear equations where a key’s equation uses a random subset of a block of  $w$  consecutive variables, e.g. with  $w = 64$ . BuRR can achieve a space consumption of less than  $1.01rn$  and supports queries in time  $O(r)$ . We do not go into detail about its working principle here but still want to mention it because several approaches in Section 5.2 use BuRR as a black box retrieval data structure. As noted by Dietzfelbinger and Pagh [57], retrieval data structures can also be used to implement AMQ data structures by storing a fingerprint of each key in the set (as discussed in Section 2).

### 3.3 Graph Properties

Several perfect hash function constructions that we describe later are based on *peeling* and *orienting* graphs. In this section, we give an introduction to these terms.

*Peelability.* Peeling is the process of iteratively taking away nodes of degree 1 from a graph, together with their adjacent edge [27, 47, 102, 123, 130, 164, 165]. We call a graph *peelable* if it can be peeled to an empty graph. Clearly peelability in ordinary graphs is equivalent to acyclicity. However, once we move to  $k$ -uniform hypergraphs,  $k \geq 3$ , where an edge is a subset of vertices of cardinality  $k$ , the two concepts are no longer equivalent.<sup>2</sup> Peelability is important for retrieval data

<sup>2</sup>Note that early papers sometimes used the term “acyclic” for “peelable”. In fact, there are several definitions of acyclicity in a hypergraph [31].

structures because we can associate a  $k$ -uniform hypergraph with the system in Eq. (1): vertices are elements of  $[m]$ , and each equation is a hyperedge containing the vertices  $h(x, i)$ ,  $0 \leq i < k$ . It is easy to see that peelability implies that the system is in echelon form up to row swaps and column swaps, as each peeled edge is an equation containing a variable not appearing elsewhere. Peeling can be performed in linear time. As each edge is peeled due to a degree 1 vertex, the associated equation and variable index are put on a stack. After peeling, the equations and variables are popped from the stack and processed in reverse order. Each equation is solved by setting the variable to the required value, exploiting that the variable does not appear in previously solved equations. At the end of the process, equations are popped from the stack and solved backwards.

Czech et al. [47, 123] showed that a graph with  $m$  vertices,  $n$  uniformly random edges, and  $n/m \leq 1/(2+\varepsilon)$ ,  $\varepsilon > 0$  is peelable with a probability  $> 0$ . In practice, they suggest to use  $n/m = 0.467$ . For  $k$ -uniform hypergraphs,  $k \geq 3$ , there is a sharp *peelability threshold*  $\text{peel}_k$ : a hypergraph with  $m$  vertices and  $n$  uniformly random hyperedges of size  $k$  is peelable with probability  $1 - o(1)$  if  $n/m < \text{peel}_k - \varepsilon$ , and peelable with probability  $o(1)$  if  $n/m > \text{peel}_k + \varepsilon$ . For  $k = 3$  the threshold is  $\text{peel}_k \approx 0.8185$ , and for  $k > 3$  the threshold decreases, so  $k = 3$  is the most interesting case for practical applications, providing a 23% space overhead for retrieval. Molloy [130] proved ten years later that the thresholds conjectured in [123] were quite accurate. We remark that a slightly different distribution for the hyperedges yields larger peelability thresholds [164, 165], e.g.,  $\approx 0.9179$  for  $k = 3$ , equal to the orientability thresholds we discuss next, which moreover improve for larger  $k$ . This has, to our knowledge, not yet been exploited in MPHF constructions.

*Orientability.* Orientability is a weaker concept than peelability: we call a hypergraph *orientable* if we can *orient* each edge, that is, select a vertex from each edge so that no two edges share the same selected vertex. Clearly, a peelable graph is orientable, as we can select for each edge the vertex from which it was peeled.<sup>3</sup> Orientability is a well known property that is analyzed, among others, in the context of cuckoo hashing [139]. Similarly to the peelability threshold, the *orientability threshold* describes how large  $m$  has to be with respect to  $n$  to ensure that the hypergraph can be almost always oriented as  $n$  goes to infinity. For  $k \geq 3$ , this threshold is (surprisingly) equivalent to the threshold for the  $k$ -XORSAT problem [53, 149]: when considering a random system on  $F_2$  with  $m$  variables,  $n$  equations, and  $k$  variables per equation, the system is almost always solvable as  $n$  goes to infinity as long as  $n/m$  is below the  $k$ -XORSAT solvability threshold, which is the same as the orientability threshold for  $k$ -uniform hypergraphs. For example, 3-uniform hypergraphs have threshold  $\approx 0.9179$ , 4-uniform hypergraphs have threshold  $\approx 0.9768$ , and when  $k$  increases further the threshold approaches 1 [149]. The threshold has also been extended to larger fields [79]. There is no threshold for 2-XORSAT, but ordinary graphs have an orientability threshold of  $1/2$ , and below the same ratio they are peelable with positive probability.

## 4 Overview

We give an overview of the origins of perfect hashing in Appendix B. In this survey we focus on modern perfect hash function constructions. For this, we divide the approaches into three categories, which we explain in the following.

*Perfect hashing through retrieval* (Section 5) builds on retrieval data structures (see Section 3.2). Early approaches explicitly store the desired hash value for each key, leading to a space consumption of  $O(\log n)$  bits per key. Later approaches achieve constant space per key by storing the index of one of several candidate output values determined by different hash functions on the key.

---

<sup>3</sup>In general, a hypergraph is  $(k, \ell)$ -orientable if it is possible to select  $k$  vertices in each edge so that no vertex is selected by more than  $\ell$  edges. While this notion might be useful for  $k$ -perfect hashing, we do not consider it further in this paper.

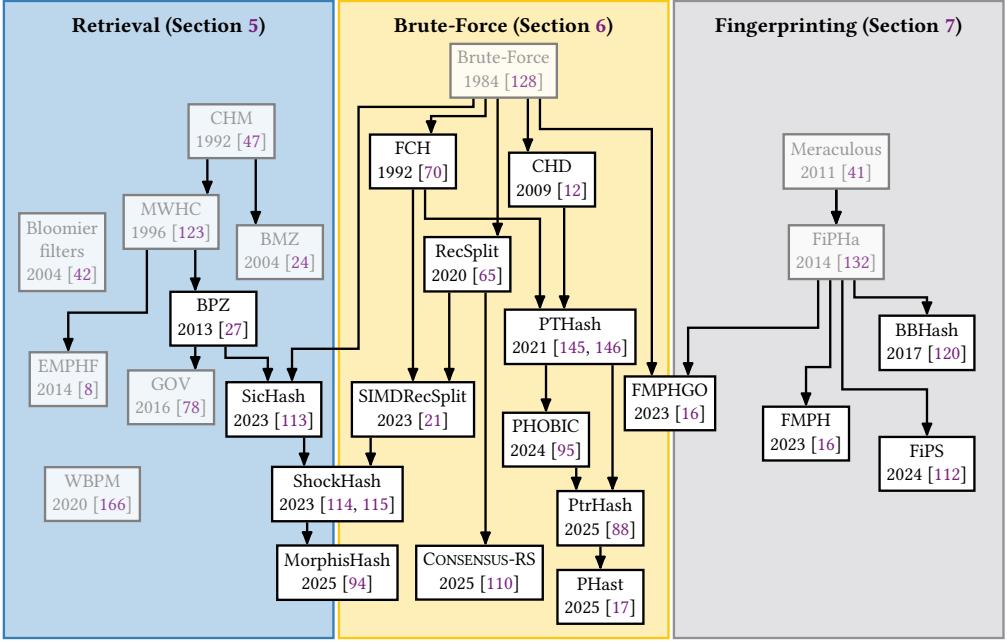


Figure 1. Perfect hashing approaches and how they influence each other. While we describe them in this paper, we do not evaluate the performance of approaches given in gray color because they are clearly larger than current ones or because they do not have a publicly available implementation.

Simple brute-force search for a PHF achieves the optimal space consumption but has exponential running time, making it impractical. In *perfect hashing through brute-force* (Section 6) we describe techniques that still use brute-force at their core, but are practical because they first reduce the size of the sets that need to be handled with brute-force.

*Perfect hashing through fingerprinting* (Section 7) describes another technique that is focused more on construction and query performance than on space consumption. There are several implementations of the same basic technique, mostly with minor algorithmic changes.

In addition to the techniques themselves, we give a comparison of the asymptotic performance in Section 8. Figure 1 gives an overview of all approaches that we consider in this survey, and shows how the ideas influence each other. We can see that the three main categories each have their own strands of research. Recent approaches start using ideas from multiple categories, as we show in the figure by placing them in between the categories. Interestingly, the number of publications in the last few years increased massively, after several years of mostly inactivity in the early 2000s. With so much new development in the field, this survey gives structure and an overview over the state of the art.

## 5 Perfect Hashing through Retrieval

We recall from Section 3.2 that a retrieval data structure represents a function  $f : S \rightarrow \{0, 1\}^r$  that maps each key in  $S \subseteq U$  to an  $r$ -bit integer. In the following, we explain a range of perfect hash function constructions that use retrieval data structures (see Section 3.2). In Section 5.1, we explain perfect hash functions that store the final output hash value in the retrieval data structure. This needs a space consumption of  $\mathcal{O}(\log n)$  bits per key. Then, in Section 5.2, we explain approaches

Table 1. Overview of perfect hashing through retrieval. We use the hypergraph interpretation here. The number  $k$  of nodes per edge corresponds to the number of choices for each key. \* The original paper [42] discusses only PHFs, not MPHFs, and with weaker bounds. We report here the actual bounds for the hypergraph technique [123]. \*\* The retrieval data structure used cannot represent all orientable graphs, so some have to be skipped.

Stored value	Approach	Year	$k$	Graph Property	$ V $	Space Usage
Output value	CHM [47]	1992	2	Acyclic	$2.09n$	$2.09 \cdot n \log_2 n$
	MWHC [123]	1996	3	Peelable	$1.23n$	$1.23 \cdot n \log_2 n$
	BMZ [24]	2004	2	$ \text{Critical}  < n/2$	$1.15n$	$1.15 \cdot n \log_2 n$
Hash function index	Bloomier filters [42]	2004	3	Peelable	$1.23n$	$2.46n^*$
	BPZ [28]	2007	3	Peelable	$1.23n$	$2.62n$
	GOV [78]	2016	3	Orientable	$1.10n$	$2.24n$
	WBPM [166]	2020	$\log_2 n$	Bipartite	$n$	$1.83n$
	SicHash [113]	2023	Mix 2, 4, 8	Orientable	$(1 + \varepsilon)n$	$2n$
	ShockHash [115]	2023	2	Orientable	$n$	$1.443n$
	MorphisHash [94]	2025	2	Orientable**	$n$	$1.443n$

where the retrieval data structures only store the index of one of multiple choices, enabling constant space per key. Some of the approaches we present here do not have an explicit name in their original paper. We therefore follow the convention of previous papers and name the approaches after the first letters of the authors' last names.

### 5.1 Storing Perfect Hash Values

Retrieval data structures can be used to explicitly store a desired perfect hash value for each key using about  $\log_2 n$  bits per key. Several early MPHFs constructions actually implement a retrieval data structure and store the desired hash values [47, 123], even if the authors do not explicitly explain their approach as a retrieval data structure. In this section, we nevertheless look at those to illustrate the progress on perfect hashing made over the past years. We give an overview in Table 1. Remember from Section 3.2 that retrieval data structures can be constructed from the following system of equations:  $w_{h(x,0)} + w_{h(x,1)} + \dots + w_{h(x,k-1)} = f(x)$  for all  $x \in S$ . All the perfect hash function constructions in this section solve a system of that form. Remember also from Section 3.3 the concepts of *peelability* (repeatedly taking away nodes of degree 1) and *orientability* (directing edges such that each node has indegree 1) of graphs.

**CHM.** Given a list of keys, Czech et al. [47] use peelability on a random graph to solve the system in Eq. (1), where  $f$  maps each key to its rank in the list. If the graph is not peelable, which rarely happens below the peelability threshold, CHM retries with another set of hash functions. The construction is a general retrieval data structure, but the authors only use it to construct an MPHFs. The space consumption is  $2.09n \log_2 n$  bits. As we noted in Section 3.3, this paper introduces the idea of peelability, that is used in many later constructions.

**MWHC.** The authors of CHM then joined forces with Wormald [123] to extend the idea of CHM to 3-uniform hypergraphs. Once again, the construction is a general retrieval data structure, but the authors use it to construct an MPHFs, lowering the space consumption to  $1.23n \log_2 n$  bits. Belazzougui et al. [8] improve the peeling step of MWHC through better cache locality.

**BMZ.** Botelho et al. [24] enhance the idea of CHM into another direction than MWHC. In contrast, the graph does not need to be peelable – only orientable. Like CHM, the hash value is given by

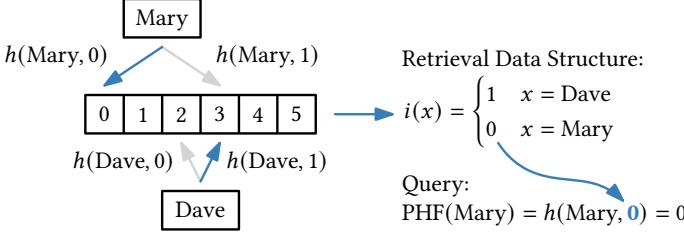


Figure 2. Multiple Choice Hashing. Each key has the choice between several locations, in this case two. A retrieval data structure stores which of the choices to take.

the sum of the values stored at the two ends of each edge. When assigning values to nodes, BMZ first uses the normal peeling process that is known from the previous approaches. When there are no more nodes of degree 1, we have arrived at the 2-core. If that 2-core consists of only nodes with degree 2, the remaining edges form loops. BMZ calls them *critical* and assigns them first using breadth-first-search. This ensures that different keys get different values, but BMZ no longer supports an arbitrary value for each key. As such, in contrast to CHM and MWHC, it is no longer a general-purpose retrieval data structure. Instead, it is a step towards smaller perfect hash functions. After the critical edges, BMZ then assigns the remaining nodes similar to the algorithms above. At all times, it keeps a list of output values that are not assigned yet to find collision-free mapping. The fact that BMZ does not require peelability has the advantage that  $|V|$  can be much closer to  $n$ . This reduces the amount of space needed to only  $1.15n \log_2 n$  bits.

## 5.2 Multiple Choice Hashing

In the following, we present several ideas that we categorize as *multiple choice hashing*. In multiple choice hashing, each input key has a set of candidate output values. Here, these candidate values are determined by evaluating different hash functions  $h(\cdot, i)$  on the input key. Let  $i(x)$  be a function assigning a number to each key, indicating which of its candidate values should be selected in order to give a collision-free mapping. Then  $h(x, i(x))$  is unique for each key  $x$ , and only the hash function index  $i(x)$  needs to be stored in a retrieval data structure. In contrast to Section 5.1, the approaches can therefore break the space consumption of  $\Omega(\log n)$  bits per key. We illustrate this idea in Figure 2. Multiple choice hashing can also be interpreted as a hypergraph, where each key corresponds to a hyperedge, connecting its candidate output values. Then an orientation of this graph gives a mapping from keys to candidate values. In this interpretation, the similarities to the older approaches in the previous section become clear. In the following, we explain approaches from the literature in more detail. Table 1 gives an overview of the differences between the approaches. We want to mention the similarity of multiple choice hashing to *cuckoo hashing* [139], which is a collision resolution strategy in hash tables. Cuckoo hashing uses the *power of multiple choices* [129] to reduce the load of hash table cells.

*Bloomier filters.* Chazelle et al. [42] introduce the concept of *Bloomier filters*, which are thought of as a generalization of Bloom filter. The setting is slightly different, as they assume that each key is representable with  $r$  bits, and they create a retrieval data structure that maps each key to itself. The purpose of the mapping is to create an approximate set membership data structure, but as in the previous case the construction can be used for general retrieval. In fact, they rediscover the idea of  $k$ -uniform hypergraph peeling introduced by Majewski et al. [123], but they use a completely different formalization, thus missing the connection, and not citing the previous results. Moreover, the peelability thresholds they provide are much weaker than the ones conjectured in [123]. However, they make an important observation that provides the first example of multiple-choice hashing: since each edge (i.e., key) is associated uniquely with the vertex it was peeled from,

we can associate a unique vertex with each key by arbitrarily numbering the vertices in each edge, and storing the function  $i$  mapping each edge  $x$  to the index of the vertex  $i(x)$  it was peeled from. This mapping just requires  $\log k$  bits and is a perfect hash function.

*BPZ.* Botelho et al. [26] rediscover the perfect hash functions described in the Bloomier filter paper [42], but they do not cite it, likely because the different formalization. However, since their goal is minimal perfect hashing, they consider the problem of how to convert the Bloomier filter [42] idea to an MPHFs. As we mentioned in the introduction, a standard technique involves a ranked bit vector: in this case, the vector would track the vertices that have been used to peel an edge, and thus are output by the perfect hash function for some input key. The authors note however that in the optimal case of 3-uniform hypergraphs one uses two bits to store three values. They suggest to store 0 as such for vertices that are not output by the perfect hash function, and to store 1, 2, or 3 for those which are. Then, after invoking the perfect hash function, one has just to count how many nonzero pairs of bits appear before the output, which can be done in constant time and  $o(n)$  space adapting standard ranking techniques [99]. In this way, they obtain a minimal perfect hash function using just  $2.62n$  bits [27].<sup>4</sup>

*GOV.* Genuzio et al. [78], reduce the space overhead of retrieval data structures and MPHFs. They exploit the fact that, as we already mentioned, the following thresholds are the same [56, 79]: (1) the threshold for orientability of  $k$ -uniform hypergraphs, (2) for solvability of random systems on  $F_2$  with  $k$  variables per equations, and (3) even for the same kind of random systems on larger fields. First they orient a random 3-uniform hypergraph, thus assigning a distinct vertex to each edge. Then they solve the system of equations on  $F_3$  representing the function that assigns to each edge the vertex selected by the orientation. To tame the cubic time of Gaussian elimination, GOV introduces a range of engineering tricks: first of all, keys are hashed into small partitions (see Section 3.1). The systems associated with each partition are first peeled as much as possible to reduce the number of equations. Then operations on equations use *broadword programming* [105] by packing multiple values into a single 64-bit word and running calculations on all of them at once. Finally, the paper introduces *lazy Gaussian elimination*, which is a heuristic for fast solution of random linear systems. Using three hash functions, and therefore two bits per key, and exploiting the same trick of Botelho et al. [26] to perform ranking on the output, they reduce the space consumption of minimal perfect hash functions to  $2.24n$  bits.<sup>5</sup>

*WBPM.* Weaver and Heule [166] use a larger number of  $O(\log n)$  candidate output values. When selecting which candidate to use for each key, they prefer choices with smaller indices. This is modeled as a weighted graph connecting each key to its candidates, where the edge of hash function  $i$  has weight  $i$ . WBPM then determines a minimum weight bipartite perfect matching (WBPM), which can be shown to reach a weight of  $1.83n$ . The selected edge indicates which hash function to use for each key. WBPM stores the hash function indices bit-by-bit in unary coding using a 1-bit retrieval data structure. The matching weight of  $1.83n$  therefore also equals the space consumption of the final data structure, except for overheads like prefix sums due to partitioning.

*SicHash.* Most earlier approaches couple the task of retrieval and perfect hashing, which means that the approaches have to store the same number of bits for every input key. SicHash [113], in contrast, separates the two tasks of key placement and retrieval. This makes it possible to efficiently use more than one retrieval data structure and store values of different widths for different keys. Using the same width for every key, even with optimal retrieval, we can achieve no better than

---

<sup>4</sup>Recall that the perfect hash function can be stored in  $2.46n$  bits but additional space is required for ranking.

<sup>5</sup>As in the case of BPZ, the additional  $0.04n$  bits are used for the ranking structure.

about 2.2 bits per key using 4 choices and remapping values to be minimal perfect. SicHash’s main innovation is using a hash function to determine how many choices each key has. Some of the keys have 2 choices, some have 4 choices, and some keys have 8 choices. As such, it stores a 1, 2, and 3-bit retrieval data structure, each handling a portion of the input keys. Using the space-efficient BuRR [60] retrieval data structure, this makes it possible to reduce the space consumption compared to previous approaches. SicHash achieves a favorable space-performance trade-off when being allowed 2–3 bits of space per key. It also achieves a rather limited gain in space efficiency by *overloading* the corresponding graph beyond the orientability threshold. Using small partitions, SicHash exploits the variance in the number of keys that can fit. Also, it uses the fact that the load factor at which construction likely fails converges to the load threshold from *above* as  $n$  grows.

*ShockHash.* ShockHash [115] can be seen as an extreme version of SicHash that directly tries to orient a graph with  $n$  nodes and  $n$  edges. Each key corresponds to one edge, connecting the two candidate output values of the key. The graph can be oriented if and only if it is a *pseudoforest* – a graph where each component contains as many edges as nodes. Because ShockHash works far above the orientability threshold, it needs to retry with many different graphs before it samples one that is orientable. ShockHash then stores the choice between the two candidate values of each key in a 1-bit retrieval data structure taking close to  $n$  bits. Additionally, it stores the hash function seed, which can be shown to need about  $0.44n$  bits. This means that the majority of the data originates from a simple linear time orientation of the graph. Only  $0.44n$  bits need to be determined by exponential time brute-force. Compared to a simple brute-force technique (see Section 6), which needs  $e^n$  tries in expectation, ShockHash needs only  $(e/2)^n$  tries, while still reaching the space lower bound. We refer to Section 8 for more details on asymptotic performance. A key idea for making ShockHash practical is the introduction of a simple bit-parallel filter that checks whether all output values are hit by some key. If one output is not hit, the graph cannot be orientable. Because the construction is still exponential time, ShockHash first partitions the keys to smaller subsets using RecSplit (see Section 6.2). This makes the overall construction time linear in expectation. ShockHash has configurations that achieve 1.52 bits per key using a construction time similar to RecSplit with 1.6 bits per key.

*Bipartite ShockHash.* Bipartite ShockHash [114] enhances the ShockHash idea. The approach stores two independent hash function seeds, one for each end of the edges. During construction, it builds a pool of hash function seeds and tests all combinations of seed pairs. By hashing each end of an edge to disjoint output ranges, the hash function pool can be filtered before building the pairs, which enables additional exponential speedups. Bipartite ShockHash-RS achieves a space consumption of 1.489 bits per key. In addition, bipartite ShockHash introduces a variant called ShockHash-Flat. The approach uses a  $k$ -perfect hash function to partition the keys to smaller subsets instead of RecSplit. This leads to higher space consumption but much faster queries due to not having to traverse the splitting tree (see Section 6.2).

*MorphisHash.* ShockHash orients pseudoforests, so each component of the graph is a cycle with trees branching from it. The cycle can be oriented in two ways, which leads to about 1 bit of redundancy for every component. MorphisHash [94] constructs a special retrieval data structure that avoids this redundancy. The idea is to formulate orienting the graph as a system of linear equations indicating that each node needs an indegree of  $1 \pmod{2}$ . The solution to the equation system is then a retrieval data structure that can return the orientation of each edge. By reducing the number of columns of the equation system, MorphisHash gives a trade-off between space consumption and success probability. Because it stores many small retrieval data structures next to the hash function seeds, MorphisHash has better cache locality during queries than ShockHash.

## 6 Perfect Hashing through Brute-Force

The fact that brute-force can be useful for constructing perfect hash functions is mentioned early after the discovery of perfect hashing [44]. Today, most of the approaches achieving the best space efficiency are based on brute-force techniques. In the following, we first describe a very simple brute-force construction and an approach that uses SAT solving for more structured brute-force search. We then illustrate the practical approaches: perfect hashing through bucket placement in Section 6.1, and recursive splitting in Section 6.2.

*A Simple Brute-Force Construction.* The idea of the simple brute-force construction is to try random hash functions (identified by different seeds) until one happens to be minimal perfect [128]. The data structure simply needs to store the found seed in binary coding. We can expect to try  $n^n/n! \approx e^n$  different hash function seeds because there are  $n^n$  functions, from which  $n!$  are minimal perfect [65, 128]. This means that it reaches the space lower bound of minimal perfect hashing, but because of its exponential running time, it is not practical for large  $n$ . Still, brute-force is used as a building block on smaller sets in a range of practical constructions, which we explain below.

*SAT Solving.* Weaver and Heule [166] describes a solution based on SAT solving. The data structure consists of a simple sequence of bits, and a hash function on each key determines which of these bits (possibly flipped) should be concatenated to give the final hash function value. To determine the values of the bits in the sequence, the algorithm searches for an assignment using SAT solving. The SAT encoding is based on an *all-different* constraint and is feasible for  $n \leq 40$ . The SAT solver can detect correlations between the bits in the data structure to prune the search space more efficiently than brute-force. Still, due to its large search space of all possible functions, we categorize it as brute-force. While the construction using SAT encoding achieves space consumption very close to the lower bound, it is much slower in practice than other approaches.

### 6.1 Bucket Placement

Perfect hashing through bucket placement is achieved via a three-step algorithm: *mapping*, *ordering*, and *searching* (MOS) [12, 44, 69–72, 138, 154]. The general idea is as follows. (1) *Mapping*. Each key  $x$  is mapped to a small bucket  $B_i = \{x | b(x) = i\}$ , using a *bucket assignment function*  $b : U \rightarrow [t]$  and  $t$  buckets. (2) *Ordering*. The buckets  $\{B_i\}_{i=0}^{t-1}$  are then sorted; usually by falling size. (3) *Searching*. For each bucket  $B_i$  in the order, we determine a value  $v_i$ , using brute-force, that the final function  $f$  uses to place all the keys in the bucket to the output domain  $[n]$  *without collisions* with keys of previously processed buckets and with one another. Once the value  $v_i$  has been determined, we say that the bucket  $B_i$  has been “placed”. (4) *Encoding*. The collection of values  $\{v_i\}_{i=0}^{t-1}$  is then stored in an array  $V[i] = v_i$  and compressed with a mechanism that retains the ability to quickly retrieve  $V[i]$  for any random  $i \in [t]$ . The array  $V$  is what the MPHF data structure actually stores.

At the beginning of the search, almost all values in the output domain  $[m]$  are available, hence the first buckets are significantly easier to place. Buckets with fewer keys are easier to place as well. This is intuitively why, to accelerate the search, it is useful to sort the buckets by falling size and process them in this order. A query for a key  $x$  then only needs to retrieve  $V[i]$ , where  $i = b(x)$ , and use it to compute the resulting hash value. Figure 3 presents an illustration of bucket placement and Table 2 outlines the approaches discussed in this section.

*FCH.* FCH [70] is the first perfect hash function construction that uses the bucket placement idea. It uses  $t = \lceil cn/(1 + \log_2 n) \rceil$  buckets, where  $c > \log_2 e$  is a parameter that trades space usage for construction speed. To further amplify the effect that the first buckets are easier to place, FCH uses a bucket assignment function that produces a *skewed* distribution of bucket sizes: it hashes 60% of the keys to 30% of the buckets. The information  $v_i$  used for placing a bucket consists of two

Table 2. Overview of perfect hashing approaches using bucket placement.

Approach	Year	Bucket Assignment	Seed Compression	Search Type	Partitions
FCH [70]	1992	Skewed	Fixed width	Additive	None
CHD [12]	2009	Uniform	SDC	Double Additive	None
PTHash [145]	2021	Skewed	Multiple options	XOR	Large [146]
PHOBIC [95]	2024	Optimized	Interleaved	Additive+Retry	Small
PtrHash [88]	2025	Approximate optimized	Byte array	Cuckoo	Medium
PHast [17]	2025	Uniform	Fixed width	Windowed	None

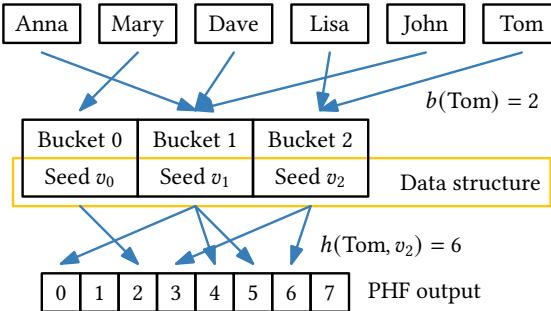


Figure 3. Illustration of perfect hashing through bucket placement, where the  $n = 6$  keys are first mapped to buckets and then, for each bucket  $i$ , a seed value  $v_i$  is computed to place the keys in the bucket without collisions to the output domain  $[m = 8]$ . Here,  $h$  has range  $[m]$ .

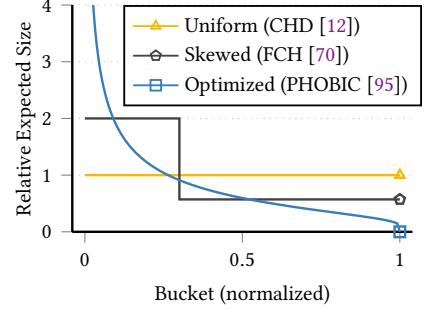


Figure 4. Relative sizes of different buckets when using various bucket mapping functions. The legend gives the original paper even if later techniques use the same functions. Adapted from [95].

parts,  $d_i$  and  $s_i$ . For each bucket  $B_i$ , a displacement value  $d_i \in [n]$  is determined so that all the keys  $x$  in  $B_i$  are placed and assigned hash values  $(h(x, s_i) + d_i) \bmod n$ . In the following (and in Table 2), we refer to this form of hash function as *additive displacement*. The seed  $s_i \in \{0, 1\}$  gives more flexibility during the search for bucket  $B_i$ : if all of the displacements cause collisions for  $s_i = 0$ , FCH repeats the search for  $s_i = 1$ . If both seeds do not work, the construction fails.

To accelerate the identification of  $d_i$ , an auxiliary data structure formed by two integer arrays, taking  $\Theta(n \log n)$  bits overall, is used to maintain the output values available in  $[n]$  during the search. Given a random available output value  $p \in [n]$ ,  $d_i$  is computed by “aligning” an arbitrary key of  $B_i$ , say  $x_i$ , to  $p$ : let  $q = h(x_i, s_i) \bmod n$ , then  $d_i = (p - q) \bmod n$ . This random alignment is rather critical for the search as it guarantees that all output values have the same probability to be occupied. On the other hand, this auxiliary structure has to be updated as well during the search, involving some extra cache misses.

Since each displacement  $d_i$  can be coded in  $\lceil \log_2 n \rceil$  bits, all the pairs  $\{(s_i, d_i)\}$  are stored in an array of  $t(1 + \lceil \log_2 n \rceil)$  bits. It follows that FCH uses about  $c$  bits/key. By decreasing the number of buckets, it is possible to lower the space usage at the cost of a larger construction time. However, in practice it is unfeasible to go below 2.5 bits/key for large values of  $n$ . On the other hand, FCH shines for its very fast query time because the pair  $(s_i, d_i)$  is coded in fixed-width. That is, FCH just spends one memory access per query (plus some inexpensive bit manipulations), which is optimal.

**CHD.** CHD [12] maps the keys uniformly to  $t = \lceil n/\lambda \rceil$  buckets, where  $\lambda > 0$  is a chosen parameter representing the average bucket size. For each bucket  $B_i$ , a displacement pair  $(d_{i,0}, d_{i,1})$

is computed to ensure that all keys  $x$  within the bucket can be placed collision-free to output values  $(h(x, 0) + d_{i,0}h(x, 1) + d_{i,1}) \bmod m$ . CHD chooses for the search an output range larger than  $n$ , hence the values are in  $[m]$ , where  $m = \lceil n/\alpha \rceil$  for a suitable load factor  $0 < \alpha < 1$ . The domain of the function is then restricted back to  $[n]$  using a rank query as we discussed in Section 3.1. Differently from FCH, CHD maintains a bitmap of  $m$  bits to mark values in  $[m]$  that are still available.

Instead of explicitly storing a displacement pair  $(d_{i,0}, d_{i,1})$  for each bucket, CHD encodes the position of the pair in the predefined sequence  $(0, 0), \dots, (0, m-1), \dots, (m-1, 0), \dots, (m-1, m-1)$ . Very importantly, CHD tries positions in increasing order starting from the smallest position, i.e., following the order  $[0, 1, 2, \dots]$ , and not at random as FCH. This intuitively helps compression because smaller numbers are tried first. The sequence of pairs' positions is then compressed within entropy bounds using the Simple Dense Coding (SDC) mechanism by Fredriksson and Nikitin [75], while permitting to retrieve an index in constant time. With  $\alpha = 0.99$  and  $\lambda = 6$ , CHD achieves 2 bits/key and it is much faster to build compared to FCH (albeit significantly slower to query).

**PTHash.** Under proper configuration, PTHash [145] combines the query time of FCH and the space effectiveness of CHD, with fast construction time. It uses the same imbalance trick of FCH by hashing 60% of the keys to 30% of  $t = \lceil cn/\log_2 n \rceil$  buckets. Like CHD, it stores the values  $\{v_i\}$  in compressed form but, rather than coding them using SDC, PTHash generalizes the encoding step to use a wide range of mechanisms (e.g., fixed-width, dictionary-based, Elias-Fano [63, 67], and others). Using an appropriate coding scheme, only a single memory access is required to find the hash value for a key and the remaining operations use simple arithmetic.

Since the first processed buckets are much easier to place, the values  $\{v_i\}$  for the first 30% of the buckets have a much smaller entropy than those for the other buckets. The *front* of the array  $V$ , of length  $\approx 0.3t$ , can thus be compressed better than the remaining *back* part. This allows PTHash to tune two different compressors for the front and the back parts respectively: a light-weight compressors for the front and a moresuccinct (but slower to query) compressor for the back part.

Like CHD, it also first generates a non-minimal function with range  $m = \lceil n/\alpha \rceil$  for some load factor  $0 < \alpha < 1$ , and uses a bitmap of  $m$  bits to mark taken output values. It then scales the output range back to  $[n]$  by remapping values larger than  $n$  using an explicit list of  $m - n$  free output values that can be effectively compressed using Elias-Fano.

To accelerate the search for the values  $\{v_i\}$ , PTHash uses the so-called *XOR displacement*: the keys of a bucket are mapped to output values given by  $(h(x, s) \oplus h(v_i, s)) \bmod m$ , where  $(A \oplus B)$  denotes the bit-wise XOR between the integers  $A$  and  $B$ , and  $s$  is a randomly chosen seed. As also noted for CHD, the values  $\{v_i\}$  are tried in linear order to improve compression, while still achieving random displacement of keys via the XOR operator. Compared to FCH, the table of random displacements is avoided altogether. Compared to CHD, instead, long clusters of already occupied slots are skipped. Also note that the seed  $s$  does not change during construction, hence all the keys can be hashed only once instead of for every seed (which is particularly important to save time when hashing long, variable-length, keys). On the other hand, this XOR-based approach, in combination with mod, only works for large output domains [95].

PTHash-HEM [146] is a parallel construction of PTHash that first partitions the input and then builds each partition (roughly holding a few million keys) independently in parallel.

**PHOBIC.** PHOBIC [95] is based on PTHash. It introduces an optimal bucket assignment function  $b(\cdot)$  that, instead of generating two expected bucket sizes as in FCH, gives a different expected size to *every* bucket. This allows each bucket to have the same probability to be placed successfully, which minimizes the construction time (in asymptotic terms) and makes the expected value of each  $v_i$  equal. The optimal bucket assignment function practically accelerates the construction compared to PTHash for large values of  $\lambda$ . Figure 4 plots the relative expected bucket sizes using different

bucket assignment functions. As apparent from the plot, PHOBIC makes the first (and easiest to place) buckets much larger than those generated by FCH.

PHOBIC uses partitioning as a key design feature. In particular, it uses small partitions (of about 2000–3000 keys), differently from PTHash-HEM. Small partitions lead to better cache locality since the bitmap used by the search is small and accesses are more localized in memory. Small partitions also enable efficient GPU acceleration. Since small partitions are created, it might be impossible to place keys without collisions using the XOR-based search. PHOBIC therefore relies on an additive displacement function and computes  $v_i = s_i \cdot P + d_i$  where  $s_i$  is a seed for the bucket  $B_i$ ,  $P$  is the size of the partition, and  $d_i \in [P]$  is the displacement value. The hash value of a key  $x$ , relative to the partition size  $P$ , is then  $(h(x, s_i) + d_i) \bmod P$ . This is similar to the function computed by FCH, but the seed  $s_i$  is not necessarily restricted in  $\{0, 1\}$ : if no displacement  $d_i \in [P]$  works for the bucket  $B_i$ , the seed  $s_i$  is incremented (and all the keys of the bucket hashed again), and the search for a new  $d_i$  continues. The pair  $(s_i, d_i)$  is computed from  $v_i$  as follows:  $s_i = \lfloor v_i / P \rfloor$  because  $d_i < P$ , and clearly  $d_i = v_i \bmod P$ . Another contribution of PHOBIC is *interleaved coding* which exploits that the values of the  $j$ -th buckets of all partitions have approximately the same distribution. This is a natural generalization of the *front-back* compression strategy of PTHash and gives PHOBIC a slight space advantage over PTHash without significantly compromising query time.

*PtrHash.* PtrHash [88] builds on both PTHash and PHOBIC with the intent of simplifying the design of the data structure to accelerate queries. The two main simplifications are: (1) a value  $v_i$  is coded using exactly 1 byte, hence PtrHash stores a plain byte array at its core; (2) the allocation of the same number of output slots for each partition. The values stored in PtrHash are therefore less than 256 by design. If none of the values  $\{0, \dots, 255\}$  is able to place a bucket, PtrHash bumps out already placed buckets in a similar way to cuckoo hashing [139]. Like PHOBIC, PtrHash partitions the input into small partitions and uses not only the same number of buckets per partition but also the same number of output slots. This has the net advantage that, during a query for key  $x$ , the size of the partition to which  $x$  belongs to does not have to be retrieved from an encoded sequence. PtrHash further uses Taylor approximations of the optimal bucket assignment function introduced in PHOBIC, which are more efficient to compute. Lastly, it supports explicit prefetching of batches of keys to increase query throughput.

*PHast.* Perfect hashing made fast (PHast) [17] simplifies several aspects of the bucket placement idea for fast queries. It uses a uniform bucket assignment function, fixed width coding of seeds, and no partitioning. This is possible due to a new search strategy for seeds. Each bucket only has a small window of possible output values, which overlaps with the windows of other buckets. Instead of taking the first collision-free seed, PHast evaluates all seeds of a bucket and takes the one giving the smallest hash values among the collision-free seeds. The order in which to place the buckets is determined by a combination of both bucket index (lowest first) and size (largest first). Therefore, taking the seed with the smallest hash values leaves more flexibility for later buckets. The few buckets that cannot be placed with any of their available seeds are bumped to a more expensive fallback data structure. PHast<sup>+</sup> designates variants that use additive placement functions. This accelerates queries and enables bit-parallel seed search, actually making PHast as fast as, or faster than, non-brute-force techniques.

## 6.2 Recursive Splitting

The simple brute-force construction explained at the beginning of this section needs  $e^n$  trials and therefore is not practical for large  $n$ . The recursive splitting idea introduced in RecSplit [65] enables space-efficiently reducing the input set into small sets of size  $\ell$ . These are then small enough that

Table 3. Overview of approaches using recursive splitting. The base case column gives the technique used after splitting the input set to small subsets of size  $\ell$ . We also include ShockHash [114, 115] again, even though it is actually explained in Section 6.1, because it uses recursive splitting to partition the keys.

Approach	Year	$\ell$	Base case	Seed encoding	Fanout
RecSplit [65]	2020	8–16	Brute-force	Golomb-Rice	Depending on $\ell$
SIMDRecSplit [21]	2023	8–18	Rotation fitting	Golomb-Rice	Depending on $\ell$
ShockHash-RS [115]	2023	40–60	ShockHash	Golomb-Rice	Depending on $\ell$
Bip. ShockHash-RS [114]	2023	90–120	Bip. ShockHash	Golomb-Rice	Depending on $\ell$
CONSENSUS-RS [110]	2025	1	None, $\ell = 1$	CONSENSUS	2

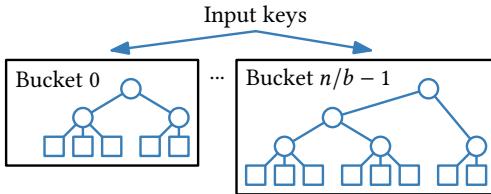


Figure 5. Illustration of the overall RecSplit data structure. Within each bucket, it constructs a splitting tree. Circular nodes represent splittings, squares represent bijections.

the brute-force construction becomes feasible. The following paragraphs introduce RecSplit, as well as improvements in later papers. Table 3 gives an overview.

*RecSplit.* RecSplit [65] uses brute-force in a novel way to enable large key sets with space close to the lower bound. It first maps all keys to buckets of expected size  $b$ , where  $b$  is a tuning parameter usually in the range 100–2000. In each bucket, it then constructs an independent *splitting tree*, as illustrated in Figure 5. The splitting tree partitions the keys into smaller and smaller sets until we arrive at the leaves which have a small configurable size  $\ell$ . At each inner node, RecSplit tries random hash functions to find one that distributes the keys to the child nodes according to a *splitting strategy*. The splitting strategy also decides the *fanout* (number of child nodes) of the tree and is optimized for balancing the work between splittings and leaves. If the number of keys in a node is not a multiple of the child node sizes, the last node might be smaller. RecSplit adjusts the hash function accordingly. The lowest level of the splitting tree is called *leaf level*. Each leaf, except for possibly the last, contains exactly  $\ell$  keys. Usual values for the leaf size are about 8–16 keys. This is small enough that it is feasible to use the simple brute-force construction. The shape of the splitting tree depends only on the leaf size  $\ell$  and the total number of keys in the bucket. Therefore, RecSplit does not need to store structural information of the tree. It is enough to store the seed for each node in DFS order, encoded using Golomb-Rice codes [80, 152]. The key theoretical observation of RecSplit is that the probability of finding an MPHF by brute force is the same as the probability of finding splittings and MPHFs for the leaves of the tree, so the space needed to represent the MPHF does not change with splitting (modulo small overheads due to approximations), but splitting makes the construction much faster. Overall, the overhead over the lower bound is constant for each node of the splitting tree.

RecSplit can be queried by traversing the splitting tree from the root to a leaf by applying the splitting hash functions. During traversal, it accumulates the number of keys stored in children to the left of the one descended into. The final hash value is then the sum of the value of leaf bijection, the number of keys to the left in the splitting tree, and the total size of previous buckets. Due to having to traverse the tree, queries are relatively slow compared to bucket placement techniques. There are configurations that need only 1.56 bits per key with reasonable construction time.

*SIMDRecSplit and GPURecSplit.* SIMDRecSplit [21] enhances RecSplit to use parallelism on multiple levels to improve the construction throughput. The paper proposes a new technique for searching for bijections called *rotation fitting*. Instead of just applying hash functions on the keys in a leaf directly, rotation fitting splits the keys into two sets using a 1-bit hash function. It then hashes each of the two sets individually, forming two words where the bits indicate which hash values are occupied. Then it tries to cyclically rotate the second word, such that the empty output values left by the first set are filled by the values of the second set. Each rotation essentially gives a new chance for a bijection [21], so it is a way to quickly evaluate additional hash function seeds in a bit-parallel way. In addition to bit parallelism, SIMDRecSplit uses parallelism on the level of words, through SIMD instructions. For this, it tests different hash function seeds or rotations simultaneously. Finally, it uses multi-threading to construct different buckets in parallel.

GPURecSplit [21] is an implementation of SIMDRecSplit on the GPU. An important observation here is that all buckets of the same size also lead to splitting trees of the same shape. Therefore, they can be constructed together using the same set of kernel calls. GPURecSplit then uses CUDA streams to construct different *tree shapes* in parallel.

*CONSENSUS-RecSplit.* Perfect hashing through brute-force often stores the seed for each subtask individually using a variable-length code like Golomb-Rice [80, 152]. With CONSENSUS [110], each seed has a fixed number of choices just barely larger than its expected number of choices. If a subtask runs out of choices without finding a successful seed, CONSENSUS backtracks to a previous subtask and looks for another successful seed. The key idea making this feasible is that hashing takes into consideration not just the current seed but the *concatenation* of all previous seeds. With this, backtracking gives a completely new chance for a seed to be successful. CONSENSUS needs less space than storing the seed of each subtask individually, even when assuming an entropy-optimal coding. CONSENSUS-RecSplit uses RecSplit with a leaf size  $\ell = 1$  which would otherwise be inefficient. However, it stores the seeds efficiently using CONSENSUS. Both RecSplit and CONSENSUS-RecSplit can achieve a space consumption of  $(1 + \epsilon) \log_2 e$  bits per key, arbitrarily close to optimal. However, while RecSplit has a running time of  $O(n \cdot e^{1/\epsilon})$ , CONSENSUS-RecSplit is the first approach to achieve this in just  $O(n/\epsilon)$ . We refer to Section 8 for details. A practical implementation achieves a space consumption of just 1.444 bits per key, very close to the lower bound of  $\log_2 e \approx 1.443$  bits per key. However, the queries are about 60% slower than RecSplit.

## 7 Perfect Hashing through Fingerprinting

The idea of perfect hashing through fingerprinting is to assign a small ( $\Theta(n)$ ) fingerprint to each input key using a hash function. We then resolve collisions between the fingerprints using recursion on the colliding keys. An advantage is the very simple and easily parallelizable construction. It is originally introduced by Chapman et al. [41] in the context of bioinformatics, but not described as a perfect hashing data structure of general interest. Müller et al. [132] then enhance and describe the idea from a data structure perspective. Table 4 gives an overview of the development.

*FIPHa.* Perfect hashing through fingerprinting [132] hashes the  $n$  keys to  $\gamma n$  fingerprints using an ordinary hash function, where  $\gamma \geq 1$  is a tuning parameter. A bit vector of length  $\gamma n$  indicates fingerprints to which exactly one key was mapped. At query time, when a key is the only one mapping to its location, a rank operation on the bit vector gives the MPHF value. The bit vector indicates with a 0-bit that the key was not the only one mapping to that location. In this case, an additional layer of the same data structure needs to be queried. Figure 6 illustrates this idea. The most space-efficient choice  $\gamma = 1$  leads to a space consumption of  $e$  bits per key [132], which is quite far from the lower bound of  $\log_2 e$  bits per key. In this case, a query traverses  $e$  layers of the

Table 4. Overview of perfect hashing through fingerprinting.

Approach	Year	New idea
Meraculous [41]	2011	Original idea
FiPHa [132]	2014	First description as data structure
BBHash [120]	2017	First public implementation
FMPH [16]	2023	Partitioning for faster construction
FMPHGO [16]	2023	Brute-Force trials for fewer levels
FiPS [112]	2024	Sorting for faster construction, interleaved rank data structure

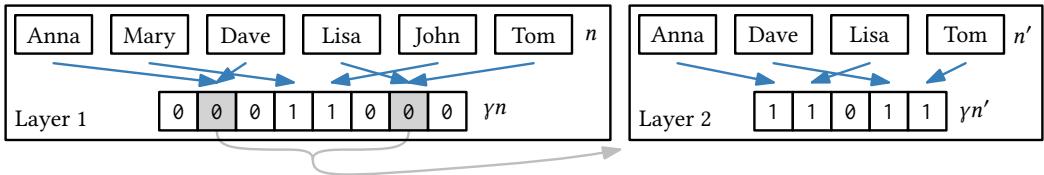


Figure 6. Illustration of PHFs using fingerprinting. Collisions are handled in the next layer.

data structure in expectation. FiPHa was developed in cooperation with the German company SAP, and the source code is not publicly available.

*BBHash.* The first publicly available implementation, BBHash [120], iterates over all keys to count the collisions. Then it iterates over the keys again to write the fingerprint bit vector and to extract colliding keys. During construction, this causes up to three random memory accesses per key and level, while we will later see that FiPS [112] achieves much higher cache locality. For parallelization, BBHash uses a large number of atomic operations in the arrays. While FiPHa already introduces the idea to scale the bit vector of fingerprints, BBHash makes this more explicit by introducing the  $\gamma$  parameter that we already used in the description of FiPHa.

*FMPH.* FMPH [16] is a fast implementation of the idea in the Rust programming language. The approach still uses the construction algorithm with two passes. For parallelization, FMPH distributes the keys to multiple threads and relies on a large number of atomic operations, just like BBHash. FMPH achieves impressive speedups compared to BBHash, even though it essentially implements the same algorithm.

*FiPS.* FiPS [112] – Fingerprint Perfect hashing through Sorting – is a third implementation of the approach. In contrast to the approaches above, FiPS focuses on cache locality when filling the bit vector during construction. It does so by sorting the fingerprints and then determining collisions by a simple scan. Using integer sorting, the construction takes linear time. This approach through sorting is already described in the FiPHa paper [132] but without a publicly available implementation. Using existing sorting libraries, the construction can be parallelized efficiently. FiPS interleaves the select data structures and the bit vector. More precisely, in each cache line, FiPS stores both the bit vector indicating fingerprints that did not collide, and the number of bits set in previous cache lines. Different cache line sizes provide different trade-offs between space consumption and query performance.

*FMPHGO.* FMPH with group optimization (FMPHGO) [16] is a new spin on the fingerprinting idea, combining it with a few brute-force tries. The idea is to hash the keys to buckets like in FCH or CHD. FMPHGO then tries a (small) number of different hash functions for each bucket and

invests additional space to store which hash function should be used. It selects the hash function that causes the least collisions in the fingerprint array. Overall, this reduces the number of required layers and the required space. Compared to FMPH, FMPHGO reduces the storage space by up to 0.7 bits/key, while the query performance stays mostly the same.

## 8 Asymptotic Performance

In this section, we discuss the asymptotic space consumption, construction time, and query time of different families of approaches. This includes how the configuration parameters can be tuned to achieve a space consumption that asymptotically approaches the lower bound.

We say a perfect hash function has space overhead  $\varepsilon$  if it uses  $(1 + \varepsilon)n \log_2 e + o(n)$  bits of space (see Section 1). It is called *succinct* if  $\varepsilon \in o(1)$ . Many of the approaches we describe in this survey admit succinct configurations, such as PTHash [145, 146], PHOBIC [95], RecSplit [21, 65], and ShockHash [114, 115]. Here, we go one step further and discuss the  $o(1)$  term in more detail. We are specifically interested in approaches with tuning parameters that can bring  $\varepsilon$  arbitrarily close to 0 without the construction or query time increasing too much. In contrast, earlier papers mostly model the tuning parameters as constants. In Table 5, we give an overview of the asymptotic construction time depending on the parameters. Note that all approaches can trivially get linear time construction through partitioning (see Section 3.1). However, the partitions introduce some space overhead. If we want to get arbitrarily close to the lower bound, we might have to select larger partition sizes, leading to construction and query times that depend on the parameter. We only use partitioning where used in the original paper. We start with brute-force approaches that are only suitable for small input sets of size  $\ell$ .

*Simple Brute-Force.* The simple brute-force technique (see Section 6) takes time  $O(e^\ell)$  to construct. The technique reaches the space lower bound but encoding the result loses a constant number of bits. Therefore, in order to decrease the space overhead, we have to increase  $\ell$ . Queries take constant time assuming that the seed of  $\approx \ell \log_2 e$  bits fits into a machine word of size  $w$ .

*ShockHash.* ShockHash [115] (see Section 5.2) is a hybrid between brute-force and multiple choice hashing. Like brute-force, its construction time is exponential in  $\ell$ , however with a much smaller base. Bipartite ShockHash [114] reduces the base even further. Queries of ShockHash access a seed of  $\approx \ell(\log_2(e) - 1)$  bits and a 1-bit retrieval data structure. Both operations take constant time, assuming  $\ell \in O(w)$ . While ShockHash uses the BuRR [60] retrieval data structure (see Section 3.2) in practice, we use a different construction [59] here because its space overhead shrinks more quickly. In the following, we now consider techniques for larger input sets.

*Multiple Choice Hashing.* Multiple choice hashing (see Section 5.2) needs a constant number of bits/key due to storing the index of the choice. Using a smart choice of hash function indices we can bring this down to 1.83 bits/key [166]. However, no matter how we tune the number of choices for each key, classical multiple choice hashing does not reach the lower bound of  $\log_2 e$  bits/key.

*Bucket Placement.* PHOBIC [95] shows that perfect hashing through bucket placement (see Section 6.1) achieves a space consumption of  $\log_2(e) + O(\log(\lambda)/\lambda)$  bits per key. Therefore, by selecting large average bucket sizes  $\lambda$ , we can get arbitrarily close to the optimal space consumption. The construction time stays linear in  $n$ , but it increases exponentially in  $\lambda$ . Depending on the bucket assignment function and the order in which the buckets are placed, it is possible to achieve a construction time of  $O(ne^\lambda/\lambda)$ . The query time is constant. Note that PtrHash [88] uses fixed size seeds for improved query time in practice, which makes it not succinct. For a similar reason, PHast [17] likely is not succinct either. However, its space consumption as a function of  $\lambda$  is an open problem.

Table 5. Asymptotic construction and query time of different approaches. The variable  $\varepsilon$  denotes the space overhead over the lower bound. While almost all approaches have some tuning parameters, not all make it possible to get arbitrarily close to the lower bound. With  $w$ , we denote the size of a machine word. Top: Brute-force approaches for small sets of size  $\ell$ , assuming  $\ell \in O(w)$ . Bottom: Approaches for large sets of size  $n$ , assuming  $\ell, b, \lambda \in O(w)$ ,  $\varepsilon \in \omega(1/\sqrt{n})$ .

Approach	Space overhead	Construction time	Query time
Simple Brute-Force [128]	$O(\frac{1}{\ell})$	$O(e^\ell)$	$O(1)$
ShockHash [115]	$O(\frac{1}{\ell})$	$O(1.36^\ell)$	$O(1)$
Bip. ShockHash [114]	$O(\frac{1}{\ell})$	$O(1.17^\ell)$	$O(1)$
Multiple Choice [26, 78, 113, 166]	$O(1)$	$O(n)$	$O(1)$
Bucket Placement [12, 70, 95, 145, 146]	$O(\frac{1}{\lambda})$	$O\left(n \frac{e^\lambda}{\lambda}\right)$	$O(1)$
Recursive Splitting [21, 65] + any brute-force base case	$O\left(\frac{1}{\ell} + \frac{\log b}{b}\right)$	$n(e^{O(\ell)} + \text{poly}(b))$	$O\left(\log \frac{b}{\ell}\right)$
CONSENSUS-RecSplit [110]	$O(\varepsilon)$	$O(\frac{n}{\varepsilon})$	$O(\log \frac{1}{\varepsilon})$
Fingerprinting [16, 41, 112, 120, 132]	$O(1)$	$O(n)$	$O(1)$
Hagerup and Tholey [90]	$O\left(\frac{(\log \log n)^2}{\log n}\right)$	$O(n)$	$O(1)$

*Recursive Splitting.* Each split in RecSplit’s splitting tree loses a constant number of bits [65]. The original implementation uses the simple brute-force technique as its base case in each leaf. Then, the space overhead in each leaf of size  $\ell$  is in  $O(1/\ell)$ . For buckets of size  $b$ , storing the prefix sum of bucket sizes causes an additional overhead of  $O(\log(b)/b)$  bits. Queries traverse the splitting tree, which takes at least time  $O(\log(b/\ell))$ , and then evaluate the base case in constant time. Note that this assumes  $\ell, b \in O(w)$ . Constructing the splittings takes polynomial time in the bucket size, in addition to the base case construction. Note that now  $\ell$  is a tuning parameter. This means that for any constant factor  $\alpha$ , we get a space overhead of  $O(\alpha/\ell) = O(1/\ell)$ . However, the construction time of the base case is  $O(e^{\alpha\ell}) = O((e^\alpha)^\ell)$ . Therefore, any exponential base case (such as ShockHash) only gives a constant factor of space improvement when included in RecSplit.

*CONSENSUS-RecSplit.* Bucketed CONSENSUS-RecSplit [110] is currently the only known approach where the construction time increases linearly in the inverse of the space overhead. This is because its space consumption is not influenced by an exponential-time base case. Through buckets of size  $O(1/\varepsilon)$ , it achieves a query time of  $O(\log(1/\varepsilon))$ . Consider the approaches using the parameter  $\ell$  (or  $\lambda$ ). Assume the word size is  $w$ , e.g.,  $w \in \Theta(\log n)$ . If we wanted to reduce the space overhead to  $o(1/w)$  by using  $\ell \in \omega(w)$ , then not only would the construction time become a problem (after all, it is exponential in  $\ell$ ), but the seed values of  $\Theta(\ell)$  bits would also no longer fit the word size. This means that the query time would start to increase linearly in  $\ell$ . CONSENSUS-RecSplit is currently the only approach for which this is not the case.

*Fingerprinting.* The fingerprint based approaches (see Section 7) have a tuning parameter  $\gamma$  that influences the space consumption. However, when selecting the smallest possible value of  $\gamma = 1$ , we get a space consumption of  $e$  bits per key, quite far from the lower bound of  $\log_2 e$  bits per key. FMPHGO [16] introduces the idea of group optimization to improve the space consumption of fingerprinting. In a hypothetical case where we would encode the group seeds with variable length and retry until there are no collisions, we get perfect hashing through bucket placement. FMPHGO can therefore interpolate between Fingerprinting and bucket placement.

*Hagerup and Tholey.* Finally, we look at a purely theoretical result that is not included in the main part of the survey. Hagerup and Tholey [90] first reduce the range of input universe to  $[n^3]$  by repeatedly taking the keys modulo a prime number. Then they hash the input to small sets of size  $\mathcal{O}(\log n / \log \log n)$ . For each possible small input set, they use brute-force to determine a minimal perfect hash function. Then they map each actual small set to one of the pre-computed MPHFs using a lookup table. Overall, Hagerup and Tholey achieve linear construction time, constant query time, and space  $1 + o(1)$  times the lower bound. More precisely, the space overhead is  $\mathcal{O}((\log \log n)^2 / \log n)$  and therefore only depends on  $n$ . This means that in order to reduce the overhead, we have to increase the input size. In contrast, other approaches have a tuning parameter that can control the space overhead independently of  $n$ . Note that even for  $n = 2^{256}$  we have  $(\log \log n)^2 / \log n > 1$ , so the overhead is large even for astronomic sizes of  $n$ . The approach has never been implemented (and is, in fact, not properly defined for  $n < 2^{150}$  [27]).

## 9 Evaluation

In this section, we compare the performance of the different state-of-the-art perfect hash function constructions. Our detailed measurements can help to pick the most fitting perfect hash function for any given application. For this comparison, we look at the three most important parameters for minimal perfect hash functions: space consumption, construction time, and query time.

*Experimental Setup.* We run most of our experiments on an Intel i7 11700 processor with 8 cores (16 hardware threads), pinned to its base clock speed of 2.5 GHz, and supporting AVX-512. In our experiments, pinning the clock speed does not influence the relative performance of the approaches. The machine runs Rocky Linux 9.5 with Linux 5.14.0. The sizes of the L1 and L2 data caches are 48 KiB and 512 KiB per core, and the L3 cache has a size of 16 MiB. We use the GNU C++ compiler version 14.2.0 with optimization flags `-O3 -march=native` and Rust 1.85.0 with release mode and `target-cpu=native`. We also give experiments on an ARM processor in Appendix F. Many of the implementations we evaluate are actively maintained and get updated with performance improvements. Our measurements refer to their state from October 12, 2025. As input data, we use strings of uniform random length in the range [10..50] containing random characters except for the zero byte. This is because essentially all approaches support string inputs. Note that, as a first step, almost all competitors generate an initial hash of each key using a high quality hash function. This makes the remaining computation largely independent of the input distribution. The code and scripts needed to reproduce our experiments are available on GitHub under the General Public License [111]. In total, we measure 9290 data points with a cumulative duration of over 32 days.

*Dominance Maps.* Visual plots give a good understanding of the trade-offs between different perfect hash functions, especially if each approach has a wide number of configurations. In order to print all three main parameters – space consumption, construction performance, and query performance – on paper, we use a type of plot that we call *dominance map* [60, 112]. We start with a 2D projection and then color each point based on the best competitor along the third dimension. This therefore provides a “front view” of the Pareto space. Pareto optimal points are the points that are not dominated in all three dimensions simultaneously. This results in a cleaner plot because it does not include data points for dominated approaches. When having a specific time and/or space budget, dominance maps show the fastest approach given that restriction.

*Overview.* We start our evaluation with Pareto fronts and dominance maps containing a wide range of configurations for each competitor. In Sections 9.1 and 9.2 we focus on the construction time trade-off and the query time trade-off. From these configurations, we pick representative ones

for each approach, details in Appendix C. We then look at how the approaches scale in the input size in Section 9.3 and in the number of threads in Section 9.4.

### 9.1 Construction Trade-Off

We start our comparison by looking at the single-threaded construction time. Figure 7 gives both a Pareto front showing all approaches, and the dominance map described above. The dominance map plots the space consumption and the construction throughput on the axes, and colors each point with the approach that has the fastest queries. For example, with a space consumption of 4 bits per key, FiPS [112] has the fastest construction. However, by sacrificing just a little bit of construction performance by looking further down, we see that we can get the faster queries of PtrHash [88]. Because some approaches are focused on very small space consumption, we additionally give a plot that uses logarithmic  $x$ - and  $y$ -axes. In the following, we look at the different perfect hash function constructions in more detail. We focus on the construction time but also briefly mention the query time from the dominance maps. For details on query time, we refer to Section 9.2.

*Multiple Choice Hashing.* SicHash [113] is up to two times faster to construct than PTHash [145] and PHOBIC [95], up to a space consumption of about 2 bits per key. However, like most other approaches, its construction throughput stays quite far below SIMDRecSplit. Even though BPZ [27] is a lot older than the other approaches, it still holds up in terms of construction time. However, the dominance map shows that newer approaches cover it almost completely with a better trade-off. Looking at even more space-efficient approaches, mainly variants of ShockHash [115] achieve below 1.55 bits per key (0.1 bits per key overhead). Especially the logarithmic plot shows how ShockHash and bipartite ShockHash improve the space consumption significantly. Bipartite ShockHash-Flat [114] trades larger space consumption and slower construction for better query speed. MorphisHash [94], as an enhancement to ShockHash, offers similar construction time but has lower space consumption. Because its queries are slightly faster due to the integrated retrieval data structure, it mostly covers ShockHash in the dominance maps.

*Bucket Placement.* PTHash [145] and CHD [12] have a similar trade-off between construction time and space consumption. PHOBIC [95] improves the construction time of PTHash significantly through its optimized bucket assignment function. The construction time of FCH [70] is rather far from the other approaches. PtrHash [88], an extension of PTHash and PHOBIC, achieves significantly faster construction, although at a larger space consumption. Finally, PHast [17] is focused mainly on query speed but also achieves very good construction performance and small space consumption. Because of its fast queries that we will look at in Section 9.2, it fills most of the dominance map about space and construction time. Because PtrHash and PHast cover such a large range of configurations, they are a good initial choice in practice.

*Recursive Splitting.* RecSplit [65] is a construction that achieved a significant step towards the space lower bound at the time of its publication. Today, it is mostly dominated by the SIMD-parallel implementation. Just like RecSplit, SIMDRecSplit [21] is originally designed for small space consumption. We make the surprising observation that SIMDRecSplit not only wins for the most space-efficient configurations, but also has very competitive construction time for less space-efficient cases. However, being based on the splitting tree of RecSplit, it has quite slow queries, so in the dominance map, it only appears in areas that no other approaches can reach. CONSENSUS-RS [110] needs the smallest amount of space. As visible in the logarithmic versions of the plots, it significantly outperforms previous approaches in terms of space efficiency.

*Fingerprinting.* The goal of perfect hashing through fingerprinting [132] is to offer efficient queries and fast construction at the cost of larger space consumption. BBHash [120] is dominated

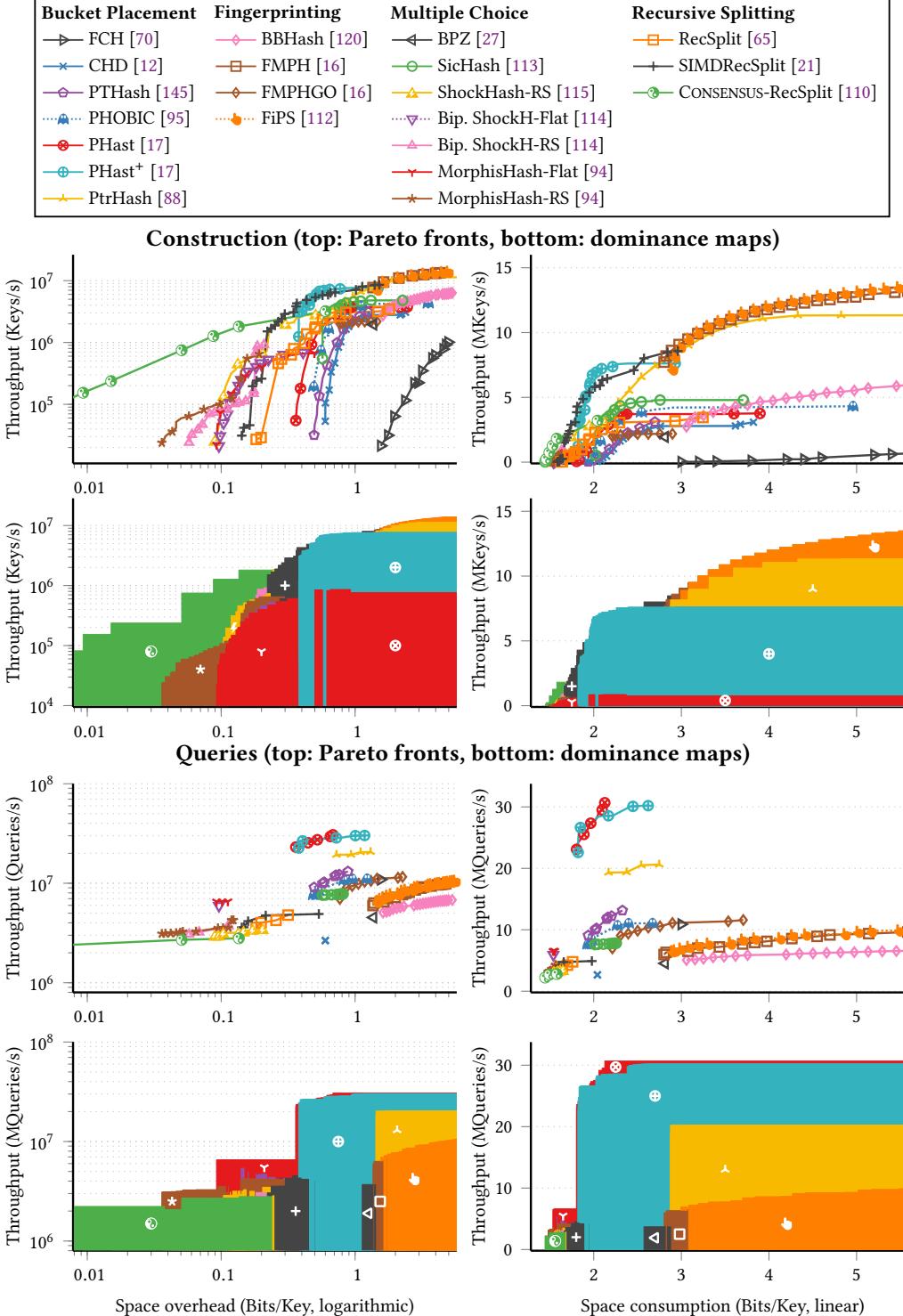


Figure 7. Trade-off construction time, query time, and space consumption. Single-threaded,  $n = 100$  million string keys of length [10..50]. We do not show all markers to increase readability. The  $x$ -axis on the left is logarithmic to the space lower bound, so an overhead of 0 bits would correspond to  $-\infty$ .

by more recent implementations. FiPS [112] and recent version of FMPH [16] achieve almost twice the construction throughput and lower minimal space consumption. FMPHGO [16] reduces the space consumption of FMPH but also has a much slower construction. Its construction performance is similar to SicHash but has slower queries.

## 9.2 Query Trade-Off

We now discuss the approaches again, this time focusing on the query time. Figure 7 also includes a Pareto front and a dominance map for the query times. The dominance map plots the space consumption and the query throughput on the axes, and colors each point with the approach that has the fastest construction. In Appendix C, we briefly consider the cache misses per query.

*Multiple Choice Hashing.* ShockHash-RS [115] and bipartite ShockHash-RS [114] use the RecSplit splitting tree but need an additional access to a retrieval data structure. However, their query performance is still very close to RecSplit. The reason is that they have fewer tree layers to traverse. It shows that the overhead of the retrieval operation is small compared to the work for traversing the heavily compressed tree. Bipartite ShockHash-Flat [114] is a variant focused on faster queries. For this, it sacrifices some of the space consumption of bipartite ShockHash-RS. For the same space consumption, it achieves 30% faster queries, which brings the query performance of very space-efficient MPHF much closer to competitors that are not focused on space consumption. For both ShockHash variants, MorphisHash [94] uses a more cache-efficient retrieval data structure but its main contribution is smaller space consumption. SicHash [113] provides a middle ground between construction and query performance.

*Bucket Placement.* PHOBIC [95] improves the construction time of PTHash [145] without sacrificing too much query performance. However, PHast [17] is the clear winner in terms of query performance. It is almost three times faster than most other competitors, and only PtrHash [88] gets close. Simultaneously, PHast achieves good space consumption. CHD [12] is much slower to query, mostly due to decoding variable-length seeds. FCH [70] has fast queries, but needs more space.

*Recursive Splitting.* RecSplit [65] and SIMDRecSplit [21] are slow to query because they have to traverse the splitting tree, decoding variable-bitlength data in each step. However, even though the operations are much more complex, the performance is still solid and not too far away from FiPS and FMPH. For extremely small space consumption, CONSENSUS-RS is the only competitor, so it fills a large area of the dominance maps. Its query performance is similar to SIMDRecSplit.

*Fingerprinting.* Perfect hashing through fingerprinting [132] is originally designed to offer fast queries. Of the different implementations, FiPS [112] and FMPH [16] offer the best query throughput, being much faster than BBHash [120]. However, it is still far away from PtrHash. This holds even for a rather large space consumption of 3.5 bits per key ( $\gamma = 2$ ), where about 73% of the keys can be handled in the first recursion layer. This indicates that the rank operation is rather costly, even when the rank data structure is interleaved with the bit vector. FMPHGO [16] achieves a smaller space consumption than the other fingerprinting approaches through some brute-force retries.

## 9.3 Scaling in the Input Size

We now look at the construction and query performance of different perfect hash functions, depending on the number of input keys  $n$ . The goal is usually to have linear construction time. To achieve this, it is always possible to partition the input to smaller MPHF. However, partitioning comes with a performance penalty, both during construction and queries. Figure 8 starts at rather small input sets of 1 million keys and goes all the way up to 200 million. In case partitions have to be used (e.g., for a parallel implementation), the plot can be used to decide on a partition size.

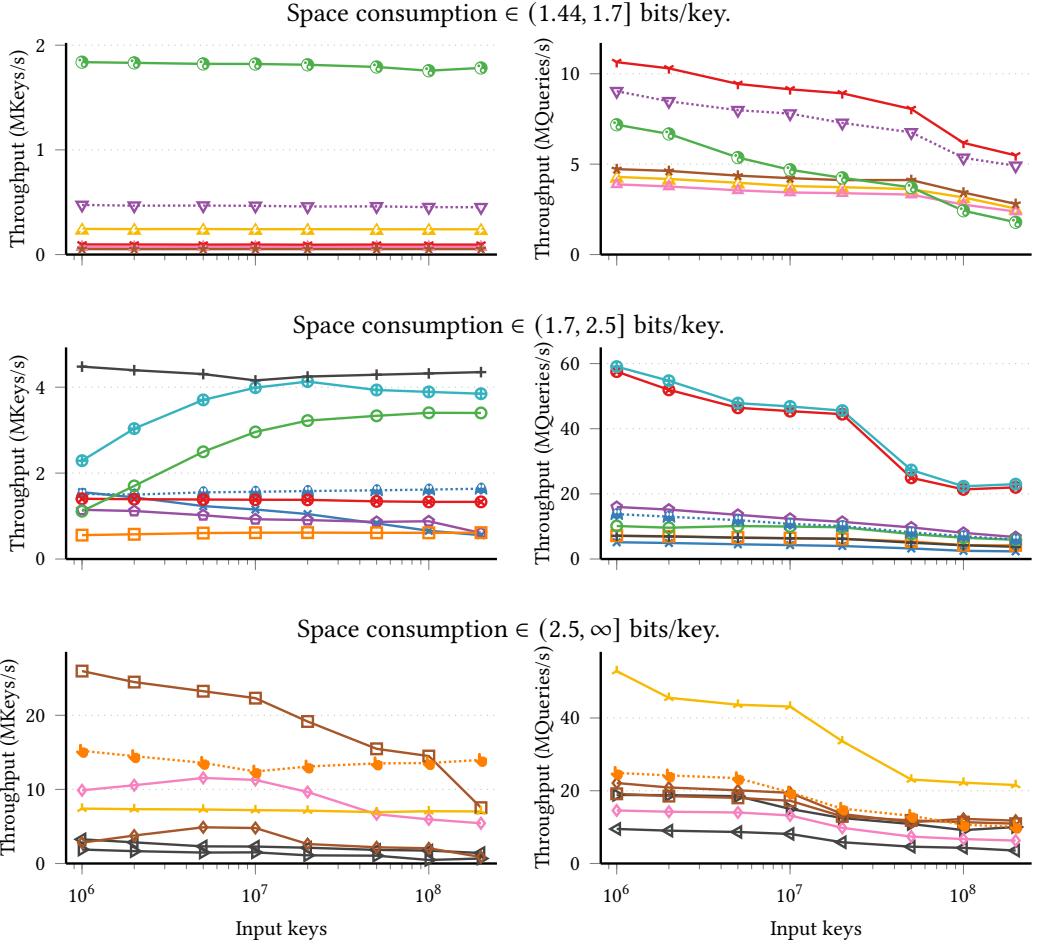
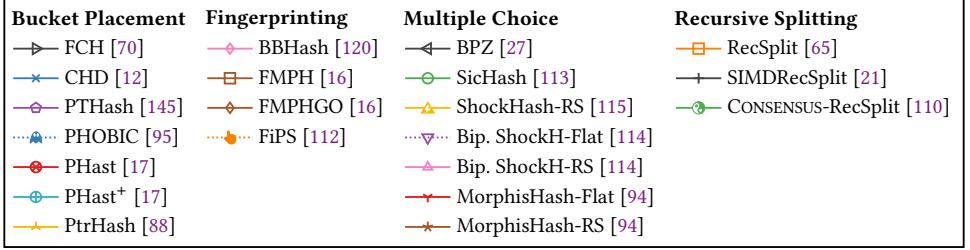


Figure 8. Comparison of construction and query performance by number of input keys  $n$ . Note that each approach has configuration parameters giving a wide trade-off between space, construction performance, and query performance. For the configurations used here, we refer to Appendix C.

For each approach, we use a configuration that is typical for it. Therefore, all competitors have a different space consumption and construction time to avoid using unfair configurations that approaches are not designed for. Due to the different space consumption, we give different plots for different space consumption. Refer to Appendix C for the exact configurations we use.

*Construction.* Figure 8 shows how the construction throughput scales in the number of input keys  $n$ . The throughput of CHD [12], BBHash [120], and FMPH [16] decreases most for larger input sizes. For the approaches based on fingerprinting, the slowdown can be explained by the random accesses to the large bit vector. FiPS [112] avoids these cache inefficient access patterns through the use of sorting, making it less sensitive to input size changes. SicHash and PHast<sup>+</sup> have a large startup overhead and become relatively faster to construct with more keys. The remaining approaches scale almost linearly in  $n$ .

*Queries.* The query throughput of all approaches in Figure 8 drops when increasing the input size  $n$ . This is expected because the data structures get larger and do not fit in cache. In general, all approaches perform pretty well regarding their query time. For techniques below 2.5 bits/key, PHast [17] by far remains the approach with the fastest queries for the entire range of input sizes. PtrHash [88] comes close for large  $n$ . The query performance of PtrHash and PHast drops significantly when passing  $10^7$  keys. This can be explained by the fact that PtrHash queries are close to the RAM throughput. As the data structure becomes larger than the cache, the throughput drops. Due to PtrHash needing more space than PHast, it drops earlier. RecSplit based approaches (RecSplit [65], SIMDRecSplit [21], ShockHash-RS [114, 115], MorphisHash [94], CONSENSUS-RS [110]) have much slower queries for the entire range of input sizes but offer faster construction. Over a large range of input sizes, their query performance has a very small slope. This indicates that they are more limited by the computation than memory access.

#### 9.4 Multi-Threaded Construction

Modern processors have many cores available, and testing single-threaded code leaves a lot of processing power unused. Additionally, most data structures like perfect hash functions are not used in isolation in actual applications. There are always other processes running on the machines. Performing multi-threaded measurements can, to a certain extent, account for this. Note that perfect hashing can be parallelized trivially by partitioning, which many approaches use. For these approaches, a major factor to their multi-threaded scaling is how efficiently they implement their partitioning step. This makes it less interesting algorithmically and causes a bias in the measurements. We still give multi-threaded measurements because they are very relevant in applications. Additionally, because there are fewer memory channels than threads, the cache locality of approaches is important in parallel measurements.

As a reminder, we run our experiments on an 8 core (16 hardware threads (HT)) processor with 2 memory channels. In Figure 9, we give parallel measurements using both *weak scaling* and *strong scaling*. With weak scaling, the number of keys per thread are constant at 10 million. With strong scaling, the total number of input keys is 100 million and stays the same. For perfect hashing, strong scaling might be the more important approach, trying to construct a certain perfect hash function as quickly as possible. However, with strong scaling there might be more threads than it is actually useful. Weak scaling ensures that each thread always has enough work to do. Like before, we use the configurations listed in Appendix C. As such, the absolute construction times are very different, so we only plot self-speedups. In the following, we first describe approaches with a direct parallelization before looking at the approaches using partitioning. In addition, Appendix E gives measurements on a 64 core machine with up to 1.28 billion keys.

*Direct Parallelization.* Some approaches use a parallel implementation of their internal data structures. An advantage of this technique compared to an external layer of partitioning is that it is transparent to the queries. Except for PHast [17], this generally does not seem to work well for the approaches that do it. We see that BBHash [120], PTHash [146], FMPH [16], and FMPHGO [16] only achieve a speedup of around 5 when running on 16 threads. We will later see that PTHash works

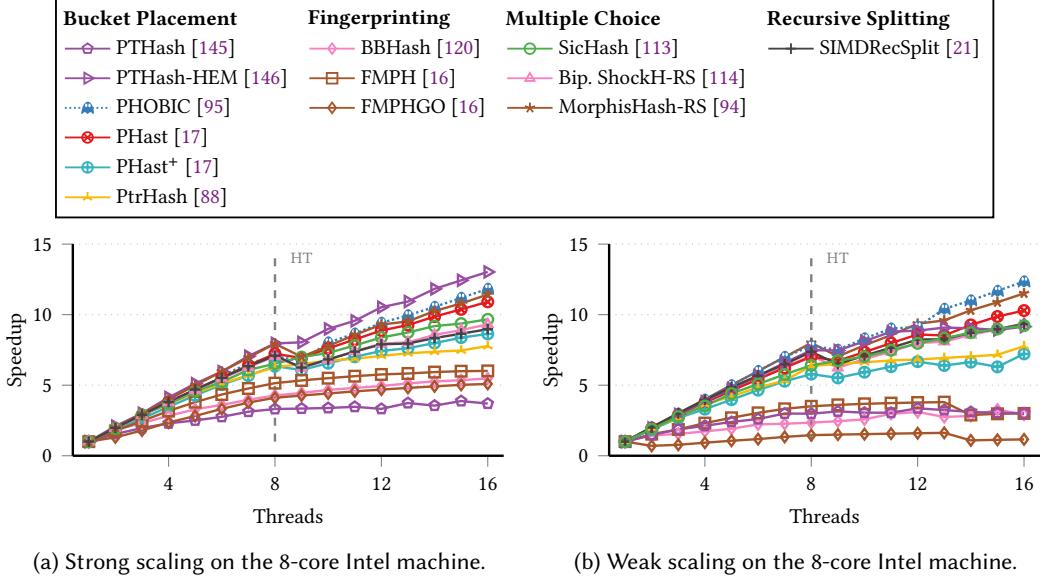


Figure 9. Multi-threaded construction by number of threads. Weak scaling with 10 million keys per thread, strong scaling with 100 million keys. We give self-speedups because each approach has a different focus.

better with partitioning (i.e., with the approach called PTHash-HEM). FMPHGO [16] uses a direct parallelization as well, even though it could use internal partitioning [16]. PHast, in contrast, places all keys of a bucket in a small window of output positions. By leaving small gaps between different threads, they can work completely independently. The gaps can be filled afterwards transparently to the queries.

*Internal Partitioning.* Some of the approaches internally partition the input anyway, so in essence they get their parallelization for free. Note that the scaling behavior of SIMDRecSplit varies strongly depending on the configuration parameters [21]. The less space-efficient configurations that it is not actually designed for scale less well because more time is spent partitioning keys to a large number of buckets. PHOBIC [95] needs internal partitioning for its interleaved coding. It scales well with strong scaling and weak scaling, which we attribute to a well implemented partitioning step. PtrHash [88] scales well but does not profit as much from hyperthreading.

*External Partitioning.* Most other approaches parallelize by adding an additional layer of partitioning, which introduces small query and construction time overheads. PTHash-HEM [146] scales best among those. SicHash [113] could theoretically use internal partitioning. However, the majority of its construction time is spent constructing the BuRR retrieval data structures covering all internal partitions. While a parallel construction if BuRR is available now [7], it was not available when SicHash was presented. It therefore uses an external partitioning step.

*GPU Parallelization.* Even though this is not the focus of this evaluation, we mention two GPU parallel constructions. PHOBIC-GPU [95] and GPURecSplit [21] are, to our knowledge, the only perfect hash functions with a GPU construction. Both approaches achieve a similar peak construction throughput of about 70 million keys per second on an Nvidia RTX 3090 GPU. In a reasonable construction time, PHOBIC-GPU can achieve a space consumption of about 1.7 bits per key, while GPURecSplit can achieve about 1.5 bits per key. The respective query implementations

are identical to the CPU versions and can only be used on the CPU, so our measurements from Section 9.2 show that PHOBIC is much faster to query. Refer to the PHOBIC paper [95] for details.

## 10 Conclusion

In this paper, we surveyed state-of-the-art perfect hash function constructions. We categorized them by their working principles into approaches based on retrieval, brute-force, and fingerprinting. Historically, retrieval based methods performed the jump from logarithmic to constant number of bits per key. For this, they tuned the number of choices per key, the retrieval data structure, load factor, and search techniques. However, right now, purely retrieval-based methods cannot get below 2 bits per key. With ShockHash and its variants, the lower bound can be reached by combining it with brute-force. Several authors implement similar variants of perfect hashing through fingerprinting. For approaches based on brute-force, PTHash combines the ideas from FCH and CHD to achieve fast queries with good space consumption. PHOBIC then optimizes this approach by determining an optimal bucket assignment function. PtrHash and PHast are variants of bucket placement radically optimized for query speed, because of which they dominate a wide range of the trade-off. Finally, we looked at the more space-efficient approaches. The simple brute-force construction is made practical through the development of RecSplit, which is the first approach that goes a significant step towards the space lower bound. SIMDRecSplit, ShockHash, and MorphisHash then tune this to search in a much more structured way. They finally get beaten again by less structured brute-force again using CONSENSUS, which achieves extremely small space consumption.

*Evaluation.* After explaining the approaches, we performed and discussed a wide range of benchmarks. We gave plots illustrating the Pareto front of the entire trade-off between construction throughput, query throughput, and space consumption. In essence, the brute-force approaches focus on showing what is possible in terms of storage space, but sacrifice on construction and query performance. The most space-efficient approach is CONSENSUS. Hash function construction through fingerprinting appealed through its simplicity and fast queries but is outperformed by more recent techniques now. Regarding query time, PHast offers the best performance, followed by PtrHash, both while also achieving rather small space. In general, over the past few years, perfect hashing has seen a large number of algorithmic improvements, leading to massive speedups and enabling larger and larger input sets.

*Future Work.* It is reasonable to assume that approaches that are more space-efficient are also slower to construct. Future work will likely give additional trade-offs and achieve even smaller space consumption. Part of this might also be enabled by additional GPU accelerated constructions. Current approaches have the tendency that the most space-efficient approaches are slower to query. There is no reason to think that this is a fundamental limitation, especially since smaller data structures should be more cache-efficient. We believe that in the future, it will be possible to construct MPHFs that are very close to the space lower bound, while still achieving fast queries. Given the large progress over the 28 years since the last survey, we are excited for the breakthroughs that the next decades might bring.

## Acknowledgments

This paper is based on and has text overlaps with the doctoral thesis of Hans-Peter Lehmann [112]. We would like to thank the anonymous reviewers for their valuable feedback.

*Funding.* H.-P.L., P.S., and S.W.: 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). This work was also supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF). R.P.: Rasmus Pagh is part of BARC, supported by the VILLUM Foundation grant 54451. G.E.P. and S.V.: This work is also partially supported by the project “SEcurity and RIghts In the CyberSpace - SERICS” (PE00000014 - CUP H73C2200089001) under the National Recovery and Resilience Plan (NRRP) funded by the European Union - NextGenerationEU. Views and opinions expressed are those of the authors only and do not necessarily reflect those of the European Union or the Italian MUR. Neither the European Union nor the Italian MUR can be held responsible for them.



## References

- [1] Anders Aamand et al. 2020. Fast hashing with strong concentration bounds. In *STOC*. ACM, 1265–1278. <https://doi.org/10.1145/3357713.3384259>
- [2] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. 1993. Mining Association Rules between Sets of Items in Large Databases. In *SIGMOD Conference*. ACM Press, 207–216. <https://doi.org/10.1145/170035.170072>
- [3] Fatemeh Almodaresi et al. 2018. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinform.* 34, 13 (2018), i169–i177. <https://doi.org/10.1093/BIOINFORMATICS/BTY292>
- [4] Michael R. Anderson and M. G. Anderson. 1979. Comments on Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets. *Commun. ACM* 22, 2 (1979), 104. <https://doi.org/10.1145/359060.359071>
- [5] Austin Appleby. 2010. SMHasher. <https://github.com/rurban/smhasher>.
- [6] Sepehr Assadi, Martin Farach-Colton, and William Kuszmaul. 2023. Tight Bounds for Monotone Minimal Perfect Hashing. In *SODA*. SIAM, 456–476. <https://doi.org/10.1137/1.9781611977554.CH20>
- [7] Matthias Becht, Hans-Peter Lehmann, and Peter Sanders. 2024. Brief Announcement: Parallel Construction of Bumped Ribbon Retrieval. *CoRR* abs/2411.12365 (2024). <https://doi.org/10.48550/ARXIV.2411.12365>
- [8] Djamel Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. 2014. Cache-Oblivious Peeling of Random Hypergraphs. In *DCC*. IEEE, 352–361. <https://doi.org/10.1109/DCC.2014.48>
- [9] Djamel Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2009. Monotone minimal perfect hashing: searching a sorted table with  $O(1)$  accesses. In *SODA*. 785–794. <https://doi.org/10.1137/1.9781611973068.86>
- [10] Djamel Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2010. Fast Prefix Search in Little Space, with Applications. In *ESA (1) (LNCS, Vol. 6346)*. Springer, 427–438. [https://doi.org/10.1007/978-3-642-15775-2\\_37](https://doi.org/10.1007/978-3-642-15775-2_37)
- [11] Djamel Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2011. Theory and practice of monotone minimal perfect hashing. *ACM J. Exp. Algorithmics* 16 (2011). <https://doi.org/10.1145/1963190.2025378>
- [12] Djamel Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. 2009. Hash, Displace, and Compress. In *ESA (LNCS, Vol. 5757)*. Springer, 682–693. [https://doi.org/10.1007/978-3-642-04128-0\\_61](https://doi.org/10.1007/978-3-642-04128-0_61)
- [13] Djamel Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. 2020. Linear-time String Indexing and Analysis in Small Space. *ACM Trans. Algorithms* 16, 2 (2020), 17:1–17:54. <https://doi.org/10.1145/3381417>
- [14] Djamel Belazzougui and Gonzalo Navarro. 2014. Alphabet-Independent Compressed Text Indexing. *ACM Trans. Algorithms* 10, 4 (2014), 23:1–23:19. <https://doi.org/10.1145/2635816>
- [15] Djamel Belazzougui, Gonzalo Navarro, and Daniel Valenzuela. 2013. Improved compressed indexes for full-text document retrieval. *J. Discrete Algorithms* 18 (2013), 3–13. <https://doi.org/10.1016/J.JDA.2012.07.005>
- [16] Piotr Beling. 2023. Fingerprinting-based Minimal Perfect Hashing Revisited. *ACM J. Exp. Algorithmics* 28 (2023), 1.4:1–1.4:16. <https://doi.org/10.1145/3596453>
- [17] Piotr Beling and Peter Sanders. 2025. PHast – Perfect Hashing made fast. *CoRR* abs/2504.17918 (2025). <https://doi.org/10.48550/arXiv.2504.17918>
- [18] Michael A. Bender et al. 2018. Bloom Filters, Adaptivity, and the Dictionary Problem. In *FOCS*. IEEE, 182–193. <https://doi.org/10.1109/FOCS.2018.00026>
- [19] Michael A. Bender, Martin Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. 2022. On the optimal time/space tradeoff for hash tables. In *STOC*. ACM, 1284–1297. <https://doi.org/10.1145/3519935.3519969>
- [20] Ioana O. Bercea, Lorenzo Beretta, Jonas Klausen, Jakob Bæk Tejs Houen, and Mikkel Thorup. 2023. Locally Uniform Hashing. In *FOCS*. IEEE, 1440–1470. <https://doi.org/10.1109/FOCS57990.2023.00089>
- [21] Dominik Bez et al. 2023. High Performance Construction of RecSplit Based Minimal Perfect Hash Functions. In *ESA (LIPIcs, Vol. 274)*. 19:1–19:16. <https://doi.org/10.4230/LIPIcs.ESA.2023.19>

- [22] 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>
- [23] Alexandra Boldyreva et al. 2011. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *CRYPTO (LNCS, Vol. 6841)*. Springer, 578–595. [https://doi.org/10.1007/978-3-642-22792-9\\_33](https://doi.org/10.1007/978-3-642-22792-9_33)
- [24] Fabiano C Botelho, David M Gomes, and Nivio Ziviani. 2004. A new algorithm for constructing minimal perfect hash functions. *differences* 100, 2 (2004), 09.
- [25] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2007. Perfect Hashing for Data Management Applications. *CoRR* abs/cs/0702159 (2007). <https://doi.org/10.48550/arXiv.cs/0702159>
- [26] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2007. Simple and Space-Efficient Minimal Perfect Hash Functions. In *WADS (LNCS, Vol. 4619)*. Springer, 139–150. [https://doi.org/10.1007/978-3-540-73951-7\\_13](https://doi.org/10.1007/978-3-540-73951-7_13)
- [27] 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>
- [28] Fabiano C. Botelho and Nivio Ziviani. 2007. External perfect hashing for very large key sets. In *CIKM*. ACM, 653–662. <https://doi.org/10.1145/1321440.1321532>
- [29] Marshall D. Brain and Alan L. Tharp. 1990. Perfect hashing using sparse matrix packing. *Inf. Syst.* 15, 3 (1990), 281–290. [https://doi.org/10.1016/0306-4379\(90\)90001-6](https://doi.org/10.1016/0306-4379(90)90001-6)
- [30] Marshall D. Brain and Alan L. Tharp. 1994. Using Tries to Eliminate Pattern Collisions in Perfect Hashing. *IEEE Trans. Knowl. Data Eng.* 6, 2 (1994), 239–247. <https://doi.org/10.1109/69.277768>
- [31] Johann Brault-Baron. 2016. Hypergraph Acyclicity Revisited. *ACM Comput. Surv.* 49, 3, Article 54 (Dec. 2016), 26 pages. <https://doi.org/10.1145/2983573>
- [32] Andrei Z. Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. *Internet Math.* 1, 4 (2003), 485–509. <https://doi.org/10.1080/15427951.2004.10129096>
- [33] Michael Burrows and D J Wheeler. 1994. A block-sorting lossless data compression algorithm. *SRC* 124 (1994).
- [34] Larry Carter, Robert W. Floyd, John Gill, George Markowsky, and Mark N. Wegman. 1978. Exact and Approximate Membership Testers. In *STOC*. ACM, 59–65. <https://doi.org/10.1145/800133.804332>
- [35] Chin-Chen Chang. 1984. An ordered minimal perfect hashing scheme based upon Euler's theorem. *Inf. Sci.* 32, 3 (1984), 165–172. [https://doi.org/10.1016/0020-0255\(84\)90032-X](https://doi.org/10.1016/0020-0255(84)90032-X)
- [36] Chin-Chen Chang and C. H. Chang. 1988. An Ordered Minimal Perfect Hashing Scheme with Single Parameter. *Inf. Process. Lett.* 27, 2 (1988), 79–83. [https://doi.org/10.1016/0020-0190\(88\)90096-8](https://doi.org/10.1016/0020-0190(88)90096-8)
- [37] Chin-Chen Chang, C. Y. Chen, and Jinn-Ke Jan. 1991. On the Design of a Machine-Independent Perfect Hashing Scheme. *Comput. J.* 34, 5 (1991), 469–474. <https://doi.org/10.1093/COMJNL/34.5.469>
- [38] Chin-Chen Chang and Chih-Yang Lin. 2005. Perfect Hashing Schemes for Mining Association Rules. *Comput. J.* 48, 2 (2005), 168–179. <https://doi.org/10.1093/COMJNL/BXH074>
- [39] Chin-Chen Chang and Tzong-Chen Wu. 1991. A Letter-oriented Perfect Hashing Scheme Based upon Sparse Table Compression. *Softw. Pract. Exp.* 21, 1 (1991), 35–49. <https://doi.org/10.1002/SPE.4380210104>
- [40] C. C. Chang. 1984. The Study of an Ordered Minimal Perfect Hashing Scheme. *Commun. ACM* 27, 4 (1984), 384–387. <https://doi.org/10.1145/358027.358051>
- [41] Jarrod A. Chapman et al. 2011. Meraculous: De Novo Genome Assembly with Short Paired-End Reads. *PLOS ONE* 6, 8 (08 2011), 1–13. <https://doi.org/10.1371/journal.pone.0023501>
- [42] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayallet Tal. 2004. The Bloomier filter: an efficient data structure for static support lookup tables. In *SODA*. SIAM, 30–39.
- [43] Rayan Chikhi, Antoine Limsabet, and Paul Medvedev. 2016. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinform.* 32, 12 (2016), 201–208. <https://doi.org/10.1093/BIOINFORMATICS/BTW279>
- [44] Richard J. Cichelli. 1980. Minimal Perfect Hash Functions Made Simple. *Commun. ACM* 23, 1 (1980), 17–19. <https://doi.org/10.1145/358808.358813>
- [45] Curtis Cook and RR Oldehoeft. 1982. A letter oriented minimal perfect hashing function. *ACM SIGPlan* (1982).
- [46] Gordon V. Cormack, R. Nigel Horspool, and Matthias Kaiserswerth. 1985. Practical Perfect Hashing. *Comput. J.* 28, 1 (1985), 54–58. <https://doi.org/10.1093/COMJNL/28.1.54>
- [47] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. 1992. An Optimal Algorithm for Generating Minimal Perfect Hash Functions. *Inf. Process. Lett.* 43, 5 (1992), 257–264. [https://doi.org/10.1016/0020-0190\(92\)90220-P](https://doi.org/10.1016/0020-0190(92)90220-P)
- [48] Zbigniew J Czech, George Havas, and Bohdan S Majewski. 1997. Perfect hashing. *Theoretical Computer Science* 182, 1-2 (1997), 1–143. [https://doi.org/10.1016/S0304-3975\(96\)00146-6](https://doi.org/10.1016/S0304-3975(96)00146-6)
- [49] Zbigniew J. Czech and Bohdan S. Majewski. 1993. A Linear Time Algorithm for Finding Minimal Perfect Hash Functions. *Comput. J.* 36, 6 (1993), 579–587. <https://doi.org/10.1093/COMJNL/36.6.579>
- [50] Nicolaas Govert De Bruijn. 1946. A combinatorial problem. *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam* 49, 7 (1946), 758–764.

- [51] Erik D. Demaine et al. 2006. De Dictionariis Dynamicis Paucō Spatio Utentibus (*lat.* On Dynamic Dictionaries Using Little Space). In *LATIN* (LNCS, Vol. 3887). Springer, 349–361. [https://doi.org/10.1007/11682462\\_34](https://doi.org/10.1007/11682462_34)
- [52] Martin Dietzfelbinger et al. 1994. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM J. Comput.* 23, 4 (1994), 738–761. <https://doi.org/10.1137/S0097539791194094>
- [53] Martin Dietzfelbinger et al. 2010. Tight Thresholds for Cuckoo Hashing via XORSAT. In *ICALP* (LNCS, Vol. 6198). Springer, 213–225. [https://doi.org/10.1007/978-3-642-14165-2\\_19](https://doi.org/10.1007/978-3-642-14165-2_19)
- [54] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. 1990. A New Universal Class of Hash Functions and Dynamic Hashing in Real Time. In *ICALP* (LNCS, Vol. 443). Springer, 6–19. <https://doi.org/10.1007/BF0032018>
- [55] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. 1992. Polynomial Hash Functions Are Reliable (Extended Abstract). In *ICALP* (LNCS, Vol. 623). Springer, 235–246. [https://doi.org/10.1007/3-540-55719-9\\_77](https://doi.org/10.1007/3-540-55719-9_77)
- [56] Martin Dietzfelbinger, Michael Mitzenmacher, and Michael Rink. 2011. Cuckoo Hashing with Pages. In *ESA* (LNCS, Vol. 6942). Springer, 615–627. [https://doi.org/10.1007/978-3-642-23719-5\\_52](https://doi.org/10.1007/978-3-642-23719-5_52)
- [57] Martin Dietzfelbinger and Rasmus Pagh. 2008. Succinct Data Structures for Retrieval and Approximate Membership. In *ICALP* (LNCS, Vol. 5125). Springer, 385–396. [https://doi.org/10.1007/978-3-540-70575-8\\_32](https://doi.org/10.1007/978-3-540-70575-8_32)
- [58] Martin Dietzfelbinger and Michael Rink. 2009. Applications of a Splitting Trick. In *ICALP* (LNCS, Vol. 5555). Springer, 354–365. [https://doi.org/10.1007/978-3-642-02927-1\\_30](https://doi.org/10.1007/978-3-642-02927-1_30)
- [59] Martin Dietzfelbinger and Stefan Walzer. 2019. Constant-Time Retrieval with O(log m) Extra Bits. In *STACS* (LIPIcs, Vol. 126). 24:1–24:16. <https://doi.org/10.4230/LIPIcs.STACS.2019.24>
- [60] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. 2022. Fast Succinct Retrieval and Approximate Membership Using Ribbon. In *SEA* (LIPIcs, Vol. 233). 4:1–4:20. <https://doi.org/10.4230/LIPIcs.SEA.2022.4>
- [61] M. W. Du, T. M. Hsieh, K. F. Jea, and D. W. Shieh. 1983. The Study of a New Perfect Hash Scheme. *IEEE Trans. Software Eng.* 9, 3 (1983), 305–313. <https://doi.org/10.1109/TSE.1983.236866>
- [62] Stefan Edelkamp, Peter Sanders, and Pavel Simecek. 2008. Semi-external LTL Model Checking. In *CAV* (LNCS, Vol. 5123). Springer, 530–542. [https://doi.org/10.1007/978-3-540-70545-1\\_50](https://doi.org/10.1007/978-3-540-70545-1_50)
- [63] Peter Elias. 1974. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM* 21, 2 (1974), 246–260. <https://doi.org/10.1145/321812.321820>
- [64] Indeed Engineering. 2018. Indeed MPH: Fast and Compact Immutable Key-Value Stores. <https://engineering.indeedblog.com/blog/2018/02/indeed-mph>.
- [65] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. 2020. RecSplit: Minimal Perfect Hashing via Recursive Splitting. In *ALENEX*. SIAM, 175–185. <https://doi.org/10.1137/1.9781611976007.14>
- [66] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *CoNEXT*. ACM, 75–88. <https://doi.org/10.1145/2674005.2674994>
- [67] Robert Mario Fano. 1971. *On the number of bits required to implement an associative memory*. Technical Report. MIT, Computer Structures Group. Project MAC, Memorandum 61”.
- [68] Paolo Ferragina, Hans-Peter Lehmann, Peter Sanders, and Giorgio Vinciguerra. 2023. Learned Monotone Minimal Perfect Hashing. In *ESA* (LIPIcs, Vol. 274). 46:1–46:17. <https://doi.org/10.4230/LIPIcs.ESA.2023.46>
- [69] Edward A. Fox, Qi Fan Chen, Amjad M. Daoud, and Lenwood S. Heath. 1991. Order-Preserving Minimal Perfect Hash Functions and Information Retrieval. *ACM Trans. Inf. Syst.* 9, 3 (1991), 281–308. <https://doi.org/10.1145/125187.125200>
- [70] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. 1992. A Faster Algorithm for Constructing Minimal Perfect Hash Functions. In *SIGIR*. ACM, 266–273. <https://doi.org/10.1145/133160.133209>
- [71] Edward A Fox, Lenwood S Heath, and Qi-Fan Chen. 1989. An O(n log n) algorithm for finding minimal perfect hash functions. (1989).
- [72] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. 1992. Practical Minimal Perfect Hash Functions for Large Databases. *Commun. ACM* 35, 1 (1992), 105–121. <https://doi.org/10.1145/129617.129623>
- [73] Michael L. Fredman and János Komlós. 1984. On the Size of Separating Systems and Families of Perfect Hash Functions. *SIAM J. Algebraic Discrete Methods* 5, 1 (1984), 61–68. <https://doi.org/10.1137/0605009>
- [74] 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>
- [75] Kimmo Fredriksson and Fedor Nikitin. 2007. Simple Compression Code Supporting Random Access and Fast String Matching. In *WEA* (LNCS, Vol. 4525). Springer, 203–216. [https://doi.org/10.1007/978-3-540-72845-0\\_16](https://doi.org/10.1007/978-3-540-72845-0_16)
- [76] Kevin P Gaffney and Jignesh M Patel. 2024. Is Perfect Hashing Practical for OLAP Systems?. In *CIDR*.
- [77] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. 2020. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *J. ACM* 67, 1 (2020), 2:1–2:54. <https://doi.org/10.1145/3375890>
- [78] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. 2016. Fast Scalable Construction of (Minimal Perfect Hash) Functions. In *SEA* (LNCS, Vol. 9685). Springer, 339–352. [https://doi.org/10.1007/978-3-319-38851-9\\_23](https://doi.org/10.1007/978-3-319-38851-9_23)
- [79] Andreas Goerdt and Lutz Falke. 2012. Satisfiability Thresholds beyond k-XORSAT. In *CSR* (LNCS, Vol. 7353). Springer, 148–159. [https://doi.org/10.1007/978-3-642-30642-6\\_15](https://doi.org/10.1007/978-3-642-30642-6_15)

- [80] Solomon W. Golomb. 1966. Run-length Encodings. *IEEE Trans. Inf. Theory* 12 (1966). <https://doi.org/10.1109/TIT.1966.1053907>
- [81] Irving John Good. 1946. Normal recurring decimals. *Journal of the London Mathematical Society* 1, 3 (1946), 167–169.
- [82] Google. 2017. Abseil. <https://abseil.io>.
- [83] Google. 2017. Abseil’s Swiss Table. <https://abseil.io/about/design/swisstable>.
- [84] Marco Gori and Giovanni Soda. 1989. An Algebraic Approach to Cichelli’s Perfect Hashing. *BIT* 29, 1 (1989), 2–13. <https://doi.org/10.1007/BF01932700>
- [85] Manfred G Grabherr et al. 2011. Full-length transcriptome assembly from RNA-Seq data without a reference genome. *Nature biotechnology* 29, 7 (2011), 644–652. <https://doi.org/10.1038/nbt.1883>
- [86] Thomas Mueller Graf and Daniel Lemire. 2020. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics (JEA)* 25 (2020), 1–16. <https://doi.org/10.1145/3376122>
- [87] Marek Greniewski and Wladyslaw M. Turski. 1963. The external language KLIPA for the URAL-2 digital computer. *Commun. ACM* 6, 6 (1963), 321–324. <https://doi.org/10.1145/366604.366654>
- [88] Ragnar Groot Koerkamp. 2025. PtrHash: Minimal Perfect Hashing at RAM Throughput. In *SEA (LIPICS, Vol. 338)*, 21:1–21:21. <https://doi.org/10.4230/LIPICS.SEA.2025.21>
- [89] Roberto Grossi, Alessio Orlandi, and Rajeev Raman. 2010. Optimal Trade-Offs for Succinct String Indexes. In *ICALP (LNCS, Vol. 6198)*. Springer, 678–689. [https://doi.org/10.1007/978-3-642-14165-2\\_57](https://doi.org/10.1007/978-3-642-14165-2_57)
- [90] Torben Hagerup and Torsten Tholey. 2001. Efficient Minimal Perfect Hashing in Nearly Minimal Space. In *STACS (LNCS, Vol. 2010)*. Springer, 317–326. [https://doi.org/10.1007/3-540-44693-1\\_28](https://doi.org/10.1007/3-540-44693-1_28)
- [91] Gary Haggard and Kevin Karplus. 1986. Finding minimal perfect hash functions. In *SIGCSE*. ACM, 191–193. <https://doi.org/10.1145/5600.5899>
- [92] George Havas and Bohdan S Majewski. 1993. *Graph theoretic obstacles to perfect hashing*. Key Centre for Software Technology, Department of Computer Science, The University of Queensland, Australia.
- [93] George Havas, Bohdan S. Majewski, Nicholas C. Wormald, and Zbigniew J. Czech. 1993. Graphs, Hypergraphs and Hashing. In *WG (LNCS, Vol. 790)*. Springer, 153–165. [https://doi.org/10.1007/3-540-57899-4\\_49](https://doi.org/10.1007/3-540-57899-4_49)
- [94] Stefan Hermann. 2025. MorphisHash: Improving Space Efficiency of ShockHash for Minimal Perfect Hashing. *CoRR* abs/2503.10161 (2025). <https://doi.org/10.48550/ARXIV.2503.10161>
- [95] Stefan Hermann et al. 2024. PHOBIC: Perfect Hashing With Optimized Bucket Sizes and Interleaved Coding. In *ESA (LIPICS, Vol. 308)*, 69:1–69:17. <https://doi.org/10.4230/LIPICS.ESA.2024.69>
- [96] Gwo-Jen Hwang, WF Tsai, and Judy CR Tseng. 2002. A minimal perfect hashing approach for mining association rules from very large databases. In *IASTED Conference*, 80–85.
- [97] Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. 2012. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature genetics* 44, 2 (2012), 226–232.
- [98] C. T. M. Jacobs and Peter van Emde Boas. 1986. Two Results on Tables. *Inf. Process. Lett.* 22, 1 (1986), 43–48. [https://doi.org/10.1016/0020-0190\(86\)90041-4](https://doi.org/10.1016/0020-0190(86)90041-4)
- [99] Guy Jacobson. 1989. Space-efficient Static Trees and Graphs. In *FOCS*. IEEE, 549–554. <https://doi.org/10.1109/SFCS.1989.63533>
- [100] G Jaeschke. 1980. On Cichelli’s minimal perfect hash functions method. *Comm. ACM* 23, 12 (1980), 728–729.
- [101] Gerhard Jaeschke. 1981. Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions. *Commun. ACM* 24, 12 (1981), 829–833. <https://doi.org/10.1145/358800.358806>
- [102] Svante Janson and Malwina J. Luczak. 2007. A simple solution to the  $k$ -core problem. *Random Struct. Algorithms* 30, 1–2 (2007), 50–62. <https://doi.org/10.1002/RSA.20147>
- [103] Jamshed Khan et al. 2022. Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2. *Genome biology* 23, 1 (2022), 190. <https://doi.org/10.1186/s13059-022-02743-6>
- [104] Donald E. Knuth. 1998. *The art of computer programming: Volume 3: Sorting and Searching*. Addison-Wesley.
- [105] Donald E. Knuth. 2011. *The Art of Computer Programming: Combinatorial algorithms. Part 1*. Vol. 4A. Addison-Wesley.
- [106] Dmitry Kosolobov. 2024. Simplified Tight Bounds for Monotone Minimal Perfect Hashing. In *CPM (LIPICS, Vol. 296)*, 19:1–19:13. <https://doi.org/10.4230/LIPICS.CPM.2024.19>
- [107] Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. 2023. PaCHash: Packed and Compressed Hash Tables. In *ALENEX*. SIAM, 162–175. <https://doi.org/10.1137/1.9781611977561.CH14>
- [108] William Kuszmaul, Aaron Putterman, Tingqiang Xu, Hangrui Zhou, and Renfei Zhou. 2025. Tight Bounds and Phase Transitions for Incremental and Dynamic Retrieval. In *SODA*. SIAM, 3974–3997. <https://doi.org/10.1137/1.9781611978322.135>
- [109] William Kuszmaul and Stefan Walzer. 2024. Space Lower Bounds for Dynamic Filters and Value-Dynamic Retrieval. In *STOC*. ACM, 1153–1164. <https://doi.org/10.1145/3618260.3649649>
- [110] Hans-Peter Lehmann et al. 2025. Combined Search and Encoding for Seeds, with an Application to Minimal Perfect Hashing. *CoRR* abs/2502.05613 (2025). <https://doi.org/10.48550/ARXIV.2502.05613>

- [111] Hans-Peter Lehmann. 2023. MPHF Experiments - GitHub. <https://github.com/ByteHamster/MPHF-Experiments>.
- [112] Hans-Peter Lehmann. 2024. *Fast and Space-Efficient Perfect Hashing*. Ph.D. Dissertation. Karlsruher Institut für Technologie (KIT). <https://doi.org/10.5445/IR/1000176432>
- [113] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. 2023. SicHash – Small Irregular Cuckoo Tables for Perfect Hashing. In *ALENEX*. SIAM, 176–189. <https://doi.org/10.1137/1.9781611977561.CH15>
- [114] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. 2024. ShockHash: Near Optimal-Space Minimal Perfect Hashing Beyond Brute-Force. *CoRR* abs/2310.14959 (2024). <https://doi.org/10.48550/ARXIV.2310.14959>
- [115] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. 2024. ShockHash: Towards Optimal-Space Minimal Perfect Hashing Beyond Brute-Force. In *ALENEX*. SIAM, 194–206. <https://doi.org/10.1137/1.9781611977929.15>
- [116] Ted G. Lewis and Curtis R. Cook. 1988. Hashing for Dynamic and Static Internal Tables. *Computer* 21, 10 (1988), 45–56. <https://doi.org/10.1109/2.7056>
- [117] Tianxiao Li, Jingxun Liang, Huacheng Yu, and Renfei Zhou. 2023. Tight Cell-Probe Lower Bounds for Dynamic Succinct Dictionaries. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6–9, 2023*. IEEE, 1842–1862. <https://doi.org/10.1109/FOCS57990.2023.00112>
- [118] Xingyu Liao, Min Li, You Zou, Fang-Xiang Wu, Yi Pan, and Jianxin Wang. 2019. Current challenges and solutions of de novo assembly. *Quant. Biol.* 7, 2 (2019), 90–109. <https://doi.org/10.1007/S40484-019-0166-9>
- [119] Hyeyontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: a memory-efficient, high-performance key-value store. In *SOSP*. ACM, 1–13. <https://doi.org/10.1145/2043556.2043558>
- [120] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. 2017. Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. In *SEA (LIPIcs, Vol. 75)*. 25:1–25:16. <https://doi.org/10.4230/LIPICS.SEA.2017.25>
- [121] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. 2006. Perfect Hashing for Network Applications. In *ISIT*. IEEE, 2774–2778.
- [122] Harry G. Mairson. 1983. The Program Complexity of Searching a Table. In *FOCS*. IEEE, 40–47. <https://doi.org/10.1109/SFCS.1983.76>
- [123] Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. 1996. A Family of Perfect Hashing Methods. *Comput. J.* 39, 6 (1996), 547–554. <https://doi.org/10.1093/COMJNL/39.6.547>
- [124] Christopher Manning and Hinrich Schütze. 1999. *Foundations of statistical natural language processing*. MIT press.
- [125] Camille Marchet et al. 2020. A resource-frugal probabilistic dictionary and applications in bioinformatics. *Discret. Appl. Math.* 274 (2020), 92–102. <https://doi.org/10.1016/J.DAM.2018.03.035>
- [126] Camille Marchet, Maël Kerbirou, and Antoine Limasset. 2021. BLight: efficient exact associative structure for k-mers. *Bioinform.* 37, 18 (2021), 2858–2865. <https://doi.org/10.1093/BIOINFORMATICS/BTAB217>
- [127] Kurt Mehlhorn. 1982. On the Program Size of Perfect and Universal Hash Functions. In *FOCS*. IEEE, 170–175. <https://doi.org/10.1109/SFCS.1982.80>
- [128] Kurt Mehlhorn. 1984. Data Structures and Algorithms, Vol. 1: Sorting and Searching. *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag (1984).
- [129] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distributed Syst.* 12, 10 (2001), 1094–1104. <https://doi.org/10.1109/71.963420>
- [130] Michael Molloy. 2005. Cores in random hypergraphs and Boolean formulas. *Random Struct. Algorithms* 27, 1 (2005), 124–135. <https://doi.org/10.1002/RSA.20061>
- [131] Christian Worm Mortensen, Rasmus Pagh, and Mihai Pătrașcu. 2005. On dynamic range reporting in one dimension. In *STOC*. ACM, 104–111. <https://doi.org/10.1145/1060590.1060606>
- [132] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. 2014. Retrieval and Perfect Hashing Using Fingerprinting. In *SEA (LNCS, Vol. 8504)*. Springer, 138–149. [https://doi.org/10.1007/978-3-319-07959-2\\_12](https://doi.org/10.1007/978-3-319-07959-2_12)
- [133] Gonzalo Navarro. 2014. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.* 46, 4 (2014), 1–47. <https://doi.org/10.1145/2535933>
- [134] Gonzalo Navarro. 2016. *Compact Data Structures - A Practical Approach*. Cambridge University Press.
- [135] Racin Nygaard and Hein Meling. 2023. SNIPS: Succinct Proof of Storage for Efficient Data Synchronization in Decentralized Storage Systems. *CoRR* abs/2304.04891 (2023). <https://doi.org/10.48550/ARXIV.2304.04891>
- [136] Anna Pagh and Rasmus Pagh. 2008. Uniform Hashing in Constant Time and Optimal Space. *SIAM J. Comput.* 38, 1 (2008), 85–96. <https://doi.org/10.1137/060658400>
- [137] Anna Pagh, Rasmus Pagh, and Milan Ruzic. 2007. Linear probing with constant independence. In *STOC*. ACM, 318–327. <https://doi.org/10.1145/1250790.1250839>
- [138] Rasmus Pagh. 1999. Hash and Displace: Efficient Evaluation of Minimal Perfect Hash Functions. In *WADS (LNCS, Vol. 1663)*. Springer, 49–54. [https://doi.org/10.1007/3-540-48447-7\\_5](https://doi.org/10.1007/3-540-48447-7_5)
- [139] 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>
- [140] Peter K Pearson. 1990. Fast hashing of variable-length text strings. *Commun. ACM* 33, 6 (1990), 677–680.

- [141] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. 2001. An Eulerian path approach to DNA fragment assembly. *Proceedings of the national academy of sciences* 98, 17 (2001), 9748–9753. <https://doi.org/10.1073/pnas.171285098>
- [142] Giulio Ermanno Pibiri. 2022. Sparse and skew hashing of K-mers. *Bioinform.* 38, Supplement\_1 (2022), i185–i194. <https://doi.org/10.1093/BIOINFORMATICS/BTAC245>
- [143] Giulio Ermanno Pibiri. 2023. On weighted k-mer dictionaries. *Algorithms Mol. Biol.* 18, 1 (2023), 3. <https://doi.org/10.1186/S13015-023-00226-2>
- [144] Giulio Ermanno Pibiri, Yoshihiro Shibuya, and Antoine Limasset. 2023. Locality-preserving minimal perfect hashing of  $k$ -mers. *Bioinform.* 39, Supplement-1 (2023), 534–543. <https://doi.org/10.1093/BIOINFORMATICS/BTAD219>
- [145] Giulio Ermanno Pibiri and Roberto Trani. 2021. PTHash: Revisiting FCH Minimal Perfect Hashing. In *SIGIR*. ACM, 1339–1348. <https://doi.org/10.1145/3404835.3462849>
- [146] Giulio Ermanno Pibiri and Roberto Trani. 2024. Parallel and External-Memory Construction of Minimal Perfect Hash Functions With PTHash. *IEEE Trans. Knowl. Data Eng.* (2024). <https://doi.org/10.1109/TKDE.2023.3303341>
- [147] Giulio Ermanno Pibiri and Rossano Venturini. 2017. Efficient Data Structures for Massive  $N$ -Gram Datasets. In *SIGIR*. ACM, 615–624. <https://doi.org/10.1145/3077136.3080798>
- [148] Giulio Ermanno Pibiri and Rossano Venturini. 2019. Handling Massive  $N$ -Gram Datasets Efficiently. *ACM Trans. Inf. Syst.* 37, 2 (2019), 25:1–25:41. <https://doi.org/10.1145/3302913>
- [149] Boris G. Pittel and Gregory B. Sorkin. 2016. The Satisfiability Threshold for  $k$ -XORSAT. *Comb. Probab. Comput.* 25, 2 (2016), 236–268. <https://doi.org/10.1017/S0963548315000097>
- [150] M. V. Ramakrishna and Per-Åke Larson. 1989. File Organization Using Composite Perfect Hashing. *ACM Trans. Database Syst.* 14, 2 (1989), 231–263. <https://doi.org/10.1145/63500.63521>
- [151] Julian Reichinger, Thomas Krismayer, and Jan S. Rellermeier. 2024. COPR – Efficient, large-scale log storage and retrieval. *CoRR* abs/2402.18355 (2024). <https://doi.org/10.48550/ARXIV.2402.18355>
- [152] Robert F. Rice. 1979. Some practical universal noiseless coding techniques. *JPL Publication* (1979).
- [153] Thomas J Sager. 1984. A new method for generating minimal perfect hash functions. (1984).
- [154] Thomas J. Sager. 1985. A Polynomial Time Generator for Minimal Perfect Hash Functions. *Commun. ACM* 28, 5 (1985), 523–532. <https://doi.org/10.1145/3532.3538>
- [155] Douglas C. Schmidt. 1990. GPERF: A Perfect Hash Function Generator. In *C++ Conference*. USENIX, 87–102.
- [156] Jeanette P. Schmidt and Alan Siegel. 1990. The Spatial Complexity of Oblivious  $k$ -Probe Hash Functions. *SIAM J. Comput.* 19, 5 (1990), 775–786. <https://doi.org/10.1137/0219054>
- [157] Sebastian Schmidt et al. 2023. Matchtigs: minimum plain text representation of  $k$ -mer sets. *Genome Biology* 24, 1 (2023), 136. <https://doi.org/10.1186/s13059-023-02968-z>
- [158] Cees F. Slot and Peter van Emde Boas. 1984. On Tape Versus Core; An Application of Space Efficient Perfect Hash Functions to the Invariance of Space. In *STOC*. ACM, 391–400. <https://doi.org/10.1145/800057.808705>
- [159] Renzo Sprugnoli. 1977. Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets. *Commun. ACM* 20, 11 (1977), 841–850. <https://doi.org/10.1145/359863.359887>
- [160] Grant P. Strimel et al. 2020. Rescore in a Flash: Compact, Cache Efficient Hashing Data Structures for n-Gram Language Models. In *INTERSPEECH*. <https://doi.org/10.21437/INTERSPEECH.2020-1939>
- [161] Robert Endre Tarjan and Andrew Chi-Chih Yao. 1979. Storing a Sparse Table. *Commun. ACM* 22, 11 (1979), 606–611.
- [162] Mikkel Thorup. 2017. Fast and powerful hashing using tabulation. *Commun. ACM* 60, 7 (2017), 94–101. <https://doi.org/10.1145/3068772>
- [163] John A. Trono. 1992. An undergraduate project to compute minimal perfect hashing functions. *ACM SIGCSE Bull.* 24, 3 (1992), 53–56. <https://doi.org/10.1145/142040.142077>
- [164] Stefan Walzer. 2021. Peeling Close to the Orientability Threshold - Spatial Coupling in Hashing-Based Data Structures. In *SODA*. SIAM, 2194–2211. <https://doi.org/10.1137/1.9781611976465.131>
- [165] Stefan Walzer. 2025. Peeling Close to the Orientability Threshold – Spatial Coupling in Hashing-Based Data Structures. *ACM Trans. Algorithms* (2025). <https://doi.org/10.1145/3711822>
- [166] Sean A. Weaver and Marijn Heule. 2020. Constructing Minimal Perfect Hash Functions Using SAT Technology. In *AAAI*. AAAI Press, 1668–1675. <https://doi.org/10.1609/AAAI.V34I02.5529>
- [167] Peter Weiner. 1973. Linear Pattern Matching Algorithms. In *SWAT*. IEEE, 1–11. <https://doi.org/10.1109/SWAT.1973.13>
- [168] Vincent G. Winters. 1990. Minimal Perfect Hashing For Large Sets of Data. In *ICCI (LNCS, Vol. 468)*. Springer, 275–284. [https://doi.org/10.1007/3-540-53504-7\\_85](https://doi.org/10.1007/3-540-53504-7_85)
- [169] Vincent G. Winters. 1990. Minimal Perfect Hashing in Polynomial Time. *BIT* 30, 2 (1990), 235–244. <https://doi.org/10.1007/BF02017345>
- [170] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann.
- [171] Wei-Pang Yang and M. W. Du. 1985. A Backtracking Method for Constructing Perfect Hash Functions from a Set of Mapping Functions. *BIT* 25, 1 (1985), 148–164. <https://doi.org/10.1007/BF01934995>

## Appendix

### A Variants

There are several variants of perfect hashing, using a different set of techniques to solve, so we only mention them here but do not go into detail. We give an illustration in Figure 10. A *k*-perfect hash function is a generalization of perfect hashing where each output value allows for at most  $k$  collisions. Minimal  $k$ -perfect hashing has a space lower bound of about  $n(\log_2(e) + \log_2(k!/k^k)/k)$  bits [12, 122]. A similar bound for non-minimal  $k$ -perfect hashing is not known [12].

An *order-preserving* MPHF maps an arbitrarily ordered list of keys to their position in the list [69]. The name is somewhat misleading as “order” usually implies an order on the universe (e.g., lexicographical order for strings). This comes with an expected space consumption of at least  $\log_2(n!)/n \approx \log_2 n - \log_2 e$  bits per key because it needs to differentiate between all  $n!$  permutations of the set. It is simple to get within  $n \log_2 e$  of this bound by constructing an MPHF and using it to index a packed array of explicitly stored positions. We can get almost optimal space consumption by storing the positions in a retrieval data structure, which we explain in Section 3.2.

A *monotone* MPHF (MMPHF) [9] preserves an *intrinsic* order of the keys implied by an order on the universe. For example, with a set of strings, the lexicographically first string would have the hash value 0, and so on. In other words, an MMPHF can answer rank queries on the input set, but without storing the set, and giving undefined results for keys not in the set. Relying on the natural order makes MMPHFs much more space-efficient than order-preserving minimal perfect hash functions. There are constructions using as few as  $O(\log \log \log u)$  bits per key [9]; recently, the tight bound  $\Theta(n \min\{\log_2 \log_2 \frac{u}{n}, \log_2 n\})$  was proven for  $u \geq (1 + \varepsilon)n$  with arbitrary  $\varepsilon > 0$  [6, 106]. Depending on the input distribution, *learned* MMPHFs can get below these worst-case lower bounds [68]. We refer to [11] for a survey on monotone MPHFs.

Finally, there has been work on dynamic perfect hashing where the key set can be changed while still avoiding collisions [51, 131]. In that case  $\Theta(n \log \log u)$  bits are necessary and sufficient [131]. If the key set is available during updates, the problem is related to augmented open addressing for which tight bounds are known [19, 117]. There is also an approach that avoids collisions as much as possible and rebuilds the perfect hash function if the key set changes too much [121]. Despite the name “dynamic perfect hashing”, Dietzfelbinger et al. [52] actually explain a dynamic set data

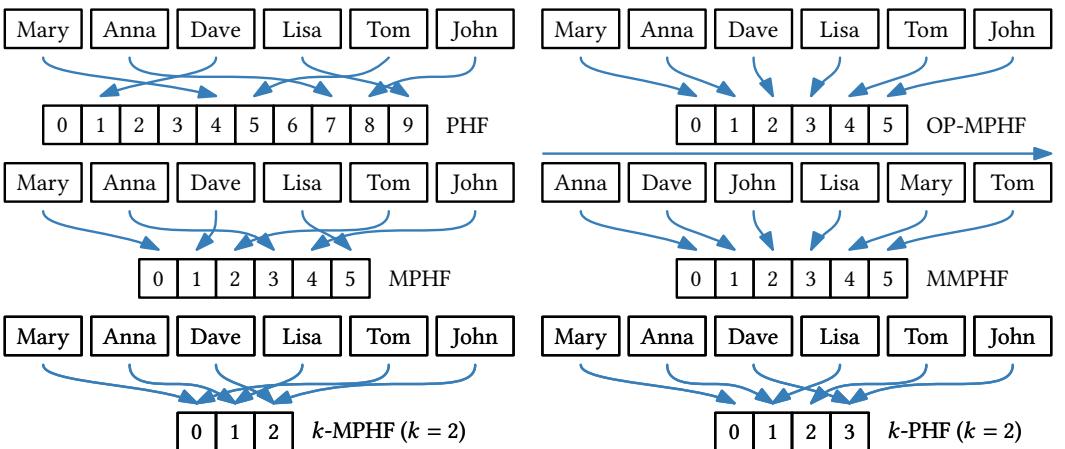


Figure 10. Illustrations of different variants of perfect hashing.

structure and not a space-efficient perfect hash function. In this survey, however, we only consider the static case.

## B The Birth of Perfect Hashing

In this section, we give a short overview of the origins of perfect hashing. These approaches already introduce a number of basic principles that are still used in modern constructions, some of which we have explained in Section 3.1. Even if there are faster and more space-efficient approaches today, they illustrate the progress that was made on the topic during the last 50 years of research.

In 1963, for identifying reserved words in an assembler, Grenievski and Turski [87] present a function that can convert tokens to integers without collisions. Given an input text  $T$ , their hash function has the form  $f_i(T) = a_i f_{i-1}(T) + T[i] + c_i$ . The idea is to determine constants  $a_i$  and  $c_i$  experimentally such that  $f_n$  avoids collisions on the input set. However, the authors do not describe a way to generalize the idea to arbitrary input sets nor do they call the resulting data structure a *perfect hash function*. In the first edition of *The Art of Computer Programming* [104] released in 1973, Knuth describes finding a hash function without collisions as an “amusing puzzle”. Knuth states that the puzzle can be solved manually in about one day of work if the number of input keys is small enough and gives an example with  $n = 31$  keys. He uses this as an introduction to a chapter about hash tables and not as an independent field of research. The second edition [104] of his book released in 1998 then already describes the first practical perfect hash function constructions.

It was Sprugnoli [159] who first used the terms *perfect hashing* and *minimal perfect hashing* in 1977. He also describes the first algorithm to systematically construct perfect hash functions. The idea is to find a linear transformation of the input keys, i.e.,  $x \mapsto \lfloor (x + c_1)/c_2 \rfloor$ , where  $c_1$  and  $c_2$  are constants determined by the algorithm. This approach only works well if the keys are uniformly distributed. To deal with this problem, Sprugnoli then proposes to scramble the input using a modulo operation. While this does not work for all input sets, it gives a fundamental basis that is still used in many modern perfect hash functions. Sprugnoli also introduces the ideas of initial hashing, as well as *partitioning* (see Section 3.1). He even already discusses the space usage of his perfect hash functions in terms of machine code that needs to be written for representing the function. Today, perfect hashing is mainly measured as the amount of space needed in a corresponding data structure, but measuring machine code was common for a long time [155].

Cichelli [44] describes a first practical algorithm for determining *minimal* perfect hash functions based on brute-force searching for a simple assignment of letters to numbers. Similar letter-based approaches are presented later, with the main innovations being to look at letters in different positions in the input string which are less likely to be correlated. Jaeschke [101] gives an algorithm that can already handle input sets of up to 1000 keys.

## C Selected Configurations

While the Pareto fronts give a good overall picture, they do not provide a guide on how to select the configuration parameters. In Table 6, we take the two most promising configurations of each competitor from the Pareto fronts. We give their construction time, query time, and space consumption. The table therefore suggests configurations that we recommend for use in an application. The number of cache misses depends on multiple factors, like the cache line size, memory bandwidth, and whether the fetches can be predicted. Therefore, the number of cache misses during a query does not necessarily correspond directly to the query performance. We still include the average number of L1 data cache misses in the table, given as the measured number minus one (due to the use of string input keys). Remember from Section 9 that the sizes of the L1 and L2 data caches are 48 KiB and 512 KiB per core, and the L3 cache has a size of 16 MiB. We use the configurations

Table 6. Selected configurations for all competitors. Measurements are all single-threaded and for  $n = 100$  million keys. We use the configurations marked with \* in Sections 9.3 and 9.4. The *cache* column gives the average number of L1 cache misses per query, excluding loading the queried keys.

	Approach	Configuration	Space bits/key	Construction ns/key	Query	
					ns	cache
Bucket Placement	FCH	$c=7.0^*$	7.000	1 100	97	1.3
		$c=3.0$	3.000	46 240	91	1.3
	CHD	$\lambda=4^*$	2.166	1 522	356	11.4
		$\lambda=6$	2.007	18 336	346	11.5
	PTHash	$\lambda=4.0, \alpha=0.99, \text{C-C}$	3.342	516	76	1.3
		$\lambda=6.0, \alpha=0.95, \text{EF}^*$	2.150	1 125	119	4.6
	PTHash-HEM	$\lambda=4.0, \alpha=0.99, \text{C-C}$	3.167	528	80	1.3
		$\lambda=6.0, \alpha=0.95, \text{EF}^*$	2.150	1 137	126	4.6
	PHOBIC	$\lambda=4.0, \alpha=1.0, \text{IC, C}$	2.966	297	87	2.3
		$\lambda=6.0, \alpha=1.0, \text{IC, Rice}^*$	2.076	603	123	7.4
Fingerprinting	PHast	$S=7, \lambda=3.7, \text{EF}^*$	1.998	750	43	1.1
		$S=11, \lambda=6.3, \text{EF}$	1.848	5 944	44	1.1
	PHast <sup>+</sup>	$\delta=2, S=8, \lambda=4.35, \text{EF}$	2.074	181	38	1.1
		$\delta=1, S=11, \lambda=6.6, \text{EF}^*$	1.853	256	41	1.1
	PtrHash	$\lambda=3.0, \text{linear, vec}^*$	2.990	141	45	1.0
		$\lambda=4.0, \text{cubic, EF}$	2.118	358	47	1.0
	BBHash	$\gamma=5.0^*$	6.870	157	127	4.0
		$\gamma=1.5$	3.293	273	173	7.4
	FMPH	$\gamma=5.0^*$	6.298	69	88	3.3
		$\gamma=1.5$	3.013	103	139	5.7
Multiple Choice	FMPHGO	$\gamma=5.0, s=4, b=16^*$	6.426	490	84	4.1
		$\gamma=1.5, s=4, b=16$	2.435	457	106	6.1
	FiPS	$\gamma=5.0, \text{LS}=256, \text{Off}=32^*$	6.980	72	82	2.4
		$\gamma=1.5, \text{LS}=256, \text{Off}=16$	3.119	103	123	5.5
	BPZ	$c=1.25, b=3^*$	7.500	512	235	4.2
		$c=1.25, b=6$	3.125	514	219	4.8
	SicHash	$\alpha=0.95, p_1=37, p_2=44^*$	2.197	285	145	4.9
		$\alpha=0.97, p_1=45, p_2=31$	2.080	349	140	4.4
	ShockHash-RS	$\ell=40, b=2000^*$	1.551	4 128	320	10.2
		$\ell=55, b=2000$	1.526	109 921	314	9.9
Rec. Splitting	Bip. ShockHash-RS	$\ell=64, b=2000^*$	1.525	14 214	352	10.0
		$\ell=128, b=2000$	1.489	371 678	296	9.6
	Bip. ShockHash-Flat	$\ell=64^*$	1.618	2 202	180	3.9
		$\ell=100$	1.547	19 716	174	4.1
	MorphisHash	$\ell=40, b=2000, \beta=\ell-4$	1.522	11 259	307	9.8
		$\ell=64, b=2000, \beta=\ell-4^*$	1.489	19 255	306	8.8
	MorphisHash-Flat	$\ell=60, eb=3, \beta=\ell-2$	1.611	3 228	167	3.5
		$\ell=84, eb=3, \beta=\ell-2^*$	1.547	10 586	156	3.5
	RecSplit	$\ell=8, b=100$	1.793	1 629	212	11.5
		$\ell=14, b=2000^*$	1.584	290 374	252	11.7
SIMDRecSplit	SIMDRecSplit	$\ell=8, b=100$	1.809	233	242	9.7
		$\ell=16, b=2000^*$	1.560	277 118	270	10.5
	CONSENSUS	$k=512, \varepsilon=0.1^*$	1.579	575	358	11.4
		$k=32768, \varepsilon=0.0005$	1.444	71 301	445	16.0

marked with the \* symbol for our evaluations in Sections 9.3 and 9.4. In Appendix D, we give the table for just 10 000 input keys to illustrate the lower order terms.

We start with approaches based on multiple-choice hashing. SicHash is faster than BPZ in construction and queries, while also achieving better space consumption. ShockHash-RS is then able to achieve 1.52 bits per key, reducing the gap to the space lower bound of  $\approx 1.443$  bits per key by about 30%. Finally, bipartite ShockHash-RS reduces the space consumption to just 1.489 bits per key, which is within 3.3% of the lower bound with practically feasible construction time. Using a single CPU thread, bipartite ShockHash-RS achieves a space consumption better than what was previously only achieved using thousands of threads on a GPU [21]. Using the same space consumption, MorphisHash improves the ShockHash construction time by 50%–100%. The flat versions of ShockHash and MorphisHash cause significantly fewer cache misses than their counterpart with splitting trees.

We next look at the approaches based on bucket placement. The partitioned implementation of PTHash, PTHash-HEM, has the same construction time and space consumption when run on a single thread. However, its queries are about 10% slower. Even though PHOBIC uses partitioning as well, with compact coding it loses less query time compared to PTHash. By using Golomb-Rice coding instead of Elias-Fano coding, PHOBIC improves the query times of PTHash. FCH and CHD are classical approaches that cannot directly compete with more recent constructions – FCH is slower to construct and CHD is slower to query. PtrHash offers, by far, the fastest queries in the table. At the same time, it has very fast construction and decent space consumption. This makes PtrHash a good choice optimizing the overall trade-off. The approaches based on bucket placement achieve the smallest number of cache misses, with FCH, PHast, PtrHash, and the fixed-length coding of PTHash being very close to a single cache miss per query. PtrHash achieves this even for its Elias-Fano coded seeds through its cache-efficient variant.

We now give a selection of configurations using tree-based search. Comparing different approaches where each is given about half an hour of construction time, RecSplit is able to produce a perfect hash function with 1.58 bits per key. RecSplit starts a chain of approaches that further reduce the space consumption. SIMDRecSplit improves the space consumption to 1.56 bits per key. Then ShockHash goes down to 1.489 and finally CONSENSUS achieves 1.444 bits per key – within 0.1% of the lower bound. The most space-efficient variant of CONSENSUS has the largest number of cache misses due to it storing each layer of the tree in an independent data structure.

For perfect hashing through fingerprinting, we give  $\gamma = 1.5$  and  $\gamma = 5.0$  for all competitors. FMPH achieves the best space consumption among those. FiPS, with its interleaved rank data structure, needs about 0.1 bits per key more space for the smaller configuration. BBHash needs an additional 0.17 bits per key. Looking at the construction time and query time, the same configuration with FMPH is consistently faster to construct and query. This is partially due to its updated implementation compared to the original paper. Through a small number of retries, FMPHGO achieves much lower space consumption but also much slower construction. Especially for large values of  $\gamma$ , the interleaved select data structure in FiPS leads to a significant reduction in cache misses. However, note that FMPHGO achieves a similar query time even though it has more cache misses.

## D Experiments with Smaller Input Sets

Table 7 gives the overview table from Table 6 with a smaller input set of just 10 000 keys. This can help to get a feeling for the lower order terms involved in the space consumption and construction time. Additionally, it gives details on the query performance when the entire data structure fits into the cache. This is also illustrated by all approaches causing only 0–0.2 L1 cache misses on average, except for PHOBIC, which has higher constant space overhead. For most competitors, the space

Table 7. Selected configurations for all competitors. Measurements are all single-threaded and for 10 000 input keys to illustrate lower order terms. The *cache* column gives the average number of L1 cache misses per query, excluding loading the queried keys. Refer to Table 6 for the main comparison with a larger input set.

	Approach	Configuration	Space bits/key	Construction ns/key	Query	
					ns	cache
Bucket Placement	FCH	$c=7.0$	7.072	233	49	0.2
		$c=3.0$	3.072	45 000	49	0.2
	CHD	$\lambda=4$	2.240	548	180	0.2
		$\lambda=6$	2.096	5 900	187	0.2
	PTHash	$\lambda=4.0, \alpha=0.99, C-C$	2.989	368	42	0.2
		$\lambda=6.0, \alpha=0.95, EF$	2.397	743	60	0.2
	PTHash-HEM	$\lambda=4.0, \alpha=0.99, C-C$	3.277	393	42	0.2
		$\lambda=6.0, \alpha=0.95, EF$	2.448	831	63	0.2
	PHOBIC	$\lambda=6.0, \alpha=1.0, IC, Rice$	62.093	326	62	4.3
		$\lambda=4.0, \alpha=1.0, IC, C$	39.654	250	48	1.3
Fingerprinting	PHast	$S=7, \lambda=3.7, EF$	2.582	60 700	17	0.0
		$S=11, \lambda=6.3, EF$	2.371	5 900	17	0.0
	PHast <sup>+</sup>	$\delta=1, S=11, \lambda=6.6, EF$	2.989	44 500	23	0.0
		$\delta=2, S=8, \lambda=4.35, EF$	2.811	44 300	19	0.0
	PtrHash	$\lambda=3.0, \text{linear, vec}$	3.159	147	17	0.0
		$\lambda=4.0, \text{cubic, EF}$	2.329	44 300	20	0.0
	BBHash	$\gamma=5.0^*$	7.475	2 100	63	0.2
		$\gamma=1.5$	3.878	2 800	77	0.2
	FMPH	$\gamma=5.0^*$	6.381	60 000	46	0.0
		$\gamma=1.5$	3.174	60	63	0.0
Multiple Choice	FMPHGO	$\gamma=5.0, s=4, b=16^*$	6.528	44 100	42	0.0
		$\gamma=1.5, s=4, b=16$	2.560	44 100	52	0.0
	FiPS	$\gamma=5.0, LS=256, Off=32^*$	7.194	68	35	0.2
		$\gamma=1.5, LS=256, Off=16$	3.360	96	56	0.2
	BPZ	$c=1.25, b=3^*$	7.519	199	92	0.2
		$c=1.25, b=6$	3.145	200	100	0.2
	SicHash	$\alpha=0.95, p_1=37, p_2=44^*$	2.441	652	82	0.2
		$\alpha=0.97, p_1=45, p_2=31$	2.330	21 400	82	0.2
	ShockHash-RS	$\ell=40, b=2000^*$	1.803	4 300	206	0.2
		$\ell=55, b=2000$	1.776	108 900	200	0.2
Rec. Splitting	Bip. ShockHash-RS	$\ell=64, b=2000^*$	1.770	15 300	231	0.2
		$\ell=128, b=2000$	1.738	405 800	180	0.2
	Bip. ShockHash-Flat	$\ell=64^*$	2.367	2 300	105	0.2
		$\ell=100$	2.283	18 500	98	0.2
	MorphisHash	$\ell=40, b=2000, \beta=\ell-4$	1.525	12 100	190	0.2
		$\ell=64, b=2000, \beta=\ell-4^*$	1.499	21 500	182	0.2
	MorphisHash-Flat	$\ell=60, eb=3, \beta=\ell-2$	2.127	5 500	91	0.2
		$\ell=84, eb=3, \beta=\ell-2^*$	2.043	9 600	87	0.2
	RecSplit	$\ell=8, b=100$	1.978	1 500	122	0.2
		$\ell=14, b=2000^*$	1.781	295 600	153	0.2
CONSENSUS	SIMDRecSplit	$\ell=8, b=100$	2.004	264	141	0.2
		$\ell=16, b=2000^*$	1.788	282 200	156	0.2
	CONSENSUS	$k=512, \varepsilon=0.1^*$	2.150	561	123	0.2
		$k=32768, \varepsilon=0.0005$	3.640	240	117	0.2

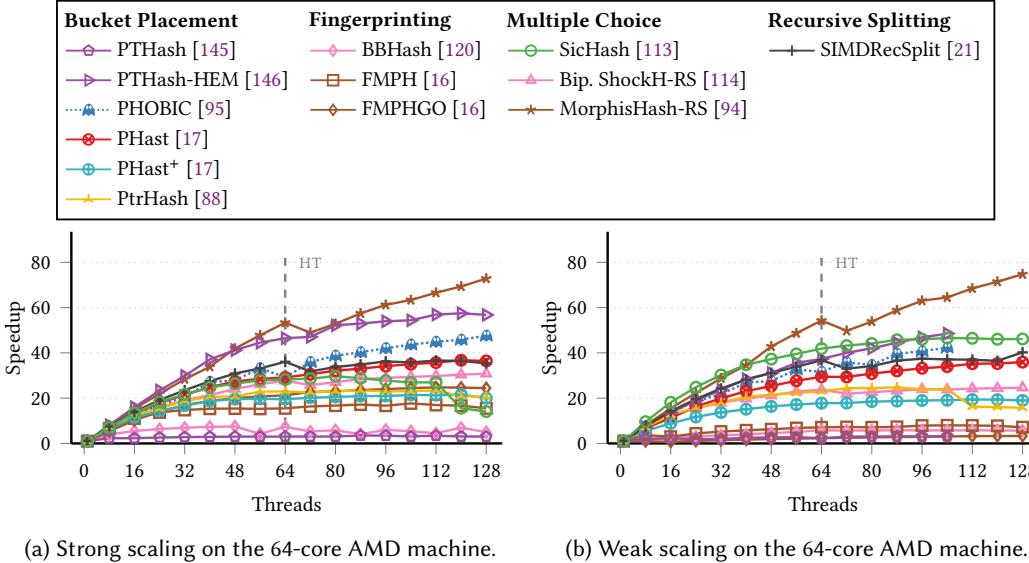


Figure 11. Multi-threaded construction by number of threads, on a 128 core machine. Weak scaling with 10 million keys per thread, strong scaling with 100 million keys. We give self-speedups because each approach has a different focus.

consumption stays almost the same. The only exception is PHOBIC [95], which needs significantly more space. This is caused by its interleaved coding of seeds, which stores a dedicated encoder for each bucket in all partitions. Using just about 4 partitions with the small input set then has considerable space overhead. This can be avoided by selecting a different encoding. Even if the entire data structure fits into the cache, PtrHash [88] keeps by far the fastest queries.

## E Multi-Threaded Construction With 128 Threads

In Figure 11, we give the multi-threaded construction performance of Figure 9 on a larger machine. The machine is equipped with an AMD EPYC 7702P processor with 64 cores (128 hardware threads), pinned to its base clock speed of 2.0 GHz. It runs Rocky Linux 9.5 with Linux 5.14.0 and supports AVX2, having 8 memory channels. For comparison, the 8-core Intel machine we use in most other experiments has 2 memory channels. With weak scaling, the number of keys per thread are constant at 10 million, so the total input size increases gradually up to a total of 1.28 billion using 128 threads.

## F Experiments on ARM Machine

In addition to the x86 machine that we perform our main experiments on, we run the comparison on an ARM machine as well. In this section, we give an overview over the results. We use an Ampere Altra Q80-3 processor with 80 cores, pinned to its base clock speed of 3.00GHz. The sizes of the L1 and L2 data caches are 64 KiB and 1 MiB per core, and the L3 cache has a size of 32 MiB. The machine runs Rocky Linux 9.4 with Linux 5.14.0. We use the GNU C++ compiler version 14.2.0 with optimization flags `-O3 -march=native`. For the competitors written in Rust, we compile in release mode with `target-cpu=native`. Because the machine does not support AVX instructions, we do not include SIMDRecSplit in our evaluation. There are no competitors that specifically support the vector instructions on ARM machines, but the compiler might still automatically vectorize parts of all competitors. Figure 12 gives our main comparison on the ARM machine. Overall, the

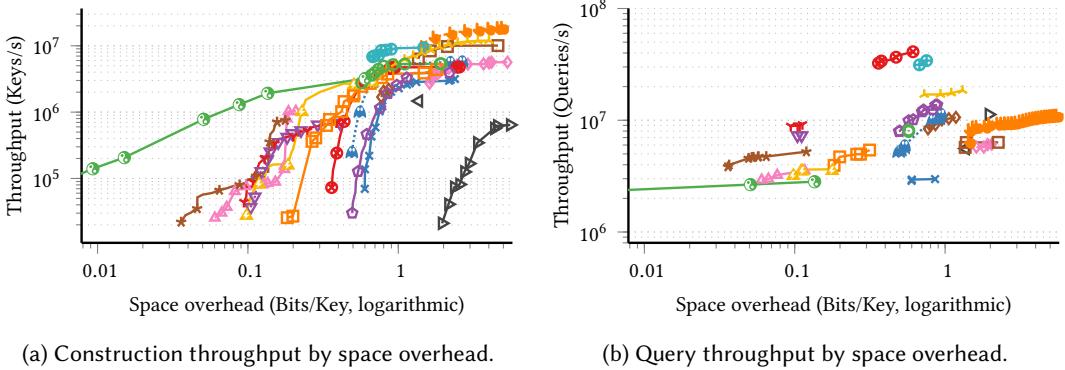
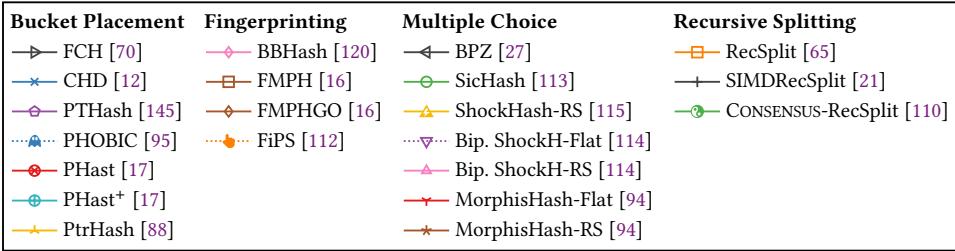


Figure 12. Trade-off of construction time, space consumption, and query time on the ARM machine. Single-threaded measurements with  $n = 100$  million keys. For some approaches, we only show markers for every fourth point to increase readability.

measurements are very similar to the Intel machine. We outline differences below. Note that an earlier version of this paper compared the machines without pinning the clock speed. This led to different results where the ARM machine was generally slower despite its higher base clock speed. This indicates that perfect hashing profits from Intel’s TurboBoost, even though it does not effect the relative performance of the approaches.

*Construction.* On the ARM machine, ShockHash gets slower. The reason for this is that the splittings profit most from SIMD parallelism, and the approaches do not have an ARM implementation. This is also the reason why SIMDRecSplit is missing completely. This makes Bipartite ShockHash-Flat better relatively, given that it does not need splittings. The construction of FiPS gets slightly faster than on the Intel machine.

*Queries.* PHast and MorphisHash have slightly faster queries on the ARM machine, while PtrHash and PHOBIC are slower. Overall, the performance is very similar.