Engineering Minimal k-Perfect Hash Functions

Stefan Hermann ✉ 🆔
Karlsruhe Institute of Technology, Germany

Sebastian Kirmayer 🆔
Karlsruhe Institute of Technology, Germany

Hans-Peter Lehmann ✉ 🆔
Karlsruhe Institute of Technology, Germany

Peter Sanders ✉ 🆔
Karlsruhe Institute of Technology, Germany

Stefan Walzer ✉ 🆔
Karlsruhe Institute of Technology, Germany

—————— **Abstract** ——————

Given a set S of n keys, a k-perfect hash function (kPHF) is a data structure that maps the keys to the first m integers, where each output integer can be hit by at most k input keys. When $m = \lceil n/k \rceil$, the resulting function is called a *minimal k-perfect hash function* (MkPHF). Applications of kPHFs can be found in external memory data structures or to create efficient 1-perfect hash functions, which in turn have a wide range of applications from databases to bioinformatics.

Several papers from the 1980s look at external memory data structures with small internal memory indexes. However, actual k-perfect hash functions are surprisingly rare, and the area has not seen a lot of research recently. At the same time, recent research in 1-perfect hashing shows that there is a lack of efficient kPHFs. In this paper, we revive the area of k-perfect hashing, presenting four new constructions. Our implementations simultaneously dominate older approaches in space consumption, construction time, and query time. We see this paper as a possible starting point of an active line of research, similar to the area of 1-perfect hashing.

1 Introduction

Given a set S of n keys, a k-perfect hash function (kPHF) is a data structure that maps the keys to the first m non-negative integers $[m]$, called *bins* in the following. Each output bin can be hit by at most k input keys. The output for any key not in S is undefined, so a k-perfect hash function does not need to store the input set. For $k = 1$, there is a wide

range of constructions covering different trade-offs between space consumption, query time, and construction time. The area is an active field of research with several publications each year [27, 28, 30, 31, 9, 20, 16, 3, 29, 17, 37]. However, there has been little prior work on k-perfect hashing. While several papers from the 1980s look at internal memory indexes for external hash tables, these are not full k-perfect hash functions. We refer to Section 3 for details. In this paper, we present four new k-perfect hash function constructions. Each covers a different trade-off between the three main performance metrics – space consumption, construction time, and query time.

Minimality. When $m = \lceil n/k \rceil$, we obtain a *minimal k-perfect hash function (MkPHF)* [26]. When using the MkPHF to partition 1-PHFs [30, 31], it might be beneficial to look at a stricter definition where all output bins contain exactly k keys, except possibly the last. The approaches presented in this paper fulfill the stricter definition. An alternative definition of non-minimal kPHFs could be that we could fill each output bin with either exactly k keys or 0 keys. For $k = 1$, this results in an ordinary non-minimal PHF and achieves a significant reduction in space consumption. However, for $k > 1$, the resulting space consumption is the same as the one of MkPHFs [33].

Applications. k-perfect hashing has a wide range of applications. In external memory hash tables, kPHFs can return the page on which a key is stored. External memory is usually accessed with page granularity anyway, so having to scan the page for the key only causes negligible overhead. Compared to a 1-PHF, however, this can significantly reduce the space needed. For internal memory hash tables it can be beneficial to store more than one key in each table cell [12]. Using an MkPHF, we can completely fill such a hash table. A k-perfect hash function can also be used to make the construction of 1-perfect hash functions more uniform by dividing the input to same-size partitions [30, 31].

Our Contributions. In this paper, we present four new or significantly improved k-perfect hash function constructions. Threshold-based bumping was previously only briefly described as an ad-hoc solution to aid 1-perfect hashing [30, 31]. In Section 4, we extend it through optimal thresholds and CONSENSUS-coded [31] thresholds. Perfect hashing through bucket placement [37, 20, 2] generalizes to k-perfect hashing in an obvious way. In Section 5, we introduce an optimal bucket assignment function. RecSplit [9] is a known 1-PHF, which we extend to a k-PHF in Section 6. Finally, we introduce a variant of the external memory hash table PaCHash [23] that can be used as an MkPHF in Section 7. To the best of our knowledge, we then give the first experimental evaluation of different k-perfect hash functions in Section 8.

2 Preliminaries

In this section we show the space lower bound for minimal k-perfect hash functions. We then explain preliminary data structures that we use throughout this paper.

Space Lower Bounds. The space lower bound for representing a minimal k-perfect hash function is rather easy to prove [32, 2, 23]. Let the n input keys be a subset of a universe of size u. There are $\binom{u}{n}$ possible input sets. Each behavior of an MkPHF can cover at most $\binom{u/(n/k)}{k}^{n/k}$ input sets. Therefore, the number of bits we need is at least

Table 1 Space lower bound for different values of k.

k	1	2	4	10	100	1000
Bits per key	1.443	0.943	0.589	0.300	0.046	0.006

$$\log_2 \left(\frac{\binom{u}{n}}{\left(\binom{u/(n/k)}{k}\right)^{n/k}} \right) \overset{n^2 \in o(u)}{\approx} n \cdot \left(\log_2(e) - \log_2(k^k/k!)/k \right) = n \cdot \left(\frac{\log_2(2\pi k)}{2k} + \Theta\left(\frac{1}{k^2}\right) \right).$$

In Table 1, we give a range of example values. The space lower bounds become more complex for non-minimal k-perfect hashing, and no tight bounds are known yet [2].

Retrieval Data Structures. A retrieval data structure stores a static function $f : S \to \{0,1\}^r$ with r-bit output values. It can return arbitrary results for any key not in S. This makes it possible to represent it using just rn bits. Bumped Ribbon Retrieval (BuRR) [7] is based on solving a system of linear equations and has a space consumption of $1.01rn$ bits in practice.

Elias-Fano Coding. Let $(a_i)_{i \in [n]}$ be a monotonically increasing sequence of integers. To store this sequence with Elias-Fano coding [8, 11], we split each integer into two parts. We store the lower $L = \lceil \log_2(a_{n-1}/n) \rceil$ bits directly in a packed array. Then we store the remaining upper bits $(h_i)_{i \in [n]}$ as a bit vector where the bits at indices $h_i + i$ are 1, and all other bits are 0. On this bit vector, we have $h_i = \text{select}_1(i) - i$. Together with a lookup in the packed array, we get constant access time. In total, storing the sequence needs $n \cdot (2 + \lceil \log_2(a_{n-1}/n) \rceil) + o(n)$ bits of space [8, 11].

Golomb-Rice Coding. For a parameter L, Golomb-Rice coding [14, 38] divides an integer x into the two parts $h = \lfloor x/2^L \rfloor$ and $l = x \bmod 2^L$. It encodes l using L bits in binary coding, and h in unary coding. Golomb-Rice coding is optimal for geometrically distributed x with $p = 2^{-L}$. We can store a sequence of integers with constant time access by using a common array for the lower bits, concatenating the upper bits, and using a select data structure.

3 Related Work

We now explain k-perfect hash functions and similar external memory hashing data structures from the literature. Note that Alon et al. [1] and Berman et al. [4] use the variable k for the range of the output values in 1-perfect hashing, which is now usually called m. In an online setting when building hash tables, Frei and Wehner [13] call a family of hash functions *k-ideal* if there is, for any input set of size n, some hash function that is k-perfect.

3.1 k-Perfect Hash Functions

Brute-Force Search. Similar to what can be done for 1-perfect hashing [34], we can use brute-force search to construct k-perfect hash functions. We use a hash function that can be parameterized by a seed value, changing its behavior completely. We greedily start trying seed values until we come across one seed where the hash function is k-perfect on our given input set. Then storing the index of that seed in binary coding is enough to represent the k-perfect hash function. Unfortunately, like for 1-perfect hashing, we need an exponential

number of hash function evaluations, making the approach unfeasible for large input sets. Mairson [32] shows that the probability of a random function being an MkPHF can be described by the multinomial coefficient $(n/k)^{-n} \underbrace{\binom{n}{k,\,k,\,\ldots,\,k}}_{n/k \text{ times}}$.

Bucket Placement. Perfect hashing through bucket placement [37, 20, 2] is a class of 1-perfect hash functions. The idea is to hash the input keys to small buckets of expected size λ, where λ is a tuning parameter usually in the range 2–6 keys. For each bucket, we then greedily search for a hash function seed such that the keys in that bucket do not collide with previous keys. After placing all buckets, we compress the list of seeds while retaining fast access. The first buckets are easier to place because all bins are empty. It is therefore helpful to insert the buckets in decreasing order of size. In CHD [2], the keys are hashed uniformly to the buckets. Despite sorting, the last buckets are still much harder to place than the first. Therefore, PHOBIC [20] introduces an optimal *bucket assignment function* $\beta : [0, 1] \to [0, 1]$. A key with a normalized hash value $x \in [0, 1]$ is then assigned to bucket $\lceil \beta(x)n/\lambda \rceil$. The bucket assignment function makes the first buckets much larger to allow even smaller buckets towards the end, resulting in roughly the same success probability of a seed in all buckets.

Bucket placement can be used as a k-perfect hash function by allowing up to k collisions for each output bin. The implementation [6] of CHD supports only non-minimal k-perfect hashing. In Section 5, we extend the idea through an optimized bucket assignment function.

Predecessor Dictionary. Pagh [36] suggests hashing all input keys with a hash function of range $[n^3]$ and sorting them by hash value. Now map the keys linearly to the output bins and store the smallest hash value in each bin. For a given hash value x, a *predecessor query* can give the largest value y in the list such that $y \leq x$. The index of that value is the k-perfect output bin. We implement a variant in Section 7.

Threshold-Based Bumping. Threshold-based bumping [30] is an ad-hoc solution to k-perfect hashing introduced for simplifying a 1-perfect hash function. The idea is to first hash all keys to $\lceil 0.9n/k \rceil$ bins. The constant of 0.9 is selected arbitrarily in the paper, ensuring that there are few bins that receive fewer than k keys. An additional hash function uniformly assigns a fingerprint value $\in (0, 1]$ to each key. For each bin, threshold-based bumping then stores a threshold such that at most k keys in the bin have fingerprints below or equal to the threshold. It *bumps* keys with fingerprints larger than the threshold to a second layer of the same data structure. To reduce the space consumption and to enable fast access, we restrict ourselves to a small list of possible thresholds and only store the index into the list. Threshold-based bumping has similarities with perfect hashing through fingerprinting [35] (which essentially uses 1-bit thresholds) and separator hashing [15, 25] (see Section 3.2). The idea of bumping is also used in BuRR [7], which solves a system of linear equations and bumps conflicting rows.

To make the resulting function *minimal k-perfect*, keys bumped in the second layer have to be placed into the empty slots left in the bins. Threshold-based bumping does that by storing an Elias-Fano coded list of bins that still have slots left. Using a minimal 1-PHF on the bumped keys, it can map each key to one of the empty slots. Refer to Figure 1 for an illustration. Queries hash the key to a bin. Most keys can immediately return the bin index because their fingerprint is below the threshold. Some have to look at the next layer and few have to query the fallback 1-PHF. In Section 4, we extend the approach through an optimal selection of threshold values, as well as more efficient overloading to minimize the number of bins with less than k keys.

3.2 External Memory Hash Tables

In external memory hash tables, a goal is to find objects while only accessing a small number of external memory pages. For this, different internal memory index data structures have been proposed. These index data structures are very similar to k-perfect hash functions. The main difference is that they either need to inspect more than one external memory page or that they do not fully utilize the external memory, meaning that they are not *minimal k-perfect*. Still, these data structures use similar ideas as MkPHFs or can even be adapted to MkPHFs.

Separator Hashing. Separator hashing [15, 25, 24] is a static, external memory hash table that guarantees a single disk access for each query. Each key has a *probe sequence* of output bins and a *fingerprint sequence*, both determined through a sequence of hash functions. For each external memory page, it stores a threshold value for the fingerprint, also called *separator*. When querying a key, we look at the bins along its probe sequence. If the key's fingerprint for that bin is smaller than the separator, we found the corresponding page. Otherwise, we continue scanning. A problem with the technique, especially for high load factors, is that bumped keys map back into the same range of bins. This causes a series of dependent memory accesses during queries, and complicates the construction.

External Robin-Hood Hashing. Instead of displacing keys based on their fingerprint, *external robin-hood hashing* [5] displaces the keys that are at the smallest index of their probe sequence. The internal memory then stores, for each page, the smallest distance of a contained key to its home address. Queries iterate over the distance array starting from their home address until the distance is at least what is stored in the array. If multiple keys have the same home address, more than one external memory page has to be searched.

Extendible Hashing. The idea of *extendible hashing* [10] is to store an internal memory table holding pointers. Each row points to an external memory page holding the keys hashed to that table row. If the keys hashed to several (adjacent) table rows fit onto an external memory page together, extendible hashing stores the same pointers multiple times. The approach achieves a load factor of about 69% [36].

PaCHash. PaCHash [23] is designed as a static external memory hash table with variable-length keys. However, we only explain the case that each external memory page can hold k fixed-length keys. Then PaCHash uses a hash function of range $[a \cdot n/k]$, where a is a tuning parameter. This divides the keys into different buckets.[1] For each external memory page, we store the index of the first bucket overlapping it. Using Elias-Fano coding (see Section 2), this takes $n/k(2 + \log(a))$ bits. To locate a key, we perform a predecessor query on that sequence to determine which external memory page should be loaded. If a bucket overlaps the page boundaries, the index cannot determine whether the key is on the page or on the previous page. In that case, which happens for roughly $1/a$ of the queries, PaCHash loads both adjacent pages. In Section 7, we explain how we can adapt PaCHash to an MkPHF.

---

[1] For consistency within this paper, we interchange the terms *bin* and *bucket* of the PaCHash paper.

■ **Listing 1** Query algorithm of k-perfect hashing trough threshold-based bumping. Hash function h_X uniformly maps into the set X.

```
Function query(key ∈ S)
    numBins := ⌈(n/k)/γ⌉
    bin := h_[numBins](key)
    thresholdIndex := thresholds[bin]
    fingerprint := h_[0,1](key)
    if fingerprint ≤ T_thresholdIndex
        return bin
    else // bumped to next layer
        return numBins + query next layer
```
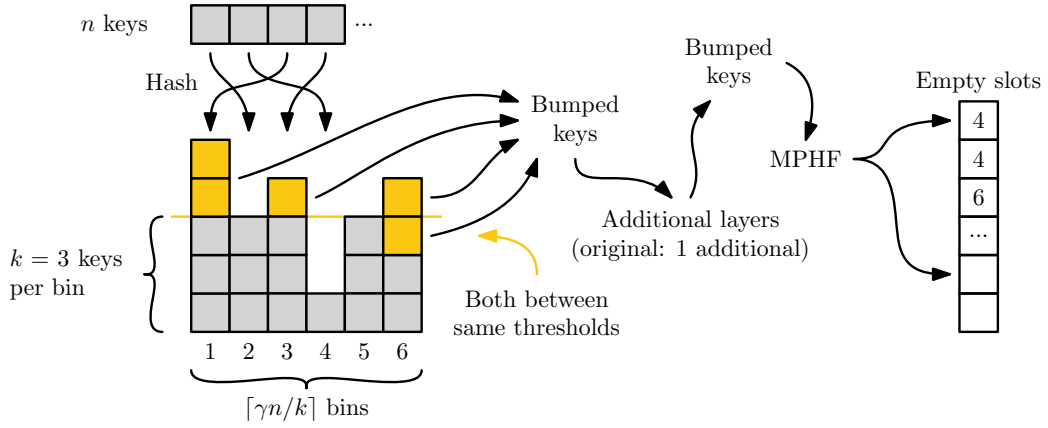

■ **Figure 1** Threshold-based k-perfect hashing. Figure adapted from [30].

4 k-Perfect Hashing Through Threshold-Based Bumping

Threshold-based bumping is introduced in ShockHash-Flat [30] as an ad-hoc implementation to aid 1-perfect hashing. In this section, we enhance it significantly. Remember from Section 3.1 that threshold-based bumping hashes each key directly to a bin. Additionally, we hash each key to determine a uniform fingerprint value $\in (0, 1]$. Each bin stores a threshold such that at most k keys with fingerprints smaller than or equal to the threshold remain. The other keys are *bumped* to another layer of the same data structure. Figure 1 gives an illustration.

4.1   Overloading

To make it more likely that each bin receives at least k keys, we *overload* the bins to contain more than k keys in expectation. While the ad-hoc implementation [30] already overloads the first layer, we now take this to its logical conclusion and overload all layers. More concretely, we start with $\lceil \frac{n}{k\gamma} \rceil$ bins where $\gamma > 1$ is the overloading factor. We determine the thresholds to store, which leaves us with n' bumped keys. From the remaining output bins, we then use $\lceil \frac{n'}{k\gamma} \rceil$ bins for the next layer. We continue this as long as the total number of bins used is still smaller than $\lceil \frac{n}{k} \rceil$. Only for the remaining keys, we have to use 1-perfect hashing to puzzle them back into the under-filled bins. Even with overloading, we maintain expected constant query times and expected linear construction times. Preliminary experiments

with $k = 1000$ show that overloading all layers reduces the number of keys bumped in the last layer from 0.2% (with the original two layers) to 0.02% (with 5 layers). Because the fallback 1-PHF needs non-negligible space, this reduces the overall space consumption. In our implementation, we use the 1-PHF FiPS [27] due to its small constant space overhead when using it for small sets.

4.2 Optimizing the Thresholds

For fast access we use a fixed number of bits to represent the threshold of each bin, which gives us t possible threshold values. In the following we consider the value of those t thresholds. For each bin we then only store the index of the threshold. We model the fingerprint of a key as a uniformly distributed hash value $x \in (0, 1]$. If x is greater than the threshold of the key's bin, the key is bumped. It is less likely that we need threshold values close to 0 because the expected number of keys in each bin is $k\gamma$. The ad-hoc implementation [30] therefore heuristically uses uniformly spaced thresholds in $[\frac{2}{3}, 1]$. In this section, we derive an optimal selection of thresholds.

Our goal is to find the t thresholds $T_1 < \cdots < T_t \in [0, 1]$ such that they minimize the expected number of empty slots. As a fallback option, we use $T_1 = 0$, bumping all keys. We also use $T_t = 1$. A value of $T_t < 1$ would mean that keys in the range $(T_t, 1]$ always get bumped, which would counteract our increased load factor. Therefore, we need to determine the values of the remaining $t - 2$ thresholds. The number of keys in a bin is binomially distributed. To simplify analysis, we approximate the number of keys using a Poisson distribution with γk keys in expectation. We consider the sorted fingerprints $0 < x_1 < x_2 < \ldots$ of the keys. The distance between any two consecutive keys $(x_{i+1} - x_i)$ and between 0 and x_1 is exponentially distributed with an expected distance of $1/(\gamma k)$. Hence, it is equivalent to obtain the fingerprints by sampling consecutive fingerprints using this exponential distribution until we pass 1. However, for notational convenience we sample an infinite number of fingerprints in this way and allow them to exceed 1. This does not change the distribution of the fingerprints smaller 1.

To minimize the number of empty slots we always choose the highest possible threshold that is still smaller than x_{k+1}, since we need to ensure that at most k keys remain in the bin. Hence, threshold T_i is chosen if $T_i < x_{k+1} \leq T_{i+1}$ for $i \in [t]$ and we formally set $T_{t+1} := \infty$ to simplify notation. A fingerprint x_j results in an empty slot if it is bumped $(x_j > T_i)$ *and* it is one of the k keys which we want to keep in the bin $(x_j < x_{k+1})$. If we know x_{k+1}, then the k smaller fingerprints are uniformly distributed in the range $(0, x_{k+1})$. Hence, the expected number of empty slots is $k\frac{x_{k+1} - T_i}{x_{k+1}}$. The fingerprint x_{k+1} follows a gamma distribution with shape $k + 1$ and rate γk. Let $\phi(x)$ be the density function of that gamma distribution. We can calculate the expected number of empty slots of the bin as

$$
\begin{aligned}
E := \mathbb{E}[\text{empty slots}] &= \int_0^\infty \mathbb{E}[\text{empty slots} \mid x_{k+1} = s]\phi(s)\mathrm{d}s \\
&= \sum_{i=1}^t \int_{T_i}^{T_{i+1}} \mathbb{E}[\text{empty slots} \mid x_{k+1} = s]\phi(s)\mathrm{d}s \\
&= \sum_{i=1}^t \int_{T_i}^{T_{i+1}} k\frac{s - T_i}{s}\phi(s)\mathrm{d}s
\end{aligned}
\tag{1}
$$

For E to be minimal, its derivative with respect to $T_2, \ldots, T_{t-1}$ must be zero. We have:

$$0 \stackrel{!}{=} \frac{\mathrm{d}E}{\mathrm{d}T_i} = \frac{\mathrm{d}}{\mathrm{d}T_i} \int_{T_{i-1}}^{T_i} k \frac{s - T_{i-1}}{s} \phi(s)\mathrm{d}s + \frac{\mathrm{d}}{\mathrm{d}T_i} \int_{T_i}^{T_{i+1}} k \frac{s - T_i}{s} \phi(s)\mathrm{d}s$$

$$= k \left(\frac{T_i - T_{i-1}}{T_i} \phi(T_i) - \int_{T_i}^{T_{i+1}} \frac{1}{s} \phi(s)\mathrm{d}s \right)$$

$$\iff T_{i-1} = T_i - \frac{T_i}{\phi(T_i)} \int_{T_i}^{T_{i+1}} \frac{1}{s} \phi(s)\mathrm{d}s \tag{2}$$

If we knew T_t and T_{t-1} we could inductively obtain the remaining thresholds. However, we only know that $T_t = 1$ and $T_1 = 0$. Our implementation therefore fixes $T_t = 1$ and uses binary search of T_{t-1} until T_1 is close to 0. This minimizes the number of empty slots under the necessary conditions $T_t = 1$ and $T_1 = 0$. For better intuition, we show in the full version of this paper [18] that the density of thresholds follows a gamma distribution for $t \to \infty$. The query algorithm of our data structure is shown in Listing 1.

4.3 Tighter Packing With Retrieval

Sometimes there is more than one key between the relevant thresholds. If we take the smaller threshold, we keep empty slots that we later need to repair. If we take the larger threshold, we would have more than k keys in the bin. In this case, we extend the data structure by querying a 1-bit retrieval data structure (see Section 2) on all keys that have fingerprints between these two thresholds. The bit tells us whether this key should be kept in the bin or if it should be bumped. This costs us 1 bit per key for all keys between thresholds, even the ones we do *not* bump. However, overall, this reduces space because each bumped key leaving an empty slot costs us several bits of space later to repair. The technique trades some query time for reduced space consumption.

4.4 CONSENSUS-Coded Thresholds

CONSENSUS [31] is a technique for space-efficiently storing seeds for randomized data structures. It basically uses a fixed number of bits to store the seed of each subtask. This means that for some subtasks, there might be multiple successful seeds that can be represented with these bits, while for other tasks, there might be none. The novel idea of CONSENSUS is to concatenate the current seed with previous seeds. This means that if we change one seed, we completely change the meaning of all following seeds. That way, if one task has no successful seeds, we can backtrack to another successful seed in a previous task and get a completely new chance for success in the current task.

While the thresholds in threshold-based k-perfect hashing are not classical seeds, we can still apply the CONSENSUS idea. The similarity is that for some bins, we might have more than one threshold that gives a sufficiently small number of empty slots, while for other bins, there might be none. We now use the thresholds of previous bins as a seed value when determining the fingerprints of the next bin. Selecting a different threshold in a previous bin therefore gives us a new chance to reduce the number of empty slots.

A problem with this approach is that there might be some bins that are much larger than γk caused by uniformly hashing keys to bins. Even if we try another hash function seed, the bins are biased towards having too many keys. Therefore, finding a seed such that exactly k keys are below one of the thresholds is unlikely. However, we still have the option to bump too many keys and repair the empty slots later. We need this option anyway because there
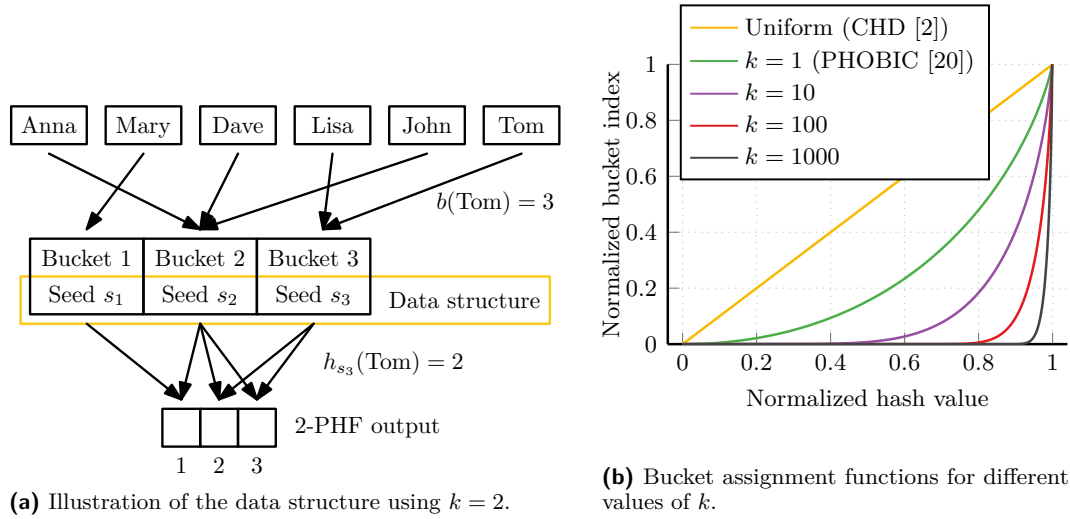
Listing 2 Query algorithm of k-perfect hashing trough bucket placement. Hash function h_X uniformly maps into the set X.

```
Function query(key ∈ S)
    bucket := ⌊(n/λ) · β_k(h_[0,1](key))⌋
    seed := seeds[bucket]
    return h_[⌈n/k⌉](key, seed)
```

(a) Illustration of the data structure using $k = 2$.

(b) Bucket assignment functions for different values of k.

Figure 2 Minimal k-perfect hashing through bucket placement.

might be bins that are not full enough. Therefore, for each actual size of a bin, we determine how many keys we have to accept bumping if we want to have at least one valid threshold in expectation. We then accept a seed if the number of empty slots is less than this limit.

5 k-Perfect Hashing Through Bucket Placement

Remember from Section 3.1 that perfect hashing through bucket placement [37, 20, 2] first hashes n keys to $\frac{n}{\lambda}$ buckets. For each bucket, it then searches for a hash function seed that can place the keys without collisions. The *bucket assignment function* $\beta : [0, 1] \to [0, 1]$ varies the expected size of each bucket such that placing each bucket has the same success probability. Figure 2a gives an illustration of the idea and Listing 2 describes the query algorithm. The idea generalizes trivially to k-perfect hashing, where we now place a bucket if all output bins stay below k collisions. Because k-perfect hashing has different success probabilities than 1-perfect hashing, we need a different bucket assignment function β_k. In this section we show how to choose β_k such that each bucket has approximately the same success probability.

We first need some general observations about bucket assignment functions. For key x, the expected size of its bucket with normalized index $\beta_k(x)$ is $\lambda/\beta'_k(x)$.[2] Let us assume that an x-fraction of keys is already mapped to bins. Then let $p_k(x)$ be the probability that a

---

[2] In the close neighborhood of x and for infinitesimal δ, a δ-fraction of the hash range (used by δn keys in expectation) is shared by a $(\beta'_k(x) \cdot \delta)$-fraction of the $\frac{n}{\lambda}$ buckets. The quotient is $\delta n/(\beta'_k(x)\delta\frac{n}{\lambda}) = \lambda/\beta'_k(x)$ [20].

random bin is not full at this point in the construction process. When placing a bucket of size s, a seed is successful if all keys map to bins that are not full yet which has a probability of $p_k(x)^s$ (ignoring self collisions). The optimal solution for $k = 1$ satisfies that $p_k(x)^{\lambda/\beta'_k(x)}$ is some constant [20]. Because λ is a constant, the equation simplifies to $p_k(x)^{1/\beta'_k(x)} = C_k$. Rearranging gives

$$\beta'_k(x) = \ln(p_k(x))/\ln(C_k). \tag{3}$$

For $k = 1$, a bin is full if there is a single key mapped to it. We therefore have $p_1(x) = 1 - x$ and $\beta'_1(x) = \ln(1 - x)/\ln(C_1)$. The constant C_k has to be chosen such that all keys are placed after the last bucket, i.e. $\beta_k(1) = 1$. After integration and proper choice of C_1 we have $\beta_1(x) = x + (1 - x)\ln(1 - x)$, the optimal bucket assignment function for $k = 1$ [20]. The difficulty for $k > 1$ is finding $p_k(x)$. Once we have $p_k(x)$, we use Equation (3) to obtain the bucket assignment function. Note that we assume without proof that using Equation (3) is still optimal for $k > 1$.

Model.   To determine $p_k(x)$, we model the following process where we insert bk keys into b bins of capacity k. We initialize a counter $c_i = 0$ for each bin $i \in [b]$. The keys are handled sequentially as described in the following. For each key we uniformly sample bins. Sampling a bin $s \in [b]$ has two possible outcomes:

- $c_s < k$ (*Success*): We increment the counter c_s by one and continue with the next key.
- $c_s \geq k$ (*Failure*): In this case, we also increment the counter c_s by one. Note that this does *not* change anything in terms of success probabilities, since the bin is already full anyway. We continue with sampling a new bin for the same key.

Determining $p_k(x)$ Numerically.   The reason for incrementing the counter even if the bin is full is that the counter of each bin now follows a Poisson distribution (assuming that the number of bins b is large). We show how $p_k(x)$ can be expressed as the solution of an integro-differential equation in the full version of this paper [18]. However, solving this equation is numerically unstable. In the following we give a numerically stable solution for $p_k(x)$. We repeatedly choose values μ. We use each μ as the expected value of the Poisson distribution that describes the distribution of the counters. For each μ we calculate the probability p that a key is inserted successfully, as well as the fraction of inserted keys x. Hence, for each μ we obtain p and x which we use to numerically determine $p_k(x)$. We choose the values μ such that the distance of consecutive x values is sufficiently small. The success probability p for a given μ is the probability that $c_s < k$, which we calculate using the cumulative distribution function of the Poisson distribution. To determine the fraction of inserted keys x for a given expected value μ we consider the counter of the bins as described in the model. For a bin where the counter is higher than k, we charge k successfully inserted keys, because all other incrementations are from failures. We then divide by k, since we are interested in the relative number of inserted keys. Let X follow a Poisson distribution with mean μ then $x = \frac{1}{k}(\mathbb{E}[X \mid X \leq k] + k\mathbb{P}[X > k]) = \frac{1}{k}\mathbb{E}[X \mid X \leq k] + \mathbb{P}[X > k]$.

The Bucket Assignment Function.   The final bucket assignment function can be obtained by numerically evaluating the integral of Equation (3) for $\beta_k(x)$. The constant C_k can be ignored by normalizing the integral as a last step such that the boundary condition $\beta_k(1) = 1$ is met. Refer to Figure 2b for an illustration of the bucket assignment functions with different values of k. Larger k make the function more aggressive, further increasing the size of the first buckets. For example, for $k = 100$, 82% of the keys are hashed to 1% of buckets. Our implementation tabulates the bucket assignment function for specific k and uses linear interpolation to maintain a fast query time.
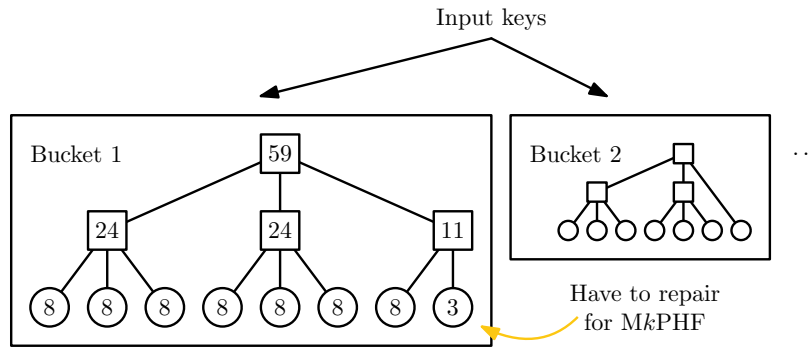
Figure 3 Illustration of RecSplit with $k = 8$ and lowest-layer fanout $\ell = 3$.

6 k-Perfect RecSplit

RecSplit [9] is a very space-efficient 1-perfect hash function. We first explain RecSplit and then show how we adapt it to a k-perfect hash function.

RecSplit. RecSplit [9] first hashes each key to a bucket of expected size b, where b is a tuning parameter with default value $b = 2000$ in the implementation. Within each bucket, RecSplit then uses brute-force to search for a hash function seed that divides the keys into two (or more) subsets. The number of these subsets (*fanout*) depends on the *splitting strategy*. This continues recursively until we are left with small input sets of $\leq \ell$ keys in the leaf nodes. These subsets are small enough that it is feasible to search for a perfect hash function using brute-force. RecSplit stores the hash function seeds determined at each node using Golomb-Rice coding (see Section 2) in DFS order. An Elias-Fano coded sequence (see Section 2) stores where the encoding of each bucket starts, as well as the total number of keys before it. A query traverses the splitting tree of the corresponding bucket, evaluating the hash function in each node. When descending into a subtree, it sums up the number of keys in sibling nodes left to it. Therefore, the final hash function value is given by the number of keys before the bucket, the number of keys to the left in the tree, and the final brute-force hash function value in the leaf. The RecSplit splitting strategy ensures that all subtrees except possibly the last are a complete tree where each leaf node receives exactly ℓ keys. Therefore, only the very last leaf in any tree can have a size smaller than ℓ. We illustrate RecSplit in Figure 3. Through the use of the buckets, RecSplit achieves linear time construction and constant time queries.

Main Idea. Our k-perfect adaptation of RecSplit is (almost) straight-forward. We perform splittings until we are left with leaves of size k. These then do not need to store seeds. We now use the parameter ℓ to select the fanout of the layer above. The main difficulty is that the last leaf in each bucket can have size $< k$. This would make the result a non-minimal kPHF. In Section 6.1, we explain how we can merge the leaves of adjacent buckets in order to get output bins of size k. As an alternative, in Section 6.2, we explain *nested k-perfect hashing*. This can avoid the problem by ensuring that each bucket's size is a multiple of k.

6.1 Merging Leaves of Size $< k$

From the prefix sum of bucket sizes, we know the number of keys x before each bucket. We start placing the output of the bucket's tree at the offset $\lfloor x/k \rfloor$. This means that we do not leave space for the previous bucket's leaves of size $< k$. Only if the total number of keys in
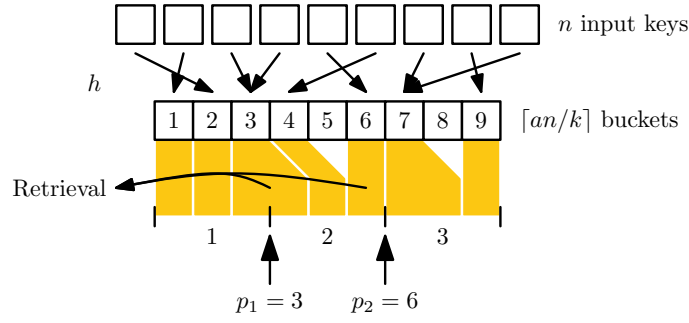
Figure 4 Illustration of PaCHash for k-perfect hashing with $a = k = 3$. Queries for keys hashed to bucket 3 cannot directly determine whether their output bin is 1 or 2 by looking at the cut-points p_i. Therefore, they have to query a retrieval data structure. For the single key in bin 6, the placement would actually be unambiguous. However, from looking at p_2, we cannot see that all keys in bin 6 have to map to the left of the threshold. Therefore, we still need to query the retrieval data structure. For p_2, can store either 6 or 7. We select the bin with fewer keys to reduce the number of keys in the retrieval data structure.

non-full leaves exceeds k, we get an additional output bin. For this, we calculate a splitting hash function on the last leaf that splits away the keys needed to fill the leaf. We store the splitting seed as an additional hash function seed in the Golomb-Rice coded seeds.

During queries, if we arrive in a leaf of size $< k$, there are two cases. (1) If the remainder of the bucket position does not cross k, we scan forward the following buckets until one does. The output bin is then the last bin of that bucket. (2) If the current bucket is the one letting the remainder cross k, we evaluate the splitting hash function. Then we either return the last bin of the current bucket, or we scan forward like in the first case.

Note that this means that we have to scan forward the bucket sizes when querying a leaf of size $< k$. However, because the number of keys in a bucket is random and much larger than k, we can assume we get a uniform random remainder. Therefore, in expectation, we only have to scan two buckets.

6.2 Avoiding Leaves of Size $< k$ Through Nesting

Another technique is to completely avoid leaves of size $< k$ in the first place. We can use a minimal b-perfect hash function (where k divides b) to partition the keys. In general, we can nest different k-perfect hash functions to interpolate their trade-offs. One function (maybe fast and less space-efficient) can map the input set to buckets of k' keys. Then, in each bucket, we can construct a k-perfect hash function. We could use the idea to implement a k-perfect version of CONSENSUS-RecSplit. However, nesting different kPHFs would introduce another dimension of experiments and make this paper harder to follow. We therefore do not go into detail about nesting or k-perfect CONSENSUS-RecSplit.

7 k-Perfect PaCHash

As we explain in Section 3.2, PaCHash hashes each key to one of $\lceil an/k \rceil$ buckets. The variable a is a tuning parameter. For each output bin i, it stores the first bucket p_i overlapping it using Elias-Fano coding (see Section 2). A predecessor query on that list then maps the bucket to the respective output bin. Figure 4 illustrates the data structure. During queries, if a searched bucket is contained in the sequence, it is not clear whether the key is supposed

▮ **Listing 3** Query algorithm of k-perfect PaCHash. If the range returned by the internal memory data structure is ambiguous, we have to query the retrieval data structure. Hash function h_X uniformly maps into the set X.

```
Function query(key ∈ S)
    b := h_{[⌈an/k⌉]}(x)
    find i such that p_{i-1} < b ≤ p_i                       // predecessor query
    if p_i = b
        i := i − 1                                          // b may start in previous block
    find first j such that p_j > b                          // predecessor query or scan
    if i = (j − 1)
        return i
    else
        return i + retrievalQuery(key)                      // (j-1-i)-bit retrieval
```

to be in the bin or in the bin before. In fact, it might be possible that d bins store the same bucket index. We get a similar problem if a bucket starts exactly at a bin boundary. Then the index cannot tell whether the bucket overlaps or not. The external memory implementation then loads all $d + 1$ candidate bins. However, for k-perfect hashing, we need a different strategy.

k-Perfect Adaptation. To decide for one of the $d + 1$ bins using only the index data structure, we use a $\lceil \log_2(d+1) \rceil$ bit retrieval data structure (see Section 2). The stored value tells us which of the bins to return. Since we rarely need more than one bit, we simply store the numbers bitwise in a single 1-bit retrieval data structure. Listing 3 gives the pseudocode of the query algorithm. PaCHash expects $1/a$ of the queries to load one external memory page too much [23]. In the context of k-perfect hashing, this means that for $1/a$ of the queries, the output bin is ambiguous. This means that the retrieval data structure essentially stores n/a bits. By selecting $a = k$, we get a total space consumption of $n/k(3 + \log(k))$, which already gets close to the lower bound for large k (see Section 2). An advantage of the technique is that its construction through (integer) sorting and a single scan is very simple. Like the other approaches it takes linear time. Queries take expected constant time due to the distribution of values in the Elias-Fano data structure [23]. This adaptation is briefly mentioned in previous papers [27, 31] but without an implementation. The CONSENSUS paper [31] gives a simple analysis for the resulting space consumption.

8 Evaluation

In this section, we compare our approaches with CHD [2], as well as the ad-hoc implementation of threshold-based bumping [31] from the literature. While CHD is not *minimal k*-perfect, we still include it in our comparison with a load factor of 97% to give a feeling for the performance of our approaches. With this paper, we build a foundation for future work on k-perfect hashing, contributing implementations for future papers to compare against.

We run our experiments on an Intel i7 11700 processor with 8 cores and a base clock speed of 2.5 GHz. The machine runs Rocky Linux 9.5 with Linux 5.14.0. We use the GNU C++ compiler version 11.2.0 with optimization flags `-O3 -march=native`. Following the practice in 1-perfect hashing [20, 27, 28], our input set consists of 100 million random strings of uniform random length $\in [10, 50]$. Because all approaches hash the input keys anyway, the choice of the input set is not very important. Our source code is public under the General Public License [19].

Because each approach has configuration parameters, it can cover a range of trade-offs between space consumption, construction time, and query time. To give an intuitive overview, we give visual plots showing different parameters of each approach. Figure 5a gives the trade-off between construction time and space consumption for three different values of k. Figure 5b gives the trade-off between query time and space consumption. We also give a *dominance map* [27, 7, 28] of the trade-off. For each point having a specific trade-off between construction time and space consumption, the dominance map in Figure 5a shows the approach achieving the best query time. As such, it can be seen as a "front view" of the 3-dimensional Pareto space. The dominance map in Figure 5b shows the method with the fastest construction for a given trade-off between space consumption and query performance. We show a selection of representative configurations from the Pareto front in the full version of this paper [18]. In the following, we discuss the different approaches in detail.

Threshold-Based Bumping. Compared to the ad-hoc implementation [31], our implementation of threshold-based bumping offers up to 3 times smaller space overhead due to the optimized thresholds. Only for $k = 100$, the linearly spaced ad-hoc thresholds happen to be close to optimal, so we only achieve small space improvements. For growing k, threshold-based bumping gets closer to the space lower bound, while the space overhead of most other approaches increases. Figure 5a also shows that our implementation is up to 50% faster to construct than the next best technique. This is mainly because the ad-hoc implementation fully sorts the keys by their bin and fingerprint, while we only partition by the bin and then use quickselect [21] to find the k-th key. By packing using retrieval, we can achieve smaller space consumption. This has only a small influence on construction and query times. Finally, using CONSENSUS-coded thresholds, we achieve smaller space consumption but slower construction and queries.

Hash and Displace. For the hash and displace technique, Rice coding (see Section 2) achieves lower space consumption than compact coding at the same construction time. This comes at the cost of slower queries, as can be seen in Figure 5b, due to the select queries. Compared to CHD [2], an implementation from the literature, our construction is much faster. Even though we use CHD with a load factor of just 97%, our construction gets close to its space consumption. The CHD implementation from the literature crashes for $k = 1000$, while our implementation supports large k as well. Our measurements show that the hash and displace technique with compact coding is fast to query, especially for smaller k. This is in contrast to the implementation from the literature, which has much slower queries. In general, the hash and displace technique seems not competitive for k-perfect hashing, especially for large k.

k-RecSplit. Our k-perfect adaptation of RecSplit achieves the smallest space consumption. With just 10% overhead over the lower bound, it has a large margin to the next smallest competitor. This is not surprising, since the RecSplit paper [9] shows that the recursive splitting idea converges towards the space lower bound. However, its construction is significantly slower than the other approaches in Figure 5a. Because splittings become even more expensive for large k, we do not include the configuration with 10% space overhead for $k = 1000$. The queries are slow compared to other approaches.

k-PaCHash. Our k-perfect adaptation of PaCHash is very simple to describe and fast to construct through sorting followed by a linear scan. However, its queries are slow due to the predecessor queries in the Elias-Fano coded sequence of bucket indexes.
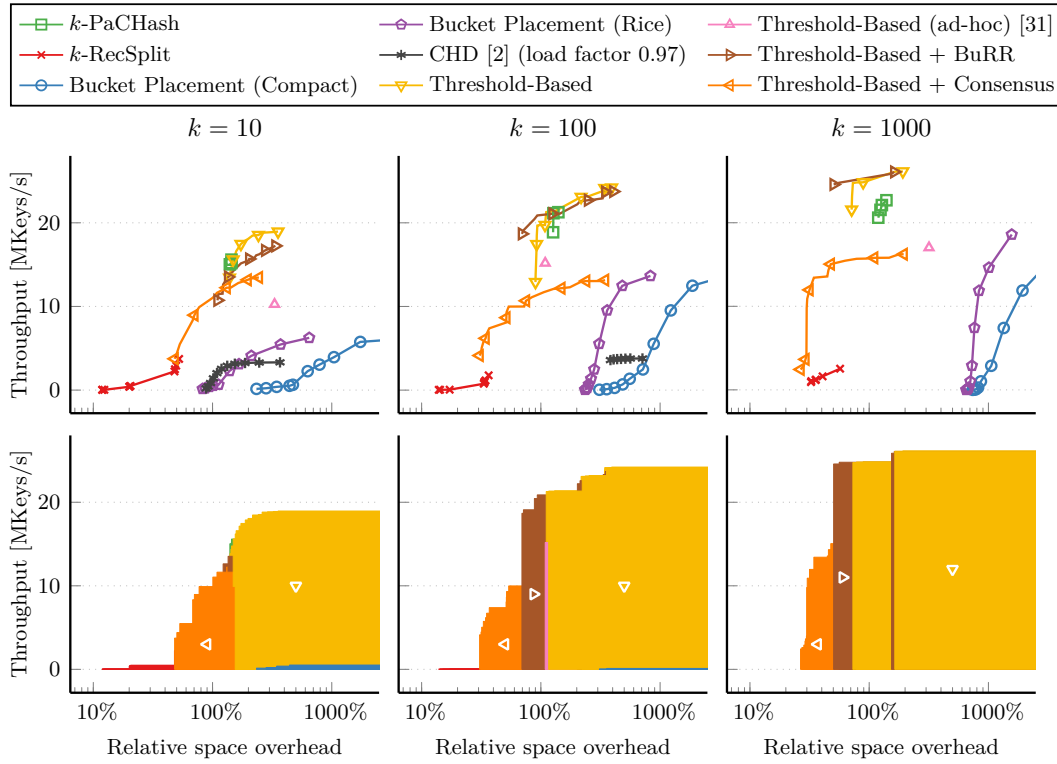
(a) Space consumption versus construction time. The bottom row shows dominance maps indicating the approach with the fastest queries, given a specific trade-off between space and construction time.
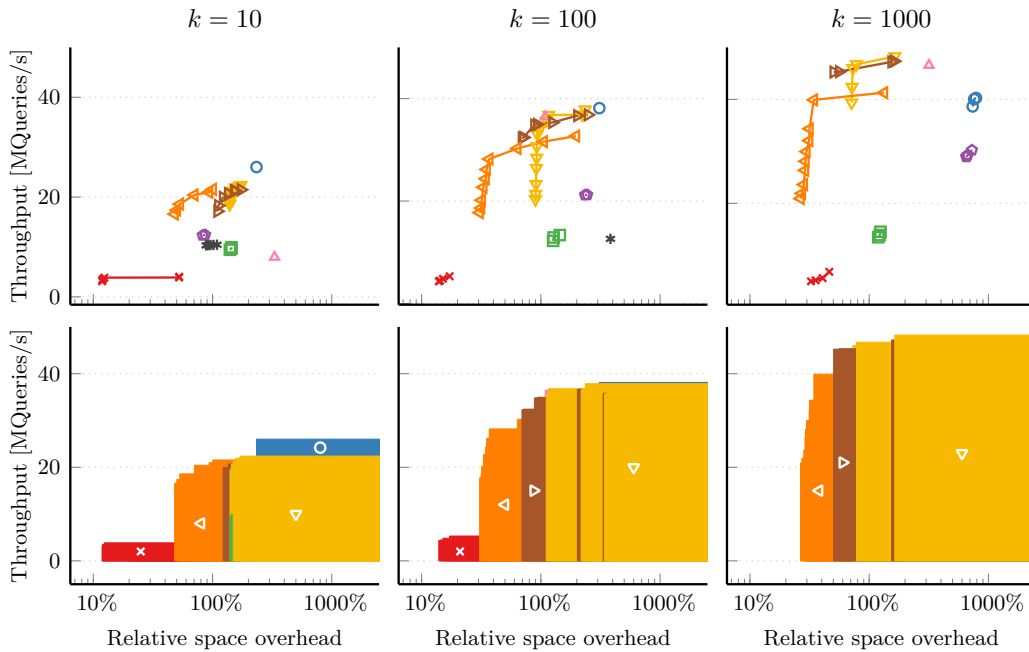
(b) Space consumption versus query time. The bottom row shows dominance maps indicating the approach with the fastest construction, given a specific trade-off between space and query time.

Figure 5 Comparison of minimal k-perfect hash functions. A relative space overhead of 100% means that the data structure needs twice the lower bound. Refer to Table 1 for a table with lower bounds in bits per key.

9   Conclusion

To our knowledge, we present the first paper fully dedicated to presenting k-perfect hash functions. The area of k-perfect hashing has been known for a long time, but has not received a lot of attention. With the advent of very space-efficient 1-perfect hash functions, however, the area is becoming more and more relevant. In this paper, we kickstart the development of kPHFs by introducing four new or significantly improved constructions. Our constructions cover various trade-offs between space consumption, construction time, and query time.

With threshold-based bumping, we significantly improve a k-perfect hash function that was previously only described briefly as an ad-hoc solution in 1-perfect hashing papers [30, 31]. We achieve a significant reduction in bumped keys by combining it with retrieval, and we optimize the threshold function. By combining the idea with CONSENSUS coded seeds, we achieve a significant reduction in space consumption while retaining its fast queries. We also extend the idea of perfect hashing through bucket placement by optimizing the bucket assignment function for k-perfect hashing. As a very space-efficient alternative, we adapt RecSplit [9] to construct k-perfect hash functions. The main difficulty is to merge leaf nodes that have size $< k$. Finally, we adapt PaCHash [23] to a k-perfect hash function by combining it with a retrieval data structure.

Our extensive evaluation compares our new techniques with competitors from the literature. Our constructions are simultaneously faster to construct, faster to query, and more space-efficient than previous approaches. The most promising techniques are threshold-based bumping and k-RecSplit. While k-RecSplit achieves the best space consumption, it is slow to construct and query. Threshold-based bumping is very fast to construct and query while still giving competitive space consumption.

Future Work.   This paper revives the research area of k-perfect hashing. In the future, we expect a wide range of new techniques, like there is in 1-perfect hashing. The approaches presented in this paper give a starting point outlining the trade-off. Especially our improvements to threshold-based bumping can now be used to improve the space efficiency of CONSENSUS-RecSplit [31]. An open theoretical problem is the space lower bound for non-minimal k-perfect hashing.

References

1   Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995. `doi:10.1145/210332.210337`.

2   Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *ESA*, volume 5757 of *Lecture Notes in Computer Science*, pages 682–693. Springer, 2009. `doi:10.1007/978-3-642-04128-0_61`.

3   Piotr Beling. Fingerprinting-based minimal perfect hashing revisited. *ACM J. Exp. Algorithmics*, 28:1.4:1–1.4:16, 2023. `doi:10.1145/3596453`.

4   Francine Berman, Mary Ellen Bock, Eric Dittert, Michael J. O'Donnell, and Darrell Plank. Collections of functions for perfect hashing. *SIAM J. Comput.*, 15(2):604–618, 1986. `doi:10.1137/0215044`.

5   Pedro Celia. External robin hood hashing. Technical report, Computer Science Department, Indiana University. TR246, 1988.

6   Davi de Castro Reis, Djamel Belazzougui, Fabiano Cupertino Botelho, and Nivio Ziviani. CMPH – C minimal perfect hashing library. `http://cmph.sourceforge.net/`, 2012.

7   Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership using ribbon. In *SEA*, volume 233 of *LIPIcs*, pages 4:1–4:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPICS.SEA.2022.4`.

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

9 Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. RecSplit: Minimal perfect hashing via recursive splitting. In *ALENEX*, pages 175–185. SIAM, 2020. `doi:10.1137/1.9781611976007.14`.

10 Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing - A fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979. `doi:10.1145/320083.320092`.

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

12 Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005. `doi:10.1007/S00224-004-1195-X`.

13 Fabian Frei and David Wehner. Bounds for c-ideal hashing. In *FCT*, volume 14292 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2023. `doi:10.1007/978-3-031-43587-4_15`.

14 Solomon W. Golomb. Run-length encodings. *IEEE Trans. Inf. Theory*, 12(3):399–401, 1966. `doi:10.1109/TIT.1966.1053907`.

15 Gaston H. Gonnet and Per-Åke Larson. External hashing with limited internal storage. *J. ACM*, 35(1):161–184, 1988. `doi:10.1145/42267.42274`.

16 Ragnar Groot Koerkamp. Ptrhash: Minimal perfect hashing at RAM throughput. *CoRR*, abs/2502.15539, 2025. `doi:10.48550/arXiv.2502.15539`.

17 Stefan Hermann. MorphisHash: Improving space efficiency of shockhash for minimal perfect hashing. *CoRR*, abs/2503.10161, 2025. `doi:10.48550/arXiv.2503.10161`.

18 Stefan Hermann, Sebastian Kirmayer, Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. Engineering minimal k-perfect hash functions. *CoRR*, abs/2504.20001, 2025. `doi:10.48550/arXiv.2504.20001`.

19 Stefan Hermann, Sebastian Kirmayer, Hans-Peter Lehmann, and Stefan Walzer. Engineering Minimal k-Perfect Hash Functions. Software, swhId: `swh:1:dir:e6756f018691a80adc68d839fd3617d8f5e2d0b0` (visited on 2025-09-08). URL: `https://github.com/stefanfred/engineering-k-perfect-hashing`, `doi:10.4230/artifacts.24698`.

20 Stefan Hermann, Hans-Peter Lehmann, Giulio Ermanno Pibiri, Peter Sanders, and Stefan Walzer. PHOBIC: perfect hashing with optimized bucket sizes and interleaved coding. In *ESA*, volume 308 of *LIPIcs*, pages 69:1–69:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. `doi:10.4230/LIPICS.ESA.2024.69`.

21 C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4(7):321–322, July 1961. `doi:10.1145/366622.366647`.

22 Sebastian Kirmayer. Engineering k-perfect hashing. Abschlussarbeit - Bachelor, Karlsruher Institut für Technologie (KIT), 2024. `doi:10.5445/IR/1000179026`.

23 Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. PaCHash: Packed and compressed hash tables. In *ALENEX*, pages 162–175. SIAM, 2023. `doi:10.1137/1.9781611977561.CH14`.

24 Per-Åke Larson. Linear hashing with separators - A dynamic hashing scheme achieving one-access retrieval. *ACM Trans. Database Syst.*, 13(3):366–388, 1988. `doi:10.1145/44498.44500`.

25 Per-Åke Larson and Ajay Kajla. File organization: Implementation of a method guaranteeing retrieval in one access. *Commun. ACM*, 27(7):670–677, 1984. `doi:10.1145/358105.358193`.

26 Per-Åke Larson and M. V. Ramakrishna. External perfect hashing. In *SIGMOD Conference*, pages 190–200. ACM Press, 1985. `doi:10.1145/318898.318916`.

27 Hans-Peter Lehmann. *Fast and Space-Efficient Perfect Hashing*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2024. `doi:10.5445/IR/1000176432`.

28   Hans-Peter Lehmann, Thomas Mueller, Rasmus Pagh, Giulio Ermanno Pibiri, Peter Sanders, Sebastiano Vigna, and Stefan Walzer. Modern minimal perfect hashing: A survey. *CoRR*, abs/2506.06536, 2025. `doi:10.48550/arXiv.2506.06536`.

29   Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. SicHash – small irregular cuckoo tables for perfect hashing. In *ALENEX*, pages 176–189. SIAM, 2023. `doi:10.1137/1.9781611977561.CH15`.

30   Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. ShockHash: Near optimal-space minimal perfect hashing beyond brute-force. *CoRR*, abs/2310.14959, 2024. `doi:10.48550/arXiv.2310.14959`.

31   Hans-Peter Lehmann, Peter Sanders, Stefan Walzer, and Jonatan Ziegler. Combined search and encoding for seeds, with an application to minimal perfect hashing. In *33nd Annual European Symposium on Algorithms (ESA 2025)*, Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. `doi:10.4230/LIPIcs.ESA.2025.110`.

32   Harry G. Mairson. The program complexity of searching a table. In *FOCS*, pages 40–47. IEEE Computer Society, 1983. `doi:10.1109/SFCS.1983.76`.

33   Harry G Mairson. The effect of table expansion on the program complexity of perfect hash functions. *BIT Numerical Mathematics*, 32:430–440, 1992. `doi:10.1007/BF02074879`.

34   Kurt Mehlhorn. Data structures and algorithms, vol. 1: Sorting and searching. *EATCS Monographs on Theoretical Computer Science, Springer-Verlag*, 1984.

35   Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and perfect hashing using fingerprinting. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 138–149. Springer, 2014. `doi:10.1007/978-3-319-07959-2_12`.

36   Rasmus Pagh. Basic external memory data structures. In *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 14–35. Springer, 2002. `doi:10.1007/3-540-36574-5_2`.

37   Giulio Ermanno Pibiri and Roberto Trani. PTHash: Revisiting FCH minimal perfect hashing. In *SIGIR*, pages 1339–1348. ACM, 2021. `doi:10.1145/3404835.3462849`.

38   Robert F. Rice. Some practical universal noiseless coding techniques. *Jet Propulsion Laboratory, JPL Publication*, 1979.