Karlsruhe Institute of Technology

Engineering k-perfect Hashing

Bachelor's Thesis of

Sebastian Kirmayer

at the KIT Department of Informatics
Institute of Theoretical Informatics
Algorithm Engineering

First examiner:	Prof. Dr. Peter Sanders
Second examiner:	T.T.-Prof. Dr. Thomas Bläsius
Advisors:	M.Sc. Hans-Peter Lehmann
	Dr. Stefan Walzer
	M.Sc. Stefan Hermann

1. June 2024 – 30. September 2024

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 30.09.2024

. . . . . .

Sebastian Kirmayer

Abstract

A minimal k-perfect hash function on a key set S is a function $h : S \to \{0, \ldots, \lceil |S|/k \rceil - 1\}$ which maps no more than k keys to the same output.

We design and implement four schemes for constructing minimal k-perfect hash functions: *RecSplit* and *Hash and Displace* are based on existing schemes for minimal perfect hash functions, and perform especially well for small k. *Threshold-based Bumping* is an improvement and generalization of an existing scheme for minimal k-perfect hashing which provides especially fast queries on a wide range of values of k. *PaCHash* is based on an external hashing technique and achieves very fast construction time for a wide range of values of k.

Zusammenfassung

Eine minimale k-perfekte Hashfunktion auf einer Schlüsselmenge S ist eine Funktion $h : S \to \{0, \ldots, \lceil |S|/k \rceil - 1\}$, die nicht mehr als k Schlüssel auf denselben Wert abbildet.

Wir entwerfen und implementieren vier Systeme zur Konstruktion minimaler k-perfekter Hashfunktionen: *RecSplit* und *Hash and Displace* basieren auf existierenden Techniken für minimal perfekte Hashfunktionen, und funktionieren besonders gut für kleines k. *Threshold-based Bumping* ist eine Verbesserung und Verallgemeinerung einer existierenden Technik für minimales k-perfektes Hashing, die auf einem großen Bereich der Werte von k besonders schnelle Anfragen bietet. *PaCHash* basiert auf einer Technik für externes Hashing und erreicht sehr schnelle Konstruktionszeiten für große wie kleine k.

Contents

1 Introduction **5**

2 Preliminaries **6**

3 Recursive Splitting **8**
3.1 Existing 1-perfect Hash Function 8
3.2 Our k-perfect Generalization 9
3.2.1 Complete Leaves 9
3.2.2 Incomplete Leaves 10
3.2.3 Implementation 11

4 Hash and Displace **12**
4.1 Existing 1-perfect Hash Functions 12
4.2 Our k-perfect Generalization 13
4.2.1 Implementation 13

5 Threshold-based Bumping **14**
5.1 Existing k-perfect Hash Function 14
5.2 Our Contribution 14
5.2.1 Choice of Available Thresholds 15
5.2.2 Implementation 16

6 PaCHash **17**
6.1 Existing External Hashing Technique 17
6.2 Our k-perfect Hash Function 17
6.2.1 Implementation 18

7 Evaluation **19**
7.1 Results 19

8 Conclusion **23**

Chapter 1

Introduction

In some universe U, consider a set $S \subset U$ of keys, $|S| =: n$. A *perfect hash function* (PHF) is a static datastructure which supports an operation $h : U \to [m]$ (where $[m]$ denotes the set $\{0, \ldots, m-1\}$ of non-negative integers less than m), which maps no two distinct keys from S to the same value — it is injective when restricted to S. No requirements are placed on h for values outside S. A PHF is called *minimal* (MPHF) if $m = n$, i.e. if the set of hash values is as small as possible. A *k-perfect hash function* (k-PHF) relaxes the requirement on h to allow up to k keys from S to be mapped to the same value. Accordingly, a k-PHF is minimal (k-MPHF) if $m = \lceil n/k \rceil$. We focus primarily on minimal k-perfect hash functions.

In the context of a hash table, a PHF avoids the need for handling collisions. When the hash table is large and stored in external memory, the primary benefit of avoiding collisions is that only one block needs to be loaded from external memory. Since a block usually stores multiple entries, a k-PHF is sufficient for this purpose. Using a k-PHF may lower the amount of internal memory required for storing the hash function and the time for a hash function evaluation. k-PHFs may also be useful for the construction of PHFs: For example, the RecSplit technique [EMV] can be seen as a k-PHF combined with a PHF on the bins (leaves) of size k. ShockHash-Flat [LSW24] more explicitly uses a k-perfect hash function.

In contrast to perfect hashing, there has been little prior work on k-perfect hashing. We adapt several known techniques to produce k-MPHFs, and implement and evaluate these in regard to construction time, space usage, and query time.

We discuss four k-perfect hash functions. Chapters 3 and 4 discuss *RecSplit* and *Hash and Displace*, which are based on existing 1-perfect hash functions. Chapter 5 describes *Threshold-based Bumping*, where we improve an existing k-perfect hash function. Chapter 6 covers *PaCHash*, which is based on an external hashing technique. For each scheme, we describe the preexisting technique, our modifications, and some implementation details. In Chapter 7, we evaluate all four techniques.

Threshold-based bumping provides fast construction and very fast queries, while PaCHash provides very fast construction and fast queries, both at reasonable sizes, across a wide range of values of k. RecSplit provides small k-PHFs, especially for small k, in exchange for slower construction and query. For small k, Hash and Displace provides even faster queries than threshold-based bumping for small k, and achieves a smaller size at the cost of slower construction.

Chapter 2

Preliminaries

Lower Bound. For 1-MPHFs, there is an information-theoretic lower bound of $n \log_2(e) \approx 1.44n$ bits [Meh82]. For k-MPHFs, this generalizes to $n \cdot (\log_2(e) - \log_2(k^k/k!)/k) = n \cdot (\log_2(e) - \log_2(k) + \log_2(k!)/k)$ bits [BBD09]. See fig. 2.1 for selected values of k. Stirling's approximation provides a simpler, but slightly worse, lower bound of $\log_2(2\pi k)/2k$ bits per key [KLS].

Retrieval Data Structures. Given a set of key-value pairs, a *retrieval data structure* allows quickly querying the value associated with some key. Querying a key not in the set produces an unspecified result. We use Bumped Ribbon Retrieval (BuRR) [Dil+22], a retrieval data structure which is only 0.1–0.5% larger than the values (no keys are stored). We are primarily interested in the case where all values are 1 bit, where n keys require slightly over n bits.

Rank and Select. Given a bit vector, we consider two operations: $\text{rank}_1(i)$ counts the 1s in the first i bits, and $\text{select}_1(i)$ returns the index of the ith 1. Analogously, we have rank_0 and select_0 operating on 0s instead. By storing a sublinear amount of additional information (i.e. $n + o(n)$ bits in total), we can answer each type of query in constant time. We use an implementation from Sux[1] [Vig08].

Compact Encoding. Given a sequence of n integers in the range $[0..2^L)$, we can encode it in nL bits by concatenating the binary representations of the integers. This allows constant time access to any integer in the sequene.

Elias-Fano Encoding. Consider a weakly monotonically increasing sequence $a_0, \ldots, a_{n-1}$ of non-negative integers. To store this sequence, we split each element into two parts: The low $L := \lceil \log_2(a_{n-1}/n) \rceil$ bits and the remaining high bits. Thus, we now have two sequences l_i and

[1] https://sux.di.unimi.it/

k	Lower bound [bits/key]
1	1.44
10	0.299
100	0.0464
1000	0.00630

Figure 2.1: Lower bounds for some values of k

h_i. The l_i all have L bits, so we will store them in compact encoding. The h_i are still weakly monotonically increasing. Thus, $h_i + i$ is strictly monotonically increasing. We now consider a bit vector where the bits at indices $h_i + i$ are 1s, and all other bits are 0s. On this bit vector, we have $\text{select}_1(i) = h_i + i$ and thus $h_i = \text{select}_1(i) - i$, allowing constant time queries. This requires $n(2 + \log_2(a_{n-1}/n)) + o(n)$ bits [Fan71; Eli74; Vig13].

The bit vector can also be viewed as a program operating on an accumulator, initialized to 0: 0 increments the accumulator, and 1 outputs its current value. This program then outputs all h_i in sequence, since there are i 1s and h_i 0s before the 1 at $h_i + i$.

Golomb and Rice Codes. Consider a random variable X which follows a geometric distribution with success probability p. We would like to encode its value in such a way that its expected size is small, and the code is prefix-free. Let m be the integer closest to $-\log(2)/log(1-p)$, i.e. $(1-p)^m \approx 1/2$. We now divide X into to parts $h := \lfloor X/m \rfloor$ and $l := X \bmod m$. First, we put h 0 bits, followed by a single 1 bit. Then, we encode l in $\lfloor \log_2(m) \rfloor$ or $\lceil \log_2(m) \rceil$ bits — smaller values of l are encoded in the smaller number of bits. This is a *Golomb Encoding* of X [Gol66].

If we choose m to be a nearby power of two instead, the encoding of l becomes simpler: It is now fixed-length. This is a *Rice Encoding* (or Golomb-Rice Encoding) of X [Ric79].

Chapter 3

Recursive Splitting

3.1 Existing 1-perfect Hash Function

In RecSplit [EMV], we first hash the keys and map them into roughly equal-sized buckets, which we then process independently.

The rest of the technique is based on *splitting* operations: We split a key set S into parts of fixed sizes $a_0 + \cdots + a_{k-1} = |S|$. Let $b_i := \sum_{j=0}^{i-1} a_j$. Then a hash function $h : S \to [0..|S|)$ splits S if it maps exactly a_i keys to the range $[b_i..b_{i+1})$. We find a seed for such a hash function by brute force.

On each bucket, we recursively split the set of keys into smaller parts, producing a splitting tree. The splitting strategy, i.e. the number and sizes of the parts, depends only on the size of the key set for each bucket, and need not be stored explicitly. Call a subtree *complete* if it is either a leaf or all of its children are the roots of complete and isomorphic subtrees. Then for each split node, all children except possibly the rightmost one are the roots of complete and isomorphic subtrees. By this procedure, we split the bucket into a number of leaves, all but one of which have the same, predetermined, size. One leaf may be smaller. We then find seeds for MPHFs on the leaves by brute force.

We encode each seed using Rice codes and split it into the fixed-length and variable-length components. We concatenate the fixed-length and variable-length components, respectively, of all the nodes in preorder. Then, we concatenate these two sequences, first the fixed-length component, then the variable-length component. Finally, we concatenate the bit sequences thus produced for all buckets. In addition, we store the number of keys before each bucket and its starting offset in the bit sequence in a single modified Elias-Fano structure which makes use of the correlation between the two values.

During queries, after finding the appropriate bucket, we walk the splitting tree from the root to a leaf. The length of the fixed-length part for each key set size is precomputed, making it possible to skip ahead to the start of the variable-length components. While walking down the tree, it may be necessary to skip left subtrees. Once again, in the fixed-length part, we skip the precomputed length. For the variable-length part, we use a hardware-supported selection operation to skip the appropriate number of 1-bits, since each variable-length part contains exactly one 1-bit. Although this selection operation is linear in the number of skipped bits, the query remains expected constant time, since each bucket only requires a constant expected number of bits.

Bucket index i	Number of keys $a_{i+1} - a_i$	Cumulative keys a_i	First bin $\lfloor a_i/k \rfloor$	Last bin
0	5	0	0	0
1	8	5	1	2
2	6	13	4	5

Figure 3.1: Example buckets for $k = 3$.

3.2 Our k-perfect Generalization

We now describe our k-perfect generalization of RecSplit. Recall that the size of the leaves of the splitting tree is an input parameter. We now set this to k. All leaves except one per bucket now have size k and can become the bins of a k-perfect hash function. There are, however, two difficulties.

First, note that RecSplit uses the leaf size as a tuning parameter, choosing the *fanout*, the number of parts each node further up the tree is split into, such that the construction of each level of the tree requires roughly the same amount of time. Since the leaf size is now k, it can no longer be varied for tuning purposes. Instead, we tune the fanout of the lowest split, using it to determine the fanout of the levels above.

The other problem is that the partition sizes may not be multiples of k. In this case, the rightmost leaf of a splitting tree becomes smaller than k, resulting in too small bins. We consider several ways of fixing this. In all cases, the complete leaves become bins of the final k-MPHF, and the incomplete leaves are somehow combined to fill up the remaining bins. We discuss first how we deal with the complete leaves, then with the incomplete leaves.

3.2.1 Complete Leaves

First, there are several options for assigning hash values to the complete leaves. RecSplit stores for each bucket the number of keys in earlier buckets. Let a_i be this cumulative number for bucket i. This allows us to determine both the number of keys in bucket i as $a_{i+1} - a_i$ and the index of the first bin in bucket i as a_i in constant time in the case of $k = 1$, but doesn't easily generalize for larger k. We discuss three approaches which can be used instead.

INTERSPERSE. One option is to place the first bin of bucket i at $\lfloor a_i/k \rfloor$. This leaves empty bins between buckets. For example, consider the example in fig. 3.1 with $k = 3$. There is an empty bin at index 3. Such bins occur whenever the cumulative number of keys in incomplete leaves passes a multiple of k — here, the 2 keys each in the incomplete leaves from the first two buckets add to 4, passing 3. Since there are less than k keys in an incomplete leaf (otherwise there would be another complete leaf), passing a multiple of k coincides with decreases in the remainder modulo k. Further, since the number of keys in complete leaves is a multiple of k, we can instead consider $a_i \bmod k$. Thus, we can find out if there is an empty bin between two adjacent buckets in constant time. This will be useful later for dealing with incomplete leaves.

RANK. Instead, we can correct for these gaps, moving each bucket's bin range down, and shifting the gaps to the end. A simple way to do this is to construct a bit vector which stores, for each bucket, whether an empty bin would be placed after it. Then a rank query on this bit vector counts gaps before a bucket, and we can compute the first bin of bucket i as $\lfloor a_i/k \rfloor - \text{rank}(i)$.

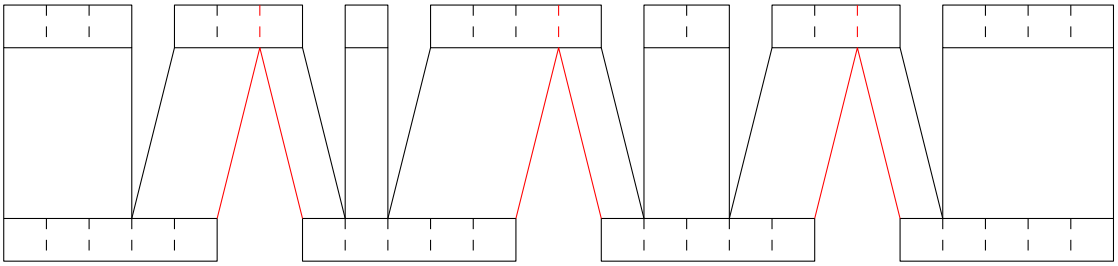

Figure 3.2: Lining up the incomplete leaves with the desired bins, with some leaves being split between bins.

BINS-MOD. Another way to obtain the same bin mapping as in the RANK approach avoids storing a_i, the cumulative number of keys in the buckets. Instead, we store the cumulative number of *bins* b_i, making it trivial to determine the first bin of the bucket.

We can also determine the number of complete leaves in bucket i as $b_{i+1} - b_i$. To determine the number of keys, we store in addition the size of the incomplete leaf r_i, or 0 if there is no incomplete leaf. This is simply the remainder of the number of keys modulo k. Now, we can determine the number of keys as $k \cdot (b_{i+1} - b_i) + r_i$. The r_i are in the range $[0..k)$ and can be stored in a straightforward compact encoding.

3.2.2 Incomplete Leaves

Now, we need to insert the incomplete leaves into the gaps left by the complete leaves. We consider two options.

RECURSE. One option is to bump all keys in the incomplete leaves into a secondary k-MPHF, e.g. a recursive RecSplit structure.

If we use the RANK or BINS-MOD methods for the complete leaves, we then simply add some offset to the hash value returned by the recursive k-MPHF.

For the INTERSPERSE method, we construct the same bit vector as for the RANK method. Then we can use a select query on this bit vector to determine the index of the bucket after which we insert the bin. We then insert this bin just before the first bin of the subsequent bucket.

An advantage of RECURSE is its conceptual simplicity and ease of implementation. The main disadvantage is that it "throws away" structure: We have already divided the keys from incomplete leaves into a sequence of chunks smaller than k, but the secondary k-MPHF starts over with an unstructured set of keys. This results in larger construction time and size.

SPLIT. To avoid discarding structure, we "line up" these chunks smaller than k against the desired sequence of bins of size k. Some chunks end up entirely inside one bin, others have parts in two adjacent bins. For the second type, we perform another splitting on the leaf to divide the keys into these two bins. Figure 3.2 shows an example of this process.

We can tell if a split is required for a bucket's incomplete leaf by looking at $a_i \bmod k$ and $a_{i+1} \bmod k$ again: If there is a decrease between these, then the incomplete leaf completes a bin. If we also have $a_{i+1} \bmod k \neq 0$, then the leaf has parts in both the completed bin and the next bin, and a split is necessary. The size of the parts of the split is then easily computed: We have $k - a_i \bmod k$ keys in the left part and $a_{i+1} \bmod k$ keys in the right part.

Since we need these a_i, BINS-MOD cannot be used here.

For RANK, we use a rank query on the bit vector to determine the number of previously-completed extra bins, and add one if the current key was split to the right. The result is then added to the number of normal bins to determine the hash value.

For INTERSPERSE, we first move ahead a bucket if we are in the right part of the split. Then we search for the next bucket with an empty bin after it, and return the index of this empty bin. Since the distribution of the sizes of incomplete leaves approaches a fixed distribution for large n, the expected number of buckets we need to search is bounded by a constant. We can use a bit vector as for RANK to speed up this search using SWAR (SIMD within a register), processing 64 buckets at once.

3.2.3 Implementation

Our implementation is based on the original RecSplit implementation from Sux[1]. It performs the necessary precomputations — the number of nodes and total length of the fixed-length part for every subtree size — at compile time, using `constexpr`.

The strategy for assigning hash values to the leaves is specified with an enum template parameter. The main RecSplit data structure contains another structure which is specialized on this parameter, containing the data structures required in each case. The implementation uses the ad-hoc `if constexpr` construct to swap out parts of the algorithm depending on the strategy.

The SPLIT strategy requires optionally storing the seed for the final split. When we need to extract this seed, we have already skipped all of the seeds for the current splitting tree. Thus, we simply store this extra seed at this point, at the very end of the bucket's seeds. Since we never need to skip this seed, we can use a full Golomb code, rather than Golomb-Rice.[2] Thus, for each bucket we store first the fixed-length parts of the splitting tree's seeds, then the variable-length parts, and last optionally the final split's seed.

[1] https://sux.di.unimi.it/

[2] This saves, in expectation, at most half a bit per block, which is hardly relevant for large block sizes.

Chapter 4

Hash and Displace

4.1 Existing 1-perfect Hash Functions

Hash and Displace is based on the idea of incrementally adding keys to a PHF by finding a hash function for the new keys which doesn't collide with the existing PHF.

We first hash the keys and map them into buckets B_i. Then, we assign hash functions to the buckets in order of decreasing size as follows: For each bucket B_i, we find a hash functions by brute force such that, together with the already mapped keys from previous buckets, no two keys are mapped to the same hash values. We then store these hash functions. At query time, after determining the key's bucket, we use the hash function for this bucket to compute the hash value. The distribution of bucket sizes and the encoding of hash functions vary [FCH92].

PTHash [PT21] maps an expected 60% of the keys into 30% of the buckets by first partitioning the key set into two parts which contain respectively, an expected 60% and 40% of keys. The keys in the first set are mapped to the first 30% of buckets, and the keys in the second set are mapped to the remaining 70% of buckets. This follows the intuition that the first buckets are easy to place, since there is nothing to collide with yet, while the last buckets are very difficult to place, since almost all bins are already filled. Thus, earlier buckets can be larger than later buckets without being more difficult.

For each bucket, hash functions $(h(\cdot, s) \oplus h(p_i, s)) \bmod n$ are considered, where p_i is the *pilot* for bucket i, chosen as small as possible. The p_i are stored using one of several encodings, such as a dictionary based encoding, Elias-Fano coding or Simple Dense Coding [FN09].

PHOBIC [Her+24] refines the bucket distribution, producing a close-to-optimal distribution. One observes that the bucket distribution can be represented by a *bucket assignment function* $\gamma : [0, 1] \to [0, 1]$ which is increasing and smooth on $(0, 1)$. A key with hash $x \in (0, 1]$ is then assigned to the bucket at index $\lceil \gamma(x) \cdot B \rceil$, where B is the number of buckets. It is then possible to find a bucket assignment function which results in good construction times for many values of B and n, without depending on B and n.

In addition, PHOBIC hashes the keys into a number of *partitions* and finds separate PHFs on each partition. It then observes that the an bucket at the same index across partitions has a similar difficulty, making it more space-efficient to encode these buckets at the same index together.

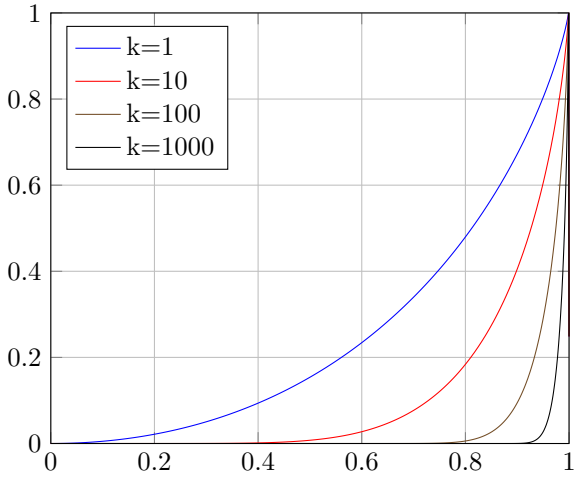

Figure 4.1: Bucket distribution functions for some values of k.

4.2 Our k-perfect Generalization

This technique is easily made k-perfect by allowing up to k keys mapped to the same bin. We do not implement PHOBIC's partitioning step, since it cannot easily be transferred to k-perfect hashing, since the partition sizes may not be multiples of k.

The bucket distribution function from [Her+24] is only optimal for $k = 1$. However, several important insights remain true for any k. In particular, the buckets should still be processed in decreasing order of size, and there is still a bucket distribution function which is good for many choices of B and n.

Furthermore, consider the expected seed values σ_i for each bin. Note that for bin i, the probability of success for a random seed is $p_i = 1/\sigma_i$. Then $\prod_i p_i = 1/\prod_i \sigma_i$ is the probability of a random combination of seeds producing a k-PHF. Since this is a constant, $\prod_i \sigma_i$ is fixed. Observe that $\sum_i \sigma_i$ represents the total number of seeds tried, and is thus an approximation of construction time. To minimise the sum, the σ_i should be equal. Note that this is only an approximation of construction time, since the time to try a seed is not constant, but depends on the number of keys in the bucket [Her+24, appendix A.4]. We numerically approximate an optimal bucket distribution function. As can be seen in fig. 4.1, the function becomes more extreme for larger values of k, putting a larger fraction of the keys in earlier buckets.

The seeds are stored in either Compact or Rice Encoding.

4.2.1 Implementation

Our implementation is generic over the bucket distribution function (Uniform or Optimal) and the encoding of the seeds (Compact or Rice). The optimal bucket distribution function requires significant precomputation; we separate this from the main construction and share the precomputation results between several instances of the structure. The precomputation is too expensive to use compile-time evaluation facilities.

We also support a non-minimal k-PHF, where we specify a load factor $\lambda \in (0..1]$. This maps n keys to $\lceil n/\lambda k \rceil$ buckets. The necessary modification of the bucket distribution function is described in [Her+24, appendix B]. This is primarily for use in Threshold-based Bumping as an overflow 1-PHF (see chapter 5).

Chapter 5

Threshold-based Bumping

5.1 Existing k-perfect Hash Function

Threshold-based bumping, also known as bumped k-perfect hashing [LSW24, section 7.2], is a k-perfect hashing technique. We first hash the keys into fewer than n/k buckets, causing most buckets to fill up or overflow. We then assign a fingerprint to each key, and select a threshold for each bucket such that at most k keys in the bucket have fingerprints below the threshold. We bump the overflowing keys and hash them into the remaining buckets. Once again, we assign fingerprints and select thresholds. We sort any keys which still overflow into the non-full buckets using another MPHF and Elias-Fano coding of the non-full bucket positions.

5.2 Our Contribution

Since this is already a k-PHF, we can use these ideas unchanged. We discuss some improvements and generalizations of the scheme.

In the existing scheme, fingerprints are small, $\log_2(k)$-bit, numbers, and thresholds are also such numbers. We use full 64-bit numbers for the fingerprints instead, and for the thresholds we store an index into a predetermined set of available thresholds $T_0 < T_1 < \cdots < T_m$. If we choose these uniformly, i.e. $T_i = 2^{64}i/(m+1)$, this is equivalent to the old way, but we will see in section 5.2.1 that we can do better. In the simple case, the number of available thresholds is a power of two, and we can store the thresholds in compact encoding. We also support packing pairs of thresholds into an odd number of bits b, resulting in $\lfloor 2^{b/2} \rfloor$ thresholds. This comes at a performance cost, since accessing a threshold now requires a division or modulo.

When there is no threshold available between two fingerprints, we might have to bump more keys than necessary: For instance, if $k = 5$, and we have 4 keys with fingerprints smaller than T_i, but two keys with fingerprint between T_i and T_{i+1}, by choosing a threshold we can either have 4 or 6 keys remaining in the bucket. We cannot allow 6, so we will bump both keys, and the bucket is not filled. We choose the available thresholds T_i to minimize the probability of this occuring (see section 5.2.1). If it does occur, some keys are between these two successive thresholds. We can use a retrieval data structure on these keys to avoid unnecessary bumping. Choosing available thresholds as before is still important, as it minimizes the number of keys in the retrieval data structure.

For overloading, we consider the *overload factor* λ. We then choose a subset of buckets such

that the average bucket receives λk keys.[1] We generalize from two levels of bumping to an arbitrary number. At each level, we keep the overload factor fixed: If n' keys remain, we hash them into n'/λ buckets. Since at every level, some fraction of buckets receives fewer than k keys, at some point, we use all remaining buckets. This gives us a natural stopping point for the procedure, with the expected number of levels depending on λ.

For the overflowing keys from the last level, we use PHOBIC as a perfect hash function. To improve construction time, we use non-minimal PHOBIC, requiring a larger Elias-Fano structure instead.

5.2.1 Choice of Available Thresholds

We now describe how we choose the set of available thresholds. First, note that we need to set $T_0 := 0$ to ensure that there is always some threshold we can choose, even if it bumps many unnecessary keys.

We call a threshold T_i *perfect* if there are exactly k keys with fingerprints below it. An optimal set of thresholds would be one where the expected number of unnecessarily bumped keys is minimized. Instead, we find a set of thresholds which maximizes the probability that there is some perfect threshold, i.e. that zero keys are unnecessarily bumped. We expect this to be a good approximation of the optimal set, and it is easier to compute.

We interpret the fingerprints, which are 64-bit integers, as reals on the interval $[0, \lambda k)$, where λ is the overload factor. Then the number of keys in a subinterval of length μ is $\text{Pois}(\mu)$-distributed. In particular, the probability that there are exactly l keys in this subinterval is

$$p_\mu(l) := \frac{\mu^l e^{-\mu}}{l!}$$

Thus, the probability that some threshold T_i is perfect is $p_{T_i}(k)$.

Now, let us consider the probability that T_i is the *first* perfect threshold. Clearly, if two thresholds are perfect, then there cannot be any keys between them. Conversely, when there are no keys between two thresholds, either both are perfect, or neither is. Thus, T_i is the first perfect threshold if and only if T_i is perfect and there is at least one key between T_{i-1} and T_i. Thus, we obtain a probability of

$$\begin{aligned}
&\mathbb{P}(T_i \text{ is perfect} \wedge \text{at least one key between } T_{i-1} \text{ and } T_i) \\
&= \mathbb{P}(T_i \text{ is perfect}) - \mathbb{P}(T_i \text{ is perfect} \wedge \text{no keys between } T_{i-1} \text{ and } T_i) \\
&= p_{T_i}(k) - \mathbb{P}(T_{i-1} \text{ is perfect} \wedge \text{no keys between } T_{i-1} \text{ and } T_i) \\
&= p_{T_i}(k) - \mathbb{P}(T_{i-1} \text{ is perfect})\mathbb{P}(\text{no keys between } T_{i-1} \text{ and } T_i) \qquad (*) \\
&= p_{T_i}(k) - p_{T_{i-1}}(k)p_{T_i - T_{i-1}}(0) \\
&= \frac{1}{k!}(T_i^k e^{-T_i} - T_{i-1}^k e^{-T_{i-1}} e^{-(T_i - T_{i-1})}) \\
&= \frac{e^{-T_i}}{k!}(T_i^k - T_{i-1}^k)
\end{aligned}$$

where $(*)$ follows by independence.

Since no two thresholds can be the first perfect threshold simultaneously, the probability P that there is at least one perfect threshold is simply

$$P = \frac{1}{k!}\sum_{i=1}^{m} e^{-T_i}(T_i^k - T_{i-1}^k)$$

[1] This is different from the parameter γ in [LSW24, section 7.2]. We have $\lambda = 1/\gamma$.

We would like to maximize P. If P is maximal, then its derivative with respect to each of $T_1, \ldots, T_m$ must be zero — otherwise, we could slightly move T_i to increase P.

For $1 \leq i < m$ we have:

$$\begin{aligned} 0 \stackrel{!}{=} \frac{\mathrm{d}P}{\mathrm{d}T_i} &= \frac{1}{k!} \sum_{j=1}^{m} \frac{\mathrm{d}}{\mathrm{d}T_j} e^{-T_j} (T_j^k - T_{j-1}^k) \\ &= \frac{1}{k!} \left(\frac{\mathrm{d}}{\mathrm{d}T_i} e^{-T_i} (T_i^k - T_{i-1}^k) + \frac{\mathrm{d}}{\mathrm{d}T_i} e^{-T_{i+1}} (T_{i+1}^k - T_i^k) \right) \\ &= \frac{1}{k!} \left(e^{-T_i} k T_i^{k-1} - e^{-T_i} (T_i^k - T_{i-1}^k) - e^{-T_{i+1}} k T_i^{k-1} \right) \end{aligned}$$

Solving for T_{i+1} gives

$$T_{i+1} = T_i - \ln \left(1 - \frac{T_i}{k} + \frac{T_{i-1}}{k} \left(\frac{T_{i-1}}{T_i} \right)^{k-1} \right)$$

Given a guess for T_1, we can now compute $T_2, \ldots, T_m$. We can then look at the next threshold T_{m+1}: It should be "exactly infinity", i.e. the expression under the logarithm should be zero. This will make the derivative of P w.r.t. T_m zero as well.

If we increase T_1, all T_i increase as well. If T_{m+1} is finite, we thus have to increase T_1, but if T_{m+1} is "beyond infinity", i.e. the term under the logarithm is negative, we have to decrease T_1. This allows us to approximate the correct guess for T_1 by binary search.

Our calculation above is incorrect in that it assumes fingerprints follow the same distribution beyond λk, even though they are limited to the interval $[0, \lambda k)$. To fix this, we note that if $T_m > \lambda k$, we should also decrease T_1.

5.2.2 Implementation

Our implementation is parameterized by the overload factor λ, the size of thresholds, the overflow PHF and its parameters, and whether we use a retrieval structure to avoid unnecessary bumping. The size of the thresholds is specified in *half-bits*: An even number of half-bits produces compact encoding, while an odd number packs the thresholds into pairs which are then stored in compact encoding.

We precompute the set of available thresholds at compile time, using the binary search described in section 5.2.1. The floating-point thresholds this produces are then already converted to 64-bit integer thresholds at compile time.

We do the bucket mapping and sorting by fingerprints in a single step: We simply sort by (bucket index $\times\ 2^{64}$ + fingerprint). Unfortunately, IPS^{2}Ra, used for this sorting step, fails when used for 128-bit keys. The problem is that the TLX library's count-leading-zeroes and count-trailing-zeroes operations, used by IPS^{2}Ra, do not support 128-bit arguments. In highly unorthodox fashion, we add the necessary specializations to the `tlx` namespace in our own source files.

Chapter 6

PaCHash

6.1 Existing External Hashing Technique

PaCHash [KLS] is an external hashing technique: A hash table is stored in external memory, with a small amount of internal memory being used to reduce the number of slow external memory accesses. The objects in the hash table may have variable-length. We first map them to a number of bins. These bins are then laid out contiguously in external memory, including across block boundaries. For each block, we store the index of the first bin which is, at least partially, contained within the block. In the special case where an empty bin aligns with a block boundary, we instead store this empty bin for the block following the boundary.

For lookup, we map the key to its bin, and determine the last block whose stored bin is less than the key's bin and the first block whose stored bin is greater than the key's bin. This provides the left-inclusive, right-exclusive range of blocks where the key's object is stored. We load these blocks from external memory, where additional metadata stored within each block allows us to recover the object. This may read one or two unnecessary blocks when a bin boundary aligns with a block boundary.

The sequence of bin indices is stored in Elias-Fano encoding. Recall from chapter 2 that, given the weakly monotonically increasing sequence a_i, this encoding splits each item into its high bits h_i and low bits l_i. To find the first a_i which is at least x, we first consider the high bits h of x. Using the program view on the Elias-Fano bit vector, $\text{select}_0(h)$ is then the index of the instruction which increases the accumulator beyond h, and $j := \text{select}_0(h) - h$ is the index of the first number in the sequence with $h_j > h$. From j, we then search the sequence backwards to find the answer. In expectation, only a constant number of elements needs to be considered.

6.2 Our k-perfect Hash Function

To turn this into a k-MPHF, our "objects" are exactly $1/k$ of a block long. Then, the hash value is simply the index of the block which contains the key. Of course, we don't actually store any objects.

For bins which overlap $l > 1$ blocks, we use a retrieval data structure which stores the index among these l blocks (in the range $[0..l)$). We store the bits of this index in separate entries of the retrieval data structure. Since experimentally, the average bin size should be 1 for optimal size, it is rare for the index to have more than 1 bit unless k is very small.

As a special case, PaCHash uses the index of an empty bin if the bin aligns with a block

boundary. As a generalization of this, when the start of a block is also the start of a bin, we pick the smallest bin touching the block boundary, reducing the number of entries in the retrieval data structure by about 10% from ≈ 1.63 to ≈ 1.47 entries per block. This generalization can also be applied to the original PaCHash, reducing the average number of blocks loaded to access an object.

6.2.1 Implementation

Our Elias-Fano implementation, instead of allowing access to individual elements of the sequence, can find the range of indices which have some value. For values which do not occur, this range is empty, but it sits between the indices with lower values and those with higher values. Since the Elias-Fano sequence contains the index of the first bin at least partially contained in each block, this range for the bin index, together with one extra bucket to the left, is the range of buckets keys from this bin have been placed into. To avoid a special case for the bin with index 0, we increment all bin indices, but still store 0 for the first bucket: Thus, no bin's range includes the first bucket, and we never look to the left of it. We add a similar sentinel bucket at the end, simplifying the Elias-Fano implementation.

The construction is also quite simple: We first map the keys to the bins, producing the list of keys sorted by their bins. A first scan looks at the keys at bucket boundaries to determine the bin index stored for each bucket. A second scan then collects the retrieval data for each key.

Chapter 7

Evaluation

We implemented all algorithms in C++ 2020 with GNU extensions, and compiled them with GCC 14.2.0 using `g++ -std=gnu++20 -march=native -O2`. Our implementations assume that the keys have already been hashed to 128 bits. We assume that these hashes are uniformly distributed.

Bucket mapping. To assign keys to B buckets (or bins, in PaCHash), we consider 64 bits of the key's hash. Multiplying this by B, we get an integer in the range $[0..2^{64}B)$. Then we divide by 2^{64}, obtaining a bucket index in the range $[0..B)$. On x86-64, this can be done in a single multiplication instruction, which places the upper 64 bits of the 128-bit result into a separate register [Lem19].

To quickly partition keys to buckets, we use IPS^{2}Ra [Axt+20], a fast radix sort implementation, to sort the keys by their bucket indices. Then, each bucket's keys are contiguous in memory, and we can find the boundaries by scanning the array of keys until we find a key in a different bucket.

Compact encoding. To access a number stored in compact encoding, we need to extract a number at an arbitrary bit offset i and bit length l. We assume that $l \leq 57$. Thus, the number is part of a byte-aligned sequence of 64 bits. The first byte we need to extract is at offset $\lfloor i/8 \rfloor$. We extract 8 bytes starting at this offset to a 64-bit integer using `memcpy`, and then shift the result right by $i \bmod 8$ bits. Finally, we mask off the high bits, keeping only the l lowest bits. GCC optimizes the `memcpy` into an unaligned memory access. This procedure assumes a little-endian system.

Remixing. At various points, we need a seeded hash function. Given a 64-bit value x and a 64-bit seed s, we use $\text{remix}(x + s)$, where remix is David Stafford's Mix13 bit mixer [Sta11].

7.1 Results

We tested several configurations for each algorithm, with $k \in \{10, 100, 1000\}$, for 10^7 keys. We only consider configurations which beat every other configuration in at least one direction — construction time, query time, or size. The resulting set is shown in fig. 7.1.

The size of the hash functions is shown on the horizontal axis. The scale is logarithmic in the *overhead*, i.e. a configuration which perfectly achieves the lower bound would be infinitely

far to the left.

The upper two rows of graphs simply show two different views at the three-dimensional dataset: The first one shows construction time on the vertical axis, the second one query time. The lower two rows answer questions of the form: If we need a k-PHF which achieves some size and construction (resp. query) time, which technique should we use to optimize query (resp. construction) time? This shows under which circumstances each technique excels.

Recursive Splitting. RecSplit behaves much like its 1-perfect model: It is the smallest technique for all values of k, but it is relatively slow.

For $k = 10$ and $k = 100$ we tried two different lowest fanouts, these correspond to the two major "arcs" visible in the graph, with the larger fanout being slower to construct, but smaller. For $k = 1000$, any fanout larger than two would be quite slow. Within each arc, different bucket sizes are visible, with larger bucket sizes producing smaller but slower functions. The fanout only has a minimal impact on query time — higher fanout produces slightly shallower splitting trees. For the bucket size, the differences are much larger, causing larger differences in the depth of the splitting tree, and thus greater variation in construction time.

For large k, RecSplit's size becomes worse. On the one hand, the average fanout decreases since high-fanout splits become impossible, causing greater overhead. On the other hand, the bucket size cannot grow with k, since the first splits would become very difficult, independent of k; this also causes greater overhead.

The INTERSPERSE approach is best for dealing with complete leaves, and SPLIT is best for incomplete leaves. Adding a bit vector to find gaps quicker provides neglegible query time benefits — this is somewhat expected, since only a small fraction of keys is in incomplete leaves, and we only need to skip ahead one bucket in expectation.

For $k = 10$, the smallest RecSplit variants require about 1.2 times the lower bound in size.

Hash and Displace. The two encodings — Compact and Rice — are clearly visible in the query time graphs as two separate lines: A faster line for Compact and a slower one for Rice. In the construction time graphs, pairs with almost equal construction time show the two encodings. Within each encoding, different average bucket sizes are visible. Larger buckets are smaller, slower to construct, and — due to cache effects — slightly faster to query.

For $k = 10$, Hash and Displace with Compact Encoding is the fastest technique, and both encodings have their niches. For larger k, it is far less competitive: Despite queries becoming slightly faster — since we can use even larger buckets now — threshold-based bumping becomes even faster, and construction time also quickly surpasses threshold-based bumping. Thus, larger bucket sizes disappear off the graphs due to being dominated by other techniques. It is unclear why construction becomes so much slower, but it is likely related to the increasingly extreme bucket assignment function (see fig. 4.1). The technique only remains competitive for fairly large sizes with small query time or construction time requirements.

Threshold-based Bumping. Threshold-based bumping provides the fastest queries for large k, and remains competitive for small k due to fast construction and fairly small size. The query time graph shows two lines: The lower line uses compact encoding for the thresholds, while the upper line packs two thresholds each into an odd number of bits, requiring a division or modulo on every query.

Queries become faster for larger k because the bucket size can be much more tightly controlled: For small k, using an overload factor λ only slightly larger than 1 fails to fill almost half the buckets, while for large k, this doesn't occur. All these empty spots in the bucket then require bumping through all levels, a query in the fallback 1-PHF and an Elias-Fano access. To avoid

this, we can use a larger λ, but then a larger fraction of keys get bumped, slowing queries down as well.

The smallest variants require about 1.8-1.95 times the lower bound in size.

PaCHash. PaCHash is the technique with the fastest construction across all values of k, and achieves fairly good size and query time as well. The construction time can be further reduced, at the cost of size, by increasing the bin size. The smallest PaCHash variants require approximately 2.2 to 2.4 times the lower bound in size, for a large range of values of k.

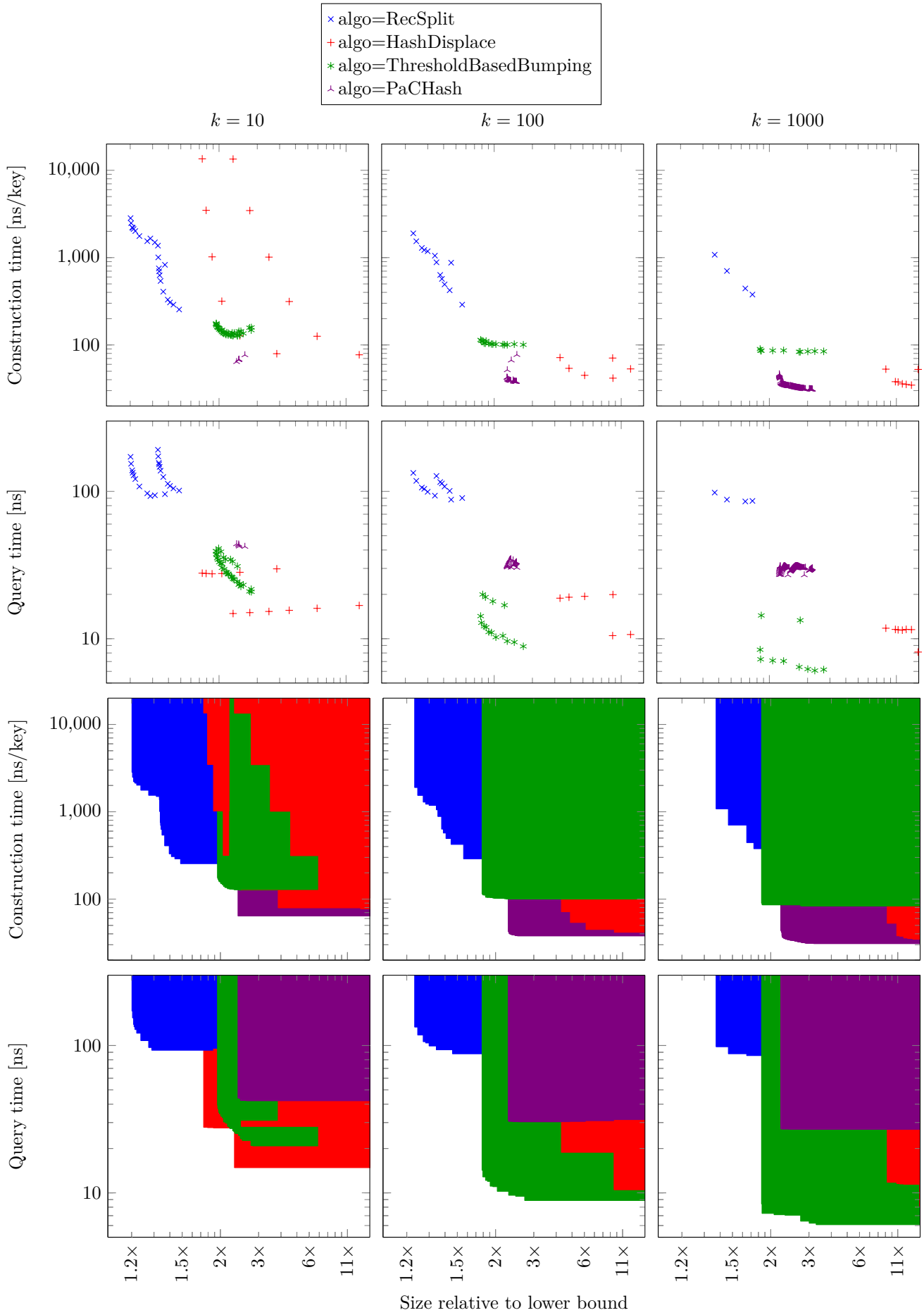

Figure 7.1: Evaluation results.

Chapter 8

Conclusion

We have provided an overview over the space of k-PHFs.

We generalize RecSplit to a k-PHF, considering several options for dealing with the mapping from leaves to hash values. We produce a small k-PHF with about 20% space overhead for small values of k.

We generalize PHOBIC to a k-PHF, adapting its bucket distribution function. This provides fast queries and, for small k, reasonable construction time and size.

We improve threshold-based bumping by encoding thresholds in an optimized way which avoids unnecessary bumping, and use a retrieval data structure to eliminate unnecessary bumping entirely. In addition, we generalize the technique to more levels. This results in a k-PHF with very fast queries, fast construction, and a size well under twice the lower bound.

We adapt PaCHash as a k-PHF, achieving very fast construction and fast queries, at approximately 2.2 to 2.4 times the lower bound in size.

Ultimately, we conclude that RecSplit is small, threshold-based bumping has fast queries, PaCHash is fast to construct and Hash and Displace is useful in some specific circumstances.

Bibliography

[Axt+20] Michael Axtmann et al. *Engineering In-place (Shared-memory) Sorting Algorithms*. Computing Research Repository (CoRR). Sept. 2020. arXiv: 2009.13569.

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. "Hash, Displace, and Compress". In: *Algorithms - ESA 2009*. Ed. by Amos Fiat and Peter Sanders. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 682–693. ISBN: 978-3-642-04128-0. DOI: 10.1007/978-3-642-04128-0_61.

[Dil+22] Peter C. Dillinger et al. "Fast Succinct Retrieval and Approximate Membership Using Ribbon". In: *20th International Symposium on Experimental Algorithms (SEA 2022)*. Ed. by Christian Schulz and Bora Uçar. Vol. 233. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 4:1–4:20. ISBN: 978-3-95977-251-8. DOI: 10.4230/LIPIcs.SEA.2022.4.

[Eli74] Peter Elias. "Efficient Storage and Retrieval by Content and Address of Static Files". In: *J. ACM* 21.2 (Apr. 1974), pp. 246–260. ISSN: 0004-5411. DOI: 10.1145/321812.321820.

[EMV] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. "RecSplit: Minimal Perfect Hashing via Recursive Splitting". In: *2020 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 175–185. DOI: 10.1137/1.9781611976007.14.

[Fan71] Robert M. Fano. *On the Number of Bits Required to Implement An Associative Memory*. Memorandum 61. MIT, Computer Structures Group, 1971.

[FCH92] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. "A faster algorithm for constructing minimal perfect hash functions". In: *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '92. Copenhagen, Denmark: Association for Computing Machinery, 1992, pp. 266–273. ISBN: 0897915232. DOI: 10.1145/133160.133209.

[FN09] Kimmo Fredriksson and Fedor Nikitin. "Simple Random Access Compression". In: *Fundam. Inf.* 92.1–2 (Jan. 2009), pp. 63–81. ISSN: 0169-2968.

[Gol66] S. Golomb. "Run-length encodings (Corresp.)" In: *IEEE Transactions on Information Theory* 12.3 (1966), pp. 399–401. DOI: 10.1109/TIT.1966.1053907.

[Her+24] Stefan Hermann et al. *PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding*. 2024. arXiv: 2404.18497 [cs.DS].

[KLS] Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. “PaCHash: Packed and Compressed Hash Tables”. In: *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 162–175. DOI: 10.1137/1.9781611977561.ch14.

[Lem19] Daniel Lemire. “Fast Random Integer Generation in an Interval”. In: *ACM Trans. Model. Comput. Simul.* 29.1 (Jan. 2019). ISSN: 1049-3301. DOI: 10.1145/3230636.

[LSW24] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. *ShockHash: Near Optimal-Space Minimal Perfect Hashing Beyond Brute-Force*. 2024. arXiv: 2310.14959 [cs.DS].

[Meh82] Kurt Mehlhorn. “On the program size of perfect and universal hash functions”. In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 1982, pp. 170–175. DOI: 10.1109/SFCS.1982.80.

[PT21] Giulio Ermanno Pibiri and Roberto Trani. “PTHash: Revisiting FCH Minimal Perfect Hashing”. In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’21. Virtual Event, Canada: Association for Computing Machinery, 2021, pp. 1339–1348. ISBN: 9781450380379. DOI: 10.1145/3404835.3462849.

[Ric79] Robert F Rice. *Some practical universal noiseless coding techniques*. Tech. rep. 1979.

[Sta11] David Stafford. *Better Bit Mixing - Improving on MurmurHash3’s 64-bit Finalizer*. Blog. Sept. 2011. URL: https://zimbry.blogspot.com/2011/09/better-bit-mixing-improving-on.html.

[Vig08] Sebastiano Vigna. “Broadword Implementation of Rank/Select Queries”. In: *Experimental Algorithms*. Ed. by Catherine C. McGeoch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 154–168. ISBN: 978-3-540-68552-4. DOI: 10.1007/978-3-540-68552-4_12.

[Vig13] Sebastiano Vigna. “Quasi-succinct indices”. In: *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*. WSDM ’13. Rome, Italy: Association for Computing Machinery, 2013, pp. 83–92. ISBN: 9781450318693. DOI: 10.1145/2433396.2433409.