

PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding

STEFAN HERMANN and HANS-PETER LEHMANN, Karlsruhe Institute of Technology, Karlsruhe, Germany

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

PETER SANDERS and STEFAN WALZER, Karlsruhe Institute of Technology, Karlsruhe, Germany

A minimal perfect hash function (or MPHf) maps a set of n keys to $[n] := \{1, \dots, n\}$ without collisions. Such functions find widespread application, e.g., in bioinformatics and databases. In this article, we revisit PTHash—a construction technique particularly designed for fast queries. PTHash distributes the input keys into small buckets and, for each bucket, it searches for a hash function seed that places its keys in the output domain without collisions. The collection of all seeds is then stored in a compressed way. Since the first buckets are easier to place, buckets are considered in non-increasing order of size. Additionally, PTHash heuristically produces an imbalanced distribution of bucket sizes by distributing 60% of the keys into 30% of the buckets.

Our main contribution is to characterize, up to lower-order terms, an *optimal* choice for the expected bucket sizes, improving construction throughput for space efficient configurations both in theory and practice. Further contributions include a new encoding scheme for seeds that works across partitions of the data structure and a GPU parallelization.

We call our technique PHOBIC—Perfect Hashing with Optimized Bucket sizes and Interleaved Coding. Compared to PTHash, PHOBIC is 0.17 bits/key more space-efficient for same query time and construction throughput. For a configuration with fast queries, our GPU implementation can construct an MPHf at 2.17 bits/key in 28 ns/key, which can be queried in 37 ns on the CPU.

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

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



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). The project also received funding from the European Union's Horizon Europe research and innovation programme (EFRA project, Grant Agreement No. 101093026). This work was also partially supported by DAIS—Ca' Foscari University of Venice within the IRIDE program. This work was also supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF).

Authors' Contact Information: Stefan Hermann (corresponding author), Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: hermann@kit.edu; Hans-Peter Lehmann, Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: hans-peter.lehmann@kit.edu; Giulio Ermano Pibiri, Ca' Foscari University of Venice, Venice, Italy; e-mail: giulioermannopibiri@unive.it; Peter Sanders, Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: sanders@kit.edu; Stefan Walzer, Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: stefan.walzer@kit.edu.



This work is licensed under a [Creative Commons Attribution 4.0 International](https://creativecommons.org/licenses/by/4.0/).

© 2026 Copyright held by the owner/author(s).

ACM 1549-6333/2026/10-ART38

<https://doi.org/10.1145/3849725>

ACM Reference Format:

Stefan Hermann, Hans-Peter Lehmann, Giulio Ermanno Pibiri, Peter Sanders, and Stefan Walzer. 2026. PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding. *ACM Trans. Algor.* 22, 4, Article 38 (October 2026), 31 pages.

<https://doi.org/10.1145/3849725>

1 Introduction

A *hash function* maps a set S of n keys to a range of integers $[m] := \{1, \dots, m\}$, regardless of whether multiple keys collide on the same output. A **Perfect Hash Function (PHF)** on S is a mapping *without collisions*. This requires $m \geq n$. The function does not necessarily have to store the keys explicitly. It only has to store enough information to prevent collisions, which are more likely when m is close to n . In the extreme case of $m = n$, the mapping is called a **Minimal PHF (MPHF)**. For simplicity, we only consider the minimal case in this article. However, our contributions are applicable to the non-minimal case as well. PHFs find widespread practical application, e.g., in compressed full-text indexes [3], computer networks [34], databases [9], prefix-search data structures [1], language models [43], bioinformatics [13, 39], and Bloom filters [8]. The three main performance parameters of an MPHF are low space consumption, fast construction, and fast queries. Concerning space, the lower bound is $\log_2 e \approx 1.44$ bits/key [36]. Practically viable approaches can achieve ≈ 1.45 bits/key [29]. This article is concerned with a technique that is focused on achieving fast queries. For example, this is very important when using perfect hashing to implement a static hash table that is both space-efficient and allows fast search.

Perfect Hashing Through Bucket Placement. To achieve fast query time, we ideally only have to retrieve a single value from memory and directly obtain the result of the query by combining this value and the key. Perfect hashing through bucket placement is an MPHF technique that can at least get close to this ideal. During construction, it takes the n keys and hashes them to small buckets. The algorithm *places* the buckets one after another by searching for a seed of a hash function that maps the keys to the output domain $[n]$. A seed is accepted if none of the keys in the bucket collide with another key of a previously placed bucket. The resulting list of unbounded seed values is stored with various compression techniques, resulting in tradeoffs between space consumption and query speed. We illustrate this in Figure 1. The first buckets are easier to place because the output domain is less full. It is therefore intuitively clear that the algorithm inserts the buckets in order of non-increasing size. A contribution of this article is to show that this bucket order is in fact optimal. A degree of freedom is that the algorithm can decide on the expected size of each bucket. While CHD [2] uses buckets of constant expected size, FCH [15] and PTHash [41, 42] set aside 30% of “heavy” buckets that receive 60% of the keys in expectation, while 70% of “light” buckets receive only 40% of the keys in expectation. The imbalance in expected bucket sizes improves construction speed by further decreasing the size of the last, hardest to place, buckets.

PHOBIC. This article aims at improving the space efficiency and construction speed of PTHash, while maintaining its fast query speed. PHOBIC first splits the input set into $P := \lceil n/\rho \rceil$ disjoint *partitions* of equal expected size ρ using a pseudo-random hash function. We construct an MPHF within each partition. The various MPHFs are then logically “concatenated” into a single MPHF taking the prefix sum of the partition sizes. Partitioning has also been applied on PTHash in PTHash-HEM [42]. Like in PTHash-HEM, we use perfect hashing through bucket placement to construct the MPHF of each partition. In contrast to PTHash-HEM, PHOBIC uses smaller partitions ($\rho \approx 2,500$) and each partition uses the same number $B := \lceil \rho/\lambda \rceil$ of buckets. To assign the keys to buckets, we first hash them uniformly into the range $h \in [0, 1]$. Each key is then assigned to bucket $\lceil B\gamma(h) \rceil$, where $\gamma : [0, 1] \rightarrow [0, 1]$ may be any function for now. We refer to γ as a *bucket*

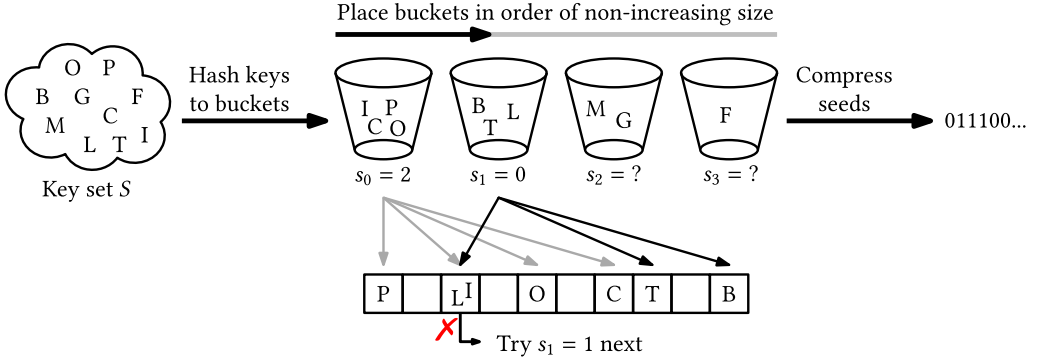


Fig. 1. Example for perfect hashing through bucket placement. The keys are assigned to buckets. The buckets are placed in order of non-increasing size. A bucket i is placed by finding a seed s_i of a hash function such that there are no collisions with previously placed keys. The s_i are stored in a compressed manner.

Algorithm 1: PHOBIC Construction and Query. Hash function h_X uniformly maps into the set X .

Function $\text{construct}(S, P, B, \gamma)$:

for all $k \in S$

 assign k to bucket $\lceil B \cdot \gamma(h_{[0,1]}(k)) \rceil$
 of partition $h_{[P]}(k)$

$\text{offsets} \leftarrow$ prefix sum of partition sizes

for all partitions $part$

$\text{size} \leftarrow$ number of keys in $part$

$\text{free} \leftarrow [\text{True}, \dots]$ of size size

 sort buckets in non-increasing size

for all bucket in $part$

$\text{keys} \leftarrow$ keys of bucket

$\text{seed} \leftarrow \text{placeBucket}(\text{keys}, \text{size}, \text{free})$

$\text{seeds}[part][\text{bucket}] \leftarrow \text{seed}$

return $\text{seeds}, \text{offsets}$

Function $\text{placeBucket}(\text{keys}, \text{size}, \text{free})$:

for $\text{seed} = 1, 2, 3, \dots$

$\text{pos} \leftarrow \{h_{[\text{size}]}(k, \text{seed}) : k \in \text{keys}\}$

if $|\text{pos}| = \text{size}$ and $\forall p \in \text{pos} : \text{free}[p]$

$\text{free}[p] \leftarrow \text{False}$ **for** $p \in \text{pos}$

return seed

Function $\text{query}(\text{key} \in S, P, B, \gamma, \text{seeds}, \text{offsets})$:

$\text{partition} \leftarrow h_{[P]}(\text{key})$

$\text{bucket} \leftarrow \lceil B \cdot \gamma(h_{[0,1]}(\text{key})) \rceil$

$\text{offset} \leftarrow \text{offsets}[\text{partition}]$

$\text{size} \leftarrow \text{offsets}[\text{partition} + 1] - \text{offset}$

$\text{seed} \leftarrow \text{seeds}[\text{partition}][\text{bucket}]$

return $\text{offset} + h_{[\text{size}]}(\text{key}, \text{seed})$

assignment function. We give pseudocode for construction and query in Algorithm 1. Our algorithm is implicitly parametrized by λ , ρ and γ .

Our Contributions. Our main contribution in this article is to characterize, up to lower-order terms, an optimal bucket assignment function γ , effectively taking the imbalance-trick used in FCH and PTHash to its logical conclusion. Our assignment function results in roughly equal work for placing each bucket. We prove that this results in minimal construction time and that any significantly different assignment function is more expensive. Our experiments confirm this finding, showing a speedup for space-efficient configurations. Our second contribution is to improve the compression of seed values when using partitioning. Seeds are searched independently for each partition, but compressed together. We exploit that the seeds of the i th bucket of each partition follow the same statistical distribution. This makes it possible to tune a compressor for each such index i . We store the seeds in an *interleaved* manner by consecutively placing the seeds for the i th buckets from all partitions. The construction of each of the many but small partitions requires only a tiny amount of memory. PHOBIC therefore naturally maps to the architecture of a **Graphics Processing Unit (GPU)**. Our final contribution is a GPU implementation to further speed up

construction compared to the CPU. Note that this journal version extends Ref. [23] through proofs on the optimality of bucket assignment functions, new general results on perfect hashing through bucket placement and more detailed experiments.

Subsequent Work. Several subsequent developments build upon PHOBIC [23]. Most notably, PtrHash [18] further improves PHOBIC's performance through several optimizations, including (i) using a fixed number of slots per partition, (ii) employing faster hash functions, and (iii) replacing our optimal bucket assignment function with a polynomial approximation that further improves query time. Subsequent work [22] also generalized our optimal bucket assignment function to support k -perfect hashing, where each output value is shared by up to k keys rather than a single key. Another PHF based on bucket placement is PHast [5]. See Section 2 for details about PtrHash and PHast. Regarding applications, PHOBIC has been used to index k -mers [19, 32].

Overview. In Section 2, we discuss related PHF constructions from the literature. We then discuss and prove general results on perfect hashing through bucket placement in Section 3. In Section 4, we present our main contribution—an optimal bucket assignment function. In Section 5, we give details on how partitioning can aid the construction, including a GPU implementation and more efficient encoding of seed values. Finally, we give a detailed experimental evaluation in Section 6 and conclude the article in Section 7.

2 Related Work

Perfect hashing is an active area of research. We provide a brief overview of state of the art approaches. For more detail, we refer to the recent survey on minimal perfect hashing [25].

Bucket Placement. The perfect hashing through bucket placement framework is described in Section 1. Earlier techniques that are based on bucket placement are CHD [2] and FCH [15]. Those techniques are superseded by PTHash [41].

Introduced after the submission of this article, PtrHash [18] and PHast [5] deviate from this framework by using fixed-width seeds to allow fast queries. In PtrHash, if no seed is found, earlier colliding buckets are evicted and a different seed is searched for those buckets similar to cuckoo hashing. PHast uses a different idea to handle situations where no seed is found called *bumping*. If none of the seeds 1, ..., 255 work, the seed is set to 0 to indicate that the keys of that bucket are handled in a next layer. Additionally, PHast does not map keys uniformly into the entire output domain. Instead, buckets map into overlapping slices of allowed values. Regarding bucket assignment functions, PHast spreads the keys uniformly into the buckets because the overlapping slices result in roughly equal success probabilities when a bucket is placed. Meanwhile, PtrHash uses a simple polynomial, intuitively approximating our optimal bucket assignment function while allowing better query performance.

Fingerprinting. Perfect hashing through fingerprinting [10, 30, 37] is a technique aimed at fast construction and queries at the cost of reduced space efficiency. The idea is to map the n keys to yn positions using a hash function, where y is a tuning parameter. A bit vector of length yn indicates positions that received exactly one key. Keys that are involved in collisions are handled recursively on another layer of the same data structure. A query operation descends through the recursive layers until it finds a 1-bit, meaning that the queried key was the only key mapping to that position. A rank operation on the bit vector for that position then gives the final MPH value. FMPH [4], FiPS [30], and BBHash [33] are publicly available implementations of this approach. FMPHGO [4] extends on this idea using a small number of brute-force re-tries to reduce the number of colliding keys. Fingerprinting based approaches are fast to construct but are outperformed in terms of space consumption and query time by PTHash.

Brute Force. RecSplit [14] finds a seed of a hash function that splits the keys into two sets of equal size. This is applied recursively resulting in a splitting tree. Practically viable implementations of this idea achieve as few as 1.45 bits/key but have fairly expensive queries [29]. To the best of our knowledge, RecSplit is the only other MPH construction technique that has a GPU implementation [6].

Perfect Hashing Through Retrieval. In perfect hashing through retrieval, every key has a number of candidate positions, determined by different hash functions. A retrieval data structure then stores which of the choices should be used for each key. Note that this implies some query overhead compared to PTHash. Early implementations include BPZ [7] and GOV [16]. More recent developments are SicHash [26], ShockHash [27, 28], and MorphisHash [21].

3 General Results on Perfect Hashing through Bucket Placement

In this section, we present general results on perfect hashing through bucket placement, for now without using any partitioning. First, we give a definition of a bucket assignment function. Then, in Section 3.2, we show that asymptotically, the space lower bound can be reached by any bucket assignment function. It is only useful for reducing the construction time. This paves the way for optimizing the bucket assignment function, which we do in Section 4. In the literature, it is assumed that in order to reduce construction time, the largest buckets should be placed first. This is intuitively clear and used in previous papers without a proof. In Section 3.3, we formally prove this intuition.

3.1 Bucket Assignment Functions

Let $w_1, \dots, w_B$ be the probability that a key hashes to bucket i for $i \in [B]$. We may assume without loss of generality that these probabilities are given in decreasing order. An equivalent view considers the prefix sums $\sigma_i := w_1 + \dots + w_i$. A key with (normalized) hash value $x \in (0, 1]$ is then assigned to bucket i if $x \in (\sigma_{i-1}, \sigma_i]$.

We can conveniently represent this information using a *bucket assignment function* $\gamma : [0, 1] \rightarrow [0, 1]$ that interpolates the points $\{(\sigma_i, i/B) \mid 0 \leq i \leq B\}$, is increasing and smooth on $(0, 1)$, and has non-decreasing derivative. The bucket assigned to hash value $x \in (0, 1]$ is then $\lceil \gamma(x) \cdot B \rceil$. It is a non-trivial insight of this article that a single bucket assignment function (not depending on B and n) can result in good construction times for many values of B and n simultaneously.

From now on, let $\lambda := n/B$ be the average bucket size. We summarize some useful intuitions about bucket assignment functions. These intuitions are valid for large n and B (when γ , γ^{-1} , and γ' are approximately constant on intervals of length $\frac{1}{n}$ and $\frac{1}{B}$). For now, we neglect edge cases related to γ or γ^{-1} not being smooth at 0 or not being smooth at 1 (but only smooth on $(0, 1)$).

INTUITION 3.1. Let $x \in (0, 1]$ be a normalized hash and $b = \gamma(x)$ the normalized bucket index of the bucket assigned to x . Then:

- (i) The expected size of the bucket assigned to x is $\lambda/\gamma'(x)$.
(Reason: In the vicinity of x and for infinitesimal δ , a δ -fraction of the hash range (used by δn keys in expectation) is shared by a $(\gamma'(x) \cdot \delta)$ -fraction of the B buckets. The quotient is $\delta n / (\gamma'(x) \delta B) = \lambda/\gamma'(x)$.)
- (ii) The expected size of the bucket with normalized index b is $\lambda/\gamma'(\gamma^{-1}(b)) = \lambda(\gamma^{-1})'(b)$.
(Follows from (i) and the inverse function rule of differential calculus.)
- (iii) The expected size of a bucket is decreasing in its normalized index.
(Follows from (ii) and monotonicity of γ' and γ^{-1} .)
- (iv) The expected fraction of keys with normalized hash in $(0, x]$ is x .

- (v) If $\mu > 0$ and $x_\mu \in (0, 1)$ is such that $\lambda/\gamma'(x_\mu) = \mu$ then the expected fraction of keys in buckets of expected size at least μ is x_μ . (Follows from (i), (iii) and (iv).)

3.2 Any Bucket Assignment Function Can Reach the Space Lower Bound

It is known that larger average bucket sizes λ result in lower space consumption at the cost of an increased construction time. This was previously shown in CHD [2] for the case in which all buckets have the same expected size. However, it was previously not analyzed how the distribution of expected sizes influences space consumption. It is a simple insight that large λ already guarantees a space consumption close to the lower bound of $\log_2 e$ bits per key, *without any assumptions* on the distribution of expected sizes. The following Propositions 3.2 and 3.3 are restated more formally and proven in Proposition 3.4

PROPOSITION 3.2. *Perfect hashing through bucket placement (as described in the Introduction) requires between $\log_2 e$ bits per key and $\log_2 e + \mathcal{O}\left(\frac{\log \lambda}{\lambda}\right)$ bits per key in expectation.*

Our goal therefore only needs to be to minimize construction time. Here, we are faced with a lower bound for our family of approaches.

PROPOSITION 3.3. *Perfect hashing through bucket placement requires $\Omega(ne^\lambda/\lambda)$ hash function evaluations in expectation during construction (on average $\Omega(e^\lambda)$ per bucket).*

It is intuitively clear (and proved in Section 3.3) that buckets should be processed in order from largest to smallest. The main remaining degree of freedom is to choose the expected sizes of the buckets. However, previous work has addressed this optimization problem only experimentally. Our main contribution (in Section 4) is to characterize, up to lower-order terms, an optimal distribution of expected sizes. We achieve an expected construction time of $e^{\lambda(1+\varepsilon)}$ per bucket. Our distribution of expected sizes is easy-to-implement and greatly improves construction time and space efficiency in practice when compared to the distribution of expected sizes of related work. We now restate Propositions 3.2 and 3.3 more formally as Proposition 3.4.

PROPOSITION 3.4. *Consider perfect hashing through bucket placement (as described in the Introduction) with n keys and $B := n/\lambda$ buckets for some $2 \leq \lambda = o(n/\log n)$. Let T be the sum of resulting seeds values and D the number of bits when encoding the seeds using Elias- δ -encoding. Then*

- (i) $\mathbb{E}[T] = \Omega(ne^\lambda/\lambda)$,
- (ii) $\mathbb{E}[D] = n(\log_2 e + \mathcal{O}\left(\frac{1}{\lambda} \log \lambda\right))$.

Note: (i) implies that the expected construction time per key of $e^{\lambda(1+\varepsilon)}$ for any $\varepsilon > 0$ that we achieve in Section 4 (when ignoring buckets of size 1, see that section for details) is almost optimal. (ii) means that the space consumption of perfect hashing through bucket placement approaches the lower bound $n \log_2 e$ for large λ , in the sense that it is $n(\log_2 e + o_{\lambda \rightarrow \infty}(1))$, *regardless* of the bucket assignment function that is used. The proof does not rely on the order in which the buckets are processed. However, note that the proof relies on the clean setup of bucket placement techniques described in the Introduction. It is incompatible with implementations like PtrHash [18] (uses evictions) and PHast [5] (non-uniform hashing to output domain).

PROOF. The first step of perfect hashing through bucket placement is to map the n keys into B buckets. The sizes $s_1, \dots, s_B$ of these buckets are random variables. However, this argument makes no use of their distributions and works for arbitrary bucket sizes. We focus on the second step where the buckets are placed one after the other using brute force. Assume the buckets are indexed

in placement order. The i th bucket is then associated with the probability p_i that randomly hashing a set of s_i keys into a table does not cause any collision assuming $s_1 + \dots + s_{i-1}$ out of n positions are already occupied.

A useful observation is that the product of all p_i is the probability that a random function of n keys to n positions is a bijection

$$\prod_{i=1}^B p_i = \prod_{i=1}^B \prod_{j=0}^{s_i-1} \frac{n - s_1 - \dots - s_{i-1} - j}{n} = \prod_{i=1}^B \frac{i}{n} = \frac{n!}{n^n} \text{ or equivalently } \prod_{i=1}^B 1/p_i = \frac{n^n}{n!}.$$

If the i th seed value is τ_i then its distribution is $\tau_i \sim \text{Geom}(p_i)$ implying $\mathbb{E}[\tau_i] = 1/p_i$. Hence,

$$\mathbb{E}[T] = \mathbb{E}\left[\sum_{i=1}^B \tau_i\right] = \sum_{i=1}^B \mathbb{E}[\tau_i] = \sum_{i=1}^B 1/p_i \geq B \cdot \left(\frac{n^n}{n!}\right)^{1/B}.$$

The last inequality uses that the sum of the side lengths of a B -dimensional cuboid with volume $V = \frac{n^n}{n!}$ is minimized by the B -dimensional cube where all B side lengths are equal to $V^{1/B}$. Continuing with Stirling's approximation yields

$$\mathbb{E}[T] \geq B \cdot \left(\frac{n^n}{n!}\right)^{1/B} = B \left(\frac{e^n}{\mathcal{O}(\sqrt{n})}\right)^{1/B} = \frac{n}{\lambda} e^\lambda n^{-\mathcal{O}(\lambda/n)} = \Omega(ne^\lambda/\lambda),$$

where the last step used $\lambda = o(n/\log n)$. This proves (i).

To encode a number $x \in \mathbb{N}$, Elias- δ -coding requires $\lfloor \log_2(x) \rfloor + 2 \lfloor \log_2(1 + \lfloor \log_2(x) \rfloor) \rfloor + 1 \leq \log_2(x) + 2 \log_2(1 + \log_2 x) + 1$ bits. Using that $\mathbb{E}[\log_2 X] \leq \log_2 \mathbb{E}[X]$ for any non-negative random variable X by Jensen's inequality allows us to bound

$$\begin{aligned} \mathbb{E}[D] &= \mathbb{E}\left[\sum_{i=1}^B (\log_2(\tau_i) + 2 \log_2(1 + \log_2 \tau_i) + 1)\right] \\ &\leq \sum_{i=1}^B \log_2 \mathbb{E}[\tau_i] + 2 \sum_{i=1}^B \log_2(1 + \log_2 \mathbb{E}[\tau_i]) + B \\ &= \sum_{i=1}^B \log_2 \frac{1}{p_i} + 2 \sum_{i=1}^B \log_2 \left(1 + \log_2 \frac{1}{p_i}\right) + B. \end{aligned}$$

This first sum can be bounded by $\sum_{i=1}^B \log_2(1/p_i) = \log_2 \frac{n^n}{n!} \leq n \log_2 e$ using Stirling's approximation. The only thing left to prove is that the second sum is subsumed by the error term $\mathcal{O}\left(n \frac{\log_2 \lambda}{\lambda}\right)$ in our stated bound. We again use Jensen's inequality and the fact that $\sum_{i=1}^B \log_2 1/p_i = \log_2 \frac{n^n}{n!} \leq n \log_2 e$:

$$\begin{aligned} \sum_{i=1}^B \log_2 \left(1 + \log_2 \frac{1}{p_i}\right) &\leq B \cdot \log_2 \left(1 + \frac{1}{B} \log_2 \frac{n^n}{n!}\right) \\ &\leq B \cdot \log_2 \left(1 + \frac{n}{B} \log_2 e\right) = \frac{n}{\lambda} \cdot \log_2 \left(1 + \lambda \log_2 e\right) = \mathcal{O}\left(n \frac{\log_2 \lambda}{\lambda}\right). \end{aligned}$$

The last step uses $\lambda \geq 2$. □

3.3 Why Buckets Should be Processed in Order of Decreasing Size

In previous papers, such as CHD [2] and PTHash [41], it is implicitly assumed that in order to reduce the number of hash function evaluations, the largest buckets should be placed first. Even though the claim is very intuitive, the argument is surprisingly subtle. For this, we present and prove Proposition 3.5.

PROPOSITION 3.5. *To minimize the expected number of hash function evaluations when constructing a PHF through bucket placement, buckets should be processed in order from largest to smallest.*

The challenge is to show that if a larger bucket is processed directly after a smaller bucket, then swapping the buckets reduces expected work. To prove this plausible but non-obvious fact, we consider a random process on a set $\{0, \dots, k\}$ of states for $k \in \mathbb{N}$ and transition probabilities $q_1, \dots, q_k \in (0, 1)$. When taking a step in state $0 \leq i < k$, the successor state is state $i + 1$ with probability q_{i+1} and state 0 with probability $1 - q_{i+1}$. We define $w(q_1, \dots, q_k)$ as the expected number of steps needed to reach state k from state 0.

Note how this captures bucket processing costs: The expected number of hash function evaluations when processing a bucket of size k when ℓ elements of the output range $[n]$ are already occupied is exactly $w\left(\frac{n-\ell}{n}, \frac{n-\ell-1}{n}, \dots, \frac{n-\ell-k+1}{n}\right)$. The following lemma generalizes the insight that when processing two buckets of combined size k , then the larger should be processed first.

LEMMA 3.6. *Assume $1 > q_1 > q_2 > \dots > q_k > 0$ and $1 \leq i < \frac{k}{2}$. Then*

$$w(q_1, \dots, q_{k-i}) + w(q_{k-i+1}, \dots, q_k) < w(q_1, \dots, q_i) + w(q_{i+1}, \dots, q_k).$$

With the abbreviations $E = (q_1, \dots, q_i)$, $M = (q_{i+1}, \dots, q_{k-i})$, $H = (q_{k-i+1}, \dots, q_k)$, we write the claim more compactly as $w(E \circ M) + w(H) < w(E) + w(M \circ H)$ where “ $\circ$ ” denotes concatenation.

For illustration, assume a juggler considers a sequence E of i easy throws, a sequence H of i hard throws, and a sequence M of $k - 2i$ throws of moderate difficulty. Performing a sequence means attempting all throws in sequence, starting over after the first unsuccessful throw, and repeating until all throws succeed. Then $w(R)$ is the expected number of throws when performing a sequence R . It is intuitively plausible that

$$w(E \circ M) + w(H) < w(E) + w(H \circ M) < w(E) + w(M \circ H).$$

The first inequality means that appending M to a difficult sequence rather than the simple sequence makes things more costly overall. The second inequality means that delaying difficult throws until the end of a sequence increases its cost because failures tend to happen *after* the easier throws have already taken place.

In the following, the Proof of Lemma 3.6 formalizes these insights. We make four observations about the function $w(q_1, \dots, q_k)$, each time justifying them briefly after stating them. The first serves as an alternative definition of $w(q_1, \dots, q_k)$:

$$w(q_1, \dots, q_k) = \sum_{i=1}^k \frac{1}{q_i \cdot \dots \cdot q_k}. \quad (1)$$

This holds because the number of visits to state i for $1 \leq i < k$ has a geometric distribution with parameter $q_i \cdot \dots \cdot q_k$ and hence has expectation $\frac{1}{q_i \cdot \dots \cdot q_k}$:

$$w(q_1, \dots, q_k) \text{ is monotonically decreasing in all of its } k \text{ parameters.} \quad (2)$$

This follows from Equation (1) because each summand is non-increasing in all parameters and the first summand is decreasing in all parameters:

$$\text{if } q_j < q_{j+1} \text{ for } 1 \leq j < k \text{ then } w(q_1, \dots, q_k) < w(q_1, \dots, q_{j-1}, q_{j+1}, q_j, q_{j+2}, \dots, q_k). \quad (3)$$

In other words, when swapping two adjacent parameters (at indices j and $j + 1$), then having the larger parameter further to the left yields a larger value of w . This follows from Equation (1) because the summand with $i = j + 1$ is then larger, and all other summands are the same:

$$\begin{aligned} w(q_1, \dots, q_k) &\text{ is maximized if the parameters appear in non-increasing order,} \\ &\text{and any other ordering of the same set of parameters yields a smaller value.} \end{aligned} \quad (4)$$

This follows by iterating Equation (3) until the parameters are sorted in non-increasing order:

$$\text{for any } 1 \leq j < k: w(q_1, \dots, q_k) = \frac{1}{q_{j+1} \dots q_k} w(q_1, \dots, q_j) + w(q_{j+1}, \dots, q_k). \quad (5)$$

This follows from Equation (1) by separating the sum into the summands for $i > j$ and those for $i \leq j$. We are now ready to proof Lemma 3.6.

PROOF OF LEMMA 3.6. Let $|A| := \prod_{q \in A} \frac{1}{q}$. Then

$$\begin{aligned} w(E \circ M) + w(H) &\stackrel{(5)}{=} |E \circ M| w(E) + w(M) + w(H) \\ &= w(E) + (|M| - 1)w(E) + w(M) + w(H) \\ &\stackrel{(2)}{<} w(E) + (|M| - 1)w(H) + w(M) + w(H) \\ &= w(E) + |M|w(H) + w(M) \\ &\stackrel{(5)}{=} w(E) + w(H \circ M) \\ &\stackrel{(4)}{<} w(E) + w(M \circ H). \end{aligned}$$

□

Now that we have proven Lemma 3.6, this directly implies Proposition 3.5.

PROOF OF PROPOSITION 3.5. Assume that the sequence of bucket sizes in processing order is $s_1, \dots, s_B$ and that $s_i < s_{i+1}$ for some $1 \leq i < B$. We will show that by switching the order of these two buckets, the expected number of hash function evaluations decreases. For this, let $\ell = \sum_{j=1}^{i-1} s_j$ be the number of keys that are handled before the i th bucket. The work for buckets i and $i + 1$ is then

$$w\left(\frac{n-\ell}{n}, \frac{n-\ell-1}{n}, \dots, \frac{n-\ell-s_i+1}{n}\right) + w\left(\frac{n-\ell-s_i}{n}, \frac{n-\ell-s_i-1}{n}, \dots, \frac{n-\ell-s_i-s_{i+1}+1}{n}\right).$$

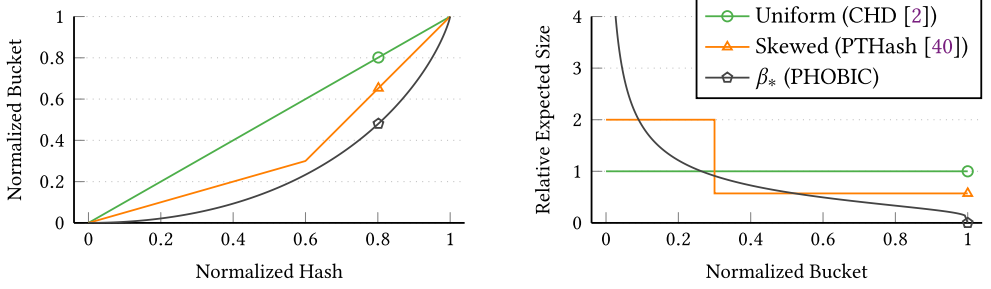
When processing the two buckets in swapped order, the expected work becomes

$$w\left(\frac{n-\ell}{n}, \frac{n-\ell-1}{n}, \dots, \frac{n-\ell-s_{i+1}+1}{n}\right) + w\left(\frac{n-\ell-s_{i+1}}{n}, \frac{n-\ell-s_{i+1}-1}{n}, \dots, \frac{n-\ell-s_i-s_{i+1}+1}{n}\right).$$

By Lemma 3.6, this is less. Note that the expected work needed for other buckets is unchanged because their sizes and the table load when they are processed are unchanged. □

4 Optimizing Bucket Sizes

Now that we know that the bucket assignment function does not influence the space consumption (except, possibly, in lower order terms), we can focus solely on minimizing the expected number of hash function evaluations as a proxy for expected construction time. In Section 4.1, we identify a bucket assignment function that is optimal in this regard and provide an intuitive explanation. This



(a) The bucket assignment functions map a normalized hash value x to a normalized bucket index $\gamma(x)$. (b) The expected bucket sizes relative to the average size λ are $(\gamma^{-1})'(b)$ for normalized bucket index b .

Fig. 2. Comparison of bucket assignment functions $\gamma(x)$ of related work and PHOBIC ($\gamma = \beta_*$).

function is the main contribution of this article. We prove its optimality in Sections 4.2 and 4.3 and discuss additional properties of this function in Sections 4.5 and 4.6. We remark that a subsequent work [22] generalizes the optimal bucket assignment function for k -perfect hashing, i.e., instead of mapping one key to one output value, k keys map to one output value.

4.1 An Optimal Bucket Assignment Function

Intuitively, we identify the following bucket assignment function to be optimal, although our precise result stated below is more subtle:

$$\beta_*(x) = x + (1 - x) \cdot \ln(1 - x) \text{ with derivative } \beta'_*(x) = -\ln(1 - x). \quad (6)$$

For comparison, Figure 2(a) shows β_* as well as the bucket assignment functions used by CHD and PTHash. In Figure 2(b), we see the distribution of expected bucket sizes, which is uniform for CHD, imbalanced for PTHash, and even more imbalanced for β_* .

Recall that $\lambda = n/B$ is the average bucket size. By Proposition 3.3, a lower bound for the expected work is $\Omega(n \cdot e^\lambda / \lambda)$. We prove, firstly, that any bucket assignment function γ that differs from β_* leads to expected work exceeding $n \cdot e^{(1+\varepsilon)\lambda}$ for some $\varepsilon = \varepsilon(\gamma)$, provided that $\lambda \geq \lambda_0(\varepsilon)$ is large enough. Conversely, we show that a slight perturbation β_ε of β_* leads to an expected work of essentially at most $n \cdot e^{(1+\varepsilon)\lambda}$ for any $\varepsilon > 0$, provided that $\lambda \geq \lambda_0(\varepsilon)$ is large enough. Essentially, we can get arbitrarily close to a cost of e^λ per key and only functions close to β_* can achieve this.

Our results bound the work $w_{n,\lambda}(\gamma)$ associated with γ and involve a “coupon collector term” w_{coupon} , which is the work required to place buckets of size 1.¹ We will define these more precisely below. Proofs are found in Sections 4.2 and 4.3. We have reason to believe that our results generalize for the non-minimal case of $m > n$, as explained in Section 4.5. Note that both theorems are phrased such that we may assume that n is much larger than λ and λ is much larger than $1/\varepsilon$. We give implementation details concerning the use of β_ε in Section 4.7. The following bounds hold **With High Probability (whp)**, i.e., with probability $1 - \mathcal{O}(n^{-c})$ for some $c > 0$.

¹The name is inspired by the famous coupon collector problem. Given an urn with n balls (or coupons), the collector repeatedly draws a uniformly random ball from the urn and puts it back. The question is how many draws are needed in expectation until each ball was drawn at least once. When placing buckets of size 1, we are similarly just waiting for unused hash values to appear when drawing them one by one.

THEOREM 4.1. *Let $\gamma : [0, 1] \rightarrow [0, 1]$ be a continuous bucket assignment function that is smooth on $(0, 1)$ with non-decreasing derivative. If $\gamma \neq \beta_*$ for β_* from Equation (6) then*

$$\exists \varepsilon > 0 : \forall \lambda \geq \lambda_0(\varepsilon) : \forall n \geq n_0(\lambda, \varepsilon) : w_{n,\lambda}(\gamma) \geq n \cdot e^{\lambda(1+\varepsilon)} + w_{\text{coupon}} \text{ whp.}$$

While this leaves the relationship between γ and $\varepsilon(\gamma)$ open, our proof suggests that any $\varepsilon < \sup_{x \in (0,1)} \frac{\beta'_*(x)}{\gamma'(x)} - 1$ is a possible choice.

THEOREM 4.2. *Let $\beta_\varepsilon(x) := \varepsilon x + (1 - \varepsilon)\beta_*(x)$ for some $\varepsilon > 0$ and β_* as in Equation (6). Then*

$$\forall \varepsilon > 0 : \forall \lambda \geq \lambda_0(\varepsilon) : \forall n \geq n_0(\lambda, \varepsilon) : w_{n,\lambda}(\beta_\varepsilon) \leq n \cdot e^{\lambda(1+\mathcal{O}(\varepsilon))} + w_{\text{coupon}} \text{ whp.}$$

What We Do Not Prove. Note that our analysis leaves undecided whether β_* is itself a “good” bucket assignment function, i.e., whether $w_{n,\lambda}(\beta_*)$ approaches e^λ in any meaningful sense. We suspect that it does and give empirical evidence in Section 4.4. However, the perturbation simplifies the analysis and improves running times in practice. Our analysis also does not imply that the particular perturbation we choose is the best choice: there may be an alternative to β_ε , in particular one that depends also on λ , such that the overall work approaches e^λ more quickly.

The Work Associated with a Bucket Assignment Function. To place a bucket of size $s \in \mathbb{N}$ into a hash table of size n that already has load factor $\alpha \in [0, 1 - \frac{s}{n}]$, we repeatedly try seeds for a hash function mapping keys to $[n]$, until all keys hash to free positions. The expected cost $c_n(s, \alpha)$ associated with this task under the simple uniform hashing assumption is described precisely in Section 3.3. We have to take into account self-collisions, i.e., while checking the keys one after the other, the load factor gradually increases and is $\alpha' = \alpha + \frac{s-1}{n}$ for the last key. For our purposes, the following bounds on $c_n(s, \alpha)$ suffice

$$(1 - \alpha)^{-s} \leq c_n(s, \alpha) \leq s \cdot (1 - \alpha')^{-s}. \quad (7)$$

This uses that $(1 - \alpha)^{-s}$ and $(1 - \alpha')^{-s}$ are lower and upper bounds on the number of seeds that have to be tried and that, to test a seed, at least 1 and at most s keys have to be considered. Now assume that we are given a bucket assignment function γ as well as $n \in \mathbb{N}$, $\lambda \in \mathbb{R}_+$, and $B = n/\lambda$. By assigning keys to buckets according to γ and hash values in $(0, 1]$, we obtain buckets. Let $s_1 \geq \dots \geq s_B$ be their sizes in decreasing order. By Proposition 3.5 in Section 3.3, it is advantageous to process the buckets in this order. Defining $\alpha_i := \frac{1}{n} \sum_{j=1}^{i-1} s_j$, the total cost is then $w_{n,\lambda}(\gamma) = \sum_{i=1}^B c_n(s_i, \alpha_i)$.

Note that while this describes the *expected* cost when given $(s_i)_{i \in [B]}$, overall $w_{n,\lambda}(\gamma)$ is still a random variable because the numbers $(s_i)_{i \in [B]}$ are random. Assume now that there are exactly k buckets of size 1 that are placed last (we may ignore buckets of size 0). For these buckets, the upper and lower bounds in Equation (7) coincide, so they incur a cost of

$$w_{\text{coupon}} := \sum_{i=1}^k c(1, \frac{n-i}{n}) = \sum_{i=1}^k \frac{n}{i} = n \cdot H_k.$$

Here, H_k is the k th harmonic number, which satisfies $H_k = \Theta(\log k)$. For fixed λ , the expectation of k is linear in n . Hence, if n is sufficiently large compared to λ , then we have $k \geq n^d$ whp for some constant $d > 0$, giving a cost of $\Theta(n \cdot H_{n^d}) = \Theta(n \log n)$. This dominates overall construction cost if n is sufficiently large compared to λ . Our theorems list this work for buckets of size 1 as a separate term because there are techniques to mitigate the problem: The hash function may permit to directly compute for a given key x and table position i , a seed for which x is mapped to i . This is the case if the seed includes an additive displacement term, as is the case in our implementation

and in FCH [15]. An alternative mitigation technique is to only construct the table up to a certain load (e.g., 99%) and use a different technique for filling the gaps using the remaining keys [41].

Intuition: What Makes β_* Uniquely Promising. For $\mu > 0$ let $x_\mu \in (0, 1)$ be such that $\lambda/\beta'_*(x_\mu) = \mu$. By Intuition 3.1 (v), roughly an expected x_μ -fraction of the keys (those with hashes in $[0, x_\mu]$) land in buckets of expected size at least μ . Assume for now that a bucket of expected size μ has actual size μ (ignoring the issue that μ may not be integer). Then, since we process buckets in order of non-increasing size, we would process a bucket of size μ when the load factor is x_μ . The expected cost for this is, by Equation (7), around $(1 - x_\mu)^{-\mu}$. Using that $\beta'_*(x) = -\ln(1 - x)$ gives $\mu = -\lambda/\ln(1 - x_\mu)$ and hence $(1 - x_\mu)^{-\mu} = (1 - x_\mu)^{\lambda/\ln(1 - x_\mu)} = e^\lambda$, i.e., a cost independent of μ . The idea behind Theorem 4.1 is that any bucket assignment function $\gamma \neq \beta_*$ fails to balance bucket sizes in this way, leading to significantly higher costs.

Complications the Proof Must Face. The simplification we made seems adequate for large μ since a bucket of large expected size typically has actual size close to its expectation. But consider $\mu = 1.5$ and cease to ignore rounding issues. If at load factor x_μ , we would process buckets of size 1 half the time and buckets of size 2 half the time, the resulting costs are around $e^{\frac{2}{3}\lambda}$ and $e^{\frac{4}{3}\lambda}$, respectively, which do not average out to e^λ . Luckily, things are more complicated. It turns out that for small $s \in \mathbb{N}$ and assuming large λ , the *expected number* of buckets of size s is meaningfully greater than the number of buckets of *expected size* in the range $[s - 1, s + 1]$.² This means that we get more small buckets than we seem to have called for, decreasing costs at high load factors. It seems clear that the flipside of this beneficial effect must be a detrimental effect for larger bucket sizes that a proof of Theorem 4.2 must quantify. When it comes to very large bucket sizes, we bail ourselves out by using β_ϵ instead of β_* : since β'_ϵ is lower bounded by ϵ , the expected bucket sizes are capped at λ/ϵ . It is buckets of intermediate sizes that have to pay the price.

4.2 Other Bucket Assignment Functions Are Worse

In this section, we give the proof for Theorem 4.1. We begin with a more formal statement of what we need from Intuition 3.1. Let $\mu_i := n \cdot w_i$ be the expected size of bucket i i.e., the expected number of keys assigned to bucket i) and again $\lambda := n/B$.

OBSERVATION 4.3. *Let $\gamma : [0, 1] \rightarrow [0, 1]$ be a continuous bucket assignment function that is smooth on $(0, 1)$ with non-decreasing derivative, $x_0 \in (0, 1)$ a hash and $i = \lceil B \cdot \gamma(x_0) \rceil$ the bucket assigned to x_0 . Let $\mu = \lambda/\gamma'(x_0)$. Then the expected bucket sizes satisfy*

- $\mu_j \geq \mu$ for all $j < i$ and
- $\mu_j \leq \mu$ for all $j > i$.

PROOF. Consider $j < i$ and the range $(\sigma_{j-1}, \sigma_j]$ assigned to bucket j . Since x_0 is assigned to bucket i we have $\sigma_j < x_0$. By monotonicity, $\gamma'(x) < \gamma'(x_0)$ for all $x \in [\sigma_{j-1}, \sigma_j]$. Since $\gamma(\sigma_j) - \gamma(\sigma_{j-1}) = 1/B$ by construction we have

$$\frac{1}{B} = \int_{\sigma_{j-1}}^{\sigma_j} \gamma'(x) dx \leq \int_{\sigma_{j-1}}^{\sigma_j} \gamma'(x_0) dx = (\sigma_j - \sigma_{j-1}) \cdot \gamma'(x_0) = w_j \cdot \frac{\lambda}{\mu} = \frac{\mu_j}{n} \frac{\lambda}{\mu} = \frac{\mu_j}{\mu} \frac{1}{B}.$$

Rearranging gives $\mu_j \geq \mu$. The second claim is obtained analogously. $\square$

²This is simply because there are so many buckets of larger expected size that might turn out to have size s . The effect is similar to the effect that the fraction of patients tested positive for a rare disease often vastly exceeds the fraction of patients with the disease, even if the false positive and false negative probability of the test coincides.

We assume the context of Theorem 4.1. The idea of the proof is to find a small region of $\mathcal{O}(\varepsilon n)$ keys where γ creates larger buckets than are optimal. Those few larger buckets alone dominate the total work in a way that they are enough to establish the lower bound.

In particular, γ is a monotonic function, smooth on $(0, 1)$ and satisfies w.l.o.g. $\gamma(0) = \beta_*(0) = 0$ and $\gamma(1) = \beta_*(1) = 1$, because it cannot be advantageous to not utilize all buckets. We have also assumed that $\gamma(x) > 0$ for $x > 0$, but never defended this assumption. Let us deal with this rather silly case now, i.e., assume $\gamma(x) = 0$ for some $x > 0$. Then by monotonicity $\gamma(x') = 0$ for all $x' \in [0, x]$. The first bucket then receives at least nx keys in expectation and $\Omega(nx)$ keys whp. A simple argument shows that if $n \geq 1/x^3$ then the expected number of seeds that need to be tried for the first bucket is $\exp(\Omega(n^{1/3}))$. This vastly exceeds the lower bound we wish to prove even for $\varepsilon = 1$.

It was therefore w.l.o.g. that we assumed $\gamma(x) > 0$ for all $x \in (0, 1]$. Since γ' is non-decreasing this implies $\gamma'(x) > 0$ for all $x \in (0, 1)$.

We now use the assumption that $\gamma \neq \beta_*$. Let $x \in (0, 1)$ with $\gamma(x) \neq \beta_*(x)$. If $\gamma(x) > \beta_*(x)$ then there is some $y \in [x, 1)$ with $\gamma'(y) < \beta'_*(y)$ and if $\gamma(x) < \beta_*(x)$ then there is some $y \in (0, x]$ with $\gamma'(y) < \beta'_*(y)$. In both cases, we have $1/\gamma'(y) > 1/\beta'_*(y)$. Because γ' and β'_* are smooth on $(0, 1)$ we can find $\varepsilon > 0$ and $z \in (0, 1 - 3\varepsilon)$ such that

$$1/\gamma'(z + 3\varepsilon) > (1 + 3\varepsilon)/\beta'_*(z).$$

Let i be the bucket that γ assigns to hash $z + 3\varepsilon$. Then all hashes in $[0, z + 3\varepsilon]$ are assigned to buckets with index at most i . Using a concentration bound and assuming large n , the number of keys assigned to buckets with index less than i is at least $n \cdot (z + 2\varepsilon)$. By Observation 4.3, all these buckets have expected size at least $\lambda/\gamma'(z + 3\varepsilon)$, hence expected size at least $\lambda(1 + 3\varepsilon)/\beta'_*(z)$. Most will have actual size at least $s := \lambda(1 + 2\varepsilon)/\beta'_*(z)$. Again, by a concentration bound, the number of keys in buckets of actual size at least s is at least $n \cdot (z + \varepsilon)$. In particular, at least εn keys are in buckets of size at least s that are processed when the load factor is already at least z . To get a lower bound on the work this causes, we may assume that these εn keys are in buckets of size exactly s and processed at load factor exactly z . By Equation (7), the expected cost of each such bucket is then lower bounded by

$$(1 - z)^{-s} = (1 - z)^{-\lambda(1+2\varepsilon)/\beta'_*(z)} = (1 - z)^{\lambda(1+2\varepsilon)/\ln(1-z)} = e^{\lambda(1+2\varepsilon)}.$$

The last step uses that $c^{1/\ln(c)} = e$ for all $c > 0$. Multiplying this by the number $\varepsilon n/s = \Theta(\varepsilon n/\lambda)$ of such buckets yields a contribution of $n \cdot \Theta(\varepsilon/\lambda) \cdot e^{\lambda(1+2\varepsilon)}$. This is at least $n \cdot e^{\lambda(1+\varepsilon)}$ if $\lambda \geq \lambda_0(\varepsilon)$ is large enough. Since the buckets of size 1 contribute a cost of w_{coupon} and are distinct from the buckets that contributed to $n \cdot e^{\lambda(1+\varepsilon)}$, we obtain the lower bound of $n \cdot e^{\lambda(1+\varepsilon)} + w_{\text{coupon}}$ as claimed.

4.3 A Perturbed Version of Our Bucket Assignment Function Is Good

In this section, we prove Theorem 4.2 and assume the corresponding context. In particular, we are given $\varepsilon > 0$, which defines a bucket assignment function β_ε , we assume $\lambda \geq \lambda_0(\varepsilon)$ is large enough and $n \geq n_0(\lambda, \varepsilon)$ is large enough, which defines $B = n/\lambda$. We begin with a few definitions and give corresponding intuition.

Definition 4.4. Let $i \in [B]$ and $s \in \mathbb{N}$. We define the following.

- $\lambda_i := n \cdot (\beta_\varepsilon^{-1}(i/B) - \beta_\varepsilon^{-1}((i-1)/B))$ is the expected size of the i th bucket. The stated formula involves the length of the range of hashes that β_ε maps to bucket i .
- s_i is the number of keys assigned to bucket i . This random variable has binomial distribution $s_i \sim \text{Bin}(n, \lambda_i/n)$.

- $\alpha_s := \frac{1}{n} \sum_{i=1}^B s_i \cdot \mathbb{1}_{s_i \geq s}$ is the random fraction of keys within buckets of size at least s and therefore the load factor after all buckets of size at least s have been processed.
- $d_s := 1 - e^{-\lambda/s}$ is the *deadline* for bucket size s . It is the load factor up to which Equation (7) guarantees that processing a bucket of size s incurs an expected cost of at most $s \cdot e^\lambda$.

The key lemma of this section proves a suitably weakened variant of the claim “ $\forall s : \alpha_s \leq d_s$,” i.e., that buckets of size s or larger are handled before their deadline d_s .

LEMMA 4.5 (DEADLINE LEMMA).

- (i) For each $2 \leq s \leq \lambda^{1/3} : \left(\frac{1-\alpha_s}{1-d_s} \right)^s \geq e^{-\mathcal{O}(\varepsilon\lambda)}$ whp.
- (ii) For each $\lambda^{1/3} < s \leq 2e^2\lambda/\varepsilon : \left(\frac{1-\alpha_s}{1-d_s} \right)^s \geq e^{-\mathcal{O}(\varepsilon\lambda)}$ whp.
- (iii) For each $2e^2\lambda/\varepsilon < s < 2\frac{\log_2 n}{\log_2 \log_2 n} : \alpha_s \leq d_s$ whp.
- (iv) For each $s \geq 2\frac{\log_2 n}{\log_2 \log_2 n} : \alpha_s = 0$ whp.

Each case is handled in a separate paragraph in Section 4.3.2. We now check that the Deadline Lemma implies Theorem 4.2.

PROOF OF THEOREM 4.2. We apply Lemma 4.5 for each $2 \leq s \leq 2\frac{\log_2 n}{\log_2 \log_2 n}$. Since each corresponding event holds whp, i.e., with probability $1 - \mathcal{O}(n^{-c})$ for some $c > 0$ and since there are $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$ events, they jointly hold whp by a union bound. We may assume that this is the case.

We have to bound the expected cost for handling the buckets by $n \cdot e^{\lambda(1+\mathcal{O}(\varepsilon))} + w_{\text{coupon}}$. Consider therefore any bucket b with actual size s . By Lemma 4.5, we know $s \leq 2\frac{\log_2 n}{\log_2 \log_2 n}$. If $s = 1$ then the work for b is accounted for by w_{coupon} . So assume $s \in \left\{2, \dots, 2\frac{\log_2 n}{\log_2 \log_2 n}\right\}$. After all buckets of size at least s are handled, the load factor is α_s . In particular, after b is handled, the load factor is at most α_s . From Equation (7) we get that the expected cost for handling b is at most $s \cdot (1 - \alpha_s)^{-s}$. If s falls into case (iii) of Lemma 4.5, then we can bound this as follows:

$$s(1 - \alpha_s)^{-s} \leq s(1 - d_s)^{-s} = s(e^{-\lambda/s})^{-s} = se^\lambda.$$

If s falls into case (i) or (ii), then we get

$$s(1 - \alpha_s)^{-s} = s(1 - d_s)^{-s} \left(\frac{1 - \alpha_s}{1 - d_s} \right)^{-s} = se^\lambda \cdot e^{\mathcal{O}(\varepsilon\lambda)} = se^{(1+\mathcal{O}(\varepsilon))\lambda}.$$

Summing this over all buckets of size at least 2, the sizes of which sum to at most n , we get an overall expected cost of at most $n \cdot e^{(1+\mathcal{O}(\varepsilon))\lambda}$, which is the main term of our bound. $\square$

4.3.1 General Observations for the Deadline Lemma. We begin with observations that relate to several of the cases (i), (ii), (iii), and (iv). To keep the presentation concise we will occasionally use $\approx$, $\geq$, and $\leq$ when a relation only holds whp and/or we suppress an error term that is clearly negligible in a given context when n is chosen large enough. This notation is not meant to indicate a gap in the proof.

We have previously encountered the random variable α_s , the fraction of keys in buckets of size at least s . This is not to be confused with the following non-random quantities.

Definition 4.6. Let $\mu > 0$ and $s \in \mathbb{N}$. We define

- $x_\mu := \frac{1}{n} \sum_{i=1}^B \lambda_i \cdot \mathbb{1}_{\lambda_i \geq \mu} \in [0, 1]$, the range of hashes assigned to buckets of *expected* size at least μ . Equivalently, the probability that a key is placed in a bucket of *expected* size at least μ .
- $\hat{x}_\mu := 1 - \exp\left(-\frac{\lambda/\mu - \varepsilon}{1 - \varepsilon}\right)$, the unique number satisfying $\lambda/\beta'_\varepsilon(\hat{x}_\mu) = \mu$ if $\mu \leq \lambda/\varepsilon$ and $\hat{x}_\mu = 0$ otherwise.³ Equivalently this is the length of the range $[0, \hat{x}_\mu]$ where $\lambda/\beta'_\varepsilon(x)$ attains values of at least μ .
- $\hat{\alpha}_s := \mathbb{E}[\alpha_s]$, the *expected* fraction of keys in buckets of size at least s .

OBSERVATION 4.7. $\max_{i \in [B]} \lambda_i \leq \lambda/\varepsilon$.

PROOF. This follows because $\beta'_\varepsilon(0) \geq \varepsilon$. See also Observation 4.3. $\square$

We now bound the difference between $\hat{x}_\mu$ and x_μ , which amounts to the number of keys in the bucket that $\hat{x}_\mu$ is assigned to.

OBSERVATION 4.8. For any $\mu > 0$ we have $|\hat{x}_\mu - x_\mu| \leq \frac{\lambda}{\varepsilon n}$, hence in most contexts $\hat{x}_\mu \approx x_\mu$.

PROOF. Let i_μ be the bucket assigned to hash $\hat{x}_\mu$. By Observation 4.3, the buckets of expected size at least μ are precisely the buckets preceding bucket i_μ and, possibly, bucket i_μ itself. Let μ' be the expected size of i_μ . If $\mu' \geq \mu$ then the entire range $[0, \hat{x}_\mu]$ of hashes is assigned to buckets of size at least μ , hence $x_\mu \geq \hat{x}_\mu$. If $\mu' \leq \mu$ then only a subset of the range $[0, \hat{x}_\mu]$ of hashes is assigned to buckets of size at least μ , hence $x_\mu \leq \hat{x}_\mu$. The difference between the two cases is μ'/n , the length of the range assigned to bucket i_μ . Since $\mu' \leq \lambda/\varepsilon$ by Observation 4.7, the claim follows. $\square$

The quantities α_s and $\hat{\alpha}_s$ are probabilistically related as follows.

LEMMA 4.9. For any $s \in \mathbb{N}$: $\Pr[|n \cdot \alpha_s - n \cdot \hat{\alpha}_s| \geq \delta] \leq 2 \exp\left(\frac{-2\delta^2}{ns^2}\right)$.

PROOF. We can directly apply the method of bounded differences [35]. The n hash values of keys are independent variables that determine $n \cdot \alpha_s$, the number of keys in buckets of size at least s . The expectation of $n \cdot \alpha_s$ is $n \cdot \hat{\alpha}_s$ by definition. Changing a single hash can affect $n \cdot \alpha_s$ by at most $\pm s$ with “+ s ” corresponding to moving a key from a bucket of size less than s to a bucket of size $s - 1$ and “− s ” to the reverse change. $\square$

By choosing $\delta = n^{2/3}$ we get

COROLLARY 4.10. For $n \geq n_0(s)$ large enough we have $|\hat{\alpha}_s - \alpha_s| \leq n^{-1/3}$ whp, hence in most contexts $\hat{\alpha}_s \approx \alpha_s$.

PROOF. Apply Lemma 4.9 with $\delta = n^{2/3}$ and assume that $n \geq s^{12}$. $\square$

In the following, we use $\text{Po}(\mu)$ to denote a Poisson distribution with expectation μ . Two further claims do not relate to our setting in particular.

CLAIM 4.11. There exists a constant $c_0 > 0$ such that for any $\mu \geq 3$ and $X \sim \text{Po}(\mu)$ we have $\Pr[X \leq \mu - 3] \geq c_0$.

³To see this, recall the definition $\beta_\varepsilon(x) = \varepsilon x + (1 - \varepsilon)\beta_*(x)$, which gives $\beta'_\varepsilon(x) = \varepsilon - (1 - \varepsilon)\ln(1 - x)$. Also recall Observation 4.3 for the relevance of $\lambda/\beta'_\varepsilon(x)$.

PROOF. The median $\text{med}(\mu)$ of $\text{Po}(\mu)$ satisfies $\text{med}(\mu) \in (\mu - 1, \mu + 1)$ [11]. For large μ , no single outcome has high probability. This implies for $X \sim \text{Po}(\mu)$:

$$\begin{aligned} \Pr[X \leq \mu - 3] &\geq \Pr[X \leq \text{med}(\mu) \wedge |\text{med}(\mu) - X| \geq 4] \\ &\geq \Pr[X \leq \text{med}(\mu)] - \Pr[|\text{med}(\mu) - X| \leq 4] = 1/2 - o_{\mu \rightarrow \infty}(1). \end{aligned}$$

In other words, $\Pr[X \leq \mu - 3]$ is bounded away from 0 for large μ . For small $\mu \geq 3$ clearly $\Pr[X \leq \mu - 3] > 0$. Hence, the desired $c_0 > 0$ exists. $\square$

CLAIM 4.12. Let $\varepsilon \in (0, 1)$, $\mu \geq \frac{1}{2\pi\varepsilon^2}$, $X \sim \text{Po}(\mu)$ and $s = \mu/(1 - \varepsilon)$. Then

$$\Pr[X \geq s] \leq (e^\varepsilon(1 - \varepsilon))^s.$$

Note that $e^\varepsilon(1 - \varepsilon) < 1$, which follows from $1 - x < e^{-x}$ for $x \in \mathbb{R} \setminus \{0\}$.

PROOF. Consider the sum $\Pr[X \geq s] = e^{-\mu} \sum_{i \geq s} \frac{\mu^i}{i!}$. The ratio between subsequent terms is $\mu/i \leq \mu/s = 1 - \varepsilon$. Hence, we can upper bound the sum by its first term and a geometric sum as follows:

$$\Pr[X \geq s] \leq e^{-\mu} \frac{\mu^s}{s!} \cdot \sum_{i \geq 0} (1 - \varepsilon)^i = e^{-\mu} \frac{\mu^s}{s!} \cdot \frac{1}{\varepsilon}.$$

Using Stirling's approximation for $s!$ and using $s \geq \mu \geq \frac{1}{2\pi\varepsilon^2}$ gives:

$$\Pr[X \geq s] \leq e^{-\mu} \frac{\mu^s e^s}{s^s \sqrt{2\pi s}} \cdot \frac{1}{\varepsilon} \leq e^{-\mu} \frac{\mu^s e^s}{s^s} = e^{-(1-\varepsilon)s} \frac{((1-\varepsilon)s)^s e^s}{s^s} = (e^\varepsilon(1 - \varepsilon))^s.$$

$\square$

4.3.2 *Proof of the Deadline Lemma.* We will now consider the cases of Lemma 4.5 one after the other.

(i) *Buckets of size* $2 \leq s \leq \lambda^{1/3}$. We have

$$1 - x_s \approx 1 - \hat{x}_s = \exp\left(-\frac{\lambda/s - \varepsilon}{1 - \varepsilon}\right) \quad \text{and} \quad 1 - x_{s+1} \approx 1 - \hat{x}_{s+1} = \exp\left(-\frac{\lambda/(s+1) - \varepsilon}{1 - \varepsilon}\right).$$

Using $s \leq \lambda^{1/3}$ we find $\lambda/s - \lambda/(s+1) = \frac{\lambda}{s(s+1)} = \Omega(\lambda^{1/3})$. This implies $\frac{\lambda/s - \varepsilon}{1 - \varepsilon} - \frac{\lambda/(s+1) - \varepsilon}{1 - \varepsilon} = \Omega(\lambda^{1/3})$ and for λ large enough we have $\exp\left(-\frac{\lambda/(s+1) - \varepsilon}{1 - \varepsilon}\right) \geq \left(1 + \frac{1}{c}\right) \exp\left(-\frac{\lambda/s - \varepsilon}{1 - \varepsilon}\right)$ for any constant $c > 0$. Hence,

$$\begin{aligned} x_s - x_{s+1} &\approx \hat{x}_s - \hat{x}_{s+1} = \exp\left(-\frac{\lambda/(s+1) - \varepsilon}{1 - \varepsilon}\right) - \exp\left(-\frac{\lambda/s - \varepsilon}{1 - \varepsilon}\right) \\ &\geq \frac{1}{c} \exp\left(-\frac{\lambda/s - \varepsilon}{1 - \varepsilon}\right) \geq \frac{1}{c} \exp\left(-\frac{\lambda/s}{1 - \varepsilon}\right). \end{aligned}$$

We now bound the probability $1 - \hat{\alpha}_s$ that a key k ends up in a bucket of size at most $s - 1$. Let $x \sim \mathcal{U}([0, 1])$ be its hash, $\mu \in (0, \lambda/\varepsilon)$ the *random variable* denoting the *expected* size of the bucket that k ends up in, and s' the number of other keys sharing the bucket with k . Conditioned on μ , we have $s' \sim \text{Bin}(n - 1, \mu/n)$, a distribution that is well known to be approximately $\text{Po}(\mu)$. In the following computation, we use that $\Pr[s' \leq s - 2 \mid \mu]$ is decreasing in μ , and we use that

$$\begin{aligned}
\Pr[\mu \in [s, s+1)] &= \Pr[x \in (x_{s+1}, x_s]] = x_s - x_{s+1}: \\
1 - \hat{\alpha}_s &= \Pr[s' \leq s-2] \geq \Pr[s' \leq s-2 \wedge \mu \in [s, s+1)] \\
&= \Pr[s' \leq s-2 \mid \mu \in [s, s+1)] \cdot \Pr[\mu \in [s, s+1)] \\
&\geq \Pr_{s'' \sim \text{Bin}(n-1, \frac{s+1}{n})}[s'' \leq s-2] \cdot (x_s - x_{s+1}) \\
&\approx \Pr_{s'' \sim \text{Po}(s+1)}[s'' \leq s-2] \cdot (x_s - x_{s+1}) \stackrel{\text{Claim 4.11}}{\geq c_0} (x_s - x_{s+1}) \stackrel{c=c_0}{\geq} \exp\left(-\frac{\lambda/s}{1-\varepsilon}\right).
\end{aligned}$$

This gives use the bound we desired, closing this case:

$$\left(\frac{1-\alpha_s}{1-d_s}\right)^s \approx \left(\frac{1-\hat{\alpha}_s}{1-d_s}\right)^s \geq \frac{\exp\left(-\frac{\lambda/s}{1-\varepsilon}\right)^s}{\exp(-\lambda/s)^s} = \frac{\exp\left(-\frac{\lambda}{1-\varepsilon}\right)}{\exp(-\lambda)} = e^{-\lambda/(1-\varepsilon)+\lambda} = e^{-\mathcal{O}(\varepsilon\lambda)}.$$

(ii) *Buckets of size* $\lambda^{1/3} \leq s \leq 2e^2\lambda/\varepsilon$. We remark that the assumption $s \leq 2e^2\lambda/\varepsilon$ is not required. However, the later cases designed for $s > 2e^2\lambda/\varepsilon$ prove a stronger conclusion.

We proceed in a similar way as in case (i) to bound the probability $\hat{\alpha}_s$ that a key k ends up in a bucket b of size at least s . Let again $\mu \in (0, \lambda/\varepsilon)$ be a random variable denoting the *expected* size of b and s' the number of other keys in b . We consider two overlapping events that cover all cases where b has size at least s . On the one hand, b might have large expected size. On the other hand, b might have small expected size and still have *actual* size at least s :

$$\begin{aligned}
\hat{\alpha}_s &= \Pr[s' \geq s-1] \leq \Pr[\mu \geq (1-\varepsilon)s \vee (\mu < (1-\varepsilon)s \wedge s' \geq s-1)] \\
&\leq \Pr[\mu \geq (1-\varepsilon)s] + \Pr[\mu < (1-\varepsilon)s] \cdot \Pr[s' \geq s-1 \mid \mu < (1-\varepsilon)s] \\
&\leq x_{(1-\varepsilon)s} + (1 - x_{(1-\varepsilon)s}) \cdot \Pr_{s'' \sim \text{Bin}(n-1, \frac{(1-\varepsilon)s}{n})}[s'' \geq s-1] \\
&\lesssim x_{(1-\varepsilon)s} + (1 - x_{(1-\varepsilon)s}) \cdot \Pr_{s'' \sim \text{Po}((1-\varepsilon)s)}[s'' \geq s-1] \\
&\leq x_{(1-\varepsilon)s} + (1 - x_{(1-\varepsilon)s}) \cdot 2((1-\varepsilon)e^\varepsilon)^s.
\end{aligned}$$

The last step uses Claim 4.12, conservatively accounting for a “-1” discrepancy with a factor of 2 and using that $\lambda \geq \lambda_0(\varepsilon)$ is large enough.

We now make another minor case distinction, first assuming that $(1-\varepsilon)s \leq \lambda/\varepsilon$. In that case $1 - x_{(1-\varepsilon)s} \approx 1 - \hat{x}_{(1-\varepsilon)s} = \exp\left(-\frac{\lambda/(s(1-\varepsilon))-\varepsilon}{1-\varepsilon}\right) \geq \exp\left(-\frac{\lambda/s}{(1-\varepsilon)^2}\right)$. We can now turn to $1 - \hat{\alpha}_s$ and quotient with $1 - d_s$. We assume ε is small enough such that $1/(1-\varepsilon)^2 \leq 1 + 3\varepsilon$ and that $\lambda \geq \lambda_0(\varepsilon)$ is large enough such that $2s((1-\varepsilon)e^\varepsilon)^s \leq \varepsilon$ for all $s \geq \lambda^{1/3}$:

$$\begin{aligned}
1 - \hat{\alpha}_s &\geq 1 - x_{(1-\varepsilon)s} - (1 - x_{(1-\varepsilon)s}) \cdot 2((1-\varepsilon)e^\varepsilon)^s \\
&= (1 - x_{(1-\varepsilon)s})(1 - 2((1-\varepsilon)e^\varepsilon)^s) \\
\Rightarrow \frac{1 - \hat{\alpha}_s}{1 - d_s} &\geq \frac{1 - x_{(1-\varepsilon)s}}{1 - d_s} (1 - 2((1-\varepsilon)e^\varepsilon)^s) \geq \frac{\exp\left(-\frac{\lambda/s}{(1-\varepsilon)^2}\right)}{\exp(-\lambda/s)} (1 - 2((1-\varepsilon)e^\varepsilon)^s) \\
&\geq \exp\left(-\frac{3\varepsilon\lambda}{s}\right) (1 - 2((1-\varepsilon)e^\varepsilon)^s) \\
\Rightarrow \left(\frac{1 - \hat{\alpha}_s}{1 - d_s}\right)^s &\geq \exp(-3\varepsilon\lambda) (1 - 2((1-\varepsilon)e^\varepsilon)^s)^s \\
&\geq \exp(-3\varepsilon\lambda) (1 - 2s((1-\varepsilon)e^\varepsilon)^s) \geq \exp(-3\varepsilon\lambda) \cdot (1 - \varepsilon) \geq \exp(-4\varepsilon\lambda).
\end{aligned}$$

The last step again assumes that λ is large enough.

We still have to consider the case where $(1 - \varepsilon)s > \lambda/\varepsilon$. By Observation 4.7, there are no buckets of expected size $(1 - \varepsilon)s$ or larger, hence $x_{(1-\varepsilon)s} = 0$. This gives $1 - \hat{\alpha}_s \geq 1 - 2((1 - \varepsilon)e^\varepsilon)^s$, meaning the above derivation only involves the less critical term that comes out as $1 - \varepsilon \geq \exp(-\varepsilon\lambda)$.

(iii) *Buckets of size $2e^2\lambda/\varepsilon < s \leq 2\frac{\log_2 n}{\log_2 \log_2 n}$.* Our use of β_ε rather than β_* renders this case quite easy. However, now that the bound on s grows with n , shortcuts involving “for n large enough” and the associated notation $\approx$ would now be suspect and have to be replaced with careful arguments.

We wish to bound $\hat{\alpha}_s$, the probability that at least $s - 1$ further keys join a given key in its bucket. Any bucket has expected size at most λ/ε by Observation 4.7. We can hence argue

$$\hat{\alpha}_s \leq \binom{n-1}{s-1} \left(\frac{\lambda/\varepsilon}{n}\right)^{s-1} \leq \left(\frac{ne}{s-1}\right)^{s-1} \left(\frac{\lambda/\varepsilon}{n}\right)^{s-1} = \left(\frac{e\lambda/\varepsilon}{s-1}\right)^{s-1} \leq e^{-s+1},$$

where the last step used $s > 2e^2\lambda/\varepsilon \geq e^2\lambda/\varepsilon + 1$. Applying Lemma 4.9 with $\delta = n \cdot e^{-s}$ gives $\Pr[n \cdot \alpha_s - n \cdot \hat{\alpha}_s \geq n \cdot e^{-s}] \leq 2\exp(-2ne^{-2s}/s^2)$. For $s \leq 2\log_2 n / \log_2 \log_2 n$ this probability is negligible. In particular, we have whp

$$\alpha_s = (\alpha_s - \hat{\alpha}_s) + \hat{\alpha}_s \stackrel{\text{whp}}{\leq} e^{-s} + \hat{\alpha}_s \leq e^{-s} + e^{-s+1} \leq e^{-s+2}.$$

In particular, α_s is *much* smaller than the deadline d_s , we can argue for instance by using that $1 - e^{-x} \geq x/2$ for $x \in [0, \frac{1}{2}]$ to get:

$$d_s = 1 - e^{-\lambda/s} \geq \lambda/(2s) \geq 1/s \geq e^{-s+2} \stackrel{\text{whp}}{\geq} \alpha_s.$$

(iv) *Buckets of Size at Least $s = 2\frac{\log_2 n}{\log_2 \log_2 n}$.* Again, the largest expected bucket size is at most λ/ε by Observation 4.3 and the probability that a key hashes to a specific bucket hence at most $\frac{\lambda/\varepsilon}{n}$. The probability $p_{\geq s}$ that the bucket has size at least s is therefore bounded by

$$p_{\geq s} \leq \binom{n}{s} \left(\frac{\lambda/\varepsilon}{n}\right)^s \leq \left(\frac{ne}{s}\right)^s \cdot \left(\frac{\lambda/\varepsilon}{n}\right)^s = \left(\frac{e\lambda/\varepsilon}{s}\right)^s = 2^{s(\log_2(e\lambda/\varepsilon) - \log_2 s)}.$$

Plugging in $s = 2\frac{\log_2 n}{\log_2 \log_2 n}$ gives $p_{\geq s} = n^{-2+o(1)}$. By a union bound, the probability that at least one bucket has size at least s is at most $B \cdot p_{\geq s} \leq n^{-1+o(1)}$. Hence, whp no such bucket exists and we have $\alpha_s = 0$ as claimed. $\square$

4.4 Evaluation of Bucket Assignment Functions for Small λ

Our theoretical analysis shows that a bucket assignment function can only be optimal if it converges to β_* for large λ . For small λ , Figure 3 compares the cost of the uniform, skewed, and β_* bucket assignment functions empirically. It shows that β_* is even slightly outperformed by the skewed bucket assignment function for $\lambda \leq 7$ and even by the uniform bucket assignment function for $\lambda \leq 3$. However, across all measurements β_* results in at most 24% more hash function evaluations compared to the uniform and skewed bucket assignment functions. Figure 3 demonstrates how β_* outperforms the uniform and skewed functions for larger λ . Figure 3 also shows the lower bound established in Proposition 3.4. The number of hash function evaluations when using β_* is at most a factor of 4.6 away from the lower bound. Reasons for this discrepancy are (i) Proposition 3.4 lower bounds the number of hash function evaluations by the number of tested seeds, but seeds do not always fail on the first key of the bucket; (ii) the lower bound assumes that the bucket sizes can be non-natural numbers; (iii) the lower bound assumes that the bucket sizes can be controlled exactly, but the bucket sizes result from a random process; and (iv) β_* is not optimal for small λ .

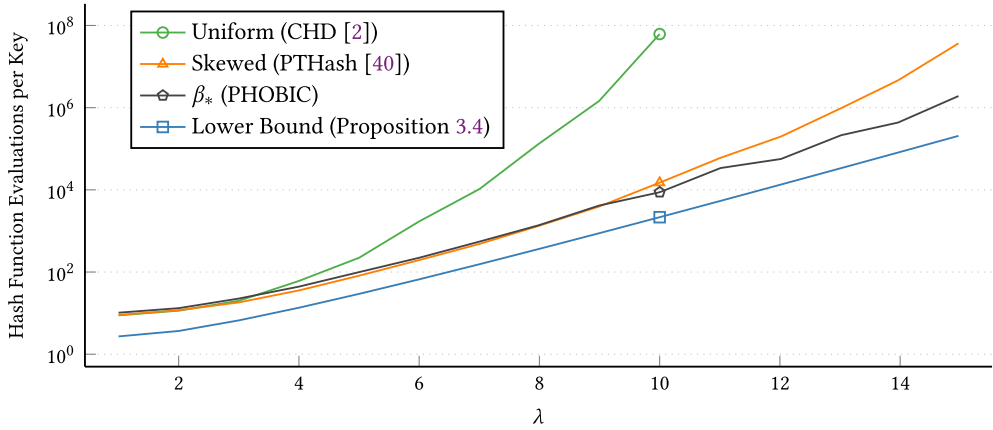


Fig. 3. Empirical comparison of different bucket assignment functions in terms of the number of hash function evaluations per key during construction for $\lambda \in [15]$ and $n = 2,500$.

4.5 Bucket Assignment for Non-Minimal PHF

Assume we wish to construct a *non-minimal* PHF using the “perfect hashing through bucket placement” framework. Hence, assume the range of the hash function is $[m]$ and $n = \alpha m$ for $\alpha \in (0, 1)$.

In the previous sections, we only discussed the case $\alpha = 1$. There is, however, a natural way to obtain from a bucket assignment function γ for $\alpha = 1$ a bucket assignment function γ_α for the non-minimal case. Namely, we can define $\gamma_\alpha(x) = \frac{1}{\gamma(\alpha)}\gamma(\alpha x)$, which rescales whatever γ does on its subdomain $[0, \alpha]$. The idea is that, in order to fill a table to load factor 1, you first need to fill it to load factor α , and if γ is optimal overall, then it should implicitly contain an optimal strategy for this first phase. Since γ defines on $[0, \alpha]$ how an α -fraction of the keys is assigned to buckets of largest expected size (presumably processed first), this is where we should look. This argument remains heuristic because the buckets of largest *expected* size may not turn out to be the buckets of largest *actual* size.

4.6 Cache Oblivious Optimality

A concept that was not considered in previous works is theoretical cache efficiency. If an MPHf does not fit into cache, then some part of it resides exclusively in slow memory. A query incurs additional overhead if it has to access information that is not cached. In the following, we outline an upper bound for the fraction of keys that can be queried without a cache miss. We also show that perfect hashing through bucket placement reaches this upper bound. We assume an explicitly managed cache. It is helpful to first analyze the situation in which an MPHf technique is aware, i.e., not oblivious, of the cache size M . To this end, we introduce the following MPHf, which is split in two parts, one resides in cache, the other one not.

Definition 4.13. Given a set $S \subset U$ with $|S| = n$ and a cache of size M bits, a *cache aware MPHf* is a pair $(f_{\text{cache}} : U \rightarrow [n] \cup \{\perp\}, f_{\text{memory}} : U \rightarrow [n])$ such that f_{cache} requires at most M bits and

$$f(x) = \begin{cases} f_{\text{memory}}(x), & \text{if } f_{\text{cache}}(x) = \perp \\ f_{\text{cache}}(x), & \text{otherwise} \end{cases}$$

is an MPHf on S .

Note that in particular, $f_{\text{memory}}(x)$ may return anything for x if $f_{\text{cache}}(x) \neq \perp$ or $x \notin S$. Now, consider the keys that can be answered using the cache only. By definition those are $S' := \{x \in S \mid f_{\text{cache}}(x) \neq \perp\}$. We call the ratio $\alpha := \frac{|S'|}{|S|}$ the load factor of f_{cache} . In the following, we assume sufficiently large U and omit corresponding lower-order terms. Let $l_n(\alpha) := n((1 - \alpha) \log_2(1 - \alpha) + \alpha \log_2 e)$ [2] be the space lower bound for a PHF with range n and αn keys. Even though f_{cache} is a PHF on S' with range n it is conceivable that the encoding of f_{cache} undercuts $l_n(\alpha)$ because the set S' is not fixed in advance. However, in the following we show that this additional freedom of choosing the subset cannot result in space savings.

LEMMA 4.14. *The space lower bound for f_{cache} is $l_n(\alpha)$.*

PROOF. The idea is to show that we can construct a f_{memory} using $l_n(1) - l_n(\alpha)$ bits such that $(f_{\text{cache}}, f_{\text{memory}})$ is a cache aware MPHF. A cache aware MPHF requires at least $l_n(1)$ bits. Hence, f_{cache} would contradict this lower bound if it used less than $l_n(\alpha)$ bits. To construct f_{memory} , we use a simple brute-force approach. We search for a single seed s of a hash function such that all keys from $S \setminus S'$ are mapped to distinct positions in $[n] \setminus f_{\text{cache}}(S')$. The probability that a seed results in an MPHF is $\frac{k!}{n^k}$ where $k = n - |S'|$ is the number of free positions. The space required to store the seed (e.g., using Golomb encoding) is $\log_2 \left(\frac{n^k}{k!} \right) \approx k \log_2 \left(\frac{ne}{k} \right)$ up to lower-order terms. With $k = n(1 - \alpha)$ we have $k \log_2 \left(\frac{ne}{k} \right) = n(1 - \alpha) \log_2 \left(\frac{e}{1 - \alpha} \right) = n(1 - \alpha) \log_2 e - n(1 - \alpha) \log_2(1 - \alpha) = l_n(1) - l_n(\alpha)$. $\square$

Hence, f_{cache} cannot have a higher load α than what $M = l_n(\alpha)$ allows. We call an MPHF technique *cache oblivious optimal* if, for any α , it answers queries for an α fraction of keys using only a cache of size $M = l_n(\alpha)$. As an example for $\alpha = 0.5$, cache oblivious optimality means that 50% of queries are answered using just $\frac{l_n(0.5)}{l_n(1)} \approx 15\%$ of the MPHF's data. For large λ , perfect hashing through bucket placement is arbitrarily close to cache oblivious optimality, if combined with a sufficiently space efficient seed encoding (e.g., Elias- δ). This is achieved by maintaining the seeds of the largest buckets in cache. Analogously to Proposition 3.4 (ii), the space for the seeds of the largest buckets containing an α fraction of keys is $l_n(\alpha) + \mathcal{O}(\alpha n^{\frac{1}{\lambda}} \log \lambda)$.

4.7 Details on the Bucket Assignment in Practice

Our implementation of bucket assignment functions differs from our theoretical results in two minor ways.

Perturbation. Recall that $\beta_* : [0, 1] \rightarrow [0, 1]$ was, modulo certain qualifications, identified as the optimal bucket assignment function in Section 4.1. One of the qualifications was that we actually analyze a slightly perturbed function $\beta_\varepsilon(x) := \varepsilon x + (1 - \varepsilon)\beta_*(x)$. This limits expected bucket sizes to λ/ε , which helped with bounding construction times of large buckets. Limiting the bucket sizes is also useful in practice to reduce self-collisions inside the small partitions. According to the birthday paradox, the probability of a self-collision is at most constant for a bucket of size $\mathcal{O}(\sqrt{\rho})$, for an expected partition size of ρ . Concretely, we choose $\varepsilon = \frac{\lambda}{5\sqrt{\rho}}$. Thus, capping the expected bucket sizes at $5\sqrt{\rho}$ (see Observation 4.7). Without this perturbation ($\varepsilon = 0$), running times are noticeably worse.

Tabulating Values. The functions β_ε involve expensive arithmetic operations such as a logarithm. We achieve a significant speedup by tabulating β_ε for 2,048 discrete values of x and interpolating linearly. Note that as an alternative PtrHash [18] approximates the optimal bucket assignment function using a polynomial. This results in even better query time. PtrHash can afford this, because

it relies on relatively small λ for construction anyway and the choice of the bucket assignment function is not as critical in that regime (see Section 4.4).

5 Fine-Grained Partitioning

In this section, we engineer the PHOBIC algorithm as presented in Algorithm 1. In particular, we show a novel technique to efficiently store seed values and present a GPU parallelization of PHOBIC. Our compression and parallelization both rely on a fine-grained partitioning. Any MPHf construction can trivially be parallelized by hashing the keys into subsets of equal expected size and building a MPHf for each subset in parallel. The various PHFs are then logically “concatenated” into a single MPHf taking the prefix sum of the partition sizes. The respective offsets have to be looked up when querying a key, imposing some query time overhead. Partitioning is widely used for a variety of construction techniques. It was also used by PTHash in the PTHash-HEM variant [42]. In this article, we use partitions of size $\approx 2,500$, i.e., several orders of magnitude smaller than the ones used in PTHash-HEM. In itself, reducing the partition size results in only marginal construction time improvements. However, small partitions enable a new, more efficient encoding scheme which we introduce in Section 5.1. Additionally, they enable a fast GPU parallelization which we describe in Section 5.2.

Encoding the Offsets of the Partitions. The offsets and sizes of the many partitions require a non-negligible amount of space. We mitigate this without incurring too much query overhead by storing sizes only implicitly as the difference of two subsequent offsets. Offsets are stored as the difference to their expectation, which roughly halves the required space because the differences are at most $\approx \sqrt{n}$. The offsets make up for about 0.007 bits per key of the total space consumption.

Hash Function. The hash function used in PTHash works by XOR-ing the hash value of the key with the hash value of the seed. Although the technique works well on large-enough partition sizes, it might fail for small sizes because of correlations in the hash values. Given the small partition sizes we use here, we have to rely on a different technique. We use the seed value p to store two numbers, namely $p = s \cdot m + d$, where m is the actual partition size and $d \in [m]$ is an additive displacement. The position of a key x is $(h(x, s) + d) \bmod m$. While searching for a seed, we calculate $h(x, s)$ for all keys of the bucket and then only have to increment the values to obtain the positions for the next seeds. If no position is found within $d \in [m]$ we continue by incrementing s , re-calculating $h(x, s)$ and setting $d = 0$. At query time, we have to calculate the position of a key x using seed p . Note that we have $d = p \bmod m$, so we can compute the position of the key x as $(h(x, \lfloor p/m \rfloor) + p) \bmod m$.

5.1 Interleaved Coding of Seeds

Once the search has finished, the seeds found for each bucket have to be stored in some compressed manner. Ideally, the seeds should be encoded such that they require little space and are quickly accessible during querying. We mainly use *Compact* and *Golomb-Rice* encoding as building blocks for our new technique. Compact encoding is also used in the original PTHash implementation. In Compact encoding, all values are stored consecutively by concatenating their binary representation. All values use the same bit length, allowing for quick access. The bit length is chosen such that the highest seed can be accommodated. Golomb-Rice [17, 44] encoding stores the b least significant bits of each seed using compact encoding. The most significant bits are stored in unary representation. A data structure [38] supporting the select operation, i.e., giving the index of the i th 1 in a bit vector, enables access to the unary part of the seeds in constant time. We apply the formula by Kiely [24] to choose b .

A straightforward approach would be to encode all seeds using a single encoder. However, it is beneficial to group seeds which follow the same distribution and encode them using the same encoder. PTHash does this only partially by using two encoders—one for the heavy buckets and

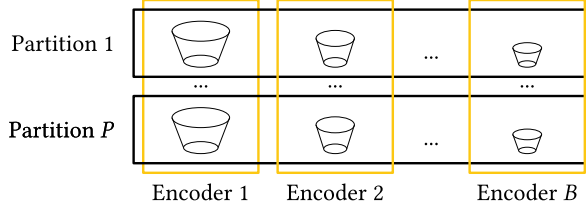


Fig. 4. Interleaved coding. Encoder i stores the seed of bucket i from all partitions.

one for the light buckets (the so-called “front-back” compression [41]). Although the idea of our optimized bucket assignment function is to give all buckets the same seed distribution, this is not fully achievable in practice. One reason is that the actual size of each bucket is an integer, while our bucket assignment function uses continuous probabilities. This results in discrete jumps in the probability that a seed is found when processing one bucket after another. Therefore, it is still useful to encode groups of different seeds with different encoders.

We now introduce our new technique. For each partition we hash to the same number of buckets $B := \lceil \rho / \lambda \rceil$, based on the average partition size ρ and average bucket size λ . The i th bucket of a partition has the same expected size and the corresponding seed follows the same statistical distribution as the i th bucket of any other partition. In *interleaved coding* we employ B encoders and the i th encoder stores the seeds of the i th buckets of all partitions. Each encoder is tuned for its specific seed distribution (e.g., different Rice parameters). Figure 4 illustrates interleaved coding.

It is also possible to mix different encoding techniques, similar to what PTHash does. Larger buckets are accessed more often than smaller buckets because they contain more keys. Hence, it is beneficial to use an encoding technique which is optimized for fast lookup time (e.g., Compact) for the larger buckets. Conversely, the encoding for the seeds belonging to smaller buckets should be tuned for space efficiency (e.g., Golomb-Rice).

We remark that each of the B encoders introduces some metadata overhead (e.g., for storing its parameters). Using rather small partition sizes ρ decreases the number B of encoders and therefore the constant overhead. To conclude this section, we point out that by using interleaved coding we maintain near cache oblivious optimality (Section 4.6) despite partitioning, by caching the encoders which correspond to the larger buckets.

5.2 GPU Parallelization

We provide a GPU implementation for even faster construction. On a GPU, each *workgroup* executes independently, typically with its own subset of data. Within each workgroup, individual *threads* execute concurrently. Threads within the same workgroup can share data and synchronize with each other through mechanisms like barriers and shared memory. Thread level parallelism allows for fine-grained parallel execution of instructions within a workgroup. However, only threads which follow the same control path and thus execute the same instruction at the same time can be executed in parallel. It is therefore crucial to avoid divergent control paths. As a first step, our parallel implementation transfers the keys to the GPU, before partitioning them. Afterwards, we sort the buckets and start the search.

Search. Our fine-grained partitioning naturally maps to the architecture of a GPU. Each partition is processed by one workgroup. The small partition sizes enable performing the search entirely using fast but small shared memory. Bucket by bucket, all threads of the workgroup cooperate to quickly find the smallest working seed. A CPU implementation would usually do so using a nested loop. The outer loop would iterate over seed values and the inner loop over keys. If a collision

Algorithm 2: Seed Search for One Bucket

```

shared sFound  $\leftarrow \infty$ 
shared sNext  $\leftarrow$  threadCount
seed  $\leftarrow$  threadId
keyIndex  $\leftarrow$  0
while sFound =  $\infty$  do
  isCollision  $\leftarrow$  COLL(seed, keyIndex)
  keyIndex  $\leftarrow$  keyIndex + 1
  if isCollision then
    keyIndex  $\leftarrow$  0
    seed  $\leftarrow$  ATOMADD(sNext, 1)
  else if keyIndex = bucketSize then
    sFound  $\leftarrow$  ATOMMIN(sFound, seed)

```

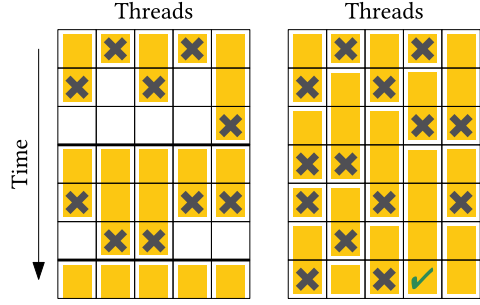


Fig. 5. Seed search for a single bucket. Each yellow box represents one seed. Cells are one key tested for one seed. Collisions are indicated by $\times$. Left: Synchronized nested loop. Right: Algorithm 2, where we continue with the first key and the next seed after a collision.

occurs, it would immediately leave the inner loop and continue with the first key and the next seed. However, on a GPU, leaving the inner loop would result in divergence because threads might encounter a collision after a different number of keys. This is illustrated in the left of Figure 5.

Instead, we use the technique described in [45] to emulate the behavior of the nested loop using a single loop to reduce divergence. Hence, our GPU implementation parallelizes over partitions, seeds, and keys. The inner loop is emulated by incrementing the key index in each iteration. If a collision occurs, we reset the key index and emulate the behavior of the outer loop by atomically incrementing a seed counter which is shared among all threads. If the last key did not collide, we found a working seed. Multiple threads can find a seed in the same iteration. To reduce entropy, we use an atomic minimum to identify the smallest of those seeds. Note that this finds the smallest working seed overall because all threads finding a working seed must have processed each key. Therefore, if there was a smaller working seed, it would have been found in an earlier loop iteration. We give pseudocode in Algorithm 2 and illustrate the behavior on the right in Figure 5. During search, we only access shared memory and perform fast arithmetic operations. Specific optimizations for our additive displacement hash function are not shown here.

6 Experiments

In Section 6.1, we gradually integrate our improvements to show the individual effects. Then, we compare our GPU and CPU implementation with the state of the art in Section 6.2. If not explicitly said otherwise, we use a machine with an Intel Core i7-11700 CPU with 64 GiB of DDR4 RAM running Ubuntu 22.04.1. Each core has 48 KiB L1 and 512 KiB L2 data cache. As a GPU, we use an Nvidia RTX 3090 and use Vulkan 1.3.236 to interface with it. We compile using GCC 11.4.0 and compiler options `-march=native` and `-O3`. All benchmarks use random strings of random length between 10 and 50 characters as input, which is adopted from previous work [6, 26–28]. Note that almost all competitors first generate master hash codes of the input. This makes the construction largely independent of the input distribution. We measure the query time by querying each key once in random order. All experiments use $n = 100$ million keys, $\lambda = 8$, an average partition size of $\rho = 2,500$ and a single thread if not stated otherwise. Our source code is public under the General Public License [20, 31, 40].

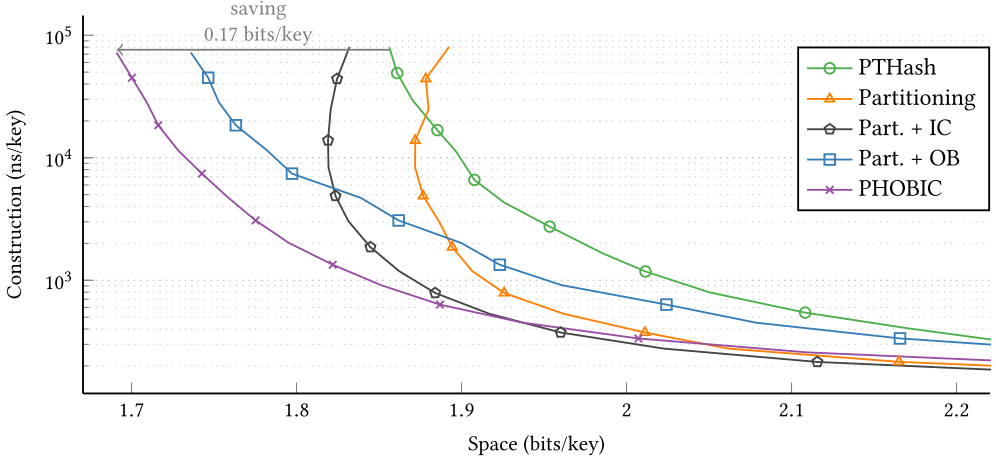


Fig. 6. Ablation study of PHOBIC showing the construction time and space tradeoffs when varying λ . In the following mono-R refers to the encoding where all seeds are stored in a single Rice encoder and IC-R refers to interleaved Rice coding. Beginning with *PTHash* (Elias-Fano, skewed bucketing), we add fine-grained *partitioning* (mono-R, skewed) with additive displacement hashing. We then add interleaved coding (*IC-R*, skewed) and optimized bucketing (mono-R, *OB*) individually. Putting it all together, we arrive at *PHOBIC*.

6.1 From PTHash to PHOBIC

We now gradually introduce our improvements to PTHash. As basic improvements, we replace the initial hash function with xxHash [12] and implement faster parallel partitioning. In all experiments, PTHash contains these changes as well to focus on our algorithmic improvements. Figure 6 gives measurements for the different improvement steps and shows them in different combinations.

Interleaved Coding. Partitioning of PTHash is already used in PTHash-HEM for parallelization [42]. PTHash-HEM uses partitions of size $\approx 10^6$. Smaller partitions only lead to minor construction time improvements due to cache friendliness. However, smaller partition sizes shine when used with our newly introduced interleaved coding (Section 5.1). Interleaved coding uses ρ/λ encoders, where ρ is the expected partition size. Using a smaller partition size, results in less encoders and can therefore significantly reduce constant space overheads, as we show in Figure 8. However, for small partition sizes, the space for storing the partition offsets becomes dominant. Figure 6 compares the technique of placing all seeds into a single Rice encoder (orange curve) to placing the keys using interleaved Rice coding (black). Interleaved coding consistently improves space efficiency by 0.06 bits/key. Figure 7 compares different combinations of encoders, which were partially used in the original implementation. Interleaved coding allows for mixing of different encoding techniques. If we use this to encode the keys using different numbers of Compact and Rice encoders, we can cover the entire query time to space tradeoff in our configuration. Note that the construction performance is similar for all approaches because the encoding is fast compared to the remaining construction.

Optimized Bucket Assignment. In Figure 6, we also show how using the optimized bucket function affects construction speed and space. Our optimization of the bucket assignment function is particularly helpful to construct very space efficient configurations. For the same construction time, the optimized function is up to 0.14 bits/key more space efficient compared to the original PTHash bucket assignment.

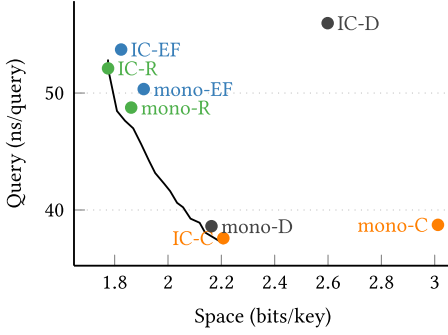


Fig. 7. Query time and space consumption of Elias-Fano, Dictionary, Compact, and Rice encoders. Points prefixed “mono” place all seeds into a single encoder and those prefixed “IC” use interleaved coding. The curve shows different mixtures (see Section 5) of Compact and Rice encoders. All points use optimized bucketing.

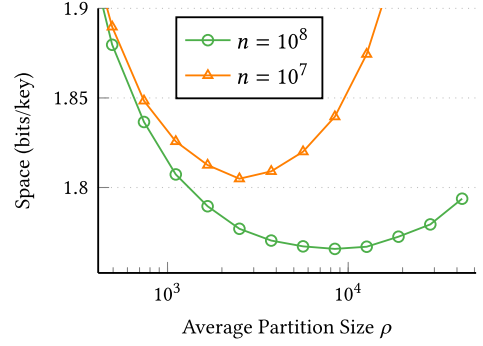
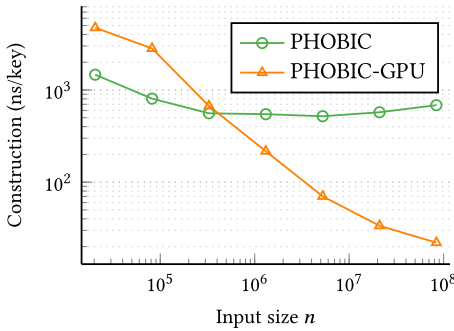
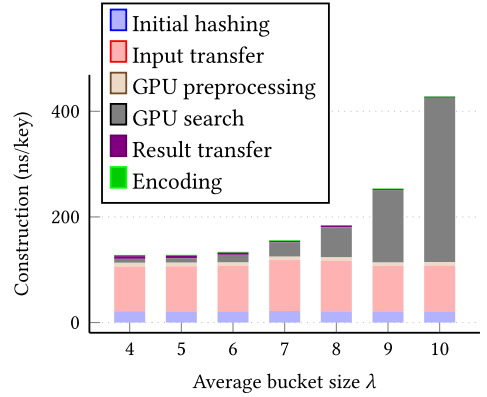


Fig. 8. Space consumption for different n depending on the average partition size ρ using interleaved Rice coding. Smaller partition sizes increase the overhead caused by storing the offset of each partition. Larger partition sizes increase the number of encoders. Each encoder has a constant overhead. The overhead therefore diminishes for larger n .



(a) Construction performance on CPU and GPU for $\lambda = 8.0$ and 8 threads, comparing different n . The GPU requires large n to fully utilize its computing resources.



(b) Different construction steps by values of average bucket size λ .

Fig. 9. GPU performance for different input sizes (left) and λ (right).

Further Remarks. With interleaved coding, an additional improvement stems from the secondary bucket ordering. While buckets are primarily sorted by non-increasing *actual* size, a secondary ordering by increasing *expected* size reduces the space consumption by 0.044 bits per key compared to ordering by decreasing expected size ($\lambda = 8$). The reason for this behavior remains an open question. However, some intuition can be gained by considering the bucket with the smallest expected size in each partition. In interleaved coding, the seeds of these buckets are compressed together. Suppose that the expected size of such a bucket is close to zero. In that case, the bucket is empty most of the time, and an empty bucket is encoded using seed 0. On the rare occasions when the bucket is non-empty, it is desirable for the probability of successfully placing the bucket to be as high as possible. Otherwise, a low success probability would disproportionately increase the space consumption, even for buckets whose expected size is close to zero. This effect is achieved by

placing such buckets first among all buckets of the same size, thereby maximizing their placement success probability.

Original PTHash achieves improvements in space usage and construction time at the expense of query time by first constructing a non-minimal PHF and subsequently repairing the resulting “gaps.” For details, see [41]. This technique does not yield a comparable improvement in PHOBIC. Gains in space and construction time can be achieved with a smaller degradation in query performance by appropriately tuning λ and adjusting the encoding.

GPU Parallelization. Our final contribution is a GPU implementation to speed up construction. Our implementation parallelizes over partitions, seeds, and keys. The GPU implementation is mainly useful for large average bucket sizes λ . This is well illustrated in Figure 9(b): For smaller values of λ , the construction time is dominated by the time to transfer the input data to the GPU. We also compare CPU and GPU construction speed for different input sizes in Figure 9(a). The GPU requires a large number of input keys and thus a large number of partitions before its computing resources are fully utilized. Overall, the GPU outperforms the CPU for a sufficiently large λ and n . We use the GPU only to accelerate construction, while measuring all queries on the CPU.

6.2 Comparison to Other Methods

We compare our new approach to several other methods from the literature. First and foremost, we compare against the original PTHash [41, 42] implementation. The comparison also includes the fingerprinting approaches FiPS [30], BBHash [33], FMPH [4], and FMPHGO [4]. We also compare against RecSplit [14] and approaches based on it, such as SIMDRecSplit [6], ShockHash-RS [27], and bipartite ShockHash-RS [28]. Finally, we also compare against CHD [2] and SicHash [26]. Note that PtrHash [18] and PHast [5] are not included because they were published after the submission of this article.

Each method has a wide range of configurations that provide a tradeoff between space, construction time, and query time. To give an initial overview, we show a Pareto front for each method in Figure 10. A configuration is on the Pareto front if no other configuration of the same method is simultaneously faster and more space efficient. For this plot, we use a single thread (a multithreaded measurement would mainly show what method implemented the partitioning step most efficiently instead of focusing on the algorithmic aspects). The figure shows that PTHash and PHOBIC are clear winners in terms of query performance. Even though BBHash [33] and FMPH [4] are also focused on fast queries, they are significantly slower than PTHash and PHOBIC. Figure 6 shows that PHOBIC consistently saves about 0.17 bits/key for a large range of different construction times while maintaining the good query speed. We remark that this is a significant reduction in space considering the proximity to the space lower bound. The competitors achieving even lower space consumption i.e., RecSplit [14], SIMDRecSplit [6], ShockHash-RS [27], and bipartite ShockHash-RS [28]) all have a rather slow query performance. However, somewhat surprisingly, SIMDRecSplit has the fastest construction even for less space efficient configurations. SicHash [26] takes a middle ground with faster construction than PHOBIC and query performance between PHOBIC and the RecSplit variants.

Table 1 gives a selection of configuration parameters for direct comparison, mostly taken from the corresponding papers. Comparing configurations with the same space consumption, PHOBIC is significantly faster to construct than the original PTHash implementation. Comparing configurations that both need 1.86 bits/key and have a similar query time, PHOBIC can be constructed 83 times faster than PTHash.

Table 2 illustrates the results of the same experiment but using a machine with more cores. This machine is equipped with an AMD EPYC Rome 7702P processor with 64 cores (with hyper-threading) and a clock frequency of 2.0 GHz. While this machine does not have a GPU, it shows

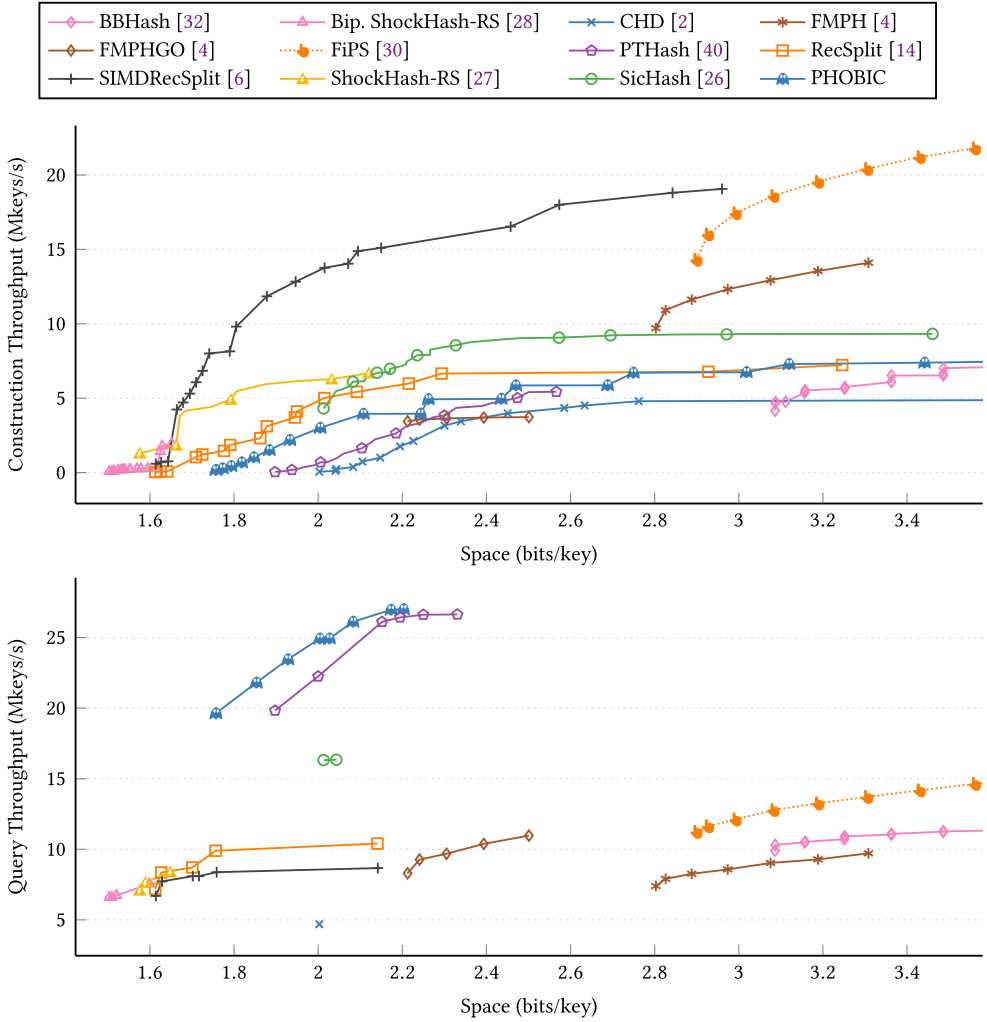


Fig. 10. Construction throughput (top) and query throughput (bottom) for various methods on 100 million keys and using a single processing thread.

how well PHOBIC scales with a large number of threads. We also note that the query times are higher compared to those measured with the machine used for Table 1 because of the different single-thread performance and memory latency.

6.3 GPU Comparison

On the GPU, we compare against the only available GPU construction, RecSplit-GPU [6]. Figure 11 shows the construction throughput of PHOBIC and RecSplit on the GPU, by varying the number of bits/key. Basically, we achieve the same peak construction throughput as RecSplit-GPU for the less space-efficient configurations. The queries of both approaches are done on the CPU, so the fact that PHOBIC offers much faster queries applies here as well (see Figure 10). Comparing the multithreaded CPU implementation and the GPU implementation of PHOBIC, we get a construction speedup of 62 for $\lambda = 9$ with interleaved Rice coding. Note that with $\lambda = 9$, the GPU still spends a

Table 1. Performance of Various Methods on 100 Million Keys

Method	Space (bits/key)	Query (ns/query)	Construction (ns/key)		
			1 Thread	8 Threads	Speedup
Bip. SH-RS, $n = 64, b = 2,000$	1.52	160	5,756	1,218	4.7
CHD, $\lambda = 3$	2.27	222	352	-	-
CHD, $\lambda = 5$	2.07	207	2,206	-	-
FMPH, $\gamma = 2.0$	3.40	100	69	17	4.0
FMPH, $\gamma = 1.0$	2.80	134	99	24	4.0
SIMDRecSplit, $n = 8, b = 100$	1.81	124	109	20	5.2
SIMDRecSplit, $n = 14, b = 2,000$	1.58	143	11,062	2,360	4.7
SicHash, $\alpha = 0.9, p_1 = 21, p_2 = 78$	2.41	72	129	25	5.0
SicHash, $\alpha = 0.97, p_1 = 45, p_2 = 31$	2.08	64	179	32	5.6
PTHash, $\lambda = 4.0, \alpha = 0.99, \text{C-C}$	3.19	35	314	143	2.2
PTHash, $\lambda = 5.0, \alpha = 0.99, \text{EF}$	2.11	54	525	252	2.1
PTHash, $\lambda = 10.5, \alpha = 0.99, \text{EF}$	1.86	49	82,721	35,048	2.4
PTHash-HEM, $\lambda = 4.0, \alpha = 0.99, \text{C-C}$	3.19	39	299	45	6.6
PTHash-HEM, $\lambda = 5.0, \alpha = 0.99, \text{EF}$	2.11	58	582	86	6.7
PHOBIC, $\lambda = 3.9, \alpha = 1.0, \text{IC-C}$	3.18	40	197	32	6.2
PHOBIC, $\lambda = 4.5, \alpha = 1.0, \text{IC-R}$	2.11	57	254	40	6.2
PHOBIC, $\lambda = 6.5, \alpha = 1.0, \text{IC-R}$	1.85	52	992	176	5.6
PHOBIC, $\lambda = 9.0, \alpha = 1.0, \text{IC-R}$	1.74	50	9,171	1,781	5.1
GPU + 8 CPU Threads					
PHOBIC-GPU, $\lambda = 9.0, \text{IC-C}$	2.17	37		28	
PHOBIC-GPU, $\lambda = 9.0, \text{IC-R}$	1.76	52		27	
PHOBIC-GPU, $\lambda = 13.0, \text{IC-R}$	1.68	50		560	
PHOBIC-GPU, $\lambda = 14.0, \text{IC-R}$	1.67	49		1,470	
RecSplit-GPU, $\ell = 8, b = 100$	1.81	126		24	
RecSplit-GPU, $\ell = 14, b = 2,000$	1.58	147		80	
RecSplit-GPU, $\ell = 18, b = 2,000$	1.55	135		1,732	

lot of its construction time on transferring the input data (see Figure 9(b)), but much larger values of λ are not feasible on the CPU.

Directly comparing the performance of CPU and GPU is always difficult because of the different hardware architectures. Given that the power consumption is a major cost factor in production environments, we measure it using a Voltcraft Energy Check 3000 wattmeter. For CPU-only and idle measurements, we dismount the GPU. The machine requires about 405 W during the search step of the GPU version and 195 W for the multithreaded CPU implementation. Thus, the above speedup of 62 translates to 30 times lower energy consumption for constructing an MPHf on the GPU. Even when first subtracting the machine's idle power consumption of 47 W, the GPU construction still has 26 times lower energy consumption. Single-threaded CPU construction requires 74 W, which is less energy efficient compared to multithreading.

Table 2. Performance of Various Methods on 100 Million Keys on a Machine with 64 Cores

Method	Space (bits/key)	Query (ns/query)	Construction (ns/key)		
			1 Thread	64 Threads	Speedup
Bip. SH-RS, $n = 64, b = 2,000$	1.52	425	9,365	356	26.3
CHD, $\lambda = 3$	2.27	357	756	-	-
CHD, $\lambda = 5$	2.07	320	5,483	-	-
FMPH, $\gamma = 2.0$	3.40	216	126	9	12.9
FMPH, $\gamma = 1.0$	2.80	269	190	15	12.1
SIMDRecSplit, $n = 8, b = 100$	1.81	315	303	13	22.1
SIMDRecSplit, $n = 14, b = 2,000$	1.59	368	40,168	763	52.6
SicHash, $\alpha = 0.9, p_1 = 21, p_2 = 78$	2.41	159	191	7	24.1
SicHash, $\alpha = 0.97, p_1 = 45, p_2 = 31$	2.08	143	260	17	15.1
PTHash, $\lambda = 4.0, \alpha = 0.99, C-C$	3.19	79	504	304	1.7
PTHash, $\lambda = 5.0, \alpha = 0.99, EF$	2.11	160	1,016	307	3.3
PTHash, $\lambda = 10.5, \alpha = 0.99, EF$	1.86	150	139,184	10,312	13.5
PTHash-HEM, $\lambda = 4.0, \alpha = 0.99, C-C$	3.19	88	489	9	49.1
PTHash-HEM, $\lambda = 5.0, \alpha = 0.99, EF$	2.11	166	930	13	66.9
PHOBIC, $\lambda = 3.9, \alpha = 1.0, IC-C$	3.18	94	299	9	31.3
PHOBIC, $\lambda = 4.5, \alpha = 1.0, IC-R$	2.11	240	369	11	33.5
PHOBIC, $\lambda = 6.5, \alpha = 1.0, IC-R$	1.85	217	1,341	29	46.0
PHOBIC, $\lambda = 9.0, \alpha = 1.0, IC-R$	1.74	205	12,138	251	48.3

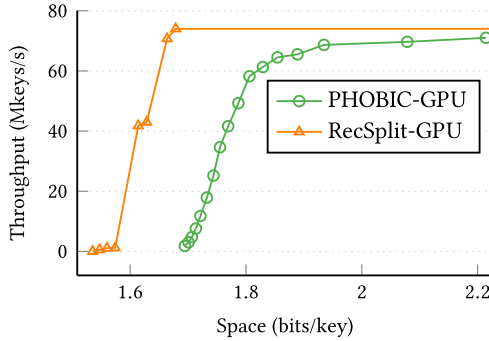


Fig. 11. Comparison between the GPU implementations of PTHash and RecSplit, for 100 million keys. The time is measured for the entire construction, including data transfers to/from the GPU.

7 Conclusion and Future Work

This article gives several results for techniques based on bucket placement: (i) we theoretically confirm the long-standing practice of placing buckets from largest to smallest, (ii) give a lower bound on the time complexity, (iii) an upper bound on the space complexity, and (iv) a bucket assignment function that is asymptotically optimal up to lower-order terms. Intuitively, our bucket assignment function results in roughly equal work for placing each bucket. In practice, it improves space efficiency by up to 0.14 bits/key compared to PTHash. Additionally, interleaved coding

allows PHOBIC to use optimal compression tuning parameters for all seeds, additionally saving up to 0.06 bits/key. In total, PHOBIC is consistently about 0.17 bits/key more space efficient than PTHash when compared for similar construction and query speed. When compared for same space consumption, PHOBIC is constructed up to 83 times faster than PTHash, while still having the same query time. Finally, our GPU implementation improves the construction by a factor of up to 62 compared to the multithreaded CPU implementation.

In future work, we intend to apply CONSENSUS [29] on PHOBIC. CONSENSUS is a technique that combines the search and encoding. We are hopeful that this puts further tradeoffs between space and time into reach.

References

- [1] Djamel Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. 2010. Fast prefix search in little space, with applications. In *ESA. Lecture Notes in Computer Science*, Vol. 6346, Springer, 427–438. DOI: https://doi.org/10.1007/978-3-642-15775-2_37
- [2] Djamel Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. 2009. Hash, displace, and compress. In *ESA. Lecture Notes in Computer Science*, Vol. 5757, Springer, 682–693. DOI: https://doi.org/10.1007/978-3-642-04128-0_61
- [3] Djamel Belazzougui and Gonzalo Navarro. 2014. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms* 10, 4, Article 23 (2014), 1–19. DOI: <https://doi.org/10.1145/2635816>
- [4] Piotr Beling. 2023. Fingerprinting-based minimal perfect hashing revisited. *ACM J. Exp. Algorithmics* 28, Article 1.4 (2023), 1–16. DOI: <https://doi.org/10.1145/3596453>
- [5] Piotr Beling and Peter Sanders. 2026. PHast—Perfect hashing made fast. In *ALENEX. SIAM*, 1–14.
- [6] Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. 2023. High performance construction of RecSplit based minimal perfect hash functions. In *ESA. LIPIcs*, Vol. 274, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 19:1–19:16. DOI: <https://doi.org/10.4230/LIPICS.ESA.2023.19>
- [7] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. 2007. Simple and space-efficient minimal perfect hash functions. In *WADS. Lecture Notes in Computer Science*, Vol. 4619, Springer, 139–150. DOI: https://doi.org/10.1007/978-3-540-73951-7_13
- [8] Andrei Z. Broder and Michael Mitzenmacher. 2003. Survey: Network applications of bloom filters: A survey. *Internet Math.* 1, 4 (2003), 485–509. DOI: <https://doi.org/10.1080/15427951.2004.10129096>
- [9] Chin-Chen Chang and Chih-Yang Lin. 2005. Perfect hashing schemes for mining association rules. *The Computer Journal* 48, 2 (2005), 168–179. DOI: <https://doi.org/10.1093/COMJNL/BXH074>
- [10] Jarrod A. Chapman, Isaac Ho, Sirisha Sunkara, Shujun Luo, Gary P. Schroth, and Daniel S. Rokhsar. 2011. Meraculous: De novo genome assembly with short Paired-End reads. *PloS One* 6, 8 (Aug. 2011), 1–13. DOI: <https://doi.org/10.1371/journal.pone.0023501>
- [11] K. P. Choi. 1994. On the medians of gamma distributions and an equation of Ramanujan. *Proc. Amer. Math. Soc.* 121 (1994), 245–251.
- [12] Yann Collet. n.d. xxHash: Extremely Fast Non-Cryptographic Hash Algorithm. Retrieved from <https://github.com/Cyan4973/xxHash>
- [13] Victoria G. Crawford, Alan Kuhnle, Christina Boucher, Rayan Chikhi, and Travis Gagie. 2018. Practical dynamic de Bruijn graphs. *Bioinformatics* 34, 24 (2018), 4189–4195. DOI: <https://doi.org/10.1093/BIOINFORMATICS/BTY500>
- [14] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. 2020. RecSplit: Minimal perfect hashing via recursive splitting. In *ALENEX. SIAM*, 175–185. DOI: <https://doi.org/10.1137/1.9781611976007.14>
- [15] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. 1992. A faster algorithm for constructing minimal perfect hash functions. In *SIGIR. ACM*, 266–273. DOI: <https://doi.org/10.1145/133160.133209>
- [16] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. 2016. Fast scalable construction of (minimal perfect hash) functions. In *SEA. Lecture Notes in Computer Science*, Vol. 9685, Springer, 339–352. DOI: https://doi.org/10.1007/978-3-319-38851-9_23
- [17] Solomon W. Golomb. 1966. Run-length encodings (corresp.). *IEEE Trans. Inf. Theory* 12, 3 (1966), 399–401. DOI: <https://doi.org/10.1109/TIT.1966.1053907>
- [18] Ragnar Groot Koerkamp. 2025. PtrHash: Minimal perfect hashing at RAM throughput. In *SEA. LIPIcs*, Vol. 338, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 21:1–21:21. DOI: <https://doi.org/10.4230/LIPICS.SEA.2025.21>
- [19] Khodor Hannoush, Camille Marchet, and Pierre Peterlongo. 2024. Cdbgtricks: Strategies to update a compacted de Bruijn graph. *bioRxiv:2024.05.24.595676*. 2024–2025. Retrieved from <https://doi.org/10.1101/2024.05.24.595676>
- [20] Stefan Hermann. 2024. PHOBIC-GPU. Retrieved from <https://github.com/stefanfred/PHOBIC-GPU>

- [21] Stefan Hermann. 2025. MorphisHash: Improving space efficiency of ShockHash for minimal perfect hashing. In *ESA. LIPIcs*, Vol. 351, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:16. DOI: <https://doi.org/10.4230/LIPICS.ESA.2025.9>
- [22] Stefan Hermann, Sebastian Kirmayer, Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. 2025. Engineering minimal k -perfect hash functions. In *ESA. LIPIcs*, Vol. 351, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 99:1–99:18. DOI: <https://doi.org/10.4230/LIPICS.ESA.2025.99>
- [23] Stefan Hermann, Hans-Peter Lehmann, Giulio Ermanno Pibiri, Peter Sanders, and Stefan Walzer. 2024. PHOBIC: Perfect hashing with optimized bucket sizes and interleaved coding. In *ESA. LIPIcs*, Vol. 308, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 69:1–69:17. DOI: <https://doi.org/10.4230/LIPICS.ESA.2024.69>
- [24] Aaron Kiely. 2004. *Selecting the Golomb Parameter in Rice Coding*. IPN Progress Report 42, 159.
- [25] Hans-Peter Lehmann, Thomas Mueller, Rasmus Pagh, Giulio Ermanno Pibiri, Peter Sanders, Sebastiano Vigna, and Stefan Walzer. 2026. Modern minimal perfect hashing: A survey. *ACM Comput. Surv.* 58, 10 (2026), 251:1–251:36. DOI: <https://doi.org/10.1145/3797036>
- [26] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. 2023. SicHash—Small irregular cuckoo tables for perfect hashing. In *ALENEX. SIAM*, 176–189. DOI: <https://doi.org/10.1137/1.9781611977561.CH15>
- [27] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. 2024. ShockHash: Towards optimal-space minimal perfect hashing beyond brute-force. In *ALENEX. SIAM*. DOI: <https://doi.org/10.1137/1.9781611977929.15>
- [28] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. 2025. ShockHash: Near optimal-space minimal perfect hashing beyond brute-force. *Algorithmica* 87, 11 (2025), 1620–1668. DOI: <https://doi.org/10.1007/S00453-025-01321-Z>
- [29] Hans-Peter Lehmann, Peter Sanders, Stefan Walzer, and Jonatan Ziegler. 2025. Combined search and encoding for seeds, with an application to minimal perfect hashing. In *ESA. LIPIcs*, Vol. 351. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 109:1–109:18. DOI: <https://doi.org/10.4230/LIPICS.ESA.2025.109>
- [30] Hans-Peter Lehmann. 2024. *Fast and Space-Efficient Perfect Hashing*. Ph.D. Dissertation. Karlsruher Institut für Technologie (KIT). DOI: <https://doi.org/10.5445/IR/1000176432>
- [31] Hans-Peter Lehmann. 2024. MPH Experiment—GitHub. Retrieved from <https://github.com/ByteHamster/MPHF-Experiments>
- [32] Victor Levallois, Yoshihiro Shibuya, Bertrand Le Gal, Yoann Dufresne, Rob Patro, Pierre Peterlongo, and Giulio Ermanno Pibiri. 2026. Kaminari: a frugal colored index for approximate k -mer queries. *Bioinformatics Advances* 6, 1 (2026), vbag120. DOI: <https://doi.org/10.1093/bioadv/vbag120>
- [33] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. 2017. Fast and scalable minimal perfect hashing for massive key sets. In *SEA. LIPIcs*, Vol. 75, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:16. DOI: <https://doi.org/10.4230/LIPICS.SEA.2017.25>
- [34] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. 2006. Perfect hashing for network applications. In *ISIT. IEEE*, 2774–2778. DOI: <https://doi.org/10.1109/ISIT.2006.261567>
- [35] Colin McDiarmid. 1989. *On the Method of Bounded Differences*. Cambridge University Press, 148–188. DOI: <https://doi.org/10.1017/CBO9781107359949.008>
- [36] Kurt Mehlhorn. 1982. On the program size of perfect and universal hash functions. In *FOCS. IEEE Computer Society*, 170–175. DOI: <https://doi.org/10.1109/SFCS.1982.80>
- [37] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. 2014. Retrieval and perfect hashing using fingerprinting. In *SEA. Lecture Notes in Computer Science*, Vol. 8504, Springer, 138–149. DOI: https://doi.org/10.1007/978-3-319-07959-2_12
- [38] Daisuke Okanohara and Kunihiko Sadakane. 2007. Practical entropy-compressed rank/select dictionary. In *ALENEX. SIAM*, 60–70.
- [39] Giulio Ermanno Pibiri. 2022. Sparse and skew hashing of k -mers. *Bioinformatics* 38, Supplement_1 (2022), i185–i194.
- [40] Giulio Ermanno Pibiri. 2024. PTHash/PHOBIC. Retrieved from <https://github.com/jermp/pthash>
- [41] Giulio Ermanno Pibiri and Roberto Trani. 2021. PTHash: Revisiting FCH minimal perfect hashing. In *SIGIR. ACM*, 1339–1348. DOI: <https://doi.org/10.1145/3404835.3462849>
- [42] Giulio Ermanno Pibiri and Roberto Trani. 2024. Parallel and external-memory construction of minimal perfect hash functions with PTHash. *IEEE Trans. Knowl. Data Eng.* 36, 3 (2024), 1249–1259. DOI: <https://doi.org/10.1109/TKDE.2023.3303341>
- [43] Giulio Ermanno Pibiri and Rossano Venturini. 2017. Efficient data structures for massive N -gram datasets. In *SIGIR. ACM*, 615–624. DOI: <https://doi.org/10.1145/3077136.3080798>
- [44] Robert F. Rice. 1979. *Some Practical Universal Noiseless Coding Techniques*. Technical Report.
- [45] Peter Sanders. 1994. Emulating MIMD Behaviour on SIMD-Machines. In *EUROSIM*. Elsevier, 313–320.

Received 3 March 2025; revised 28 August 2026; accepted 9 September 2026