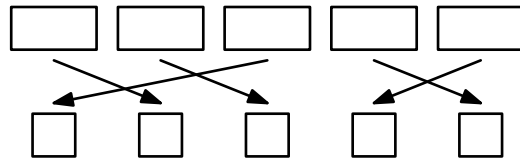


Fast and Space-Efficient Perfect Hashing



Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Hans-Peter Lehmann, M.Sc.

aus Ulm

Gemäß der Promotionsordnung vom 12. Januar 2017

Tag der mündlichen Prüfung: 24. Oktober 2024

1. Referent: Prof. Dr. Peter Sanders
Karlsruher Institut für Technologie
Deutschland
2. Referent: Prof. Rasmus Pagh, PhD.
Universität Kopenhagen
Dänemark

Abstract

From data analytics to machine learning, large amounts of input data become more and more important. The volume of collected data grows continuously. Compact data structures enable efficient access to this data and help with processing. Perfect hashing is a fundamental basis for many compact data structures and is used for example in databases, hash tables, retrieval data structures, and approximate membership data structures. A perfect hash function (PHF) maps a set S of n keys to the first m integers without collisions, and is called *minimal* perfect (MPHF) if $m = n$. It may map keys that are not in S to an arbitrary value. This makes it possible to store the function without representing the set S itself.

In this dissertation, we present three MPHF construction algorithms. With SIMDRecSplit, we focus on space efficiency, scratching the space lower bound of representing MPHFs. We enhance the existing construction RecSplit through parallelism on many levels – bits, vectors, multicores, and the GPU. As a base case, RecSplit uses brute-force search – here we achieve impressive speedups by replacing several retries with simple bit operations or even table lookups. With SicHash, we aim for a good balance between query and construction performance. SicHash is based on sampling a random graph where the nodes are the output hash values. Each key, hashed with different ordinary hash functions, gives a hyperedge connecting candidate output values. The perfect hash function is given by an orientation of the edges such that the indegree of each node is at most one. SicHash works close to the *orientability threshold* – the factor n/m below which such an orientation likely exists. With ShockHash, we combine the ideas of SIMDRecSplit and SicHash. We try to orient a graph far *above* the orientability threshold, which requires retrying with many different graphs. Compared to plain brute-force, this reduces the number of retries massively, leading to more than 2^n times faster construction asymptotically. At the same time, it still maintains the asymptotically optimal space consumption.

Monotone minimal perfect hash functions (MMPHFs) retain the natural order of the input keys. After many years in which tree-based structures were predominant, our LeMonHash moves into a novel direction. It is based on learning regularities in the input data and is efficient to query because it uses a flat structure. Finally, as a relaxation of perfect hashing, we present PaCHash – a static external memory hash table for objects of variable size. PaCHash can fetch an object using a single contiguous access to the external memory. For objects of fixed size, this is similar to what is possible using k -perfect hashing, where at most k collisions are allowed. However, in some way, PaCHash breaks the space lower bounds of k -perfect hashing at the cost of sometimes fetching one external memory block too much.

The approaches presented in this dissertation are a large step forward from the state of the art. Our approaches dominate the majority of the space-time trade-off, as shown by our detailed experimental evaluation, and inspire future research in the area of perfect hashing.

Acknowledgements

A big thank you to Prof. Dr. Peter Sanders for having me in his research group. I am impressed how Peter can supervise such a large group and can still find time to go into so much detail about individual student's topics. Without Peter's guidance, this dissertation would not have been possible.

I am also grateful to my current and former colleagues who created a workspace that I was happy to go to every day. Even outside work, it was great traveling together, giving workshops, enjoying board game evenings, and eating barbecues. Thank you, Daniel Funke, Lars Gottesbüren, Stefan Hermann, Demian Hesse, Tobias Heuer, Lukas Hübner, Lorenz Hübschle-Schneider, Markus Iser, Florian Kurpicz, Sebastian Lamm, Moritz Laupichler, Nikolai Maas, Tobias Maier, Matthias Schimek, Dominik Schreiber, Daniel Seemaier, Tim Niklas Uhl, Stefan Walzer, Marvin Williams, and Sascha Witt.

Thank you to my students over the years: Matthias Becht, Dominik Bez, Stefan Hermann, Sebastian Kirmayer, Tobias Paweletz, Jan Benedikt Schwarz, Benedikt Waibel, Felix Wedler, and Jonatan Ziegler. I would like to thank the coauthors of the papers written during my time in the algorithm engineering group: Dominik Bez, Paolo Ferragina, Stefan Hermann, Lorenz Hübschle-Schneider, Florian Kurpicz, Giulio Ermanno Pibiri, Peter Sanders, Giorgio Vinciguerra, and Stefan Walzer.

I also want to thank Niko Wilhelm, Colin Bretl and Liam Wachter, who were convinced before myself that I would start this journey as a PhD student. Thank you to Stefan Hermann, Florian Kurpicz, Dominik Schreiber, and Stefan Walzer for the valuable feedback about early versions of this dissertation. Thank you to my family for their support.



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).

Table of Contents

1	Introduction	1
1.1	Perfect Hashing	1
1.2	Contributions	3
1.3	Outline	4
1.4	Applications	5
2	Preliminaries	9
2.1	Rank and Select	9
2.2	Golomb-Rice Coding	10
2.3	Elias-Fano Coding	10
2.4	Retrieval Data Structures	12
2.5	Cuckoo Hashing	13
2.6	Perfect Hashing Through Retrieval	15
2.7	Space Lower Bounds	15
2.8	Achieving Minimality	17
3	Contributions	19
3.1	Minimal Perfect Hashing Through Tuned Brute-Force	19
3.2	Small Irregular Cuckoo Tables for Perfect Hashing	21
3.3	Small, Heavily Overloaded Cuckoo Hash Tables for Minimal Perfect Hashing	22
3.4	Practical Comparison of Modern Perfect Hashing	24
3.5	Learned Monotone Minimal Perfect Hashing	26
3.6	Perfect Hashing for Variable Size Objects	27
3.7	Summary	29
4	A Brief History of Perfect Hashing	31
4.1	The Birth of Perfect Hashing	31
4.2	Categorization	33
4.3	Random Hypergraphs	34
4.4	Brute-Force	37
4.5	Fingerprinting	40
4.6	Summary	42
5	Minimal Perfect Hashing Through Tuned Brute-Force	43
5.1	Rotation Fitting	44
5.2	SIMD Parallelization	46
5.3	Multi-Threaded Parallelization	47
5.4	GPUs	47
5.5	GPU Parallelization	48
5.6	Internal Experiments	49
5.7	Summary	52
6	Small Irregular Cuckoo Tables for Perfect Hashing	55
6.1	Overloading	56
6.2	SicHash Perfect Hash Functions	57
6.3	Enhancements	59
6.4	Analysis	60
6.5	Internal Experiments	61

6.6	Summary	62
7	Small, Heavily Overloaded Cuckoo Hash Tables for Minimal Perfect Hashing	63
7.1	Pairing Functions	64
7.2	ShockHash	65
7.3	Bipartite ShockHash	66
7.4	Analysis	69
7.5	Partitioning	86
7.6	Variants and Refinements	88
7.7	Internal Experiments	93
7.8	Summary	95
8	Practical Comparison of Modern Perfect Hashing	97
8.1	Plotting Three-Dimensional Measurements	97
8.2	Experimental Setup	98
8.3	Construction Performance	99
8.4	Query Performance	101
8.5	Scaling in the Input Size	103
8.6	Multi-Threaded Construction	104
8.7	Selected Configurations	106
8.8	Summary	108
9	Learned Monotone Minimal Perfect Hashing	111
9.1	Related Work	112
9.2	LeMonHash	114
9.3	LeMonHash-VL	115
9.4	Variants and Refinements	118
9.5	Analysis	119
9.6	Experiments	121
9.7	Summary	126
10	Perfect Hashing for Variable Size Objects	129
10.1	Related Work	131
10.2	The PaCHash Data Structure	133
10.3	Analysis	135
10.4	Variants and Refinements	139
10.5	Experiments	141
10.6	Summary	144
11	Conclusion	147
	Appendix	149
	Publications and Supervised Theses	151
	Bibliography	153

1 Introduction

Summary: *Space-efficient data structures are a vital ingredient for dealing with quickly growing data volumes. One such data structure is a perfect hash function (PHF). A PHF maps a set S of n keys to the first m integers $[m]$ without collisions. It is a fundamental building block of other space-efficient data structures and has a wide range of applications from hash tables with guaranteed constant access time to approximate membership data structures. Several different variants of this scheme are known, most importantly minimal perfect hashing where $m = n$. In this dissertation, we present new perfect hash functions that significantly outperform existing approaches.*

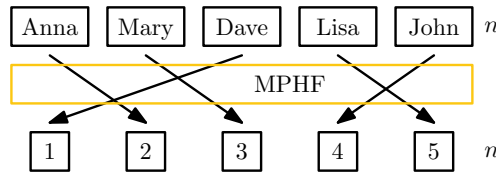
From user metrics to particle accelerators, the amount of data that is collected and stored grows rapidly. The International Data Corporation estimates that the total amount of data stored will be about 175 zettabytes by 2025 [RGR20]. This means that for every human on earth, there will be about 21 terabytes of data. This huge amount of data needs to be stored and accessed efficiently. Space-efficient data structures provide ways to deal with that data and save cost by reducing space consumption and network bandwidth.

External memory like SSDs or hard disks are much slower to access than main memory. Therefore, to process the data, we need to quickly find the location of the relevant content. Index data structures are a crucial ingredient to handle the amounts of data. They not only help in external memory, but are also useful in other layers of the memory hierarchy. The index can be located in the cache, which is smaller but provides faster access.

In this dissertation, we focus on *Perfect Hash Functions* (PHFs), which are fundamental space-efficient data structures that can be used as a building block for a wide range of algorithms. In this chapter, we first explain perfect hashing in Section 1.1. We then briefly describe our contributions in Section 1.2. Afterwards, we give an introduction to the structure of this dissertation in Section 1.3. Finally, we describe applications of perfect hashing in Section 1.4.

1.1 Perfect Hashing

A *Perfect Hash Function* (PHF) maps a set S of n keys to the first m integers $[m]$ without collisions. When simply using an ordinary hash function with a very large output domain m , this property is likely fulfilled. However, in perfect hashing, we usually look for hash functions where the target domain m is not much larger than the input set. In practice, load factors $\alpha = n/m$ of $\alpha = 0.5$ or $\alpha = 0.81$ are common [BBD09]. A smaller target domain significantly increases the probability of collisions between input keys, as illustrated by the birthday paradox. Therefore, a perfect hash function is actually a data structure that stores enough information to avoid collisions for that particular input set S . Perfect hash functions do not store S , which would need at least $\log \binom{U}{n}/n = \Omega(\log \frac{U}{n})$ bits per key, where U is the universe size. This means that a perfect hash function cannot distinguish between a key from S and a key that it was not constructed with. Therefore, for any key not in S , a perfect hash function can give an arbitrary output value.



■ **Figure 1.1** Illustration of a minimal perfect hash function.

In contrast to cryptographic hash functions, perfect hash functions are allowed to be easily reversible or even non-uniform as long as there are no collisions for the given input set. In practice, the focus is mostly on fast evaluation of the function, so implementations often use simple bit shuffling algorithms. To still be resistant against malicious inputs, PHFs usually initially hash the input using a high quality hash function with large output range. This also makes the construction largely independent of the input distribution.

Minimal Perfect Hashing. A *minimal* perfect hash function (MPHF) is a perfect hash function with $\alpha = 1$ and therefore provides a bijective mapping between the keys and $[n]$. Minimality is interesting for space-efficient applications, for example to avoid empty table cells, even though achieving this needs more space to store the perfect hash function itself. Figure 1.1 gives an illustration of a minimal perfect hash function with $n = 5$ input keys. We show in Section 2.7 that there is a space lower bound of 1.44 bits per key needed to represent an MPHF.

k -perfect Hashing. A *k -perfect hash function* is a generalization of a perfect hash function where each output value allows for at most k collisions. A possible application is external memory algorithms where each external memory block can hold k objects. Because memory access operations load an entire block at a time anyway, knowing the block is enough. Searching for the object within that block then does not need further memory accesses [BBD09].

Order-Preserving Perfect Hashing. An *order-preserving* MPHF retains an arbitrary order on the keys [Fox+91]. This comes with a space consumption of at least $\log(n!)/n \in \Omega(\log n)$ bits per key because it needs to differentiate between all $n!$ permutations of the set.¹ The space bound can trivially be reached by using a retrieval data structure taking $\lceil \log n \rceil$ bits per key (plus a small overhead). Therefore, given the development of very good retrieval data structures (see Section 2.4), order-preserving MPHFs are basically a solved problem.

Monotone Perfect Hashing. A *monotone* MPHF (MMPHF) retains the natural order of the keys. This makes it much more efficient than an order-preserving minimal perfect hash function. There are constructions using as few as $\mathcal{O}(\log \log \log U)$ bits per key [Bel+09]. This bound was recently proven to be optimal [AFK23; Kos24]. Another interpretation is that MMPHFs return the rank of each key in the input set. Monotone MPHFs can be useful for range queries in databases.

¹ Throughout this dissertation, $\log x$ stands for $\log_2 x$.

Simple Brute-Force Construction. Many perfect hash functions have been proposed in the literature, each with its own advantages and disadvantages. In Chapter 4, we discuss a wide range of approaches in more detail. As an introduction, we describe a simple construction [Meh84] that can reach the space lower bound. The idea is to try random hash functions (identified by different seeds) using brute-force until one happens to be minimal perfect. Then the data structure simply consists of storing the index of the function in binary coding. The probability for this going well can be analyzed as follows.

► **Lemma 1.1** (from [EGV20]). *A random function $h : S \rightarrow [n]$ on a set S of n keys, is minimal perfect (i.e. is a bijection) with probability $e^{-n} \sqrt{2\pi n} \cdot (1 + o(1))$.*

Proof. Given S , there are n^n possible functions from S to $[n]$. $n!$ of them are permutations of the bijective mapping. Therefore, the probability that a randomly selected function is minimal perfect is $n!/n^n$. The claim is obtained by applying Stirling’s approximation to simplify the factorial. ◀

Given that \log is concave, we can use Jensen’s inequality [Jen06] to get an expected space consumption for storing the seed of $\leq n \cdot \log(e) - o(n) \approx 1.44n$ bits. This simple brute-force construction therefore matches the lower bound given in Section 2.7, up to lower order terms.

The simple construction needs to try an exponential number of different hash functions and is therefore not practical for large n . However, brute-force is still used as a building block in a range of practical constructions. RecSplit [Bez+23; EGV20] partitions the input set into many small subsets using brute-force search for a hash function. Within the partitions, it uses brute-force to find an MPH. A range of approaches based on bucket assignment [BBD09; FCH92; Her+24a; PT21] are also based on brute-force. Refer to Chapter 4 for more details.

1.2 Contributions

In this dissertation, we explore a wide variety of perfect hash function constructions. We develop new approaches that can beat the state of the art in the three main performance parameters: construction throughput, query throughput, and space consumption. In particular, we get close to the space lower bound while still keeping practical construction and query performance. For this, we make use of recent advances in space-efficient data structures such as retrieval, bit vectors, and index data structures. We also harness the processing power of modern hardware architectures by using SIMD instructions, multi-threading, and GPUs. We now give a very brief overview of our contributions. We refer to Chapter 3 for a more detailed introduction to our contributions.

SIMDRecSplit. RecSplit [EGV20] is a space-efficient minimal perfect hash function. With SIMDRecSplit, we parallelize the construction of RecSplit on the levels of bits, vectors, and multicores. We also propose *rotation fitting* – a new technique to accelerate the brute-force search, which can derive additional hash function candidates for the entire set using a few simple bit operations. From a theory point, it enables using small lookup tables to reduce the construction time by a linear factor. In combination, we get speedups of up to 239 on an 8-core CPU, compared to the original single-threaded implementation. We also give a GPU implementation that achieves speedups of up to 5438. Compared with approaches from the literature, SIMDRecSplit is the perfect hash function with the fastest construction for a wide range of configurations.

SicHash. In SicHash, each key has a small number of candidate positions, determined by different hash functions. It derives an assignment using cuckoo hashing and uses a retrieval data structure to store which of the hash functions was finally used to place each key. For some configurations, SicHash has up to 4.3 times faster construction than the next best competitor, and is one of the constructions with the fastest queries. It provides a very good trade-off between space, construction time, and query time.

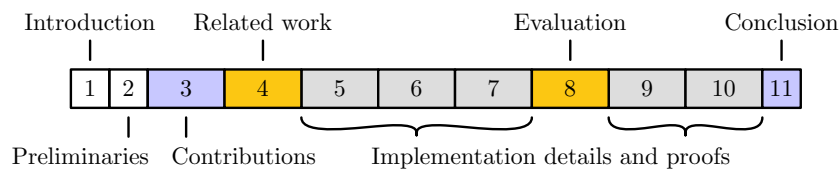
ShockHash. ShockHash can be seen as a combination of SicHash and RecSplit. We use a variant of SicHash with two hash functions for each key and retry construction until we can completely fill the cuckoo hash table. We prove that storing the seed for retries, as well as the retrieval data structure indicating which candidate position to use for each key, is asymptotically space optimal. ShockHash can construct minimal perfect hash functions with just 1.489 bits per key, which is within 3% of the lower bound of 1.442 bits per key. At the same space consumption, ShockHash is up to three orders of magnitude faster than competing approaches.

LeMonHash. *Monotone* minimal perfect hash functions (MMPHF) retain the natural order of the input keys. LeMonHash offers a fresh new perspective on MMPHFs, building upon recent advances in (learning-based) index data structures and in retrieval data structures. We call our proposal *LeMonHash*, because it *learns* the smoothness of the input data to build a space-time efficient *monotone* MPHF. On synthetic random datasets, LeMonHash needs 34% less space than the next competitor, while achieving about 16 times faster queries.

PaCHash. Hash tables or k -perfect hash functions cannot directly be used with objects of variable size. Approaches usually store references in the table cells or leave a lot of space unused. PaCHash is a relaxation of perfect hashing that packs variable-size input objects contiguously in external memory without leaving free space. It can fetch an object using a single contiguous access to the external memory by using a small internal memory index data structure. For objects of fixed size, PaCHash in a way breaks the space lower bounds of k -perfect hashing at the cost of sometimes fetching one external memory block too much. An implementation for fast SSDs needs about 5 bits of internal memory per *block* of external memory and matches or even outperforms competing approaches that only support objects of fixed size.

1.3 Outline

We now give an outline over the content of this dissertation, including a guide on how to read it. This dissertation consists of 11 chapters, of which Figure 1.2 gives a visual overview.



■ **Figure 1.2** Structure of this dissertation.

Depending on what the reader is interested in, it is not necessary to read the entire dissertation from top to bottom. In the following paragraphs, we now give guidelines for different types of readers. More specifically, we discuss readers interested in the contributions of this dissertation, interested in a survey of the state of the art in perfect hashing, or interested in the details of our approaches. This introduction in Chapter 1 and the explanation of preliminary tools and data structures in Chapter 2 are recommended for all readers.

■ **Contributions.** When interested in the new ideas introduced in this dissertation, Chapters 3 and 11 are most relevant. Chapter 3 gives a high-level explanation of the approaches developed here and is enough to understand the main ideas. It also includes a brief evaluation comparing the new ideas to the state of the art. Chapter 11 then summarizes our new approaches and discusses their impact.

■ **Survey.** When interested in a survey discussing the state of the art in perfect hashing, Chapters 4 and 8 can be read independently. In particular, Chapter 4 discusses related work and repeats short summaries of our new approaches. Chapter 8 then gives a detailed evaluation of the state of the art in perfect hashing, focusing on the three most important performance parameters – construction throughput, query throughput, and space consumption.

■ **Proofs and Implementation Details.** The remaining chapters give detailed explanations of our new techniques including proofs and implementation details. They also include evaluations of the effects of different tuning parameters. We explain GPURecSplit / SIMDRecSplit in Chapter 5, SicHash in Chapter 6, and ShockHash in Chapter 7. We then continue with LeMonHash in Chapter 9 and PaCHash in Chapter 10.

1.4 Applications

Perfect hashing was originally developed in the 1970s to quickly handle a small number of reserved keywords in programming languages. Today, perfect hashing can handle millions of input keys and has a wide range of applications. In the following section, we outline some of these applications.

Hash Tables and Retrieval. Hash tables are one of the most fundamental data structures used today. There is a wide range of techniques on how to handle collisions of hash values. Examples include linear probing [Knu98], hashing with chaining [Knu98], and cuckoo hashing [PR04]. When the key set is static, a perfect hash function can be used to directly index the hash table without collision resolution. This guarantees constant access times [FKS84], efficient access with fewer cache faults than standard hash tables [Bot+08], and fewer external memory access operations [KLS23].

Similar to a hash table, a *retrieval data structure* (see Section 2.4) gives a mapping from keys to values. However, a retrieval data structure is allowed to return an arbitrary value for keys that are not stored in the data structure. This makes it possible to store the values without representing the input set. Compared to a hash table, this can significantly reduce the space consumption. Classically, retrieval data structures are static, meaning that changing any value or changing the key set requires a full rebuild of the data structure. Using perfect hashing, it is easy to build a *value-dynamic* retrieval data structure where the values can be changed and the key set stays static. This is possible by indexing a hash table with a perfect hash function and storing only the values in the hash table cells, not the keys.

AMQ Data Structures. An Approximate Membership Query (AMQ) data structure can answer membership queries (“is $x \in S$?”) allowing false-positive replies. The most widely known AMQ data structure is the Bloom filter [Blo70], which needs $n(1.44 \cdot \log(1/\epsilon))$ bits for a false-positive rate of ϵ . Using perfect hashing, $n(1.44 + \lceil \log(1/\epsilon) \rceil)$ bits of space can be achieved. We give more details on lower bounds for perfect hashing in Section 2.7. The idea is to store fingerprints of the input keys in the perfect hash table cells [Ben+18; BM03; Fan+14; GL20]. When a fingerprint collides, the filter then assumes that the key is stored in the data structure and generates a false-positive. Note that the space lower bound for AMQ data structures is $n \log(1/\epsilon)$ bits, which can almost be achieved using a retrieval data structure (see Section 2.4).

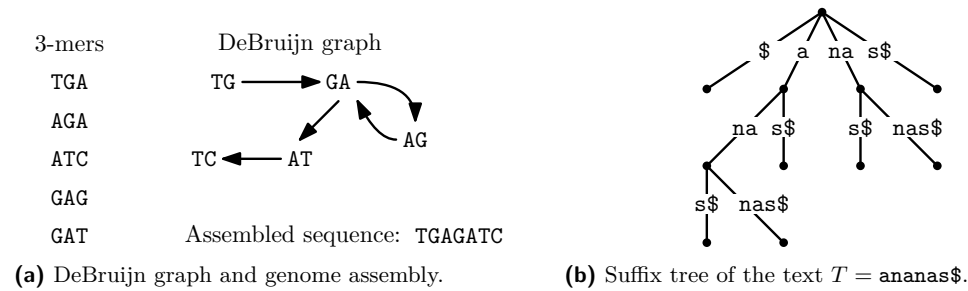
Databases. Perfect Hashing can be used in static (parts of) databases to map stored items from a large universe (for example strings) to smaller identifiers that can be stored with less space [BPZ07a; Mü+14]. The COPR database [RKR24] stores log files and enables efficient search within these logs. Its *immutable sketch* implements a static, approximate multi-set membership query data structure. It generates a perfect hash function on tokens extracted from the log lines. Then the perfect hash function values are used to index a fingerprint array as an approximate membership filter. Additionally, it uses the same hash function value to look up the starting position for indexing a variable-length array.

Association rules [AIS93] can be used to analyze a database of transactions. Each transaction is a set of items, such as products that are bought together in an online shop. Then an association rule has the form $X \Rightarrow Y$, which says that if a transaction contains the items in set X , it (likely) also contains the items in set Y . Perfect hashing can be used to speed up finding these rules in a given database [CL05; HTT02]. The idea is to use perfect hashing to index an array of counters storing the number of occurrences of sets of items.

For synchronizing the content of a distributed database, SNIPS [NM23] divides each file into chunks. Each storage peer then constructs and exchanges a perfect hash function for the list of chunks it stores. This allows for determining missing chunks on peers without communicating the chunks or their IDs. *Monotone* minimal perfect hash functions keep the lexicographical order of the input (see Section 1.1). These are useful for range queries in databases [KLS23; Lim+11], where a range $[a, b]$ in the input universe can directly be mapped to the range $[h(a), h(b)]$ of hash values. This can also help with cache efficiency.

Bioinformatics. For a given string, a k -mer is a substring of length k . In bioinformatics, a k -mer represents k consecutive bases of a genome. When looking at all k -mers that can be formed from a specific string, they share significant overlaps. In genome assembly, these overlaps are used to align pieces of DNA reads together to longer sequences. One such technique uses de Bruijn graphs, which use $(k - 1)$ -mers as their nodes. Edges connect the two $(k - 1)$ -mers that overlap to give a corresponding k -mer [Lia+19]. Figure 1.3a gives an illustration of a de Bruijn graph. An Eulerian trail in the de Bruijn graph then gives the assembled genome sequence.

To navigate the de Bruijn graph, Chapman et al. [Cha+11] need to map a k -mer to the base that follows it. With a hash table, this would mean that the key takes $2k$ bits, while the stored value just takes 2 bits. Therefore, the majority of the size of the hash table consists of storing the key set. Meraculous [Cha+11] uses a perfect hash function instead to query a hash table that does not store the keys. Perfect hashing can aid de Bruijn graphs in more areas [Alm+18; CLM16; CSM13], for example to implement union-find data structures on the k -mers [CLM16]. To efficiently store a set of k -mers supporting membership queries,



■ **Figure 1.3** Illustrations of applications of perfect hashing.

perfect hashing can be used as well [Pib22; Pib23]. Pibiri et al. [PSL23] introduce a perfect hash function specifically designed for k -mers. They use the fact that the input keys overlap to beat the space lower bound that applies to general inputs (see Section 2.7) and to achieve locality of the hash values of overlapping k -mers.

Text Indexing. A suffix of a text T is a substring $T[i..|T|]$ starting from a specific position $1 \leq i \leq |T|$ and continuing to the end of the string. A suffix tree [Wei73] efficiently stores all suffixes of a given text. Every leaf node represents one suffix, while every edge is annotated with a character (sequence) that differentiates between the child nodes. Suffix trees can be used to efficiently search for all occurrences of a pattern in a text by following the edges of the tree and stopping when the pattern ends. Figure 1.3b gives an illustration of a suffix tree of the text $T = \text{ananas}\$$. Perfect hashing can be used to index child nodes of a suffix tree in constant time [BN14]. In general, perfect hashing can be used to store trees, for example in prefix search [Bel+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 [WMB99]. The Burrows-Wheeler Transform (BWT) [Bur94] is a reversible transformation of a text that is easier to compress than the original text. For directly accessing a position in the original text given only the BWT, a rank operation on the occurrences of specific characters is needed. Using monotone minimal perfect hashing, this rank can be computed in constant time and with low space overhead [BN14]. For reporting the number of occurrences of each character in a substring, Belazzougui et al. [BNV13] store the rank of each character occurrence using an MMPHF. From a list of documents, this then enables efficiently finding the k documents with the most occurrences of a pattern (top- k retrieval) [BNV13; Nav14]. Finally, in pattern matching [Bel+20; GNP20; GOR10], MMPHFs are applied mostly to integer sequences representing the occurrences of certain characters in a text.

Machine Learning. N-gram language models are a crucial ingredient of natural language processing, machine learning, and spell checking [PV17b]. Pibiri et al. [PV17b] introduce a compressed tree representation of N-gram language models and use perfect hashing to achieve faster lookups. For real-time speech recognition at Amazon, Strimel et al. [Str+20] 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 instead of storing the IDs explicitly and having to perform a lookup.

Further Applications. Perfect hashing can be used in network applications such as the flow lookup tables in routers [LPB06]. In this application, dynamic perfect hashing becomes relevant. The paper presents a partially dynamic perfect hash function that only needs to be

rebuilt if the key set changes too much. Dynamic perfect hashing is also analyzed in more detail by Dietzfelbinger et al. [Die+94]. LTL (Linear Time Temporal Logic) is a logic used in formal verification of software. For LTL model checking, Edelkamp et al. [ESS08] generate a semi-external graph, where some information about the nodes but not the edges can be stored in internal memory. The approach uses a minimal perfect hash function to store additional information about the nodes in external memory and achieve efficient access. A further application is to use monotone MPHFs for efficient queries in encrypted data [BCO11].

2 Preliminaries

Summary: *Space-efficient data structures are a large field of research. Our perfect hash functions build on existing space-efficient data structures, which we explain in this chapter. Starting with operations on bit vectors, $\text{rank}_1(i)$ determines the number of 1-bits before position i and $\text{select}_1(i)$ finds the position of the i -th 1-bit. Elias-Fano coding builds on these operations to efficiently store a monotone sequence of integers. For a set S , a retrieval data structure stores a static function $S \rightarrow \{0, 1\}^r$ in close to $r|S|$ bits. This small space consumption is possible because it can return arbitrary values for keys not in S . Finally, cuckoo hashing is a technique to solve collisions in hash tables. Each object can be located in one of two candidate cells, which need to be retrieved during queries. Insertion is based on recursively pushing out the object already stored in a cell and re-inserting it with its other hash function.*

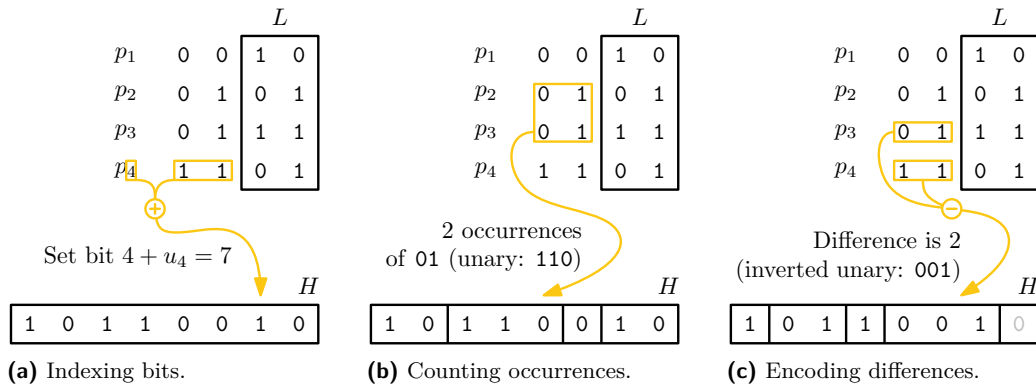
Attribution: None of the data structures described in this chapter are new. However, we give three interpretations of Elias-Fano coded sequences. One of these interpretations is, to the best of our knowledge, not mentioned in the literature yet and explains the connection between Elias-Fano and Golomb-Rice codes.

In this chapter, we give an introduction to several space-efficient data structures. These are building blocks for the perfect hash functions described in this dissertation. Section 2.1 gives an introduction to rank and select operations on bit vectors. Section 2.3 then explains how to store sequences of integers using Elias-Fano coding and gives a detailed comparison of different interpretations of the data structure. Section 2.4 explains retrieval data structures that can store static functions. Section 2.5 then explains cuckoo hashing, which is a technique to resolve collisions in hash tables. We explain in Section 2.6 how retrieval data structures can be combined with ideas from cuckoo hashing to construct perfect hash functions. Finally, we give space lower bounds for perfect hashing in Section 2.7.

2.1 Rank and Select

The most fundamental data structure in a computer is probably the bit vector – at the lowest level, every data structure is a bit vector. After the access operation, rank and select operations are arguably the most fundamental operations on bit vectors. $\text{rank}_1(i)$ determines the number of 1-bits before position i . Note that this directly gives the number of 0-bits by taking $\text{rank}_0(i) = i - \text{rank}_1(i)$. The $\text{select}_1(i)$ operation finds the position of the i -th 1-bit in the bit vector. Figure 2.1a gives an example of these two operations.

The operations can be performed in constant time with sublinear space overhead [Cla96; Jac89] and have very fast and space-efficient implementations [Kur22; Vig08]. A common idea is to divide the bit vector into *superblocks* of size $b_s = \log^2(n)$. An array of size n/b_s stores $\log(n)$ bit values indicating how many bits are set before each superblock. Each superblock is in turn divided into subblocks of size $\log(n)/2$. An additional array of size $n/(\log(n)/2)$ then stores $\log(b_s)$ bit values indicating how many bits are set before each subblock within its superblock. To give ranks for all subblock combinations in constant time, we can use a



■ **Figure 2.2** Interpretations of the same Elias-Fano data structure for the sequence $\langle 2, 5, 7, 13 \rangle$.

Indexing Bits. As a first interpretation, we briefly repeat the one described before, where value u_i is represented as a 1-bit in position $i + u_i$ (see Figure 2.2a). The interpretation is described, for example, by Pibiri and Venturini [PV17a] or Okanohara and Sadakane [OS07]. An advantage of this interpretation is that it shows how insertions can happen in arbitrary order when n , U , and the index of each integer are already known. This can be useful when constructing an Elias-Fano coded sequence in parallel. Additionally, it shows that the space to allocate for the bit vector is known in advance given just n and U .

Counting Occurrences. In Figure 2.2b, we illustrate a different interpretation of the same Elias-Fano coded sequence. This interpretation is described, for example, by Ottaviano and Venturini [OV14] or Pibiri and Venturini [PV17a]. Take the input keys and split them into the upper and lower parts like before. Then count the number of occurrences of each combination of upper bits. Finally, concatenate in unary coding how often each combination occurs. The intuition for both interpretations leading to the same bit vector is that all occurrences of the same upper bits generate consecutive bits in the indexing bits interpretation. Still, in contrast to the interpretation above, the counting occurrences interpretation sounds like insertions have to happen in order, which is not the case. The counting occurrences interpretation is useful when thinking about predecessor queries, which we explain later.

Encoding Differences. A third interpretation is that H stores the unary coded differences of consecutive upper bits. As an intuition why this describes the same bit vector, consider n Elias-Fano coded integers with the indexing bits interpretation. Assume that we do not store the trailing zeroes, so the last bit is set to 1. Then the last bit, representing u_n , has position $u_n + n$. Now let us look at what happens when we append an additional value. This new value has upper bits u_{n+1} , setting a 1-bit in position $u_{n+1} + (n + 1)$. Subtracting the positions, we get the number of newly added bits: $u_{n+1} + (n + 1) - (u_n + n) = u_{n+1} - u_n + 1$, which is $u_{n+1} - u_n$ times a 0-bit and one 1-bit. This matches a unary coding of the difference between the neighboring upper bits, just with the roles of 0 and 1 inverted. We illustrate this in Figure 2.2c. The encoded differences interpretation is useful to see that storing a prefix sum of values with Elias-Fano coding is almost the same as storing the original values with Golomb-Rice codes. The only difference is that the roles of 0 and 1 in H are inverted and that Golomb-Rice codes need one less subtraction operation on the lower bits. For storing perfect hash function seeds, Pibiri and Trani [PT21] encode a prefix sum with Elias-Fano and Hermann et al. [Her+24a] achieve performance improvements by using Golomb-Rice instead.

Operations. We now look at the two most important operations on Elias-Fano coded sequences – access and predecessor queries. The $access(i)$ operation returns the i -th stored integer. The lower bits can simply be accessed from array L at position i . For the upper bits, the interpretation of indexing bits is most useful. Because the corresponding 1-bit of integer i is stored at position $i + u_i$, we can find u_i by performing a $select_1(i)$ operation and then subtracting i . Using a constant time select data structure (see Section 2.1), the access operation takes constant time.

A predecessor query for an integer x returns the (position of the) largest integer in the sequence that is $\leq x$. For performing this operation on an Elias-Fano coded sequence, the interpretation of counting occurrences is most helpful. A $select_0$ operation in H locates the start of a cluster of integers that share the same upper bits as x . Another $select_0$ operation locates the end of the cluster. For the integers inside the cluster we have to check the lower bits in L . We can do that using binary search in worst-case time $\mathcal{O}(\log n)$.

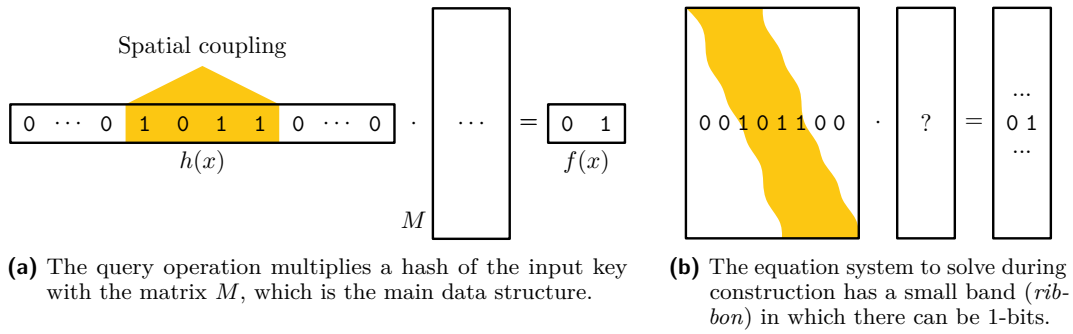
2.4 Retrieval Data Structures

A *retrieval data structure* or *static function data structure* on a set S of n keys stores a function $f : S \rightarrow \{0, 1\}^r$ that returns a specific r -bit value for each key. At first, this might sound similar to a hash table. In fact, a hash table supports this operation as well. However, a retrieval data structure cannot answer membership queries (“is $x \in S$?”), which makes it possible to represent f without representing S . In particular, applying the function on a key not in S can return an arbitrary value. The space lower bound for arbitrary inputs is rn bits. In contrast, a hash table needs at least $n \log(n)$ bits to differentiate the keys.

MWHC [Maj+96] is one of the first retrieval data structures and is based on assigning an integer value to each node in a random hypergraph. It needs $1.23rn$ bits of space and can be evaluated in constant time by hashing a key to different nodes and summing up the integers stored for them. A variation is 2-step MWHC [Bel+11], which can have a smaller overhead than MWHC for some inputs by using two MWHC functions of different widths. A fallback value then indicates that the second data structure with larger width should be queried. In the following, we describe a more recent approach, BuRR [Dil+22], in more detail.

Bumped Ribbon Retrieval. The more recently proposed *Bumped Ribbon Retrieval* (BuRR) data structure [Dil+22] is the one we use in this dissertation. BuRR basically consists of a matrix. The query algorithm multiplies a hash of the key with that matrix to get the output value (see Figure 2.3a). The matrix can be constructed by solving a system of linear equations assigning all input keys to their desired output value. BuRR uses hash functions with *spatial coupling* [Wal21] where all 1-bits are in a local range. This makes the equation system almost a diagonal matrix and therefore very efficient to solve. Additionally, it is helpful for the cache locality of the queries. In the case of $r = 1$, queries are very simple, basically calculating the AND of the key’s hash and a section of a bit vector, and reporting the parity of the result. Figure 2.3b illustrates spatial coupling in the equation system.

When some rows of the equation system would prevent successful solving, BuRR *bumps* out these rows (and the corresponding keys). It handles these keys recursively by adding an additional layer of the same data structure. In fact, BuRR achieves space improvements by intentionally *overloading* the equation system with more equations than can be solved, and relying on bumping. In practice, BuRR achieves space overheads well below 1% over the space lower bound of rn bits. At the same time, it is faster than widely used data structures with much larger overhead [Dil+22].

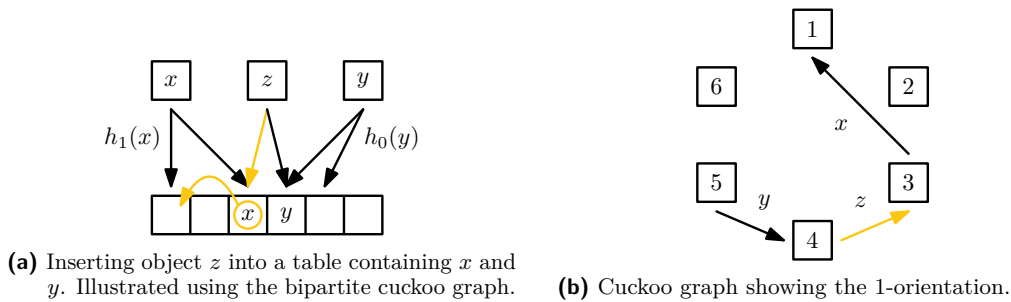


■ **Figure 2.3** Illustrations of Bumped Ribbon Retrieval (BuRR) [Dil+22] with $r = 2$ bits. Simplified here to ignore bumping.

2.5 Cuckoo Hashing

Cuckoo hashing [PR04] is a well known approach to handle collisions in hash tables. In a cuckoo hash table, each object can be placed in one of two candidate cells, determined by two hash functions. Queries load the two candidate cells and compare the searched key with the keys of both objects. Insertion applies one of the hash functions and places the new object in the corresponding cell. If the cell is already occupied, the object previously placed in that cell is pushed out and is recursively inserted using its other hash function. Figure 2.4a illustrates this approach. It also illustrates that cuckoo hashing can be interpreted as a bipartite *cuckoo graph* where one set of nodes represents the objects and the other set represents the table cells. Edges connect objects to their candidate cells.

Variants. Just like the load factor of a perfect hash function, the load factor of a cuckoo hash table is $\alpha = n/m$, where n is the number of objects inserted and m is the number of table cells. Higher load factors can be achieved by making the cells larger, so that they hold more than one object [DW07]. For our application to perfect hashing, we only consider cells of size 1. Instead of locating each object in one of two cells, the idea can be generalized to d cells [Fot+05] by using d hash functions. For an external memory hash table, I/O operations can be reduced by ensuring that most candidate cells are selected on the same memory page [DMR11]. In *irregular* cuckoo hash tables, different objects can have a different number of choices [Die+10]. For example, some percentage of the objects get d_1 choices, some d_2 choices, and some d_3 choices. Averaging over the numbers of choices, the method enables d -ary cuckoo hashing with non-integer d and higher load factors than a simple interpolation



■ **Figure 2.4** Illustration of different graph interpretations of the same cuckoo hash table.

between two ordinary cuckoo hash tables [Die+10]. A similar idea can also be found in coding theory, where each message bit is covered by an irregular number of check bits [Lub+01]. Another related result is the weighted Bloom filter [BGJ06], where objects get a different number of hash functions (and therefore false positive probability) based on their query frequency and membership likelihood.

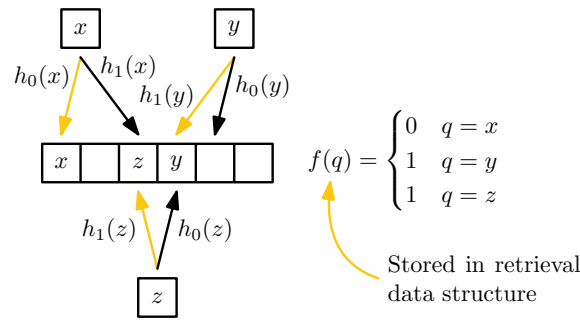
Load Thresholds and Space Usage. The *load threshold* is the load factor $\alpha = n/m$ such that the probability of successful construction tends to 1 for smaller load factors and to 0 for larger load factors when taking $m \rightarrow \infty$ [FKP16; FP12; Lel12]. Classic cuckoo hashing with $d = 2$ hash functions has a load threshold of 0.5. Therefore, for large n , constructing a cuckoo hash table where 51% of the cells are filled will likely fail. Using $d = 4$ hash functions already increases the load threshold to 0.976 [FP12; Wal21].

Dietzfelbinger et al. [Die+10] give load thresholds for irregular cuckoo hashing, depending on the distribution of how many hash function choices each object has. For any desired average number of hash functions $d' \in \mathbb{R}$, the best load threshold is achieved by assigning each object either $\lfloor d' \rfloor$ or $\lceil d' \rceil$ hash functions [Die+10]. As we will see in Section 6.1, this is not the case in the context of perfect hashing because we are looking at the storage space instead of the average value of the hash function indices.

Construction. For binary cuckoo hashing, the construction can simply displace keys to their other candidate cell, as described above. The variant with $d > 2$ or irregular cuckoo hashing make insertions more complex because it is no longer clear which of the alternative cells to displace objects to. Common ways to perform insertion are to find a shortest move sequence by performing *breadth-first-search (BFS)* in a graph defining possible object moves or by performing a *random walk* in that graph. Both approaches need constant expected time when the table is not too highly loaded [FMM09; Fot+05; FPS13; KA19; Kho13; Wal22].

In this dissertation, we are only interested in the static case, where all objects to be stored are known from the start. In that case, it is also possible to construct the whole hash table at once instead of using incremental insertions. Looking at the cuckoo hash table as a bipartite graph again (see Figure 2.4a), a matching of size n gives a collision-free assignment from objects to table cells. This can be calculated using, for example, the Hopcroft-Karp-Karzanov algorithm [HK73] in time $\mathcal{O}(n\sqrt{n})$ or the LSA algorithm [KA19; Kho13] in linear time with high probability.

Cuckoo Graph and Pseudoforests. We now switch back to binary cuckoo hashing with $d = 2$ hash functions. In the following, it will be useful to model cuckoo hashing as a different type of graph. In that graph G , each node represents a table cell and each edge represents one object, connecting its two candidate cells. We illustrate this in Figure 2.4b with the same hash functions as in Figure 2.4a. It is easy to see that the table can be constructed successfully if and only if the edges of G can be directed such that the indegree of each node is ≤ 1 . We call this a 1-orientation. A 1-orientation exists if and only if G is a *pseudoforest*, i.e., every connected component of G is a pseudotree. A pseudotree is either a tree or a cycle with trees branching from it. A way to check whether a graph is a pseudoforest is to check whether each component contains at most as many edges as nodes. This is possible in linear time using depth first search.



■ **Figure 2.5** Perfect hashing through retrieval. Each key has $d = 2$ candidate output values. Yellow arrows indicate a mapping that gives unique values for all keys.

2.6 Perfect Hashing Through Retrieval

To the best of our knowledge, perfect hashing through retrieval with cuckoo hashing is only mentioned briefly before [Dil+22]. In this section, we give a more detailed and intuitive introduction to the idea. A related idea is the construction of PHFs by solving a matching as described by, e.g., Botelho et al. [BPZ13] and Navarro [Nav16, Section 4.5.3]. This essentially solves the same graph theoretic problem but uses a different interpretation.

For perfect hashing through retrieval, every key has a small number d of candidate output values determined by d hash functions. We then find an assignment of each key to one of its candidate positions such that each position is taken at most once. We can remember the assignment using $\lceil \log(d) \rceil$ bits per key storing the hash function index. This can be stored in a retrieval data structure using close to $\lceil \log(d) \rceil \cdot n$ bits. A query for a key x then retrieves the hash function index $i(x)$ and calculates $h_{i(x)}(x)$ to obtain a perfect hash function. Figure 2.5 illustrates the idea.

Construction through Cuckoo Hashing. One way to find such an assignment is to construct a cuckoo hash table. Take the input keys of the perfect hash function and use them as objects to store in the d -ary cuckoo hash table. The final table then implicitly describes an injective mapping from keys to table cells, because each cell only stores one key. For perfect hashing, we are not interested in storing the hash table or the keys themselves. We are only interested in the hash function index that was finally used to place each key. We store this using a retrieval data structure as described before. In our construction, the load factor of the perfect hash function equals the load factor of the cuckoo hash table, and the storage space is determined by the number of hash functions d . By applying results from cuckoo hashing, we therefore immediately get details on the load factor of the perfect hash function. A PHF from binary cuckoo hashing using two choices needs about $\log(2) = 1$ bit per key and can achieve a load factor of $\alpha = 0.5$. A PHF with a load factor of 0.976 can be implemented using 4 choices, leading to 2 bits per key. This result can be converted to an MPHf by investing about 0.14 bits per key in addition, as we will see in Section 2.8.

2.7 Space Lower Bounds

Somewhat surprisingly, the space needed to avoid collisions in a perfect hash function is constant per input key. Let us first look at space lower bounds of minimal perfect hashing and then continue with perfect hashing and k -perfect hashing.



■ **Figure 2.6** Preimages of a minimal perfect hash function and a possible input set that the function is minimal perfect on.

Minimal Perfect Hashing. For minimal perfect hashing, the space lower bound is $n \cdot \log(e) + \mathcal{O}(\log(n)) \approx 1.44n$ bits per key [Mai83; Meh82]. This bound is quite simple to explain: Take a function f 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 must 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 $f^{-1}(1), f^{-1}(2), \dots, f^{-1}(n)$. We illustrate this in Figure 2.6. 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 , where U is the size of the key universe. 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 functions to be able to cover every possible input set. A minimal perfect hash function then needs to store which of these functions has to be used on the respective input set. Therefore, the number of bits needed to represent a minimal perfect hash function can be bounded as follows. The equation assumes that U is large compared to n and ignores logarithmic terms.

$$\log \left(\frac{\binom{U}{n}}{\left(\frac{U}{n}\right)^n} \right) \approx \log \left(\frac{\left(\frac{Ue}{n}\right)^n}{\left(\frac{U}{n}\right)^n} \right) = \log(e^n) = n \log e \approx 1.44n$$

Perfect Hashing. Intuitively, giving up on the minimality reduces the probability for collisions between keys. This makes it possible to represent PHFs with less space, depending on the load factor $\alpha = n/m$. Using a similar combinatorial argument like for minimal perfect hashing, it is possible to determine the space lower bound to represent a PHF with range m .

$$\begin{aligned} & \log \left(\frac{\binom{U}{n}}{\left(\frac{U}{m}\right)^n \cdot \binom{m}{n}} \right) \stackrel{[\text{BBD09}]}{\approx} (m-n) \log \left(1 - \frac{n}{m}\right) - \log(n) - (U-n) \log \left(1 - \frac{n}{U}\right) \\ &= n \left(\frac{1}{\alpha} - 1\right) \log(1 - \alpha) - \log(n) - \log \left(\left(1 - \frac{n}{U}\right)^U \right) + n \log \left(1 - \frac{n}{U}\right) \\ & \stackrel{U \rightarrow \infty}{\approx} n \left(\frac{1}{\alpha} - 1\right) \log(1 - \alpha) - \log(n) - \log(e^{-n}) \\ &= n \left(\log(e) + \left(\frac{1}{\alpha} - 1\right) \log(1 - \alpha) \right) - \log(n) \end{aligned}$$

Minimal k -perfect Hashing. Extending the proof ideas above once more [BBD09], we get a space lower bound for minimal k -perfect hashing as follows. Using Stirling’s approximation, we derive a new space lower bound that is easier to interpret.

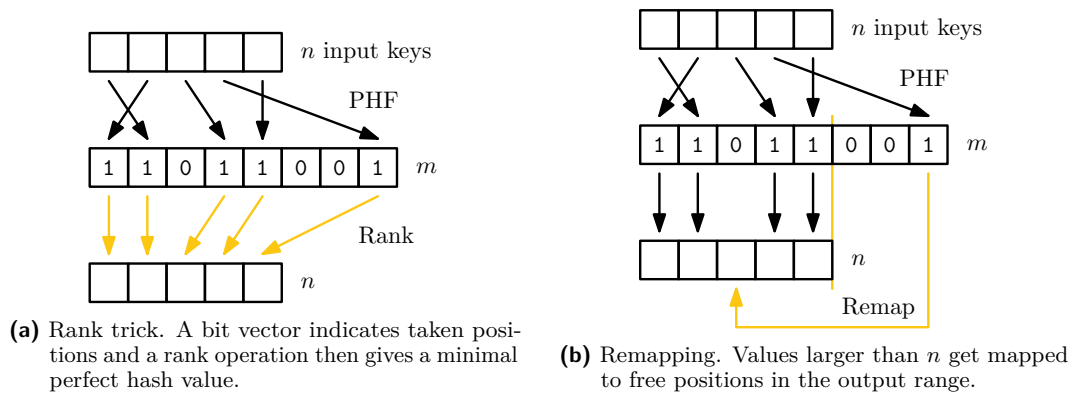
$$\begin{aligned}
\log \left(\frac{\binom{U}{n}}{\binom{U/(n/k)}{n/k}} \right) &\stackrel{[\text{BBD09}]}{\approx} n \cdot \left(\log(e) + \log \left(\frac{k!}{k^k} \right) / k \right) \\
&\approx n \cdot \left(\log(e) + \log \left(\frac{\sqrt{2\pi k} (k/e)^k}{k^k} \right) / k \right) = n \cdot \left(\log(e) + \log \left(\sqrt{2\pi k} (1/e^k) \right) / k \right) \\
&= n \cdot \left(\log(e) + \frac{\log(\sqrt{2\pi k})}{k} - \frac{\log(e^k)}{k} \right) = n \cdot \left(\log(e) + \frac{\log((2\pi k)^{1/2})}{k} - \log(e) \right) \\
&= \frac{n}{k} \cdot \frac{1}{2} \log(2\pi k)
\end{aligned}$$

Mairson [Mai83] gives a slightly tighter space lower bound without a proof. He shows that $\frac{n}{k} \cdot \frac{1}{2} \log(2\pi k) - \frac{1}{2} \log(2\pi n) + \mathcal{O}\left(\frac{n}{k^2} + \frac{n^2}{U} + \frac{1}{n}\right)$ bits are sufficient, where U is the size of the key universe. Mairson [Mai92] then considers a variant of non-minimal k -perfect hashing where we increase the output range of the hash function but require either exactly k collisions or no collisions on each output value. In this setting, decreasing the load factor does not improve the space consumption asymptotically, which is different to ordinary perfect hashing.

2.8 Achieving Minimality

Any perfect hash function with output range m can easily be converted to an MPHf with output range n . Even though it can make the construction more efficient [LSW23b; PT21], this introduces some space and query time overhead. An additional disadvantage is that it introduces an additional input parameter. In this dissertation, we follow the practice in the literature (see Chapter 4) and focus on *minimal* perfect hash functions. We now describe two approaches to convert a PHF to an MPHf.

Botelho et al. [BPZ07b] introduce the *rank trick* where a bit vector of size m indicates which output positions are taken. A rank operation (see Section 2.1) then gives the MPHf value. This approach has a space overhead of more than m bits, and also some query time overhead due to the rank operation. We give an illustration in Figure 2.7a.



■ **Figure 2.7** Techniques to convert a PHF to an MPHf.

PTHash [PT21] introduces a new technique for converting PHFs to MPHFs that has less overhead. The idea is to check if the output value of the hash function is greater than n . If it is not, the value can directly be returned. If it is greater, PTHash looks up the value to be returned in a compressed sequence of size $(m - n)$ that stores the *free* positions in $[n]$. Given that the free positions are a monotonic sequence of integers, they can be compressed with Elias-Fano coding (see Section 2.3). Usually, m is very close to n , so the sequence is short and needs to be queried rarely. This technique therefore has significantly lower space and query time overhead than the rank trick. We give an illustration in Figure 2.7b.

3 Contributions

Summary: *In this chapter, we give an overview over the perfect hash functions presented in this dissertation. This includes five main results. SicHash orients random graphs to achieve a good balance between query and construction performance. GPURecSplit parallelizes an existing brute-force construction and reduces the search space for significant speedups. Finally, ShockHash achieves exponential speedups by reducing the search space of the brute-force construction even further. Additionally, we present LeMonHash as the first learning based monotone MPHf, and PaCHash as an efficient static hash table using a relaxation of perfect hashing.*

Attribution: This section is based on and has text overlaps with the introduction sections of the corresponding papers [Bez+23; Fer+23a; KLS23; LSW23b; LSW24b]. However, it is reworked to include more algorithmic details and supporting figures.

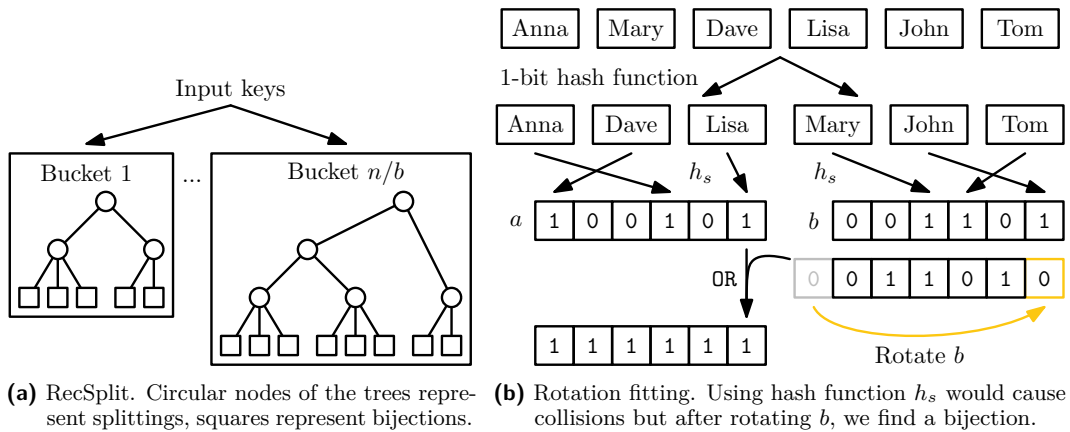
Following the same order as the overall structure of this dissertation (see Section 1.3), we now give an introduction to the contributions of this dissertation. We begin with minimal perfect hash functions in Sections 3.1–3.3 and then demonstrate the performance improvements through a brief evaluation in Section 3.4, similar to the full evaluation in Chapter 8. We then continue with our new monotone minimal perfect hash function in Section 3.5 and our perfect hash function for variable size objects in Section 3.6.

This chapter explains the main ideas behind each approach in enough detail to give a basic understanding of the algorithms. However, for the full explanation of the approaches including enhancements, implementation tricks, proofs, as well as the evaluation of tuning parameters, we refer to the corresponding chapters.

3.1 Minimal Perfect Hashing Through Tuned Brute-Force

Brute-force construction of perfect hash functions can reach the space lower bound (see Section 1.1). However, it needs to try an exponential number of seeds, which makes it impractical for larger input sets. RecSplit [EGV20] is an efficient perfect hash function that uses brute-force in a novel way. It starts with hashing the keys to buckets of expected constant size to ensure linear construction time. The main idea of RecSplit is now to recursively split the set of keys in a tree structure until small sets of constant size are left. In each inner node of the tree, it uses brute-force to search for a hash function that splits up the keys to subsets of very specific size. In the leaves, RecSplit then performs the simple brute-force construction explained in Section 1.1 to find a perfect hash function, often called *bijection*. Figure 3.1a gives an illustration. In Chapter 4, we describe RecSplit in more detail.

When increasing the leaf size, RecSplit gets closer to the space lower bound. However, the construction quickly becomes slow because of the large number of brute-force tries. We introduce two new improvements in the RecSplit framework. First, we describe *rotation fitting*, which is a new way to reduce the number of hash function evaluations when searching for bijections with brute-force. We then continue with a parallelization of the approach using bit-parallelism, SIMD instructions, multi-threading, and GPUs. We call our resulting algorithms SIMDRecSplit and GPURecSplit.



■ **Figure 3.1** Illustration of the overall RecSplit data structure and our enhancement to the leaf nodes, *rotation fitting*.

Rotation fitting. *Rotation fitting* makes it possible to test additional hash function seeds significantly faster than additional brute-force trials. It hashes the keys to two sets A and B using a static 1-bit hash function. The sets do not necessarily have the same size. During a brute-force iteration, it then evaluates the hash function on both sets, calculating two bit vectors a and b that indicate which positions are hit by some key. If there are any collisions within a set (e.g., $\text{popcount}(a) \neq |A|$), the construction cannot succeed, so the search retries with another hash function seed. Otherwise, the logical OR of the bit vectors a OR b could be used to check the entire set. However, we realize that *rotating* (cyclically shifting) the bits of b gives a new chance for finding a bijection. Essentially, we try to fit one set into the “holes” of the other set by rotating it. Figure 3.1b gives an example of rotation fitting. There, hashing both sets A and B directly does not lead to a perfect hash function. However, adding 1 (modulo 6) to all hash values in B results in a perfect hash function. This addition modulo 6 can happen efficiently in registers for all keys at once through bit shifts of b .

In Chapter 5, we show that the probability of a rotation leading to a bijection is similar to the probability of a new hash function seed leading to a bijection. Therefore, rotation fitting can reduce the number of hash function evaluations by almost a factor of n . Given that evaluating the hash functions is the main bottleneck of the brute-force technique, rotation fitting enables significant speedups. In practice, rotation fitting makes the overall RecSplit construction up to 3 times faster. In Chapter 5, we also discuss a variant that can replace the n rotations by 2 table lookups. However, in practice, simplicity (in the inner loops), and parallelism wins against any attempt at algorithmic sophistication in this case.

CPU Parallelization. In addition to adding rotation fitting, we parallelize RecSplit. For this, we use the vector parallelism available with *Single Instruction Multiple Data* (SIMD) instructions and the thread parallelism available with multicore CPUs. The parallelization is rather straightforward. We use SIMD to test multiple hash function seeds at once. Additionally, different threads construct different buckets independently in parallel. Using SIMD and rotation fitting, we get a speedup of up to 50, and when additionally using multi-threading with 16 threads, we get a speedup of up to 239, compared to the original single-threaded implementation. For a more detailed explanation of our improvements, we refer to Chapter 5. We compare the performance of SIMDRecSplit to other approaches from the literature in Section 3.4 after introducing our other perfect hash functions.

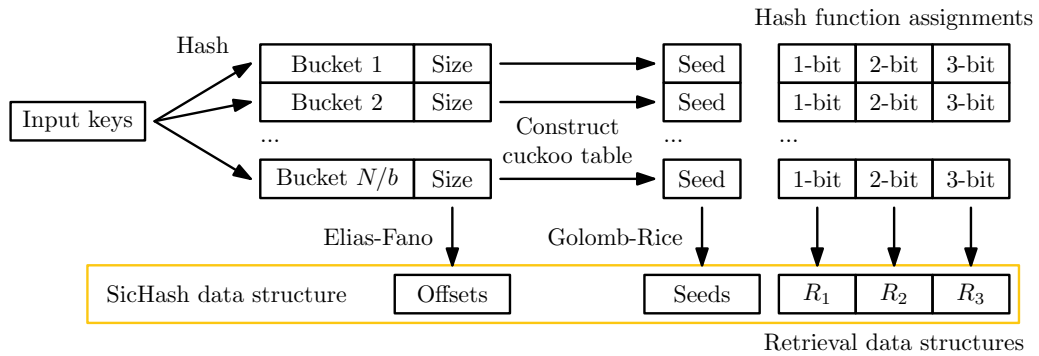
GPU Parallelization. Utilizing GPUs for evaluating hash functions is known from mining of cryptocurrencies with proof-of-work approach (e.g., Bitcoin). Given that hash function construction here is mostly compute bound through using brute-force, the GPU is an ideal hardware. We therefore also give a GPU parallelization of RecSplit. While Lefebvre and Hoppe [LH06] describe the GPU *evaluation* of MPHFs, to the best of our knowledge, our GPURecSplit is the first technique that *constructs* MPHFs on the GPU. Similar to the SIMD implementation, each GPU thread tests one hash function seed independently in parallel. On the CPU, we parallelize over the buckets. On the GPU, we take a different approach, using the fact that the shape of the splitting trees only depends on the number of keys hashed to them. We therefore construct all trees of the same *shape* together in the same set of GPU kernel calls, significantly reducing constant overheads. We then use GPU streams to compute multiple different tree shapes in parallel. Together, we achieve a speedup of up to 5438, compared to the original single-threaded implementation without rotation fitting. Because GPUs are so much faster at constructing MPHFs, they also lead to a better energy efficiency than the CPU, as we show in Chapter 5.

Summary. We harness parallelism at all available levels – bits, vectors, cores, and GPUs. Additionally, we give a new algorithm for improving the bijection search in the base case. Together, we dramatically accelerate the construction of highly space-efficient minimal perfect hash functions using the brute-force RecSplit approach [EGV20].

3.2 Small Irregular Cuckoo Tables for Perfect Hashing

In the previous section, we have looked at a brute-force construction. We now look at the other end of the spectrum of perfect hashing algorithms. SicHash – small irregular cuckoo tables for perfect **hashing** – is based on almost linear time construction of cuckoo hash tables. In SicHash, we combine and refine several known ideas in a novel way leading to excellent trade-offs between space and construction time while using very low query time. SicHash is based on perfect hashing through retrieval (see Section 2.6). There, each key has a small number of candidate hash values, determined by different hash functions. We then use a retrieval data structure to store a collision free mapping from each key to one of its candidate hash functions. We solve the assignment using cuckoo hashing. From cuckoo hashing, we inherit load thresholds where construction likely does not succeed. We therefore first construct a non-minimal perfect hash function and repair it later. The two novel ideas in SicHash are to use *irregular* cuckoo hashing [Die+10] and to use a number of small tables that we *overload* beyond their load threshold. We give details in the following paragraphs.

Irregular Cuckoo Hashing. In *irregular* cuckoo hashing [Die+10] (see Section 2.5), instead of having the same number of candidate positions, different keys can get a different number of candidate positions. We determine the number of candidates to use for each key by evaluating an additional hash function. Irregular cuckoo hashing was previously considered for reducing the search time in hash tables. For that application, it was of little help apart from allowing to interpolate between two integer numbers of hash functions. In contrast, for perfect hashing by retrieval, it is helpful even when the average number of hash functions already is an integer. The key difference is that the space consumption per key is logarithmic in the number of hash functions it can choose from. As an example, a retrieval-based perfect hash function (see Section 2.6) using 5 choices would need at least $\log(5) \approx 2.32$ bits per key. When hashing 50% of the keys with 2 hash functions and the other 50% with 8 hash



■ **Figure 3.2** SicHash construction. We first hash keys to buckets of expected equal size. Within each bucket, we construct a (possibly overloaded) cuckoo hash table. We then store the hash function assignments from all small hash tables together.

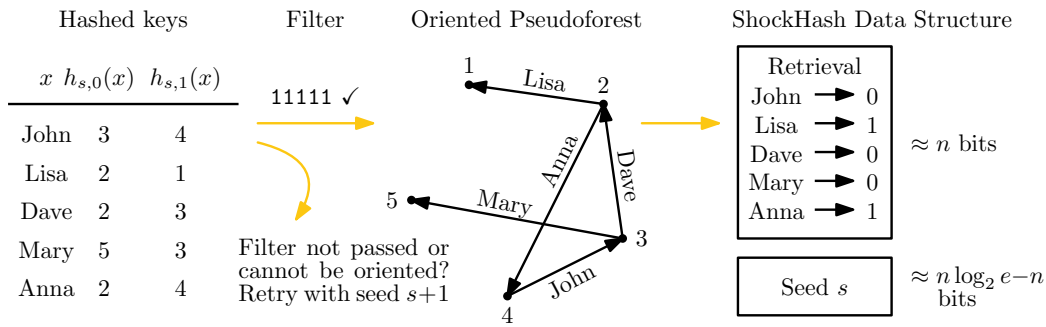
functions, the average number of hash functions is 5 as well. However, the expected space usage is just $50\% \cdot \log(2) + 50\% \cdot \log(8) = 2$ bits per key. At the same time, our analysis shows that the load threshold (see Section 2.5) is better.

Overloading. We reduce the space consumption further by using the novel idea to *overload* the cuckoo hash tables. The idea is to load the tables with more keys than the load threshold would permit (see Section 2.5). There are two factors that make overloading possible, both enabled by partitioning the keys to small buckets. First, the small tables lead to a higher variance in the achievable load factor. Second, we determine experimentally that the point at which insertions start to fail converges to the load threshold from *above* as n grows. Therefore, the construction of *small* overloaded tables is still likely to succeed. Using small tables also increases the cache locality during construction. Because we use overloading, we sometimes need to retry constructing a table, and store a hash function seed. Therefore, overloading is a small step towards brute-force. In typical configurations, we need just a little more than one try per bucket in expectation. We illustrate the construction in Figure 3.2.

Summary. The most space-efficient previous algorithms perform brute-force search as a core step to determine a perfect hash function. SicHash is more directed than this because it constructs cuckoo hash tables as its base case, which is possible in near linear time. The directedness is also visible in the experiments, where SicHash can construct PHFs with the same space requirements up to 4.3 times faster than competitors that have a similar query time. The novel combination of existing techniques keeps the SicHash queries extremely simple – basically the cost for a single access to a retrieval data structure. This further profits from recent advances on fast static retrieval data structures with virtually no space overhead [Dil+22]. For a more detailed explanation of SicHash, we refer to Chapter 6. We compare the performance of SicHash to other approaches in Section 3.4 after introducing our last minimal perfect hash function construction.

3.3 Small, Heavily Overloaded Cuckoo Hash Tables for Minimal Perfect Hashing

ShockHash – Small, heavily overloaded cuckoo hash tables – can be seen as an extreme version of SicHash where we use two hash functions for each key and retry construction



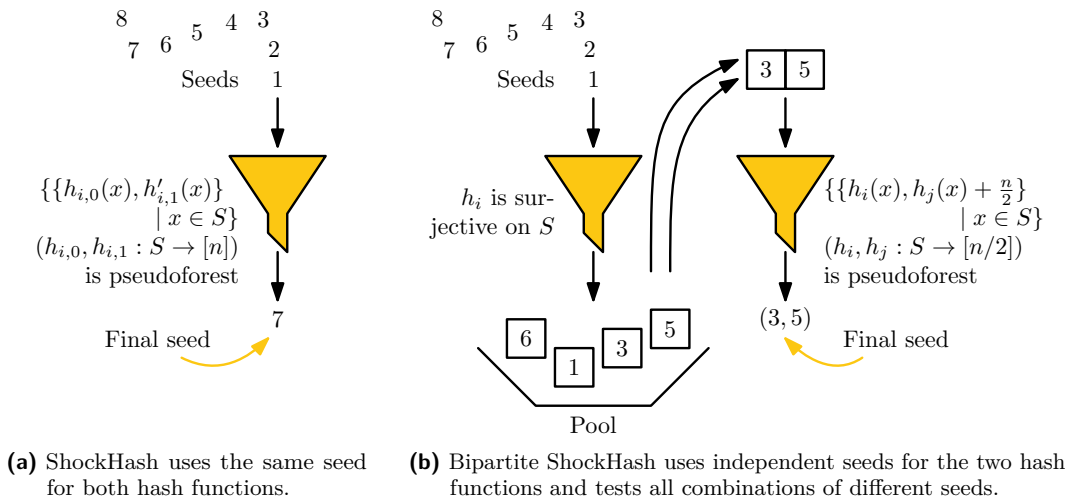
■ **Figure 3.3** Illustration of the ShockHash construction. Functions $h_{s,0}$ and $h_{s,1}$ are randomly sampled hash functions using a seed s . Here, s is a seed value where the resulting graph is a pseudotree. During construction, many seeds need to be tried.

until we can completely fill the small cuckoo hash table. That way, we achieve an MPHf without an intermediate non-minimal PHF. In graph terminology, ShockHash repeatedly generates a graph with n edges and n vertices. Each key corresponds to one edge, connecting the candidate positions of the key. The table can be filled if and only if the graph is a *pseudoforest* – a graph where no component contains more edges than nodes (see Section 2.5). While the ShockHash idea is straightforward in principle, we can *prove* that there is only an insignificant amount of redundancy in the retrieval data structure. We show that ShockHash approaches the information theoretic space lower bound for large n and has a running time of $(e/2)^n \cdot \text{poly}(n) \approx 1.359^n$ (nearly a factor 2^n faster than brute-force). Figure 3.3 illustrates the idea. An efficient bit-parallel filter can skip most of the checks for pseudoforests by checking if the two hash functions together are *surjective*.

Bipartite ShockHash. In *bipartite* ShockHash, further exponential improvements are possible. Instead of using a pair of fresh hash functions for each construction attempt, we build a growing pool of hash functions and consider all pairs that can be formed from this pool. Also, we let the two hash functions hash to disjoint ranges, meaning we effectively sample a bipartite graph where each edge has one endpoint in both partitions. In this bipartite setting, the hash functions of both partitions need to be *individually* surjective. We can therefore filter the set of candidate hash functions individually – before testing all combinations. This improves the construction time by an additional exponential factor, to about $1.166^n \cdot \text{poly}(n)$. Figure 3.4 illustrates the pool of hash function candidates.

Partitioning. Still being an exponential time algorithm, we use ShockHash as a building block after partitioning the input. While plain brute-force is practical for subsets of about 16 keys, bipartite ShockHash works with up to 128 keys. We obtain ShockHash-RS by using ShockHash instead of brute-force as a base case within the RecSplit framework. While there is a small penalty in query time due to the additional access to a retrieval data structure, ShockHash-RS is about two orders of magnitude faster to construct than tuned RecSplit (see Section 3.1) for space-efficient configurations. Bipartite ShockHash-RS improves this by a factor of 20 again. We also demonstrate that ShockHash is useful outside the RecSplit framework. When using a novel k -perfect hash function for partitioning the input, we obtain ShockHash-Flat, which achieves a space usage similar to the most space-efficient previous approaches. At the same time, it is faster to construct and reduces the query time by about 30%, which brings it closer to the query time of way less space-efficient approaches.

3.4 Practical Comparison of Modern Perfect Hashing



■ **Figure 3.4** Illustration of the filtering involved in ShockHash and bipartite ShockHash. The construction is complete if we find one final seed that passes all filters.

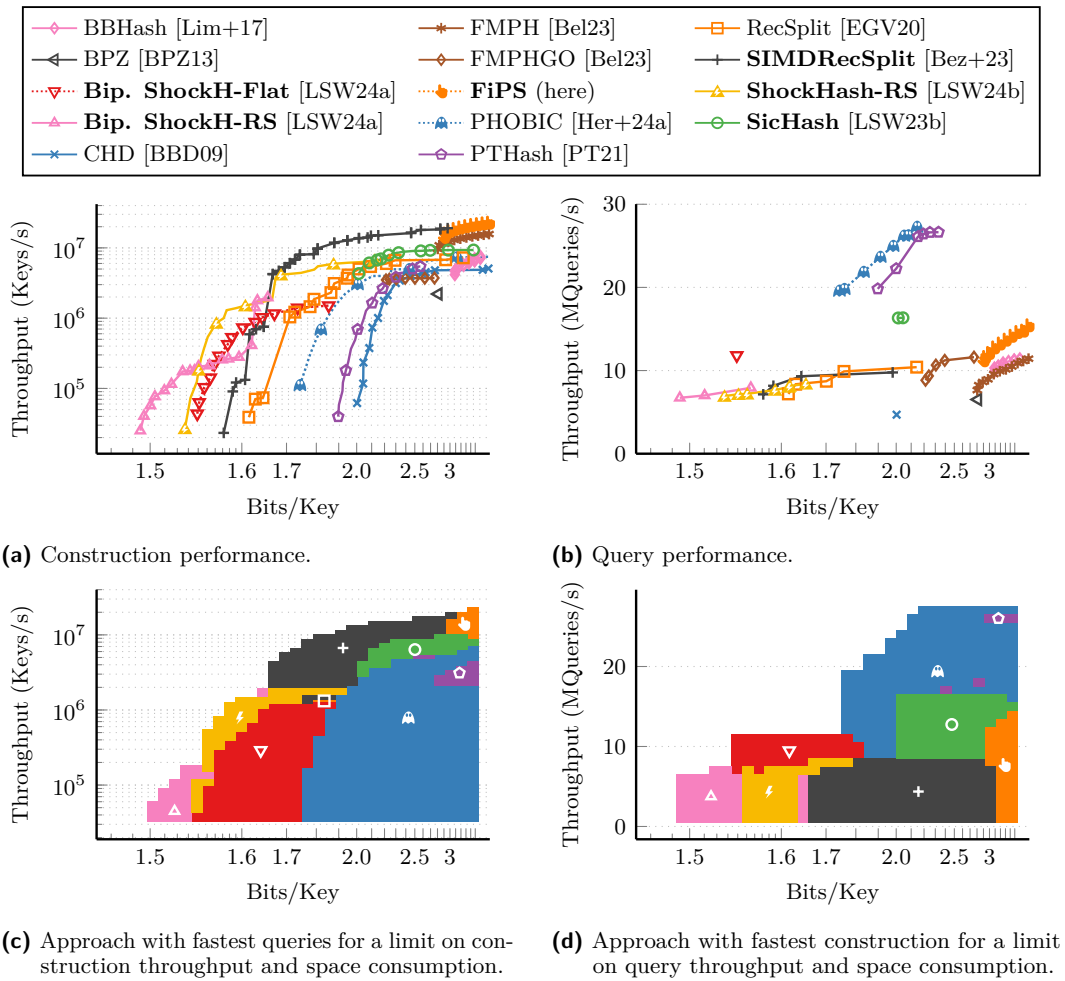
Summary. ShockHash is a new way to construct minimal perfect hash functions on small sets. By combining trial-and-error search with cuckoo hashing and retrieval data structures, ShockHash achieves an exponential speedup over plain brute-force. ShockHash is a major step away from brute-force while still reaching the space lower bound. Bipartite ShockHash maintains a pool of candidate hash functions and gives additional exponential improvements. By filtering out candidate hash functions before combining them, it improves the speedup to well over 2^n compared to plain brute-force.

Constructing with a single thread, ShockHash-RS is even faster than a tuned GPU implementation of the brute-force technique (see Section 3.1) achieving the same space consumption. For a more detailed explanation of ShockHash, we refer to Chapter 7.

3.4 Practical Comparison of Modern Perfect Hashing

We now compare the three minimal perfect hash function constructions introduced in the sections above with approaches from the literature. A difficulty here is that all main performance parameters – space usage, query time, and construction time – can be equally important. One approach might have fast construction and queries, but need a lot of space. Another approach might be space-efficient and fast to construct, but has slow queries. In this section, we only give a short comparison. To read more about the competitors, we refer to Chapter 4. For a full discussion, as well as an explanation of the experimental setup, we refer to Chapter 8. For now, it is enough to know that we perform single-threaded experiments on a consumer machine. In our figures, we plot the Pareto front containing only measurements that are not dominated by another configuration of the same competitor.

Construction. In Figure 3.5a, we plot the construction throughput. Even though our SIMDRecSplit [Bez+23] (see Chapter 5) is designed for space-efficient configurations, it surprisingly has the fastest construction up to a space consumption of 3 bits per key. For even larger perfect hash functions, FiPS (see Section 4.5), our simple implementation of the fingerprinting [Mü+14] approach, becomes most efficient. Directly after SIMDRecSplit, we have SicHash [LSW23b] (see Chapter 6), which has its best configurations at about



■ **Figure 3.5** Comparison of our perfect hash functions with approaches from the literature using 100 million input keys. Uses logarithmic x -axis (to the space lower bound) and y -axis. For a full comparison, we refer to Chapter 8.

2–3 bits per key. Most other competitors are significantly slower to construct. Looking at the more space-efficient approaches, we can see how the approaches presented in this dissertation move closer and closer to the space lower bound. In particular, bipartite ShockHash-RS [LSW24a] (see Chapter 7) reduces the gap to the space lower bound from 0.16 bits per key (RecSplit [EGV20]) to just 0.05 bits per key. This is a reduction by more than 60%, while still maintaining the same construction time.

Queries. Figure 3.5b gives measurements of the query throughput. PTHash [PT21] and PHOBIC [Her+24a] are the clear winners in this regard, with PHOBIC achieving a better space consumption. Note that PHOBIC is not described in this dissertation but originates from a master’s thesis supervised by the author of this dissertation. Compared to PTHash and PHOBIC, SicHash trades faster construction for slower queries. As shown in Figure 3.5d, SicHash achieves a good trade-off between the performance parameters. The queries of the RecSplit based approaches (ShockHash-RS, RecSplit, SIMDRecSplit) are rather slow. This is because they need to traverse the compressed splitting tree. Bipartite ShockHash-

Flat [LSW24a] (see Chapter 7), which does not need the tree, shines with significantly faster queries than other approaches with the same space consumption. FiPS dominates the other implementations of the fingerprinting idea, BBHash [Lim+17] and FMPH [Bel23].

Summary. The approaches presented in this dissertation dominate a wide range of the space vs construction time trade-off. During the course of just a few years, they bring the space consumption of practical perfect hashing significantly closer to the lower bound. The dominance map in Figure 3.5c shows the approach with the fastest queries, given a limit on space consumption and construction throughput. Similarly, Figure 3.5d shows the fastest construction for a given limit on space consumption and query throughput. The dominance maps show that the approaches presented in this dissertation dominate the full trade-off between the three performance parameters. Of the competitors, only PTHash [PT21] and PHOBIC [Her+24a] can fill some of the area.

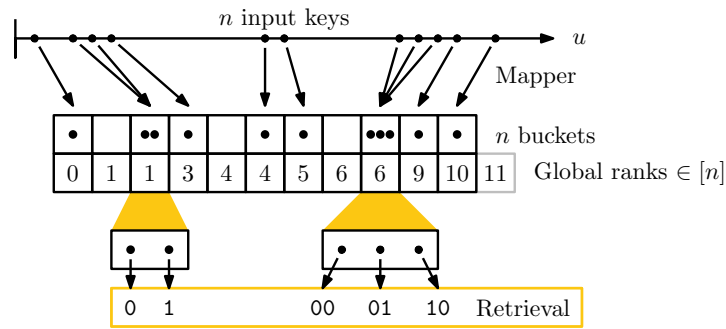
3.5 Learned Monotone Minimal Perfect Hashing

A *monotone* minimal perfect hash function (MMPHF) is a hash function that retains the natural order of the keys in S . In other words, the hash function maps keys from S to their rank, and returns an arbitrary value for keys not in S (see Section 1.1). Despite the widespread use of MMPHFs and recent advancements on their asymptotic bounds [AFK23; Kos24], the practical implementations have not made significant progress in terms of new designs and improved space-time trade-offs since their introduction more than a decade ago [Bel+11]. Only some exceptions target the query time [GO14]. Existing approaches are mostly based on building a trie-like data structure on the keys.

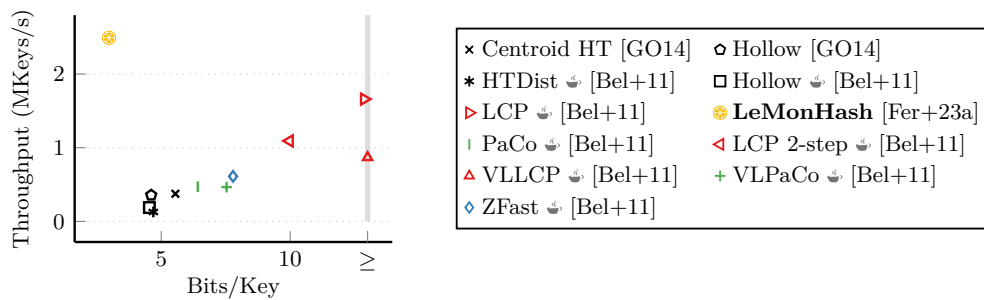
LeMonHash. With \otimes LeMonHash, we offer a fresh new perspective on MMPHFs. We build upon recent advances in (learning-based) index data structures, namely the PGM-index [FLV21; FV20b], and in retrieval data structures, namely BuRR [Dil+22]. The former learns a piecewise linear approximation mapping keys in S to their rank estimate. The latter allows associating a small fixed-width integer to each key in S , without storing S . We combine these two seemingly unrelated data structures in a surprisingly simple and effective way. First, we use the PGM to monotonically map keys to buckets according to their rank estimate, and we store the global rank of each bucket’s first key in a compressed data structure. Second, since the rank estimates of some keys might coincide, we solve such bucket collisions by storing the local ranks of these keys using BuRR. We call our proposal *LeMonHash*, because it *learns* and *leverages* the smoothness of the input data to build a space-time efficient *monotone* MPHf. We illustrate the data structure in Figure 3.6.

In contrast to MPHFs, where the input distribution usually does not matter, MMPHFs are strongly influenced by the input. In Figure 3.7 we give measurements with synthetic data having an exponential distribution. In Section 9.6, we give a full evaluation with different distributions. The figure shows that LeMonHash is faster to query than the fastest competitor. Simultaneously, it is more space-efficient than the most space-efficient competitor. We get a similar view for the construction time (see Section 9.6), making LeMonHash a significant step forward from the previous state of the art.

LeMonHash-VL. We also extend LeMonHash to support variable-length string keys, and obtain LeMonHash-VL. The idea is to look at fixed-width prefixes of the keys. Prefixes might collide, either because many strings share the same prefixes or because the learned mapping



■ **Figure 3.6** Illustration of the LeMonHash data structure. LeMonHash maps input keys to an approximate rank and solves collisions using retrieval.



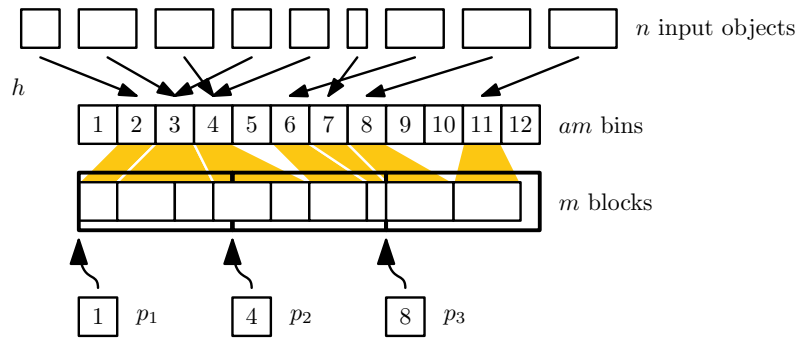
■ **Figure 3.7** LeMonHash query performance for synthetic input with exponential distribution. Competitors with the \cup symbol in the legend are implemented in Java.

is imperfect. If too many prefixes collide, we use the same data structure recursively on the bucket. Before recursing, we remove the longest common prefixes of the keys in the bucket. LeMonHash-VL therefore consists of trees, but they are significantly more flat and efficient to traverse than competitors. This enables extremely fast queries with space consumption similar to competitors.

Summary. LeMonHash, unlike previous solutions, learns and leverages data smoothness to obtain a small space usage and significantly faster queries. On most synthetic and real-world datasets, LeMonHash dominates all competitors – simultaneously – on space usage, construction throughput and query throughput. Our adaption to variable-length string keys, LeMonHash-VL, needs space within 13% of the best competitors while being up to 3 times faster to query. We refer to Chapter 9 for more details, including implementation tricks.

3.6 Perfect Hashing for Variable Size Objects

Using perfect hashing, it is possible to build static hash tables that guarantee access in a single memory access (see Section 1.4). This is especially interesting for external memory applications, but is relevant for internal data structures as well to reduce cache faults. In this section, we focus on the external memory case. Because external memory is organized in *blocks* that are loaded as a whole, we often only need to know which block an object is stored in, not the exact location. The object itself can then be found by loading and searching the block. This is a typical application of minimal k -perfect hashing, where each of the n/k output values is hit by exactly k keys (see Section 1.1).



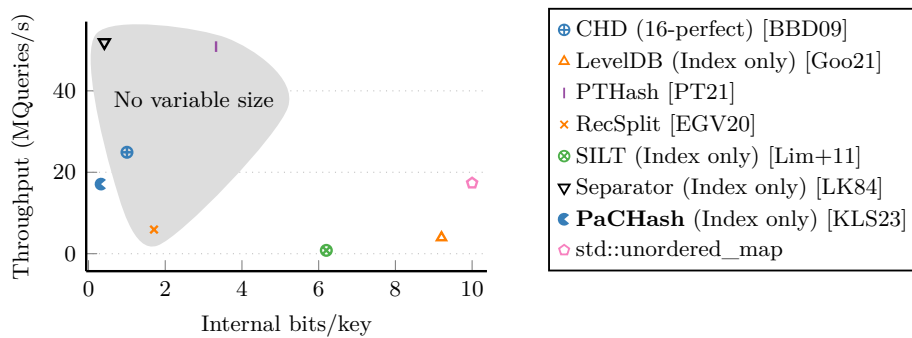
■ **Figure 3.8** Example of a PaCHash data structure with $a = 4$.

Unfortunately, perfect hashing or classical hash tables [ANS10; Ben+23; Fot+05; Knu98; KPR22; PR04] can only be used with objects of identical size which makes it impossible to compress the objects with variable bit-length codes. Most hash tables for objects of variable size store references from table entries to the actual data which entails a significant space overhead. Alternatively, they make the table cells large enough that most objects fit in, which leaves a lot of space unused.

Our data structure PaCHash (**P**acked and **C**ompressed **H**ash Tables) works similar to a k -perfect hash function, giving an approximate location of an object. However, in contrast to a k -perfect hash function, it supports objects of variable size. PaCHash eliminates fragmentation by *packing* the objects contiguously in external memory without leaving free space. It then uses a highly space-efficient search data structure that translates queried objects to memory locations. More precisely, objects are first hashed to *bins*. The bins are stored contiguously in m *blocks* of size B . For each external memory block i , PaCHash stores the first bin index p_i that intersects the block boundary. It does so using the compressed Elias-Fano representation (see Section 2.3), which can be searched efficiently. Figure 3.8 illustrates the overall data structure.

Queries. To search for an object, we perform a predecessor query for its bin on the Elias-Fano coded sequence. PaCHash then yields a near-optimal range of blocks that contain the object. We show in Chapter 10 that this predecessor query needs constant expected time in our case. Our analysis basically shows that for a tuning parameter a , the expected number of block reads to retrieve an object x of size $|x|$ is about $1 + 1/a + |x|/B$ while the internal memory data structure needs $2 + \log(a)$ bits per *block*.

Identical Size Objects. For objects of identical size s , perfect hashing enables finding the exact block that an object is stored in and requires a constant number of bits per object. PaCHash approximates this when choosing $B = s$, also needing a (slightly larger) constant number of bits per object but lower construction time. The picture changes when we look at larger block sizes $B = ks$ and the corresponding approach of minimal k -perfect hashing. Now, PaCHash still needs only a constant number of bits per block, while there is a *lower bound* of $\Omega(\log k)$ bits per block using Mk PHs (see Section 2.7). This comes at the price of sometimes accessing one external memory block too much. Figure 3.9 compares PaCHash with competitors from the literature when using objects of identical size. PaCHash outperforms approaches that support objects of variable size, and is close to or even outperforms approaches that only handle objects of identical size. We refer to Section 10.5 for a more detailed comparison including the experimental setup.



■ **Figure 3.9** Space and query throughput of the PaCHash internal memory index data structure compared with competitors. Does not contain any I/O. Approaches marked with gray background only support objects of identical size.

Summary. In contrast to previous approaches, PaCHash is the first construction that combines single-access retrieval, support for variable size objects, and full external memory utilization. Our implementation of PaCHash considerably outperforms previous object stores for variable size objects and even matches or outperforms systems that are purely internal memory or only handle objects of identical size. For more details, we refer to Chapter 10.

3.7 Summary

In this dissertation, we present three novel minimal perfect hash function constructions. First, we extend RecSplit to efficiently evaluate additional brute-force trials using rotation fitting. Together with a highly tuned parallel implementation, our SIMDRecSplit constructs perfect hash functions very efficiently. On the opposite end of the spectrum we have SicHash, which uses cuckoo hashing and retrieval to determine perfect hash functions without the need for brute-force. Even without partitioning, its construction needs almost linear time, but it does not reach the space lower bound. Finally, ShockHash opens an interesting and novel middle ground between these two. It still uses brute-force and asymptotically reaches the space lower bound, but the majority of the stored data does not come from the brute-force search. Instead, most of the data is stored in a retrieval data structure, whose state can be determined in near linear time by cuckoo hashing. While developing parallel RecSplit, well engineered brute-force seemed to be the best way to get close to the space lower bound. However, ShockHash gives a sophisticated algorithm that significantly prunes the search space. This changes the view completely and brings algorithm design back into business.

Our new approaches cover different areas of the three main performance parameters. SIMDRecSplit offers good space consumption and fast construction. SicHash offers fast construction and fast queries. Finally, ShockHash focuses on small space consumption while at the same time providing a configuration that is competitive in terms of query time. Our three algorithms bring the research area of perfect hashing a significant step forward.

As a closely related problem, we also study *monotone* minimal perfect hashing. The key ingredient in our LeMonHash is to learn regularities in the input data and to build a data structure that is significantly more flat than competitors. LeMonHash often outperforms all competitors with respect to all three performance parameters – space, construction time, and query time – simultaneously.

Finally, we study perfect hashing for variable size objects. A small internal search data

structure can locate the external memory location of each object in expected constant time. The internal memory search data structure, in a way, breaks the space lower bound of k -perfect hashing at the cost of sometimes accessing one external memory block too much. PaCHash matches or even outperforms competitors from the literature that are purely internal memory or only handle objects of identical size.

4 A Brief History of Perfect Hashing

Summary: *Perfect hashing is an active field of research with a large number of publications. Modern approaches can handle millions to billions of input keys and achieve space close to the lower bound. This chapter starts with explaining the origins of perfect hashing. It then categorizes modern perfect hash functions from the literature by their main working principle and provides a comprehensive overview over the different approaches.*

Attribution: In this chapter, we describe existing approaches from the literature. Some of the descriptions are based on the related work sections of our previous papers [Bez+23; LSW23b; LSW24b]. However, we enhance them with more details and figures. The categorization and the analysis of how the approaches influence each other are new. The description of the origins of perfect hashing is new as well. We also introduce the new and simple perfect hash function FiPS.

Perfect hashing is an active field of research with a large number of publications. Since the last comprehensive survey [CHM97], significant progress has been made in the field. This can make it hard to get an overview over the state of the art when newly getting into the topic of perfect hashing. In this chapter, we start with a brief insight into the origins of perfect hashing in Section 4.1. It illustrates the large progress that was made during the last 50 years of research. We then give a comprehensive survey with a focus on modern approaches. This can serve as a starting point to get familiar with the topic of perfect hashing. Additionally, together with the evaluation in Chapter 8, it can serve as a guide to select a perfect hash function for use in a specific application. In Section 4.2, we first present a categorization of the approaches into *random hypergraphs*, *brute-force* and *fingerprinting*. We then explain the approaches in Sections 4.3–4.5. This chapter includes short descriptions of our own approaches as well. This makes it possible to read this survey in isolation from the remaining dissertation.

4.1 The Birth of Perfect Hashing

In this section, we give a short overview over the origins of perfect hashing. Most of the approaches are no longer relevant in practice today, but they illustrate the progress that was made on the topic during the last 50 years of research. Also, the approaches already introduce a number of basic principles that are still used in modern constructions. We explain modern approaches starting with Section 4.2.

In 1963, for identifying reserved words in an assembler, Grenievski and Turski [GT63] present a function that can convert symbols to integers without collisions. The idea is to use a linear congruential generator and determine constants experimentally such that it 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 [Knu73] of *The Art of Computer Programming*, 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 of his book [Knu98] then already describes the first practical perfect hash function constructions.

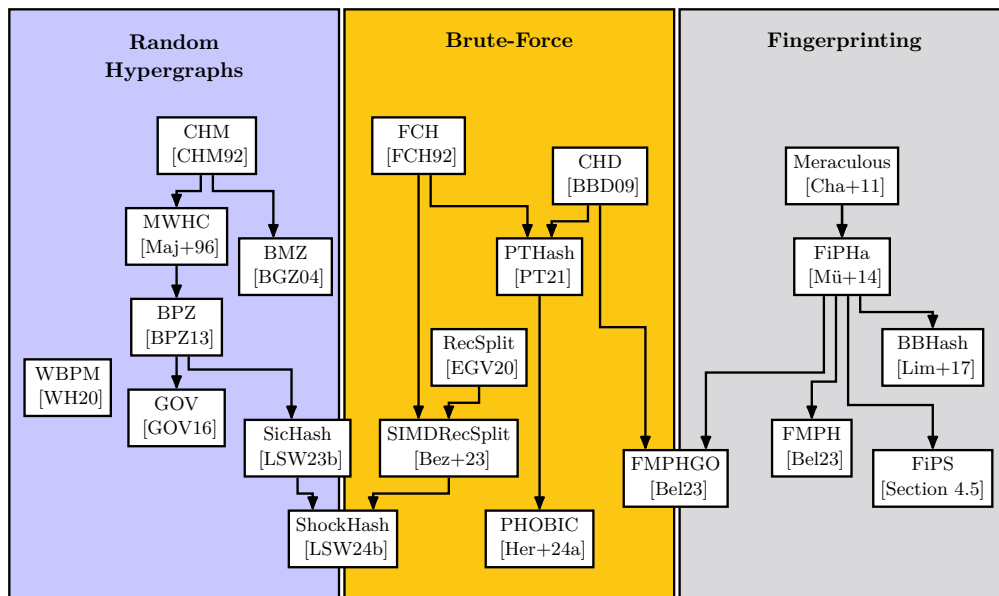
Sprugnoli [Spr77] introduces the terms *perfect hashing* and *minimal perfect hashing* in 1977. He also describes the first algorithm to systematically construct perfect hash functions. The idea is to find a linear transformation of the input keys, such as $x \mapsto (x + c_1)/c_2$, where c_1 and c_2 are constants determined by the algorithm. This approach is highly dependent on the distribution of the input keys and works well only for uniform distribution. 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: An initial *master hash code* (MHC) is generated using a high quality non-perfect hash function with large output range. This makes the construction mostly independent of the input distribution. A second fundamental idea introduced by Sprugnoli is the idea of *bucketing* or *partitioning*. In order to determine a perfect hash function for a larger key set, it is possible to first divide the input keys into small subsets of approximately the same size, for example using a non-perfect 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.

Sprugnoli 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 [Sch90].

Cichelli [Cic80] 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 [Jae81] gives an algorithm that can already handle input sets of up to 1000 keys.

Lewis and Cook [LC88] review early letter-based approaches. Czech et al. [CHM97] give a review of additional approaches until the year 1997. We do not go into detail about these here because most of the approaches are no longer relevant in practice today.

Theoretical Construction. While our survey is mainly focused on practical approaches, we also want to mention an important theoretical result. A tempting way to replace expensive brute-force search is precomputation of solutions with subsequent table lookup – a standard technique used in many compressed data structures. For a rough idea, suppose for a subproblem with s keys, we first map them injectively to a range of size $U' \in \Omega(s^2)$ using an intermediate hash function. A smaller range would lead to collisions, as illustrated by the birthday paradox. Then, using a lookup table of size $2^{U'}$, we can find precomputed perfect hash functions in constant time. Polynomial running time limits the subproblem size to $s \in \mathcal{O}(\sqrt{\log n})$. Hagerup and Tholey [HT01] develop this approach to a comprehensive theoretical solution of the perfect hashing problem yielding linear construction time, constant query time, and space $1 + o(1)$ times the lower bound. However, this approach is merely a theoretical result and not practical. In fact, it is not even well-defined for $n < 2^{150}$ [BPZ13] because it creates buckets that would then have an expected size of less than one key. Therefore, (minimal) perfect hashing remains an interesting topic for algorithm engineering. A long sequence of previous work has developed a range of practical approaches with different space-time trade-offs.



■ **Figure 4.1** Modern perfect hashing approaches and how they influence each other.

4.2 Categorization

After this short introduction to the origins of perfect hashing, we now consider more modern approaches. Even though there is a wide range of constructions in the literature, many of them use similar overall techniques to achieve their goal. In Figure 4.1, we categorize the approaches presented in this survey into the following three techniques: *random (hyper)graphs*, *brute-force*, and *fingerprinting*. Additionally, there are some techniques that combine ideas from multiple of these categories. The figure also shows how the approaches influence each other. In the following, we describe the categories in more detail.

■ **Random (hyper)graphs.** Some of the first constructions are based on orienting random (hyper)graphs, and these approaches are still in use today. The idea is that each key is represented by an edge in the graph. Depending on the approach, nodes can be candidate output values or can store data that is needed during queries. This usually leads to polynomial time construction algorithms that are, however, not as space-efficient as other approaches. We describe perfect hash functions using random hypergraphs in Section 4.3.

■ **Brute-Force.** We already described a simple construction that tries many hash function seeds using brute-force in Section 1.1. While this is not practical for large key sets, the most space-efficient approaches to construct PHFs still use brute-force search as a base-case. Many of these approaches can also be shown to approach the space lower bound when selecting their tuning parameters accordingly. We refer to Section 4.4 for details.

■ **Fingerprinting.** Our final category is the use of fingerprinting. This technique hashes each key to a fingerprint and indicates collisions between the fingerprints using a bit vector. It handles keys with colliding fingerprints recursively. While its space consumption is higher than other approaches, it can be very efficient to construct and query. We explain perfect hashing through fingerprinting in Section 4.5.

4.3 Random Hypergraphs

While basically all perfect hashing approaches can be interpreted as a graph $G = (V, E)$ in some way, the approaches described here do it much more explicitly. Some of the approaches do not have an explicit name in their paper. In that case, it is common to name them after the first letters of the authors' last names. Table 4.1 gives an overview over the properties of the (hyper)graphs that each approach uses. Looking at the space consumption, the table shows the steady progress made in each paper.

In the following, some approaches are based on graph peeling. This is the process of iteratively taking away nodes of degree 1, together with their adjacent edge [BPZ13; Wal21]. We call a graph *peelable* if it can be peeled to an empty graph.

CHM. Czech et al. [CHM92] construct an undirected graph with $|V| = 2.09n$ nodes. They generate the edges $E = \{(h_1(x), h_2(x)) \mid x \in S\}$ by applying two independent hash functions on each input key. To query the function with a key x , they consider the edge $e = (u, v) = ((h_1(x), h_2(x)) \in E$. The final hash value is given by $h(x) = g(u) + g(v) \bmod |V|$. The construction of CHM determines the function g , which assigns an integer to each node. For this, CHM enumerates the edges and assigns a desired $\log n$ bit output value to each edge. It starts with an arbitrary node y and assigns $g(y) := 0$. Then, performing a depth-first-search, CHM calculates the values of g of the neighbors by simple subtraction. This assumes that the graph is peelable. If the graph is not peelable, which rarely happens with n edges and $|V| = 2.09n$ nodes, CHM retries with another set of hash functions. CHM stores $2.09n$ integers and therefore needs $\mathcal{O}(n \log(n))$ space. It can even store order-preserving perfect hash functions (see Section 1.1). Note that space near $n \log n$ can also be achieved through a retrieval data structure (see Section 2.4) that stores the output value for each key explicitly.

MWHC. The overall idea of MWHC [Maj+96] is similar to CHM. Instead of a graph, MWHC now generates a random *hypergraph* with $|V| = cn$ nodes and n edges. The variable c is a tuning parameter that influences the construction success probability. Each edge connects r nodes, determined by r hash functions, where $r = 3$ for the best space efficiency. Like CHM, it determines a function g , assigning an integer to each node. The function is selected such that the sum of all nodes connected to an edge is unique modulo $|V|$. MWHC can therefore be seen as a generalization of CHM to edges connecting more than two nodes.

Because MWHC now deals with a hypergraph instead of a graph, assigning the values to nodes is now no longer possible using depth-first-search. The graph property required for the construction to succeed is peelability, even though the MWHC paper calls it *acyclicity*. Let us take a peelable graph and re-insert all edges in the reverse order of the peeling process. Then each inserted edge is adjacent to at least one node that no other edge is adjacent to yet. MWHC uses this reverse peeling order to assign a value to each node. Like CHM, it can assign an arbitrary number to each edge, which makes MWHC an order-preserving perfect hash function. Because, in contrast to CHM, it uses r hash functions, the peelability threshold, i.e., the proportion between $|V|$ and $|E| = n$ at which peeling likely succeeds, is higher. This makes it possible to reduce the number of nodes to $|V| = 1.23n$ and therefore store only $1.23n$ integers. Belazzougui et al. [Bel+14] improve the peeling step of MWHC through better cache locality. MWHC can also be used as a retrieval data structure (see Section 2.4). However, used for perfect hashing, it still needs space in $\mathcal{O}(n \log n)$, far above the space lower bound.

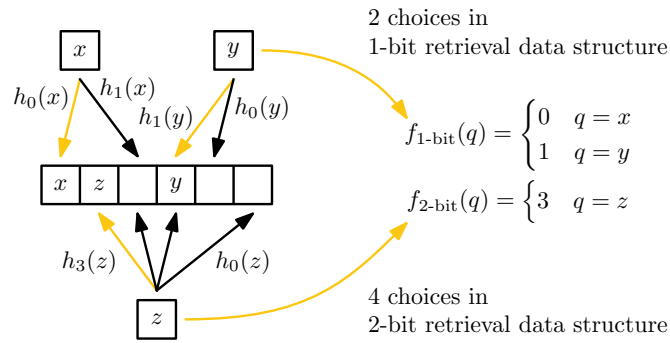
■ **Table 4.1** Overview over properties of approaches based on random graphs.

Approach	Nodes per edge	Graph Property	$ V $	Space Consumption
CHM [CHM92]	2	Peelable	$2.09n$	$2.09 \cdot n \log n$
MWHC [Maj+96]	3	Peelable	$1.23n$	$1.23 \cdot n \log n$
BMZ [BGZ04]	2	$ \text{Critical} < n/2$	$1.15n$	$1.15 \cdot n \log n$
BPZ [BPZ13]	3	Peelable	$1.23n$	$1.23 \cdot 2n$
GOV [GOV16]	3	Orientable	$1.15n$	$1.15 \cdot 2n$
WBPM [WH20]	$\log n$	Bipartite	$2n$	$1.83n$
SicHash [LSW23b]	Mix 2, 4, 8	Orientable	$(1 + \varepsilon)n$	$2n$
ShockHash [LSW24b]	2	Orientable	n	$1.44n$

BMZ. BMZ [BGZ04] enhances the idea of CHM into another direction than MWHC. In contrast, the graph does not need to be peelable. Like CHM, the hash value is given by 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. It 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 a 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 $1.15n$ integers.

BPZ. The BPZ algorithm, also called RAM algorithm, is introduced in Ref. [BPZ07b]. In the cmph library [CR+12], the algorithm is called BDZ. An external memory implementation (later called HEM) is introduced in Ref. [BZ07]. The journal article [BPZ13] offers a more complete analysis. In BPZ, each input key is mapped to an edge in a random hypergraph using r independent hash functions h_i , like in MWHC. By peeling the graph, BPZ determines a 1-orientation, i.e., a unique mapping from edges to nodes. Let $i(x)$ be a function assigning a number to each edge, indicating which hash function index gives the node it is oriented towards. Then $h_{i(x)}(x)$ is unique for each key x . General perfect hashing through retrieval (see Section 2.6) now stores the function $i(x)$ in a retrieval data structure viewed as a black box. Instead, BPZ describes a concrete construction of a retrieval data structure that uses the same underlying graph. It solves a linear equation system to determine a function g , such that $i(x) = \sum_{0 \leq i < r} g(h_i(x)) \bmod r$. In contrast to the previous graph-based approaches that stored an integer of $\log n$ bits for each node, BPZ only needs to store $\lceil \log r \rceil$ bits.

GOV. The GOV algorithm [GOV16] is based on BPZ, but uses a higher load factor $|E|/|V|$ to make the resulting hypergraph likely orientable, but no longer peelable. This reduces the number of nodes and therefore hash function indices that need to be stored. After peeling away as many edges as possible, GOV solves the remaining assignments as a linear equation system. For this, it introduces a range of engineering tricks. For example, it uses *broadword programming* by packing multiple values into a single word and running calculations on all of them at once. Also, it introduces *lazy Gaussian elimination*, which is a heuristic for reducing the size of the equation system. Using three hash functions, and therefore two bits per key, this reduces the space consumption to $1.15 \cdot 2n$ bits.



■ **Figure 4.2** Illustration of SicHash. A classification hash function determines that key z gets 4 choices, while the others get only 2 choices each. The hash function choice is then stored in the corresponding retrieval data structure.

SicHash. SicHash [LSW23b] uses perfect hashing through retrieval, like BPZ. However, it separates the two tasks of key placement and storage. It uses a retrieval data structure as a black-box to store the n hash function indices, one for each input key. In contrast, BPZ needs to store $|V| = 1.15 \cdot 2n$ integers, one for each node of the graph. An additional advantage of this idea is that it can use multiple retrieval data structures of different widths. SicHash’s main innovation is using a careful mix of 1–3 bit retrieval data structures. For queries, SicHash first hashes the key to determine the number of choices. It then looks up the key in the retrieval data structure of the appropriate width. Finally, it hashes the key with the hash function index retrieved from the retrieval data structure. Figure 4.2 gives an example for a SicHash perfect hash function.

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 hypergraph beyond the point where it is likely orientable. For this, it uses small sets and retries the construction multiple times. SicHash exploits the variance in the number of keys that can fit, as well as the fact that the load factor at which construction likely fails converges to the load threshold from *above* as n grows. For a more detailed explanation of SicHash, we refer to Chapter 6.

WBPM. Weaver et al. [WH20] describe an algorithm that builds on perfect hashing through retrieval. However, it uses a significantly larger number of hash function choices and tries to select small hash function indices. WBPM construction creates a weighted bipartite graph. The left set of nodes is given by the n input keys and the right set is given by the n possible hash values. This is similar to the bipartite graph in cuckoo hashing (see Figure 2.4a). The edges are determined by applying $\mathcal{O}(\log(n))$ hash functions to each key, where an edge determined from the i -th hash function has weight i . WBPM then determines a minimum weight bipartite perfect matching (WBPM), which can be shown to reach a weight of $1.83n$. This matching indicates which hash function to use for each key, which WBPM stores using a 1-bit retrieval data structure. The retrieval data structure stores the hash function indices using unary coding. This is possible by indexing it using tuples of key and bit index. A minimal matching therefore also minimizes the space consumption of this scheme. The resulting weight of the matching of $1.83n$ therefore also equals the space consumption of the final data structure, except for overheads like prefix sums due to bucketing.

SAT Encoding. In addition to WBPM, Weaver et al. [WH20] also describe an MPH construction close to the space lower bound based on encoding the construction in SAT. The data structure is a sequence of bits, and a hash function determines for each key which of these bits (possibly flipped) should be concatenated to give the final hash function value. The SAT encoding is based on an *all-different* constraint and is feasible for $n \leq 40$.

4.4 Brute-Force

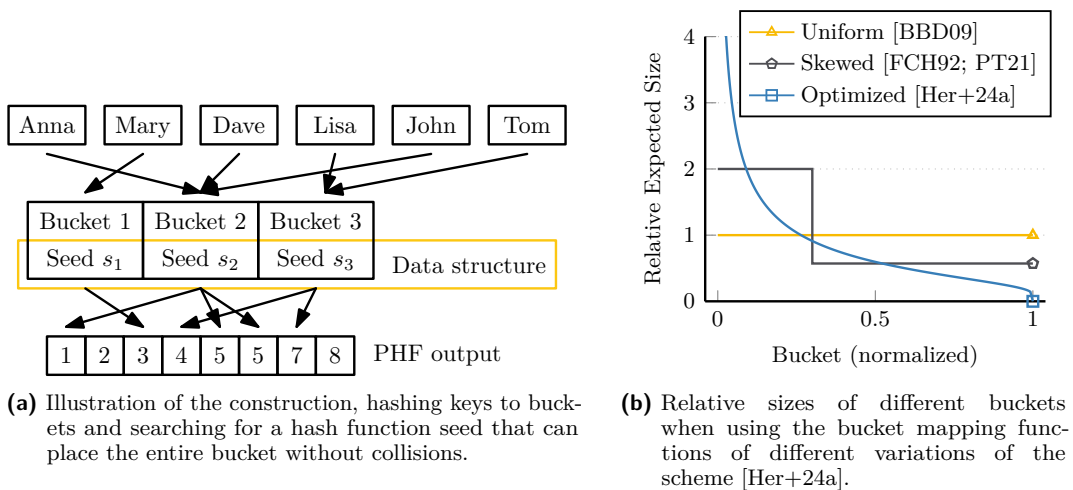
The fact that brute-force can be useful for constructing perfect hash functions is mentioned early after the discovery of perfect hashing [Cic80]. Today, most of the approaches achieving the best space efficiency are based on brute-force techniques in their base case. In the following, we give an introduction to the modern approaches based on brute-force.

Perfect Hashing Through Bucket Placement. A common approach is perfect hashing through bucket placement [BBD09; FCH92]. We now describe the overall idea before continuing with several concrete implementations. The idea is to first hash each key x to a small bucket $b(x)$. The average size of each bucket is usually about 5–10 keys. For each bucket b , the approach greedily determines a hash function seed $i(b)$ such that its keys do not collide with keys of previously placed buckets or with one another. Because it places the first buckets into an almost empty output domain, these are significantly easier to place. Additionally, buckets with fewer keys are easier to place. To accelerate the search, it is therefore useful to sort the buckets by their size and insert the largest ones first while the output domain is less full. A query for a key x then simply needs to evaluate $h_{i(b(x))}(x)$ to get the resulting hash value. We illustrate the structure of perfect hashing by bucket placement in Figure 4.3a.

FCH. FCH [FCH92] is the first perfect hash function construction that uses the bucket placement idea. It reserves a fixed number of bits per bucket to store the seed. To amplify the effect that the first buckets are easier to place, it uses an asymmetric bucket assignment: it hashes 60% of the keys to 30% of the buckets. The hash function that FCH uses is based on additive displacements. This means that it hashes all keys and then generates additional candidate positions by linearly shifting all keys in the bucket. If all of the displacements cause collisions, FCH has one additional bit per bucket that indicates to switch to a fallback hash function with all its displacements. If both hash functions do not work, the construction fails. FCH produces MPHs with about 2–3 bits per key.

CHD. CHD [BBD09] uses perfect hashing through bucket placement as well. While it does not explicitly build on FCH, it is published later and uses a very similar structure. Instead of using an asymmetric assignment from keys to buckets, it hashes the keys to buckets uniformly. Instead of reserving a fixed number of bits for the seeds, it just retries until one seed does not cause collisions. In each retry, CHD uses a fresh hash function seed instead of using displacements. CHD then uses the variable bit length code by Fredriksson et al. [FN07] to store the seeds in a compressed way. With a load factor of 81%, CHD can construct a PHF using 1.4 bits per key. With a load factor of 99%, it achieves 1.98 bits per key.

PTHash. PTHash [PT21] combines FCH and CHD. It hashes 60% of the keys to 30% of the buckets like FCH. Like CHD, it stores the seeds in compressed form. PTHash offers several compression schemes with different trade-offs between space and query time. Using



■ **Figure 4.3** Perfect hashing through bucket placement.

an appropriate compression scheme, only a single memory access is required to find the hash value, and the remaining operations use simple arithmetic. To speed up searching for the hash function in each bucket, it first generates a non-minimal PHF, but it compresses the output later by remapping values larger than n (see Section 2.8).

As a hash function, PTHash uses so-called *XOR displacement*, where it calculates the XOR of a hash of the seed with the input key. During construction, this has the advantage that it only needs to evaluate the full hash function once and can then hash each key using a simple XOR operation. However, due to regularities in the output, this approach only works for large output domains [Her+24a]. PTHash constructs minimal perfect hash functions with a space consumption of 2–4 bits per key. PTHash-HEM [PT24] is an implementation that first partitions the input and then constructs each partition independently in parallel.

PHOBIC. PHOBIC [Her+24a] is based on PTHash. It refines the idea originating from FCH to generate two expected bucket sizes and instead gives a different expected size to *every* bucket. PHOBIC introduces an optimal bucket assignment function that ensures that, asymptotically, each bucket has the same probability to be placed successfully. This minimizes the construction time. Figure 4.3b plots the relative expected bucket sizes using different bucket assignment functions. In comparison to FCH, PHOBIC makes the first and easiest to place bucket much larger.

Because the actual bucket sizes need to be integers, PHOBIC cannot completely reach the optimal values. Therefore, not all seeds have the exact same distribution. To still store the seeds efficiently, PHOBIC introduces *interleaved coding*. It partitions the input set and calculates an independent MPHf on each partition. By selecting the same number of buckets in each partition, the i -th bucket of every partition has the same distribution. PHOBIC then stores one compressed sequence per bucket, holding the seeds of all partitions. Together, we get **P**erfect **H**ashing with **O**ptimized **B**ucket sizes and **I**nterleaved **C**oding – PHOBIC.

The rather small partitions also enable a fast GPU implementation. For its hash function, PHOBIC uses a mix between additive displacements (like FCH) and full retries (like CHD). It uses the lower bits of its seed to store the additive displacement and the upper bits to store the actual hash function seed. Compared to PTHash, PHOBIC saves up to 0.17 bits per key, while still having the same construction and query throughput.

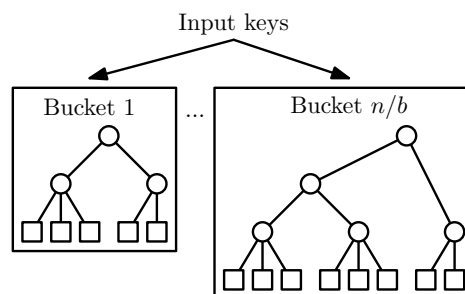
RecSplit. As we have seen in Section 1.1, the simple brute-force construction for perfect hashing is not practical for large n . RecSplit [EGV20] uses brute-force in a novel way to enable large key sets with space close to the lower bound. Because several of our approaches build on RecSplit, we describe it in a bit more detail. RecSplit 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 4.4.

Splitting Trees. The splitting tree partitions the keys into smaller and smaller sets until we arrive at the leaves which have a small configurable size ℓ . It has a well-defined shape, depending only on the leaf size ℓ and the total number of keys in the bucket. At each inner node, RecSplit tries random hash functions to find one that distributes the keys to the child nodes according to the tree structure. The number of child nodes of an inner node is called *fanout*. The RecSplit paper gives a fanout such that the expected amount of work to find the splitting is roughly equal to the amount of work in all children combined. This results in fanouts of the two bottom-most levels of $\max\{2, \lceil 0.35\ell + 0.55 \rceil\}$ and $\max\{2, \lceil 0.21\ell + 0.9 \rceil\}$. In the terminology of the RecSplit paper, these levels are called *lower aggregation levels*. The levels above, also called *upper aggregation levels*, simply use a fanout of 2.

Bijections. The lowest level of the splitting tree is called *leaf level*. Each leaf, except for possibly the last, contains exactly ℓ 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 (see Section 1.1). RecSplit implements this construction in a very efficient way. The inner loop converts the hash value of each key to a bit by taking two to the power of it. It then sets the corresponding bit in a bit vector of length ℓ using a logical OR operation. RecSplit continues with all keys without checking for collisions. If the final bit vector has all its bits set to 1, it means that the hash function is a bijection. This avoids a conditional jump after each key.

Representation. Because the splitting trees have a well-defined shape, it is enough to store the hash function identifier at each node in preorder. RecSplit stores these numbers with Golomb-Rice codes (see Section 2.2). It stores all unary parts together and all fixed length parts together. The optimal Golomb parameter τ is different based on the layer in the tree, but it can be pre-calculated and stored in a lookup table. To combine all buckets into a single perfect hash function, RecSplit concatenates the encodings of all splitting trees from all buckets in a single bit vector. An additional sequence based on Elias-Fano coding stores both the prefix sums of the number of keys in each bucket and the positions where the encoding of each bucket starts.

Query. RecSplit can be queried by determining the bucket of a key and locating its encoding. The splitting tree in the bucket is traversed from the root to a leaf by applying the splitting hash function, which determines the child to descend into. A lookup table helps to



■ **Figure 4.4** Illustration of the overall RecSplit data structure. Within each bucket, it constructs a splitting tree. Circular nodes represent splittings, squares represent bijections.

determine how many Rice coded seeds need to be skipped to navigate to the correct subtree. During traversal, the number of keys stored in children to the left of the one descended into are accumulated. 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.

The combination of brute-force splittings and bijections is highly efficient from an information-theoretical point of view. Due to metadata and overheads during encoding, RecSplit loses only about 2–3 bits per node. Consequently, as the leaf size ℓ gets larger, RecSplit approaches optimal space [EGV20]. There are configurations that need only 1.56 bits per key in practice.

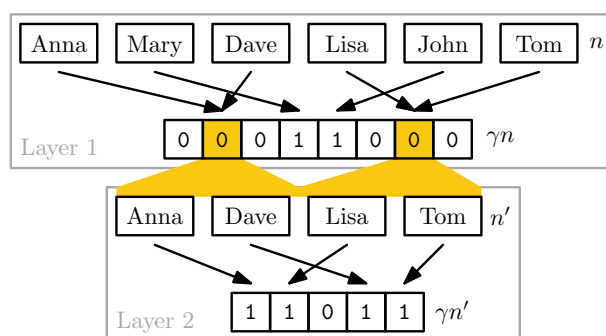
SIMDRecSplit. SIMDRecSplit [Bez+23] enhances RecSplit to use parallelism on multiple levels to improve the construction throughput. More precisely, it uses multi-threading, bit-parallelism and SIMD vectors. GPURecSplit [Bez+23] then implements the idea on the GPU. The paper proposes a new technique for 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 positions left by the first set are filled by the positions of the second set. The paper shows that each rotation essentially gives a new chance for a bijection, so it is a way to quickly evaluate additional hash function seeds in a bit-parallel way. In Chapter 5, we describe this approach in more detail.

ShockHash. ShockHash [LSW24b] can be seen as a combination of RecSplit and SicHash. It uses a variant of SicHash with two hash functions for each key and a fully loaded cuckoo hash table ($m = n$) as a base case within the RecSplit framework. Because constructing a fully loaded table usually does not succeed, ShockHash needs many retries. In graph terminology, ShockHash repeatedly generates an n -edge random graph where each key corresponds to one edge, connecting the two candidate positions of the key. The table can be filled if and only if the graph is a *pseudoforest* – a graph where each component contains as many edges as nodes. ShockHash then stores the choice between the two candidate positions of each key in a 1-bit retrieval data structure taking 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, while only $0.44n$ bits need to be determined by exponential time brute-force. Compared to brute-force, which needs e^n tries in expectation, ShockHash needs only $(e/2)^n$ tries, while still reaching the space lower bound. A key idea for making ShockHash practical is the introduction of a simple bit-parallel filter that checks whether all entries of the cuckoo hash table are hit by some key.

Bipartite ShockHash 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 practice. In Chapter 7, we describe ShockHash and bipartite ShockHash in detail.

4.5 Fingerprinting

The idea of perfect hashing through fingerprinting is to assign a small fingerprint to each input key using a hash function. The approach then resolves collisions between the fingerprints using



■ **Figure 4.5** Illustration of perfect hashing through fingerprinting.

recursion on the colliding keys. An advantage is the very simple and easily parallelizable construction. It is originally introduced by Chapman et al. [Cha+11] in the context of bioinformatics, but not described as a perfect hashing data structure of general interest. Müller et al. [Mü+14] then enhance and describe the idea from a data structure perspective.

FiPHa. Perfect hashing through fingerprinting [Mü+14] hashes the n keys to γn positions (fingerprints) using an ordinary hash function, where γ is a tuning parameter. A bit vector of length γn indicates positions 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 MPH 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 4.5 illustrates this idea. The most space-efficient choice $\gamma = 1$ leads to a space consumption of e bits per key [Mü+14], which is quite far from the lower bound of $\log e$ bits per key. However, the approach offers fast construction and queries. Perfect hashing through fingerprinting provides efficient queries when about 4 or more bits per key are available (using larger values of γ). FiPHa was developed in cooperation with SAP and the source code is not publicly available.

BBHash. The first publicly available implementation, BBHash [Lim+17] 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. Depending on the size of the bit vector and counter arrays, this causes up to three random memory accesses per key and level. 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 γ parameter that we already used in the description of FiPHa. Just like FiPHa, BBHash gets most efficient for about 4 bits per key.

FMPH. FMPH [Bel23] is a fast implementation of the idea in the Rust programming language. The approach still uses the construction algorithm with two passes, but its single-threaded construction is about twice as fast as BBHash. 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 the previous implementation. It offers decent performance starting with about 3 bits per key.

FiPS. FiPS – **F**ingerprint **P**erfect Hashing through **S**orting – is a third implementation of the approach, which we introduce in this dissertation. 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 original paper [Mü+14] but without a publicly available implementation. Using existing sorting libraries, the construction can be parallelized efficiently. A second advantage of this sorting-based approach is that it can be performed efficiently in external memory. To improve query performance, FiPS interleaves the select data structures and the bit vector. More precisely, FiPS stores blocks of size 512 bits. 480 bits store the bit vector indicating keys that did not collide. The remaining 32 bits store the number of 1-bits before the block. Therefore, the rank data structure has a space overhead of about 7%, which is more than the 4% achievable by modern rank data structures [Kur22]. However, it means that the rank operation does not cause additional cache faults. At query time, FiPS calculates the rank within the block using `popcount` instructions and adds it to the stored 32 bit number. There is a large design space. For example, we could encode the number of 1-bits within *sub-blocks* to increase query throughput. Alternatively, we could store the number of bits before each block as the difference to the expected value exploiting that the bits are uniformly distributed. We could also speed up the rank calculation within the block using SIMD instructions. However, we only present FiPS to illustrate the efficiency and simplicity of the fingerprinting approach and leave an exploration of the design space for future work.

FPHGO. FMPHGO [Bel23] 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. It selects the hash function that causes the least collisions in the fingerprint array. So, in contrast to CHD, collisions are allowed. FMPHGO invests additional space to store which hash function should be used in each bucket, but this reduces the recursion depth. In turn, it reduces the overall space consumption significantly. Compared to FMPH, FMPHGO reduces the storage space by up to 0.7 bits per key, while the query performance stays mostly the same.

4.6 Summary

In this chapter, we have seen several perfect hash function constructions. For approaches based on random (hyper)graphs, we illustrate the gradual process bringing the space consumption from $\mathcal{O}(n \log n)$ very close to the lower bound. For perfect hashing through bucket placement, we have seen significant improvements in the bucket assignment function. Finally, in fingerprinting, we have seen several implementations improving the construction performance.

Most of the perfect hash function constructions that are relevant in practice were presented in the last four years. This shows how active and growing the field of research is.

5 Minimal Perfect Hashing Through Tuned Brute-Force

Summary: *RecSplit [EGV20] is a very space-efficient practical minimal perfect hash function. It heavily relies on trying out hash functions using brute-force.*

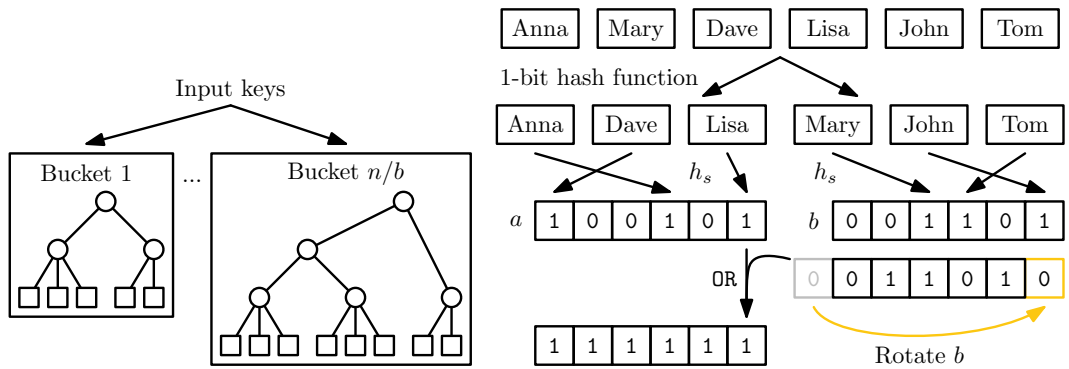
In this chapter, we introduce rotation fitting, a new technique that makes the search more efficient by drastically reducing the number of tried hash functions. Additionally, we greatly improve the construction time of RecSplit by harnessing parallelism on the level of bits, vectors, cores, and GPUs.

In combination, the resulting improvements yield speedups up to 239 on an 8-core CPU and up to 5438 using a GPU. The original single-threaded RecSplit implementation needs 1.5 hours to construct an MPH for 5 million keys with 1.56 bits per key. On the GPU, we achieve the same space consumption in just 5 seconds. Given that the speedups are larger than the increase in energy consumption, our implementation is more energy efficient than the original implementation.

Attribution: This chapter is based on “High Performance Construction of RecSplit Based Minimal Perfect Hash Functions” [Bez+23]. Large parts of this chapter are copied verbatim or with minor changes from that publication. The paper is in turn based on the Master’s thesis of Dominik Bez [Bez22], supervised by Peter Sanders, Florian Kurpicz, and the author of this dissertation. The author of this dissertation is the main author of the paper. He contributed to the implementation through improvements for configurations with small buckets, as well as constructing trees with the same shape together on the GPU. He also performed the experiments and wrote most of the manuscript. Florian Kurpicz contributed the analysis of the success probability of rotation fitting. All authors made significant contributions, to algorithm design, analysis, design & interpretation of the experiments, and the write-up.

RecSplit [EGV20] is a very space-efficient practical minimal perfect hash function. The general idea of RecSplit consists of two steps, *splittings* and *bijections*. As described in more detail in Section 4.4, RecSplit uses brute-force to determine hash function seeds that recursively split the key set in a specific way. This leads to a tree structure that we illustrate in Figure 5.1a. In the leaf nodes of the tree, RecSplit finds bijections by brute-force as well.

In this chapter, we explain several improvements inside the RecSplit framework. In Section 5.1, we give an algorithmic improvement for the bijection search that we call *rotation fitting*. The idea is to replace hash function evaluations with simple bit-parallel operations. Additionally, we parallelize the algorithm using SIMD instructions (see Section 5.2), multi-threading (see Section 5.3) and GPUs (see Section 5.5). We call our algorithms SIMDRecSplit and GPURecSplit. We conclude with an evaluation of the tuning parameters in Section 5.6. For a full comparison with competitors from the literature, we refer to Chapter 8.



(a) Illustration of the overall RecSplit data structure. Circular nodes of the trees represent splittings, squares represent bijections. (b) Rotation fitting. Just using hash function h_s would cause collisions. If we cyclically rotate set B , we can find a bijection.

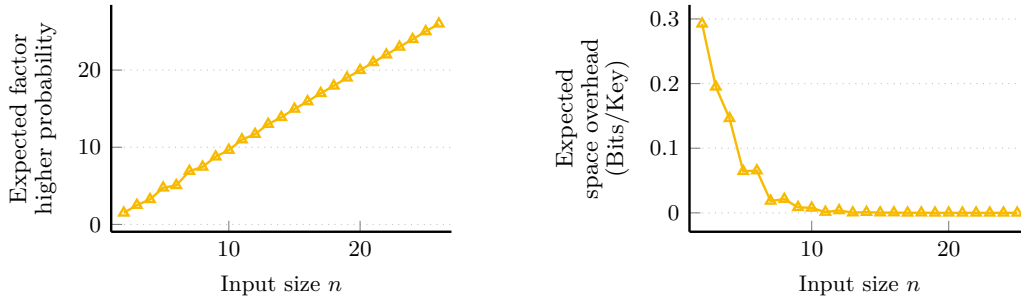
■ **Figure 5.1** Illustration of the overall RecSplit data structure and our enhancement to the leaf nodes, *rotation fitting*.

5.1 Rotation Fitting

In this section, we introduce a new method for searching for bijections in RecSplit's leaf nodes. As a reminder, given n keys in a leaf, we are looking for a way to quickly find a mapping of the keys to the first n integers without any collisions. Note that $n = \ell$ except for possibly the last leaf of each tree, where $n \leq \ell$. The original RecSplit implementation tries out hash functions using brute-force until one of them is a bijection.

Rotation fitting ensures that we need significantly fewer hash function evaluations. From the result of one evaluation, we derive additional candidates that are very fast to compute. Rotation fitting is particularly efficient when $n \leq w$, where w is the size of a machine word. We randomly distribute the keys into two sets A and B by using a 1-bit hash function. The 1-bit hash function is the same for all leaf nodes and does not ensure that A and B have the same size. Now we search for a hash function h that gives a bijection on the leaf. Like in the original RecSplit implementation, we calculate the hash value of all keys in A and set the respective bits in the word a to 1. The function h may be ruled out as a valid bijection by calculating the **popcount** of a . Analogously, the set B is mapped to the word b using the same hash function h . Let us now rotate (i.e., cyclically shift) the bits in b . If we can find a rotation value such that the 1-bits in b fit exactly onto the 0-bits in a , we have found a bijection on the leaf. More formally, this is the case if there is an $r \in [n]$, such that $a \text{ OR } \text{rot}_n^r(b)$ has the n least significant bits all set. The function $\text{rot}_n^r(b)$ rotates the n least significant bits in the word b by r positions. It can be implemented in a bit-parallel way using shifting and masking. Figure 5.1b illustrates the idea of rotation fitting. In Section 5.1, we show that for large n the probability of finding a bijection using rotation fitting is about n times higher than the probability when using RecSplit's brute-force approach. Also, the space overhead per key introduced by rotation fitting tends to 0 for large n (see Section 5.1).

To efficiently store r , we only try hash function identifiers which are multiples of n . This number plus r is stored for each leaf. We can restore r later by calculating modulo n and restore the hash function index by rounding down to the next multiple of n . At query time, a rotation corresponds to an addition modulo n to each key in the set B .



■ **Figure 5.2** Expected factor of higher probability to find a bijection using rotation fitting (left) and expected space overhead introduced by storing hash function index *and* rotation compared to RecSplit’s brute-force approach.

Lookup Tables. It is possible to avoid trying out all n rotations by using a lookup table t . For all possible values of a , this table contains a rotation parameter $t[a]$ such that $\text{rot}_n^{t[a]}(a)$ is minimal. If a value x can be rotated to get the value y , then $\text{rot}_n^{t[x]}(x) = \text{rot}_n^{t[y]}(y)$. Let $c = 2^n - 1$ be the word where the n least significant bits are set. The value $\hat{b} = b \oplus c$ is b with the n least significant bits flipped. Note that b can fill the holes in a if and only if \hat{b} can be rotated to match a . Thus, the necessary rotation of b can be calculated as $r = (t[\hat{b}] - t[a]) \bmod n$ using two table lookups. Rotation r is valid if $a \text{ OR } \text{rot}_n^r(b) = c$.

Because rotation is a very cheap operation, preliminary experiments show no improvement by lookup tables. Especially on GPUs, shared memory is a scarce resource and global memory is too slow. Our implementation therefore does not use lookup tables. Nonetheless, rotation fitting with lookup tables improves the asymptotic running time by a factor of n .

Success Probability. We now show that rotation fitting improves the construction time by a factor close to n , while having negligible space overhead. We refer to Figure 5.2 for numeric evaluations of the formula in the proof below.

► **Lemma 5.1.** *Let $|A| = \mathfrak{a}$, $|B| = \mathfrak{b}$, and $\Pr(R)$ be the probability of finding a bijection using rotation fitting. Furthermore, let $\Pr(B)$ denote the probability of finding a bijection using RecSplit’s brute-force strategy. If \mathfrak{a} and \mathfrak{b} are relatively prime, then $\Pr(R) \geq n \Pr(B)$. Otherwise, $\Pr(R) \rightarrow n \Pr(B)$ for $n \rightarrow \infty$.*

Proof. First, we consider the number of different injective functions under cyclic shifts, i.e., equivalence classes under rotation. We have a bit vector of length m with \mathfrak{b} set bits (and \mathfrak{a} unset bits). Then, the total number of equivalence classes under rotation is $\frac{1}{n} \sum_{d \text{ divides } \text{gcd}(\mathfrak{a}, \mathfrak{b})} \phi(d) \binom{n/d}{\mathfrak{b}/d}$, where gcd gives the greatest common divisor [Ada+21]. Let \mathcal{I} be the event that there is an r such that $a \text{ OR } \text{rot}_n^r(b)$ has the n least significant bits set. Then

$$\Pr(\mathcal{I}) \geq n \frac{1}{\sum_{d \text{ divides } \text{gcd}(\mathfrak{a}, \mathfrak{b})} \phi(d) \binom{n/d}{\mathfrak{b}/d}},$$

where $\phi(i) = |\{j \leq i : \text{gcd}(i, j) = 1\}|$ is Euler’s totient function. Now, we determine the probability $\Pr(R)$ using the events \mathcal{A} : $\text{popcount}(a) = \mathfrak{a}$ and \mathcal{B} : $\text{popcount}(b) = \mathfrak{b}$.

$$\begin{aligned} \Pr(R) &= \Pr(\mathcal{A}) \Pr(\mathcal{B}) \Pr(\mathcal{I}) \\ &\geq \frac{n!}{(n - \mathfrak{a})! n^{\mathfrak{a}}} \cdot \frac{n!}{(n - \mathfrak{b})! n^{\mathfrak{b}}} \cdot \Pr(\mathcal{I}) = \frac{n!}{n^n} \cdot \frac{n!}{\mathfrak{a}! \mathfrak{b}!} \cdot \Pr(\mathcal{I}) = \Pr(B) \cdot \frac{n!}{\mathfrak{a}! \mathfrak{b}!} \cdot \Pr(\mathcal{I}) \end{aligned}$$

$$\begin{aligned}
&\geq \Pr(B) \cdot \frac{n!}{\mathfrak{a}!\mathfrak{b}!} \cdot n \frac{1}{\sum_{d \text{ divides } \gcd(\mathfrak{a},\mathfrak{b})} \phi(d) \binom{n/d}{\mathfrak{b}/d}} \\
&= \Pr(B) \cdot n \cdot \frac{n!}{n! + (\mathfrak{a}!\mathfrak{b}!) \sum_{d \text{ divides } \gcd(\mathfrak{a},\mathfrak{b}), d \neq 1} \phi(d) \binom{n/d}{\mathfrak{b}/d}} \\
&= \Pr(B) \cdot n \cdot \frac{1}{1 + \sum_{d \text{ divides } \gcd(\mathfrak{a},\mathfrak{b}), d \neq 1} \phi(d) \frac{(n/d)!\mathfrak{a}!\mathfrak{b}!}{n!(\mathfrak{a}/d)!(\mathfrak{b}/d)!}} \\
&\sim \Pr(B) \cdot n \cdot \frac{1}{1 + \sum_{d \text{ divides } \gcd(\mathfrak{a},\mathfrak{b}), d \neq 1} \phi(d) \sqrt{d} \frac{\mathfrak{a}^{\mathfrak{a}-\mathfrak{a}/d} \mathfrak{b}^{\mathfrak{b}-\mathfrak{b}/d}}{n^{n-n/d}}} \\
&\rightarrow \Pr(B) \cdot n \text{ for } n \rightarrow \infty
\end{aligned}$$

Note that if \mathfrak{a} and \mathfrak{b} are relatively prime the sum is zero, as $\gcd(\mathfrak{a}, \mathfrak{b}) = 1$. ◀

5.2 SIMD Parallelization

For the SIMD parallelization, we focus on the description of bijections and splittings, which (in most configurations) take most time of the construction. While we additionally accelerate the construction of the Elias-Fano data structure, the ideas are less interesting algorithmically.

SIMD Instructions. It is common, especially in perfect hashing, that the same operation needs to be executed on different data. This can be achieved with a simple loop, which means that the corresponding instructions must be decoded by the hardware several times. This can be improved by using *Single Instruction, Multiple Data* (SIMD) [Fly72]. A single instruction is used to apply the same operation on a *vector* of several elements. We refer to a single element within a SIMD vector as a *lane*. For example, a vector may contain 16 lanes with 32 bits each, i.e., the vector contains 512 bits overall. The exact set of operations depends on the concrete implementation of the SIMD model. On many Intel and AMD processors, SIMD operations are available through the Advanced Vector Extensions (AVX) [Int11]. AVX-512 [Int13] extends these operations to 512-bit vectors and is divided into many smaller subsets that offer additional operations. A subset that is useful for our implementation is AVX512VPOPCNTDQ, which provides `popcount` on 512-bit vectors with lanes of size 32 and 64 bits. The rot_k^i function that cyclically shifts bits can be implemented in parallel using SIMD as well.

The main idea of our SIMD parallelization is to try multiple hash function seeds simultaneously. Depending on the operation, we use SIMD lanes with a width of either 32 bits or 64 bits. In the following, we explain bijections and splittings separately.

Bijections. For the bijections, each SIMD lane is responsible for trying one hash function. For this, we load consecutive hash function identifiers and the same input key to each lane of a SIMD vector, and evaluate the hash function. The resulting hash value in each lane is converted to a single bit by taking two to the power of it. After calculating the logical OR of these bits for all keys in the set, we check for a bijection by comparing each lane with a constant that has all n lower bits set to 1. For rotation fitting, remember that the number we store as a seed is the hash function identification plus the rotation. This number should be as small as possible to avoid wasting space, so caution must be taken when trying out the rotations. If one lane finds a bijection, it might be possible that a higher rotation leads to a bijection on a lane with a smaller hash function index. Because this gives a smaller overall number to store, we always try all rotation values, even if a bijection is found.

Splittings. For the splittings, the original implementation uses small arrays of counters. Each counter contains the number of keys hashed to the respective split section. Instead, we use two different methods. For the *upper* aggregation levels with fanout 2, we use a single counter for the number of keys hashed to the left child. The number of keys in the right child can then be determined by subtraction. For all practical leaf sizes ($\ell \leq 24$), each counter of a valid *lower* level splitting fits into a single byte. Because an overflowing counter for one child would then just add 1 to the next counter, such overflows cannot make an invalid splitting look valid. When a seed for a valid splitting is found, we need to redistribute the keys. Here we use SIMD to apply the same hash function to several keys at once, and store the results in an array. We then redistribute the keys without SIMD parallelism.

5.3 Multi-Threaded Parallelization

The original RecSplit implementation only uses a single thread. This leaves a lot of processing power unused since most modern processors contain multiple processing cores. As stated in its paper [EGV20], parallelizing RecSplit is fairly easy because the buckets are completely independent of each other. First, we sort the input keys by their bucket index in parallel, and then determine the bucket borders. We then start several threads and assign a consecutive portion of the buckets to each thread. Because the number of buckets is large and the input keys are hashed to buckets uniformly, the load of all threads is reasonably balanced.

After a splitting or bijection is found, it must be stored in the Golomb-Rice coded sequence. To avoid synchronization, each thread uses its own local sequence and treats its input as if it was the complete input. This means it also stores the pointers to the start of each bucket encoding locally. After all threads are done, we sequentially concatenate the Golomb-Rice sequences and build the combined Elias-Fano data structure holding the prefix sum of bucket sizes and pointers to the bucket encodings.

5.4 GPUs

Graphics Processing Units (GPUs) are specialized processors initially designed for computer graphics applications. Over the last decades, GPUs evolved to general purpose processors for highly parallelizable tasks. We now describe the hardware and programming interface of GPUs. To provide a grasp of the dimensions of a current GPU, we give metrics of the NVIDIA RTX 3090 [Nvi20], which is also used for our experiments (see Section 5.6).

Compute Hardware. A GPU consists of several streaming multiprocessors (SMs) (RTX 3090: 82). Each SM contains many arithmetic logic units (ALUs) to perform computations (RTX 3090: 64 integer ALUs). Several threads (RTX 3090: 32) operate in *lock-step*, i.e., they execute the same instruction at the same time. Such a bundle of threads is called *warp*. Threads are masked out for instructions they should not execute. This means that in loops, each thread in a warp has to iterate as many times as the thread with the largest number of iterations. To hide latencies, e.g., for memory access, each SM is oversubscribed with more threads than ALUs, and the GPU schedules the threads efficiently. Multiple warps of threads form a *thread block*. Thread blocks are guaranteed to reside on the same SM, which enables them to cooperate.

Memory. The *global memory* is the largest and slowest memory on the GPU (RTX 3090: 24 GB). When multiple threads of a warp access the memory simultaneously, the hardware

serves the requests with as few memory transactions as possible. *Shared memory* is a fast memory placed on each SM. It is shared between the threads of the same thread block. On the RTX 3090, shared memory and L1 cache are allocated on the same memory areas. The data in shared memory is partitioned into 32 memory banks, and the i -th 32-bit word is stored in bank $i \bmod 32$. When multiple threads simultaneously access different words within the same bank, the access operations have to be serialized.

CUDA. An efficient way to develop applications on NVIDIA GPUs is CUDA [Nvi22]. Functions to be executed on the GPU are called *kernels*. Each kernel is executed on a *grid* of *thread blocks*. The grid size and the number of threads per block can be selected by the user. The user can create several *streams*. Kernels and data transfers launched into the same stream are executed in order, but operations in different streams can arbitrarily overlap.

5.5 GPU Parallelization

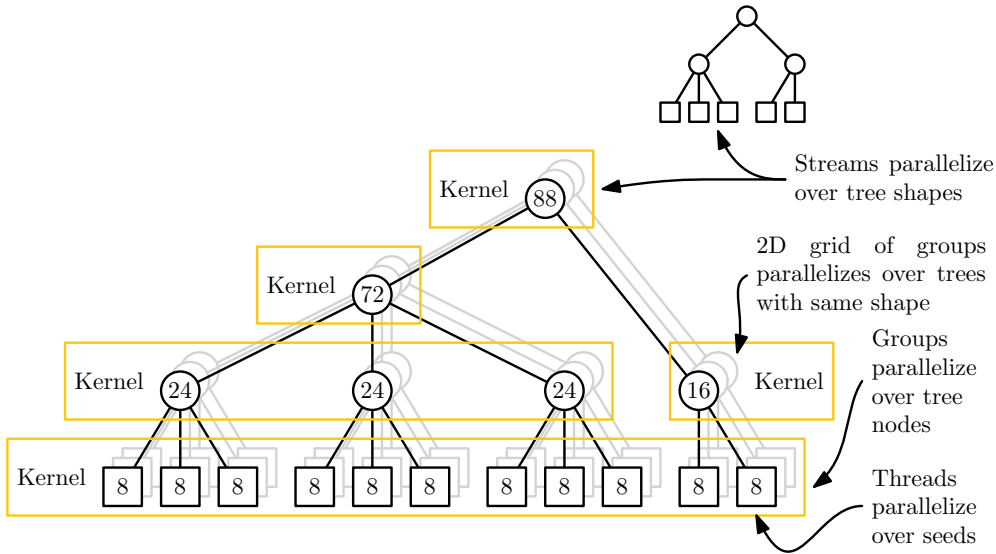
In the GPU implementation, we first partition the keys to their buckets and partition the buckets by their respective size. We then use the GPU to determine the splittings and bijections within the buckets. Buckets with the same size have splitting trees with the same shape and can therefore be handled within the same set of kernel calls. While this means that we need to perform additional work on the CPU to partition buckets by size, it significantly reduces constant overheads due to GPU kernel calls. Especially for configurations with small expected bucket sizes b , we get a large number of buckets but only a few distinct sizes. This is where the optimization shines most. Preliminary experiments show a speedup of about 2 compared to constructing the buckets individually.

Using CUDA's streams, we additionally construct different bucket shapes concurrently. This is most interesting for large b where sometimes the number of buckets having a specific shape is small. The streams enable better utilization of the GPU in that case. For an overview of the GPU construction, see Figure 5.3.

Bijections. All leaf nodes¹ of all trees with the same shape are constructed with a single kernel call. For each leaf node, we start one block of threads. First, the threads in each block cooperate to load all keys relevant for that leaf node into the shared memory. Similar to the SIMD implementation, where each *lane* tried a different hash function, now each GPU *thread* tries a different hash function. After each hash function, the threads synchronize, and check if a bijection was found. If it was, we store the hash function index into global memory.

Splittings. Like for the bijections, each splitting is handled by a thread block. The threads cooperate to load the keys into the shared memory and then each thread tries a different hash function index. For the two lowest aggregation levels, the thread blocks of all nodes in that level are started together using one kernel call (see Figure 5.3). Note that on these levels, the size of a node and the starting seed is constant. Therefore, the levels are very homogeneous. Conversely, the higher levels with fanout $s = 2$ are more heterogeneous. In particular, the number of keys on a specific level may be different for different nodes on the same level. Therefore, we launch individual kernels for each of those splittings, which contain the thread block for all trees with the same shape. We use multiplication and shifts to

¹ All leaf nodes except possibly the last of each tree, which might have fewer keys.



■ **Figure 5.3** Illustration of how all equally-shaped splitting trees are handled together on the GPU. Here we use a leaf size of $\ell = 8$ with $n = 88$ input keys in the tree.

increment the counters of how many keys ended up in each lane. An alternative variant that stores counters in shared memory is slower in preliminary experiments, even when padding the counters to reduce the probability of bank conflicts. After a valid splitting is found, the threads in a block cooperate to reorder the keys in that node accordingly.

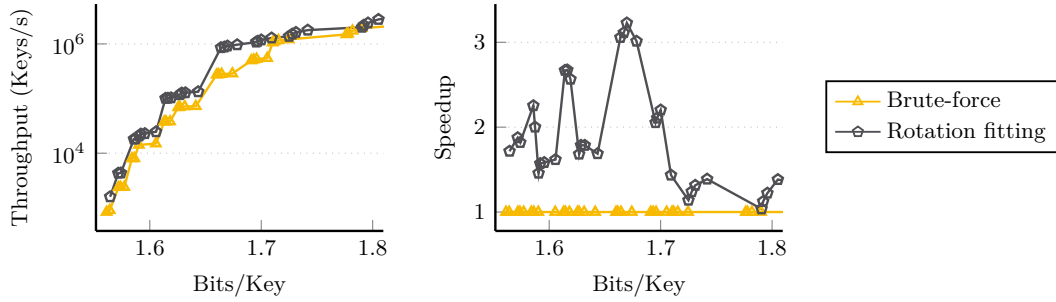
Assembly. Because the kernels are launched per level, the results are stored in BFS order. For the final data structure, we need to store them in preorder. The CPU unpacks the resulting seeds recursively and writes them to an encoded sequence.

5.6 Internal Experiments

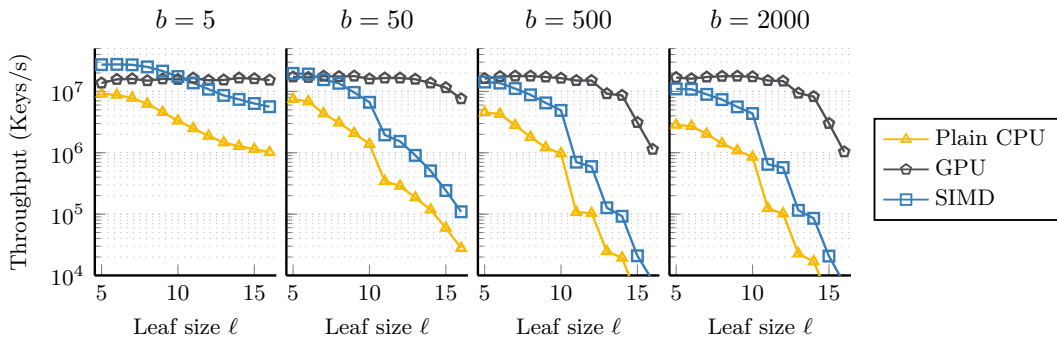
In this section, we give an internal evaluation of our implementation. It compares tuning parameters, and the GPU implementation with the CPU implementation. For a comparison with competitors, and an explanation of the experimental setup, we refer to Chapter 8. For now, it is enough to know that we use an 8-core (16 hardware threads) consumer machine equipped with an NVIDIA RTX 3090 GPU. The code and scripts needed to reproduce our experiments are available on GitHub under the General Public License [Leh23a].

The SIMD implementation only supports x86 CPUs and is optimized towards AVX-512 using the Vector Class Library [Fog13]. The GPU implementation uses CUDA 11. As a reminder, only the *construction* is using SIMD, multi-threading, and/or the GPU. The query implementation is identical for the SIMD and GPU implementation and almost equal to the original implementation [EGV20]. We therefore do not compare the query performance of SIMD and GPU implementation.

While the original implementation [EGV20] uses `std::sort` to partition keys into buckets, we use `IPS2Ra` [Axt+22]. For the less space-efficient configurations ($\ell < 5, b < 100$), constructing the buckets is fast, so significant time is spent on sorting. In that case, `IPS2Ra` both speeds up the sequential case and also enables sorting in parallel. For more space-efficient configurations ($\ell > 8$), the partitioning step needs less than 1% of the total construction time, both in the



■ **Figure 5.4** Pareto front over the construction throughput of different variants of searching for bijections in the leaves. Single-threaded, non-vectorized measurements with $n = 5$ million keys. The plot on the right gives speedups relative to brute-force.³



■ **Figure 5.5** Construction throughput with different hardware architectures based on different input parameters. $n = 5$ million keys, 1 CPU thread.

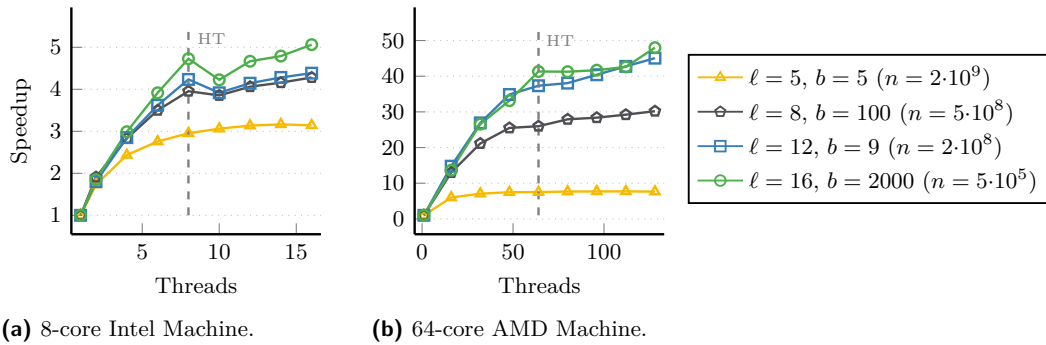
parallel and the sequential case. In this section, we compare against a slight adaption of the original implementation, using IPS²Ra and supporting parallel construction.

Rotation Fitting. In order to compare rotation fitting with the brute-force variant, we give a Pareto front² of space usage versus construction time in Figure 5.4. The construction time refers to the entire MPHf construction, including the time used for splittings. Rotation fitting is consistently faster, making the entire MPHf construction up to 3 times faster. The space overhead of rotation fitting becomes negligible for moderately large ℓ (see Section 5.1). Unless otherwise noted, all following experiments use rotation fitting.

Dependence on Input Parameters. In Figure 5.5, we plot the throughput of the SIMD, GPU and non-vectorized versions for different leaf sizes ℓ and bucket sizes b . For better comparability with the original paper [EGV20], we include a wide range of configurations, even ones that are not very competitive. The SIMD version is consistently up to 4.5 times

² A configuration is on the Pareto front if it is not dominated by any other configuration with respect to both construction time and space consumption.

³ Note that giving speedups is non-trivial here because there might not be a configuration that achieves the same space usage that we could compare with. We therefore calculate the speedup relative to an interpolation of the next larger and next smaller data points. This is reasonable since RecSplit instances can be interpolated as well by hashing a certain fraction of keys into data structures with different configurations.



■ **Figure 5.6** Construction self-speedup by number of threads used, for different configurations. The number of input keys n is selected such that construction takes a similar amount of time on all configurations. Configurations are the examples that are highlighted in the RecSplit paper [EGV20].

faster than the non-vectorized version and shows the same scaling behavior in ℓ . The plot indicates that there is no configuration where one would prefer the non-vectorized version. While the GPU offers significant speedups for space-efficient configurations, it helps less for the space inefficient configurations. A reason for this is data transfers to and from the GPU.

Multi-Threading. Table 5.1 shows that the parallel construction is up to 5 times faster on an 8-core machine than the single-threaded version. Figure 5.6 shows how the SIMD version scales when selecting a different number of CPU threads. The configurations are adopted from the RecSplit paper [EGV20]. On the Intel machine, we can see that the most space-efficient configuration scales best, closely followed by the other configurations. Only a variant with tiny buckets ($b = 5$) does not scale as well. In this case, the entire construction is dominated by partitioning the keys to buckets. Given that we already use the highly optimized sorter IPS²Ra [Axt+22] and that this is a rather unusual RecSplit configuration with a lot of space overhead, having non-optimal speedups here is acceptable. On an additional AMD machine with 64 cores (128 hardware threads), the difference between the different configurations is slightly more pronounced.

Overall Speedup. Our rotation fitting technique leads to a speedup of up to 3 (see Figure 5.4) and SIMD parallelism improves the construction speed by up to a factor of 4.5 (see Figure 5.5). Multi-threading for highly space-efficient configurations shows a speedup of close to 5. Table 5.1 shows the overall improvement of our implementation on CPU and GPU when compared to the original RecSplit implementation. The original RecSplit paper says that MPHf construction at 1.56 bits per key is possible. This configuration with 5 million keys takes about 1.5 hours using the original implementation. Our SIMD implementation achieves the same space usage in just 2 minutes on the CPU and 5 seconds on the GPU. Investing about 40 minutes of GPU time, our implementation achieves a space usage of only 1.495 bits per key. This is about 40% closer to the lower bound [BBD09] of 1.44 bits, and simultaneously more than twice as fast as the original implementation.

Energy Consumption. Of course, directly comparing CPU and GPU implementations is unfair. A sensible metric to compare them is the energy consumption, which can be a major cost factor. Additionally, the energy consumption is not influenced by the market prices of GPUs. Table 5.2 gives energy consumption measurements for different configurations and

hardware architectures. The energy consumption is homogeneous throughout most of the execution time, except for a short ramp-up in the beginning. We do not count the ramp-up to the energy consumption. Measurements are performed using a Voltcraft 870 Multimeter.

Even though SIMD instructions need slightly more power, the total energy consumption of constructing one MPHf is lower. The GPU, even though it needs significantly more power, is so much faster that the resulting energy usage is about 1000 times lower than the original single-threaded CPU implementation. For basic RecSplit, the AMD machine needs about 1.5 times more time than the Intel machine. This can be readily explained by a lower clock frequency. This performance gap grows to a factor 4.6 for sequential SIMDRecSplit. The likely main reason is that the AMD machines lack the AVX-512 vector units of the Intel machine. Still, since both processors have two 256-bit AVX2 units per core, it seems that better performance might be achievable with careful tuning for the AMD architecture. On the contrary, the AMD machine shows good scalability so that the energy consumption when using the entire machine is only a factor 1.3 larger than on the Intel machine – despite the fact that our implementation was tuned for the Intel architecture.

5.7 Summary

We have shown that by harnessing parallelism at all available levels – bits, vectors, cores, and GPUs – one can dramatically accelerate the construction of highly space-efficient minimal perfect hash functions (MPHF) using the brute-force RecSplit approach [EGV20]. This leads to speedups of up to 239 on SIMD and 5438 on the GPU and also dramatically reduces the energy consumption. Our new technique *rotation fitting* reduces the work needed per tried hash function while adding a tiny bit of space requirement. We refer to Chapter 8 for a comparison with other approaches from the literature.

■ **Table 5.1** Construction time of the GPU implementation compared to our multi-threaded adaption of the original RecSplit implementation. $n = 5$ million keys (strong scaling). Construction times are given in $\mu\text{s}/\text{key}$. We do not report speedups for $\ell = 24$ because the CPU baseline takes too long for this configuration.

Configuration	Method	Bijections	Threads	B/Key	Constr.	Speedup
$\ell = 16, b = 2000$	RecSplit [EGV20]	Brute-force	1	1.560	1175.4	1
	RecSplit	Brute-force	16	1.560	206.5	5
	SIMDRecSplit	Rotation fitting	1	1.560	138.0	8
	SIMDRecSplit	Rotation fitting	16	1.560	27.9	42
	GPURecSplit	Brute-force	GPU	1.560	1.8	655
	GPURecSplit	Rotation fitting	GPU	1.560	1.0	1173
$\ell = 18, b = 50$	RecSplit [EGV20]	Brute-force	1	1.707	2942.9	1
	RecSplit	Brute-force	16	1.713	504.0	5
	SIMDRecSplit	Rotation fitting	1	1.709	58.3	50
	SIMDRecSplit	Rotation fitting	16	1.708	12.3	239
	GPURecSplit	Brute-force	GPU	1.708	5.2	564
	GPURecSplit	Rotation fitting	GPU	1.709	0.5	5438
$\ell = 24, b = 2000$	GPURecSplit	Brute-force	GPU	1.496	2300.9	—
	GPURecSplit	Rotation fitting	GPU	1.496	467.9	—

■ **Table 5.2** Energy consumption with $\ell = 18, b = 50$ and $n = 5$ million keys. Energy consumption is both given as difference to the idle power, as well as total energy consumption of the whole system. For CPU-only measurements of the 8-core Intel machine, we dismount the GPU.

Machine	Method	Threads	Constr. Seconds	Total system		Δ to idle	
				Power Watt	Energy Joule	Power Watt	Energy Joule
8-core Intel	RecSplit [EGV20]	1	14 714.5	78	1 147 731	37	544 436
	SIMDRecSplit	1	291.5	87	25 360	46	13 409
	SIMDRecSplit	16	61.5	104	6 396	63	3 874
	GPURecSplit	—	2.5	457	1 142	380	950
64-core AMD	RecSplit [EGV20]	1	21 620.8	223	4 821 438	91	1 967 492
	SIMDRecSplit	1	1 328.7	224	297 629	92	122 240
	SIMDRecSplit	128	23.6	364	8 590	232	5 475

6 Small Irregular Cuckoo Tables for Perfect Hashing

Summary: *In this chapter, we present SicHash – Small irregular cuckoo tables for perfect hashing. At its core, SicHash uses a known technique: it places keys in a cuckoo hash table and then stores the final hash function choice of each key in a retrieval data structure. We combine the idea with irregular cuckoo hashing, where different keys can have a different number of hash functions. Additionally, we use many small tables that we overload beyond their asymptotic maximum load factor. The most space-efficient competitors often use brute-force methods to determine the PHFs. SicHash provides a more direct construction algorithm that only rarely needs to re-compute parts. Our implementation improves the state of the art in terms of space usage versus construction time for a wide range of configurations.*

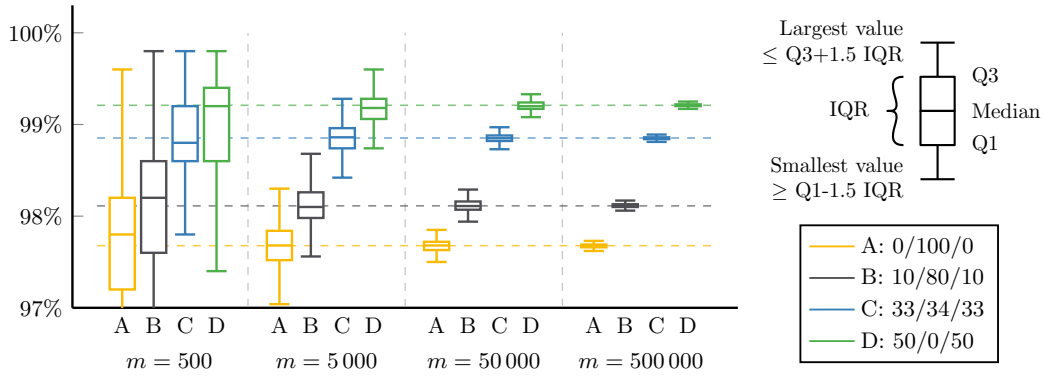
Attribution: This chapter is based on “SicHash – Small Irregular Cuckoo Tables for Perfect Hashing” [LSW23b]. Large parts of this chapter are copied verbatim or with minor changes from that publication. We do not include the analysis of the load thresholds by Stefan Walzer. The author of this dissertation is the main author of the paper. He implemented the algorithm, performed the experiments and wrote most of the manuscript. All authors made significant contributions, to algorithm design, analysis, design & interpretation of the experiments, and the write-up. The authors would like to thank Martin Dietzfelbinger for early discussions leading to the paper.

In perfect hashing through retrieval (see Section 2.6), each key has a small number of candidate hash values, determined by different hash functions. Using cuckoo hashing, one can determine an assignment from keys to one of their candidates such all the hash values are different. The choices can be stored efficiently in a retrieval data structure (see Section 2.4). SicHash – Small irregular cuckoo tables for perfect hashing – extends this approach through *overloading* and *irregular cuckoo hashing*.

In Section 6.1, we describe our new technique *overloading*. The idea is to fill the cuckoo hash tables beyond their asymptotic load threshold. This is possible when making the tables small and retrying construction with a new seed if insertion fails. Overloading mainly relies on the variance in the achievable load factor. Additionally, we find experimentally that the point at which insertions start to fail converges to the load threshold from *above* as n grows.

In irregular cuckoo hashing, different keys get a different number of candidate positions based on a hash function. Irregular cuckoo hashing was previously considered for reducing search time in hash tables. For that application, it was of little help apart from allowing to interpolate between two integer numbers of hash functions. In contrast, for our application to reduce space in perfect hashing, it is helpful even when the average number of hash functions already is an integer. We describe how we apply this to SicHash in Section 6.2.

In Section 6.3, we describe possible enhancements of the approach and give an analysis in Section 6.4. We conclude the chapter by evaluating tuning parameters in Section 6.5. We give a full evaluation comparing SicHash to competitors from the literature in Chapter 8.



■ **Figure 6.1** Achieved load factors when running different irregular cuckoo hashing configurations, which all need the same storage space (2 bits). The labels describe the percentages of keys with 2/4/8 choices, having a space usage of 1/2/3 bits, respectively. The configuration 0/100/0 refers to ordinary 4-ary cuckoo hashing. Horizontal lines indicate numerically calculated load thresholds [LSW23b, Section 7].

6.1 Overloading

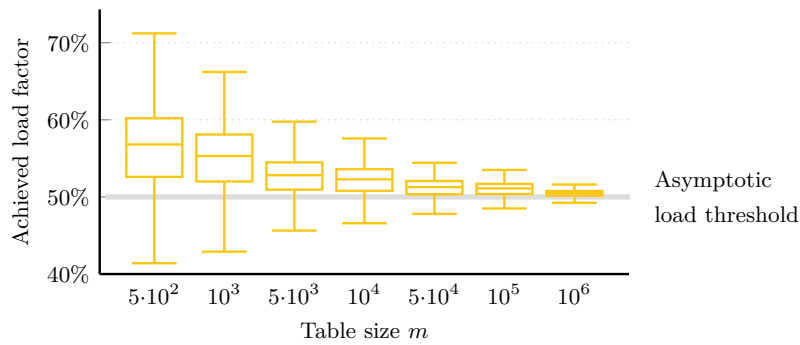
In cuckoo hashing, the load threshold of a table is a widely studied subject (see Section 2.5). For example, it is well known that a table with $d = 2$ hash functions has a *load threshold* of 50%. For $n \rightarrow \infty$, the probability of successful table construction with load factor $> 50\%$ approaches 0, while it approaches 1 for load factors $< 50\%$. Let us now look at a very small table of size $m = 3$ storing $n = 2$ keys. This table has a load factor of $\approx 66\%$, which is more than the load threshold. Still, the probability of successful construction, i.e., not all four hash function values being the same, is $1 - 3 \cdot (1/3)^4 \approx 88\%$. This shows that the load factors can be considerably higher than the asymptotic limits when using small tables. We call a table that contains more keys than the asymptotic limit *overloaded*.

To experimentally illustrate the achievable load factors, we incrementally construct cuckoo hash tables and record the load factors at which the insertion finally fails. Figure 6.1 gives a box plot of the achieved load factors. It shows three fundamental observations that we use in SicHash to increase the load factor while decreasing the amount of memory needed.

(1) Variance. Unsurprisingly, small tables have a higher variance in their achieved load factors. Therefore, improved load by the standard deviation is possible by just retrying a constant number of times in expectation.

(2) Median. For some configurations, small tables not only enable higher load factors because of the variance, but also because their median is higher than the load threshold. The effect is even more pronounced for ordinary binary cuckoo hashing, where a table with $m = 500$ has a median load factor of 56% (see Figure 6.2). This is significantly more than the asymptotic load threshold of 50%. A similar observation can be found in perfect hashing: The space lower bound to store an MPH with small n is significantly lower than the asymptotic value for $n \rightarrow \infty$ [WH20].

(3) Space Usage. A metric for the lookup efficiency in irregular cuckoo hash tables is the average number of hash functions. For any desired average number of hash functions $d' \in \mathbb{R}$, the best load thresholds are given by a combination of $\lfloor d' \rfloor$ and $\lceil d' \rceil$ hash functions [Die+10].



■ **Figure 6.2** Achieved load factors when constructing binary cuckoo hash tables of different sizes. The median of small tables is higher than the asymptotic load threshold (50%).

This picture changes fundamentally in the context of PHFs because the choice of hash functions is stored in binary coding. While, for example, an irregular cuckoo hash table with 50% 2-choice and 50% 4-choice has 3 choices on average, the storage space of the corresponding PHF is only $0.5 \log(2) + 0.5 \log(4) = 1.5 < \log(3)$. The configurations in Figure 6.1 all need the same storage space, but the load threshold increases the farther we are from the optimal configuration derived in Ref. [Die+10].

Conclusion. Smaller cuckoo hash tables enable higher load factors than larger tables. Equivalently, by making the tables smaller, we can achieve the same load factor using a hash function mixture that needs less space. Even though we have to store a seed because the variance in the load factors is higher, we can use the described effects to save overall space.

6.2 SicHash Perfect Hash Functions

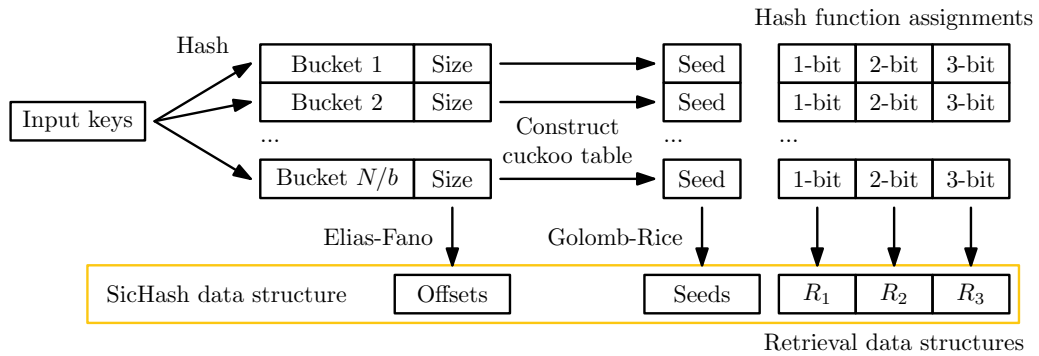
We are now ready to introduce the main result of this chapter: SicHash perfect hash functions. SicHash combines irregular cuckoo hashing with overloading. Additionally, it builds on recent advances in fast and space-efficient retrieval.

6.2.1 Construction

Building a SicHash function consists of the main steps partitioning, cuckoo hashing, storing bucket metadata, and constructing the retrieval data structures. Figure 6.3 gives an overview.

Partitioning. First, we hash the keys to a number of buckets that all have the same expected size b , for example $b = 5000$. The rather small size enables overloading and also keeps the storage space during construction small enough that the whole table fits into the cache.

Cuckoo Hashing. Within each bucket, we generate an irregular cuckoo hash table, using the same load factor as the overall perfect hash function. The number of cells in each of the small cuckoo hash tables is determined by the number of keys hashed to it, so the small tables have different sizes. However, the probability of a successful construction is similar for all of them. To determine how many hash functions (and therefore candidate cells) should be used for each key, we hash each key to a *class*. A certain percentage p_1 of keys is placed with $d = 2$ choices, a percentage p_2 with $d = 4$ choices, and the remaining keys with $d = 8$ choices.



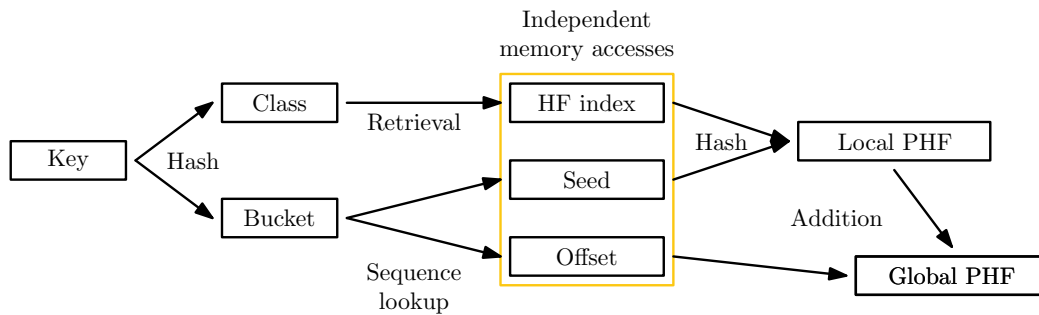
■ **Figure 6.3** Data flow during construction. Keys are hashed to buckets of expected equal size. Within each bucket, a cuckoo hash table is constructed. The resulting hash function assignments from all small hash tables are stored together in three large retrieval data structures.

To insert keys into the hash tables, we use *rattle kicking* [Kus16] instead of the classical random walk. Rattle kicking maintains a counter for each key, indicating how often it was moved to a new cell. A key is then only evicted from its cell when its rattle counter is lower than the counter of the key to be inserted into the same cell. The next hash function index to use for inserting is the rattle counter modulo d . With rattle kicking, we can avoid the cost of random number generation and empirically need a lower number of steps during the insertion. For normal cuckoo hash tables, storing the counter would decrease the space efficiency. In our case, we only store the hash table temporarily, and also need the hash function index to construct the PHF later. This makes SicHash an attractive application for rattle kicking. Constructing a bucket may fail, in particular, when we configure a high degree of overloading. In this case, we retry construction while incrementing a seed value determining the used hash functions.

Storing Bucket Metadata. The result is a number of small hash tables, each with a seed leading to successful construction. In order to determine a global PHF, we need to offset each small table. We can do that by storing the exclusive prefix sum of table sizes. When trying out seeds, we can simply count up, starting with 0. One could encode the prefix sum using Elias-Fano coding [Eli74; Fan71] (see Section 2.3) and the seed using Golomb-Rice coding [Gol66; Ric79] (see Section 2.2). However, in practice, storing the two sequences in arrays offers better query time and negligible space overhead.

Retrieval. Now we only need to store the assignment from keys to cells within the small hash tables by storing which of the hash functions finally placed each key. Because the hash tables are irregular, we get indices of 1, 2, and 3 bits. While small hash tables enable overloading, retrieval data structures in contrast profit from handling many keys and can achieve overheads as low as 1% [Dil+22]. We therefore build 3 large retrieval data structures that hold the 1, 2, and 3-bit values from *all* the small hash tables. The space usage of the final PHF is dominated by the retrieval data structures.

Parameters. A SicHash PHF has three main tuning parameters: The load factor $\alpha = n/m$ to try construction for, and the class sizes for irregular cuckoo hashing, p_1 and p_2 . Ignoring overloading, it is then possible to numerically determine a configuration that maximizes the



■ **Figure 6.4** Data flow during a query. From the class of a key, we get the retrieval data structure to query. From the bucket index, we get the bucket’s seed and offset. Combining this, we get a global hash function value.

load factor (see [LSW23b, Section 7]). Calculating efficient configurations *with* overloading remains an open problem. When a specific space budget of β bits per key for the retrieval data structures is given, we get $p_2 = 3 - \beta - 2p_1$. The only remaining parameter p_1 can then be selected from the interval $[\max(0, 2 - \beta), (3 - \beta)/2]$.

6.2.2 Queries

A query for a key first hashes it to find its class, e.g., its number of candidate cells in the small hash table. This determines which retrieval data structure needs to be queried for the hash function index. Additionally, the key is hashed to find its bucket, and therefore the seed and offset. The value of the PHF is then given by hashing the key with the retrieved hash function index and the bucket’s seed, and by adding the bucket’s offset. Figure 6.4 gives an illustration for the data flow during a query, showing that all three memory access operations are independent of each other, so they can be prefetched or performed in parallel.

6.3 Enhancements

SicHash lends itself to numerous enhancements, which we outline below.

Minimal Perfect Hashing. SicHash can be converted to an MPHf by applying the same technique as in PTHash [PT21]. The idea is to re-map keys with hash function values $> n$ to smaller values by using an Elias-Fano coded sequence of size $m - n$. We refer to Section 2.8 for details. Note that the number of lower bits of the Elias-Fano coded sequence only depends on the load factor that was used before re-mapping. In practice, we can therefore use a compile-time parameter for faster bit operations in the Elias-Fano sequence.

Parallel Construction. Perfect hash functions can always be parallelized trivially by introducing a new layer on top of the data structure. SicHash can be parallelized more directly and without effect on the query speed. The small cuckoo hash tables of each bucket can be constructed independently. Retrieval data structures can also be computed in parallel at some small space overhead linear in the number of processors and without query overhead [Dil+22].

External Memory Construction. SicHash can be adapted to very large inputs: First use external sorting to partition the keys into buckets. Then, for each bucket, construct the cuckoo hash table – outputting sequences of seeds, offsets, and key-value pairs for the retrieval

data structures. The latter can be fed into an external memory construction of the retrieval data structures [Dil+22].¹

6.4 Analysis

In this section, we discuss the space consumption, construction time, and query time of the SicHash data structure. We omit the proof outline showing load thresholds [LSW23b, Section 7], which is far from self-contained and does not give any intuition.

Let N be the number of input keys before bucketing and let M be the overall output range of all buckets combined. We look at the case of constant expected bucket size and a bucket load factor $\alpha = N/M$ that ensures constant success probability of cuckoo hash table construction. We construct the tables using BFS. For the analysis, we use space-efficient encoding of offsets using Elias-Fano coding and Golomb-Rice coding of seeds. We also assume that we use a retrieval data structure that needs space $(1 + o(1))$ times the lower bound, can be constructed in linear time, and supports queries in constant time [Dil+22]. Afterwards, we informally discuss what changes for the simple implementation used in our experiments.

► **Theorem 6.1.** *A SicHash data structure can be queried in constant time.*

Proof. A query evaluates a constant number of hash functions which takes constant time and performs one access to a retrieval data structure, which takes constant time as assumed above. Additionally, it decodes a number each in an Elias-Fano coded and a Golomb-Rice coded sequence. Decoding the numbers boils down to constant time $select_1$ -operations in a bit vector (see Section 2.3). ◀

► **Theorem 6.2.** *The expected space consumption of a non-minimal SicHash data structure is $N(3 - 2p_1 - p_2 + o(1) + \mathcal{O}(1/b))$ bits. Minimal perfect hashing needs an additional $N(\frac{1}{\alpha} - 1)(2 + \log \frac{\alpha}{1-\alpha} + o(1))$ bits.*

Proof. With average bucket size b , the Elias-Fano data structure takes $(2 + \log \frac{Mb}{N} + o(1)) \frac{N}{b}$ bits, including the select data structure. For constant success probability of construction, the seed has a geometric distribution and constant expected length, i.e., the expected space consumption is $\mathcal{O}(N/b)$ bits. The 1-, 2-, and 3-bit retrieval data structures need $N(1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3)(1 + o(1)) = N(3 - 2p_1 - p_2 + o(1))$ bits by our assumption. Together, this gives $N(3 - 2p_1 - p_2 + o(1) + \mathcal{O}(1/b))$ bits. For *minimal* perfect hash functions, we need to re-map values into the range $[N]$. For a load factor α close to 1, this takes $(M - N)(2 + \log \frac{N}{M-N} + o(1)) = N(\frac{1}{\alpha} - 1)(2 + \log \frac{\alpha}{1-\alpha} + o(1))$ additional bits, ignoring issues due to rounding (see Sections 2.3 and 2.8). ◀

To get a feeling for this space consumption, we give the idealized space consumption of different SicHash configurations in Table 6.1. All configurations need 2 bits per key in the retrieval data structures. However, the mixes that are less balanced have a higher load threshold (see [LSW23b, Section 7]). We can therefore construct them at higher load factors, which in turn saves space when re-mapping (see Section 2.8) to a minimal perfect hash function. Compared to ordinary cuckoo hashing with $d = 4$ hash functions, irregular cuckoo hashing can save about 0.1 bits per key. This is about 10% of the distance to the space lower bound.

¹ BuRR [Dil+22] sorts elements by a hashed starting position in an equation system. By making this position monotonic in the bucket index one could save that sorting step.

■ **Table 6.1** Idealized space consumption for different SicHash configurations.

p_1	p_2	p_3	Load threshold	Space (Bits/Key)	
				Retrieval	Elias-Fano
0%	100%	0%	0.9767	2.0	0.176
10%	80%	10%	0.9811	2.0	0.148
33%	34%	33%	0.9885	2.0	0.098
50%	0%	50%	0.9921	2.0	0.071

► **Theorem 6.3.** *A SicHash data structure can be constructed in expected time $\mathcal{O}(N)$.*

Proof. Constructing the retrieval data structures takes linear time by assumption. Building the data structures for seeds and offsets is obviously possible in linear time as well. Construction time for a bucket is at most quadratic in the bucket size (and, with constant success probability, retries contribute only a constant factor in expectation). With constant expected bucket size, we get the same execution time as a bucket sorting algorithm that uses a quadratic algorithm per bucket, which is expected linear [San+19, Theorem 5.9]. A similar argument can also be found in RecSplit [EGV20]. ◀

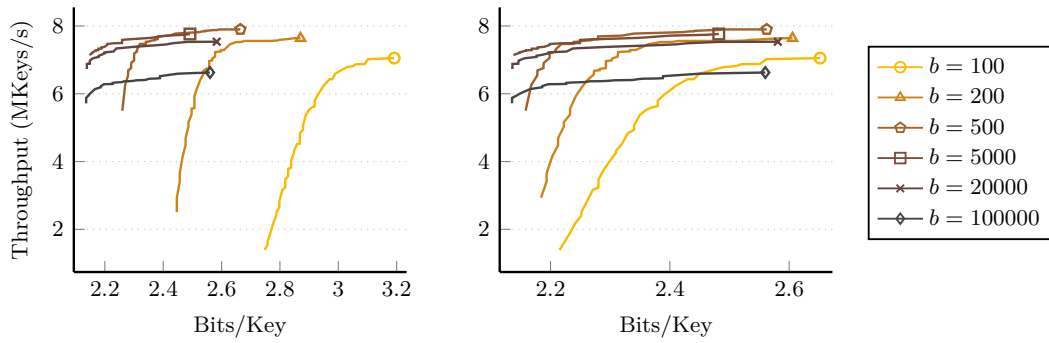
Our implementation gains simplicity and query speed by storing offsets and seeds directly using $\log M$ bits. For this to be space-efficient, we would need average bucket size $b = \Omega(\log N)$. At least with our simple estimation of construction time, this would result in superlinear construction time of $\Omega(N \log N)$. This can be improved using faster construction algorithms. For example, using the Hopcroft-Karp-Karzanov [HK73] algorithm, time $\mathcal{O}(N\sqrt{\log N})$ can be proven. Assuming a result for random graph matching [Bas+06] also transfers to our case, we would even get $\mathcal{O}(N \log \log N)$. At least when we are sufficiently far from the load threshold, various previous results indicate that linear construction time is possible [FMM09; Fot+05; FPS13; KA19; Kho13; Wal22]. With overloading, tight construction time bounds remain an open problem.

6.5 Internal Experiments

In this section, we give internal experiments of SicHash. For a comparison with competitors, as well as an explanation of the experimental setup, we refer to Chapter 8. For now, it is enough to know that we use a consumer machine and run single-threaded experiments. The code and scripts needed to reproduce our experiments are available on GitHub under the General Public License [Leh23f].

As a retrieval data structure, we use Bumped Ribbon Retrieval [Dil+22] with two alternative configurations: $w = 64$ with 2-bit bumping info, and $w = 32$ with 1-bit bumping info. Instead of encoding the per-bucket metadata using Elias-Fano coding and Golomb-Rice coding, we use a plain array, which is faster to access and causes only a small space overhead for sufficiently large buckets. The extension to minimal perfect hashing stores the re-mapping with Elias-Fano coding, which is based on sdsl’s [Gog+14] arrays of flexible bit width and the select data structures by Kurpicz [Kur22].

It is possible to construct the cuckoo hash table with random walk insertion, as well as matching based methods. In our experiments, the random walk variant *rattle kicking* [Kus16] is usually faster than a construction based on Hopcroft-Karp-Karzanov [HK73].



(a) Space consumption by bucket size.

(b) Hypothetical space consumption, assuming that bucket metadata is encoded with Elias-Fano and Golomb-Rice.

■ **Figure 6.5** Pareto plot over the construction throughput of different SicHash configurations by bucket size. 10 million keys, load factor before making minimal: $\alpha = 0.9$.

Bucket Size. For larger buckets, the relative overhead of encoding the metadata is reduced, but they can be overloaded less. This leads to a trade-off that we illustrate in Figure 6.5. Figure 6.5a shows a Pareto front² of SicHash configurations using different bucket sizes and indicates that a bucket size of $b = 5000$ is optimal, which is why we choose that parameter in all other measurements. Figure 6.5b shows hypothetical values for the space usage when assuming that the per-bucket metadata is encoded with Elias-Fano and Golomb-Rice coding instead of arrays (see Section 6.2).

6.6 Summary

With SicHash, we present a new perfect hash function which places keys in a number of small, irregular cuckoo hash tables. Making the tables small enables *overloading*, which achieves higher load factors than the asymptotic bound. Using irregular cuckoo hashing enables fine-grained control over the load factors and lower space usage. We then use space-efficient retrieval data structures to store the final placement. Our implementation improves the state of the art in perfect hash functions for a wide range of load factors and space usage configurations. It profits from the fact that building cuckoo hash tables is a more directed approach for finding bijections than the brute-force methods used at the core of many competitors.

² A configuration is on the Pareto front if it is not dominated by any other configuration with respect to both construction time and space consumption.

7 Small, Heavily Overloaded Cuckoo Hash Tables for Minimal Perfect Hashing

Summary: *In this chapter, we introduce ShockHash – Small, heavily overloaded cuckoo hash tables for minimal perfect hashing. ShockHash uses perfect hashing through retrieval, but works far above the load threshold, relying on retries. In graph terminology, ShockHash samples n -edge random graphs until stumbling on a pseudoforest – a graph where each component contains as many edges as nodes. Using cuckoo hashing, ShockHash then derives an MPHf from the pseudoforest in linear time. We show that ShockHash needs to try only about $(e/2)^n \approx 1.359^n$ seeds in expectation. Compared to brute-force, this reduces the space for storing the seed by roughly n bits and speeds up construction by almost a factor of 2^n . Together with a 1-bit retrieval data structure storing the choice for n keys, ShockHash maintains the asymptotically optimal space consumption. Bipartite ShockHash reduces the expected construction time again to about 1.166^n by maintaining a pool of candidate hash functions and checking all possible pairs.*

Using ShockHash as a building block within the RecSplit framework we obtain ShockHash-RS, which can be constructed up to 3 orders of magnitude faster than competing approaches. ShockHash-RS can build an MPHf for 10 million keys with 1.489 bits per key in about half an hour. When instead using ShockHash after an efficient k -perfect hash function, it achieves space usage similar to the best competitors, while being significantly faster to construct and query.

Attribution: This chapter is based on “ShockHash: Near Optimal-Space Minimal Perfect Hashing Beyond Brute-Force” [LSW24a; LSW24b]. Large parts of this chapter are copied verbatim or with minor changes from that publication. The author of this dissertation is the main author of the paper. He implemented the algorithm, performed the experiments and wrote most of the manuscript. Most ideas for the analysis by Stefan Walzer, with write-up by the author of this dissertation. All authors made significant contributions, to algorithm design, analysis, design & interpretation of the experiments, and the write-up. Peter Sanders more on the side of design and experiments and Stefan Walzer more on the side of the analysis.

In this chapter, we introduce *ShockHash* – Small, heavily overloaded cuckoo hash tables, which can be seen as an extreme version of SicHash [LSW23b] (see Chapter 6). Here we use two hash functions for each key and retry construction until we can completely fill the cuckoo hash table. That way, we achieve an MPHf without an intermediate non-minimal PHF. We describe ShockHash in detail in Section 7.2. In graph terminology, ShockHash repeatedly samples a graph with n edges and n vertices. Each key corresponds to one edge, connecting the candidate positions of the key. The table can be filled if and only if the graph is a *pseudoforest* – a graph where no component contains more edges than nodes. While the

ShockHash idea is straightforward in principle, we can *prove* that when using binary cuckoo hashing with two choices (and thus 1-bit retrieval) there is only an insignificant amount of redundancy. We show that ShockHash approaches the information theoretic lower bound of $n \log e - \mathcal{O}(\log n) \approx 1.44n$ for large n . At the same time, ShockHash has running time $(e/2)^n \cdot \text{poly}(n) \approx 1.359^n$, which is nearly a factor 2^n faster than brute-force.

In *bipartite* ShockHash (see Section 7.3), further exponential improvements are possible. Instead of using a pair of fresh hash functions for each construction attempt, we build a growing pool of hash functions and consider all pairs that can be formed from this pool. Also, we let the two hash functions hash to disjoint ranges, meaning we effectively sample a bipartite graph where each edge has one endpoint in both partitions. In this bipartite setting, the hash functions of both partitions need to be *individually* surjective. We can therefore filter the set of candidate hash functions in each partition individually – before testing all combinations. This improves the construction time by an additional exponential factor, to about $1.166^n \cdot \text{poly}(n)$.

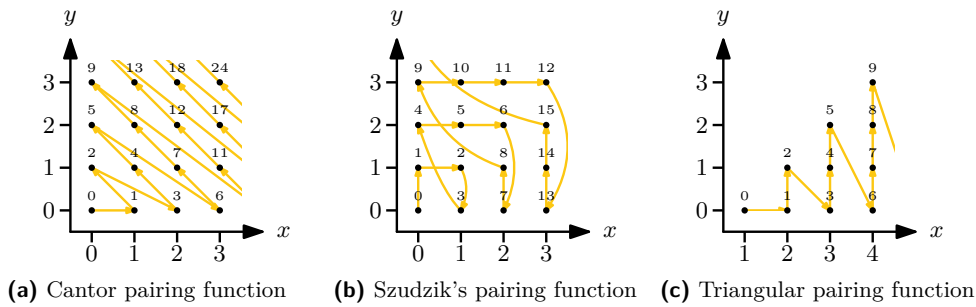
In Section 7.4, we analyze ShockHash and bipartite ShockHash. Still being exponential time algorithms, we use them as a building block after partitioning the input. We describe this in Section 7.5. By using ShockHash instead of brute-force as a base case within the RecSplit framework we obtain ShockHash-RS. We also demonstrate that ShockHash is useful outside the RecSplit framework. When using k -perfect hashing for partitioning the input, we obtain ShockHash-Flat, which achieves a space usage similar to the most space-efficient competitors. At the same time, it is a lot faster to query. In Section 7.6, we give additional variants and refinements that improve the construction in practice. We conclude with an evaluation of the tuning parameters in Section 7.7 before summarizing the results in Section 7.8. For a full comparison with competitors from the literature, we refer to Chapter 8.

7.1 Pairing Functions

A pairing function encodes two natural numbers in a single natural number. More precisely, a pairing function is a bijection between the grid \mathbb{N}_0^2 and \mathbb{N}_0 . We are interested in pairing functions that can be inverted efficiently. The most popular pairing function is the Cantor pairing function, which enumerates the 2D grid diagonally (see Figure 7.1a). It can be calculated by $\text{pair}_c(x, y) = (x + y)(x + y + 1)/2 + y$. Another pairing function is the one by Szudzik [Szu06], which enumerates the 2D grid following the edges of a square (see Figure 7.1b). The pairing function can be calculated by $\text{pair}_s(x, y) = y^2 + x$ if $x \leq y$ and $\text{pair}_s(x, y) = x^2 + x + y$ otherwise.

In this chapter, we require a function that enumerates only those coordinates of the 2D grid with $x > y$. We will still call it a pairing function in slight abuse of traditional terminology. Our *triangular pairing function* (see Figure 7.1c) can be calculated by $\text{pair}_t(x, y) = x(x - 1)/2 + y$ with the intuition stemming from the *Little Gauss* formula. The basic idea for inverting our function $(x', y') = \text{pair}_t^{-1}(z)$ is to set $y = 0$ in the definition and solve for x . This gives $x' = \lfloor 1/2 + \sqrt{1/4 + 2z} \rfloor$ and $y' = z - \text{pair}_t(x', 0)$. In our bipartite implementation, we use both our triangular pairing function and Szudzik's pairing function, depending on the distribution of the numbers we want to encode.

While pairing uses only integer operations, all three pairing functions rely on the square root operation and rounding for inverting. This means that inverting the functions in practice can lead to problems due to floating point inaccuracies. Whether inverting z succeeded can easily be checked by verifying that $\text{pair}(x', y') = z$. In our implementation, we check invertibility at construction time, so we do not get a run-time overhead during queries.



■ **Figure 7.1** Illustrations of different pairing functions.

7.2 ShockHash

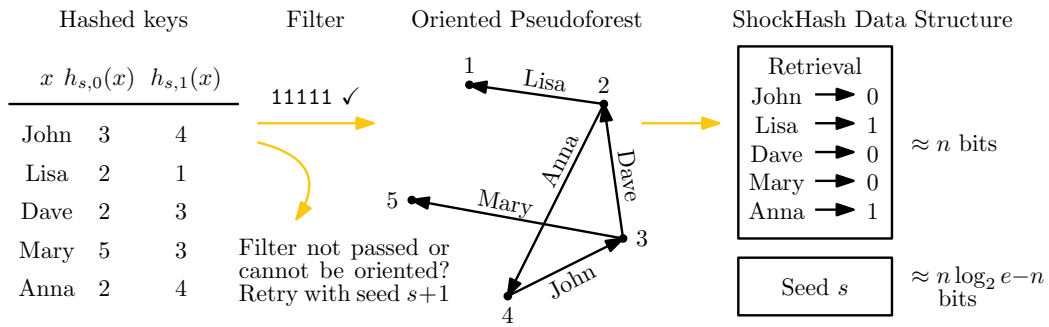
We now introduce the main idea of this chapter, ShockHash. The asymptotic load threshold of a binary cuckoo hash table is $c = 0.5$ (see Section 2.5), so the success probability of constructing a table with n cells and more than $n/2$ keys tends to zero. ShockHash overloads a cuckoo hash table far beyond its asymptotic load threshold – it inserts n keys into a binary cuckoo hash table of size n . As we will see in Theorem 7.7, the construction succeeds after $(e/2)^n \text{poly}(n)$ tries in expectation. We then record the successful seed

$$s = \min\{s \in \mathbb{N} \mid \exists f \in \{0, 1\}^S : x \mapsto h_{s,f(x)}(x) \text{ is MPHf}\} \quad (h_{i,0})_{i \in \mathbb{N}}, (h_{i,1})_{i \in \mathbb{N}} : S \rightarrow [n]$$

and a successful choice f between the two candidate positions of each key. The seed needs $n \cdot \log(e/2) + o(n) \approx 0.44n + o(n)$ bits in expectation using Golomb-Rice codes [Gol66; Ric79] (see Section 2.2). The choices are stored in a 1-bit retrieval data structure, requiring $n + o(n)$ bits. This means that the majority of the MPHf description is not stored in the seed, like with the brute-force construction, but in the retrieval data structure. A query for key x retrieves $f(x)$ from the retrieval data structure and returns $h_{s,f(x)}(x)$. Figure 7.2 gives an illustration of the ShockHash construction.

The beauty of ShockHash is that it can check 2^n different possible hash functions (determined by the 2^n different functions representable by the retrieval data structure) in $\mathcal{O}(n)$ time. This enables significantly faster construction than brute-force while still consuming the same amount of space up to lower order terms. As discussed in Section 2.5, a seed leads to a successful cuckoo hash table construction if and only if the corresponding random (multi)graph with edges $\{\{h_{s,0}(x), h_{s,1}(x)\} \mid x \in S\}$ forms a pseudoforest. Each component of size c is a pseudotree if and only if it contains no more than c edges. This can be checked in linear time using connected components algorithms, or in close to linear time using an incremental construction of a cuckoo hash table. However, compared to the simple bit-parallel perfectness test of brute-force [EGV20], each individual check is slower by a large constant factor. In the following paragraph, we discuss a way to address this bottleneck.

Filter by Bit Mask. To reduce the time spent checking if a graph is a pseudoforest, we use a filter to quickly reject most seeds. More specifically, we reject seeds for which some table cell is not a candidate position of any of the keys. If there is such a cell, we already know that cuckoo hashing cannot succeed, so we can skip the full test. Otherwise, cuckoo hashing might succeed. The filter can be implemented using simple shift and comparison operations. Also, the filter can use registers, in contrast to the more complex full construction. It is one of the main ingredients making ShockHash practical and is easily proven to be very effective:



■ **Figure 7.2** Illustration of the ShockHash construction. Functions $h_{s,0}$ and $h_{s,1}$ are randomly sampled hash functions using a seed s . Here, s is a seed value where the resulting graph is a pseudotree. During construction, many seeds need to be tried.

► **Lemma 7.1.** *The probability for a seed to pass the filter, i.e. for every table cell to be hit by at least one key, is at most $(1 - e^{-2} + o(1))^n \approx 0.864^n$.*

Proof. Let X_i denote the number of times that cell $i \in [n]$ is hit. Then (X_1, \dots, X_n) follows a multinomial distribution. The variables X_1, \dots, X_n are *negatively associated* in the sense introduced in [JDP83] and satisfy

$$\Pr(\forall i \in [n] : X_i \geq 1) \leq \prod_{i=1}^n \Pr(X_i \geq 1),$$

the intuition being that since the sum $X_1 + \dots + X_n = 2n$ is fixed, the events $\{X_i \geq 1\}$ for $i \in [n]$ are less likely to co-occur compared to corresponding independent events. Since $X_i \sim \text{Bin}(2n, \frac{1}{n})$ for all $i \in [n]$ we have

$$\Pr(X_i \geq 1) = 1 - (1 - \frac{1}{n})^{2n} = 1 - e^{-2} + o(1) \approx 0.864$$

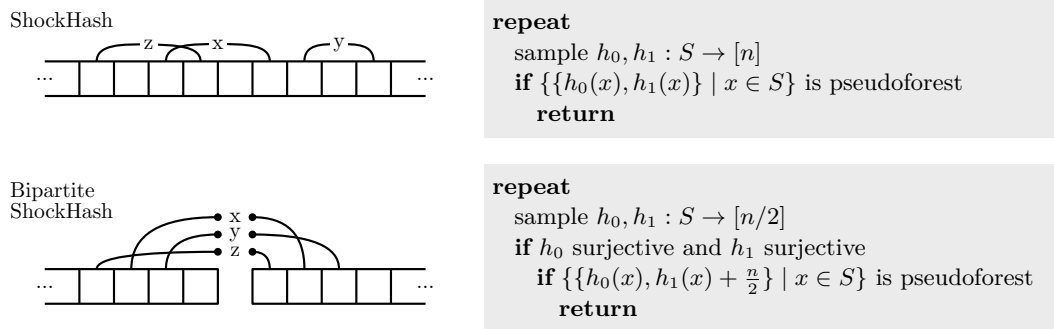
and the claim follows. ◀

A more careful analysis [Wal24] reveals that the probability to pass the filter is around b^n where $b = 2e^\lambda / (\lambda e^2) \approx 0.836$ and where $\lambda \approx 1.597$ is the solution to $2 = \lambda / (1 - e^{-\lambda})$.

Enhancements. In Section 7.6, we explain additional enhancements that improve the construction performance in practice. This includes applying the idea of *rotation fitting* [Bez+23] (see Chapter 5) to ShockHash, as well as faster orientability checks.

7.3 Bipartite ShockHash

Bipartite ShockHash is an extension of the ShockHash idea. It enables significantly faster construction compared to plain ShockHash. In turn, this enables more aggressive parameter choices, thereby leading to improved space-efficiency. While ShockHash samples random graphs, *bipartite* ShockHash now samples *bipartite* random graphs. Figure 7.3 gives an illustration and very simple pseudocode. In plain ShockHash, each edge is connected to two nodes using two independent hash functions. In bipartite ShockHash, the hash functions have a range of $[n/2]$, but we shift the hashes of one of the hash functions by $n/2$, meaning each edge gets one endpoint in $[n/2]$ and one in $n/2 + [n/2]$. This is similar to the original implementation of cuckoo hashing using two independent hash tables [PR04]. The idea might



■ **Figure 7.3** ShockHash and bipartite ShockHash. The pseudocode illustrates the overall idea but does not lead to any performance improvements yet.

■ **Algorithm 7.1** Pseudocode of bipartite ShockHash.

```

Function construct( $S$ )
  surjectiveCandidates  $\leftarrow \emptyset$ 
  for  $s_0 = 0$  to  $\infty$ 
    if  $h_{s_0}$  is surjective on  $S$ 
      for  $s_1 \in$  surjectiveCandidates
        if  $\exists f \in \{0, 1\}^S : x \mapsto \frac{n}{2} \cdot f(x) + h_{s_{f(x)}}(x)$  is a bijection
          return  $f$  as retrieval data structure,  $s_0, s_1$ 
      surjectiveCandidates  $\leftarrow$  surjectiveCandidates  $\cup \{s_0\}$ 

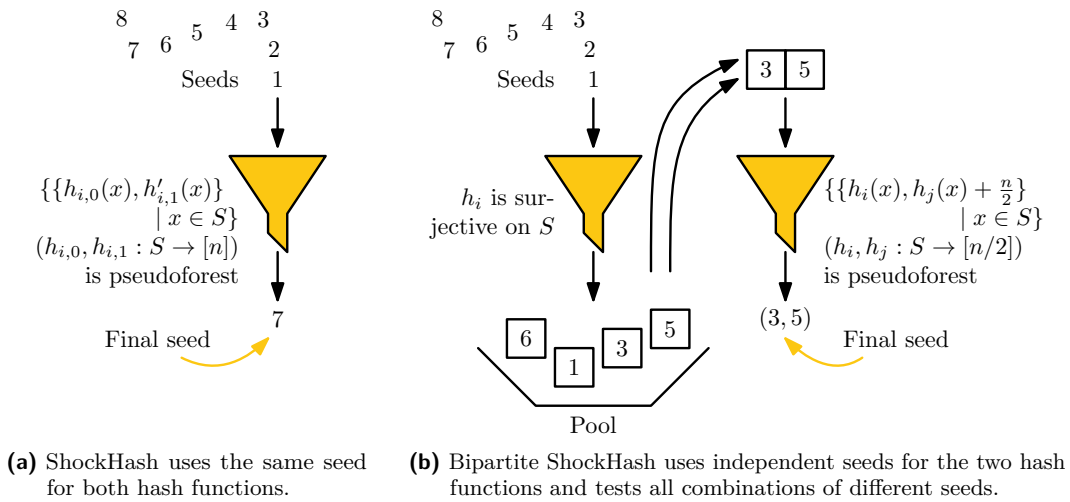
Function evaluate( $x$ )
  if  $f(x) = 0$  // Retrieval
    return  $h_{s_0}(x)$ 
  else
    return  $\frac{n}{2} + h_{s_1}(x)$ 

```

sound not very helpful at first, but opens up several ways of pruning the search space. In the following, we assume that n is an even number. We give an extension to uneven numbers in Section 7.6.4.

Filtering Seed Candidates. We show in Section 7.4 that testing about $(e/2)^n \approx 1.359^n$ pairs of hash functions is sufficient for plain ShockHash. The idea of bipartite ShockHash is that it is almost as good to consider roughly $\sqrt{(e/2)^n} = (e/2)^{n/2}$ hash functions and all pairs that can be formed from them. This already improves the practical running time because fewer hash functions need to be evaluated. However, the asymptotic construction time is not improved much because we still need to test all combinations. A key realization is that in the bipartite case, a pair (h_0, h_1) of hash functions can only work if *both* h_0 and h_1 (both with range $[n/2]$) are *individually* surjective. In the non-bipartite case, in contrast, the check was that h_0 and h_1 (both with range $[n]$) *together* hit each position in $[n]$ at least once. This means that we can filter the list of hash functions *before* pairing them up.

In each of the partitions, we look at n keys mapping to $n/2$ positions. Similar to Lemma 7.1, the probability of passing the filter is $0.836^{n/2}$ [Wal24]. This suggests that if we pair up only the hash functions passing the filter then we will be considering at most $((e/2)^{n/2} \cdot 0.836^{n/2})^2 \approx 1.136^n$ pairs. We refer to Section 7.4 for details.



■ **Figure 7.4** Illustration of the filtering involved in ShockHash and bipartite ShockHash. The construction is complete if we find one final seed that passes all filters.

The Bipartite ShockHash Algorithm. The following paragraph describes our new bipartite ShockHash algorithm. We maintain a pool of seed candidates that are surjective on $[n/2]$. To find a new candidate, we linearly check hash function seeds until we find a seed s_0 that gives a surjective hash function. Given that new candidate, we try to combine it with all previous candidates s_1 from the pool. More precisely, we check if the graph defined by the nodes $[n]$ and the edges $\{\{h_{s_0}(x), h_{s_1}(x) + n/2\} \mid x \in S\}$ is a pseudoforest. If it is a pseudoforest, we have found a perfect hash function. We only need to store the assignment from keys to their candidate hash function (h_{s_0} or h_{s_1}) in a retrieval data structure, as well as the two seeds s_0 and s_1 . If the combination with none of the previous seed candidates leads to a pseudoforest, we add the new candidate s_0 to the set of surjective candidates and search for the next one. Algorithm 7.1 gives a pseudocode for this algorithm and Figure 7.4 illustrates the idea of filtering hash functions before putting them in the pool.

Note that it does not matter which of the two hash functions we use for which partition of the graph. Switching the partitions just gives an isomorphic graph and does not influence orientability. We therefore always use the newly determined candidate directly and shift the old candidate by $n/2$ to be in the second partition. Also, we neglect the possibility that a hash function combined with itself on both partitions leads to successful construction.¹ This allows us to store the two seeds s_0 and s_1 , knowing that $s_1 < s_0$. We do so in one integer using our triangular pairing function that we explain in Section 7.1. Note that the pairing function enumerates the seed pairs in exactly the same order that we test them in. Compared to storing two variable-length integers, pairing reduces constant overheads in the encoding.

Enhancements. In Section 7.6, we give additional enhancements that improve the construction performance significantly in practice. This includes other ways of coming up with a stream of hash function candidates, bit-parallel filtering, and support for uneven input sizes.

¹ The function would have to map exactly two keys to each of the $n/2$ positions, which happens with probability $\binom{n}{2 \dots 2} \left(\frac{n}{2}\right)^{-n} = e^{-n} 2^{n/2} \text{poly}(n)$.

7.4 Analysis

In this section, we analyze the space usage and construction time of ShockHash. The main challenge is to lower bound the probability that a hash function seed enables successful construction of the heavily overloaded cuckoo hash table. First, in Section 7.4.1, we give a very simple analysis of the success probability of plain ShockHash. It is less tight than our more complex proof, but it is significantly shorter. In Section 7.4.2, we explain tools used in the analysis and prove small building blocks of the full proof. In Sections 7.4.3 and 7.4.4, we then analyze the success probability of plain and bipartite ShockHash, respectively. We then show in Section 7.4.5 that a pool containing about $(\sqrt{e/2})^n$ hash function candidates is usually sufficient. Finally, we give the construction time and space consumption of ShockHash and bipartite ShockHash in Section 7.4.6. In the following we assume that a seed is given. We suppress it in notation.

It will be useful to consider the graph $G = ([n], \{\{h_0(x), h_1(x)\} \mid x \in S\})$. While similar to an Erdős-Renyi random graph, G may have self-loops² and multi-edges. Our model matches Model A in [FK16].

7.4.1 A Simple Proof

First, in Theorem 7.2, we give a simple combinatorial argument showing that the probability for G to be a pseudotree is at least $(e/2)^{-n} \sqrt{\pi/(2n)}$. This lower bounds the probability of G being a pseudoforest consisting of potentially more than one tree. Section 7.4.3 then shows that the probability is at least $(e/2)^{-n} \pi/e$. Therefore, the simple argument is only a factor of $\mathcal{O}(\sqrt{n})$ less tight than the much more complex proof.

► **Theorem 7.2.** *Let G be a multigraph with n nodes and n edges. The probability space underlying G is that of sampling $2n$ nodes (with replacement) and creating an edge from the samples $2i - 1$ and $2i$ for each $i \in [n]$. Then the probability that G is a pseudotree is at least $(e/2)^{-n} \sqrt{\pi/(2n)}$.*

Proof. For G to be a pseudotree it is sufficient (though not necessary) that the first $n - 1$ created edges form a tree. There are n^{n-2} labeled n -node trees (Cayley's Formula [Cay78]). Since the ordering of the edges and the order of the two samples forming an edge does not matter, each of the trees can be generated in $2^{n-1}(n - 1)!$ ways. The last two samples can be anything, giving us n^2 choices. By applying Stirling's approximation, namely

$$n! \in \left[\left(\frac{n}{e}\right)^n \sqrt{2\pi n} \cdot e^{1/(12n+1)}, \left(\frac{n}{e}\right)^n \sqrt{2\pi n} \cdot e^{1/(12n)} \right],$$

we can show that the total probability to draw a pseudotree is at least

$$\frac{n^{n-2} 2^{n-1} (n-1)! n^2}{n^{2n}} \geq \left(\frac{e}{2}\right)^{-n} \sqrt{\pi/(2n)}. \quad \blacktriangleleft$$

7.4.2 Tools

In this section, we explain tools that are needed later in the analysis, such as the configuration model and graph peeling. We start with proving two small lemmas that we use in the remaining analysis but are very generic in nature.

² Graphs with self-loops are easier to analyze here but we avoid them in practice for better performance.

► **Lemma 7.3.** *Let $X \in \mathbb{N}_0$ be a random variable. Then the probability that X is at least 1 is*

$$\Pr(X > 0) = \frac{\mathbb{E}(X)}{\mathbb{E}(X \mid X > 0)}.$$

Proof. For any non-negative random variable X we can apply the law of total expectation to get $\mathbb{E}(X) = \Pr(X = 0) \cdot \mathbb{E}(X \mid X = 0) + \Pr(X > 0) \cdot \mathbb{E}(X \mid X > 0) = \Pr(X > 0) \cdot \mathbb{E}(X \mid X > 0)$. Rearranging this for $\Pr(X > 0)$ yields the desired result. ◀

► **Lemma 7.4.** *For $n, c \in \mathbb{N}$, it holds that $\binom{cn}{n} \leq (ec)^n$*

Proof. We first upper bound all factors $(cn - k)$ by cn and then apply Stirling's approximation.

$$\binom{cn}{n} = \frac{(cn)(cn-1)(cn-2)\dots(cn-n+1)}{n!} \leq \frac{(cn)^n}{n!} \leq \frac{(cn)^n}{\sqrt{2\pi n} (n/e)^n} \leq (ec)^n \quad \blacktriangleleft$$

Configuration Model. The *configuration model* [New10] is a way to describe distributions of random graphs. In the model, we can fix the exact degree of every node in the graph by giving each node a number of *stubs* (half-edges). The graph is obtained by repeatedly sampling, uniformly at random, two unconnected stubs and connecting them. In other words, if we take one stub and look at its partner, all other stubs are equally likely.

Graph Peeling. In several sections of this chapter, we are interested in *peeling* [JL07; Mol05; Wal21] graphs. For this, we iteratively take any node of degree 1 and remove it together with its corresponding edge. The process continues until all nodes have degree > 1 . A graph is 1-orientable or a pseudoforest, if all nodes in the remaining graph have degree 2, meaning that the graph consists of only cycles.

Graph Peeling in the Configuration Model. To analyze the peeling process, it will be useful to reveal G in two steps. First the degree of each node is revealed by randomly distributing $2n$ stubs among the n nodes. This yields a configuration model from which the edges are then obtained by randomly matching the stubs. The following lemma should clarify what exactly we need.

► **Lemma 7.5.** *Let $x_1, \dots, x_{2n} \in [n]$ be independent and uniformly random. The graphs G_1, G_2, G_3 defined in the following have the same distribution as G .*

1. $G_1 = ([n], \{\{x_{2i-1}, x_{2i}\} \mid i \in [n]\})$.
2. $G_2 = ([n], \{\{x_i, x_j\} \mid \{i, j\} \in M\})$ where M is a uniformly random perfect matching of $[2n]$, i.e. a partition of $[2n]$ into n sets of size 2.
3. G_3 is defined like G_2 , except that M is obtained in a sequence of n rounds. In each round an unmatched number $i \in [2n]$ is chosen arbitrarily and matched to a distinct unmatched number j , chosen uniformly at random. The choice of i may depend on x_1, \dots, x_{2n} and on the set of numbers matched previously.

The reason for considering these alternative probability spaces for G is that they permit conditioning on partial information about G (such as its degree sequence implicit in x_1, \dots, x_{2n}) but retaining a clean probability space for the remaining randomness.

Proof. Compared to G , the definition of G_1 simply collects the $2n$ relevant hash values in a single list.³ Concerning G_2 , imagine that M is revealed first. Conditioned on M , G_2 is composed of n uniformly random edges like G_1 . Concerning G_3 , the key observation is that M is a uniformly random matching even if the number to be randomly matched in every round is chosen by an adversary. A formal proof could consider any arbitrary adversarial strategy and use induction. ◀

Therefore, when peeling in the configuration model, we can interleave the peeling process and the process of uncovering the sampled graph. To peel, we take a node with degree 1 and look at the other endpoint of its adjacent edge, which is uniformly distributed between all other stubs. If the node connected to it has degree 2, removing the edge gives a new degree-1 node that we can directly continue peeling. Otherwise, we have to start with a new node of degree 1 in a next iteration.

7.4.3 Success Probability in Plain ShockHash

In this section, we give the tighter analysis of the success probability of ShockHash. Given two hash functions $h_0, h_1 : S \rightarrow [n]$ and a function $f : S \rightarrow \{0, 1\}$, let $\text{ori}(f)$ be the event that $x \mapsto h_{f(x)}(x)$ is bijective. We are now interested in the probability that there exists such a function f that leads to a bijective function, namely $\Pr(\exists f : \text{ori}(f))$. There is a one-to-one correspondence between functions f with $\text{ori}(f)$ and 1-orientations of G , i.e. ways of directing G such that each node has indegree at most 1.⁴

We write $\text{PF}(G)$ for the event that G is a pseudoforest. As pointed out in Section 2.5:

$$\text{PF}(G) \Leftrightarrow \exists f : \text{ori}(f). \quad (7.1)$$

In our case with n nodes and n edges, $\text{PF}(G)$ implies that G is a *maximal* pseudoforest, where every component is a pseudotree and not a tree. Note that a pseudotree that is not a tree admits precisely two 1-orientations because the unique cycle can be directed in two ways and all other edges must be directed away from the cycle. A useful observation is therefore

$$\text{PF}(G) \Rightarrow \#\{f : \text{ori}(f)\} = 2^{c(G)} \quad (7.2)$$

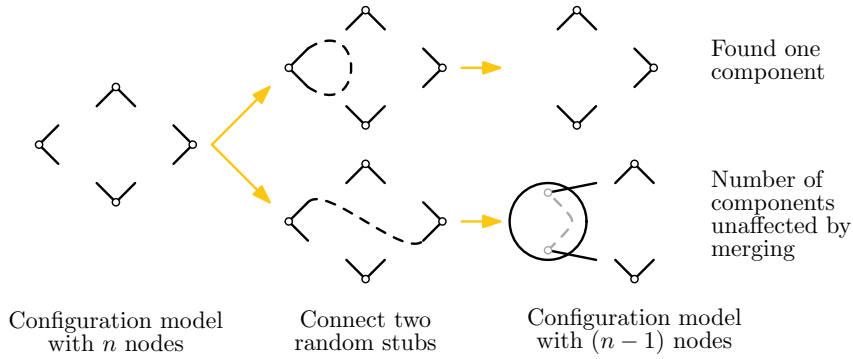
where $c(G)$ is the number of connected components of G .

The basic idea of our proof is as follows. The probability that a random function is minimal perfect is $e^{-n} \text{poly}(n)$ (see Lemma 1.1). Each of the 2^n functions $f : S \rightarrow \{0, 1\}$ has that chance of satisfying $\text{ori}(f)$ and yielding an MPHf. However, simply multiplying $e^{-n} \text{poly}(n)$ by 2^n does not necessarily yield an approximation for the probability that such an f exists. The key point here is that the 2^n functions $(x \mapsto h_{f(x)}(x))_{f \in \{0,1\}^S}$ determined by the 2^n different options for f are correlated. If there are some graphs that permit many different 1-orientations, we may find many MPHfs at once and the probability that there is *at least one* 1-orientation is reduced. This intuition is stated more formally in Lemma 7.3 using $X = \#\{f : \text{ori}(f)\}$:

$$\Pr(\exists f : \text{ori}(f)) = \frac{\mathbb{E}(\#\{f : \text{ori}(f)\})}{\mathbb{E}(\#\{f : \text{ori}(f)\} \mid \exists f : \text{ori}(f))}.$$

³ Here, we assume that h_0 and h_1 are fully random hash functions and given for free, which is common in previous papers (Simple Uniform Hashing Assumption) [DH90; DR09; PP08; PPR07].

⁴ Note that in our model, there are two ways of directing a self-loop.



■ **Figure 7.5** Number of components in the configuration model.

As discussed above, a key step in the analysis is to show that we usually find only a few MPHFs at once. This amounts to analyzing the distribution of the number of components in random maximal pseudoforests, which we do in Lemma 7.6. The main proof in Theorem 7.7 then formally bounds the probability that a random graph is a pseudoforest, juggling different probability spaces.

► **Lemma 7.6.** *Let G_n be the random graph sampled from the configuration model with n nodes of degree 2, i.e. the 2-regular graph obtained by randomly joining $2n$ stubs that are evenly distributed among n nodes. Then the number $c(G_n)$ of components of G satisfies $\mathbb{E}(2^{c(G_n)}) \leq e \cdot \sqrt{2n}$.*

We remark that a similar proof shows that $\mathbb{E}(c(G_n)) \in \mathcal{O}(\log n)$. Note also the similarity to the locker puzzle, which analyzes the length of the largest cycle in a random permutation [Sta11].

Proof. We will find a recurrence for $d_n := \mathbb{E}(2^{c(G_n)})$. Consider an arbitrary node v of G_n and one of the stubs at v . This stub forms an edge with some other stub. We have $n - 1$ other nodes, each with 2 stubs, and we have the second stub at v . Each of these $2n - 1$ stubs is matched with v with equal probability. Therefore, the probability that v has a self-loop is $\frac{1}{2n-1}$.

(1) Conditioned on v having a self-loop, we have found an isolated node. The distribution of the remaining graph is that of G_{n-1} and the conditional expectation of $2^{c(G)}$ is therefore $\mathbb{E}(2^{1+c(G_{n-1})}) = 2d_{n-1}$.

(2) Now condition on the formed edge connecting v to $w \neq v$. We can now merge the nodes to a single one without affecting the number of components. The merged node inherits two unused stubs, one from v and one from w . The distribution of the remaining graph is that of G_{n-1} . Therefore, in this case, the conditional expectation of $2^{c(G)}$ is simply d_{n-1} .

These two cases are illustrated in Figure 7.5 and lead us to the following recurrence:

$$d_n = \frac{1}{2n-1}2d_{n-1} + \left(1 - \frac{1}{2n-1}\right)d_{n-1} = \left(1 + \frac{1}{2n-1}\right)d_{n-1}.$$

With the base case $d_0 = 1$, we can solve the recurrence and bound its value as follows, using that $\ln(1+x) \leq x$ for $x \geq 0$ as well as $H_n := \sum_{i=1}^n \frac{1}{i} \leq 1 + \ln n$:

$$\begin{aligned}
d_n &= \prod_{i=1}^n \left(1 + \frac{1}{2i-1}\right) = \exp\left(\sum_{i=1}^n \ln\left(1 + \frac{1}{2i-1}\right)\right) \leq \exp\left(\sum_{i=1}^n \frac{1}{2i-1}\right) \\
&= \exp\left(1 + \sum_{i=2}^n \frac{1}{2i-1}\right) = \exp\left(1 + \frac{1}{2} \sum_{i=2}^n \left(\frac{1}{2i-1} + \frac{1}{2i-1}\right)\right) \\
&\leq \exp\left(1 + \frac{1}{2} \sum_{i=2}^n \left(\frac{1}{2i-1} + \frac{1}{2i-2}\right)\right) = \exp\left(1 + \frac{1}{2} \sum_{i=2}^{2n-1} \frac{1}{i}\right) \\
&= \exp((1 + H_{2n-1})/2) \leq \exp(1 + \ln(2n)/2) \leq e \cdot \sqrt{2n}. \quad \blacktriangleleft
\end{aligned}$$

Finally, we can derive the success probability of the ShockHash search as follows.

► **Theorem 7.7.** *Let $h_0, h_1 : S \rightarrow [n]$ be uniformly random functions. The probability that there exists $f : S \rightarrow \{0, 1\}$ such that $x \mapsto h_{f(x)}(x)$ is bijective is at least $(e/2)^{-n} e^{-1} \sqrt{\pi}$.*

Proof. Recall our shorthand $\text{ori}(f)$ for the event that $x \mapsto h_{f(x)}(x)$ is bijective. As given in Lemma 7.3, we can calculate the success probability as follows.

$$\Pr(\exists f : \text{ori}(f)) = \frac{\mathbb{E}(\#\{f : \text{ori}(f)\})}{\mathbb{E}(\#\{f : \text{ori}(f)\} \mid \exists f : \text{ori}(f))}.$$

We will consider the numerator and denominator in turn.

Numerator: Expectation. Linearity of expectation (holding even for dependent variables) yields

$$\mathbb{E}(\#\{f : \text{ori}(f)\}) = \sum_f \Pr(\text{ori}(f)) = \sum_f \Pr(x \mapsto h_{f(x)}(x) \text{ is bijective on } S).$$

For any fixed f , the function $x \mapsto h_{f(x)}(x)$ assigns independent random numbers to each $x \in S$, i.e. is a random function as considered in Lemma 1.1 and hence bijective with probability $e^{-n} \sqrt{2\pi n} \cdot (1 + o(1))$. We therefore get

$$\mathbb{E}(\#\{f : \text{ori}(f)\}) \geq 2^n \cdot e^{-n} \sqrt{2\pi n}. \quad (7.3)$$

Denominator: Conditional Expectation. Using observations (7.1) and (7.2) we can shift our attention to the graph G :

$$\mathbb{E}(\#\{f : \text{ori}(f)\} \mid \exists f : \text{ori}(f)) = \mathbb{E}(2^{c(G)} \mid \text{PF}(G)).$$

By virtue of Lemma 7.5 we can moreover move to a configuration model à la G_3 . We first reveal the locations x_1, \dots, x_{2n} of the $2n$ stubs (hence the degree sequence of G_3) and then consider the following peeling process (see Section 7.4.2) that reveals edges of G_3 and simplifies G_3 in a step-by-step fashion.

As long as there exists a node v with only one stub, firstly, match it to a random stub to form a corresponding edge $\{v, w\}$ (consuming the two stubs) and, secondly, remove the node v and the newly formed edge $\{v, w\}$. These removals do not affect the number of components of the resulting graph (since v was connected to w), nor whether the resulting graph is a pseudoforest (since the component of w lost one node and one edge).

Let n' be the number of nodes that remain after peeling and let G' be the graph obtained by matching the remaining stubs. As discussed we have $\text{PF}(G_3) \Leftrightarrow \text{PF}(G')$ and $c(G') = c(G_3)$.

Since the average degree of G_3 is 2 and since we removed one node and one edge in every round, the average degree of G' is also 2. There are two cases.

(1) Some node of G' has degree 0. Then $\neg\text{PF}(G')$ because some component of G' must have average degree > 2 .

(2) No node of G' has degree 0. Since we ran the peeling process, there is also no node of G' with degree 1. Hence, every node of G' has degree 2. This makes G' a collection of cycles. In particular $\text{PF}(G')$ holds. Moreover, the generation of G' is precisely the situation discussed in Lemma 7.6.

Because the two cases imply opposite results on G' being a pseudoforest, we know that $\text{PF}(G')$ holds if *and only if* we arrive in Case 2. While we have no understanding of the distribution of n' , we can nevertheless compute:

$$\begin{aligned} \mathbb{E}(2^{c(G)} \mid \text{PF}(G)) &= \mathbb{E}(2^{c(G_3)} \mid \text{PF}(G_3)) = \mathbb{E}(2^{c(G')} \mid \text{PF}(G')) = \mathbb{E}(2^{c(G')} \mid \text{Case 2}) \\ &\leq \max_{1 \leq i \leq n} \mathbb{E}(2^{c(G')} \mid \text{Case 2 with } n' = i) \leq \max_{1 \leq i \leq n} e\sqrt{2i} = e\sqrt{2n}. \end{aligned} \quad (7.4)$$

Putting the Observations Together. Combining our bounds on the numerator (7.3) and the denominator (7.4) gives the final result

$$\Pr(\exists f : \text{ori}(f)) \geq 2^n e^{-n} \sqrt{2\pi n} / (e\sqrt{2n}) = (e/2)^{-n} e^{-1} \sqrt{\pi}. \quad \blacktriangleleft$$

7.4.4 Success Probability in Bipartite ShockHash

In the bipartite case, we can basically perform the same steps as in the non-bipartite version. For simplicity, we restrict ourselves to even n in the analysis. While our implementation does support uneven n (see Section 7.6.4), this complicates the analysis and can be largely avoided when ShockHash is integrated into a partitioning framework like RecSplit (see Section 7.5). In this section, we again suppress the seed of the hash functions. Let $h_0, h_1 : S \rightarrow [n/2]$ be hash functions and $f : S \rightarrow \{0, 1\}$ be a function that selects between the two hash functions.

We now look at the effect of testing all correlated choices of the function f . As argued in Section 7.4.3, we start with peeling the corresponding graph until there are no nodes with degree 1 left. Conditioned on the graph being a pseudoforest, this leaves us with a graph where each node has degree 2. The distribution of this graph is captured again by a configuration model (see Section 7.4.2), namely giving a random bipartite matching between the stubs. Additionally, remember that we started with a bipartite graph, so both partitions have the same size. Similar to Lemma 7.6, we can now show in Lemma 7.8 that the number of components c in the remaining graph satisfies $\mathbb{E}(2^c) \leq e \cdot \sqrt{n}$. Because the peeling process does not change the number of components, the same applies also to the original bipartite graph. This gives us a bound for the expected number of orientations of the graph, e.g., the number of different functions f that all make the hash function pair (h_0, h_1) minimal perfect.

► **Lemma 7.8.** *Let n be an even number, and let G_n be a random bipartite graph with $n/2$ nodes in each partition, where all nodes have degree 2, sampled from the corresponding bipartite configuration model. Hence, the stubs from one partition are matched to the stubs of the other partition uniformly at random. Then the number $c(G_n)$ of components of G_n satisfies $\mathbb{E}(2^{c(G_n)}) \leq e \cdot \sqrt{n}$.*

Proof. We will find a recurrence for $d_n := \mathbb{E}(2^{c(G_n)})$. Consider an arbitrary node v from the first partition of G_n and one of the stubs at v . Because the graph is bipartite, this stub forms an edge to a node r from the other partition of the graph. The node r has a second

stub that is connected back to the first partition. We now have $n/2 - 1$ other nodes in the first partition, each with 2 stubs, and we have the second stub at v . Each of these $n - 1$ stubs is matched with equal probability. Therefore, the probability that this edge closes a cycle is $\frac{1}{n-1}$.

(1) Conditioned on closing the cycle, the distribution of the remaining graph is that of G_{n-2} and the conditional expectation of $2^{c(G_n)}$ is therefore $\mathbb{E}(2^{1+c(G_{n-2})}) = 2d_{n-2}$.

(2) Now condition on the edge not closing a cycle. We can now merge the three considered nodes to a single one without affecting the number of components. The merged node inherits two unused stubs, and the graph is now bipartite with $n/2 - 1$ nodes in each partition. The distribution of the remaining graph therefore is that of G_{n-2} . Therefore, the conditional expectation of $2^{c(G)}$ is simply d_{n-2} .

These two cases are similar to the non-bipartite case illustrated in Figure 7.5 and lead us to the following recurrence:

$$d_n = \frac{1}{n-1}2d_{n-2} + \left(1 - \frac{1}{n-1}\right)d_{n-2} = \left(1 + \frac{1}{n-1}\right)d_{n-2}.$$

With the base case $d_0 = 1$, we can solve the recurrence and bound its value as follows, using that $\ln(1+x) \leq x$ for $x \geq 0$ as well as $H_n := \sum_{i=1}^n \frac{1}{i} \leq 1 + \ln n$:

$$\begin{aligned} d_n &= \prod_{i=1}^{n/2} \left(1 + \frac{1}{2i-1}\right) = \exp\left(\sum_{i=1}^{n/2} \ln\left(1 + \frac{1}{2i-1}\right)\right) \leq \exp\left(\sum_{i=1}^{n/2} \frac{1}{2i-1}\right) \\ &= \exp\left(1 + \sum_{i=2}^{n/2} \frac{1}{2i-1}\right) = \exp\left(1 + \frac{1}{2} \sum_{i=2}^{n/2} \left(\frac{1}{2i-1} + \frac{1}{2i-1}\right)\right) \\ &\leq \exp\left(1 + \frac{1}{2} \sum_{i=2}^{n/2} \left(\frac{1}{2i-1} + \frac{1}{2i-2}\right)\right) = \exp\left(1 + \frac{1}{2} \sum_{i=2}^{n-1} \frac{1}{i}\right) \\ &= \exp((1 + H_{n-1})/2) \leq \exp(1 + \ln(n)/2) \leq e \cdot \sqrt{n}. \quad \blacktriangleleft \end{aligned}$$

We can lower bound the success probability by applying Lemma 7.3 similarly as in Theorem 7.7. In contrast to Theorem 7.7, we now use $\text{ori}(f)$ adapted for the bipartite case. Given two hash functions $h_0, h_1 : S \rightarrow [n/2]$ and a function $f : S \rightarrow \{0, 1\}$, let $\text{ori}(f)$ be the event that $x \mapsto h_{f(x)}(x) + f(x) \cdot \frac{n}{2}$ is bijective. We write $\text{PF}(h_0, h_1)$ for the event that the graph defined by the two hash functions h_0 and h_1 is a pseudoforest, and again get $\text{PF}(h_0, h_1) \Leftrightarrow \exists f : \text{ori}(f)$.

► **Theorem 7.9.** *Let $h_0, h_1 : S \rightarrow [n/2]$ be uniformly random functions. The probability that there exists $f : S \rightarrow \{0, 1\}$ such that $x \mapsto h_{f(x)}(x) + f(x) \cdot \frac{n}{2}$ is bijective is at least $(e/2)^{-n} \sqrt{n}/e$.*

Proof. All bipartite ShockHash functions have the form $(x \mapsto h_{f(x)}(x) + f(x) \cdot (n/2))$. While it is clear that the results of different x are independent, let us first justify why the function is uniform. For this, let $c \in [n]$ be a constant, x an input value, and $g : S \rightarrow \{0, 1\}$ a uniform random function. Then we get

$$\Pr(h_{g(x)}(x) + g(x) \cdot \frac{n}{2} = c) = \begin{cases} \Pr(h_0(x) = c \wedge g(x) = 0), & c < \frac{n}{2} \\ \Pr(h_1(x) = c - \frac{n}{2} \wedge g(x) = 1) & c \geq \frac{n}{2} \end{cases} = \frac{1}{n/2} \cdot \frac{1}{2} = \frac{1}{n}.$$

For uniform random functions g it holds that $\mathbb{E}(\#\{f : \text{ori}(f)\}) = 2^n \cdot \Pr(\text{ori}(g))$. Because we now also know that bipartite ShockHash with random g gives a uniform random function,

we know that $\Pr(\text{ori}(g))$ matches the probability that a random function is a bijection. Applying Lemma 1.1, this gives $\mathbb{E}(\#\{f : \text{ori}(f)\}) \geq 2^n \cdot e^{-n\sqrt{2\pi n}}$.

To determine the overall success probability, we can now continue similar to Theorem 7.7. Therefore, a random pair of hash functions h_0, h_1 permits at least one valid placement f with at least the following probability.

$$\begin{aligned} \Pr(\exists f : \text{ori}(f)) &\stackrel{\text{Lem. 7.3}}{=} \frac{\mathbb{E}(\#\{f : \text{ori}(f)\})}{\mathbb{E}(\#\{f : \text{ori}(f)\} \mid \exists f : \text{ori}(f))} \geq \frac{2^n \cdot e^{-n\sqrt{2\pi n}}}{\mathbb{E}(\#\{f : \text{ori}(f)\} \mid \exists f : \text{ori}(f))} \\ &\stackrel{\text{Lem. 7.8}}{\geq} e^{-n\sqrt{2\pi n}} \cdot 2^n / (e \cdot \sqrt{n}) = (e/2)^{-n} \sqrt{n}/e \quad \blacktriangleleft \end{aligned}$$

Our bound is a factor of $\sqrt{2}$ better than with plain ShockHash, which reduces our bound on the expected space consumption by $\log(\sqrt{2}) = 0.5$ bits. It might be possible to save a few additional bits in the retrieval data structure because f is known to map exactly half of the keys to 1, i.e., only a $1/\Theta(\sqrt{n})$ fraction of all functions can occur as f . However, we do not consider this in more detail here.

7.4.5 Hash Function Pools

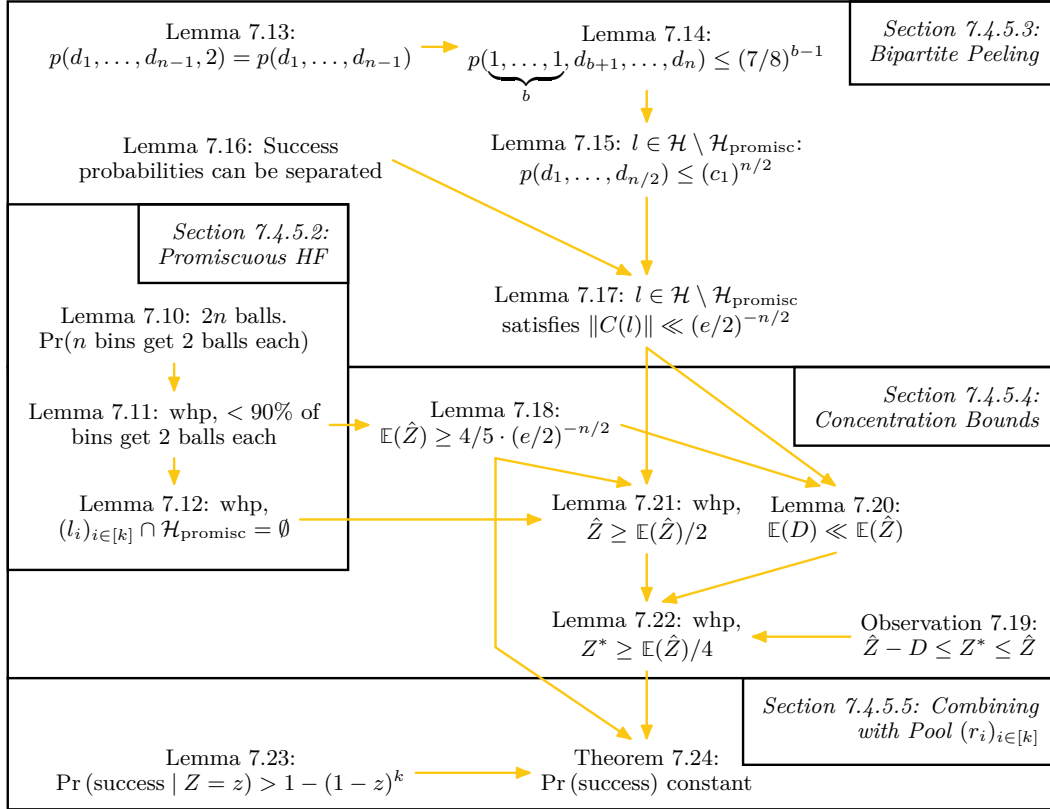
An important ingredient for making bipartite ShockHash so much more efficient than the plain version is the fact that we can combine hash functions from a pool of candidates. This makes it possible to filter the candidates *before* combining them. In the following section, we show why testing all combinations of two hash functions from a pool of candidates has a similar success probability as always sampling two fresh functions. For this analysis, let us form *two* pools of hash functions of size $k = (e/2)^{n/2}$. Note that this is slightly different to the construction explained before, which uses a single, growing pool. However, this does not influence the asymptotic construction time.⁵ The first pool $(l_i)_{i \in [k]}$ contains uniformly drawn hash functions l_1, \dots, l_k and the second pool $(r_i)_{i \in [k]}$ contains uniformly drawn hash functions r_1, \dots, r_k . Our algorithm tests all combinations between two hash functions l_i and r_j ($i, j \in [k]$) from the pools. We are therefore interested in the probability that the pools contain two hash functions that are *compatible*, meaning that their combined graph is a pseudoforest. For easier presentation, we suppress polynomial factors in this section and assume large n .

7.4.5.1 Notation and Overview

Let $\mathcal{H} := [n/2]^S$ be the set of all (hash) functions from S to $[n/2]$. Sampling two hash functions $l, r \sim \mathcal{U}(\mathcal{H})$ uniformly at random gives $\Pr(\text{PF}(l, r)) = (e/2)^{-n}$, as shown in Theorem 7.9 (ignoring polynomial factors). In the following, it will be useful to also consider $\mathcal{H}_{\text{promisc}}$ containing *promiscuous* functions. The intuition is that a promiscuous function has a very large number of compatible partners, which is very unlikely. For reasons we will discuss later, we define $\mathcal{H}_{\text{promisc}}$ as the set of functions that hit more than 90% of the hash values exactly two times:

$$\mathcal{H}_{\text{promisc}} := \left\{ f \in \mathcal{H} \mid \left| \{i \in [n/2] \mid |f^{-1}(i)| = 2\} \right| \geq 0.9n/2 \right\}.$$

⁵ One could view the two pools as one single pool of twice the size, where we only test a subset of the combinations. The time for testing more seed combinations would then be a constant factor larger. In practice, however, we test all combinations within the single pool, so it does not actually need to be twice as large.



■ **Figure 7.6** Illustration of the proof structure showing the success probability of sampling from hash function pools instead of independent hash functions. Variables \hat{Z} , D and Z^* are defined in Section 7.4.5.4. Functions p and q are defined in Section 7.4.5.3.

For a specific function l , we define the set $C(l)$ as all hash functions that are compatible with l . We also define $C^*(l)$ as only those compatible functions that are not promiscuous:

$$C(l) := \{r \in \mathcal{H} \mid \text{PF}(l, r)\}, \quad C^*(l) := C(l) \setminus \mathcal{H}_{\text{promisc}}.$$

For a set $X \subseteq \mathcal{H}$ of hash functions, let $\|X\| := |X|/|\mathcal{H}|$. This is the probability that a randomly sampled hash function is one of the hash functions in X . Then $\|C(l)\|$ is the probability that a randomly sampled hash function is compatible with l . If l is a random variable, this probability is a random variable as well. The expected value for uniform random l is $\mathbb{E}_{l \sim \mathcal{U}(\mathcal{H})}(\|C(l)\|) = \Pr_{l, r \sim \mathcal{U}(\mathcal{H})}(\text{PF}(l, r)) = (e/2)^{-n}$ (ignoring polynomial factors).

The main random variable we are interested in is $Z := \|\bigcup_{i \in [k]} C(l_i)\|$. It depends on the hash functions that we sampled for the pool. Z is the probability that a randomly sampled hash function is compatible with at least one function from our pool $(l_i)_{i \in [k]}$.

If Z was small, it would mean that the hash functions in our pool need a very specific set of partners. Therefore, in this case, it would be unlikely that one of the compatible partners was drawn for the pool $(r_i)_{i \in [k]}$. However, we will show that Z is large enough that $(r_i)_{i \in [k]}$ likely contains a compatible partner. More specifically, we will show that Z is closely concentrated around $(e/2)^{-n/2}$. Since we have $k = (e/2)^{n/2}$ hash functions in each pool, we get a constant probability that a compatible function is in $(r_i)_{i \in [k]}$.

Figure 7.6 gives an overview over the proof structure. We start our proof in Section 7.4.5.2, showing that it is unlikely that a hash function is promiscuous. We then show in Section 7.4.5.3

that functions $\notin \mathcal{H}_{\text{promise}}$ do not have too many compatible partners. This is a key ingredient for providing concentration bounds on Z in Section 7.4.5.4. Finally, we combine this with the pool $(r_i)_{i \in [k]}$ in Section 7.4.5.5.

7.4.5.2 Promiscuous Hash Functions

Before being able to give concentration bounds, we have to rule out a special case of hash functions with too many compatible partners. As stated before, we call these *promiscuous*. In this section, we show that these functions are very rare. To show this, we interpret the output values of our hash functions as a balls-into-bins process. In the following, we show a general property of balls-into-bins processes that we then later apply to our hash functions.

► **Lemma 7.10.** *When throwing $2n$ balls into n bins independently and uniformly at random, the probability that each bin receives exactly two balls is $p_n \leq 5\sqrt{n}(2e^{-2})^n$.*

Proof. To assign the balls to the bins such that each bin receives exactly two balls, we choose n subsets of size 2 from the set of $2n$ balls. The number of ways we can do this is given by the multinomial coefficient $\binom{2n}{2, 2, \dots, 2}$. Each of these combinations has a probability of n^{-2n} . Using Stirling's approximation in the step annotated with *, this gives

$$\begin{aligned} p_n &= \binom{2n}{\underbrace{2, 2, \dots, 2}_{n \text{ times}}} \cdot n^{-2n} = \frac{(2n)!}{\underbrace{2 \cdot 2 \cdot \dots \cdot 2}_{n \text{ times}}} n^{-2n} \stackrel{*}{\leq} \sqrt{2\pi 2n} \left(\frac{2n}{e}\right)^{2n} e^{\frac{1}{24n}} \cdot 2^{-n} \cdot n^{-2n} \\ &= \sqrt{4\pi n} \cdot (2e^{-2})^n \cdot e^{\frac{1}{24n}} \leq 5\sqrt{n}(2e^{-2})^n. \quad \blacktriangleleft \end{aligned}$$

► **Lemma 7.11.** *When throwing n balls into $n/2$ bins independently and uniformly at random, the probability $p_{90\%}$ that more than 90% of the bins receive exactly 2 balls is $p_{90\%} \leq 0.66^n$.*

Proof. Let us consider a set A of $0.9 \cdot n/2$ bins that will each receive exactly 2 balls each. We do not care *which* bins receive exactly 2 balls, so there are $\binom{n/2}{0.9 \cdot n/2} = \binom{n/2}{0.1 \cdot n/2}$ ways of selecting the set A . The probability that exactly $2|A|$ balls land in a bin in A is $\binom{n}{0.1n} 0.1^{0.1n} 0.9^{0.9n}$. Conditioned on this, the probability that these $2|A|$ balls are evenly distributed among the $|A|$ bins is $p_{|A|} \leq 5\sqrt{0.9 \cdot n/2} (2e^{-2})^{0.9 \cdot n/2}$ by Lemma 7.10. Bringing this together we can bound $p_{90\%}$ as follows, applying Lemma 7.4.

$$\begin{aligned} p_{90\%} &\leq \binom{n}{0.1n} 0.1^{0.1n} 0.9^{0.9n} \cdot \binom{n/2}{0.1 \cdot n/2} \cdot (2e^{-2})^{0.9 \cdot n/2} 5\sqrt{0.9 \cdot n/2} \\ &\leq (10e)^{0.1n} \cdot 0.1^{0.1n} 0.9^{0.9n} \cdot (10e)^{0.1 \cdot n/2} \cdot (2e^{-2})^{0.9 \cdot n/2} 5\sqrt{0.45n} \\ &= \left((10e)^{0.1} \cdot 0.1^{0.1} 0.9^{0.9} \cdot (10e)^{0.05} \cdot (2e^{-2})^{0.45} \right)^n 5\sqrt{0.45n} \\ &< 0.66^n \text{ for } n \text{ large enough.} \quad \blacktriangleleft \end{aligned}$$

► **Lemma 7.12.** *With a probability of $1 - 0.77^n$, none of the hash functions in the pool $(l_i)_{i \in [k]}$ is promiscuous.*

Proof. Each of our hash functions maps the n keys to $n/2$ nodes in the graph. Because the hash functions are sampled at random, this can be modeled as a balls-into-bins process. We can apply Lemma 7.11, telling us that the probability that a function is promiscuous is $p_{90\%} \leq 0.66^n$. We sample a pool of $k = (e/2)^{n/2}$ hash functions. Therefore, the probability

that at least one of our hash functions is promiscuous can be upper bounded as follows using a union bound.

$$\Pr\left(\bigvee_{i \in [k]} l_i \in \mathcal{H}_{\text{promisc}}\right) \leq k \cdot p_{90\%} = ((e/2)^{1/2} \cdot 0.66)^n < 0.77^n \quad \blacktriangleleft$$

7.4.5.3 Peeling Bipartite Graphs

For plain ShockHash, we have looked at the event $\text{PF}(G)$ that the resulting graph G is a pseudoforest, and have given bounds for $\Pr(\text{PF}(G))$. In Section 7.4.2, we have shown that we can uncover the graph in two steps: we first reveal the degree of each node and then match these stubs at random. Staying with plain ShockHash for the moment, we now condition on the degrees of the nodes. For this, let (d_1, \dots, d_n) be the degrees of each node, and let $p(d_1, \dots, d_n) := \Pr(\text{PF}(G) \mid \text{nodes of } G \text{ have degrees } d_1, \dots, d_n)$ be the conditioned success probability. Note that the order of the function arguments does not matter to the success probability because the stubs are matched randomly. Let \mathcal{B}_n be the distribution of balls in the balls-into-bins process with n balls and $n/2$ bins. Before we give additional properties of $p(d_1, \dots, d_n)$ in the following lemmas, let us look at a simple observation about the function:

$$\mathbb{E}_{(d_1, \dots, d_n) \sim \mathcal{B}_{2n}}(p(d_1, \dots, d_n)) = \Pr(\text{PF}(G)) \stackrel{\text{Lem. 7.7}}{=} (e/2)^{-n}. \quad (7.5)$$

The graph G is a pseudoforest if we can repeatedly peel away nodes of degree 1 and end up with a graph that consists of only nodes of degree 2 (forming cycles). When peeling, we repeatedly take a node of degree 1 and follow its edge to a random stub. There are now three possible outcomes.

- (1) The edge leads to a node with degree 1. Then we have found a component with more nodes than edges, meaning that the remaining graph cannot be a pseudoforest.
- (2) The edge leads to a node with degree ≥ 3 . Then we have peeled away a subtree of a component that is potentially still a pseudotree. We now have one less node of degree 1 in our graph. In that case, we continue the peeling process with another node of degree 1.
- (3) The edge leads to a node with degree 2. Then it generates a new node with degree 1, and we can continue the peeling process with that same node.

► **Lemma 7.13.** *Nodes of degree 2 do not influence the success probability. More formally, $p(d_1, \dots, d_i, 2) = p(d_1, \dots, d_i)$.*

Proof. If the peeling process arrives in case (3) where it connects to a node of degree 2, we can immediately continue peeling with that node. This always succeeds and does not cause an abort of the peeling process. Afterwards, we have one less node but did not influence the degrees of all other nodes. Therefore, the probabilities to arrive in the more interesting cases (1) and (2) stay the same, relative to each other. ◀

► **Lemma 7.14.** *If the graph corresponding to our hash function has b nodes of degree 1, p is exponentially small in b . More formally, $p(\underbrace{1, \dots, 1}_b, d_{b+1}, \dots, d_n) \leq (7/8)^{b-1}$.*

Proof. During the peeling process, we can ignore all nodes of degree 2 because they do not influence the success probability (see Lemma 7.13). Let us therefore now look at the remaining nodes. If we connect to one of the nodes of degree 1, we have found a tree. This means that the construction fails because we need each component to be a pseudotree and not a tree. Because the average degree is 2, we have at most $3b$ stubs at nodes with degree

≥ 3 . Therefore, the probability that we connect to a node of degree 1 (possibly indirectly through nodes of degree 2) is at most $(b-1)/(4b)$. We can now apply this iteratively until all nodes of degree 1 are peeled away. This gives the following probability that we never connect to a node of degree 1 (and therefore fail):

$$\begin{aligned} p(\underbrace{1, \dots, 1}_b, d_{b+1}, \dots, d_n) &\leq \prod_{j=1}^b \left(1 - \frac{j-1}{4j}\right) = \prod_{j=1}^b \left(\frac{3}{4} + \frac{1}{4j}\right) \\ &= \prod_{j=2}^b \left(\frac{3}{4} + \frac{1}{4j}\right) \leq \prod_{j=2}^b \left(\frac{7}{8}\right) = \left(\frac{7}{8}\right)^{b-1}. \quad \blacktriangleleft \end{aligned}$$

We call a hash function promiscuous if more than 90% of the nodes in its corresponding (stub) graph have degree 2. This means that it cannot have too many nodes of degree 1. Promiscuous hash functions are rare, as we have seen in Section 7.4.5.2. In the following, we bound the number of compatible hash functions when our function is *not* promiscuous.

► **Lemma 7.15.** *Let l be a hash function from $\mathcal{H} \setminus \mathcal{H}_{\text{promisc}}$ and $(d_1, \dots, d_{n/2})$ be the corresponding degree sequence. Then $p(d_1, \dots, d_{n/2}) \leq (c_1)^{n/2}$ where c_1 is a constant $\in (0, 1)$ and n large enough.*

Proof. Let $X_{\neq 2}$ be the set of nodes with degree $\neq 2$. Because the functions considered here are not promiscuous, we have $|X_{\neq 2}| \geq 0.1 \cdot n/2$. Because there are two times more stubs than nodes, the nodes in $X_{\neq 2}$ receive 2 stubs on average. If one of the nodes receives 0 stubs, the success probability is 0. Otherwise, at least half of the nodes in $X_{\neq 2}$ have to receive exactly 1 stub to satisfy the average. Therefore, l has at least $b = 0.05n/2$ nodes with degree 1. Applying Lemma 7.14 and selecting $c_1 > (7/8)^{0.05} \in (0, 1)$ then concludes the proof. ◀

Now let us turn back to bipartite ShockHash. Here we have two random hash functions l and r that give us a bipartite graph G . Let $(d_1, \dots, d_{n/2})$ and $(d'_1, \dots, d'_{n/2})$ be the degrees of the nodes in the two partitions and remember that \mathcal{B}_n is the distribution of balls in the balls-into-bins process with n balls and $n/2$ bins. Then we can define q as the following probability conditioned on the degree sequence of G :

$$\begin{aligned} q((d_1, \dots, d_{n/2}), (d'_1, \dots, d'_{n/2})) \\ &:= \Pr\left(\text{PF}(G) \mid G \text{ has degree sequence } (d_1, \dots, d_{n/2}), (d'_1, \dots, d'_{n/2})\right) \\ &= \Pr\left(\text{PF}(l, r) \mid l, r \text{ give degree sequence } (d_1, \dots, d_{n/2}), (d'_1, \dots, d'_{n/2})\right) \end{aligned}$$

► **Lemma 7.16.** *For any $d_1, \dots, d_{n/2}, d'_1, \dots, d'_{n/2}$ we have $q((d_1, \dots, d_{n/2}), (d'_1, \dots, d'_{n/2})) \leq p(d_1, \dots, d_{n/2}) \cdot p(d'_1, \dots, d'_{n/2})$.*

Proof. Recall that the peeling process in the configuration model iteratively matches stubs that are the only remaining stub of their node. It can therefore be understood as the process of alternately removing a lonely stub and a random stub (and emitting a corresponding edge). Failure means that a randomly removed stub was lonely. If in the bipartite case we alternate between picking the lonely stubs on the left and on the right, then within each of the two partitions we are alternating between removing a lonely stub and a random stub. Therefore, we are basically running the peeling process within the two partitions separately. This suggests that the probability $q((d_1, \dots, d_{n/2}), (d'_1, \dots, d'_{n/2}))$ that a bipartite graph is a pseudoforest when sampled from the configuration model with degree sequence

$((d_1, \dots, d_{n/2}), (d'_1, \dots, d'_{n/2}))$ is equal to the product of the probabilities $p(d_1, \dots, d_{n/2})$ and $p(d'_1, \dots, d'_{n/2})$ that two graphs are pseudoforests, namely those sampled from the configuration model with degree sequences $(d_1, \dots, d_{n/2})$ and $(d'_1, \dots, d'_{n/2})$, respectively.

However, there is no strict equality due to a slight asymmetry: In the first partition we begin with removing a lonely stub while in the second partition we begin with removing a random stub. This slightly increases the failure probability for the second partition because there is always one lonely stub more when selecting a random stub than there would otherwise be. Note also that the first partition may run out of lonely stubs while the second partition has at least one lonely stub remaining. In that case we would remove a lonely stub from the second partition twice in a row, putting things back on track for the second partition.

Overall, the increase in failure probability for the second partition during (parts of) the process is accounted for by the “ \leq ” in our statement. \blacktriangleleft

Both $p(d_1, \dots, d_{n/2})$ and $\|C(l)\|$ provide a way to rate the quality of a hash function candidate. However, there is a subtle but important difference between the two. While $\|C(l)\|$ looks at the compatible partners in the bipartite case, $p(d_1, \dots, d_{n/2})$ looks at the graph when connected to itself. For example, a hash function candidate that leads to only nodes of degree 2 has $p(2, \dots, 2) = 1$. However, we get $\|C(l)\| < 1$ because l is not compatible with partners that do not hit all output values.

We now formally connect the two probabilities $p(d_1, \dots, d_{n/2})$ and $\|C(l)\|$. This gives a bound for the probability of drawing a compatible partner for a function in $\mathcal{H} \setminus \mathcal{H}_{\text{promisc}}$.

► Lemma 7.17. *Each $l \in \mathcal{H} \setminus \mathcal{H}_{\text{promisc}}$ satisfies $\|C(l)\| < (e/2)^{-n/2} \cdot (c_1)^{n/2}$ where c_1 is a constant $\in (0, 1)$ and n large enough.*

Proof. Take any hash function $l \in \mathcal{H} \setminus \mathcal{H}_{\text{promisc}}$ and its corresponding degree sequence $(d_1, \dots, d_{n/2})$. As a reminder, \mathcal{B}_n is the distribution of balls in the balls-into-bins process with n balls and $n/2$ bins. Then

$$\begin{aligned}
\|C(l)\| &= \Pr(\text{PF}(l, r)) \\
&= \mathbb{E}_{(d'_1, \dots, d'_{n/2}) \sim \mathcal{B}_n} \left(\Pr \left(\text{PF}(l, r) \mid r \text{ has degree sequence } (d'_1, \dots, d'_{n/2}) \right) \right) \\
&= \mathbb{E}_{(d'_1, \dots, d'_{n/2}) \sim \mathcal{B}_n} \left(q((d_1, \dots, d_{n/2}), (d'_1, \dots, d'_{n/2})) \right) \\
&\stackrel{\text{Lem. 7.16}}{\leq} \mathbb{E}_{(d'_1, \dots, d'_{n/2}) \sim \mathcal{B}_n} \left(p(d_1, \dots, d_{n/2}) \cdot p(d'_1, \dots, d'_{n/2}) \right) \\
&= p(d_1, \dots, d_{n/2}) \cdot \mathbb{E}_{(d'_1, \dots, d'_{n/2}) \sim \mathcal{B}_n} \left(p(d'_1, \dots, d'_{n/2}) \right) \\
&\stackrel{(7.5)}{=} p(d_1, \dots, d_{n/2}) \cdot (e/2)^{-n/2} \stackrel{\text{Lem. 7.15}}{\leq} c_1^{n/2} \cdot (e/2)^{-n/2}. \quad \blacktriangleleft
\end{aligned}$$

7.4.5.4 Concentration Bounds

Remember variable $Z = \|\bigcup_{i \in [k]} C(l_i)\|$, which is the probability that a randomly selected function is compatible with a function in our pool. Also remember $C^*(l) = C(l) \setminus \mathcal{H}_{\text{promisc}}$. Unfortunately, we cannot directly give concentration bounds on Z or even calculate $\mathbb{E}(Z)$ because we do not know how using a pool of hash functions influences the probabilities. We therefore look at three additional random variables, defined as follows:

$$Z^* := \left\| \bigcup_{i \in [k]} C^*(l_i) \right\|, \quad \hat{Z} := \sum_{i \in [k]} \|C^*(l_i)\|, \quad D := \sum_{1 \leq i < j \leq k} \|C^*(l_i) \cap C^*(l_j)\|$$

Z^* considers only partners in $\mathcal{H} \setminus \mathcal{H}_{\text{promisc}}$. \hat{Z} is easier to calculate because it looks at each set separately. In the remainder of this section, we then determine bounds on Z^* , \hat{Z} , and D , which we can later use to bound Z .

► **Lemma 7.18.** *Let Z^* , \hat{Z} and D be as defined above. Then $\mathbb{E}(\hat{Z}) > 4/5 \cdot (e/2)^{-n/2}$.*

Proof.

$$\begin{aligned} \mathbb{E}(\hat{Z}) &= \sum_{i \in [k]} \mathbb{E}(\|C^*(l_i)\|) \geq \sum_{i \in [k]} (\mathbb{E}(\|C(l_i)\|) - \|\mathcal{H}_{\text{promisc}}\|) \\ &= k \cdot ((e/2)^{-n} - \|\mathcal{H}_{\text{promisc}}\|) \stackrel{\text{Lem. 7.11}}{\geq} k \cdot \left((e/2)^{-n} - \frac{0.66^n \cdot |\mathcal{H}|}{|\mathcal{H}|} \right) \\ &= (e/2)^{-n/2} - 0.66^n (e/2)^{n/2} = (1 - (0.66 \cdot e/2)^n) \cdot (e/2)^{-n/2} \\ &\geq (1 - 0.9^n) \cdot (e/2)^{-n/2} \geq 4/5 \cdot (e/2)^{-n/2} \text{ for } n \text{ large enough.} \quad \blacktriangleleft \end{aligned}$$

► **Observation 7.19.** *For \hat{Z} , D and Z^* defined as above, it holds that $\hat{Z} - D \leq Z^* \leq \hat{Z}$.*

Proof. To show the bounds, we use the inclusion-exclusion principle, namely that for sets A and B , $|A \cup B| = |A| + |B| - |A \cap B| \leq |A| + |B|$. We can give an upper bound for the variable Z^* by applying the inequality repeatedly using associativity of the union operation.

If we take \hat{Z} and subtract the sizes of all pairwise intersections, we get a lower bound. If all partners were compatible with at most two hash functions in our pool, subtracting the pairwise intersections would give Z^* . However, if a partner is compatible with more than two functions, this subtracts too much, therefore giving the lower bound $\hat{Z} - D \leq Z^*$. ◀

We can now show that $\mathbb{E}(D)$ is exponentially smaller than $\mathbb{E}(\hat{Z})$. Intuitively, this means that $\mathbb{E}(\hat{Z}) \approx \mathbb{E}(Z^*)$ by Observation 7.19. Our proof idea is to bound the intersections using the bound on $\|C(l_i)\|$ shown in Lemma 7.17.

► **Lemma 7.20.** *Let D and \hat{Z} be as defined above. Then $\mathbb{E}(D) \leq (c_2)^{n/2} \cdot \mathbb{E}(\hat{Z})$.*

Proof.

$$\begin{aligned} \mathbb{E}(D) &= \mathbb{E} \left(\sum_{1 \leq i < j \leq k} \|C^*(l_i) \cap C^*(l_j)\| \right) = \binom{k}{2} \mathbb{E}_{l_1, l_2 \sim \mathcal{U}(\mathcal{H})} (\|C^*(l_1) \cap C^*(l_2)\|) \\ &= \binom{k}{2} \frac{1}{|\mathcal{H}|} \mathbb{E}_{l_1, l_2 \sim \mathcal{U}(\mathcal{H})} (|C^*(l_1) \cap C^*(l_2)|) \\ &= \binom{k}{2} \frac{1}{|\mathcal{H}|} \mathbb{E}_{l_1 \sim \mathcal{U}(\mathcal{H})} (\mathbb{E}_{l_2 \sim \mathcal{U}(\mathcal{H})} (|C^*(l_1) \cap C^*(l_2)|)) \\ &= \binom{k}{2} \frac{1}{|\mathcal{H}|} \mathbb{E}_{l_1 \sim \mathcal{U}(\mathcal{H})} \left(\sum_{r \in C^*(l_1)} \Pr_{l_2 \sim \mathcal{U}(\mathcal{H})} (r \in C^*(l_2)) \right) \\ &\leq \binom{k}{2} \frac{1}{|\mathcal{H}|} \mathbb{E}_{l_1 \sim \mathcal{U}(\mathcal{H})} \left(\sum_{r \in C^*(l_1)} \Pr_{l_2 \sim \mathcal{U}(\mathcal{H})} (l_2 \in C(r)) \right) \\ &= \binom{k}{2} \frac{1}{|\mathcal{H}|} \mathbb{E}_{l_1 \sim \mathcal{U}(\mathcal{H})} \left(\sum_{r \in C^*(l_1)} \|C(r)\| \right) \\ &\stackrel{\text{Lem. 7.17}}{\leq} \binom{k}{2} \frac{1}{|\mathcal{H}|} \mathbb{E}_{l_1 \sim \mathcal{U}(\mathcal{H})} (|C^*(l_1)| \cdot (e/2)^{-n/2} \cdot (c_1)^{n/2}) \end{aligned}$$

$$\begin{aligned} &\leq (e/2)^n \mathbb{E}_{l_1 \sim \mathcal{U}(\mathcal{H})}(\|C(l_1)\|) \cdot (e/2)^{-n/2} \cdot (c_1)^{n/2} = (c_1)^{n/2} \cdot (e/2)^{-n/2} \\ &\stackrel{\text{Lem. 7.18}}{\leq} (c_2)^{n/2} \cdot \mathbb{E}(\hat{Z}) \text{ for some } c_2 \in (0, 1) \text{ and } n \text{ large enough.} \end{aligned} \quad \blacktriangleleft$$

Let us now show that \hat{Z} does not get much smaller than its expected value.

► **Lemma 7.21.** *Let Z^* , \hat{Z} and D be as defined above. Then $\Pr(\hat{Z} \geq \mathbb{E}(\hat{Z})/2) > 1 - (c_3)^n$ where c_3 is a constant $\in (0, 1)$ and n large enough.*

Proof. The Bernstein inequality [Ber24] states that for independent and zero-mean random variables V_i with $|V_i| \leq M$, it holds that:

$$\Pr\left(\sum_{i=1}^n V_i \geq t\right) \leq \exp\left(-\frac{\frac{1}{2}t^2}{\sum_{i=1}^n \mathbb{E}(V_i^2) + \frac{1}{3}Mt}\right)$$

To apply the inequality, we now center our variables $\|C^*(l_i)\|$ and mirror them around the value 0, giving us $V_i = \mathbb{E}(\|C^*(l_i)\|) - \|C^*(l_i)\|$. Centering only makes the maximum smaller. Through Lemma 7.12, we can assume that none of our functions l_i is promiscuous. This can be formalized by increasing c_3 accordingly. Therefore, we get $\max_{i \in [k]} (V_i) \leq \max(\|C^*(l_i)\|) \leq \max(\|C(l_i)\|) \leq (c_1)^{n/2} \cdot (e/2)^{-n/2} =: M$ through Lemma 7.17. Before we can get to the Bernstein inequality, we need another ingredient. Let us upper bound the value of $\mathbb{E}((V_i)^2)$ as follows:

$$\begin{aligned} \mathbb{E}((V_i)^2) &= \mathbb{E}\left(\left(\|C^*(l_i)\| - \mathbb{E}(\|C^*(l_i)\|)\right)^2\right) = \mathbb{E}(\|C^*(l_i)\|^2) - \mathbb{E}(\|C^*(l_i)\|)^2 \\ &\leq \mathbb{E}(\|C^*(l_i)\|^2) \leq \mathbb{E}\left(\max_{j \in [k]} \|C^*(l_j)\| \cdot \|C^*(l_i)\|\right) = \max_{j \in [k]} \|C^*(l_j)\| \cdot \mathbb{E}(\|C^*(l_i)\|) \\ &\stackrel{\text{Lem. 7.17}}{\leq} c_1^{n/2} \cdot (e/2)^{-n/2} \cdot (e/2)^{-n} = c_1^{n/2} \cdot (e/2)^{-n/2-n} \end{aligned}$$

Setting $t = 4/10 \cdot (e/2)^{-n/2}$ and applying the Bernstein inequality in the step annotated with *, we get:

$$\begin{aligned} \Pr(\hat{Z} \leq \mathbb{E}(\hat{Z})/2) &= \Pr\left(\sum_{i \in [k]} \|C^*(l_i)\| \leq \mathbb{E}(\hat{Z})/2\right) \\ &= \Pr\left(\sum_{i \in [k]} \mathbb{E}(\|C^*(l_i)\|) - \sum_{i \in [k]} V_i \leq \mathbb{E}(\hat{Z})/2\right) \\ &= \Pr\left(\mathbb{E}(\hat{Z}) - \sum_{i \in [k]} V_i \leq \mathbb{E}(\hat{Z})/2\right) = \Pr\left(\sum_{i=1}^k V_i \geq \mathbb{E}(\hat{Z})/2\right) \\ &\stackrel{\text{Lem. 7.18}}{\leq} \Pr\left(\sum_{i=1}^k V_i \geq 4/10 \cdot (e/2)^{-n/2}\right) \\ &\stackrel{*}{\leq} \exp\left(-\frac{\frac{1}{2}(4/10 \cdot (e/2)^{-n/2})^2}{k \cdot c_1^{n/2} \cdot (e/2)^{-n/2-n} + \frac{1}{3} \cdot c_1^{n/2} \cdot (e/2)^{-n/2} \cdot ((e/2)^{-n/2}/2)}\right) \\ &= \exp\left(-\frac{\frac{2}{25}(e/2)^{-n}}{c_1^{n/2} \cdot (e/2)^{-n} + \frac{1}{6} \cdot c_1^{n/2} \cdot (e/2)^{-n}}\right) \\ &= \exp\left(-12/175 \cdot c_1^{-n/2}\right) = \left(e^{-12/175}\right)^{\left(c_1^{-1/2}\right)^n} \leq (c_3)^n \end{aligned}$$

for some $c_3 \in (0, 1)$ and n large enough. ◀

Lemma 7.20 gives a bound on $\mathbb{E}(D)$ and Lemma 7.21 gives a concentration bound on \hat{Z} . With these insights, we now give a bound on Z^* in terms of $\mathbb{E}(\hat{Z})$.

► **Lemma 7.22.** *Let Z^* , \hat{Z} and D be as defined above. Then $\Pr\left(Z^* < \mathbb{E}(\hat{Z})/4\right) \leq (c_4)^n$ where c_4 is a constant $\in (0, 1)$ and n large enough.*

Proof. We can bound the probability that Z^* deviates too much from $\mathbb{E}(\hat{Z})/4$ as follows. In the step annotated with *, we use the Markov inequality ($\Pr(D \geq a) \leq \mathbb{E}(D)/a$).

$$\begin{aligned}
\Pr\left(Z^* < \mathbb{E}(\hat{Z})/4\right) &\stackrel{\text{Obs. 7.19}}{\leq} \Pr\left(\hat{Z} - D < \mathbb{E}(\hat{Z})/4\right) \\
&\leq \Pr\left(\hat{Z} < \mathbb{E}(\hat{Z})/2 \vee D \geq \mathbb{E}(\hat{Z})/4\right) \\
&\leq \Pr\left(\hat{Z} < \mathbb{E}(\hat{Z})/2\right) + \Pr\left(D \geq \mathbb{E}(\hat{Z})/4\right) \\
&\stackrel{\text{Lem. 7.21}}{\leq} (c_3)^n + \Pr\left(D \geq \mathbb{E}(\hat{Z})/4\right) \stackrel{*}{\leq} (c_3)^n + \frac{\mathbb{E}(D)}{\mathbb{E}(\hat{Z})/4} \\
&\stackrel{\text{Lem. 7.20}}{\leq} (c_3)^n + (c_2)^{n/2}/4 \leq (c_4)^n \text{ for some } c_4 \in (0, 1) \text{ and } n \text{ large enough.} \quad \blacktriangleleft
\end{aligned}$$

7.4.5.5 Combining with Pool $(r_i)_{i \in [k]}$

We can now plug together the previous results, giving us the success probability of bipartite ShockHash when using a pool of size k . With $\Pr(\text{success})$, we denote the probability that there is a pair of compatible hash functions l_i and r_j , $i, j \in [n/2]$ in our pools.

► **Lemma 7.23.** *Let Z be defined as above. Then $\Pr(\text{success} \mid Z = z) = 1 - (1 - z)^k$.*

Proof.

$$\begin{aligned}
\Pr(\text{success} \mid Z = z) &= \Pr\left(\exists i \in [k] : r_i \in \bigcup_{j \in [k]} C(l_j) \mid \left\| \bigcup_{j \in [k]} C(l_j) \right\| = z\right) \\
&= \Pr_{p_1, \dots, p_k \sim \text{Ber}(z)}(\exists i \in [k] : p_i = 1) = 1 - (1 - z)^k. \quad \blacktriangleleft
\end{aligned}$$

We already have most of the proof done. We just need to factor in the probability that the precondition of the previous lemma holds.

► **Theorem 7.24.** *Let us take two pools $(l_i)_{i \in [k]}$ and $(r_i)_{i \in [k]}$ of size $k = (e/2)^{n/2}$ containing randomly sampled hash functions. Then the probability that there are two hash functions l_i and r_j ($i, j \in [k]$) in the pools such that $\text{PF}(l_i, r_j)$ is > 0.17 for n large enough.*

Proof.

$$\begin{aligned}
\Pr(\text{success}) &\geq \Pr\left(\text{success} \wedge Z \geq \mathbb{E}(\hat{Z})/4\right) \\
&= \Pr\left(\text{success} \mid Z \geq \mathbb{E}(\hat{Z})/4\right) \cdot \Pr(Z \geq \mathbb{E}(\hat{Z})/4) \\
&\geq \Pr\left(\text{success} \mid Z \geq \mathbb{E}(\hat{Z})/4\right) \cdot \Pr(Z^* \geq \mathbb{E}(\hat{Z})/4) \\
&= \sum_{z=\mathbb{E}(\hat{Z})/4}^{\infty} \left(\Pr(\text{success} \mid Z = z) \cdot \Pr(Z = z \mid Z \geq \mathbb{E}(\hat{Z})/4)\right) \cdot \Pr(Z^* \geq \mathbb{E}(\hat{Z})/4)
\end{aligned}$$

$$\begin{aligned}
& \stackrel{\text{Lem. 7.23}}{=} \sum_{z=\mathbb{E}(\hat{Z})/4}^{\infty} \left((1 - (1 - z)^k) \cdot \Pr(Z = z \mid Z \geq \mathbb{E}(\hat{Z})/4) \right) \cdot \Pr(Z^* \geq \mathbb{E}(\hat{Z})/4) \\
& \geq \sum_{z=\mathbb{E}(\hat{Z})/4}^{\infty} \left(\left(1 - (1 - \mathbb{E}(\hat{Z})/4)^k\right) \cdot \Pr(Z = z \mid Z \geq \mathbb{E}(\hat{Z})/4) \right) \cdot \Pr(Z^* \geq \mathbb{E}(\hat{Z})/4) \\
& = \left(1 - (1 - \mathbb{E}(\hat{Z})/4)^k\right) \cdot \underbrace{\sum_{z=\mathbb{E}(\hat{Z})/4}^{\infty} \left(\Pr(Z = z \mid Z \geq \mathbb{E}(\hat{Z})/4) \right)}_{=1} \cdot \Pr(Z^* \geq \mathbb{E}(\hat{Z})/4) \\
& \stackrel{\text{Lem. 7.22,7.18}}{>} \left(1 - \left(1 - \frac{(e/2)^{-n/2}}{5}\right)^k\right) \cdot (1 - (c_4)^n) = \left(1 - \left(1 - \frac{1}{5k}\right)^k\right) \cdot (1 - (c_4)^n) \\
& \geq \left(1 - e^{-1/5}\right) \cdot (1 - (c_4)^n) > 0.18 \text{ for } n \text{ large enough.} \quad \blacktriangleleft
\end{aligned}$$

This concludes the proof of combining hash functions from a pool of candidates. We have seen that taking the pools of size $k = (e/2)^{n/2}$ gives us a constant probability that there are two compatible functions in the pools. Note again that our actual implementation uses one single, growing pool, not two fixed size pools. Therefore, we do not need to retry the construction if the initial pool size is not enough, but can just continue adding more functions to the pool.

7.4.6 ShockHash Construction

ShockHash tries different hash function seeds, which is equivalent to generating random graphs. Given the probability that a random graph is a pseudoforest, it is easy to determine the expected number of graphs ShockHash needs to try in order to find an MPHf. This leads directly to the space usage and construction time of ShockHash, which we state in the following theorem.

► **Theorem 7.25.** *A ShockHash minimal perfect hash function mapping n keys to $[n]$ needs $\log(e)n + \mathcal{O}(\log n)$ bits of space in expectation and can be constructed in expected time $\mathcal{O}((e/2)^n \cdot n)$.*

Proof. From Theorem 7.7, we know that the probability of the graph being a pseudoforest is $\geq (e/2)^{-n} e^{-1} \sqrt{\pi}$. We construct these graphs uniformly at random, so the expected number of seeds to try is $\leq (e/2)^n e / \sqrt{\pi}$. The space usage is given by the $n + o(n)$ bits for the retrieval data structure, plus the bits to store the hash function index. In the step annotated with *, we use Jensen's inequality [Jen06] and the fact that log is concave.

$$\mathbb{E}(\log(\text{seed value})) \stackrel{*}{\leq} \log(\mathbb{E}(\text{seed value})) \leq \log\left(\frac{(e/2)^n e}{\sqrt{\pi}}\right) = \log(e)n - n + \mathcal{O}(1).$$

For determining if at least one of the 2^n functions corresponding to such a seed is valid, we can use an algorithm for finding connected components, as described in Section 7.2. This takes linear time for each of the seeds, resulting in an overall construction time of $\mathcal{O}((e/2)^n \cdot n)$. Constructing the retrieval data structure is then possible in linear time [Dil+22] and happens only once, so it is irrelevant for the asymptotic time here. ◀

Looking back at the simple brute-force approach, each of its $e^n / \sqrt{2\pi n}$ expected trials needs n hash function evaluations, leading to a construction time of $\mathcal{O}(e^n \sqrt{n})$. Now, as shown in Theorem 7.25, ShockHash needs time $\mathcal{O}((e/2)^n \cdot n)$. This makes ShockHash almost

2^n times faster than the previous state of the art. Given the observations in Ref. [Bez+23], we conjecture that ShockHash with rotation fitting reduces the number of hash function evaluations by an additional factor of n , while the space overhead tends to zero. In the following Theorem, we now give the resulting construction time and space consumption of the bipartite version.

► **Theorem 7.26.** *A bipartite ShockHash minimal perfect hash function needs $\log(e)n + \mathcal{O}(\log n)$ bits of space in expectation and can be constructed in expected time $\mathcal{O}(1.166^n)$.*

Proof. We know that we need to test $(e/2)^n e/\sqrt{n}$ candidate pairs (h_0, h_1) in expectation before we find a perfect hash function. As described in Section 7.3, instead of sampling two hash functions independently, we use a pool of hash functions and test all combinations of them. Theorem 7.24 shows that if we use a pool size of $k = (e/2)^{n/2} \approx 1.166^n$, we get a constant success probability. Until here, this does not improve the construction time asymptotically because all k^2 combinations need to be tested. However, instead of combining all of the candidates, we can filter them directly while building the candidate pool. The filter, as with plain ShockHash, is very effective: The probability that the n hash values in a partition of size $n/2$ hit all output positions is $\Theta(0.836^{n/2})$ [Wal24]. This means that we are only considering about $((e/2)^{n/2} \cdot 0.836^{n/2})^2 = (e/2 \cdot 0.836)^n \approx 1.136^n$ pairs of hash functions in expectation. The construction time is therefore bounded by $\max\{1.166^n, 1.136^n\}$. Note that both of these values were rounded up anyway, so polynomial factors are dominated.

Looking at the space consumption of bipartite ShockHash, we need to encode two seeds of expected value $\leq k$ each. Using Jensen’s inequality and the retrieval data structure just like in Theorem 7.25, we get the resulting space usage. The fact that we suppressed polynomial factors in the analysis disappears in the $\mathcal{O}(\log n)$ term. ◀

7.5 Partitioning

Even though ShockHash demonstrates significant speedups, by itself, it still needs exponential running time. As mentioned in the introduction, real world MPHFs constructions usually do not search for a function for the entire input set directly. Instead, they partition the input of size N and then search on smaller subproblems of size n . In this section, we now give details on how to partition the input set efficiently before using ShockHash as a building block.

7.5.1 ShockHash-RS = ShockHash + RecSplit

A first option is to integrate ShockHash as a base case into the highly space-efficient RecSplit framework (see Section 4.4) and obtain ShockHash-RS. We keep the general structure of RecSplit intact. Only in the leaves, we use ShockHash instead of brute-force. We store the mapping from its keys to their hash function indices in one large retrieval data structure. Finally, after all leaves are processed, we construct the 1-bit retrieval data structure with all the N entries together.

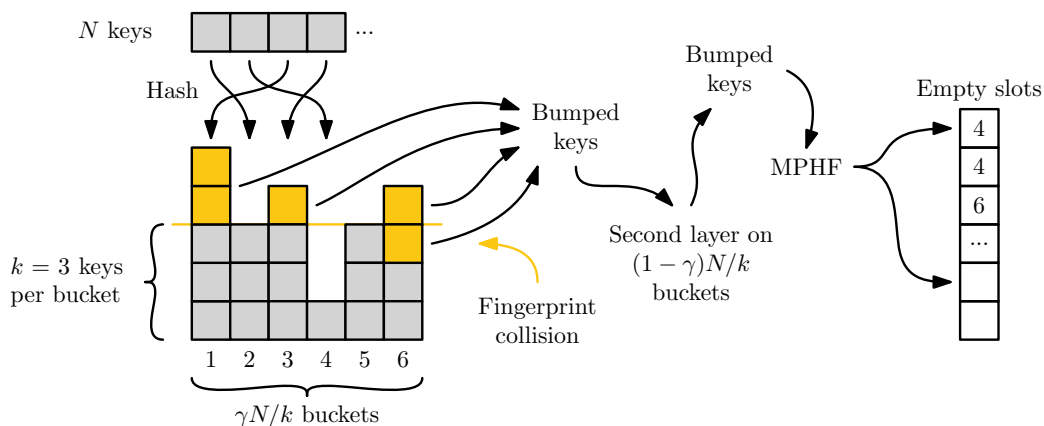
Fanouts. RecSplit tries to balance the difficulty between the splittings and the bijections. ShockHash improves the performance of the bijections significantly but does not modify the way that the splittings are calculated. In this chapter, we focus only on the bijections. To balance the amount of work done between splittings and bijections, we need to adapt the splitting parameters using the same techniques as the RecSplit paper. The RecSplit paper proves and uses optimal fanouts $\lceil 0.35n + 0.5 \rceil$ and $\lceil 0.21n + 0.9 \rceil$ for the two last splitting

levels (see [EGV20, Section 5.4]). For ShockHash-RS, we can adapt their formulas accordingly and get fanouts of $\lfloor 0.10n + 0.5 \rfloor$ and $\lfloor 0.073n + 0.9 \rfloor$. However, preliminary experiments show that this is not optimal in practice. ShockHash is so much faster that the additional time invested into the splittings does not pay off. We find experimentally that setting the lowest splitting level to 4 and the second lowest to 3 achieves much better results in practice. To also provide faster and space-inefficient configurations, we set all fanouts to 2 when selecting leaf size $n \leq 24$.

7.5.2 ShockHash-Flat = ShockHash + k -Perfect Hashing

An additional way to partition the input keys is to use k -perfect hashing. A minimal k -perfect hash function maps N keys to N/k output values, where each output value is hit exactly k times (assuming N divides k). This has applications in external memory data structures [KLS23; LR85] and there are existing constructions [BBD09]. The idea how to integrate ShockHash with k -perfect hashing is straightforward and similar to what we do in ShockHash-RS (see Section 7.5.1). We simply run a two-step process of first determining a k -perfect hash function, and then we construct small ShockHash data structures for the k keys hitting each output value. In contrast to ShockHash-RS (see Section 7.5.1), where some base cases could be smaller than n , here all of them have the same size.

Bumped k -Perfect Hashing. We now briefly describe a new k -perfect hash function. This function is focused on fast queries while still having rather small space consumption. Let us take N keys and hash them uniformly at random to a set of $\gamma N/k$ buckets, $\gamma \in (0, 1]$. By choosing $\gamma < 1$, we can overload the buckets to ensure that most buckets receive at least k keys. In the experiments, we use $\gamma = 0.9$. We handle overflowing buckets by determining a fingerprint of each key. Each bucket then stores a threshold value using $\log(k)$ bits that indicates which of the keys to bump from the bucket. This idea of bumping keys based on a threshold is inspired by bumped ribbon retrieval (BuRR) [Dil+22]. Separator hashing [GL88] uses a similar idea, however, without bumping keys completely and only supporting non-minimal perfect hash functions. Figure 7.7 gives an illustration of the idea. We use a second level of the same data structure for the bumped keys, mapping them to the remaining $(1 - \gamma)N/k$ buckets. Finally, we have a small number of keys that still get bumped in the second level. We first enumerate them by constructing a minimal perfect hash function. In



■ **Figure 7.7** Illustration of our bumped k -perfect hash function.

our implementation, we use ShockHash-RS. With this, we then index an Elias-Fano coded sequence [Eli74; Fan71] (see Section 2.3) storing all empty slots in the output range.

The advantage of this technique is that the majority of queries need a single access to the array of thresholds and a comparison with the key’s threshold. Few need to evaluate the second level, and only a tiny fraction of the queries needs to evaluate the explicit re-mapping.

ShockHash-Flat. From the bumped k -perfect hash function, we derive an MPHf that has a significantly more flat structure than ShockHash-RS. Instead of traversing a tree structure, it can perform a simple comparison with the threshold value for a majority of the input keys. Because we need to access both the threshold and then (usually) the seed of that same bucket, ShockHash-Flat stores thresholds and ShockHash seeds in an interleaved way.

7.6 Variants and Refinements

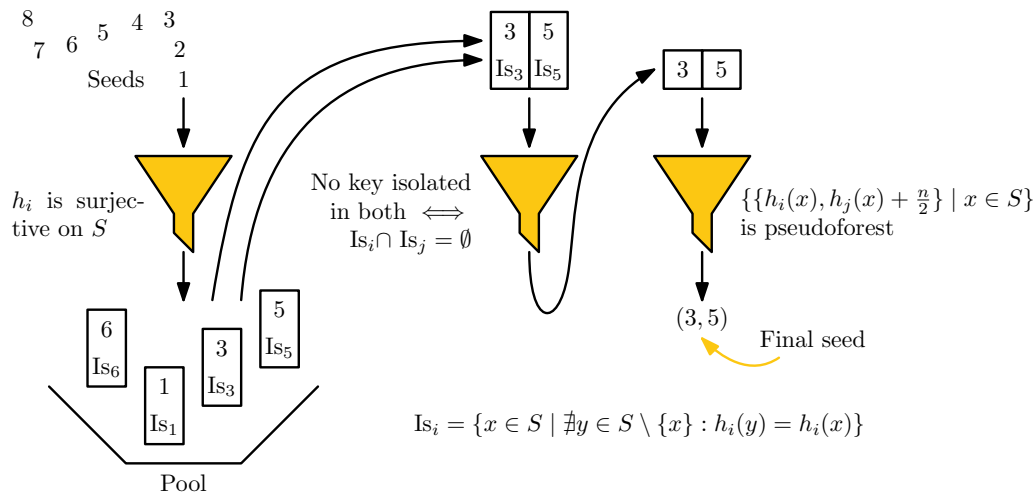
In the following section, we describe variants and implementation details of ShockHash. In Section 7.6.1, we first explain how to achieve significant improvements in practice by using a bit-parallel filter. We then describe two techniques to come up with hash function candidates more efficiently, rotation fitting [Bez+23] (see Chapter 5) and quad split, in Sections 7.6.2 and 7.6.3. To use bipartite ShockHash with uneven input sizes n , only small tweaks are necessary, which we describe in Section 7.6.4. We then continue with practical implementation tricks in Section 7.6.5. Finally, we describe ideas for parallelization in Section 7.6.6.

7.6.1 Isolated Keys Filter

In Section 7.3, we have already described how making the graph bipartite enables efficient filtering of hash function candidates. In the following section, we describe an additional way of filtering seeds. Bipartite ShockHash generates a set of surjective hash function candidates and then tests all combinations for orientability. By using an additional filter, we can speed up this test for orientability. The idea is to look at the case that a key is the only one mapping to a position. We refer to this key as *isolated* for that hash function seed. More formally, a key x is isolated using a hash function candidate h , if $\{y \in S \mid h(x) = h(y)\} = \{x\}$. If a key is isolated in *both* of the candidate hash functions, then in graph terminology this corresponds to a connected component with two nodes and one edge. Since each connected component of the final graph must have the same number of nodes and edges, there is then no need to perform the full test for orientability. We can determine bit patterns for each seed, indicating which of the keys are isolated. Then seed combinations can be ruled out using a simple bit-parallel AND operation checking if a key is isolated in both candidates. Note that the bit patterns used here refer not to the output positions but to the input keys (and therefore have size n). Figure 7.8 illustrates the process.

A key is isolated in a partition if none of the other keys hash to its position, which happens with probability $(1 - 1/(n/2))^{n-1} \rightarrow e^{-2}$. A seed combination passes the filter if it has no key that is isolated in both partitions. This is approximately $(1 - e^{-4})^n \approx 0.98^n$, so the filter makes it possible to avoid the full check for a vast majority of seed combinations. Note that we apply the filter conditioned on the case that both functions are surjective, which should only make the filter more effective.

What makes this method interesting from a theoretical point of view is that we can be even smarter about filtering here. As stated before, if one of the hash functions has an isolated key at a position, we can skip testing it with all other hash functions that have an isolated key at the same position. We can organize all candidate hash functions in a binary



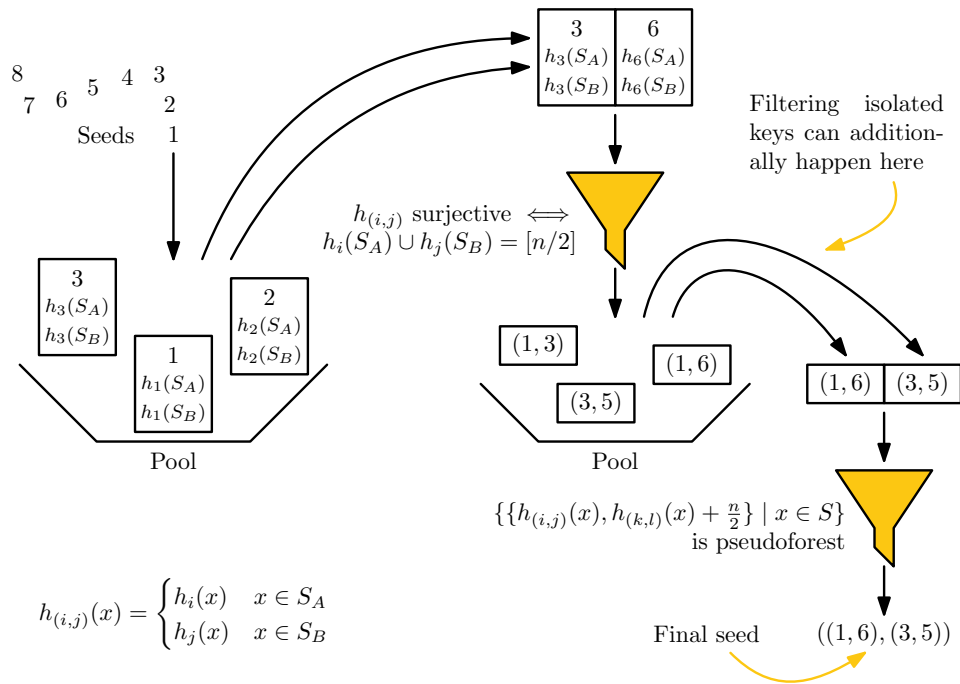
■ **Figure 7.8** Filtering for isolated keys. Each seed in the pool stores a bit vector of isolated keys, here annotated with Is_i . A seed combination can only work if no key is isolated in both partitions, which can be efficiently checked using bit operations. Only if the filter is passed, we need to do the full orientability check.

trie data structure based on the isolated keys. Testing a new candidate hash function now boils down to traversing the trie. In theory, this gives additional exponential improvements in the construction time. However, preliminary experiments show that it is not helpful for the values of n we use in practice.

7.6.2 Rotation Fitting

A technique to speed up brute-force search for perfect hash functions is *rotation fitting* [Bez+23] (see Chapter 5). The same idea can be used in ShockHash to accelerate the search. We distribute the keys to two sets using an unseeded 1-bit hash function. We then determine the bit mask of output values that are hit in both of the sets. Like in the bit mask filter, which we use before checking for orientability (see Section 7.2), only if the logical OR of both masks has all bits set, it is worth testing the seed more closely. If we now cyclically rotate one of the bit masks and try again, we basically get a new chance of all output values being hit, without having to hash each key again. We then consider the distance to rotate the keys as part of the hash function seed. This corresponds to an addition modulo n to all hash values of the second set. We conjecture that – as in Ref. [Bez+23] – this reduces the number of hash function evaluations by a factor of n , while the space overhead tends to zero.

For bipartite ShockHash, rotation fitting can be applied as well, though in a slightly different way. Rotating one of the partitions of the bipartite graph within itself is not useful because it generates isomorphic graphs. Rotating the two partitions into each other would violate the bipartite condition, thus preventing to use the hash function pools. Instead, we can use rotation fitting to find seed candidates within each partition. More specifically, when looking for seed candidates for one partition, we distribute the keys to two subsets using an unseeded 1-bit hash function. We can then rotate one of the sets (modulo $n/2$) to get additional hash function candidates. Each rotation can be tested for surjectivity using simple bit shifts that can happen in registers. In practice, this significantly improves the construction time because fewer hash functions need to be evaluated. However, the quad split technique described in the following section even enables exponential speedups.



■ **Figure 7.9** Additional filtering opportunities in quad split. We create a pool of all seeds without filtering but annotate each seed a bit vector containing hash function output values. Then we combine two seeds to form a seed for one partition. Surjectivity can be checked efficiently using bit operations. Therefore, the resulting seed is a tuple of four seeds – one for each half of the keys and each partition. In our implementation, we also add the filter for isolated keys on top (see Section 7.6.1 and Figure 7.8).

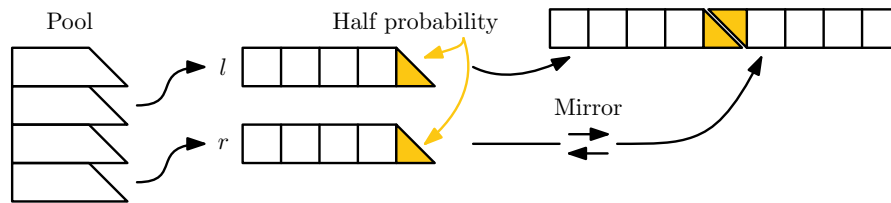
7.6.3 Quad Split

The construction is dominated by the time spent evaluating hash function candidates (see Theorem 7.26), so it is natural to look at this step for improvements. For bipartite ShockHash, the quad split technique reduces the amount of time spent on finding surjective seed candidates. It basically applies the idea of bipartite ShockHash on another level of the same data structure. Like in rotation fitting, we split the input set into two sets S_A and S_B using a constant 1-bit hash function. Now we can hash each of the two sets using *independent* hash functions. In particular, we can test all combinations of assigning some hash function to each of the two sets. This reduces the number of hash function evaluations significantly.

For a seed i , let $h_i(S_A)$ and $h_i(S_B)$ indicate the sets of hash function output values of the two subsets S_A and S_B . A seed i for S_A can be used together with a seed j for S_B if $h_i(S_A) \cup h_j(S_B) = [n/2]$. By storing $h_i(S_A)$ and $h_i(S_B)$ as bit vectors indicating the output values, this compatibility check is a simple and efficient OR operation. In the quad split technique, we therefore annotate each hash function seed with this bit vector to enable fast search for a possible combination of seeds that is surjective. This process is illustrated in Figure 7.9.

Like with the isolated keys filter described in Section 7.6.1, we can again use a trie structure to avoid testing all combinations. We believe that this enables exponential improvements in the construction time, which could be implemented in future work. Quad split is orthogonal to the isolated keys filter, so we can actually combine both optimizations.

To encode the combined seed, we use a pairing function again. In contrast to the bipartite



■ **Figure 7.10** Illustration of how we support uneven n by giving one position half the probability.

tries in ShockHash, the hash functions cannot be exchanged, so we cannot assume that one seed is larger than the other. We therefore need a more general pairing function. The most fitting pairing function here is Szudzik's pairing function (see Section 7.1), which enumerates, for all $k \in \mathbb{N}$, all pairs in $[k] \times [k]$ before moving on to pairs involving numbers bigger than k . This means that we can test all combinations of previous hash functions before having to evaluate the next one. In our implementation, we make sure to try hash function combinations in linear order in the value of the pairing function.

7.6.4 Supporting Uneven n

To support uneven numbers n of input keys, we can relax the bipartite property. The idea is that the output value $\lceil n/2 \rceil$ can be hit by both hash functions, but each with half the probability. When combining the two halves, the value then gets the same probability as all other output values. For filtering candidate hash functions for surjectivity, the corresponding bit needs to be ignored – a seed candidate can be valid both if the bit is set or not set. Now, in order to use hash functions from a single pool for both the left and the right part, we have to mirror the functions used for the right part.⁶ Refer for Figure 7.10 for an illustration.

7.6.5 Engineering

In the following section, we explain implementation tricks that we add to make our construction even faster in practice.

Orientability Check. Determining whether a given graph is a pseudoforest can be achieved in linear time using a connected components algorithm. However, this incurs significant overheads for building a graph data structure. To avoid this, our implementation therefore uses incremental cuckoo hash table construction with near linear time. For this, we keep an array representing the graph nodes. In order to avoid re-calculating a hash every time we evict a key, and to avoid case distinctions between first and second candidate cell, we use the known XOR trick. We annotate each key with the XOR of both its candidate cells. When evicting a key, we can calculate its other candidate cell efficiently by XORing the current cell with the stored value. This uses that XOR is commutative. We abort the insertion as soon as we detect a cycle.

⁶ For simplicity, our current implementation uses plain shifting, accepting a doubled probability for the middle bit. Supporting uneven n then just boils down to rounding compile time constants to the right direction. When integrated into RecSplit, we expect uneven n to happen only once in every two buckets, so it has a negligible overhead of about $1/2b$ bits per key.

Partial Hash Calculation. As described in Section 7.6.2, we can use rotation fitting in plain ShockHash. There we make the observation that hashing the first set of keys almost always yields a graph that, by itself, is a pseudoforest. This is not surprising because the load factor is usually close to the load threshold $c = 0.5$ and n is small (which enables higher load [LSW23b], see Chapter 6). We make use of this fact and reduce the number of hash function evaluations by keeping the hashes for the first set the same and just retrying hash functions for the second set. More precisely, if x is the hash function seed, we hash each key in the first set with seed $x - (x \bmod k)$, where k is a tuning parameter, and the keys of the second set with seed x . Therefore, the hash values of the first set can be cached over multiple iterations. In preliminary experiments, we find a value of $k = 8$ to be a good fit – values much larger than that have diminishing returns in performance improvement and start to influence the space consumption. At $k = 8$, however, the influence on the space consumption is negligible when n is large. Given that hashing the keys is a bottleneck during construction, this reduces the number of keys that need to be hashed by a factor of close to 2. We only apply this optimization for large $n > 32$.

Hash Cache. In bipartite ShockHash, we regularly combine two hash function candidates from our pool to see if the resulting graph is a pseudoforest. While we skip that test for many of the candidates using the simple bit-parallel filter described in Section 7.6.1, there is still a large number of candidates to compare. Re-evaluating the hash functions for these candidates can be a bottleneck depending on the input size n . An obvious idea is to cache the hash function output values of the seed candidates. Because the input sets and therefore the hash values are very small, we can store each hash value in a single byte. This makes the amount of space needed for each seed candidate relatively small.

Sentinels. For large n , the quad split technique spends most of its construction time calculating the logical OR of bit patterns looking for a result that has all bits set. This inner loop consists of only a very small number of assembly instructions. We can achieve considerable speedups here by adding a sentinel element to the end of the array that already has all bits set. Then we no longer need the repeated bounds check for the array. When using SIMD parallelization (see Section 7.6.6), we use multiple sentinels depending on the number of SIMD lanes.

7.6.6 Parallelization

The main computational load behind ShockHash looks for seeds yielding pseudoforests and can be parallelized on multiple levels: Over buckets when using RecSplit for partitioning, ShockHash building blocks, seeds, hash-function evaluations, and bit-parallel filters. The remaining operations are also well parallelizable: Hashing of keys to buckets can be split between processors. Parallel construction of the retrieval data structure can be done similarly [Dil+22]. In the following we explain one possible parallelization with respect to SIMD instructions and multi-threading. We also outline a hybrid CPU/GPU implementation.

SIMD. In plain ShockHash, we use SIMD parallelism in two locations. First, we use SIMD to determine the two candidate positions of all keys and to determine the bit mask for filtering. A key point here is to collect the bitwise OR of individual lanes and to only add the lanes together after all keys are done. Second, we use SIMD to evaluate the bit mask filter (see Section 7.2) with different rotations in parallel. Our implementation uses AVX-512 (8 64-bit values) if available and AVX2 (4 64-bit values) otherwise.

Bipartite ShockHash can also be parallelized using SIMD instructions. When using the quad split technique, we parallelize the test for candidate functions, which involves iterating over long lists of bit patterns, calculating the logical OR with each, and looking for a result that has all bits set. However, the other more involved data structures are harder to parallelize using SIMD because of more complex control flows. Therefore, we use SIMD only for checking lists of bit patterns, which is the main bottleneck for large n .

Multi-Threading. Because ShockHash is intended to be integrated into a partitioning framework, we can naively parallelize over the different ShockHash base cases. A simple coarse-grained source of parallelism are the RecSplit buckets. We can use any kind of load balancing to split them between threads. Even static load balancing may work because variances in construction time will average out. However, some kind of dynamic load balancing is likely to be more efficient and also works with cores of different speed that are now becoming standard in many multi-core processors. The retrieval data structure we use, BuRR [Dil+22], can be parallelized as well.

GPUs. A full GPU parallelization might be difficult and inefficient for cuckoo hashing as it has irregular control flow and memory access. Since filtering asymptotically dominates the computations for highly space-efficient variants, one might look at a hybrid implementation where a GPU produces a stream of seeds defining random graphs that cover all nodes and where a multicore CPU performs further stages of computation. For bipartite ShockHash with quad split, for example, the majority of the construction time is spent on comparing bit patterns to check if two hash function candidates are compatible. Here we can use the massive parallelism of GPUs to compare many patterns in parallel. Then the CPU can perform the less frequent and more complex checks for orientability.

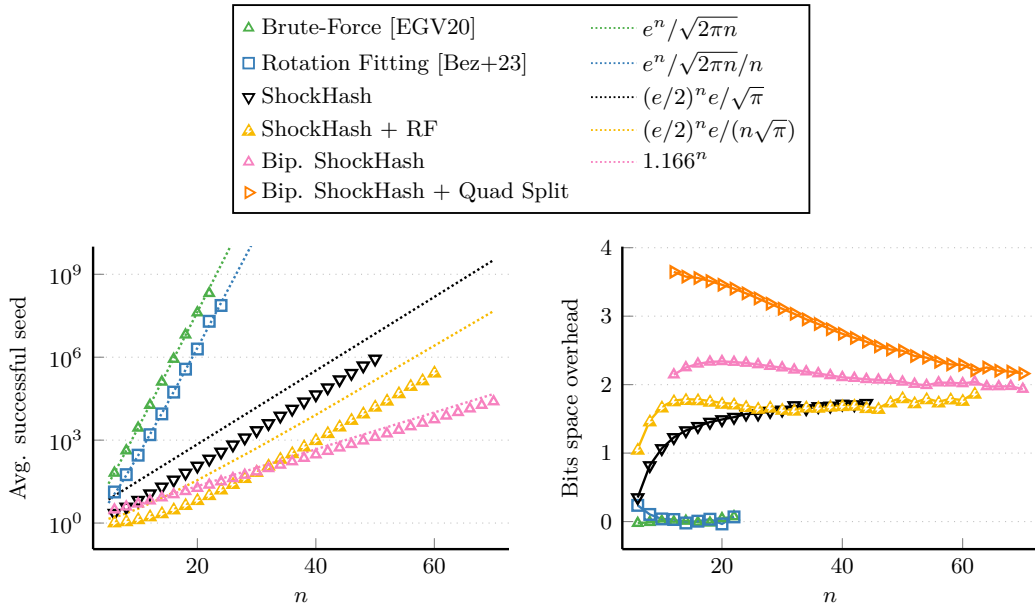
7.7 Internal Experiments

In this section, we give an internal evaluation of our implementation, comparing tuning parameters and variants. For a comparison with competitors, as well as an explanation of the experimental setup, we refer to Chapter 8. For now, it is enough to know that we use a consumer machine and perform our experiments with a single thread. The code and scripts needed to reproduce our experiments are available on GitHub under the GNU General Public License [Leh23c; Leh24].

Our implementation of ShockHash uses the BuRR retrieval data structure [Dil+22] with 128-bit ribbon width and 2-bit bumping information. For ShockHash-RS, we use partitioning based on RecSplit [EGV20], in particular the SIMD-parallel implementation [Bez+23]. For partitioning keys in ShockHash-Flat, we sort them using IPS²Ra [Axt+22].

7.7.1 Number of Trials in Theory and Practice

In Figure 7.11a, we compare the average number of hash function trials for each bijection search technique. From the different slopes of the curves, it is clearly visible that rotation fitting [Bez+23] saves a polynomial factor compared to plain brute-force, while ShockHash saves an exponential factor. Additionally, we plot the shown upper bounds for the number of trials of brute-force and ShockHash. For the rotation fitting variants, we plot the base variants divided by n , which is not formally shown to be a theoretical bound, but is an obvious conjecture. The plot shows that brute-force and rotation fitting are close to the given functions. For plain ShockHash, the measurements are even better than the theory,



(a) Average successful seed. For bipartite ShockHash, we plot the average of the largest seed that was evaluated before pairing up seeds. Because a similar approach would be misleading for quad split due to additional pairing and testing of combinations, we omit it here.

(b) Idealized space overhead over the lower bound $\log(n^n/n!)$ in bits. If the average seed is s we charge $\log(s)$ bits, plus n bits for retrieval (if applicable).

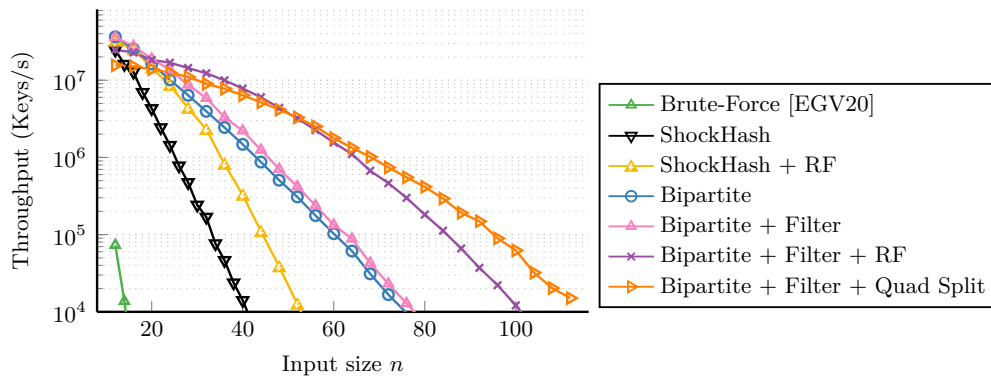
■ **Figure 7.11** Hash function evaluations and space overhead of ShockHash compared with more simple brute-force techniques. The dotted functions are upper bounds. Note that for bipartite ShockHash, we only know an upper bound of $\mathcal{O}(1.166^n)$ and not 1.166^n .

which suggests that our proof in Theorem 7.7 is not tight. Surprisingly, ShockHash seems to match the function we get when dividing our analysis by \sqrt{n} . We conjecture that the expected number of 1-orientations of a random pseudoforest might actually not be $e \cdot \sqrt{2n}$, but close to constant. This makes ShockHash an even better replacement for the brute-force technique. Bipartite ShockHash matches the slope of our analysis as well, but note that our analysis only shows an upper bound of $\mathcal{O}(1.166^n)$, not an exact value.

Figure 7.11b gives the difference between the idealized space consumption and the space lower bound $\log(n^n/n!)$. It indicates that ShockHash loses space close to constant, which becomes negligible for larger n . This explains why we need to select larger n in ShockHash-RS compared to RecSplit with brute-force to achieve the same space consumption per key. Even with these larger n , ShockHash construction is significantly faster than brute-force. Bipartite ShockHash appears to have only a small constant space overhead over plain ShockHash.

7.7.2 Seed Candidate Generation

Figure 7.12 shows different methods to generate seed candidates. For comparison, the plot also includes brute-force search. It is clearly visible that ShockHash is significantly faster than brute-force. Also, the bipartite version shows clear speedups compared to the plain version. Filtering based on isolated keys (see Section 7.6.1) makes the construction about two times faster. While rotation fitting already gives impressive speedups, our quad split technique (see Section 7.6.3) is even faster for large n . Starting with about $n = 60$, the quad



■ **Figure 7.12** Construction throughput using different methods to come up with seed candidates. For comparison, the plot also includes brute-force search. Variants annotated with *RF* use rotation fitting. The filtered variants of bipartite ShockHash use the filter for isolated keys.

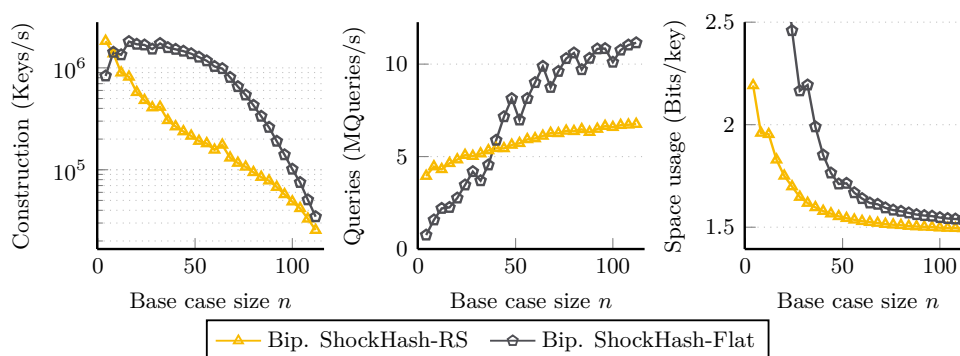
split technique is up to one order of magnitude faster than the basic bipartite ShockHash implementation. Note that the comparison here needs to be taken with a grain of salt because different methods have different space overheads based on n . However, for larger n , the methods have almost the same space usage. For a plot that takes space consumption into account, we refer to the overall evaluation in Chapter 8.

7.7.3 Partitioning

In Figure 7.13, we give the construction time, query time, and space consumption for the two different partitioning schemes, ShockHash-RS and ShockHash-Flat. For small n , the ShockHash-RS construction time is dominated by the splittings, which can be seen by the gap between ShockHash-RS and ShockHash-Flat. For large n , the base case dominates and both techniques have a similar throughput. Looking at the query performance, ShockHash-Flat is faster for $n > 48$, which is the most interesting range for good space consumption. The query throughput increases for larger base cases because we need to spend less time in the partitioning step. The jumps in the query throughput in ShockHash-Flat are caused by the fixed-width coding of ShockHash seeds, which needs a fallback data structure when a seed does not fit. For the same base case size, the space consumption of the flat partitioning scheme is higher than with RecSplit. Compared to ShockHash-RS, ShockHash-Flat trades space consumption for faster queries.

7.8 Summary

ShockHash is a new way to compute minimal perfect hash functions on small sets. By combining trial-and-error search with cuckoo hashing and retrieval data structures, ShockHash achieves an exponential speedup over plain brute-force (almost a factor 2^n). While the plain brute-force technique samples functions and hopes for one to be an MPHf, ShockHash samples graphs and hopes for one to be a pseudoforest. With bipartite ShockHash, we present an extension that samples bipartite graphs and performs aggressive filtering to achieve additional exponential speedups. These improvements enable the currently fastest way to achieve near space-optimal minimal perfect hash functions and breaks the dominance of the previous best methods that relied on pure brute-force for their base-case subproblems.



■ **Figure 7.13** Space usage, construction performance and query performance for different partitioning schemes and base case sizes n . The total number of input keys is $N = 100$ million. As a base case we use bipartite ShockHash with quad split.

We integrate ShockHash as a base case into different partitioning frameworks. When using ShockHash inside RecSplit, we get ShockHash-RS, which can be constructed up to three orders of magnitude faster than the previous state of the art when comparing sequential codes. Constructing with a single thread, ShockHash-RS is even faster than our tuned GPU implementation of the brute-force technique (see Chapter 5). Using ShockHash inside a newly developed k -perfect hash function, we get ShockHash-Flat, which can be constructed about 32 times faster than SIMDRecSplit. At the same time, it has 30% faster queries. This starts to close the gap to constructions that are way less space-efficient and brings space-efficient perfect hash functions closer to practical applications.

8 Practical Comparison of Modern Perfect Hashing

Summary: *In this chapter, we compare different state-of-the-art perfect hash function constructions. The most important properties for their evaluation are construction performance, query performance, and space consumption. Papers about perfect hashing naturally give a focus on their own approach. In this chapter, we explicitly aim at a balanced comparison to provide a thorough overview over the landscape of modern perfect hashing.*

Attribution: The majority of the content featured in this chapter is new. It is inspired by the evaluations of our previous papers [Bez+23; Her+24a; LSW23b; LSW24b] but gives a much more balanced comparison. Also, it uses a new way to plot the three-dimensional measurements.

In this chapter, we compare the performance of state-of-the-art perfect hash function constructions. We give a detailed evaluation that 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 perfect hash function construction – space consumption, construction performance, and query performance. The evaluations in papers often naturally give a focus on their own approach or on a subset of these properties. In this chapter, we explicitly aim at a balanced comparison. It can be used to get a thorough overview over the performance of modern perfect hashing. We do not evaluate k -perfect hash functions or perfect hash functions that are not minimal in order to keep the comparison more focused.

While almost all papers about perfect hashing evaluate the parameters we mention above, they often do so in tabular form. Especially if approaches have a wide range of configurations, this makes it hard to discern the trade-offs that each algorithm offers. We therefore use a visual plot in our papers that shows a large number of configurations at once [Bez+23; Her+24a; LSW23b; LSW24b]. We start this chapter in Section 8.1 with discussing problems with these visualizations and propose a new type of plot. In Section 8.2, we explain our experimental setup. We then present and discuss our actual measurements, focusing on construction performance (in Section 8.3) and query performance (in Section 8.4). Afterwards, we measure how the approaches scale in the input size in Section 8.5 and how they scale in a multi-threaded construction in Section 8.6. We then combine our findings in Section 8.7, discussing representative configurations of each approach.

8.1 Plotting Three-Dimensional Measurements

Each perfect hash function construction provides a range of configurations, leading to different trade-offs. Evaluating multiple configurations for each competitor gives a 3-dimensional point cloud of space consumption, construction performance, and query performance. Because we cannot print a 3D point cloud onto a 2D piece of paper, we need to look at only a subset of the data. In the following, we discuss several ways of making the data readable.

Pareto Fronts. A point is Pareto optimal if there is no other point that dominates it with respect to all three parameters simultaneously. In perfect hashing, it is meaningful to connect these points. The reason is that we can essentially interpolate between different hash function constructions. This works by hashing a certain percentage of the keys to independent perfect hash functions with different configurations. Pareto fronts of three-dimensional data are still three-dimensional. Before we can calculate the Pareto front, we therefore have to apply transformations to the data in order to print them on paper. In the following, we describe several ideas together with advantages and disadvantages.

Projecting to two Dimensions. One way to reduce the data to two dimensions is to project the points onto one of the axis aligned planes. For example, we might want to look at the trade-off between space consumption and query time, while ignoring the construction performance. This approach is easy to understand and already forms an intuition for the data. We use this projection in our papers on perfect hashing [Bez+23; Her+24a; LSW23b; LSW24b]. However, because it ignores one dimension, this approach does not rule out degenerate cases. For example, when ignoring the construction time, the naive brute-force approach would look good in the plot while being entirely unpractical. A second problem emerges when looking at multiple plots with different projections. Next to each other, it looks like there is a correspondence between the points in both plots. However, one point that is Pareto optimal in terms of space consumption and query performance might be dominated when projecting along a different axis. Therefore, the projection might give readers a wrong impression of the performance of an algorithm.

Mixing two Dimensions. An approach that rules out the degenerate cases is to plot a mix of two dimensions. As an example, one could look at the construction time plus the time for $10n$ queries. This can then be a better indicator of the performance in practical applications and is used in some plots in the paper presenting BuRR [Dil+22]. However, the number of queries to perform is quite arbitrary when not considering real applications. Additionally, when construction and query time are very different, one dimension gets much more weight. This happens, for example, when constructing on the GPU and querying on the CPU.

Slicing Along a Dimension. Another approach is to look at a slice of the 3D space. This approach is used in a master’s thesis supervised by the author of this dissertation [Her23]. A problem with this is that it is quite arbitrary which plane should be looked at. Also, it makes it hard to get a feeling for the overall trade-off without plotting a larger number of slices.

Dominance Maps. Finally, we describe a plot that provides a thorough overview over the best perfect hash function constructions. We start similar to the 2D projection and then rasterize the plane into rectangles. We then color each rectangle based on the best competitor along the third dimension. In a way, this therefore provides a “front view” of the Pareto space. BuRR [Dil+22] and SicHash [LSW23b] use a similar type of plot for another purpose. This plot results in a very clean image with fewer data points because it does not print data points for dominated approaches.

8.2 Experimental Setup

We run most of our experiments on an Intel i7 11700 processor with 8 cores (16 hardware threads) and a base clock speed of 2.5 GHz, supporting AVX-512. The machine runs Ubuntu

22.04 with Linux 5.15.0. The L1, L2, and L3 caches have a size of 768 KiB, 3 MiB, and 30 MiB, respectively. We use the GNU C++ compiler version 11.2.0 with optimization flags `-O3 -march=native`. For the competitors written in Rust, we compile in release mode with `target-cpu=native`. The code and scripts needed to reproduce our experiments are available on GitHub under the General Public License [Leh23c]. The repository also contains a Docker image that can be used to reproduce our experiments with low setup time. In total, we measure 3648 data points with a cumulative duration of more than 280 hours.

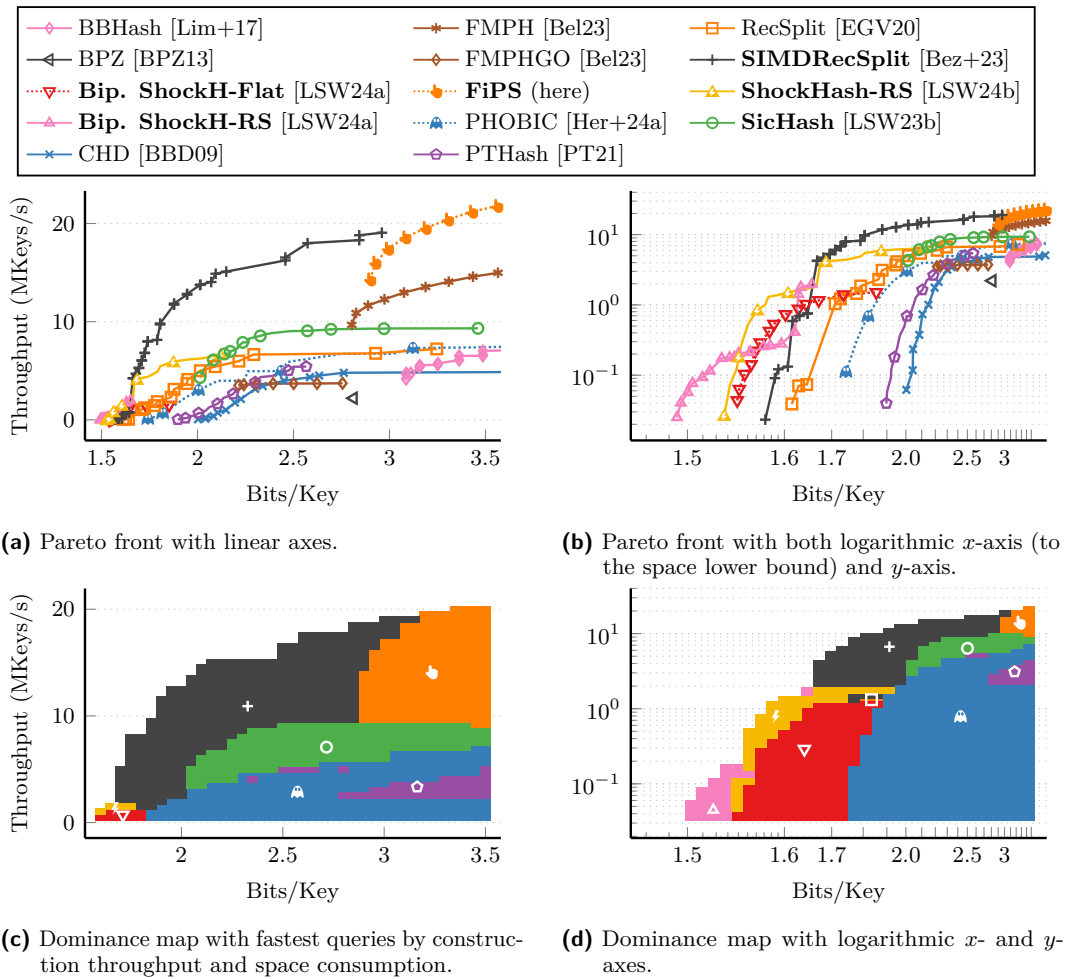
As input data, we use strings of uniform random length $\in [10, 50]$ containing random characters except for the zero byte. Note that, as a first step, almost all competitors generate a *master hash code* of each key using a high quality hash function. This makes the remaining computation largely independent of the input distribution. Arguably, integer inputs would give more consistent measurements because they then do not include the time for running the initial hash function. However, many of the approaches do not directly take a list of integers. They need to be given their own internal data structures to work with hash codes, which could give them an unfair advantage. Because essentially all approaches support string inputs, we use that instead.

Competitors. In our comparison, we include a wide range of perfect hash function constructions. From our own approaches, we include SIMDRecSplit [Bez+23] (see Chapter 5), SicHash [LSW23b] (see Chapter 6), and ShockHash [LSW24a; LSW24b] (see Chapter 7). Additionally, we include PHOBIC [Her+24a], which is based on a master’s thesis supervised by the author of this dissertation. From the literature, we include RecSplit [EGV20], PTHash [PT21], PTHash-HEM [PT24], BPZ [BPZ13] (called BDZ in the *cmph* library), and CHD [BBD09]. Finally, we include different implementations of the fingerprinting idea, namely BBHash [Lim+17], FMPH [Bel23], FMPHGO [Bel23], and our own implementation, FiPS (see Section 4.5). We do not include BMZ [BGZ04] because it needs more than 30 bits per key. We also do not include FCH [FCH92] because the implementation only supports non-minimal perfect hashing. We give more details about the competitors in Chapter 4.

Hash Functions. From the master hash codes we explain above, many competitors derive additional hash values using simpler hash functions. The approaches RecSplit, SIMDRecSplit, ShockHash, and PHOBIC all perform brute-force search over different seeds of the *MurmurHash3 finalizer function* [App10]. For sufficiently random inputs, the function achieves good pseudo-randomness while still being fast to evaluate. More formally, if we assume that we generate our master hash codes using a random function, each hash code has $\Theta(\log n)$ bits of entropy. Each brute-force trial has a very small success probability. Therefore, even if we condition on previous trials being unsuccessful, this does not reduce the entropy too much. The finalizer function is similar to multiply-shift [Die+97], which can be shown to be 2-universal [Tho15]. Refs. [CMV20; MV08] show that 2-universal hash functions are almost indistinguishable from random functions if each new input item has an entropy of $\Theta(\log n)$ bits. This suggests that the use of a simple hash function not only works well in practice, but can also be justified in theory.

8.3 Construction Performance

We now start with the actual comparison, first looking at the single-threaded construction time. Figure 8.1 gives both a Pareto front showing all approaches, and the dominance map described above. We do not explicitly state competitor configurations here because



■ **Figure 8.1** Trade-off of construction time vs space consumption. Single-threaded measurements with $n = 100$ million keys. For some approaches, we only show markers for every fourth point to increase readability.

our experiments include a wide range of configurations of every competitor. However, in Section 8.7, we later give a selection of representative configurations of the competitors. The dominance map plots the space consumption and the construction throughput on the axes, and colors each box with the approach that has the fastest queries. Additionally, because some approaches are focused on very small space consumption, we give a plot that uses logarithmic x - and y -axes. The x -axis is logarithmic to the space lower bound, so 1.44 bits per key would be plotted as $-\infty$. Approaches close to the space lower bound achieve improvements that are large relative to the distance to the bound. The logarithmic axis helps to make this more visible. 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 8.4.

RecSplit/SIMDRecSplit. RecSplit [EGV20] 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 of the same approach. Just like RecSplit,

SIMDRecSplit [Bez+23] is originally designed for small space consumption. We make the surprising observation that SIMDRecSplit not only wins for the most space-efficient configurations, but dominates all the other approaches also 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.

ShockHash. Looking at even more space-efficient approaches, only variants of ShockHash [LSW24b] achieve below 1.55 bits per key. Especially the logarithmic plot shows how ShockHash and bipartite ShockHash improve the space consumption significantly.

Bipartite ShockHash-Flat [LSW24a] trades larger space consumption for better query speed. Its construction is slightly slower than ShockHash-RS. However, we will see later that it has much faster queries. Arguably, a perfect hash function will be queried more often than it is constructed. We therefore now look at approaches achieving even faster queries.

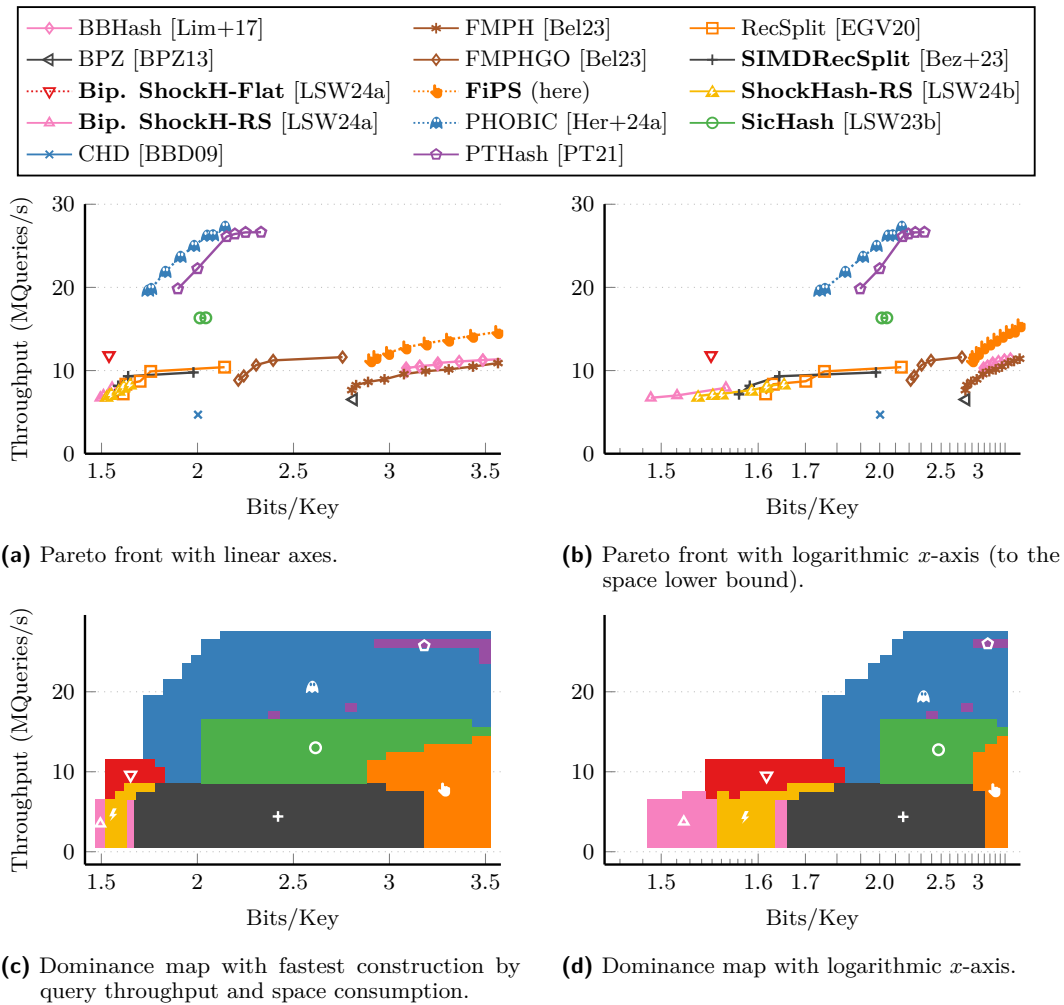
SicHash. SicHash [LSW23b] has a fast construction up to a space consumption of about 2 bits per key. Like most other approaches, its construction throughput stays quite far below SIMDRecSplit. Compared to PTHash and PHOBIC, SicHash is up to two times faster to construct. In essence, SicHash gives a good balance between space consumption, construction performance, and query performance.

Perfect Hashing Through Bucket Placement. We now look at perfect hashing through bucket placement. The construction of PHOBIC [Her+24a], an extension of PTHash [PT21], is consistently faster than the other approaches based on bucket placement. Additionally, the dominance map shows that it has very fast queries. PTHash and CHD [BBD09] have a similar trade-off between construction time and space consumption.

Perfect Hashing Through Fingerprinting. The goal of perfect hashing through fingerprinting [Mü+14] is to offer efficient queries and fast construction at the cost of larger space consumption. BBHash [Lim+17] is the first publicly available implementation. However, it is dominated by more recent implementations of the same approach. FMPH [Bel23] achieves almost twice the construction throughput and lower minimal space consumption. FMPH is written in Rust and opens up perfect hashing to a rather new ecosystem. However, FMPH does not seem to reach the performance of SIMDRecSplit even for fairly large space consumptions. FMPHGO [Bel23] reduces the space consumption of FMPH but also has a much slower construction. Its construction performance is similar to SicHash but has slower queries. Finally, FiPS (see Section 4.5) is our fast implementation of the fingerprinting approach. It achieves a similar space consumption as FMPH but is much faster to construct. FiPS outperforms SIMDRecSplit for space consumptions above 3 bits per key.

8.4 Query Performance

After having focused on the construction time, we now discuss the approaches again, focusing on the query time. Figure 8.2 again includes both a Pareto front and a dominance map. The dominance map plots the space consumption and the query throughput on the axes, and colors each box with the approach that has the fastest construction. The figure also gives a variant with a logarithmic x -axis, but in contrast to Figure 8.1, the y -axis stays linear.



■ **Figure 8.2** Space consumption versus query time of different competitors. $n = 100$ million keys with random single-threaded queries. For some competitors, we only show markers for every fourth point to increase readability.

Perfect Hashing Through Bucket Placement. PTHash [PT21] and PHOBIC [Her+24a] are clear winners in terms of query performance, being more than two times faster than most other competitors. At the same time, they achieve a solid space consumption. PHOBIC improves the space consumption of PTHash without sacrificing query performance. Given that it is also faster to construct, it is almost always the preferred approach over PTHash. This is illustrated by the fact that the dominance map mainly shows PHOBIC. CHD [BBD09] is much slower in terms of queries, mostly due to decoding variable-length seeds.

Perfect Hashing Through Fingerprinting. Perfect hashing through fingerprinting [Mü+14] is originally designed to offer fast queries. Of the different implementations, FiPS offers the best query throughput, being faster than BBHash [Lim+17] and FMPH [Bel23]. However, it is still far away from PTHash and PHOBIC. 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 [Bel23] achieves a smaller space consumption than the other fingerprinting approaches through a small number of brute-force retries. However, it is slower to construct and query than SicHash.

RecSplit/SIMDRecSplit. RecSplit [EGV20] and SIMDRecSplit [Bez+23] are rather 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 BBHash and FMPH. When caring less about query performance, SIMDRecSplit is a good choice because of its very fast construction. This is why it fills the base of the dominance map.

ShockHash. ShockHash-RS [LSW24b] and bipartite ShockHash-RS [LSW24a] use the RecSplit splitting tree as well but need an additional access to a retrieval data structure. However, their query performance is still very close to RecSplit. This shows that the overhead of the retrieval operation is small compared to the work for traversing the heavily compressed tree. Also, the larger leaves mean that the splitting tree has fewer layers to traverse.

Bipartite ShockHash-Flat [LSW24a] is a variant focused on faster queries. For this, it sacrifices some of the space consumption of bipartite ShockHash-RS. It can be constructed about 32 times faster than SIMDRecSplit for the same space consumption. Simultaneously, it achieves 30% faster queries, which brings the query performance of very space-efficient MPHFs much closer to competitors that are not focused on space consumption.

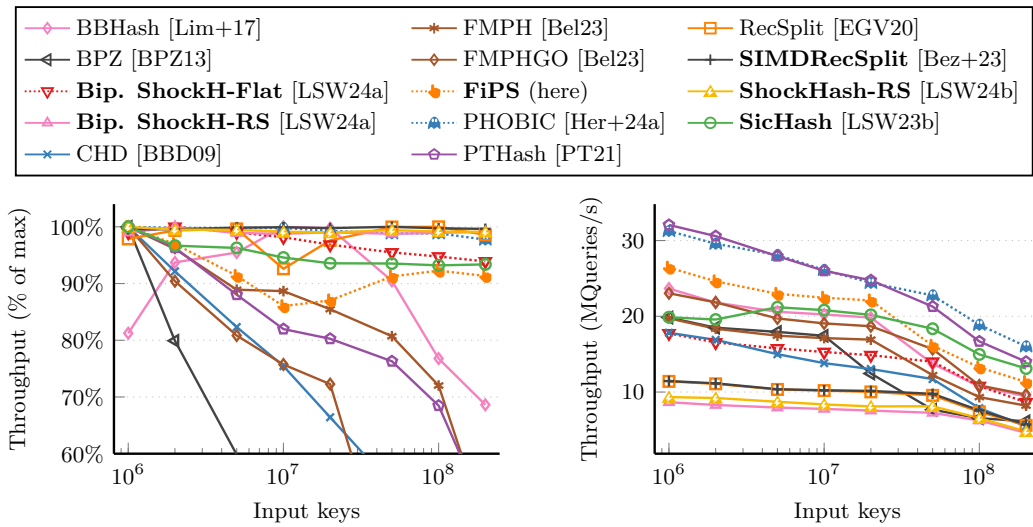
SicHash. SicHash [LSW23b] provides a good middle ground between construction and query performance. As such, it sits in the center of the dominance map. Compared to SIMDRecSplit, it has much faster queries. Compared to PTHash and PHOBIC, SicHash focuses a bit more on construction performance.

8.5 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 hash functions. However, partitioning comes with a performance penalty, both during construction and query. Figure 8.3 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. For each approach, we use a configuration that is typical for it. Therefore, all competitors have a different space consumption and construction time. This is to avoid using unfair configurations that approaches are not designed for. Refer to Section 8.7 to see the exact configurations we use.

Construction. Figure 8.3a shows how the construction throughput scales in the number of input keys n . To be more fair about the fact that competitors are tuned for different space consumption, we give the construction throughput *relative* to each competitor's maximum throughput. For most approaches, the throughput stays almost the same. This can be explained by the fact that many of them perform partitioning internally anyway.

PTHash [PT21], CHD [BBD09], BBHash [Lim+17], BPZ [BPZ13], FMPH [Bel23], and FMPHGO [Bel23] are influenced more strongly by the input size. For PTHash and CHD, which are both based on bucket placement, this can be explained by the fact that the



(a) Construction throughput relative to the maximum of each approach. (b) Query throughput.

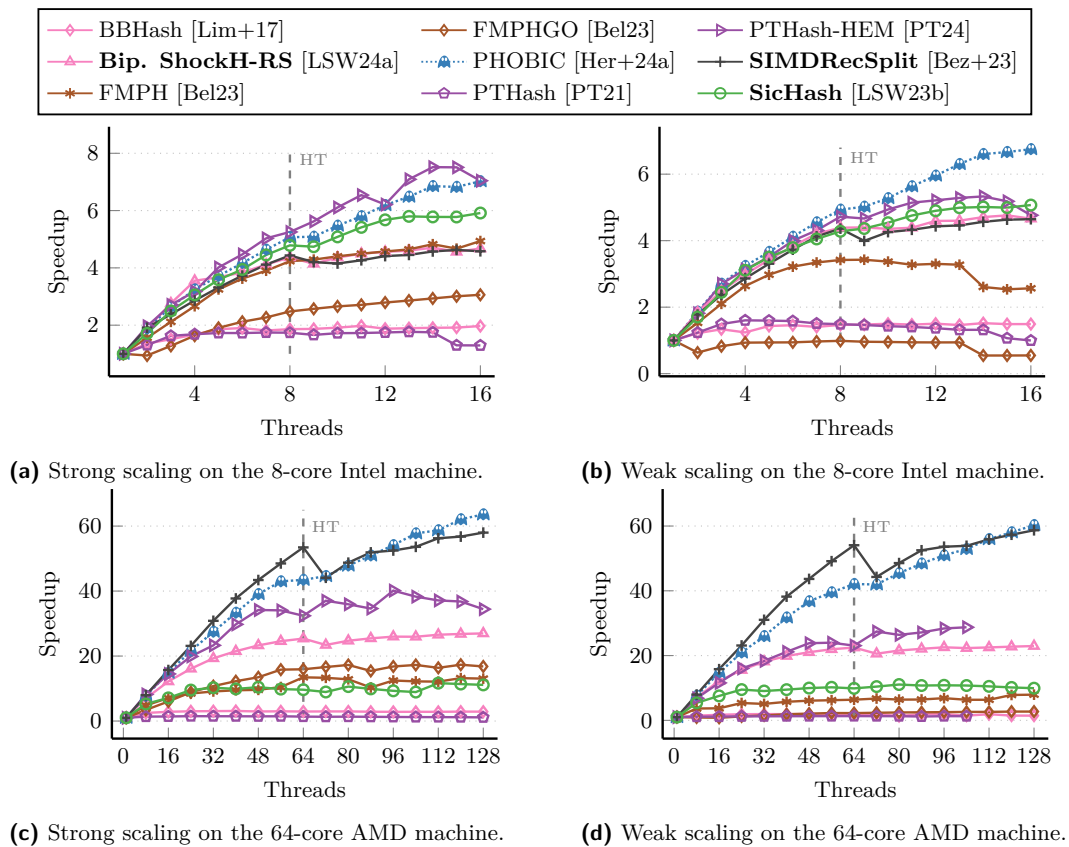
■ **Figure 8.3** Comparison of construction and query performance by number of input keys n . Comparing parameters where each approach has the same space consumption would be unfair because of the vastly different focus of each approach. We therefore use typical parameters of each competitor, and do not directly compare the absolute values of the construction throughput.

bucket placement idea without optimized bucket sizes [Her+24a] inherently has a non-linear construction time. Additionally, the larger bit vectors to detect collisions cause more cache faults. PHOBIC [Her+24a] uses partitioning internally and optimizes the bucket sizes, so it has linear scaling in n . For the approaches based on fingerprinting, namely BBHash, FMPH, and FMPHGO, the slowdown can be explained by the random accesses to the large bit vector. FiPS avoids these cache inefficient access patterns through the use of sorting. Even though it uses perfect hashing through fingerprinting as well, it scales a lot more close to linear. The initial increase of BBHash’s construction throughput can be explained by a constant startup overhead, including it launching a dedicated thread even in single-threaded mode.

Queries. The query throughput of all approaches in Figure 8.3b drops when increasing the input size n . This is expected because the data structures get larger and can be cached less. In general, all approaches perform pretty well regarding their query time. PTHash [PT21] and PHOBIC [Her+24a] remain the approaches with the fastest queries for the entire range of input sizes. However, SicHash [LSW23b] comes close to them for large n , while having a more favorable trade-off between space consumption and construction throughput. RecSplit based approaches (RecSplit [EGV20], SIMDRecSplit [Bez+23], ShockHash-RS [LSW24a; LSW24b]) have much slower queries for the entire range of input sizes. They have a smaller slope, indicating that they are more limited by the computation than memory access.

8.6 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



■ **Figure 8.4** Comparison of the 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.

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. We perform most of our experiments on an Intel processor with 8 cores (16 hardware threads (HT)), supporting AVX512 as mentioned above. For additional experiments in this section, we use a machine with an AMD EPYC 7702P processor with 64 cores (128 hardware threads) and a base clock speed of 2.0 GHz. The machine runs Ubuntu 20.04 with Linux 5.4.0 and supports AVX2. In Figure 8.4, we give parallel measurements on the two machines using both weak scaling with 10 million keys per thread, and strong scaling with 100 million keys. A problem of strong scaling is that with a very large number of threads, the total construction time is very short. Therefore, constant overheads have a significant impact on the construction time. Like in the previous section, we use a typical configuration for each competitor, listed in Section 8.7. As such, the absolute construction times are very different, so we only plot self-speedups.

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. Our

8-core Intel machine has 2 memory channels and our 64-core AMD machine has 8 memory channels. In the following, we first describe approaches with a direct parallelization before looking at the approaches using partitioning.

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. Unfortunately, this generally does not seem to work well for the approaches that do it. We see that BBHash [Lim+17] and PTHash [PT21] only achieve a speedup of around 2 when running on 16 threads. We will later see that PTHash works better with partitioning. Interestingly, even though FMPH [Bel23] also performs a direct parallelization with atomic operations, it scales quite well. FMPHGO [Bel23] uses a direct parallelization as well, even though it could use internal partitioning [Bel23]. Its authors find that the small performance gain is not worth the complexity.

Internal Partitioning. Some of the approaches internally partition the input anyway, so in essence they get their parallelization for free. SIMDRecSplit scales better than PTHash-HEM on the 64-core AMD machine. Note that the scaling behavior of SIMDRecSplit varies strongly depending on the configuration parameters. 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 (see Section 5.6). The speedups of FMPH and SIMDRecSplit remain close to constant for more than 8 threads. PHOBIC [Her+24a] 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.

External Partitioning. Most other approaches implement their parallelization by adding an additional layer of partitioning, which introduces small query and construction time overheads. PTHash-HEM [PT24] scales well on the 8-core Intel machine. SicHash [LSW23b] could theoretically use internal parallelization, so building the hash tables would be possible in parallel. However, the majority of its construction time is spent constructing BuRR retrieval data structures, which does not have a parallel implementation yet. It therefore uses an external partitioning step. On the 8 core machine, PTHash-HEM and SicHash profit from hardware threads during strong scaling. On the 64-core machine, however, they both do not scale well.

GPU Parallelization. Even though this is not the focus of this evaluation, we want to quickly mention two GPU parallel constructions. PHOBIC-GPU [Her+24a] and GPURecSplit [Bez+23] 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 8.4 show that PHOBIC-GPU is faster to query. Refer to the PHOBIC paper [Her+24a] for details.

8.7 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 8.1, we take the two most promising

■ **Table 8.1** Selected configurations of all competitors. Single-threaded, 100 million keys. We use approaches marked with * in Sections 8.5 and 8.6.

	Approach	Configuration	Space bits/key	Construction ns/key	Query ns/query
Brute-force	RecSplit	$n=8, b=100$	1.793	734	118
		$n=14, b=2000^*$	1.584	126 020	135
	SIMDRecSplit	$n=8, b=100$	1.810	110	124
		$n=16, b=2000^*$	1.560	126 465	131
	ShockHash-RS	$n=40, b=2000^*$	1.551	1 860	150
		$n=55, b=2000$	1.526	49 481	147
	Bip. ShockHash-RS	$n=64, b=2000^*$	1.525	5 766	157
		$n=128, b=2000$	1.489	188 661	136
Bip. ShockHash-Flat	$n=64^*$	1.618	1 023	93	
	$n=100$	1.547	9 684	90	
Bucket placement	CHD	$\lambda=3$	2.266	353	224
		$\lambda=5^*$	2.066	2 222	206
	PHash	$\lambda=4.0, \alpha=0.99, C-C$	3.189	291	35
		$\lambda=4.0, \alpha=0.95, EF^*$	2.294	268	60
	PHash-HEM	$\lambda=4.0, \alpha=0.99, C-C$	3.189	292	39
		$\lambda=4.0, \alpha=0.95, EF^*$	2.294	267	64
PHOBIC	$\lambda=6.5, \alpha=1.0, IC, C$	2.338	990	37	
	$\lambda=6.5, \alpha=1.0, IC, Rice^*$	1.847	992	53	
Fingerprinting	BBHash	$\gamma=2.0$	3.710	135	85
		$\gamma=1.5^*$	3.288	172	93
	FMPH	$\gamma=2.0$	3.401	69	95
		$\gamma=1.5^*$	3.013	79	107
	FMPHGO	$\gamma=2.0, s=4, b=16$	2.859	271	81
		$\gamma=1.5, s=4, b=16^*$	2.435	272	95
FiPS	$\gamma=2.0$	3.517	45	69	
	$\gamma=1.5^*$	3.117	51	75	
Graphs	BPZ	$c=1.25, b=3^*$	7.500	451	151
		$c=1.25, b=6$	3.125	450	156
	SicHash	$\alpha=0.95, p_1=37, p_2=44^*$	2.197	148	66
		$\alpha=0.97, p_1=45, p_2=31$	2.080	180	64

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. We use the configurations marked with the * symbol for our evaluations in Sections 8.5 and 8.6.

We first give a selection of configurations using the brute-force approach. 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. During the course of three years, SIMDRecSplit started a chain of work, first improving the space consumption to 1.56 bits per key. ShockHash-RS is then able to achieve 1.52 bits per key, reducing the gap to the space lower bound of ≈ 1.442 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 [Bez+23].

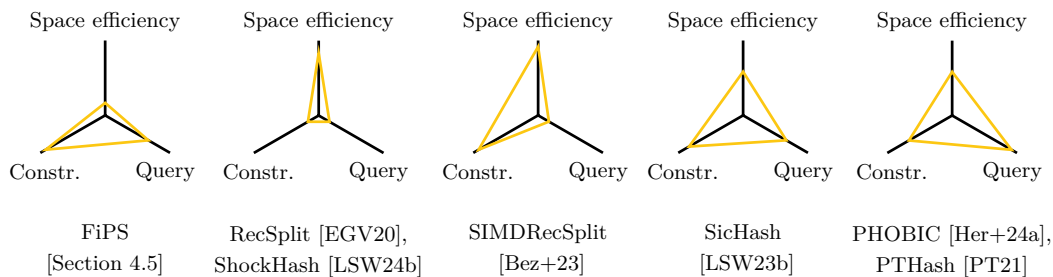
We now look at the approaches based on bucket placement. The partitioned implementation of PTHash, PTHash-HEM, has the same construction time and space consumption when run on a single thread. However, its queries are about 10% slower. Even though PHOBIC uses partitioning as well, with compact coding 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. In Section 2.3 we explain why Golomb-Rice coding is a better choice here.

For perfect hashing through fingerprinting, we give $\gamma = 1.5$ and $\gamma = 2.0$ for all competitors. FMPH achieves the best space consumption. FiPS, with its interleaved rank data structure, needs about 0.1 bits per key more space. BBHash needs an additional 0.17 bits per key. Looking at the construction time and query time, the same configuration with FiPS is consistently faster. Through a small number of retries, FMPHGO achieves much lower space consumption but also much slower construction compared to FiPS.

Finally, we look at two approaches based on orienting random hypergraphs. SicHash is faster than BPZ in construction and queries, while also achieving better space consumption.

8.8 Summary

In this chapter, we perform a wide range of benchmarks of modern perfect hash function constructions. We give plots illustrating the Pareto front of the entire trade-off between construction throughput, query throughput, and space consumption. Different approaches focus on different areas of these parameters. In essence, hash function construction through fingerprinting focuses on fast construction and queries, at the cost of less space-efficient



■ **Figure 8.5** Focus of different perfect hashing approaches. Shows only qualitative idea of typical configurations of approaches, not actual measurements.

representation. FiPS is a fast implementation of the approach. 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 method is bipartite ShockHash-RS [LSW24a], which massively reduces the brute-force search space. SIMDRecSplit [Bez+23] is an exception because it achieves good construction performance, even though it is tuned for small space consumption. SicHash [LSW23b] gives a good balance between all three parameters with a slight focus on construction performance. Regarding query time, PHOBIC [Her+24a] offers the best performance. In Figure 8.5, we give a qualitative idea of the trade-off of the most promising approaches.

Overall, the evaluation shows the progress of modern perfect hashing. While we include many of the classical results, they cannot keep up with the more recent implementations. The approaches presented in this dissertation cover the vast majority of the Pareto front, which confirms the success of the techniques presented here and, more generally, the merits of the Algorithm Engineering methodology.

9 Learned Monotone Minimal Perfect Hashing

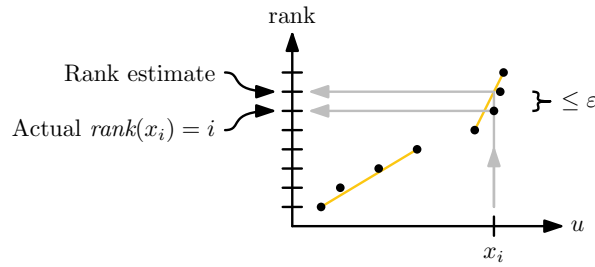
Summary: *A Monotone Minimal Perfect Hash Function (MMPHF) constructed on a set S of keys is a data structure that maps each key in S to its rank. On keys not in S , it returns an arbitrary value. Applications range from databases, search engines, data encryption, to pattern-matching algorithms.*

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

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

Attribution: This chapter is based on “Learned Monotone Minimal Perfect Hashing” [Fer+23a]. Large parts of this chapter are copied verbatim or with minor changes from that publication. Giorgio Vinciguerra and the author of this dissertation implemented the algorithm together with equal contributions. The author of this dissertation wrote the initial implementation, initial manuscript and performed the experiments. All authors made significant contributions, to algorithm design, analysis, design & interpretation of the experiments, and the write-up. We thank Stefan Walzer for early discussions leading to the paper.

Given a set S of n keys drawn from a universe $[U]$, a *Monotone Minimal Perfect Hash Function* (MMPHF) is a data structure that maps keys from S to their rank, and returns an arbitrary value for keys not in S . We refer to Section 1.1 for details. With LeMonHash, we offer a fresh new perspective on MMPHFs that departs from existing approaches, which are mostly based on a trie-like data structure on the keys. We build upon recent advances in (learning-based) indexing data structures, namely the PGM-index [FLV21; FV20b], and in retrieval data structures, namely BuRR [Dil+22] (see Section 2.4). The former learns a piecewise linear approximation mapping keys in S to their rank estimate. The latter allows associating a small fixed-width integer to each key in S , without storing S . We combine these two seemingly unrelated data structures in a surprisingly simple and effective way. First, we use the PGM to monotonically map keys to buckets according to their rank estimate, and we



■ **Figure 9.1** Illustration of the PGM-index. Here, it gives a wrong rank estimate for x_i .

store the global rank of each bucket’s first key in a compressed way. Second, since the rank estimate of some keys might coincide, we solve such bucket collisions by storing the local ranks of these keys using BuRR. We call our proposal *LeMonHash*, because it *learns* and *leverages* the smoothness of the input data to build a space-time efficient *monotone* MPHf.

In this chapter, we first discuss related work on MMPHF in Section 9.1. In Section 9.2, we describe LeMonHash for integers and extend it to support variable-length string keys in Section 9.3. In Section 9.4, we discuss variants and refinements, before proving the space-time guarantees of LeMonHash in Section 9.5. In Section 9.6, we present a detailed evaluation.

PGM-index. The PGM-index [FLV21; FV20b] (Piecewise Geometric Model) is a space-efficient data structure for predecessor and rank queries on a sorted set of n keys from an integer universe $[U]$. Given a query $q \in [U]$, it computes a rank estimate that is guaranteed to be close to the correct rank by a given integer parameter ϵ . If one stores the input keys, then the correct rank can be recovered via an $\mathcal{O}(\log \epsilon)$ -time binary search on $2\epsilon + 1$ keys around the rank estimate. The PGM is constructed in $\mathcal{O}(n)$ time by first mapping the sorted integers x_1, \dots, x_n in S to points $(x_1, 1), \dots, (x_n, n)$ in a key-position Cartesian plane, and then learning a piecewise linear ϵ -approximation of these points, i.e. a sequence of m linear models each approximating the rank of the keys in a certain sub-range of $[U]$ with a maximum absolute error ϵ . Figure 9.1 illustrates the idea. The value m , which impacts on the space of the PGM, can range between 1 and $m \leq n/(2\epsilon)$ [FV20b, Lemma 2] depending on the “approximate linearity” of the points. In practice, it is very low and can be proven to be $m = \mathcal{O}(n/\epsilon^2)$ when the gaps between keys are random variables from a proper distribution [FLV21]. The time complexity to compute the rank estimate with a PGM is given by the time to search for the linear model that contains the searched key q , which boils down to a predecessor search on m integers from a universe of size U . For this, there exist many trade-offs in various models of computations [FV20b; NR21].

9.1 Related Work

In the following, we look at related work on monotone minimal perfect hash functions, first describing the idea of bucketing before then continuing with specific MMPHF constructions. For non-monotone perfect hash functions, we refer to Chapter 4. We do not review order-preserving minimal perfect hash functions (see Section 1.1) because their theoretical lower bound can trivially be reached by using a retrieval data structure taking $\log n$ bits per key (plus a small overhead). A loosely related result is using learned models as a replacement for hash functions in traditional hash tables [Kra+18; Sab+22], but it generally has a negative impact on the probe/insert throughput (and most likely on the space too, due to the storage of the models’ parameters, which these studies do not evaluate).

Bucketing. Bucketing [Bel+11] is a general technique to break down MMPHF construction into smaller sub-problems. The idea is to store a simple monotone, but not necessarily minimal or perfect *distributor* function that maps input keys to buckets. Each bucket receives a smaller number of keys that can then be handled using some (smaller) MMPHF data structure. To determine the global rank of a key, we need the prefix sum of the bucket sizes. For equally-sized buckets, this is trivial. Otherwise, this sequence can be stored with Elias-Fano coding (see Section 2.3). In the paper by Belazzougui et al. [Bel+11], where many of the following techniques are described, the authors use MWHC [Maj+96] to explicitly store the ranks within each bucket. LeMonHash uses a learned distributor and buckets of expected size 1 (see Section 9.2).

Longest Common Prefix. Bucketing with Longest Common Prefixes (LCP) [Bel+09] maps keys to equally sized buckets. A first retrieval data structure maps all keys to the *length* of the LCP among all keys in its bucket. A second one then maps the *value* of the LCP to the bucket index. Overall, it uses $\mathcal{O}(\log \log U)$ bits per key and query time $\mathcal{O}((\log U)/w)$, and in practice it has been shown to be the fastest but the most space-inefficient MMPHF [Bel+11].

Partial Compacted Trie. First map the keys to equally sized buckets and consider the last key of each bucket as a *router* indexed by a *compacted trie*, e.g., a binary tree where every node contains a bit string denoting the common prefix of its descending keys. During queries, the trie is traversed by comparing the bit string of the traversed nodes with the key to decide whether to stop the search operation at some node (if the prefix does not match), or descend into the left or right subtree based on the next bit of the key. A *Partial Compacted Trie* (PaCo Trie) [Bel+11] compresses the compacted trie above by 30–50% by exploiting the fact that, in an MMPHF, the trie needs to correctly rank only the input keys. Therefore, each node can store a shorter bit string just long enough to correctly route all input keys.

Hollow Trie. A *Hollow Trie* [Bel+11] only stores the *position* of the next bit to look at. Hollow tries can be represented succinctly using balanced parentheses [MR01]. To use hollow tries for bucketing, and thus allow the routing of not-indexed keys, we need a modification to the data structure. The *Hollow Trie Distributor* [Bel+11] uses a retrieval data structure that maps the compacted substrings of each key in each tree node to the behaviour of that key in the node (stopping at the left or right of the node, or following the trie using the next bit of the key). Overall, it uses $\mathcal{O}(\log \log \log U)$ bits per key and query time $\mathcal{O}(\log U)$.

ZFast Trie. To construct a *ZFast Trie* [Bel+09], we first generate a path-compacted trie. Then, for prefixes of a specific length (*2-fattest number*) of all input keys, a dictionary stores the trie node that represents that prefix. A query can then perform a binary search over the length of the queried key. If there is no node in the dictionary for a given prefix, the search can continue with the pivot as its upper bound. If there is a node, the lower bound of the search can be set to the length of the longest common prefix of all keys represented by that node. The ZFast trie uses $\mathcal{O}(\log \log \log U)$ bits per key and query time $\mathcal{O}((\log U)/w + \log \log U)$.

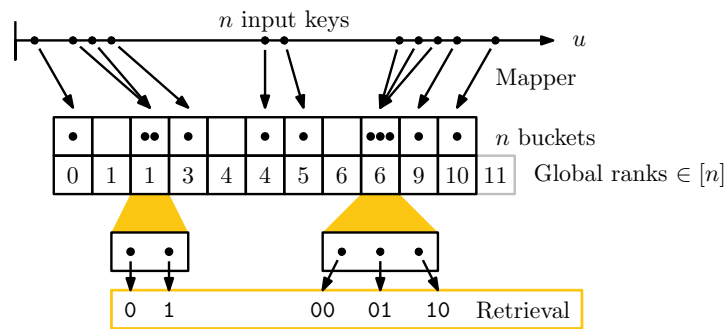
Path Decomposed Trie. In the previous paragraphs, we described binary tries with a rather high height. However, those tries are inefficient to query because of the pointer chasing to non-local memory areas. The main idea behind *Path Decomposed Tries* [Fer+08], which can be used as an MMPHF [GO14], is to reduce the height of the tries. We first select one path all the way from the root node to a leaf. This path is now contracted to a single node, which

becomes the root node in our new path decomposed trie. The remaining nodes in the original trie form subtrees branching from every node in that path. We take all of these subtrees, make them children of the root node, and annotate them by their branching character with respect to the selected path. The subtrees are then converted to path decomposed tries recursively. In *centroid* path decomposition, the path to be contracted is always the one that descends to the node with the most leaves in its subtree.

9.2 LeMonHash

We now introduce the main contribution of this chapter – the MMPHF LeMonHash. The core idea of LeMonHash is surprisingly simple. We take all the n input integers and map them to n buckets using some *monotone* mapping function, that we will describe later. We store an Elias-Fano coded sequence with the *global ranks* of the first key in each bucket using $2n + o(n)$ bits (see Section 2.3). Given a bucket of size b , we use a $\lceil \log b \rceil$ -bit retrieval data structure (see Section 2.4) to store the *local ranks* of all its keys. Note that we do not need to store local ranks if the bucket has only 0 or 1 keys. For squeezing space, instead of storing one retrieval data structure per bucket, we store a collection of retrieval data structures so that the i th one stores the local ranks of all keys mapped to buckets whose size b is such that $i = \lceil \log b \rceil$. We give an illustration of the overall data structure in Figure 9.2.

Bucket Mapping Function. The space efficiency of LeMonHash is directly related to the quality of the monotone mapping function. For uniform random integers, a linear mapping from input keys to n buckets, i.e. a mapping from a key x to the bucket number $\lfloor xn/U \rfloor$, leads to an MMPHF with a space consumption of just 2.915 bits per key (see Theorem 9.1). Intuitively, such a linear mapping returns a rank estimate in $[n]$ for a given key. However, for skewed distributions, the rank estimate can be far away which can create large buckets whose local ranks are expensive to store. For example, if the majority of the keys are such that $x < U/n$, then the first bucket will be large enough to require $\Theta(\log n)$ bits per key, i.e. our MMPHF degenerates to a trivial OPMPHF. To tackle this problem, we implement the mapping function with a *PGM-index* [FV20b]. As we mention in the introduction of this chapter, the PGM is originally designed as a predecessor-search data structure. Here, we use the PGM as a *rank estimator* that, for a given key, returns an ε -bounded estimate of its rank. To achieve this result in LeMonHash, we do not store the list of indexed keys and simply use the PGM’s rank estimate as the bucket index. The PGM internally adapts to the input data by learning the smoothness in the distribution via a piecewise linear ε -approximation model, thus it can be thought of as a “local” approximation of the linear mapping above. Real-world data sets can often be approximated using piecewise linear models, as discussed in the literature [FLV21] and also demonstrated by the good space efficiency of our experiments (see Section 9.6). There is a trade-off between the amount of space needed to represent the PGM and the quality of the mapping, which depends on both the input data distribution and the given integer parameter ε . In Section 9.6, we test both a version with a constant ε value and a version that auto-tunes its value by constructing multiple PGMs and then selecting the optimal ε . Finally, we observe that with the PGM mapper, unlike for the linear mapping and other non error-bounded learning-based approaches [FV20a; KRT22], the number of retrieval data structures we need to keep is bounded by $\mathcal{O}(\log \varepsilon)$ regardless of the input key distribution (see Theorem 9.2).



■ **Figure 9.2** Illustration of the LeMonHash data structure. Keys are mapped to buckets. Ranks within buckets are stored in (a collection of) retrieval data structures.

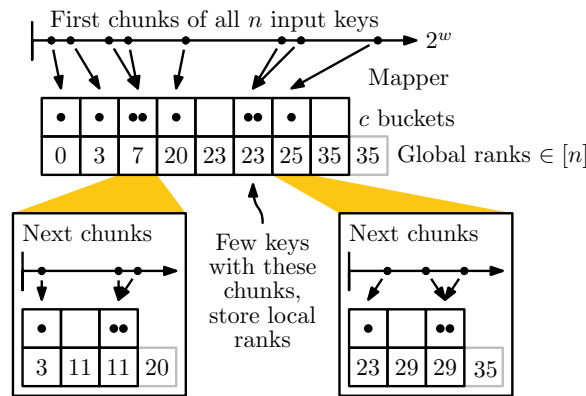
Queries. Given a key q , we obtain its bucket i using the mapping function. The global rank of the (first key in the) bucket is the i th integer in the Elias-Fano coded sequence of global ranks, which can be accessed in constant time, and the bucket size is computed by subtraction from the next integer in that sequence. The bucket size b directly tells us which retrieval data structure to query, i.e. the $\lceil \log b \rceil$ th one. Evaluating the retrieval data structure with q gives us its local rank in the bucket. Adding this to the global rank of the bucket gives us the rank of q . As we show in Section 9.5, for uniform data, the linear bucket mapper gives constant time queries, while for other inputs we use the PGM mapper and the query time is $\mathcal{O}(\log \log U)$.

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

9.3 LeMonHash-VL

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

Overview. We start with a root node representing all the string keys in S and consider the set of chunks extracted from each key starting from position $|p|$ (which we store), where p is the LCP among the keys in S . Given these c distinct chunks, we construct a PGM mapper to distribute the keys to buckets in $[c]$, and we store an Elias-Fano coded sequence with the



■ **Figure 9.3** Illustration of the LeMonHash-VL data structure. Global ranks in each level are stored together. Buckets that are not handled recursively use retrieval data structures like before.

global ranks of the first key in each bucket. Clearly, different keys can be mapped to the same bucket because the PGM mapper is not perfect (as in the integer case) and because they share the same chunk value (unlike in the integer case). For example, for the strings $S = \{\text{cherry, cocoa, coconut}\}$ with $p = c$ and chunks composed of 3 characters, the keys `cocoa` and `coconut` share the chunk value `oco` and will be mapped to the same bucket.

If a bucket of size b contains fewer input strings than a specific threshold t , we store the local ranks of the strings in the bucket in a $\lceil \log b \rceil$ -bit retrieval data structure. Once again, we do not need to store local ranks if the bucket has only 0 or 1 keys. If instead the bucket is large (i.e. $b \geq t$), we create a *child node* in the tree data structure by applying the same idea recursively on the strings S' of that bucket. This means that we compute a PGM mapper on the chunks extracted from each string in S' starting from position $|p'|$, where p' is the LCP among the bucket strings S' . Notice that $|p'| \geq |p|$ but we always guarantee that $S' \subsetneq S$, so the recursion is bounded. In practice, we set the threshold $t = 128$ (see Section 9.6.1).

At query time, we can use the sequence of global ranks to calculate the bucket size b , which allows determining whether we need to continue recursively on a child (because $b \geq t$) or directly return the global rank of the bucket plus the local rank stored in the $\lceil \log b \rceil$ -bit retrieval data structure. Figure 9.3 gives an overview of the data structure.

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

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

Given the overall idea, there is a wide range of optimizations that we use. In the following, we outline the main algorithmic ones and refer the interested reader to our implementation [LV23] and Section 9.4 for the many other optimizations, such as the use of specialized instructions like `popcount` and `bextr`, or lookup tables.

Alphabet Reduction. The number of nodes and the depth of LeMonHash-VL depend on both the length and distribution of the input strings, and on how well the PGM mapper at each node can map strings to distinct buckets given their w -bit chunks. Therefore, we

should aim to fit as much information as possible in the w -bit chunks. We do so by exploiting the fact that, in real-world data sets, often only a very small alphabet Σ of branching characters distinguish the strings in each bucket, and that we do not care about the other characters. We extract chunks from the suffix of each string starting from the position following the LCP p , as before, but interpret the suffix as a number in radix $\sigma = |\Sigma|$ where each character is replaced by its 0-based index in Σ if present, or by 0 if not present. For example, for a node on the strings `{shoppers, shopping, shops}` whose LCP is $p = \text{shop}$, we would store the alphabet $\Sigma = \{\mathbf{e}, \mathbf{i}, \mathbf{p}, \mathbf{s}\}$ and map the suffix “pers” of “shoppers” to $\text{index}(\mathbf{p})\sigma^3 + \text{index}(\mathbf{e})\sigma^2 + \text{index}(\mathbf{r})\sigma^1 + \text{index}(\mathbf{s})\sigma^0 = 2\sigma^3 + 0\sigma^2 + 0\sigma^1 + 3\sigma^0$. Observe that the chunks computed in this way still preserve the lexicographic order of the strings. The number of characters we extract is computed to fit as many characters as possible in a w -bit word, i.e. $\lfloor w/\log \sigma \rfloor$ characters. In our implementation over bytes, we store Σ via a bitmap of size 128 or 256, depending on whether its characters are a subset of ASCII or not. Finally, we mention that a mapping from strings to numbers in radix σ has also been used to build compressed string dictionaries [Bof+22], but the twist here is that we are considering only the alphabet of the branching characters since we do not need to store the keys.

Elias-Fano Sequences. The large per-level Elias-Fano sequences of global ranks have a very irregular structure. For example, if many of the strings in a node share the same chunks, there is a large gap between two of the stored ranks. We can deal with these irregularities and reduce the overall space consumption by using partitioned Elias-Fano [OV14]. Furthermore, the PGM mappers do not always provide a very uniform mapping, which thus results in empty buckets. An empty bucket corresponds to a duplicate offset value being stored in the Elias-Fano sequences (see e.g. the duplicate offset 23 in Figure 9.3). To optimize the space consumption of such duplicates, we filter them out before constructing the partitioned Elias-Fano sequence. We do this by grouping the stored numbers in groups of 3 numbers. If all 3 numbers are duplicates of the number before that group, we do not need to store the group. A bit vector with rank support indicates which groups were removed.

Perfect Chunk Mapping. In many datasets, there might be only a small number of different chunks, even if the number of strings they represent is large. For instance, chunks computed on the first bytes of a set of URLs might be a few due to the scarcity of hostnames, but each host may contain many distinct pages. In these cases, instead of a PGM, it might be more space-efficient to build a (perfect) map from chunks to buckets in $[c]$ via a retrieval data structure taking $c\lceil \log c \rceil$ bits overall (plus a small overhead), where c is the number of distinct chunks. In practice, we apply this optimization whenever $c < 128$ (see Section 9.6.1).

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

9.4 Variants and Refinements

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

External Memory Construction. To construct the PGM-index with a specific ε value, a single scan over the input data is sufficient. As soon as one of the segments is constructed, the corresponding keys can be mapped to buckets and the input for the retrieval data structures can be generated. The retrieval data structures can be constructed in external memory as well [Dil+22]. The construction of LeMonHash can therefore be performed entirely in external memory. External memory queries are possible by selecting a suitable data structure for predecessor queries inside the PGM-index (such as the recursive structure in [FV20b]), as well as an external-memory encoding of the bucket sizes. LeMonHash-VL can be constructed and queried in external memory using similar considerations. While the recursion needs additional passes over the input data, note that the construction is performed in depth-first order, so it can profit from the locality between different levels.

Parallel Construction. As described in [Bel+11], it is easy to divide any MMPHF into multiple buckets (see Section 9.1). The buckets can then be constructed independently in parallel, but this naive construction introduces some query overhead due to adding another layer on top of the data structure. Instead, the LeMonHash construction can be parallelized transparently to the queries. We can divide the input data into ranges and construct independent PGM-indexes on each range. When concatenating the linear models of all ranges, we get a PGM-index for the whole input set. An advantage of this approach is that it is transparent to the queries. With the naive division, this index stores a negligible number of additional segments linear in the number of processors, but these cut-points can likely be “repaired” locally, so that we do not get a space overhead for most inputs. Mapping all keys to buckets by evaluating the PGM and therefore determining the input for the retrieval data structures is possible in parallel as well. Finally, the retrieval data structures can be constructed in parallel. This is again transparent to the queries and introduces only a negligible space overhead linear in the number of processors [Dil+22]. For variable-length strings, each node of the LeMonHash-VL construction can be parallelized just like described above. On top of that, different child nodes can be constructed independently in parallel.

Recursive Bucket Splitting. Inside a bucket, our implementation explicitly stores the ranks of all keys. Let us call this strategy *Direct Rank Storing* (DRS). An alternative method to determine the ranks within a bucket is *Recursive Bucket Splitting* (RBS). Take a bucket of size b that can contain keys from the range (L, R) . We can now split this bucket in half by storing how many of the keys are smaller than $M = (L + R)/2$. This takes $\lceil \log(b + 1) \rceil$ bits and splits the bucket into two sub-buckets of average size $b/2$. The two sub-buckets can be

handled recursively. For uniform random inputs with an average bucket size of $b \geq 3$, RBS needs less space than DRS. This reduction in space consumption comes at the cost of more expensive query operations. In particular, we need to query the retrieval data structures for every level in that bucket-internal tree. An additional problem with this variant is that it depends on the distribution of keys. In the worst case, when all key values are very close to L , the approach repeatedly needs to store the fact that b keys are smaller than the midpoint. This can lead to a space consumption close to $\log(b) \log(R - L)$, which can be arbitrarily large depending on the universe size. We therefore did not implement this construction for LeMonHash. Whether the RBS technique still works well with real-world data sets remains an open question. Given that many MMPHF construction algorithms use the bucketing technique (see Section 9.1), the RBS technique might still be of general interest for MMPHFs.

Indexed Chunk Extraction. As described in Section 9.3, the chunks in LeMonHash-VL are generated from consecutive characters. Now consider an input where the positions of branching characters of the keys are very far. Then the chunks encode a lot of data that is not necessary to differentiate the keys. Instead, it is possible to determine the distinct minima of the LCP values of strings in the corresponding node. Then chunks can be generated from the positions at these minima, which reduces the height of the tree. In practice, however, we find that the plain version is faster and more space-efficient (see Section 9.6.1).

Low-Level Optimizations In addition to the main algorithmic optimizations described in the main part, we here detail some more low-level optimizations of our implementation.

We encode the alphabet reduction as a bitmap and use the `popcount` instruction to determine a character’s index. For determining how many characters fit into a chunk with a given alphabet, we use a lookup table of size 256 because that is more efficient than a (floating point) logarithm and division. Depending on the dataset, multiple nodes of the tree might use alphabet reduction with a similar alphabet. When constructing a node, we therefore look if another node stores a superset of the alphabet that still leads to the same number of characters fitting into a chunk, and possibly re-use the alphabet. If no alphabet reduction is used, we use the `bswap` instruction to immediately convert the next 8 characters to a chunk.

To speed up access in Elias-Fano coded sequences, we use the `clz` instruction, which counts the number of leading zeroes in a word. When calculating the LCP of strings, we do so for multiple bytes at once using 64-bit comparisons. This general idea was already evaluated in Ref. [Din+20]. To avoid accessing the strings during alphabet map creation (which would lead to cache faults), we annotate the LCP array with the branching characters.

To decode the PGM metadata, which is stored as integers of small width, we use the `bextr` instruction to extract specific bits from a word. To evaluate the PGM, we use a 64-bit division with overflow detection instead of a 128-bit division because in practice, 64 bits are often enough to store the operands. For the PGM that auto-tunes its ε value, we abort early when we detect that the PGM itself is already larger than the optimal cost. This way, very small ε values can often be ruled out earlier.

9.5 Analysis

We now prove some properties of our LeMonHash data structure for integers. In our analysis, we use succinct retrieval data structures taking $rn + o(n)$ bits per stored value and answering queries in constant time (see Section 2.4 and [Dil+22]). Furthermore, since our bucket

mappers need multiplications and divisions, we make the simplifying assumption $U = 2^w$ to avoid dealing with the increased complexity of these arithmetic operations over large integers.

► **Theorem 9.1.** *A LeMonHash data structure with a bucket mapper that simply performs a linear interpolation of the universe on a list of n uniform random keys needs $\approx n(2.91536 + o(1))$ bits on average¹ and answers queries in constant time.*

Proof. For n uniform random integers mapped to n buckets, the number of keys per bucket follows a binomial distribution with $p = 1/n$. For large n , we can approximate this by the Poisson distribution with $\lambda = n \cdot 1/n = 1$. Therefore, the probability that a bucket has size k is $\frac{\lambda^k e^{-\lambda}}{k!} = \frac{1}{k!} e^{-1}$. Storing a bucket of size k requires k entries in the corresponding retrieval data structure, and each needs $\lceil \log k \rceil$ bits. Note that buckets of size 0 and 1 do not need to store ranks. Using the linearity of expectation, the average total number of bits to store in retrieval data structures is:

$$\mathbb{E}(\text{space}) = n \cdot \mathbb{E}(\text{space per bucket}) = n \cdot \sum_{k=2}^{\infty} k \lceil \log k \rceil \cdot \frac{1}{k!} e^{-1} \approx 0.91536n.$$

A succinct retrieval data structure can then store this using $\approx 0.91536n + o(n)$ bits of space. The Elias-Fano coded sequence of global ranks takes $2n + o(n)$ bits. Overall, we get a space consumption of $\approx n(2.91536 + o(1))$ bits.

For queries, the evaluation of the linear function and rounding can be executed in constant time. Now that we have the bucket index, we retrieve its offset and size from that binary sequence using two constant time select_1 queries. From that, we know which retrieval data structure to query, and the actual query works in constant time [Dil+22]. ◀

While this result is formally only valid for a global uniform distribution, for use in LeMonHash it suffices if each segment computed by the PGM-index is sufficiently smooth. It need not even be uniformly random as long as each local bucket has a constant average size. As long as the space for encoding the segments is in $\mathcal{O}(n)$ bits, we retain the linear space bound of Theorem 9.1. Moreover, the following worst-case analysis gives us a fallback position that holds regardless of any assumptions.

► **Theorem 9.2.** *A LeMonHash data structure with the PGM mapper takes $n(\lceil \log(2\varepsilon+1) \rceil + 2 + o(1)) + \mathcal{O}(m \log \frac{U}{m})$ bits of space in the worst case and answers queries in $\mathcal{O}(\log \log_w \frac{U}{m})$ time, where m is the number of linear models in a PGM with an integer parameter $\varepsilon \geq 0$ constructed on the n input keys.*

Proof. The rank estimate returned by the PGM is guaranteed to be far from the correct rank by ε . In other words, given a bucket number $i \in [n]$, any of the input keys with rank between $\max\{1, i - \varepsilon\}$ and $\min\{i + \varepsilon, n\}$ can be mapped to it, thus yielding a bucket of size at most $b = 2\varepsilon + 1$. In the worst case, there are $n/(2\varepsilon + 1)$ of such size- b buckets, which overall require storing n local ranks in a $\lceil \log b \rceil$ -bit retrieval data structure. Additional $2n + o(n)$ bits are needed for the Elias-Fano coded sequence of global ranks.

The remaining term of the space bound is given by the PGM, that we encode with an Elias-Fano representation of linear models' (x, y) -endpoints in $m(\log \frac{U}{m} + \log \frac{n}{m} + 2 \log(2\varepsilon + 1)) + \mathcal{O}(m)$ bits [FMV22]. This can be bounded by $\mathcal{O}(m \log \frac{U}{m})$ bits, since from [FV20b,

¹ Numerically, we find that a better space consumption of $\approx 2.902n$ bits can be achieved by mapping the n keys to only $\approx 0.909n$ buckets, but this difference is irrelevant in practice. It is also interesting to note that this is close to the space requirements of many practical non-monotone MPHFs (see Chapter 8).

Lemma 2] it holds $2\varepsilon \leq n/m \leq u/m$. Finally, we build the predecessor structure of [BN15, Theorem A.1] on the linear models' keys, which takes $\mathcal{O}(m \log \frac{u}{m})$ bits and yields a query time of $\mathcal{O}(\log \log_w \frac{u}{m})$. ◀

The worst-case bounds obtained in Theorem 9.2 are hard to compare with the ones of classic MMPHF (see Section 9.1) due to the presence of m (and ε), which depends on (and must be tuned according to) the approximate linearity of the input data, which classic MMPHFs are oblivious to.² We refer to Chapter 9 for bounds on m . Our experiments show that we obtain better space or space close to the best classic MMPHFs, while being much faster (we use a weaker but practical predecessor search structure than the one in Theorem 9.2). We refer to Section 9.6 for details.

9.6 Experiments

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

Experimental Setup. We perform our experiments on an Intel Xeon E5-2670 v3 with a base clock speed of 2.3 GHz running Ubuntu 20.04 with Linux 5.10.0. We use the GNU C++ compiler version 11.1.0 with optimization flags `-O3 -march=native`. As a retrieval data structure, we use BuRR [Dil+22] (see Section 2.4) with 64-bit ribbon width and 2-bit bumping info. To store the bucket sizes, we use the select data structure by Kurpicz [Kur22] in LeMonHash and Partitioned Elias-Fano [OV14] in LeMonHash-VL. To map tree paths to the node metadata, we use the MPHF PTHash [PT21] because it has very fast queries (see Chapter 8). For the PGM implementation in LeMonHash, we use the encoding from Theorem 9.2 and use a predecessor search on the Elias-Fano sequence (see Section 2.3). In LeMonHash-VL, since the number of linear models in a node is typically small, we encode them explicitly as fixed-width triples (*key*, *slope*, *intercept*) and find the predecessor via a binary search on the keys. All our experiments are executed on a single thread. Because the variation is very small, we run each experiment only twice and report the average. We run the Java competitors on OpenJDK 17.0.4 and perform one warm-up run for the just-in-time compiler that is not measured. With this, the Java performance is expected to be close to C++ [Bel+11]. Because Java does not have an unsigned 64-bit integer type, we subtract 2^{63} from each input key to keep their relative order.

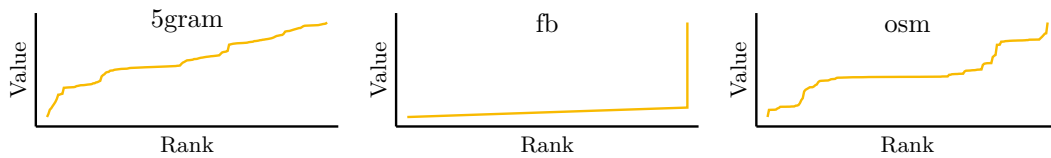
The code and scripts needed to reproduce our experiments are available on GitHub under the General Public License [Leh23b; LV23].

Datasets. In ordinary minimal perfect hashing, construction is mostly independent of the input data set (see Section 1.1). Because it is, in contrast, relevant for monotone minimal perfect hashing, we evaluate LeMonHash with several input sets. Our datasets, as in previous evaluations [Bel+11; GO14], are a *text* dataset that contains terms appearing in the text of web pages [Bel+11] and *urls* crawled from .uk domains in 2007 [BSV08]. Additionally, we also test with *dna* sequences consisting of 32-mers [FN]. Regarding real-world integer datasets, *5gram* contains positions of the most frequent letter in the BWT of a text file containing 5-grams found in books indexed by Google [BFV22; Goo]. The *fb* dataset contains Facebook user IDs [Kip+19] and *osm* contains OpenStreetMap locations [Kip+19]. We plot

² This happens also in other problems in which data is encoded with linear models [BFV22; FMV22].

■ **Table 9.1** Datasets we use, together with their length or average (ϕ) length. Top: real-world string datasets. Middle: real-world integer datasets. Bottom: synthetic integer datasets.

Dataset	n	Length	Description
text	35M	ϕ 11 bytes	Terms appearing on web pages, GOV2 corpus [Bel+11]
dna	367M	32 bytes	32-mer from a DNA sequence, Pizza&Chili corpus [FN]
urls	106M	ϕ 105 bytes	Web URLs crawled from .uk domains in 2007 [BSV08]
5gram	145M	32 bits	Positions of the most frequent letter in the BWT of a text file containing 5-grams found in Google Books [BFV22; Goo]
fb	200M	64 bits	Facebook user IDs [Kip+19]
osm	800M	64 bits	OpenStreetMap locations [Kip+19]
uniform	100M	64 bits	Uniform random
normal	100M	64 bits	Normal distribution ($\mu = 10^{15}$, $\sigma^2 = 10^{10}$)
exponential	100M	64 bits	Exponential distribution ($\lambda = 1$, scaled with 10^{15})



■ **Figure 9.4** Distribution of keys in integer datasets.

the distribution of these integer datasets in Figure 9.4. Note that the last 20 keys of the *fb* dataset contain very large special values. We deliberately keep those to evaluate the resilience to outliers. As synthetic integer datasets, we use 64-bit *uniform*, *normal*, and *exponential* distributions. We refer to Table 9.1 for details.

9.6.1 Tuning Parameters

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

LeMonHash. Different ways of mapping the keys to buckets have their own advantages and disadvantages. Table 9.2 gives measurements of the construction and query throughput, as well as the space consumption of different bucket mappers. Our implementation of LeMonHash with a linear bucket mapper achieves a space consumption of $2.94n$ bits, which is remarkably close to the theoretical space consumption of $2.91n$ bits (see Theorem 9.1). Of course, a global, linear mapping does not work for all datasets. A bucket mapper that creates equal-width segments by interpolating between sampled keys (denoted as “Segmented” in the table) is fast to construct and query, and it achieves good space consumption. But, as for the global linear mapping, this approach is not robust enough to manage arbitrary input distributions. In particular, for this heuristic mapper, it is easy to come up with a worst-case input that degenerates the space consumption. Conversely, with the PGM mapper, LeMonHash still achieves $2.96n$ and $2.98n$ bits on uniform random integers, but it is more performant and robust on other datasets (except on *osm*, where the heuristic mapper obtains a good enough mapping with only its equal-width segments, which are inexpensive to store). In fact, we explicitly avoided heuristic design choices in our PGM mapper (such as sampling

■ **Table 9.2** Comparison of different bucket mappers. The space consumption is given in bits per key, the query throughput in kQueries/second, and the construction throughput (c.t.) in MKeys/second.

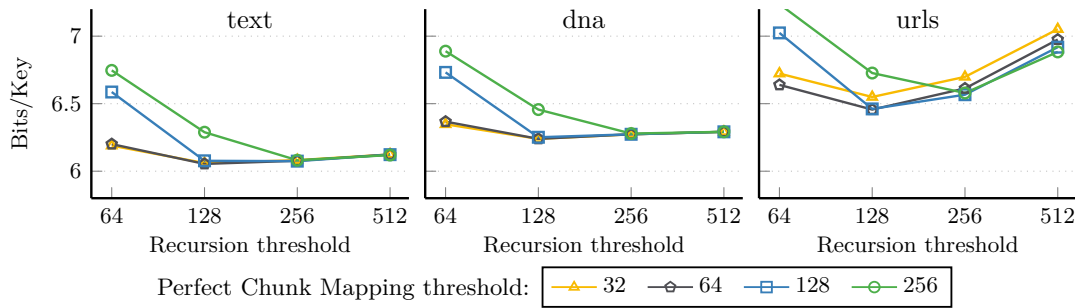
Dataset	Linear mapper			PGM $\varepsilon = \text{auto}$			PGM $\varepsilon = 31$			Segmented		
	bpk	kq/s	c.t.	bpk	kq/s	c.t.	bpk	kq/s	c.t.	bpk	kq/s	c.t.
5gram	5.60	1833.5	6.2	2.62	1747.0	3.8	2.63	1779.4	8.5	2.64	2145.9	14.5
fb	34.35	0.8	5.1	4.91	1156.1	2.8	4.91	1150.7	5.1	4.93	1441.3	7.2
osm	12.92	1525.3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

■ **Table 9.3** Comparison of different variants of LeMonHash-VL. The space consumption is given in bits per key, the query throughput in kQueries/second, and the construction throughput (c.t.) in MKeys/second. Variants with and without alphabet reduction (AR), a special indexed variant (Idx, see Section 9.4), and a variant with fixed instead of auto-tuned parameter ε for the bucket mapper.

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

input keys, removing outliers, or using linear regression) to not inflate our performance on the tested datasets at the expense of robustness on unknown ones (see Ref. [KRT22]). Finally, on most input distributions, auto-tuning the value of $\varepsilon \in \{15, 31, 63\}$ does not have a large effect on the space consumption.

LeMonHash-VL. Table 9.3 lists the effect of alphabet reduction on the query and construction performance. In general, alphabet reduction enables noticeable space improvements with only a small impact on the construction time. For the dna dataset, which uses only 15 different characters, the alphabet reduction has the largest effect, saving 1.3 bits per key and simultaneously making the queries 40% faster. The faster queries can be explained by the reduced tree height. Note that alphabet reduction makes the queries slightly slower for the other datasets. The reason is that instead of one single `bswap` instruction for chunk extraction, it needs multiple arithmetic operations (including `popcount`) for each input character. The indexed variant that builds chunks from the distinguishing bytes instead of a contiguous byte range (see Section 9.4) is slower to construct but does not show clear space savings, which can be explained by larger per-node metadata. We also experimented with different thresholds for when to stop recursion, as well as the perfect chunk mapping (see Section 9.3). Given that the space overhead from each bucket mapper is the same for all data sets, it is not surprising that the same threshold (128 keys) works well for all datasets (see Figure 9.5). While we have not plotted the query performance here, note that queries get slightly faster when increasing the recursion threshold because that reduces the height of



■ **Figure 9.5** Different thresholds for when to store ranks (of keys and chunks) explicitly.

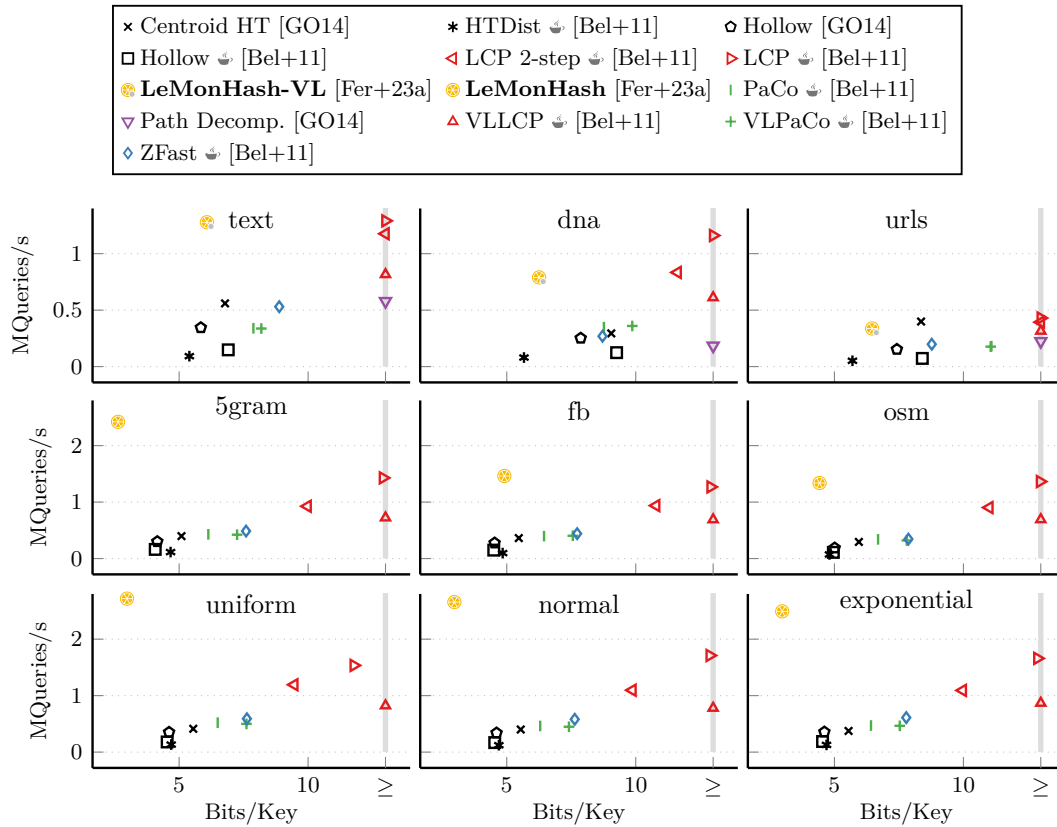
the tree. Finally, making the ε value of the PGM mapper constant instead of auto-tuned, we naturally get faster construction. As in the integer case, one would expect a fixed ε value to always produce results that are the same or worse than the auto-tuned version. This is not the case because, in the recursive setting, it is hard to estimate the effect of a mapper on the overall space consumption. Therefore, an ε value that needs more space locally can lead to a mapping that proves useful on a later level of the tree. This is why $\varepsilon = 63$ can achieve better space consumption than the auto-tuned version on the dna dataset.

9.6.2 Comparison with Competitors

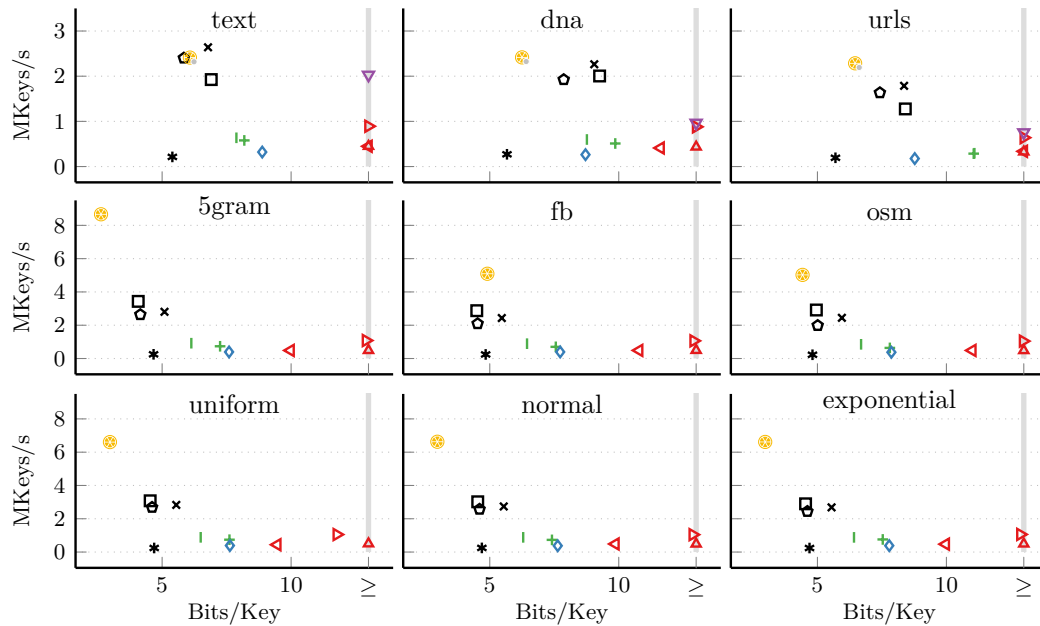
We now compare the performance of LeMonHash and LeMonHash-VL with competitors from the literature. Competitors include the C++ implementation by Grossi and Ottaviano [GO14] of the Centroid Hollow Trie, Hollow Trie, and Path Decomposed Trie. Because that implementation only supports string inputs, we convert the integers to a list of fixed-length strings. We point out that the Path Decomposed Trie crashes at an internal assertion when being run on integer datasets. For the Hollow Trie, we encode the skips with either Gamma or Elias-Fano coding, whatever is better on the dataset. We also include the Java implementations by Belazzougui et al. [Bel+11] of a range of techniques (see Section 9.1). We use either the FixedLong or PrefixFreeUtf16 transformation, depending on the data type of the input. For LeMonHash, we use the PGM mapper with $\varepsilon = 31$. For LeMonHash-VL, we use the PGM mapper with $\varepsilon = 63$, alphabet reduction and a recursion threshold $t = 128$.

Queries. Figure 9.6 plots the query throughput against the achieved storage space. In Table 9.4, we additionally detail the numbers in tabular format. The LCP-based methods (see Section 9.1) have very fast queries but also need the most space (in fact, they appear to the top-right of the plots). At the same time, LeMonHash matches or even outperforms the query throughput of LCP-based methods, while being significantly more space-efficient (in fact, it appears towards the top-left of the plots). Compared to competitors with similar space consumption, LeMonHash offers significantly higher query throughput.

Construction. Figure 9.7 plots the construction throughput against the space needed. On most synthetic integer datasets, LeMonHash shows significant improvements to the state of the art, whereas it matches or outperforms the competitors on real-world datasets. LeMonHash improves the construction throughput by up to a factor of 2, compared to the competitor with the next best space consumption (typically, variants of the Hollow Trie). While LeMonHash-VL only has the second best space consumption after the Hollow Trie Distributor, its construction is significantly faster.



■ **Figure 9.6** Query throughput for string, integer, and synthetic integer datasets vs space consumption. Competitors with the ☞ symbol in the legend are implemented in Java.



■ **Figure 9.7** Construction throughput for string, integer, and synthetic integer datasets.

9.7 Summary

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

Table 9.4 Comparison of string and integer datasets. Measurements from Figures 9.6 and 9.7 in tabular form. Query throughput is given in kQueries/s and space consumption is given in bits/key (bpk).

Method	text		dna		urls		5gram		fb		osm		uniform		normal		exponential	
	kq/s	bpk	kq/s	bpk	kq/s	bpk	kq/s	bpk	kq/s	bpk	kq/s	bpk	kq/s	bpk	kq/s	bpk	kq/s	bpk
Centroid HT [GO14]	560	6.78	294	9.05	399	8.36	398	5.09	363	5.47	295	5.95	413	5.55	400	5.54	375	5.55
Hollow [GO14]	345	5.84	252	7.87	153	7.42	300	4.15	276	4.53	187	5.01	351	4.61	339	4.60	356	4.61
Path Decomp. [GO14]	579	54.44	185	148.27	224	228.88												
HTDist ↵ [Bel+11]	92	5.40	80	5.67	52	5.70	115	4.67	97	4.84	73	4.81	133	4.69	122	4.69	127	4.69
Hollow ↵ [Bel+11]	148	6.90	124	9.26	73	8.41	162	4.07	150	4.50	110	4.96	179	4.54	169	4.53	188	4.54
LCP 2-step ↵ [Bel+11]	1176	13.12	834	11.62	394	17.81	926	9.98	938	10.79	903	11.00	1193	9.46	1096	9.87	1093	9.97
LCP ↵ [Bel+11]	1291	21.61	1161	16.23	430	22.74	1429	12.90	1269	12.90	1364	12.97	1535	11.77	1711	12.87	1660	12.87
PaCo ↵ [Bel+11]	339	7.88	350	8.77	181	11.09	429	6.13	397	6.44	340	6.69	522	6.50	463	6.30	471	6.42
VLLCP ↵ [Bel+11]	816	18.43	611	20.13	315	22.59	723	16.30	690	17.56	692	16.86	823	16.26	780	16.27	868	16.27
VLPaCo ↵ [Bel+11]	337	8.19	360	9.86	177	11.06	423	7.25	404	7.56	320	7.81	500	7.61	449	7.41	465	7.53
ZFast ↵ [Bel+11]	530	8.88	269	8.71	198	8.77	487	7.59	441	7.73	345	7.87	591	7.63	581	7.64	611	7.78
LeMonHash-VL [Fer+23a]	1278	6.08	790	6.25	338	6.46	1458	2.98	1111	4.91	857	4.39	1572	3.33	1647	3.32	1635	3.33
LeMonHash [Fer+23a]				only supports integers			2421	2.63	1463	4.91	1338	4.42	2718	2.98	2657	2.97	2493	2.98

10 Perfect Hashing for Variable Size Objects

Summary: *Perfect hashing can be used to build hash tables that can retrieve objects with a single probe. However, this only works for objects of fixed size or with large space overhead. PaCHash stores its objects contiguously in an array without intervening space, even if the objects have variable size. In particular, each object can be compressed using standard compression techniques. A small search data structure allows locating the objects in constant expected time.*

PaCHash is most naturally described as a static external hash table where it needs a constant number of bits of internal memory per block of external memory. In some sense, it beats a lower bound on the space consumption of k -perfect hashing. An implementation for fast SSDs needs about 5 bits of internal memory per block of external memory, requires only one disk access (of variable length) per search operation, and has small internal search overhead compared to the disk access cost. Our experiments show that it has lower space consumption than all previous approaches even when considering objects of identical size.

Attribution: This chapter is based on “PaCHash: Packed and Compressed Hash Tables” [KLS23]. Large parts of this chapter are copied verbatim or with minor changes from that publication. In the evaluation, we rework the main figures. The author of this dissertation is the main author of the paper. He implemented the algorithm, performed the experiments and wrote most of the manuscript. The analysis of the space consumption when using Succincter is due to Florian Kurpicz. All authors made significant contributions, to algorithm design, analysis, design & interpretation of the experiments, and the write-up. The authors would like to thank Peter Dillinger and Stefan Walzer for early discussions leading to the paper.

Hash tables support constant time key-based retrieval of objects and are one of the most widely used data structures. There is a lot of work on hash tables which need little more space than just the stored objects themselves [ANS10; Ben+23; Fot+05; Knu98; KPR22; PR04]. Using perfect hashing, it is possible to build a static hash table that can retrieve objects with a single probe. However, all these approaches are only space-efficient for objects of identical size which makes it impossible to compress the objects with variable bit-length codes. Compressed data structures store data in a space-efficient way, preferably approaching the information theoretical limit, and support various kinds of operations without the need to decompress the entire data structure first [AKS15; FM05; Gog+14; Zha+18]. There has been intensive previous work on hash tables and compressed data structures but, surprisingly, the intersection leaves big gaps. Currently, most hash tables for objects of variable size store references from table entries to the data which entails a space overhead of at least $\log N$ bits per object, where N is the total size of all objects in the table.

In this chapter, we introduce PaCHash, an external memory hash table that supports objects of variable size. PaCHash eliminates fragmentation by *packing* the objects contiguously

■ **Table 10.1** Symbols used in this chapter.

S	Set of objects
n	Number of objects
N	Total size of objects (bits)
p	Internal index data structure
a	Tuning parameter: Bins per block
$m = N/\bar{B}$	Number of blocks
B	Block size (bits)
$\bar{B} = B - d$	Payload data per block
$d \in 0.. \log B$	Encoding-dependent number of bits to store position of first bin of block

in memory without leaving free space. Like a perfect hash function, PaCHash then uses a highly space-efficient search data structure to translate input objects to memory locations. We deliberately use the word *object* for the stored data because that highlights the flexibility of PaCHash. Naturally, an object stores a key-value-pair, but we will look at other uses as well. During construction, objects are first hashed to *bins*. The bins are stored contiguously in m blocks of size B . PaCHash essentially stores one bin index per block using a searchable compressed representation which enables finding the block(s) where a bin is stored.

Even though hash tables like PaCHash have applications in object stores, there is little previous work on space-efficient hash tables for objects of variable size (see Section 10.1). For objects of identical size s , minimal perfect hashing requires a constant number of bits per object. PaCHash approximates this when choosing $B = s$, also needing a (slightly larger) constant number of bits per object. The picture changes when we look at larger block sizes $B = ks$ and the corresponding approach of *minimal k -perfect hashing (MkPH)* [BBD09]. Now, PaCHash still needs only a constant number of bits per block, while there is a *lower bound* of $\Omega(\log k)$ bits per block using MkPH (see Section 2.7).

In Section 10.2 we describe PaCHash in detail. Section 10.4 describes different implementation variants including fully internal and fully external versions as well as a variant that is usable as variable-bit-length array (VLA) [Nav16]. We analyze PaCHash in Section 10.3. Finally, in Section 10.5, we describe experiments for an external memory implementation. As close contenders, we also implement *Separator Hashing* [GL88; LK84] and *Cuckoo Hashing* [Aza+94; PR04] with adaptations that partially allow variable size objects. We summarize the results in Section 10.6.

Model of Computation. We describe our results in a variant of the external memory model [VS94] adapted to a situation where objects are compressed to variable length sequences of bits. We have a *fast memory* of size M bits. Accesses to a large *external memory* are *I/Os* to blocks of B consecutive bits. In contrast to the original model, we analyze both I/Os and internal work. $scan(N)$ denotes the cost (I/Os *and* internal work) of scanning N bits of data.¹ $sort(N)$ denotes the cost of sorting N bits.² In particular, we are interested in a high load factor, which is N divided by the total external space consumption.

¹ The internal work may depend on the encoding of the data. For example, we may need $\Theta(N)$ machine instructions, or, a faster encoding may enable bit-parallel processing in $\mathcal{O}(N/\log n)$.

² This entails $(N/B)(1 + \lceil \log_{M/B}(N/M) \rceil)$ I/Os. In this chapter, algorithms with linear internal work are possible exploiting random integer keys. The cost also includes (de)coding overhead as in *scan* operations.

10.1 Related Work

The following section introduces related data structures from the literature. Table 10.2 provides an overview over the most important parameters. There are close contenders in the form of *object stores* from the database literature. BerkeleyDB [OBS99] uses a B⁺-Tree [Com79] of order d , where each node branches between d and $2d$ times. LevelDB [Goo21] and RocksDB [Fac21] use a Log-Structured Merge tree [O’N+96], which stores multiple levels of a static data structure with increasing size. Insertions go into the first level and when a level gets too full, it is merged into the next level. SILT’s *LogStore* [Lim+11], Facebook *Haystack* [Bea+10] and *FAWN* [And+09] simply store a pointer of size $\Omega(\log N)$ to each object. Real world instances often store very small objects [Nis+13], so the pointers add a considerable amount of overhead.

Sorted Objects. *LevelDB*’s static part [Goo21] stores objects in key order, enabling range searches and common-prefix-compression. *SortedStore* in SILT [Lim+11] sorts the objects by their hashed key and uses entropy coded tries as an index. Pagh [Pag03] proposes to sort the n objects by a hash function with range $\geq n^3$. The internal memory stores the first hash function value mapped to each block. This data structure can be queried using a predecessor data structure in time $\mathcal{O}(\log \log n)$. A novel idea in PaCHash is that it uses a hash function range based on the total space N instead of the number of objects n , which enables efficient queries and compact representation.

External Hash Tables. In external hash tables, each table cell corresponds to a fixed size block. A common technique to support variable size objects is using indirection by internally storing a pointer to the object contents, possibly inlining parts of the objects [Lim+11, Section 4]. NVMKV [Már+15] and KallaxDB [Che+21] use an SSD as one large hash table and rely on SSD internals to handle empty blocks in a space-efficient way. Overflowing blocks due to hash collisions can be handled with perfect hashing [LR85; RT89] or using one of the following techniques.

With *Hashing with Chaining*, objects of overflowing blocks are stored in linked lists. *SkimpyStash* [DSL11] chains objects using an external successor pointer for each object. This trades internal memory space for latency because of multiple dependent I/Os. Jensen and Pagh’s [JP08] data structure reserves parts of the external memory as a buffer to reduce the need for chaining. *Extendible Hashing* [Fag+79] keeps a balanced tree of blocks. Overflowing blocks are split into two children indexing one more bit of the hashed key.

Another method for resolving collisions is *open addressing*, where each object could be located in multiple blocks. *Cuckoo Hashing* [DW07; PR04] (see Section 2.5) locates each object in one of two (or more [Fot+05]) independently hashed blocks. Queries can load both blocks in parallel to reduce latency. With *Separator Hashing* [GL88; LK84], each object has a sequence of blocks it could be stored in and a corresponding sequence of signatures. When a block overflows, the objects with the highest signature values are pushed out to the next block in their respective sequences. The internal memory stores the highest signature value of the objects placed in each block. A query follows the object’s sequence of blocks and stops when it finds a separator that is larger than the corresponding signature. *Linear hashing with separators* [Lar88] is a dynamic variant with a linear probe sequence. *External Robin Hood Hashing* [Cel88] is similar to linear separator hashing, but it instead pushes out objects that are closest to their respective home address. For each block, the internal memory stores the smallest distance of its objects to their respective home address.

■ **Table 10.2** Space-efficient object stores from the literature. To unify the notation, we convert all values so that they refer to objects of size $s = 256$ bytes stored in blocks of $B = 4096$ bytes. Each block contains $B/s = 16$ objects. Top: Stores for objects of identical size. Can be used for objects of variable size by using indirection or for some methods by accepting significantly lower load factors. Bottom: Dedicated variable size object stores. This table also contains VLAs, even though those are a slightly different field.

	Method	Internal memory	Load Factor	I/Os
Fixed size	Extendible Hashing [Fag+79]	$\log m$ bits/block	90%	1
	Larson et al. [LR85]	96 bits/block	<96%	1
	SILT SortedStore [Lim+11]	51 bits/block	100%	1
	Linear Separator [Lar88]	8 bits/block	85%	1
	Separator [GL88; LK84]	6 bits/block	98%	1
	Robin Hood [Cel88]	3 bits/block	99%	1.3
	Ramakrishna et al. [RT89]	4 bits/block	80%	1
	Jensen, Pagh [JP08]	0 bits/block	80%	1.25
	Cuckoo [Aza+94; PR04]	0 bits/block	<100%	2
	PaCHash , $a = 1$	2 bits/block	100%	2^3
	PaCHash , $a = 8$	5 bits/block	100%	1.13^3
Variable size	SILT LogStore [Lim+11]	832 bits/block	100%	1
	Küleki [Kü14] (VLA)	176 bits/block	<100%	$0-11^4$
	SkimpyStash [DSL11]	32 bits/block	$\leq 98\%$	8
	Blandford, Blelloch [BB08] (VLA)	16 bits/block	$\leq 50\%$	1
	PaCHash , $a = 1$	2 bits/block	99.95%	2.06^3
	PaCHash , $a = 8$	5 bits/block	99.95%	1.19^3

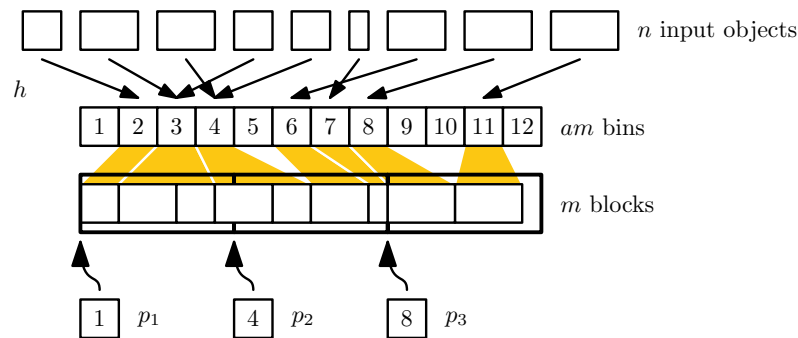
Variable-Bit-Length Arrays. Variable-bit-length arrays (VLAs) are arrays containing objects of variable size. Oftentimes, VLAs are used to efficiently access variable-length codes, e.g., Elias- γ and $-\delta$ codes [Eli74] or Golomb codes [Gol66]. VLAs are closely related to PaCHash, which can be used also as VLA by using the array index instead of the hash function, see Section 10.4. Conversely, PaCHash can be seen as a VLA where each entry stores a PaCHash bin. However, most VLAs have some limitations that rule out storing the PaCHash bins efficiently. A major difference to all VLAs described below is PaCHash allowing objects to span over multiple blocks of fixed size.

Navarro [Nav16, Section 3.2] describes several techniques for implementing VLAs. However, none of them achieves the same favorable space-time trade-off as the PaCHash VLA. The closest one – sampled pointers – needs $N + n \log(N)/k$ bits of space with access cost bounded by the time needed to skip k objects. Note that this time can be large when large objects need to be skipped.⁵ All the other described VLAs need several bits of space overhead per object (multiplied with a factor that depends on the maximum or average object size). The VLA introduced by Küleki [Kü14] uses wavelet trees [FM00] to partition the universe. This makes the query time depend double logarithmically on the largest object stored in

³ PaCHash performs one I/O of variable size which is faster than the competitors' multiple I/Os.

⁴ Using 256 byte objects, we have an alphabet size of $2^{8 \cdot 256}$, and $\log \log 2^{8 \cdot 256} = 11$.

⁵ Space could be reduced to $N + \frac{n}{k} (2 + \log \frac{kN}{n})$ bit using Elias-Fano coding of the pointers – resulting in similar space as the PaCHash VLA with $B = kN/n$ but with worse access costs.



■ **Figure 10.1** Example of PaCHash with $n = 9$ objects. The hash function h maps the objects to 12 bins, i.e., $a = 4$. The bin content is then contiguously written to the $m = 3$ external memory blocks. The internal memory index p stores the first bin intersecting with each block. Note that locating bin 8 will return the range 2..3, i.e., block 2 is loaded superfluously because there is no preceding empty bin that can encode whether it overlaps into the previous block. All other bins are located optimally.

the VLA, a limitation not existing in PaCHash. Blandford and Blelloch [BB08] describe dynamic VLAs and hash tables for variable sized objects. However, their technique incurs a constant factor of space overhead and is limited to objects of bounded size. They partition the objects into blocks, but the blocks are generally only partially filled and do not allow objects crossing block boundaries as in PaCHash.

10.2 The PaCHash Data Structure

We now present PaCHash in detail – a hash table which considerably improves on the data structures from the literature. It natively supports variable size objects without the need for indirection or empty cells. It needs only a few bits of internal memory per *block* and still needs only one single I/O operation (of variable length) per query. PaCHash consists of an *external part* subdivided into m blocks of exactly B bits each that store the actual objects and an *internal part* that allows finding the blocks storing an object. Figure 10.1 gives an example for the external and internal memory data structures. We deliberately use the word *object* for the stored data because that highlights the flexibility of PaCHash. Naturally, an object stores a key-value-pair, but it can also store only a value to obtain an external dictionary data structure. It is even possible to use quotienting by storing the bin index inside the first object of each bin.

10.2.1 External Object Representation

PaCHash stores the objects sorted by a hash function h with a rather small domain, namely $h : K \rightarrow 1..am$, where K is the set of possible keys, m is the number of blocks and a is a tuning parameter that we assume to be a power of two. The hashes can collide and therefore group the objects into am bins. The objects are now basically stored contiguously. “Basically” means that blocks may also contain information needed to find the first object or bin stored in them. Refer to Section 10.4 for a discussion of alternative encodings. Our default assumption is as follows: Each external block stores an offset of size $d = \log B$ bits indicating the bit where the first bin in the block starts. The remaining space stores the objects contiguously where an object may have an arbitrary size in bits. No space is left between subsequent objects. In particular, object representations may overlap block boundaries. We assume

- **Algorithm 10.1** A query for an object x calls $\text{locate}(x)$, loads the returned block range, and scans the blocks to find the object content. Determining the range boils down to predecessor queries on p .

```

Function locate( $x$ )
   $b := h(x)$ 
  find  $i$  such that  $p_{i-1} < b \leq p_i$  // predecessor query
  if  $p_i = b$  then  $i := i - 1$  //  $b$  may start in previous block
  find first  $j$  such that  $p_j > b$  // predecessor query or scan
  return  $i..(j - 1)$ 

```

that objects are encoded in a self-delimiting way, i.e., when we know where an object starts, we can also find its end. For example, we could have a prefix-free code for the objects. Construction first sorts the objects by their hash function value. Then it scans the sorted objects, constructing both the external and the internal data structure along the way. Refer to Section 10.3 for more details. If the internal data structure gets lost, for example due to a power outage, it can be re-generated using a single scan over the external memory data.

10.2.2 Internal Memory Data Structure

Given a bin b , the internal memory data structure p can be used to determine a (near-)minimal range $i..j$ of block indices such that b is stored in that range. When performing a query, that block range can then be loaded from external memory and scanned for the sought key. In practice, the resulting latency is often close to that of loading a single block. Conceptually, p stores a sequence $\langle p_1, \dots, p_m \rangle$ where p_i specifies the first bin whose data is at least partially contained in block i .⁶ We can use a predecessor query on p to determine i . When the predecessor is b itself, we also need to load the previous block. Another predecessor query or scanning then determines j , as illustrated by the pseudocode in Algorithm 10.1. To get the most out of this specification, we take empty bins into account: When a bin starts exactly at a block boundary and has an empty predecessor, we store that predecessor. This implies that if (and only if) a bin b starts at a block boundary and the previous bin $b - 1$ is nonempty, retrieving bin b will load one block too much. Note that p is a monotonically increasing sequence of integers which can be represented with different methods and trade-offs.

Elias-Fano Coding. A standard technique for storing monotonic sequences is Elias-Fano coding, where predecessor queries generally take logarithmic time (see Section 2.3). However, we will prove in Lemma 10.7 that they take constant expected time in the case of PaCHash. The internal memory usage is $m(2 + \log(a) + o(1))$ bits (see Lemma 10.2).

Bit Vector with Succincter. It is also possible to store p as a bit vector with rank and select support. An item p_i at position i is then represented as a 1-bit in position $i + p_i$. The position of the predecessor of a bin b can be found in constant time by calculating $\text{select}_0(b) - b$. The actual value can be calculated using a select_1 query. Because the bit vector is sparse, we can use Succincter [Pua08] to compress it and its rank and select structures down to about $m(1.44 + \log(a + 1) + o(1))$ bits (see Lemma 10.4).

⁶ An alternative would be to store the first bin that *starts* in each block. This introduces a special case when a block is fully overlapped by a bin and needs slightly more work when performing queries.

Entropy Coding. We observed that in practice, the bit vector is considerably more regular than a truly random one and thus allows additional compression. This can be made fast by splitting it into ranges that are compressed individually, e.g., using dictionary compression. In our experimental evaluation in Section 10.5.2, we see a space-time trade-off, where we can achieve internal memory space consumption less than the best generic results described above in Section 10.2.2.

10.3 Analysis

We now formalize the properties of PaCHash in Theorem 10.1 which basically says the following: External space is just the space needed to store the variable sized objects plus possibly a few bits per block to know where the first object in the block starts. Internal space is about $2 + \log a$ bits per block where a is a tuning parameter that also shows up in a term adding $1/a$ expected I/Os to the retrieval cost.

While proving the theorem, we discuss some variants and implications. Section 10.3.1 considers construction cost and final space consumption, while Section 10.3.2 looks at I/Os and internal work of queries.

► **Theorem 10.1.** *Consider n objects of total size N bits which are stored in m blocks of size B . Let $d \in [0, \log B]$ be an encoding-dependent number of bits needed to specify where the first bin or object of a block starts and $\bar{B} = B - d$ be the payload size per block, i.e., $m = N/\bar{B}$. For a parameter a , let a random uniform hash function map the objects to am bins.*

Then, PaCHash with Elias-Fano coding needs $m(2 + \log a + o(1))$ bits of internal memory and $N(1 + d/\bar{B})$ bits of external memory. The construction cost is the same as that of sorting the objects using am random integer keys. The expected time for retrieving an object of size $|x|$ bits is constant plus the time for scanning $1 + |x|/\bar{B} + 1/a$ blocks. The unsuccessful search time is the same except that $|x|$ is replaced by 0.

10.3.1 Construction

Assuming that the set of input objects is stored in compressed form on external memory, we mainly need to sort the objects by their hash function value. In our model, this has complexity $\text{sort}(N)$. In most practically relevant situations, this can even be done in $\mathcal{O}(\text{scan}(N))$ using integer sorting, see Section 10.3.3 for details.

The sorted representation is then scanned and basically copied to the output, only adding d bits of information within each block, which allow a query to initialize the scanning operation. What d is depends on the concrete encoding of the data, ranging from $d = 0$ for objects of identical size or for 0-terminated strings to $d = \log(B)$ bits when we explicitly encode the starting position of an object or bin. Refer to Section 10.4 for examples.

► **Lemma 10.2.** *When using Elias-Fano coding to store p , the index needs $2 + \log a + o(1)$ bits of internal memory per block and can be constructed in time $\mathcal{O}(m)$.*

Proof. p consists of $k = m$ integers $\leq am = U$. Inserting this into the space consumption of Elias-Fano coded sequences (see Section 2.3) gives us $\text{space}(p) = k(2 + \log(U/k)) + 1 = m(2 + \log(am/m)) + 1 = m(2 + \log a) + 1$. The select_0 data structure on the upper bits H can be stored in $o(m)$ bits (see Section 2.1). Each of the m insertions into the sequence can be done in constant time while generating the external object representation. The construction of the select_0 data structure takes time $\mathcal{O}(m)$. ◀

We can also use Succincter [Pua08] to store the compressed sequence p . In this case, we get an almost optimal space consumption of $1.4427 + \log(a + 1) + o(1)$. The following Lemmas prove this. First, we start with a general property of binomial coefficients.

► **Lemma 10.3.** *For any $c > 1, n > 0$, let $f(n, c) := \sqrt{\frac{c}{(c-1)2\pi n}} \left(\frac{c^c}{(c-1)^{c-1}}\right)^n$, then*

$$f(n, c) \left(1 - \frac{c^2 - c + 1}{12c(c-1)n}\right) < \binom{cn}{n} < f(n, c)e^{-\frac{1}{12n+1}} = f(n, c) \left(1 - \frac{1}{12n} + \mathcal{O}\left(\frac{1}{n^2}\right)\right).$$

Proof. We use the identity $\binom{cn}{n} = \frac{(cn)!}{n!(cn-n)!}$ as well as Stirling's approximation

$$\sqrt{2\pi m} \left(\frac{m}{e}\right)^m e^{\frac{1}{12m+1}} < m! < \sqrt{2\pi m} \left(\frac{m}{e}\right)^m e^{\frac{1}{12m}}.$$

For the upper bound we get

$$\begin{aligned} \binom{cn}{n} &= \frac{(cn)!}{(cn-n)!} \cdot \frac{1}{(cn-n)!} < \frac{\sqrt{2\pi cn} \left(\frac{cn}{e}\right)^{cn} e^{\frac{1}{12cn}}}{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\frac{1}{12n+1}}} \cdot \frac{1}{\sqrt{2\pi(c-1)n} \left(\frac{(c-1)n}{e}\right)^{(c-1)n} e^{\frac{1}{12(c-1)n+1}}} \\ &= \sqrt{\frac{c}{(c-1)2\pi n}} \cdot \left(\frac{c^c}{(c-1)^{c-1}}\right)^n \cdot e^{\frac{1}{12cn} - \frac{1}{12n+1} - \underbrace{\frac{1}{12(c-1)n+1}}_{\leq 12cn}}. \end{aligned}$$

The claim follows by observing that the leftmost and rightmost term in the exponent of e cancel out in the estimation. The asymptotic expansion of the upper bound can be obtained using Taylor series expansion.

Similarly, for the lower bound we get

$$\begin{aligned} \binom{cn}{n} &= \frac{(cn)!}{n!} \cdot \frac{1}{(cn-n)!} > \frac{\sqrt{2\pi cn} \left(\frac{cn}{e}\right)^{cn} e^{\frac{1}{12cn+1}}}{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}} \cdot \frac{1}{\sqrt{2\pi(c-1)n} \left(\frac{(c-1)n}{e}\right)^{(c-1)n} e^{\frac{1}{12(c-1)n}}} \\ &= \sqrt{\frac{c}{(c-1)2\pi n}} \cdot \left(\frac{c^c}{(c-1)^{c-1}}\right)^n \cdot e^{\frac{1}{12cn+1} - \frac{1}{12n} - \frac{1}{12(c-1)n}} \\ &> \sqrt{\frac{c}{(c-1)2\pi n}} \cdot \left(\frac{c^c}{(c-1)^{c-1}}\right)^n \cdot \left(1 - \frac{c^2 - c + 1}{12c(c-1)n}\right). \end{aligned}$$

◀

► **Lemma 10.4.** *When using Succincter [Pua08] to store p , the index needs $1.4427 + \log(a + 1) + o(1)$ bits of internal memory per block.*

Proof. Remember that the internal memory data structure p of PaCHash stores m integers in the range $1..am$ and must support predecessor queries. We represent all integers in a bit vector of length $(a + 1)m$, using the same idea used for the most significant bits in Elias-Fano coding. That is, each of the m integers p_i is represented as a 1-bit in position $i + p_i$. Answering predecessor queries (which we do not consider here) becomes harder to analyze, as we have no information about the distribution of 1-bits in the bit vector.

Using Succincter, we can store a size- U bit vector that contains n ones and supports rank and select queries using only $\log \binom{U}{n} + \frac{U}{\log U} + \tilde{O}U^{\frac{3}{4}}$ bits. Since we have a length- $(a + 1)m$ bit vector that contains m ones, we require $\log \binom{(a+1)m}{m} + \frac{(a+1)m}{\log((a+1)m)} + \tilde{O}((a + 1)m)^{\frac{3}{4}}$ bits of space. We now show the upper bound for required memory using Lemma 10.3 and $\tilde{O}((a + 1)m)^{\frac{3}{4}} = o(m)$.

$$\begin{aligned}
\log \binom{(a+1)m}{m} + o(m) &< \log \left(\sqrt{\frac{(a+1)}{2\pi am}} \left(\frac{(a+1)^{a+1}}{a^a} \right)^m e^{-\frac{1}{12m+1}} \right) + o(m) \\
&= \underbrace{\log \sqrt{\frac{(a+1)}{2\pi am}}}_{\leq 0} + \log \left(\left(\frac{(a+1)^{a+1}}{a^a} \right)^m \right) + \underbrace{\log e^{-\frac{1}{12m+1}}}_{\leq 0} + o(m) \\
&\leq \log \left(\left(\frac{(a+1)^{a+1}}{a^a} \right)^m \right) + o(m) = m((a+1)\log(a+1) - a\log a) + o(m) \\
&= m \left(a \log \left(\frac{a+1}{a} \right) + \log(a+1) \right) + o(m) \leq m(1.4427 + \log(a+1)) + o(m)
\end{aligned}$$

The last inequality is due to the fact that $a \log \left(\frac{a+1}{a} \right)$ converges to $\log e \approx 1.4427$ from below. Overall, we require less than $1.4427 + \log(a+1) + o(1)$ bits for each block. ◀

► **Lemma 10.5.** *Using Succincter for representing monotonic sequences is almost space optimal.*

Proof. In Lemma 10.4 we have already seen that Succincter needs close to $m(\log(e) + \log(a+1))$ bits of space. $\binom{am}{m}$ is the number of *strictly* monotonic sequences of m numbers in the range $1..am$ and thus a lower bound for the number of monotonic sequences. Using Lemma 10.3 once more, we get

$$\log \binom{am}{m} \approx m((a-1)\log \left(\frac{a}{a-1} + \log a \right))$$

bits as a lower bound. Looking at the difference divided by m (i.e. bits per block), we get

$$\begin{aligned}
&a \log \frac{a+1}{a} + \log(a+1) - (a-1)\log \frac{a}{a-1} - \log a \\
&= a \log \frac{a^2-1}{a^2} + \log \frac{a+1}{a-1} = \frac{\log e}{a} + \mathcal{O}\left(\frac{1}{a^3}\right).
\end{aligned}$$

This difference (obtained using Taylor series development) is much smaller than the $\log e + \log(a+1)$ bits per block needed by the Succincter data structure – at least for sufficiently large a . ◀

As we show in Section 2.7, k -perfect hash functions have a space lower bound of $\frac{n}{k} \cdot \frac{1}{2} \log(2\pi k)$ bits. The value n/k is the number of blocks, so Mk PHFs need $\Omega(\log k)$ bits of space per block, while we show above that PaCHash needs a constant number. In a way, PaCHash therefore breaks the space lower bounds of Mk PHFs while keeping the same $\mathcal{O}(1)$ query time. Choosing parameter a large can bring the number of I/O operations arbitrarily close to optimal, independently of k .

10.3.2 Query

We first show that a query loads a small expected number of blocks, depending only on the size of that specific object – not the other objects in the data structure. We then show that the exact blocks to be loaded can be determined upfront without any I/O operations, using constant time.

► **Lemma 10.6.** *Retrieving an object x of size $|x|$ from a PaCHash data structure loads $\leq 1 + |x|/\bar{B} + 1/a$ consecutive blocks from the external memory in expectation (setting $|x| = 0$ if x is not in the table).⁷*

Proof. We first derive the expected number of blocks overlapped by the bin $b_x = h(x)$ that x is stored in. We then analyze the edge case that PaCHash sometimes loads one additional block unnecessarily even though it is not overlapped.

The expected size $\mathbb{E}(|b_x|)$ of b_x is the sum of $|x|$ and all other objects from the input set S that are mapped to it:

$$\begin{aligned} \mathbb{E}(|b_x|) &= |x| + \sum_{y \in S, y \neq x} |y| \Pr(y \in b_x) \\ &\leq |x| + \sum_{y \in S} |y| \Pr(y \in b_x) = |x| + \sum_{y \in S} |y| \cdot \frac{1}{am} = |x| + \bar{B}m \cdot \frac{1}{am} = |x| + \frac{\bar{B}}{a} \end{aligned}$$

Let X denote the number of blocks overlapped by bin b_x . Assuming that the block boundaries and bin boundaries are statistically independent,⁸ and using the linearity of the expected value, we get $\mathbb{E}(X) = 1 + (\mathbb{E}(|b_x|) - 1)/\bar{B} = 1 + |x|/\bar{B} + 1/a - 1/\bar{B}$.

At a position i , the sequence p stores the first bin b_i that intersects with block i . Most of the time, this also means that b_i extends into block $i - 1$, which is why queries load that block as well. When a bin starts *exactly* at a block boundary, though, the previous block is not actually needed. Because bin boundaries are statistically independent of block boundaries, the probability of that happening is $1/\bar{B}$.⁹

We get the result by putting together the expected blocks overlapped by a bin and the probability for loading one single block too much. For negative queries, we are interested in the size of the bin that x would be hashed to, so we can simply set $|x| = 0$. ◀

► **Lemma 10.7.** *When using Elias-Fano coding for the index data structure of PaCHash, the range of blocks containing the bin of an object x can be found in expected constant time.*

Proof. A query for an object x consists of four steps. First, we hash x to get the corresponding bin $b_x = au + \ell$, where a is the tuning parameter of PaCHash. We then execute a constant time $select_0$ query on the upper bits H (see Section 2.1). That gives us the start of a cluster of entries in the sequence that all have the same $\log(m)$ most significant bits u . We need to iterate over the cluster entries which are $< b_x$ until we find the predecessor. Each cluster entry corresponds to a stored bin index. Let us bound the expected size $\mathbb{E}(Y_u)$ of all bins that have most significant bits u and are $< b_x$.

⁷ Using fewer estimates in the proof one can derive a bound of $1 + \frac{|x| - c + 1 - e^{-\beta}}{\bar{B}} + \frac{1}{a}$ where $\beta = \frac{n\bar{B}}{Na}$ is the average number of objects per bin and c is the greatest common divisor of \bar{B} and all object sizes. In particular, for objects of identical size dividing \bar{B} , the bound is close to $1 + 1/a$.

⁸ We can guarantee the independence by cyclically shifting the data structure, i.e., we set the offset of the first block to a random number in $0..(\bar{B} - 1)$ and let the last bins wrap around into the first block.

⁹ When the preceding bin b_{-1} is empty, PaCHash stores that empty bin in p , as described in Section 10.2. This means that the probability of unnecessary block loads actually is smaller, namely $\frac{1}{\bar{B}}(1 - \Pr(|b_{-1}| > 0))$, where $\Pr(|b_{-1}| > 0) = (1 - \frac{1}{am})^n \approx e^{-\frac{n}{am}}$ is the probability of b_{-1} being empty.

$$\begin{aligned} \mathbb{E}(Y_u) &= \sum_{y \in S} |y| \cdot \Pr(h(y) \text{ has MSB} = u; h(y) < h(x)) \\ &\leq \sum_{y \in S} |y| \cdot \Pr(h(y) \text{ has MSB} = u) = \frac{1}{m} \sum_{y \in S} |y| = \frac{m\bar{B}}{m} = \bar{B} \end{aligned}$$

The expected number of cluster entries we need to scan is therefore $\mathbb{E}(Y_u)/\bar{B} = 1$. The practical implementation then further scans the cluster to find the last block overlapping b_x . This takes non-constant time $\mathcal{O}(1 + |x|/\bar{B})$, which is not a problem since a proportional number of blocks are loaded anyway. However, we strengthen the lemma by observing that we can also use another *select*₀ query followed by a backward scan of the cluster. ◀

10.3.3 Details on External Sorting

We now show that the external sorting needed during construction of a PaCHash data structure can be done in scanning complexity using very modest additional assumptions. First note that the problem of sorting objects during construction is easy when the average object size exceeds the block size, i.e., $N/n > B$ and thus $n < N/B$. In that case, a variant of bucket sort that maps the keys to $\mathcal{O}(n)$ buckets runs with linear internal expected work and $\mathcal{O}(n + N/B) = \mathcal{O}(N/B)$ I/Os [San+19, Theorem 5.9].

Otherwise, the average object size N/n must be at least $\log n$ since we are looking at objects with unique keys. For the remaining case $\log n \leq N/n \leq B$, we additionally make a *tall cache assumption* quite usual for external memory [Fri+99] where $M > B^2$. Since the index data structure has at least N/B bits, we also know that $M \geq N/B$. A single scan of the input can partition it into pieces of size about $\frac{N}{M/B} \leq \frac{N}{(N/B)/B} = B^2 \leq M$ which fit into internal memory. Moreover, since the average object size is $\geq \log n$, we can afford to replace the objects in an internally sorted fragment of the input by key-pointer pairs which once more allows us to use bucket sort – this time running in internal memory.

10.4 Variants and Refinements

Up until now, PaCHash was described as a static, external hash table for objects of variable size. The following section describes variants of the scheme.

Object Encoding. Instead of storing objects contiguously with a self-delimiting encoding, PaCHash allows for a wide range of other options, as shown in Table 10.3. In general, we have a trade-off between the space needed to decode the objects in a block and the strength of assumptions made on object representation. For example, explicitly storing the offsets of objects in blocks removes the restriction to a self-delimiting encoding, without increasing the size of the internal data structure. Another important case are objects of identical size where we can calculate the block offset at query time and therefore need no external space overhead. When the object size divides the block size, it can be shown that the expected number of I/O operations is close to $1 + 1/a$.

Memory Locations. PaCHash can be stored fully externally. By doing so, the number of I/Os for a query is increased by three (two I/Os to query the rank and select data structure on the bit vector of the Elias-Fano coding and one I/O to get the remaining bits). The number of I/Os can be reduced by interleaving the arrays of the Elias-Fano coding. PaCHash

■ **Table 10.3** External space overhead of d bits per block in order to facilitate scanning that block. The term $+1$ when $d \neq 0$ is needed for the case that no object starts in a block.

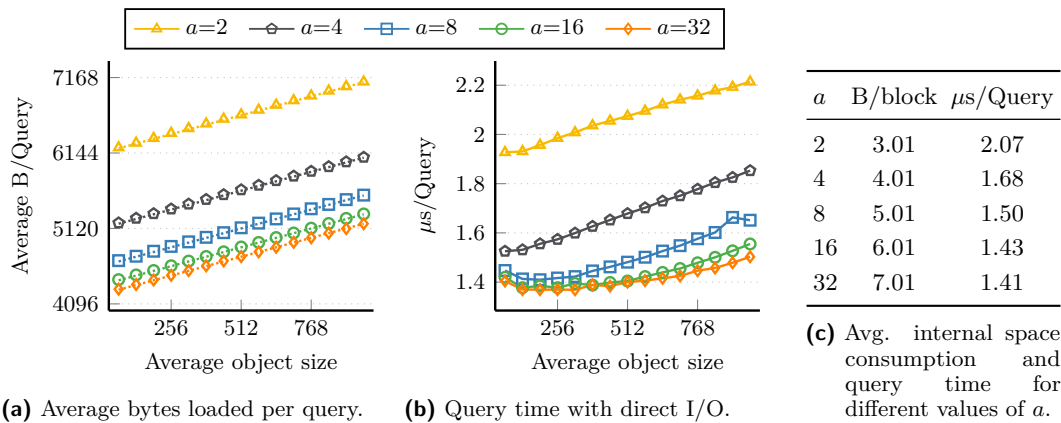
d	Case Description
0	Identical object sizes, zero terminated strings and analogous cases
$\lceil \log(w + 1) \rceil$	Objects that use variable bit-length encoding with $\leq w \leq B$ bits
$\lceil \log(W/w + 1) \rceil$	Objects of size divisible by w with $W = \min(B, \text{max object size})$
$\lceil \log(B) \rceil$	Explicit storage of a starting position of a bin

is also interesting as a purely internal data structure since it allows for configurations that need less space than any previous approach, even for objects of identical size. A variant that simplifies the external memory representation is to store the d bits of offsets per block in an internal memory data structure, possibly interleaved with the Elias-Fano representation. A variant enabling faster scanning of blocks separates keys and values [Lu+17], for example by storing $\log B$ bits of offset for each object.

Functional Enhancements. Because PaCHash sorts objects by their hashed key, it does not immediately support *range queries* over the original keys. Litwin and Lomet [LL86] implement range queries for hash tables by partitioning the key space into smaller pieces. An index tree then leads to a number of small (PaCHash) tables that are fully scanned. Order-preserving hash functions [GG86] are another alternative. PaCHash can be made *dynamic* using standard techniques like a Log-Structured Merge Tree [LC20; O’N+96]. Merging multiple PaCHash data structures is possible efficiently. The idea is to construct the hash function h by first hashing to a larger range and then mapping it linearly to the range am . When updating h to the new total number of blocks, the objects of both input data structures are already sorted and can be merged with a linear sweep.

PaCHash as Variable-Bit-Length Array. Since one of PaCHash’s key features is to store objects of variable size efficiently, it can also be used as variable-bit-length array. To this end, we simply use the array index as hash function if we also store the number of previously stored objects. However, we then have to assume that objects stored in the PaCHash VLA are self-delimiting, as this allows us to identify the objects within a block. Note that this assumption is satisfied in a lot of applications VLAs are used in, e.g., when storing variable length codes like Elias- γ and $-\delta$ codes [Eli74] or Golomb codes [Gol66]. Alternatively, in external memory, we can lift the restriction to self-delimiting objects by storing offsets as described above. The number of previously stored objects is necessary to identify the object within the block, and requires at most $\lceil \log n \rceil$ bits per external memory block.

PaCHash as Minimal k -Perfect Hash Function. The space lower bound of minimal k -perfect hashing is $\frac{1}{2} \log(2\pi k)$ bits per block (see Section 2.7). In a way, PaCHash therefore breaks the space lower bound of minimal k -perfect hashing at the cost of accessing $1/a$ blocks too much in expectation (see Lemma 10.6). Assuming objects of fixed size, about k/a of the objects in each block cannot be located exactly from the internal memory index. Whenever the index returns more than one block, we can use a retrieval data structure (see Section 2.4) to store the relative block index. This gives a minimal k -perfect hash function with a space of $2 + \log a + k/a + o(1)$ bits per block. With $a = k$, we get a minimal k -perfect hash function close to the lower bound, while inheriting the fast construction and queries from PaCHash.



■ **Figure 10.2** Dependence of I/O volume and query time on the average object size s and parameter a . Sizes are normal distributed with variance $s/5$, rounded to the next positive integer. Using other distributions gives equivalent results. Dotted lines show theoretic I/O volumes, closely matching measurements given by marks. Note that the values in Figure 10.2c do not depend on the object size.

10.5 Experiments

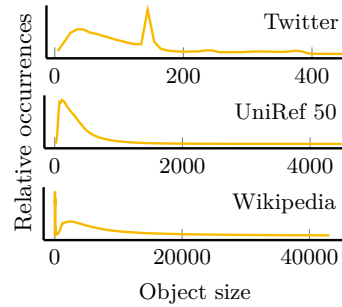
The code and scripts needed to reproduce our experiments are available on GitHub under the General Public License [Leh23d]. The code for the comparison with competitors (including our competitors' code with some patches) is available on GitHub as well [Leh23e]. The latter repository also contains a Docker image that can build and run a simplified version of the experiments in about 30 minutes.

Experimental Setup. We run our experiments on an Intel i7 11700 processor with 8 cores and a base clock speed of 2.5 GHz. We use a Samsung 980 Pro NVMe SSD with a capacity of 1 TB. The machine runs Ubuntu 21.10 with Linux 5.13.0. We use the GNU C++ compiler version 11.2.0 with optimization flags `-O3 -march=native`. Externally, each block of size $B = 2^{15}$ bits (4096 bytes) stores a table of 8 byte keys and 2 byte object offsets. During construction, we sort pointers to the objects using IPS²Ra [Axt+22]. Unless otherwise specified, the index is an Elias-Fano coded sequence based on sds1's [Gog+14] arrays of flexible bit width and the select data structures by Kurpicz [Kur22]. For the I/O operations, we use `io_uring`. Query operations keep a queue of 128 asynchronous requests in flight.

10.5.1 PaCHash Configurations

The parameter a provides a trade-off between internal space consumption and query performance, see Figure 10.2c. Figure 10.2 plots the bytes read per query, depending on the average object size and parameter a . It confirms the results of our theoretical analysis in practice. The throughput of the Elias-Fano representation increases when parameter a gets larger because the SSD needs to load fewer blocks. We also see that (at least for larger a) query times grow more slowly with object size than the I/O volume. We choose $a = 8$ for the comparison with competitors because it achieves a good balance between space consumption (≈ 5 bits/block) and throughput ($\approx 700\text{k}$ Queries/second).

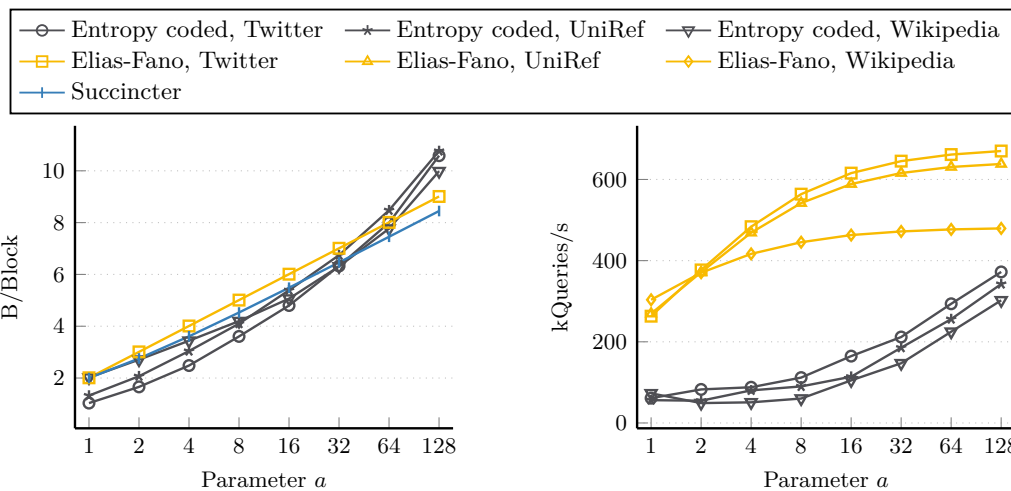
	Twitter	UniRef 50	Wikipedia
Objects n	20 238 968	48 531 431	16 181 427
Average size	115 B	281 B	1731 B
Median size	94 B	194 B	77 B
Maximum size	560 B	45 KB	272 KB
Total size N	2.4 GB	13.2 GB	26.3 GB
Objects $> B$	0%	0.08%	12%



(a) Key metrics. The median of 77 bytes of the Wikipedia data set is caused by pages that are redirects.

(b) Relative occurrences of object sizes.

■ **Figure 10.3** Real world data sets we use for the evaluation.



(a) Internal space consumption.

(b) Query throughput with direct I/O.

■ **Figure 10.4** PaCHash with real world data sets using different index data structures. There is no practical implementation of Succincter [Pua08], so we only give calculated values and no throughput. The space usage of Elias-Fano and Succincter is independent of the object size distribution, so we plot only one of the three data sets.

10.5.2 PaCHash with Real World Data Sets

Unlike minimal perfect hashing, PaCHash is influenced by the input length distribution. Figure 10.4 compares throughput and space consumption of PaCHash using real world size distributions and different index data structures. The Twitter data set contains tweets from 01.08.–05.08.2021 and has only small objects. The UniRef 50 protein database [Suz+07] contains some objects larger than the block size and the LZ4 compressed [Col] English Wikipedia from November 2021 contains significantly larger objects. See Figures 10.3a and 10.3b for details.

The entropy coded bit vector saves up to one bit of internal memory per block for small a . While it comes with a performance penalty caused by decompression (up to eight times slower than Elias-Fano), it is fast enough that it can be useful for some applications. Succincter provides space consumption lower than Elias-Fano but has no implementation. Note that for $a \leq 16$, the entropy coded bit vector requires even less space than succincter. Only for $a \geq 64$ it requires more space than Elias-Fano.

■ **Table 10.4** Configurations of competitors.

Competitor	Configuration parameters
CHD [BBD09]	Load factor 0.98. $k = 16$ collisions. Bin size 12.
Cuckoo (here, based on [PR04])	2 hash functions, loaded in parallel to reduce latency. Streamed queries (<i>await any</i>). Load factor 0.95. Random walk insertion.
LevelDB [Goo21]	No compression or Bloom filters. Construction in a single write batch.
PaCHash (here)	$a = 8$. Blocks store table of keys and offsets. Streamed queries (<i>await any</i>).
PTHash [PT21]	$\alpha = 0.94, c = 7$, D-D Encoding.
RecSplit [EGV20]	Leaf size $\ell = 8$. Bucket size $b = 2000$.
RocksDB [Fac21]	No block cache, Bloom filters, memory mapping or WAL. Queries use batches of size 64.
Separator (here, based on [GL88])	6 bit separators. Load factor 0.96. Streamed queries (<i>await any</i>).
SILT [Lim+11]	<code>testCombi.xml</code> configuration from original repository.
<code>std::unordered_map</code>	8 byte keys. 64 bit pointers to object contents.

10.5.3 Comparison with Competitors

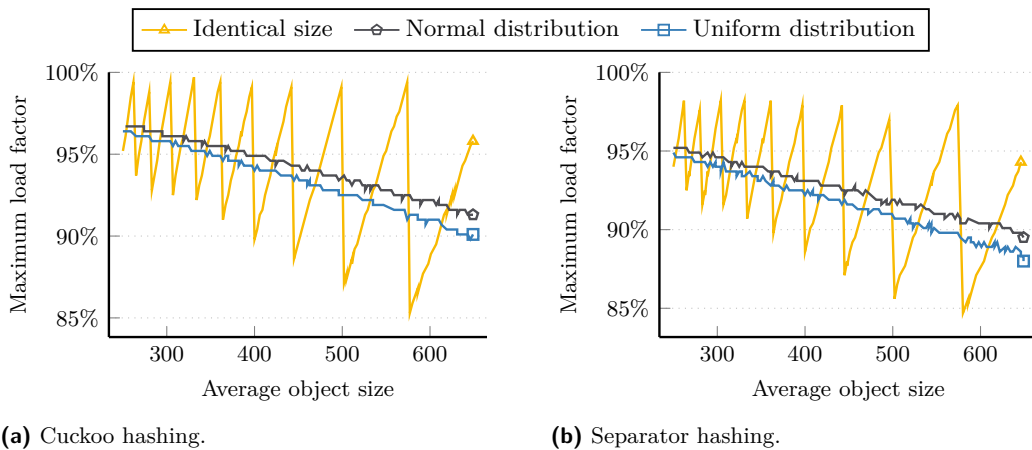
In the following section, we compare PaCHash to other data structures from the literature. First, we start with explaining our competitors.

Competitors. To our knowledge, there is no existing implementation of a hash table for variable size objects that is simultaneously aimed at low internal memory usage and few I/O operations. As the main competitors, we choose LevelDB [Goo21], RocksDB [Fac21], and SILT [Lim+11]. To abstract from the different implementations of I/O operations, we also extract the internal memory index (address calculation) from some competitors. Additionally, we compare PaCHash to `std::unordered_map`, as well as the perfect hash functions RecSplit [EGV20], CHD [BBD09; CR+12], and PTHash [PT21]. Despite `std::unordered_map` not being tuned for efficiency, it is a widely available, general purpose hash table that can be seen as baseline for simply storing pointers instead of building a compressed index data structure.¹⁰ Table 10.4 gives the exact configurations we use for each competitor.

We also implement Separator Hashing [GL88; LK84] and Cuckoo Hashing [Aza+94]. Our implementations can be used with objects of variable size $\leq B$ when setting the load factor low enough. Note that decreasing the load factor increases the number of blocks and therefore the space needed for indexing. The construction of PaCHash always succeeds, while it can fail for Separator and Cuckoo Hashing depending on the preselected load factor or tuning parameter. Figure 10.5 shows load factors between 85% and 95% in typical cases.

Comparison. Figure 10.6 shows measurements for identical size objects in order to allow for a large set of competitors. Perhaps the closest contender to PaCHash is Separator hashing where our implementation partially allows objects of variable size. It needs comparable internal space and has faster queries (always a single block access). However, Separator not only has slower construction, but it also cannot achieve a load factor close to 100% except for objects with identical size when the block size is divisible by the object size (see Figure 10.5).

¹⁰In this setting, general purpose internal memory hash tables do not work well, as they introduce an overhead of at least $\log m$ bits per object to store the positions, in addition to the object length.



■ **Figure 10.5** Maximum achievable load factor with different distributions of object sizes. For an average object size s , the normal distribution has a variance of $s/5$ and the uniform random sizes are drawn from $[0.25s, 1.75s]$.

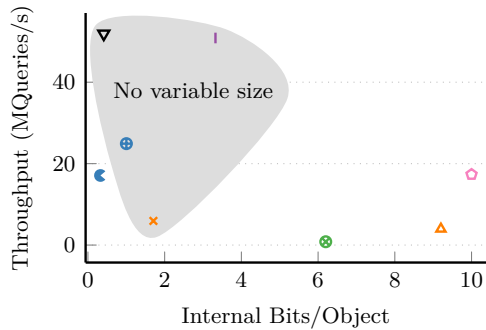
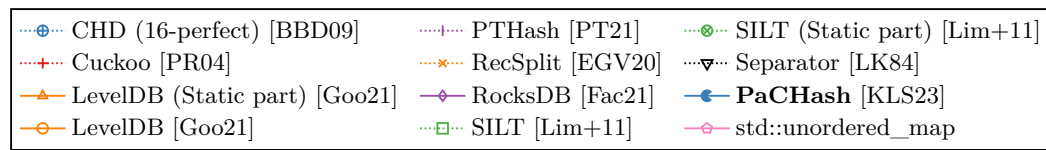
The perfect hashing methods have similar problems with respect to variable size objects and are more expensive with respect to internal space and construction costs. Cuckoo hashing needs no internal space but has more expensive queries and the same problems with variable size objects as Separator or perfect hashing.

The object stores LevelDB, RocksDB, and SILT have much larger internal space requirements *and* some external overhead. In part this comparison is unfair since they have additional functionality like dynamic operation. For SILT and LevelDB we have been able to extract the static part of the data structure. Still, they need considerably more space and have lower performance than PaCHash. Figure 10.6 contains measurements for both the full competitors and their static parts. Comparing query throughput is complicated because of different file access modes, internal caching, and history dependent performance for the actual SSD accesses (the controller uses caching and rearranges data outside the control of the user). Most competitors do not support direct I/O. For those that do, preliminary experiments show that the relative performance between the approaches stays similar. Overall, we get a consistent picture with Separator being the fastest method followed by PaCHash. A comparison with the internal hash table `std::unordered_map` is also instructive. We naturally get faster construction and high internal space consumption. Surprisingly, access to the internal data structure is only faster than PaCHash for very small inputs that fit into cache.

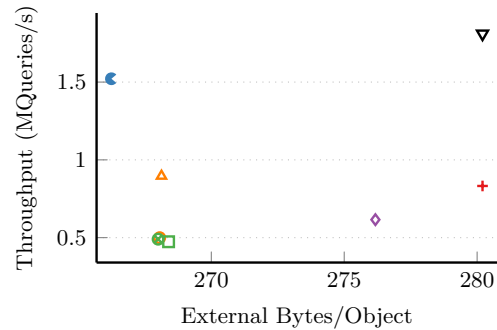
While not as surprisingly, it should be noted that all object stores supporting variable size objects do not show any difference with respect to (internal and/or external) space requirements, construction and query throughput when storing variable size objects compared to identical size objects. Thus, all benefits of PaCHash described above hold true for variable size objects as well.

10.6 Summary

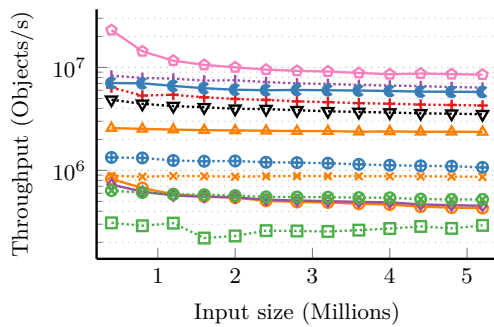
With PaCHash, we present a static external memory hash table. In contrast to a k -perfect hash function, it can space-efficiently store (possibly compressed) objects of variable size. The objects are stored contiguously without the usual need for empty space to equalize the nonuniformity in assignment by a hash function. This is facilitated by a small internal memory index data structure that needs only a constant number of bits per external memory



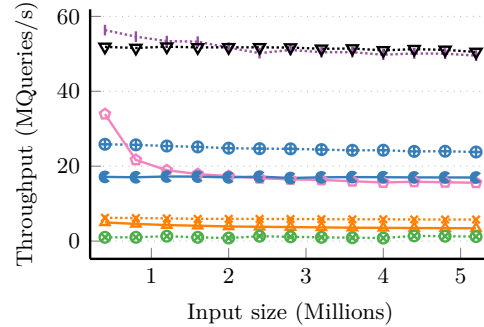
(a) Space and query throughput of internal memory index data structure. No I/O, $n = 2$ million objects. Approaches marked with gray background support only objects of fixed size.



(b) Space and query throughput of the external memory data structure. Buffered I/O, $n = 2$ million objects.



(c) Construction throughput with buffered I/O, based on the input size.



(d) Query throughput of the index data structure, based on the input size.

■ **Figure 10.6** Comparison of object stores using objects of identical size 256 bytes. Keys are 8 byte random strings. Dotted lines indicate methods supporting only objects of identical size natively.

block. In constant expected time, it yields a near-optimal range of blocks that contain the sought object. Our implementation of PaCHash considerably outperforms previous object stores for variable size objects and even matches or outperforms systems that are purely internal memory or only handle objects of identical size like perfect hash functions.

11 Conclusion

Summary: *Perfect hash functions are an important building block of many space-efficient data structures. In this dissertation, we present significant improvements to the state of the art. Our perfect hash functions get closer to the space lower bound than any competitor from the literature before. We also present algorithms that achieve very fast construction and queries. Together with a detailed evaluation, this dissertation is a significant step forward in state-of-the-art perfect hashing.*

To manage the ever-growing volumes of data, space-efficient data structures are a vital ingredient. Perfect hashing is a building block of space efficient data structures. It is an active field of research offering a large number of techniques with different trade-offs. In this dissertation, we have developed three new techniques for fast and space-efficient minimal perfect hashing. The techniques cover a wide range of trade-offs between space consumption, query throughput, and construction throughput. Our techniques dominate almost the entire Pareto front, being faster or more space-efficient than any other approach before.

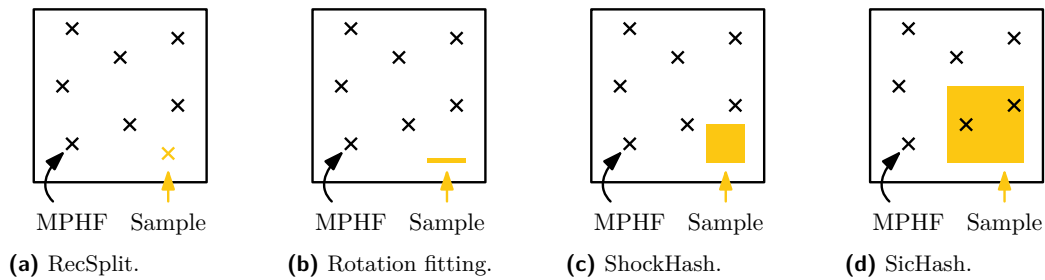
SIMDRecSplit uses brute-force and parallelizes the construction on the levels of bits, SIMD vectors, threads, and the GPU. Brute-force constructions probe random functions and hope for them to be minimal. In Figure 11.1a, we illustrate the room of all functions and the subset of perfect hash functions. A brute-force iteration simply tests one of the functions. SIMDRecSplit also contributes a step away from plain brute-force through *rotation fitting*. There we can efficiently derive additional candidate hash functions from a single probe. In Figure 11.1b, we illustrate this by probing a narrow bar in the hash function space. The main focus of SIMDRecSplit is on small space consumption, but it turns out to be also fast to construct for larger space consumption.

On the other end of the spectrum, we introduce SicHash, which offers a very directed search through constructing cuckoo hash tables. SicHash offers a good balance between construction and query time, as well as space consumption. It is not space optimal partly because its internal state can redundantly represent multiple different perfect hash functions. We illustrate this in Figure 11.1d by having each probe large enough such that it usually contains multiple perfect hash functions.

Finally, we introduce ShockHash, which is still based on brute-force but replaces a significant portion of the search by cuckoo hash table construction. Its bipartite variant is more than 2^n times faster than brute-force, while still achieving the asymptotically optimal space consumption. For a brute-force construction, ShockHash searches a considerably large space of hash function candidates at once. As we illustrate in Figure 11.1c, it usually does not find too many of them at once. This enables its good space-efficiency.

In addition to the minimal perfect hash functions, we also give an efficient *monotone* minimal perfect hash function which keeps the natural order of the input keys. LeMonHash uses a learning-based index data structure to estimate the rank of a key and resolves collisions using retrieval. For many data sets, LeMonHash is – simultaneously – more space-efficient, faster to query, and faster to construct than competing approaches.

Finally, we look at a generalization of k -perfect hashing and introduce PaCHash, an external memory hash table for objects of variable size. From a small internal memory index,



■ **Figure 11.1** Search space of different perfect hash function constructions. The boxes illustrate the space of all functions, while we mark the perfect hash functions among those.

PaCHash can determine a near optimal range of external memory blocks that need to be loaded. PaCHash has faster queries than other variable size object stores, while being more space-efficient both in internal and external memory.

Impact. The approaches presented in this dissertation make up for almost the entire Pareto front of the best query time, construction time, and space consumption. This confirms the success of the techniques presented in this dissertation and, more generally, the merits of the Algorithm Engineering methodology. Given the large number of applications (see Section 1.4), our perfect hash functions can accelerate algorithms from various research domains. Our extensive review of related work and our detailed evaluation can help to further push perfect hashing research.

Future Work. In GPURecSplit, we harness the processing power of modern GPUs to build perfect hash functions. In the future, it would be interesting to accelerate additional perfect hash functions using the GPU. While it is not the focus of this dissertation, we have introduced two k -perfect hash functions (see Sections 7.5.2 and 10.4). An interesting topic for future work would be a thorough survey on the performance of k -perfect hashing. Finally, it would be interesting to integrate the approaches developed here into real world applications to demonstrate their performance in practice.

Appendix

Publications and Supervised Theses

In Conference Proceedings

- F. Kurpicz, H.-P. Lehmann, and P. Sanders. “PaCHash: Packed and Compressed Hash Tables”. In: *ALLENEX*. SIAM, 2023, pages 162–175. DOI: 10.1137/1.9781611977561.CH14
- H.-P. Lehmann, P. Sanders, and S. Walzer. “SicHash – Small Irregular Cuckoo Tables for Perfect Hashing”. In: *ALLENEX*. SIAM, 2023, pages 176–189. DOI: 10.1137/1.9781611977561.CH15
- D. Bez, F. Kurpicz, H.-P. Lehmann, and P. Sanders. “High Performance Construction of RecSplit Based Minimal Perfect Hash Functions”. In: *ESA*. volume 274. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 19:1–19:16. DOI: 10.4230/LIPIcs.ESA.2023.19
- H.-P. Lehmann, P. Sanders, and S. Walzer. “ShockHash: Towards Optimal-Space Minimal Perfect Hashing Beyond Brute-Force”. In: *ALLENEX*. SIAM, 2024, pages 194–206. DOI: 10.1137/1.9781611977929.15
- P. Ferragina, H.-P. Lehmann, P. Sanders, and G. Vinciguerra. “Learned Monotone Minimal Perfect Hashing”. In: *ESA*. volume 274. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 46:1–46:17. DOI: 10.4230/LIPIcs.ESA.2023.46
- S. Hermann, H.-P. Lehmann, G. E. Pibiri, P. Sanders, and S. Walzer. “PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding”. In: *ESA*. volume 308. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 69:1–69:17. DOI: 10.4230/LIPIcs.ESA.2024.69

Journal Articles

- H.-P. Lehmann, P. Sanders, and S. Walzer. “ShockHash: Near Optimal-Space Minimal Perfect Hashing Beyond Brute-Force”. In: *arXiv preprint, invited to Algorithmica* (2024). DOI: 10.48550/ARXIV.2310.14959

Technical Reports

- F. Kurpicz, H.-P. Lehmann, and P. Sanders. “PaCHash: Packed and Compressed Hash Tables”. In: *CoRR* abs/2205.04745 (2022). DOI: 10.48550/ARXIV.2205.04745
- H.-P. Lehmann, P. Sanders, and S. Walzer. “SicHash – Small Irregular Cuckoo Tables for Perfect Hashing”. In: *CoRR* abs/2210.01560 (2022). DOI: 10.48550/ARXIV.2210.01560
- D. Bez, F. Kurpicz, H.-P. Lehmann, and P. Sanders. “High Performance Construction of RecSplit Based Minimal Perfect Hash Functions”. In: *CoRR* abs/2212.09562 (2022). DOI: 10.48550/ARXIV.2212.09562
- H.-P. Lehmann, P. Sanders, and S. Walzer. “ShockHash: Towards Optimal-Space Minimal Perfect Hashing Beyond Brute-Force”. In: *CoRR* abs/2308.09561 (2023). DOI: 10.48550/ARXIV.2308.09561

- P. Ferragina, H.-P. Lehmann, P. Sanders, and G. Vinciguerra. “Learned Monotone Minimal Perfect Hashing”. In: *CoRR* abs/2304.11012 (2023). DOI: 10.48550/ARXIV.2304.11012
- H.-P. Lehmann, P. Sanders, and S. Walzer. “ShockHash: Near Optimal-Space Minimal Perfect Hashing Beyond Brute-Force”. In: *CoRR* abs/2310.14959 (2024). DOI: 10.48550/ARXIV.2310.14959
- S. Hermann, H.-P. Lehmann, G. E. Pibiri, P. Sanders, and S. Walzer. “PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding”. In: *CoRR* abs/2404.18497 (2024). DOI: 10.48550/ARXIV.2404.18497

Theses

- Hans-Peter Lehmann. “Weighted Random Sampling – Alias Tables on the GPU”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2020
- Hans-Peter Lehmann. “Practicality of shoulder-surfing proof, graphical password input methods”. Bachelor’s thesis. Karlsruhe Institute of Technology (KIT), 2018

Supervised Theses

- Jonatan Ziegler. “Compacting Minimal Perfect Hashing using Symbiotic Random Search”. Bachelor’s thesis. Karlsruhe Institute of Technology (KIT), 2024
- Sebastian Georg Kirmayer. “Engineering k-perfect Hashing”. Bachelor’s thesis. Karlsruhe Institute of Technology (KIT), 2024
- Benedikt Waibel. “Cuckoo-PTHash: Exploring Cuckoo Hashing in the PTHash Framework”. Bachelor’s thesis. Karlsruhe Institute of Technology (KIT), 2024
- Stefan Hermann. “Accelerating Minimal Perfect Hash Function Construction using GPU Parallelization”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2023
- Jan Benedikt Schwarz. “Engineering Succinct Predecessor Data Structures”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2023
- Tobias Paweletz. “Compressed Bit Vectors with Rank and Select Support”. Bachelor’s thesis. Karlsruhe Institute of Technology (KIT), 2023
- Dominik Bez. “Perfect Hash Function Generation on the GPU with RecSplit”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2022

Bibliography

- [Ada+21] D. Adamson, A. Deligkas, V. V. Gusev, and I. Potapov. “Combinatorial algorithms for multidimensional necklaces”. In: *CoRR* abs/2108.01990 (2021). DOI: 10.48550/ARXIV.2108.01990.
- [AFK23] S. Assadi, M. Farach-Colton, and W. Kuszmaul. “Tight Bounds for Monotone Minimal Perfect Hashing”. In: *SODA*. SIAM, 2023, pages 456–476. DOI: 10.1137/1.9781611977554.CH20.
- [AIS93] R. Agrawal, T. Imielinski, and A. N. Swami. “Mining Association Rules between Sets of Items in Large Databases”. In: *SIGMOD Conference*. ACM Press, 1993, pages 207–216. DOI: 10.1145/170035.170072.
- [AKS15] R. Agarwal, A. Khandelwal, and I. Stoica. “Succinct: Enabling Queries on Compressed Data”. In: *NSDI*. USENIX Association, 2015, pages 337–350.
- [Alm+18] F. Almodaresi, H. Sarkar, A. Srivastava, and R. Patro. “A space and time-efficient index for the compacted colored de Bruijn graph”. In: *Bioinform.* 34.13 (2018), pages i169–i177. DOI: 10.1093/BIOINFORMATICS/BTY292.
- [And+09] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. “FAWN: a fast array of wimpy nodes”. In: *SOSP*. ACM, 2009, pages 1–14. DOI: 10.1145/1629575.1629577.
- [ANS10] Y. Arbitman, M. Naor, and G. Segev. “Backyard Cuckoo Hashing: Constant Worst-Case Operations with a Succinct Representation”. In: *FOCS*. IEEE Computer Society, 2010, pages 787–796. DOI: 10.1109/FOCS.2010.80.
- [App10] A. Appleby. *SMHasher*. <https://github.com/rurban/smhasher>. 2010.
- [Axt+22] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. “Engineering In-place (Shared-memory) Sorting Algorithms”. In: *ACM Trans. Parallel Comput.* 9.1 (2022), 2:1–2:62. DOI: 10.1145/3505286.
- [Aza+94] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. “Balanced allocations (extended abstract)”. In: *STOC*. ACM, 1994, pages 593–602. DOI: 10.1145/195058.195412.
- [Bas+06] H. Bast, K. Mehlhorn, G. Schäfer, and H. Tamaki. “Matching Algorithms Are Fast in Sparse Random Graphs”. In: *Theory Comput. Syst.* 39.1 (2006), pages 3–14. DOI: 10.1007/S00224-005-1254-Y.
- [BB08] D. K. Blandford and G. E. Blelloch. “Compact dictionaries for variable-length keys and data with applications”. In: *ACM Trans. Algorithms* 4.2 (2008), 17:1–17:25. DOI: 10.1145/1361192.1361194.
- [BBD09] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger. “Hash, Displace, and Compress”. In: *ESA*. Volume 5757. Lecture Notes in Computer Science. Springer, 2009, pages 682–693. DOI: 10.1007/978-3-642-04128-0_61.
- [BCO11] A. Boldyreva, N. Chenette, and A. O’Neill. “Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions”. In: *CRYPTO*. Volume 6841. Lecture Notes in Computer Science. Springer, 2011, pages 578–595. DOI: 10.1007/978-3-642-22792-9_33.

- [Bea+10] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. “Finding a Needle in Haystack: Facebook’s Photo Storage”. In: *OSDI*. USENIX Association, 2010, pages 47–60.
- [Bel+09] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. “Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses”. In: *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2009, pages 785–794. DOI: 10.1137/1.9781611973068.86.
- [Bel+10] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. “Fast Prefix Search in Little Space, with Applications”. In: *ESA (1)*. Volume 6346. Lecture Notes in Computer Science. Springer, 2010, pages 427–438. DOI: 10.1007/978-3-642-15775-2_37.
- [Bel+11] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. “Theory and practice of monotone minimal perfect hashing”. In: *ACM J. Exp. Algorithmics* 16 (2011). DOI: 10.1145/1963190.2025378.
- [Bel+14] D. Belazzougui, P. Boldi, G. Ottaviano, R. Venturini, and S. Vigna. “Cache-Oblivious Peeling of Random Hypergraphs”. In: *DCC*. IEEE, 2014, pages 352–361. DOI: 10.1109/DCC.2014.48.
- [Bel+20] D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. “Linear-time String Indexing and Analysis in Small Space”. In: *ACM Trans. Algorithms* 16.2 (2020), 17:1–17:54. DOI: 10.1145/3381417.
- [Bel23] P. Beling. “Fingerprinting-based Minimal Perfect Hashing Revisited”. In: *ACM J. Exp. Algorithmics* 28 (2023), 1.4:1–1.4:16. DOI: 10.1145/3596453.
- [Ben+18] M. A. Bender, M. Farach-Colton, M. Goswami, R. Johnson, S. McCauley, and S. Singh. “Bloom Filters, Adaptivity, and the Dictionary Problem”. In: *FOCS*. IEEE Computer Society, 2018, pages 182–193. DOI: 10.1109/FOCS.2018.00026.
- [Ben+23] M. A. Bender, A. Conway, M. Farach-Colton, W. Kuzmaul, and G. Tagliavini. “Iceberg Hashing: Optimizing Many Hash-Table Criteria at Once”. In: *J. ACM* 70.6 (2023), 40:1–40:51. DOI: 10.1145/3625817.
- [Ber24] S. Bernstein. “On a modification of Chebyshev’s inequality and of the error formula of Laplace”. In: *Ann. Sci. Inst. Sav. Ukraine, Sect. Math* 1.4 (1924), pages 38–49.
- [Bez22] D. Bez. “Perfect Hash Function Generation on the GPU with RecSplit”. Master’s thesis. Karlsruhe Institute for Technology (KIT), 2022. DOI: 10.5445/IR/1000152719.
- [Bez+22] D. Bez, F. Kurpicz, H.-P. Lehmann, and P. Sanders. “High Performance Construction of RecSplit Based Minimal Perfect Hash Functions”. In: *CoRR* abs/2212.09562 (2022). DOI: 10.48550/ARXIV.2212.09562.
- [Bez+23] D. Bez, F. Kurpicz, H.-P. Lehmann, and P. Sanders. “High Performance Construction of RecSplit Based Minimal Perfect Hash Functions”. In: *ESA*. Volume 274. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 19:1–19:16. DOI: 10.4230/LIPICSA.2023.19.
- [BFV22] A. Boffa, P. Ferragina, and G. Vinciguerra. “A Learned Approach to Design Compressed Rank/Select Data Structures”. In: *ACM Trans. Algorithms* 18.3 (2022), 24:1–24:28. DOI: 10.1145/3524060.
- [BGJ06] J. Bruck, J. Gao, and A. Jiang. “Weighted Bloom filter”. In: *ISIT*. IEEE, 2006, pages 2304–2308. DOI: 10.1109/ISIT.2006.261978.
- [BGZ04] F. C. Botelho, D. M. Gomes, and N. Ziviani. “A new algorithm for constructing minimal perfect hash functions”. In: *differences* 100.2 (2004), page 09.

- [Blo70] B. H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Commun. ACM* 13.7 (1970), pages 422–426. DOI: 10.1145/362686.362692.
- [BM03] A. Z. Broder and M. Mitzenmacher. “Survey: Network Applications of Bloom Filters: A Survey”. In: *Internet Math.* 1.4 (2003), pages 485–509. DOI: 10.1080/15427951.2004.10129096.
- [BN14] D. Belazzougui and G. Navarro. “Alphabet-Independent Compressed Text Indexing”. In: *ACM Trans. Algorithms* 10.4 (2014), 23:1–23:19. DOI: 10.1145/2635816.
- [BN15] D. Belazzougui and G. Navarro. “Optimal Lower and Upper Bounds for Representing Sequences”. In: *ACM Trans. Algorithms* 11.4 (2015), 31:1–31:21. DOI: 10.1145/2629339.
- [BNV13] D. Belazzougui, G. Navarro, and D. Valenzuela. “Improved compressed indexes for full-text document retrieval”. In: *J. Discrete Algorithms* 18 (2013), pages 3–13. DOI: 10.1016/J.JDA.2012.07.005.
- [Bof+22] A. Boffa, P. Ferragina, F. Tosoni, and G. Vinciguerra. “Compressed String Dictionaries via Data-Aware Subtrie Compaction”. In: *SPIRE*. Volume 13617. Lecture Notes in Computer Science. Springer, 2022, pages 233–249. DOI: 10.1007/978-3-031-20643-6_17.
- [Bot+08] F. C. Botelho, H. R. Langbehn, G. V. Menezes, and N. Ziviani. “Indexing Internal Memory with Minimal Perfect Hash Functions”. In: *SBBD*. SBC, 2008, pages 16–30. DOI: 10.5555/1498932.1498935.
- [BPZ07a] F. C. Botelho, R. Pagh, and N. Ziviani. “Perfect Hashing for Data Management Applications”. In: *CoRR* abs/cs/0702159 (2007). DOI: 10.48550/arXiv.cs/0702159.
- [BPZ07b] F. C. Botelho, R. Pagh, and N. Ziviani. “Simple and Space-Efficient Minimal Perfect Hash Functions”. In: *WADS*. Volume 4619. Lecture Notes in Computer Science. Springer, 2007, pages 139–150. DOI: 10.1007/978-3-540-73951-7_13.
- [BPZ13] F. C. Botelho, R. Pagh, and N. Ziviani. “Practical perfect hashing in nearly optimal space”. In: *Inf. Syst.* 38.1 (2013), pages 108–131. DOI: 10.1016/J.IS.2012.06.002.
- [BSV08] P. Boldi, M. Santini, and S. Vigna. “A large time-aware web graph”. In: *SIGIR Forum* 42.2 (2008), pages 33–38. DOI: 10.1145/1480506.1480511.
- [Bur94] M. Burrows. “A block-sorting lossless data compression algorithm”. In: *SRS Research Report* 124 (1994).
- [BZ07] F. C. Botelho and N. Ziviani. “External perfect hashing for very large key sets”. In: *CIKM*. ACM, 2007, pages 653–662. DOI: 10.1145/1321440.1321532.
- [Cay78] A. Cayley. “A theorem on trees”. In: *Quart. J. Math.* 23 (1878), pages 376–378.
- [Cel88] P. Celia. *External Robin Hood Hashing*. Technical report. Computer Science Department, Indiana University. TR246, 1988.
- [Cha+11] J. A. Chapman, I. Ho, S. Sunkara, S. Luo, G. P. Schroth, and D. S. Rokhsar. “Meraculous: De Novo Genome Assembly with Short Paired-End Reads”. In: *PLOS ONE* 6.8 (Aug. 2011), pages 1–13. DOI: 10.1371/journal.pone.0023501. URL: <https://doi.org/10.1371/journal.pone.0023501>.
- [Che+21] X. Chen, N. Zheng, S. Xu, Y. Qiao, Y. Liu, J. Li, and T. Zhang. “KallaxDB: A Table-less Hash-based Key-Value Store on Storage Hardware with Built-in Transparent Compression”. In: *DaMoN*. ACM, 2021, 3:1–3:10. DOI: 10.1145/3465998.3466004.

- [CHM92] Z. J. Czech, G. Havas, and B. S. Majewski. “An Optimal Algorithm for Generating Minimal Perfect Hash Functions”. In: *Inf. Process. Lett.* 43.5 (1992), pages 257–264. DOI: 10.1016/0020-0190(92)90220-P.
- [CHM97] Z. J. Czech, G. Havas, and B. S. Majewski. “Perfect hashing”. In: *Theoretical Computer Science* 182.1-2 (1997), pages 1–143. DOI: 10.1016/S0304-3975(96)00146-6.
- [Cic80] R. J. Cichelli. “Minimal Perfect Hash Functions Made Simple”. In: *Commun. ACM* 23.1 (1980), pages 17–19. DOI: 10.1145/358808.358813.
- [CL05] C. Chang and C. Lin. “Perfect Hashing Schemes for Mining Association Rules”. In: *Comput. J.* 48.2 (2005), pages 168–179. DOI: 10.1093/COMJNL/BXH074.
- [Cla96] D. R. Clark. “Compact Pat Trees”. PhD thesis. University of Waterloo, Canada, 1996.
- [CLM16] R. Chikhi, A. Limasset, and P. Medvedev. “Compacting de Bruijn graphs from sequencing data quickly and in low memory”. In: *Bioinform.* 32.12 (2016), pages 201–208. DOI: 10.1093/BIOINFORMATICS/BTW279.
- [CMV20] K. Chung, M. Mitzenmacher, and S. P. Vadhan. “When Simple Hash Functions Suffice”. In: *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, 2020, pages 567–585. DOI: 10.1017/9781108637435.033.
- [Col] Y. Collet. *LZ4: Extremely Fast Compression algorithm*. <https://github.com/lz4/lz4>.
- [Com79] D. Comer. “The Ubiquitous B-Tree”. In: *ACM Comput. Surv.* 11.2 (1979), pages 121–137. DOI: 10.1145/356770.356776.
- [CR+12] D. de Castro Reis, D. Belazzougui, F. C. Botelho, and N. Ziviani. *CMPH - C Minimal Perfect Hashing Library*. <http://cmph.sourceforge.net/>. 2012.
- [CSM13] Y. Chen, B. Schmidt, and D. L. Maskell. “A hybrid short read mapping accelerator”. In: *BMC Bioinform.* 14 (2013), page 67. DOI: 10.1186/1471-2105-14-67.
- [DH90] M. Dietzfelbinger and F. M. auf der Heide. “A New Universal Class of Hash Functions and Dynamic Hashing in Real Time”. In: *ICALP*. Volume 443. Lecture Notes in Computer Science. Springer, 1990, pages 6–19. DOI: 10.1007/BFB0032018.
- [Die+10] M. Dietzfelbinger, A. Goerdts, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink. “Tight Thresholds for Cuckoo Hashing via XORSAT”. In: *ICALP (1)*. Volume 6198. Lecture Notes in Computer Science. Springer, 2010, pages 213–225. DOI: 10.1007/978-3-642-14165-2_19.
- [Die+94] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. “Dynamic Perfect Hashing: Upper and Lower Bounds”. In: *SIAM J. Comput.* 23.4 (1994), pages 738–761. DOI: 10.1137/S0097539791194094.
- [Die+97] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. “A Reliable Randomized Algorithm for the Closest-Pair Problem”. In: *J. Algorithms* 25.1 (1997), pages 19–51. DOI: 10.1006/JAGM.1997.0873.
- [Dil+22] P. C. Dillinger, L. Hübschle-Schneider, P. Sanders, and S. Walzer. “Fast Succinct Retrieval and Approximate Membership Using Ribbon”. In: *SEA*. Volume 233. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 4:1–4:20. DOI: 10.4230/LIPICS.SEA.2022.4.
- [Din+20] P. Dinklage, J. Fischer, A. Herlez, T. Kociumaka, and F. Kurpicz. “Practical Performance of Space Efficient Data Structures for Longest Common Extensions”. In: *ESA*. Volume 173. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 39:1–39:20. DOI: 10.4230/LIPICS.ESA.2020.39.

- [DMR11] M. Dietzfelbinger, M. Mitzenmacher, and M. Rink. “Cuckoo Hashing with Pages”. In: *ESA*. Volume 6942. Lecture Notes in Computer Science. Springer, 2011, pages 615–627. DOI: 10.1007/978-3-642-23719-5_52.
- [DR09] M. Dietzfelbinger and M. Rink. “Applications of a Splitting Trick”. In: *ICALP (1)*. Volume 5555. Lecture Notes in Computer Science. Springer, 2009, pages 354–365. DOI: 10.1007/978-3-642-02927-1_30.
- [DSL11] B. K. Debnath, S. Sengupta, and J. Li. “SkimpyStash: RAM space skimpy key-value store on flash-based storage”. In: *SIGMOD Conference*. ACM, 2011, pages 25–36. DOI: 10.1145/1989323.1989327.
- [DW07] M. Dietzfelbinger and C. Weidling. “Balanced allocation and dictionaries with tightly packed constant size bins”. In: *Theor. Comput. Sci.* 380.1-2 (2007), pages 47–68. DOI: 10.1016/J.TCS.2007.02.054.
- [EGV20] E. Esposito, T. M. Graf, and S. Vigna. “RecSplit: Minimal Perfect Hashing via Recursive Splitting”. In: *ALENEX*. SIAM, 2020, pages 175–185. DOI: 10.1137/1.9781611976007.14.
- [Eli74] P. Elias. “Efficient Storage and Retrieval by Content and Address of Static Files”. In: *J. ACM* 21.2 (1974), pages 246–260. DOI: 10.1145/321812.321820.
- [ESS08] S. Edelkamp, P. Sanders, and P. Simecek. “Semi-external LTL Model Checking”. In: *CAV*. Volume 5123. Lecture Notes in Computer Science. Springer, 2008, pages 530–542. DOI: 10.1007/978-3-540-70545-1_50.
- [Fac21] Facebook. *RocksDB. A Persistent Key-Value Store for Fast Storage Environments*. <https://rocksdb.org>. 2021.
- [Fag+79] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. “Extendible Hashing - A Fast Access Method for Dynamic Files”. In: *ACM Trans. Database Syst.* 4.3 (1979), pages 315–344. DOI: 10.1145/320083.320092.
- [Fan+14] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. “Cuckoo Filter: Practically Better Than Bloom”. In: *CoNEXT*. ACM, 2014, pages 75–88. DOI: 10.1145/2674005.2674994.
- [Fan71] R. M. Fano. *On the number of bits required to implement an associative memory*. Technical report. Project MAC, Memorandum 61". MIT, Computer Structures Group, 1971.
- [FCH92] E. A. Fox, Q. F. Chen, and L. S. Heath. “A Faster Algorithm for Constructing Minimal Perfect Hash Functions”. In: *SIGIR*. ACM, 1992, pages 266–273. DOI: 10.1145/133160.133209.
- [Fer+08] P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter. “On searching compressed string collections cache-obliviously”. In: *PODS*. ACM, 2008, pages 181–190. DOI: 10.1145/1376916.1376943.
- [Fer+23a] P. Ferragina, H.-P. Lehmann, P. Sanders, and G. Vinciguerra. “Learned Monotone Minimal Perfect Hashing”. In: *ESA*. Volume 274. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 46:1–46:17. DOI: 10.4230/LIPICS.ESA.2023.46.
- [Fer+23b] P. Ferragina, H.-P. Lehmann, P. Sanders, and G. Vinciguerra. “Learned Monotone Minimal Perfect Hashing”. In: *CoRR* abs/2304.11012 (2023). DOI: 10.48550/ARXIV.2304.11012.
- [FK16] A. Frieze and M. Karoński. *Introduction to random graphs*. Cambridge University Press, 2016. DOI: 10.1017/CB09781316339831.

- [FKP16] N. Fountoulakis, M. Khosla, and K. Panagiotou. “The Multiple-Orientability Thresholds for Random Hypergraphs”. In: *Comb. Probab. Comput.* 25.6 (2016), pages 870–908. DOI: 10.1017/S0963548315000334.
- [FKS84] M. L. Fredman, J. Komlós, and E. Szemerédi. “Storing a Sparse Table with $O(1)$ Worst Case Access Time”. In: *J. ACM* 31.3 (1984), pages 538–544. DOI: 10.1145/828.1884.
- [FLV21] P. Ferragina, F. Lillo, and G. Vinciguerra. “On the performance of learned data structures”. In: *Theor. Comput. Sci.* 871 (2021), pages 107–120. DOI: 10.1016/J.TCS.2021.04.015.
- [Fly72] M. J. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *IEEE Trans. Computers* 21.9 (1972), pages 948–960. DOI: 10.1109/TC.1972.5009071.
- [FM00] P. Ferragina and G. Manzini. “Opportunistic Data Structures with Applications”. In: *FOCS*. IEEE Computer Society, 2000, pages 390–398. DOI: 10.1109/SFCS.2000.892127.
- [FM05] P. Ferragina and G. Manzini. “Indexing compressed text”. In: *J. ACM* 52.4 (2005), pages 552–581. DOI: 10.1145/1082036.1082039.
- [FMM09] A. M. Frieze, P. Melsted, and M. Mitzenmacher. “An Analysis of Random-Walk Cuckoo Hashing”. In: *APPROX-RANDOM*. Volume 5687. Lecture Notes in Computer Science. Springer, 2009, pages 490–503. DOI: 10.1007/978-3-642-03685-9_37.
- [FMV22] P. Ferragina, G. Manzini, and G. Vinciguerra. “Compressing and Querying Integer Dictionaries Under Linearities and Repetitions”. In: *IEEE Access* 10 (2022), pages 118831–118848. DOI: 10.1109/ACCESS.2022.3221520.
- [FN] P. Ferragina and G. Navarro. *Pizza&Chili Corpus*. Accessed: February 2023. URL: <http://pizzachili.dcc.uchile.cl/texts.html>.
- [FN07] K. Fredriksson and F. Nikitin. “Simple Compression Code Supporting Random Access and Fast String Matching”. In: *WEA*. Volume 4525. Lecture Notes in Computer Science. Springer, 2007, pages 203–216. DOI: 10.1007/978-3-540-72845-0_16.
- [Fog13] A. Fog. *C++ vector class library*. <http://www.agner.org/optimize/vectorclass.pdf>. 2013.
- [Fot+05] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis. “Space Efficient Hash Tables with Worst Case Constant Access Time”. In: *Theory Comput. Syst.* 38.2 (2005), pages 229–248. DOI: 10.1007/S00224-004-1195-X.
- [Fox+91] E. A. Fox, Q. F. Chen, A. M. Daoud, and L. S. Heath. “Order-Preserving Minimal Perfect Hash Functions and Information Retrieval”. In: *ACM Trans. Inf. Syst.* 9.3 (1991), pages 281–308. DOI: 10.1145/125187.125200.
- [FP12] N. Fountoulakis and K. Panagiotou. “Sharp load thresholds for cuckoo hashing”. In: *Random Struct. Algorithms* 41.3 (2012), pages 306–333. DOI: 10.1002/RSA.20426.
- [FPS13] N. Fountoulakis, K. Panagiotou, and A. Steger. “On the Insertion Time of Cuckoo Hashing”. In: *SIAM J. Comput.* 42.6 (2013), pages 2156–2181. DOI: 10.1137/100797503.
- [Fri+99] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. “Cache-Oblivious Algorithms”. In: *FOCS*. IEEE Computer Society, 1999, pages 285–298. DOI: 10.1109/SFFCS.1999.814600.

- [FV20a] P. Ferragina and G. Vinciguerra. “Learned Data Structures”. In: *Recent Trends in Learning From Data*. Edited by L. Oneto, N. Navarin, A. Sperduti, and D. Anguita. Springer International Publishing, 2020, pages 5–41. ISBN: 978-3-030-43883-8. DOI: 10.1007/978-3-030-43883-8_2.
- [FV20b] P. Ferragina and G. Vinciguerra. “The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds”. In: *Proc. VLDB Endow.* 13.8 (2020), pages 1162–1175. DOI: 10.14778/3389133.3389135.
- [GG86] A. K. Garg and C. C. Gotlieb. “Order-Preserving Key Transformations”. In: *ACM Trans. Database Syst.* 11.2 (1986), pages 213–234. DOI: 10.1145/5922.5923.
- [GL20] T. M. Graf and D. Lemire. “Xor filters: Faster and smaller than bloom and cuckoo filters”. In: *Journal of Experimental Algorithmics (JEA)* 25 (2020), pages 1–16. DOI: 10.1145/3376122.
- [GL88] G. H. Gonnet and P. Larson. “External hashing with limited internal storage”. In: *J. ACM* 35.1 (1988), pages 161–184. DOI: 10.1145/42267.42274.
- [GNP20] T. Gagie, G. Navarro, and N. Prezza. “Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space”. In: *J. ACM* 67.1 (2020), 2:1–2:54. DOI: 10.1145/3375890.
- [GO14] R. Grossi and G. Ottaviano. “Fast Compressed Tries through Path Decompositions”. In: *ACM J. Exp. Algorithmics* 19.1 (2014). DOI: 10.1145/2656332.
- [Gog+14] S. Gog, T. Beller, A. Moffat, and M. Petri. “From Theory to Practice: Plug and Play with Succinct Data Structures”. In: *SEA*. Volume 8504. Lecture Notes in Computer Science. Springer, 2014, pages 326–337. DOI: 10.1007/978-3-319-07959-2_28.
- [Gol66] S. W. Golomb. “Run-length Encodings”. In: *IEEE Trans. Inf. Theory* 12.3 (1966), pages 399–401. DOI: 10.1109/TIT.1966.1053907.
- [Goo] Google. *Google Ngram Exports*. Accessed: March 2023. URL: <https://storage.googleapis.com/books/ngrams/books/datasetsv3.html>.
- [Goo21] Google. *LevelDB is a Fast Key-Value Storage Library Written at Google*. <https://github.com/google/leveldb>. 2021.
- [GOR10] R. Grossi, A. Orlandi, and R. Raman. “Optimal Trade-Offs for Succinct String Indexes”. In: *ICALP (1)*. Volume 6198. Lecture Notes in Computer Science. Springer, 2010, pages 678–689. DOI: 10.1007/978-3-642-14165-2_57.
- [GOV16] M. Genuzio, G. Ottaviano, and S. Vigna. “Fast Scalable Construction of (Minimal Perfect Hash) Functions”. In: *SEA*. Volume 9685. Lecture Notes in Computer Science. Springer, 2016, pages 339–352. DOI: 10.1007/978-3-319-38851-9_23.
- [GT63] M. Greniewski and W. M. Turski. “The external language KLIPA for the URAL-2 digital computer”. In: *Commun. ACM* 6.6 (1963), pages 321–324. DOI: 10.1145/366604.366654.
- [Her23] S. Hermann. “Accelerating Minimal Perfect Hash Function Construction Using GPU Parallelization”. Master’s thesis. Karlsruhe Institute for Technology (KIT), 2023. DOI: 10.5445/IR/1000164413.
- [Her+24a] S. Hermann, H.-P. Lehmann, G. E. Pibiri, P. Sanders, and S. Walzer. “PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding”. In: *ESA*. Volume 308. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 69:1–69:17. DOI: 10.4230/LIPIcs.ESA.2024.69.

- [Her+24b] S. Hermann, H.-P. Lehmann, G. E. Pibiri, P. Sanders, and S. Walzer. “PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding”. In: *CoRR* abs/2404.18497 (2024). DOI: 10.48550/ARXIV.2404.18497.
- [HK73] J. E. Hopcroft and R. M. Karp. “An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs”. In: *SIAM J. Comput.* 2.4 (1973), pages 225–231.
- [HT01] T. Hagerup and T. Tholey. “Efficient Minimal Perfect Hashing in Nearly Minimal Space”. In: *STACS*. Volume 2010. Lecture Notes in Computer Science. Springer, 2001, pages 317–326. DOI: 10.1007/3-540-44693-1_28.
- [HTT02] G.-J. Hwang, W. Tsai, and J. C. Tseng. “A minimal perfect hashing approach for mining association rules from very large databases”. In: *The 6th IASTED International Conference on Internet and Multimedia Systems and Applications, Kaua’i, Hawaii, USA*. 2002, pages 80–85.
- [Int11] Intel. *Advanced Vector Extensions Programming Reference*. <https://www.intel.com/content/dam/develop/external/us/en/documents/36945>. 2011.
- [Int13] Intel. *AVX-512 Instructions*. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html>. 2013.
- [Jac89] G. Jacobson. “Space-efficient Static Trees and Graphs”. In: *FOCS*. IEEE Computer Society, 1989, pages 549–554. DOI: 10.1109/SFCS.1989.63533.
- [Jae81] G. Jaeschke. “Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions”. In: *Commun. ACM* 24.12 (1981), pages 829–833. DOI: 10.1145/358800.358806.
- [JDP83] K. Joag-Dev and F. Proschan. “Negative Association of Random Variables with Applications”. In: *The Annals of Statistics* 11.1 (1983), pages 286–295. DOI: 10.1214/aos/1176346079.
- [Jen06] J. L. W. V. Jensen. “Sur les fonctions convexes et les inégalités entre les valeurs moyennes”. In: *Acta mathematica* 30.1 (1906), pages 175–193. DOI: 10.1007/BF02418571.
- [JL07] S. Janson and M. J. Luczak. “A simple solution to the k -core problem”. In: *Random Struct. Algorithms* 30.1-2 (2007), pages 50–62. DOI: 10.1002/rsa.20147.
- [JP08] M. S. Jensen and R. Pagh. “Optimality in External Memory Hashing”. In: *Algorithmica* 52.3 (2008), pages 403–411. DOI: 10.1007/S00453-007-9155-X.
- [KA19] M. Khosla and A. Anand. “A Faster Algorithm for Cuckoo Insertion and Bipartite Matching in Large Graphs”. In: *Algorithmica* 81.9 (2019), pages 3707–3724. DOI: 10.1007/S00453-019-00595-4.
- [Kho13] M. Khosla. “Balls into Bins Made Faster”. In: *ESA*. Volume 8125. Lecture Notes in Computer Science. Springer, 2013, pages 601–612. DOI: 10.1007/978-3-642-40450-4_51.
- [Kip+19] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. “SOSD: A Benchmark for Learned Indexes”. In: *CoRR* abs/1911.13014 (2019). DOI: 10.48550/ARXIV.1911.13014.
- [KLS22] F. Kurpicz, H.-P. Lehmann, and P. Sanders. “PaCHash: Packed and Compressed Hash Tables”. In: *CoRR* abs/2205.04745 (2022). DOI: 10.48550/ARXIV.2205.04745.
- [KLS23] F. Kurpicz, H.-P. Lehmann, and P. Sanders. “PaCHash: Packed and Compressed Hash Tables”. In: *ALLENEX*. SIAM, 2023, pages 162–175. DOI: 10.1137/1.9781611977561.CH14.

- [Knu73] D. E. Knuth. *The art of computer programming: Volume 3: Sorting and Searching (First edition)*. Addison-Wesley, 1973. ISBN: 0-201-03803-X.
- [Knu98] D. E. Knuth. *The art of computer programming: Volume 3: Sorting and Searching (Second edition)*. Addison-Wesley, 1998. ISBN: 978-0-201-89685-5.
- [Kos24] D. Kosolobov. “Simplified Tight Bounds for Monotone Minimal Perfect Hashing”. In: *CPM*. Volume 296. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 19:1–19:13. DOI: 10.4230/LIPICS.CPM.2024.19.
- [KPR22] D. Köppl, S. J. Puglisi, and R. Raman. “Fast and Simple Compact Hashing via Bucketing”. In: *Algorithmica* 84.9 (2022), pages 2735–2766. DOI: 10.1007/S00453-022-00996-Y.
- [Kra+18] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. “The Case for Learned Index Structures”. In: *SIGMOD Conference*. ACM, 2018, pages 489–504. DOI: 10.1145/3183713.3196909.
- [KRT22] E. M. Kornaropoulos, S. Ren, and R. Tamassia. “The Price of Tailoring the Index to Your Data: Poisoning Attacks on Learned Index Structures”. In: *SIGMOD Conference*. ACM, 2022, pages 1331–1344. DOI: 10.1145/3514221.3517867.
- [Kur22] F. Kurpicz. “Engineering Compact Data Structures for Rank and Select Queries on Bit Vectors”. In: *SPIRE*. Volume 13617. Lecture Notes in Computer Science. Springer, 2022, pages 257–272. DOI: 10.1007/978-3-031-20643-6_19.
- [Kus16] W. Kuzmaul. “Fast Concurrent Cuckoo Kick-Out Eviction Schemes for High-Density Tables”. In: *CoRR* abs/1605.05236 (2016). DOI: 10.48550/arXiv.1605.05236.
- [Kü14] M. O. Külekci. “Enhanced Variable-Length Codes: Improved Compression with Efficient Random Access”. In: *DCC*. IEEE, 2014, pages 362–371. DOI: 10.1109/DCC.2014.74.
- [Lar88] P. Larson. “Linear Hashing with Separators - A Dynamic Hashing Scheme Achieving One-Access Retrieval”. In: *ACM Trans. Database Syst.* 13.3 (1988), pages 366–388. DOI: 10.1145/44498.44500.
- [LC20] C. Luo and M. J. Carey. “LSM-based storage techniques: a survey”. In: *VLDB J.* 29.1 (2020), pages 393–418. DOI: 10.1007/s00778-019-00555-y.
- [LC88] T. G. Lewis and C. R. Cook. “Hashing for Dynamic and Static Internal Tables”. In: *Computer* 21.10 (1988), pages 45–56. DOI: 10.1109/2.7056.
- [Leh23a] H.-P. Lehmann. *GpuRecSplit - GitHub*. <https://github.com/ByteHamster/GpuRecSplit>. 2023.
- [Leh23b] H.-P. Lehmann. *MMPHF-Experiments - GitHub*. <https://github.com/ByteHamster/MMPHF-Experiments>. 2023.
- [Leh23c] H.-P. Lehmann. *MPHF Experiments - GitHub*. <https://github.com/ByteHamster/MPHF-Experiments>. 2023.
- [Leh23d] H.-P. Lehmann. *PaCHash - GitHub*. <https://github.com/ByteHamster/PaCHash>. 2023.
- [Leh23e] H.-P. Lehmann. *PaCHash Experiments - GitHub*. <https://github.com/ByteHamster/PaCHash-Experiments>. 2023.
- [Leh23f] H.-P. Lehmann. *SicHash - GitHub*. <https://github.com/ByteHamster/SicHash>. 2023.
- [Leh24] H.-P. Lehmann. *ShockHash - GitHub*. <https://github.com/ByteHamster/ShockHash>. 2024.
- [Lel12] M. Lelarge. “A new approach to the orientation of random hypergraphs”. In: *SODA*. SIAM, 2012, pages 251–264. DOI: 10.1137/1.9781611973099.23.

- [LH06] S. Lefebvre and H. Hoppe. “Perfect spatial hashing”. In: *ACM Trans. Graph.* 25.3 (2006), pages 579–588. DOI: 10.1145/1141911.1141926.
- [Lia+19] X. Liao, M. Li, Y. Zou, F. Wu, Y. Pan, and J. Wang. “Current challenges and solutions of de novo assembly”. In: *Quant. Biol.* 7.2 (2019), pages 90–109. DOI: 10.1007/S40484-019-0166-9.
- [Lim+11] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. “SILT: a memory-efficient, high-performance key-value store”. In: *SOSP*. ACM, 2011, pages 1–13. DOI: 10.1145/2043556.2043558.
- [Lim+17] A. Limasset, G. Rizk, R. Chikhi, and P. Peterlongo. “Fast and Scalable Minimal Perfect Hashing for Massive Key Sets”. In: *SEA*. Volume 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 25:1–25:16. DOI: 10.4230/LIPICS.SEA.2017.25.
- [LK84] P. Larson and A. Kajla. “File Organization: Implementation of a Method Guaranteeing Retrieval in One Access”. In: *Commun. ACM* 27.7 (1984), pages 670–677. DOI: 10.1145/358105.358193.
- [LL86] W. Litwin and D. B. Lomet. “The Bounded Disorder Access Method”. In: *ICDE*. IEEE Computer Society, 1986, pages 38–48. DOI: 10.1109/ICDE.1986.7266204.
- [LPB06] Y. Lu, B. Prabhakar, and F. Bonomi. “Perfect Hashing for Network Applications”. In: *ISIT*. IEEE, 2006. DOI: 10.1109/ISIT.2006.261567.
- [LR85] P. Larson and M. V. Ramakrishna. “External Perfect Hashing”. In: *SIGMOD Conference*. ACM Press, 1985, pages 190–200. DOI: 10.1145/318898.318916.
- [LSW22] H.-P. Lehmann, P. Sanders, and S. Walzer. “SicHash – Small Irregular Cuckoo Tables for Perfect Hashing”. In: *CoRR* abs/2210.01560 (2022). DOI: 10.48550/ARXIV.2210.01560.
- [LSW23a] H.-P. Lehmann, P. Sanders, and S. Walzer. “ShockHash: Towards Optimal-Space Minimal Perfect Hashing Beyond Brute-Force”. In: *CoRR* abs/2308.09561 (2023). DOI: 10.48550/ARXIV.2308.09561.
- [LSW23b] H.-P. Lehmann, P. Sanders, and S. Walzer. “SicHash – Small Irregular Cuckoo Tables for Perfect Hashing”. In: *ALENEX*. SIAM, 2023, pages 176–189. DOI: 10.1137/1.9781611977561.CH15.
- [LSW24a] H.-P. Lehmann, P. Sanders, and S. Walzer. “ShockHash: Near Optimal-Space Minimal Perfect Hashing Beyond Brute-Force”. In: *CoRR* abs/2310.14959 (2024). DOI: 10.48550/ARXIV.2310.14959.
- [LSW24b] H.-P. Lehmann, P. Sanders, and S. Walzer. “ShockHash: Towards Optimal-Space Minimal Perfect Hashing Beyond Brute-Force”. In: *ALENEX*. SIAM, 2024, pages 194–206. DOI: 10.1137/1.9781611977929.15.
- [Lu+17] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. “WiscKey: Separating Keys from Values in SSD-Conscious Storage”. In: *ACM Trans. Storage* 13.1 (2017), 5:1–5:28. DOI: 10.1145/3033273.
- [Lub+01] M. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman. “Efficient erasure correcting codes”. In: *IEEE Trans. Inf. Theory* 47.2 (2001), pages 569–584. DOI: 10.1109/18.910575.
- [LV23] H.-P. Lehmann and G. Vinciguerra. *LeMonHash - GitHub*. <https://github.com/ByteHamster/LeMonHash>. 2023.
- [Mai83] H. G. Mairson. “The Program Complexity of Searching a Table”. In: *FOCS*. IEEE Computer Society, 1983, pages 40–47. DOI: 10.1109/SFCS.1983.76.
- [Mai92] H. G. Mairson. “The Effect of Table Expansion on the Program Complexity of Perfect Hash Functions”. In: *BIT* 32.3 (1992). DOI: 10.1007/BF02074879.

- [Maj+96] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech. “A Family of Perfect Hashing Methods”. In: *Comput. J.* 39.6 (1996), pages 547–554. DOI: 10.1093/COMJNL/39.6.547.
- [Már+15] L. Mármol, S. Sundararaman, N. Talagala, and R. Rangaswami. “NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store”. In: *USENIX ATC*. USENIX Association, 2015, pages 207–219.
- [Meh82] K. Mehlhorn. “On the Program Size of Perfect and Universal Hash Functions”. In: *FOCS*. IEEE Computer Society, 1982, pages 170–175. DOI: 10.1109/SFCS.1982.80.
- [Meh84] K. Mehlhorn. “Data Structures and Algorithms, Vol. 1: Sorting and Searching”. In: *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag (1984).
- [Mol05] M. Molloy. “Cores in random hypergraphs and Boolean formulas”. In: *Random Struct. Algorithms* 27.1 (2005), pages 124–135. DOI: 10.1002/RSA.20061.
- [MR01] J. I. Munro and V. Raman. “Succinct Representation of Balanced Parentheses and Static Trees”. In: *SIAM J. Comput.* 31.3 (2001), pages 762–776. DOI: 10.1137/S0097539799364092.
- [MV08] M. Mitzenmacher and S. P. Vadhan. “Why simple hash functions work: exploiting the entropy in a data stream”. In: *SODA*. SIAM, 2008, pages 746–755. DOI: 10.5555/1347082.1347164.
- [Mü+14] I. Müller, P. Sanders, R. Schulze, and W. Zhou. “Retrieval and Perfect Hashing Using Fingerprinting”. In: *SEA*. Volume 8504. Lecture Notes in Computer Science. Springer, 2014, pages 138–149. DOI: 10.1007/978-3-319-07959-2_12.
- [Nav14] G. Navarro. “Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences”. In: *ACM Comput. Surv.* 46.4 (2014), pages 1–47. DOI: 10.1145/2535933.
- [Nav16] G. Navarro. *Compact Data Structures – A Practical Approach*. Cambridge University Press, 2016.
- [New10] M. E. J. Newman. *Networks: An Introduction*. Oxford University Press, 2010. DOI: 10.1093/ACPROF:OSO/9780199206650.001.0001.
- [Nis+13] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. “Scaling Memcache at Facebook”. In: *NSDI*. USENIX Association, 2013, pages 385–398.
- [NM23] R. Nygaard and H. Meling. “SNIPS: Succinct Proof of Storage for Efficient Data Synchronization in Decentralized Storage Systems”. In: *CoRR* abs/2304.04891 (2023). DOI: 10.48550/ARXIV.2304.04891.
- [NR21] G. Navarro and J. Rojas-Ledesma. “Predecessor Search”. In: *ACM Comput. Surv.* 53.5 (2021), 105:1–105:35. DOI: 10.1145/3409371.
- [Nvi20] Nvidia. *Nvidia Ampere GA102 GPU Architecture*. <https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf>. 2020.
- [Nvi22] Nvidia. *CUDA C++ Programming Guide*. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. 2022.
- [OBS99] M. A. Olson, K. Bostic, and M. I. Seltzer. “Berkeley DB”. In: *USENIX ATC, FREENIX Track*. USENIX, 1999, pages 183–191.

- [O'N+96] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. "The Log-Structured Merge-Tree (LSM-Tree)". In: *Acta Informatica* 33.4 (1996), pages 351–385. DOI: 10.1007/s002360050048.
- [OS07] D. Okanohara and K. Sadakane. "Practical Entropy-Compressed Rank/Select Dictionary". In: *ALLENEX*. SIAM, 2007. DOI: 10.1137/1.9781611972870.6.
- [OV14] G. Ottaviano and R. Venturini. "Partitioned Elias-Fano indexes". In: *SIGIR*. ACM, 2014, pages 273–282. DOI: 10.1145/2600428.2609615.
- [Pag03] R. Pagh. "Basic External Memory Data Structures". In: *Algorithms for Memory Hierarchies*. Volume 2625. Lecture Notes in Computer Science. Springer, 2003, pages 14–35. DOI: 10.1007/3-540-36574-5_2.
- [Pib22] G. E. Pibiri. "Sparse and skew hashing of k-mers". In: *Bioinformatics* 38.Supplement_1 (2022), pages i185–i194. DOI: 10.1093/bioinformatics/btac245.
- [Pib23] G. E. Pibiri. "On weighted k-mer dictionaries". In: *Algorithms Mol. Biol.* 18.1 (2023), page 3. DOI: 10.1186/S13015-023-00226-2.
- [PP08] A. Pagh and R. Pagh. "Uniform Hashing in Constant Time and Optimal Space". In: *SIAM J. Comput.* 38.1 (2008), pages 85–96. DOI: 10.1137/060658400.
- [PPR07] A. Pagh, R. Pagh, and M. Ruzic. "Linear probing with constant independence". In: *STOC*. ACM, 2007, pages 318–327. DOI: 10.1145/1250790.1250839.
- [PR04] R. Pagh and F. F. Rodler. "Cuckoo hashing". In: *J. Algorithms* 51.2 (2004), pages 122–144. DOI: 10.1016/j.jalgor.2003.12.002.
- [PSL23] G. E. Pibiri, Y. Shibuya, and A. Limasset. "Locality-preserving minimal perfect hashing of k-mers". In: *Bioinform.* 39.Supplement-1 (2023), pages 534–543. DOI: 10.1093/BIOINFORMATICS/BTAD219.
- [PT21] G. E. Pibiri and R. Trani. "PTHash: Revisiting FCH Minimal Perfect Hashing". In: *SIGIR*. ACM, 2021, pages 1339–1348. DOI: 10.1145/3404835.3462849.
- [PT24] G. E. Pibiri and R. Trani. "Parallel and External-Memory Construction of Minimal Perfect Hash Functions With PTHash". In: *IEEE Trans. Knowl. Data Eng.* 36.3 (2024), pages 1249–1259. DOI: 10.1109/TKDE.2023.3303341.
- [Pua08] M. Puatracscu. "Succincter". In: *FOCS*. IEEE Computer Society, 2008, pages 305–313. DOI: 10.1109/FOCS.2008.83.
- [PV17a] G. E. Pibiri and R. Venturini. "Dynamic Elias-Fano Representation". In: *CPM*. Volume 78. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 30:1–30:14. DOI: 10.4230/LIPICS.CPM.2017.30.
- [PV17b] G. E. Pibiri and R. Venturini. "Efficient Data Structures for Massive N -Gram Datasets". In: *SIGIR*. ACM, 2017, pages 615–624. DOI: 10.1145/3077136.3080798.
- [RGR20] D. Reinsel, J. Gantz, and J. Rydning. *The Digitization of the World: From Edge to Core*. <https://www.seagate.com/files/www-content/our-story/trends/files/dataage-idc-report-final.pdf>. 2020.
- [Ric79] R. F. Rice. "Some practical universal noiseless coding techniques". In: *Jet Propulsion Laboratory, JPL Publication* (1979).
- [RKR24] J. Reichinger, T. Krismayer, and J. S. Rellermeyer. "COPR – Efficient, large-scale log storage and retrieval". In: *CoRR* abs/2402.18355 (2024). DOI: 10.48550/ARXIV.2402.18355.
- [RT89] M. V. Ramakrishna and W. R. Tout. "Dynamic External Hashing with Guaranteed Single Access Retrieval". In: *FODO*. Volume 367. Lecture Notes in Computer Science. Springer, 1989, pages 187–201. DOI: 10.1007/3-540-51295-0_127.

- [Sab+22] I. Sabek, K. Vaidya, D. Horn, A. Kipf, M. Mitzenmacher, and T. Kraska. “Can Learned Models Replace Hash Functions?” In: *Proc. VLDB Endow.* 16.3 (2022), pages 532–545. DOI: 10.14778/3570690.3570702.
- [San+19] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. DOI: 10.1007/978-3-030-25209-0.
- [Sch90] D. C. Schmidt. “GPERF: A Perfect Hash Function Generator”. In: *C++ Conference*. USENIX Association, 1990, pages 87–102.
- [Spr77] R. Sprugnoli. “Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets”. In: *Commun. ACM* 20.11 (1977), pages 841–850. DOI: 10.1145/359863.359887.
- [Sta11] R. P. Stanley. “Enumerative Combinatorics Volume 1 second edition”. In: *Cambridge studies in advanced mathematics* (2011). DOI: 10.1017/CB09781139058520.
- [Str+20] G. P. Strimel, A. Rastrow, G. Tiwari, A. Piérard, and J. Webb. “Rescore in a Flash: Compact, Cache Efficient Hashing Data Structures for n-Gram Language Models”. In: *INTER_SPEECH*. ISCA, 2020, pages 3386–3390. DOI: 10.21437/INTER_SPEECH.2020-1939.
- [Suz+07] B. E. Suzek, H. Huang, P. B. McGarvey, R. Mazumder, and C. H. Wu. “UniRef: comprehensive and non-redundant UniProt reference clusters”. In: *Bioinform.* 23.10 (2007), pages 1282–1288. DOI: 10.1093/BIOINFORMATICS/BTM098.
- [Szu06] M. Szudzik. “An elegant pairing function”. In: *Wolfram Research (ed.) Special NKS 2006 Wolfram Science Conference*. 2006, pages 1–12.
- [Tho15] M. Thorup. “High Speed Hashing for Integers and Strings”. In: *CoRR* abs/1504.06804 (2015). DOI: 10.48550/ARXIV.1504.06804.
- [Vig08] S. Vigna. “Broadword Implementation of Rank/Select Queries”. In: *WEA*. Volume 5038. Lecture Notes in Computer Science. Springer, 2008, pages 154–168. DOI: 10.1007/978-3-540-68552-4_12.
- [VS94] J. S. Vitter and E. A. M. Shriver. “Algorithms for Parallel Memory I: Two-Level Memories”. In: *Algorithmica* 12.2/3 (1994). DOI: 10.1007/BF01185207.
- [Wal21] S. Walzer. “Peeling Close to the Orientability Threshold - Spatial Coupling in Hashing-Based Data Structures”. In: *SODA*. SIAM, 2021, pages 2194–2211. DOI: 10.1137/1.9781611976465.131.
- [Wal22] S. Walzer. “Insertion Time of Random Walk Cuckoo Hashing below the Peeling Threshold”. In: *ESA*. Volume 244. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 87:1–87:11. DOI: 10.4230/LIPICs.ESA.2022.87.
- [Wal24] S. Walzer. “The Probability to Hit Every Bin with a Linear Number of Balls”. In: *CoRR* abs/2403.00736 (2024). DOI: 10.48550/ARXIV.2403.00736.
- [Wei73] P. Weiner. “Linear Pattern Matching Algorithms”. In: *SWAT*. IEEE Computer Society, 1973, pages 1–11. DOI: 10.1109/SWAT.1973.13.
- [WH20] S. A. Weaver and M. Heule. “Constructing Minimal Perfect Hash Functions Using SAT Technology”. In: *AAAI*. AAAI Press, 2020, pages 1668–1675. DOI: 10.1609/AAAI.V34I02.5529.
- [WMB99] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.
- [Zha+18] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and W. Chen. “Efficient Document Analytics on Compressed Data: Method, Challenges, Algorithms, Insights”. In: *Proc. VLDB Endow.* 11.11 (2018), pages 1522–1535. DOI: 10.14778/3236187.3236203.