The Algorithmics Column

by

Thomas Erlebach

Department of Computer Science
Durham University
Upper Mountjoy Campus, Stockton Road, Durham, DH1 3LE, UK
thomas.erlebach@durham.ac.uk

What if we tried Less Power?

Lessons from studying the power of choices in hashing-based data structures

Stefan Walzer

Abstract

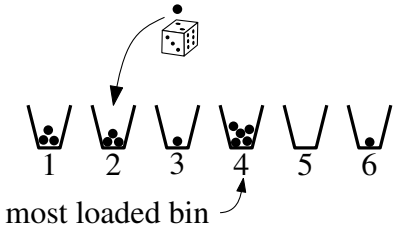
The celebrated *power of two choices* paradigm underlies *cuckoo hash tables* as follows: If you have n balls and $m = (2 + \varepsilon)n$ bins and throw each ball into a bin at random, then likely some bin will receive $\Omega(\frac{\log n}{\log \log n})$ balls. If, however, you can choose between *two* random bins for each ball, you can likely arrange for a *private bin* for each ball.

In the first part of this column, we review some related space-efficient data structures on a high level. We'll find that the additional power afforded by *more than* 2 *choices* is often outweighed by the additional costs they bring. In the second part, we present a data structure where choices play a role at coarser than per-ball granularity. In some sense, we rely on the *power of* $1 + \varepsilon$ *choices* per ball.

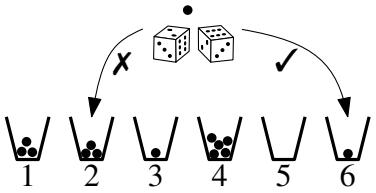
This column is a "best-of" of my dissertation and related work reviewed from a fresh perspective. I've tried to make it a pleasant read conveying intuition while being unencumbered by technical details. So allow me to be your guide through the garden of my interests, present and past. Our winding path will circle a recent construction called Bumped Ribbon Retrieval *[24], with which our tour will conclude.*

Part 1: Data Structures using the Power of Two Choices

The classical demonstration of the *power of two choices* goes as follows [5, 55]. Assume you have n balls, $m = \Theta(n)$ bins and you distribute the balls independently and uniformly at random into the bins.



Then the most loaded bin will contain $\Theta(\frac{\log n}{\log \log n})$ balls with high probability (whp)[1]. In contrast, assume you generate **two options** for each of the balls and place the balls sequentially, putting each ball into the **bin with the least load** among its two options (breaking ties arbitrarily).

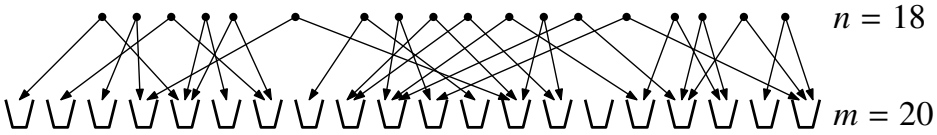


Now the **maximum load** is only $\Theta(\log \log n)$ whp, which is **exponentially less**! The described setting is known as **online load balancing**. Bins might correspond to servers and balls to jobs that have to be assigned to servers on creation. What if we generate further options for each ball? For $d \geq 2$ choices per ball, the maximum load becomes $\frac{\ln \ln n}{\ln d} + \Theta(1)$ whp, i.e. d only affects a constant factor. In other words, there is a massive difference between one and two choices and just a small difference between 2 and $d \geq 3$ choices. Hence the name power of *two* choices.

Since its discovery, the power of two choices paradigm has been influential in data structure design, which is the focus of this paper.

1.1 The Dictionary Problem & Cuckoo Hashing

Now consider **offline load balancing** where we generate the two random bins for all balls in advance and think carefully about all choices at the same time.



Which option should each ball take to avoid collisions?

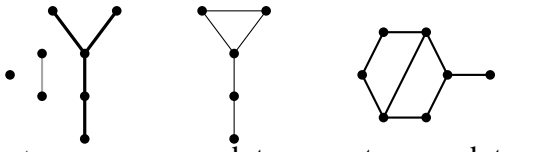
It is helpful to use a different visualisation where each bin is a vertex and each ball an edge connecting the two bins it may be placed into:

(1)

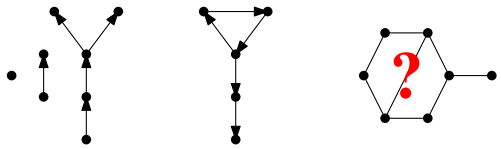
If we ignore the possibility of duplicate edges, then we get a graph with m vertices and n uniformly random edges.This is known as an *Erdős-Renyi random graph*.

[1]Defined as probability $1 - o(1)$.

In their ancient and seminal paper "on the evolution of random graphs" [28] Erdős and Renyi show that if the edge density satisfies $\frac{n}{m} < \frac{1}{2} - \varepsilon$ then whp all connected components of the graph are small **trees** (#edges = #vertices − 1) or pseudotrees (#edges = #vertices). For $\frac{n}{m} > \frac{1}{2} + \varepsilon$, on the other hand, there is whp a "**giant component**" (comprising $\Theta(n)$ vertices) that has more edges than vertices. Redrawing our example we find:



The task of placing all balls without collision corresponds to the task of assigning a **direction** to each of the edges in the graph such that every vertex has indegree at most 1. For tree and pseudotree components this is easy: For trees, we pick an arbitrary root vertex and direct each edge away from the root. For a pseudotree, which contains a single cycle, we direct the cycle in some consistent way (say "clockwise") and every other edge away from the cycle. For the remaining component(s) there is no solution by the pigeon hole principle.



The important observation here is: If we have n balls and $m = (2 + \varepsilon)n$ bins (for constant $\varepsilon > 0$) then **only trees and pseudotrees** arise whp and we can place all balls without a single collision.

Classic Cuckoo Hash Table. This observation gives us a cuckoo hash table [58], a simple data structure that stores n elements, which we call *keys*[2], using one[3] array of m memory cells and two hash functions that assign two uniformly random[4] cells to each key.

Say we wish to store the set {☆, △, □, ◌}.[5] We consider the keys' hashes and find a **collision-free placement** that puts each key into one of its two assigned cells. (If no placement exists, we restart the construction with fresh hash functions.) To

[2] In general elements could be key-value pairs, but values play no role in the following.

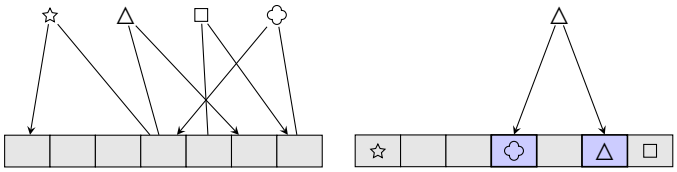
[3] Most implementations use two arrays for reasons that need not concern us here.

[4] See Digression 1.

[5] We'll use comically undersized examples throughout this text that hopefully still get the point across. Practically relevant instances would typically have thousands or millions of keys.

Digression 1: Simple Uniform Hashing Assumption (SUHA).
We assume that hash functions assign hash values to the keys independently. This *simple uniform hashing assumption* is **unrealistic** as $\tilde{\Omega}(n)$ bits of entropy would be needed for independence while popular practical hash functions like *MurmurHash* [1] or *xxhash* [12] use seeds of only $\tilde{O}(1)$ bits.
There are many standard ways of addressing this missmatch. We can try to work with weaker notions like ***k*-independence**, where any set of k keys have independent hash values but any $k+1$ hash values may be correlated [69]. A good overview on how this can help is given in [63] and highly practical 2-independent families are described in [64]. A cryptographer might instead offer some insight into how **pseudorandomness** can be indistinguishable from randomness. [3]
A well-subscribed lazy approach, that we also adopt here, is to simply use the SUHA as a **modelling assumption** and point to its excellent track record of capturing how popular hash functions behave in practice.

answer a query – say we wish to know if △ is in the set – we evaluate the hash functions on △ and search both cells for the requested key.



situation during construction situation during query

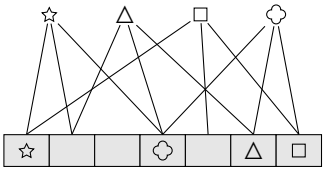
(2)

Query times are $O(1)$ in the **worst-case** (not just in expectation). A down-side is that the **load factor** is $c = \frac{n}{m} < \frac{1}{2}$, meaning twice as much memory is required compared to naively storing keys consecutively. That's less than ideal. But what if we tried more power?

Illustration from *What If?* by Randall Munroe

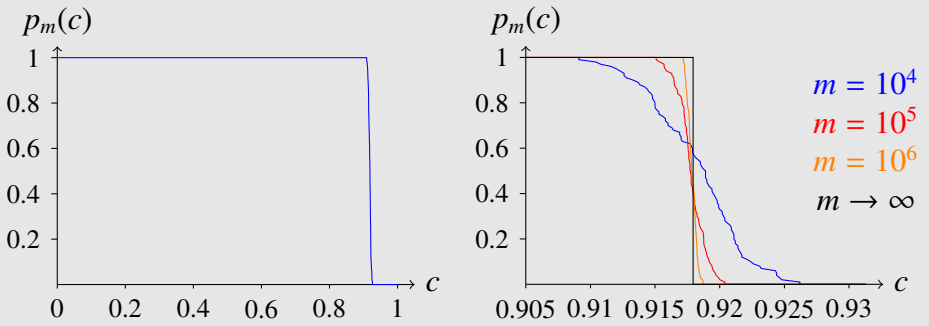
1.1.1 Higher Power: Generalisations of Cuckoo Hashing

A natural generalisation is to **assign more cells to each key** [30]. For $k > 2$ cells the graph with m vertices and n edges from (1) becomes a *hyper*graph with m vertices and n *hyper*edges of size k. This would make for a messy picture so we stick with bipartite illustrations like in (2).



(3)

Digression 2: Sharp Load Thresholds.
Consider the probability p that all n keys can be placed into a table of size m when each key is assigned $k = 3$ cells at random. In the picture on the left, we plot (experimental approximations of) $p = p_m(c)$ for varying load factor $c = \frac{n}{m} \in [0, 1]$ and fixed table size $m = 10^4$.



As expected, p decreases as c increases. What is *not* obvious is that the transition from $p_m(c) \approx 1$ to $p_m(c) \approx 0$ happens within a tiny interval. On the right, we zoom in and also plot the function for $m = 10^5$ and $m = 10^6$. This shows that the **transition becomes steeper for larger m** and starts to resemble a **step function**. In fact, there is a **sharp load threshold** $c_3^* \approx 0.9179$ such that for $c < c_3^*$ we have $\lim_{m\to\infty} p_m(c) = 1$ and for $c > c_3^*$ we have $\lim_{m\to\infty} p_m(c) = 0$.

Using $k > 2$ allows for the assumption $\frac{n}{m} < \frac{1}{2} - \varepsilon$ on the load factor to be relaxed. For $k = 3$ for instance, we find an increased *load threshold* of $c_3^* \approx 0.92$ up to which all keys can be placed whp. See Digression 2 for some background on the phenomenon of thresholds.

Load thresholds c_k^* are known for any k. They can be characterised implicitly as solutions to certain equations, but for our purposes, tabulated values will do. We include $c_1^* = 0$ to emphasise that avoiding collisions with just **one hash function requires $m = \Omega(n^2)$** due to the **birthday paradox**, which gives a load factor of $\frac{n}{n^2} \to 0$.

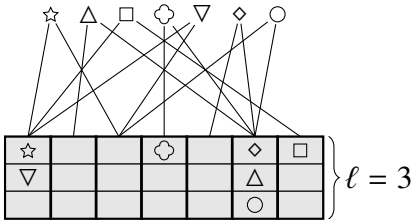
k	1	2	3	4	5	6	7
c_k^*	0	0.5	0.91794	0.97677	0.99244	0.99738	0.99906

Values as determined in [58, 17, 35, 32].

The values are of order $c_k^* = 1 - e^{-\Theta(k)}$ (this can be derived form [32]) so we can achieve load factors arbitrarily close to 1 by choosing k large enough. However, any practitioner will be quick to point out that a query operation that has to check k random memory locations is likely to incur **k cache misses**, so increasing k is

not particularly enticing.

The far more popular generalisation sticks with $k = 2$ hash functions, but each hash value identifies a *bucket* of ℓ memory cells [8, 29, 31]. Each key may then be placed into any cell in any one of its **two buckets**.



Thresholds $c^*_{2,\ell}$ for the load factor achievable this way are also known:

ℓ	1	2	3	4	5	6
$c^*_{2,\ell}$	0.5	0.89701	0.95915	0.98037	0.98955	0.99407

Values as determined in [58, 8, 29, 17, 31].

Unsurprisingly, both avenues for generalisation can be combined such that a key can be placed within any of k buckets of size ℓ each. The corresponding thresholds $c^*_{k,\ell}$ are all known [31, 49, 46].

Which generalisation is better? Should we use **more hash functions or bigger buckets?** *On the one hand*, if you consider the **number of memory cells a query touches** in the worst case, then using more hash functions seems superior to using bigger buckets. For instance, with 3 hash functions we get a threshold of $c^*_3 \approx 0.918$ and have to scan three memory cells per query. When using 2 hash functions and buckets of size 2 we get a lower threshold of $c^*_{2,2} \approx 0.897$ and have to scan four memory cells per query. It turns out the four correlated cells in the two buckets do not constitute as powerful a choice as three independent locations.

On the other hand, the reduced number of **hash function evaluations and cache misses** strongly favours using larger buckets rather than additional hash functions, even if the number of memory cells associated with a key has to be higher to achieve the same load factor.

From what I have seen, practitioners aiming for high load factors seem to be pretty happy with either $k = 2$ or a middle ground using $k = 3$ hash functions and some bucket size $\ell \in \{3, \ldots, 8\}$, see e.g. [50, 71]. A related compromise assigns to each key k cells *within the same memory page* and 1 additional cell in a *backup page* [18] (see also [62]). It turns out most keys can then be stored on their primary page.

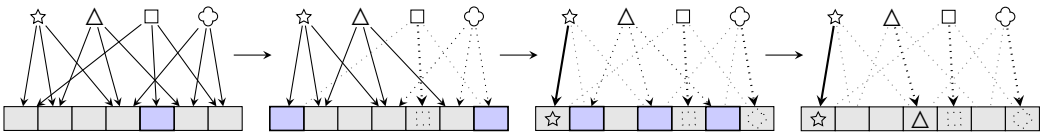
The message so far can be summarised as follows: Cuckoo hash tables are powered by the first two independent choices. Adding the third choice can help but competes for attention with other measures for increasing the load factor. **Two choices is all you really need.**

1.1.2 Power Dynamics: Cuckoo Table Construction and Insertion

The thresholds mentioned so far only relate to whether or not a rule-conforming placement of all keys in the hash table *exists*. But how can such a placement be *found and maintained*? For simplicity, we focus on cuckoo hashing with k hash functions and buckets of size $\ell = 1$.

Table construction. Constructing a cuckoo hash table is about finding a matching in a bipartite graph such as (3). Out of the box, the maximum matching algorithm by Hopcroft and Carp [41] has a *worst case* running time of $O(n^{3/2})$, but a specialised algorithm with *expected* running time $O(n)$ for $k \geq 2$ is known [44].

A conceptually interesting greedy algorithm is ***peeling***. It **identifies cells in the table that are an option for only one (remaining) key**. In the following illustration on the left, at first only □ can be placed in this way into cell 5, then ☆ and ◌ can be placed, and, finally, △ ends up with three cells to itself and can be placed in any one of them.



The peeling process

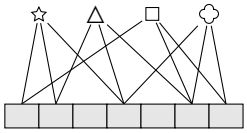
To better assess the utility of peeling, consider the load thresholds $c_k^{\triangle}$ up to which peeling manages to place all keys in a cuckoo hash table with k hash functions:

k	3	4	5	6	7
$c_k^{\triangle}$	0.81847	0.77228	0.70178	0.63708	0.58178

Values as determined in [56]. Variants in [14, 60, 43] and [51, Chapter 18].

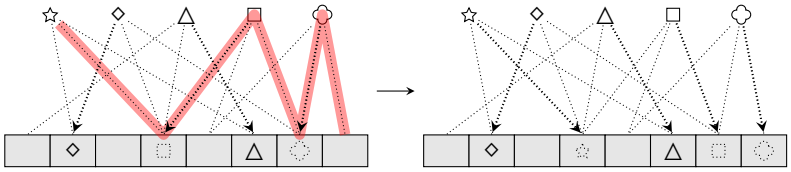
There are two things to notice here. *Less relevant* is that the value for $k = 2$ is missing. To see why, recall the graph with m vertices and n edges from (1). Peeling can handle trees but gets stuck if there is at least one cycle. Unfortunately, the probability for a cycle to exist is bounded away from zero as soon

as $\frac{n}{m}$ is bounded away from 0. So for peeling to work *whp* we need $k \geq 3$. *More relevant* is that $c_k^* < c_k^\triangle$, meaning there are load factors where a **placement exists whp but peeling fails to find one whp**. Even worse, the gap $c_k^* - c_k^\triangle$ between the thresholds increases with k, even converging to 1 for $k \to \infty$. This **disqualifies peeling as a general-purpose construction** algorithm for cuckoo hash tables. However, peeling will make a comeback in more specific settings (stay tuned!).



A placement exists, but peeling is stuck immediately.

Insertions. A construction algorithm suffices for a *static* key set, but for a dynamic data structure, we need to maintain a placement as keys are inserted and deleted over time. A deletion is trivial: Simply locate the key, by checking all associated cells, and remove it from there. An insertion on the other hand amounts to modifying an existing matching to incorporate a new key. This suggests that we look for an **augmenting path**.



In the picture, ☆ is the newly added key and moves into the cell previously used by □, which moves into the cell previously used by ◌, which moves into an empty cell.

There are two well-known strategies for finding such an augmenting path, both proposed in [30]. ***Breadth first search (BFS)*** insertion computes the **shortest augmenting path** in the natural BFS way. ***Random walk* (RW)** insertion goes ahead and places the unplaced key into one of its cells at random and *evicts* the key that was there before (if any). The evicted key is then also placed randomly and so on until an evicted key is placed into an empty cell. Strategies more clever than this have also been considered, some of which store some auxiliary data in the cells [45]. How good are these algorithms? It seems that, up to constant factors between them, they are equally **excellent in practice** in the sense that their running time does not depend on n, i.e. is $O(1)$.

Conjecture 1 (Echoing sentiments from [34, 68, 45, 36, 30])**.** *Consider a cuckoo hash table with $k \geq 2$ hash functions, buckets of size $\ell \geq 1$, and a load factor $\frac{n}{m} < c_{k,\ell}^* - \varepsilon$ for some $\varepsilon > 0$. Assume the keys are inserted sequentially using BFS insertion or RW insertion. Conditioned on the high probability event that a placement of all keys exists, the expected time to perform each insertion is $O(f(\varepsilon))$ for some function f that does not depend on n.*

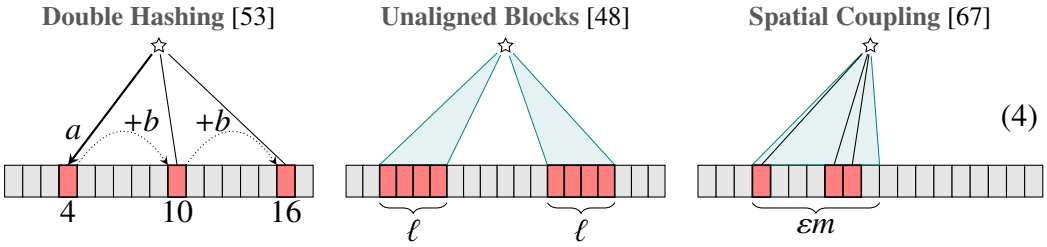
I know of **no proof**, neither for RW nor for BFS and for no pair of k and ℓ, except for the classical case of $k = 2$ and $\ell = 1$, where there is no choice regarding which key to evict from a given bucket (there is just one) and no choice regarding where to relocate an evicted key to (there is just one alternative). Partial proofs exist for the case with $\ell = 1$ for

- BFS, for $k > 8$ and under a stronger restriction on the load factor [30],
- RW, for large k and under a stronger restriction on the load factor [34],
- RW, for load factors below the *peeling* threshold $c_k^{\triangle}$ [68],
- RW, but only guaranteeing running times of $O(\text{polylog}\, n)$ [36, 33].

More progress towards proving the conjecture would be exciting. Maybe techniques from statistical physics, which have been a powerful tool for determining thresholds in the static case [49, 47, 46] can help with the dynamic case as well.

1.1.3 Creatively Wielding the Power

The number k of hash functions and the size ℓ of buckets are not the only degrees of freedom in the design space. Here is a short list of **further variants** that were considered and the reasons why.



(4)

Double Hashing. Mitzenmacher and Thaler [53] proposed that the buckets $b_1, \ldots, b_k$ assigned to a key are not chosen independently. Instead, only b_1 and an offset d are chosen at random, and $b_2, \ldots, b_n$ are defined as $b_i := b_1 + (i-1) \cdot d$, modulo the number of buckets. This **reduces** the amount of **entropy** in a key's hash values from $k \log m$ to $2 \log m$ with no apparent downsides. In particular the thresholds $c^*_{k,\ell}$ remain the same [47, 52].

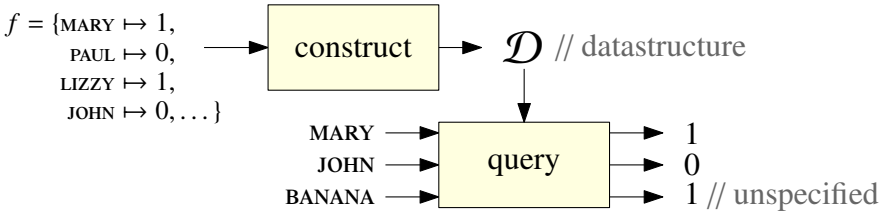
Unaligned Blocks. Lehman and Panigrahy proposed to use buckets that do not form a partition of the set of cells. Rather, *any contiguous sequence* of ℓ cells can occur as a bucket, regardless of the offset. This scheme yields **higher thresholds** than $c^*_{k,\ell}$ without affecting the access pattern of queries [48, 65].

Spatial Coupling. Walzer proposed that the k buckets assigned to a key are chosen within the same interval of εm buckets for some $\varepsilon > 0$. Assuming ε is small enough and the load factor is less than $c^*_{k,\ell}$, not only does a placement exist whp, but **peeling works whp**. [67]

I cannot help but wonder which other surprising effects can be achieved by further creative cuckoo hashing variants.

1.2 The Retrieval Problem & Random Matrices

In the ***retrieval problem***[6] we are given a set S and a function $f : S \to \{0,1\}^r$ for some $r \in \mathbb{N}$. Let us assume $r = 1$. The task is to construct a data structure that returns $f(x)$ when queried for $x \in S$. What is unusual is that a query for some $y \notin S$ may return an **arbitrary result**. In an instructive example due to Pagh and Dietzfelbinger [19], S is a set of names and f tells us if a name $x \in S$ is female ($f(x) = 1$) or male ($f(x) = 0$).



When f reflects typical English names, the data structure should return 0 for JOHN $\in$ domain(f) and 1 for MARY $\in$ domain(f). When queried for BANANA $\notin$ domain(f) both 0 and 1 are allowed, we don't care.

The **trivial solution** for this problem stores the **set of pairs**

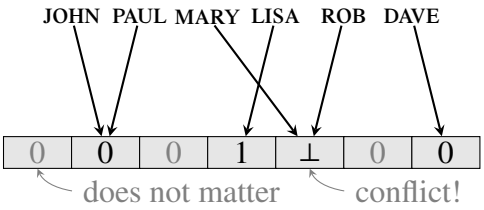
$$f = \{(\text{JOHN}, 0), (\text{MARY}, 1), (\text{LIZZY}, 1), (\text{PAUL}, 0), \dots\}$$

using a dictionary. This requires storing at least $n = |S|$ strings. However, we will soon see that the most space-efficient solutions require little more than n *bits*. Note that n bits are necessary if we make no further assumptions on the input.[7]

As a warm-up, here is a **compact solution** that needs $O(n)$ bits. We use an array of $m = O(n)$ cells that can store values from $\{0, 1, \perp\}$. We associate each $x \in S$ with a random cell $h(x) \in [m]$ and set a cell to 0 or 1 whenever a consistent value exists and $\perp$ in case of conflicts:

[6] The first mention of the problem under the name "retrieval" that I could find is in [16] in 2006. Bloomier filters [10] in 2004 are a clear spiritual predecessor and related to approximate membership. More background is given in [19].

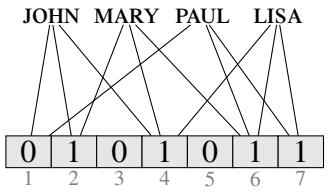
[7] If there are regularities such as the majority of names in domain(f) being male or most female names ending in a vowel, then compression maybe possible, see [42, 6, 38].



The keys involved in a conflict make up a constant fraction of all keys in expectation. For these, we can build a fallback data structure recursively. Some readers may have fun working out that we get **constant expected access time** and a total space consumption of roughly $e/2$ array cells per key for $m = n/2$.[8] This amounts to **$e \approx 2.72$ bits per key** when using the naive encoding $\{0 \mapsto 00, 1 \mapsto 11, \perp \mapsto 01\}$.An improved version of this idea is known as **filtered retrieval** [57].

1.2.1 The Power to be Independent: Retrieval via Random Linear Systems

To get closer to **succint retrieval** we consider strange cousins of cuckoo hash tables, cf. [7, 39]. While not employing the power of two choices in the traditional sense, their setup and analysis are closely related. Hash functions assign to each key **several cells** in an array of size $m \geq n$ that is populated with bits in such a way that taking the **XOR of all bits** associated with $x \in S$ yields $f(x)$.



(5)

A query for JOHN would, for instance, compute $f(\text{JOHN}) = 0 \oplus 1 \oplus 1 = 0$, where $\oplus$ denotes XOR. To construct the data structure we had to choose values $z_1, z_2, \ldots, z_7 \in \{0, 1\}$ to put into the array to simultaneously satisfy the equations

$$\begin{aligned} z_1 \oplus z_2 \oplus z_4 &= 0 && (\text{JOHN}) \\ z_2 \oplus z_4 \oplus z_6 &= 1 && (\text{MARY}) \\ z_1 \oplus z_6 \oplus z_7 &= 0 && (\text{PAUL}) \\ z_4 \oplus z_6 \oplus z_7 &= 1 && (\text{LISA}) \end{aligned}$$

Since $\oplus$ is addition in the **two element field** $\mathbb{F}_2 = \{0, 1\}$ these equations are **linear equations** over $\mathbb{F}_2$. So fasten your seatbelt for a bit of linear algebra, but don't

[8] Hint: Assume that $n/2$ names are male and $n/2$ names are female (you can later check that this is the worst case). Then use that for a fixed name x, the number of names of the opposite gender that share the hash value of x has distribution $\text{Bin}(\frac{n}{2}, \frac{1}{m})$, which converges to the Poisson distribution $\text{Po}(\frac{n}{2m}) = \text{Po}(1)$ that satisfies $\Pr[\text{Po}(1) > 0] = 1 - 1/e$.

worry, it's not so bad. We can write the above equations in **matrix form** as

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \begin{matrix} (\text{JOHN}) \\ (\text{MARY}) \\ (\text{PAUL}) \\ (\text{LISA}) \end{matrix}$$

When does such a system have a solution $\vec{z} \in \{0,1\}^m$? Well, the *columns* of the $n \times m$ matrix A should better span all of $\{0,1\}^n$, so that the right hand side vector $(f(x))_{x \in S} \in \{0,1\}^n$ can surely be attained as a linear combination of these columns. In other words, the column rank of A should be n. Since column rank and row rank are the same thing, the n *rows* of A have to be **linearly independent**.

What are our goals here? Remember that n is part of the input and we are free to choose two things: The number m of columns and the way in which keys are associated with row vectors via hash functions. Ideally we want that ***m* is small** so that $z \in \{0,1\}^m$ is **cheap to store** and we want row vectors to have **small Hamming weight** so queries are **cheap to evaluate**. Both of these goals are intuitively in tension with the independence requirement.

An encouraging fact is that even *square* matrices (i.e. $m = n$) where every entry is chosen by a biased coin with probability $p = \frac{\log n}{n}$ (i.e. rows have expected Hamming weight $\log n$) are regular with constant probability [13]. This is, however, not the most natural setup for our purposes.

1.2.2 New Data Structure, Same Thresholds: How Cuckoo Hashing Connects to Retrieval

A more natural setup already depicted in (5) associates k random cells with every key, which produces a matrix with ***k* ones per row** in random positions.

There is a **threshold** $c_k^\square$ for the ratio $c = \frac{n}{m}$ such that A has rank n whp when $c < c_k^\square - \varepsilon$ and A has rank less than n whp when $c > c_k^\square + \varepsilon$. Remarkably, it **coincides with the threshold for cuckoo hashing** with k hash functions, i.e. $c_k^\square = c_k^*$. Only the direction $c_k^\square \leq c_k^*$ is easy to show here. If A has rank n, then some selection of n columns induces a regular submatrix A' (highlighted below).

$$\begin{pmatrix} 1 & \mathbf{1} & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \mathbf{1} & 0 & 1 & 0 \\ \mathbf{1} & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & \mathbf{1} \end{pmatrix}$$

The determinant of A' is a non-zero number in $\mathbb{F}_2$, hence $\det(A') = 1$. By the Leibniz formula for determinants we have $1 = \det(A') = \sum_{\pi \in S_n} \prod_{i \in [n]} a'_{i,\pi(i)}$ where $(a'_{i,j})_{i,j \in [n]}$ are the entries of A' and S_n is the symmetric group with n elements. At least one $\pi \in S_n$ must yield a non-zero contribution to $\det(A')$, and all entries $a'_{i,\pi(i)}$ for $i \in [n]$ must then be 1. These entries are shown in bold in the matrix above and correspond to an injective placement of all keys in a cuckoo hashing setting. Bluntly:

$$c < c_k^{\square} \Leftrightarrow \text{“retrieval works whp”} \Rightarrow \text{“cuckoo hashing works whp”} \Leftrightarrow c < c_k^*$$

hence $c_k^{\square} \leq c_k^*$. The converse statement $c_k^{\square} \geq c_k^*$ requires a lot more work to prove [26, 59, 17, 11].

1.2.3 Longer Blocks and Bitparallel Queries

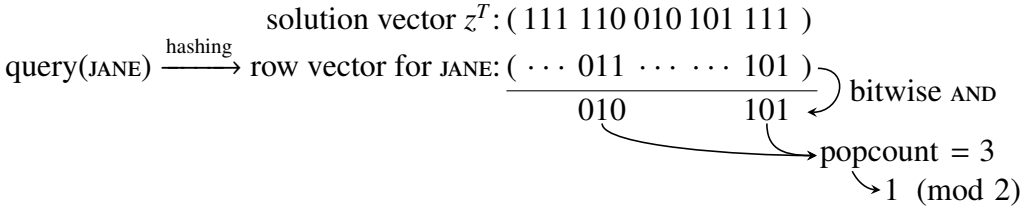
To get **thresholds closer to 1** there are similar options as in cuckoo hashing. **Increasing *k*** works but requires queries to combine data from more independent cells, resulting in **more cache misses**. Meh. We can also adopt the idea of using buckets of size ℓ. Naively connecting each key to each cell in two blocks of size ℓ *does not work* however. We merely obtain copies of identical columns that do not contribute to the matrix rank, as shown below on the left (from now on we use a dot “·” to indicate implicit zeroes that do not explicitly occur in any representation and we omit the right hand side of equations). *Instead* we should associate each key with a **random non-empty *subset* of cells of each of its blocks** as shown on the right.

$$\begin{pmatrix} \cdots & 111 & \cdots & 111 & \cdots \\ 111 & \cdots & \cdots & \cdots & 111 \\ \cdots & 111 & \cdots & \cdots & 111 \\ 111 & \cdots & 111 & \cdots & \cdots \\ \cdots & 111 & 111 & \cdots & \cdots \\ 111 & \cdots & \cdots & 111 & \cdots \end{pmatrix} \qquad \begin{pmatrix} \cdots & 010 & \cdots & 110 & \cdots \\ 100 & \cdots & \cdots & \cdots & 110 \\ \cdots & 111 & \cdots & \cdots & 010 \\ 010 & \cdots & 110 & \cdots & \cdots \\ \cdots & 111 & 001 & \cdots & \cdots \\ 101 & \cdots & \cdots & 001 & \cdots \end{pmatrix} \tag{6}$$

bad idea: each key (row) is associated with all cells within two blocks of size $\ell = 3$.

okay idea: each key (row) is associated with *some* cells within two blocks of size $\ell = 3$.

Precise thresholds are not known and not identical to the cuckoo hashing thresholds $c^*_{2,\ell}$. What we do know is that for suitable $\ell = \Theta(\log n)$ and $m = n + \Theta(\log n)$ we obtain a matrix with rank n whp [20]. This yields a **succinct** retrieval data structure with an almost optimal space requirement of $n + O(\log n)$ bits. Now consider a query. Given the solution vector $z \in \{0,1\}^m$ and the row vector associated with a key $x \in S$ we have to compute their **scalar product** to obtain $f(x)$.



An element-wise product of two vectors in $\mathbb{F}_2^\ell$ is a **bitwise AND** and the sum of the entries of a vector in $\mathbb{F}_2^\ell$ is a **population count modulo 2**. On a **word RAM** with word size $w = \Omega(\ell)$, we can perform all we need for a **query in $O(1)$** steps.

With constant time queries and almost optimal space, this seems like an ideal data structure until you realise that there is no fast way to construct it.

1.2.4 To Gauss or not to Gauss: Constructing Retrieval Data Structures

A problem with all this is that computing the solution vector $z \in \{0,1\}^m$ **requires solving a system** of linear equations. Spending $O(n^3)$ time on Gaussian elimination seems prohibitively expensive. Let us consider some alternatives.

Linear Time Solvers using Peeling. We can use a setup where peeling works whp, traditionally with $k = 3$ cells per key and a load factor below $c_3^\triangle \approx 0.82$ [7, 56]. Peeling means we look for a variable only appearing in one equation. We postpone initialising this variable to the end and ignore the equation until then. The remaining system may again have a variable appearing in only one of the remaining equations and so on. A different way of saying that a linear system is peelable is that its matrix can be transformed into echelon form by **row and column *permutations* alone**, with no need for row *additions*.

$$\begin{array}{c} \\ 1\\2\\3\\4\\5 \end{array}\!\!\begin{array}{c} 1\,2\,3\,4\,5\,6\,7 \\ \left(\begin{smallmatrix} 1&1&1&\cdot&\cdot&\cdot&\cdot\\ \cdot&\cdot&1&\cdot&\cdot&1&1\\ 1&\cdot&\cdot&\cdot&1&1&\cdot\\ 1&1&\cdot&\cdot&\cdot&\cdot&1\\ \cdot&1&\cdot&\mathbf{1}&1&\cdot&\cdot \end{smallmatrix}\right) \end{array} \rightarrow \begin{array}{c} \\ 5\\2\\3\\4\\1 \end{array}\!\!\begin{array}{c} 4\,2\,3\,1\,5\,6\,7 \\ \left(\begin{smallmatrix} 1&1&\cdot&\cdot&1&\cdot&\cdot\\ \cdot&\cdot&1&\cdot&\cdot&1&1\\ \cdot&\cdot&\cdot&1&\mathbf{1}&1&\cdot\\ \cdot&1&\cdot&1&\cdot&\cdot&1\\ \cdot&1&1&1&\cdot&\cdot&\cdot \end{smallmatrix}\right) \end{array} \rightarrow \begin{array}{c} \\ 5\\3\\2\\4\\1 \end{array}\!\!\begin{array}{c} 4\,5\,3\,1\,2\,6\,7 \\ \left(\begin{smallmatrix} 1&1&\cdot&\cdot&1&\cdot&\cdot\\ \cdot&1&\cdot&1&\cdot&1&\cdot\\ \cdot&\cdot&1&\cdot&\cdot&\mathbf{1}&1\\ \cdot&\cdot&\cdot&1&1&\cdot&\mathbf{1}\\ \cdot&\cdot&1&1&1&\cdot&\cdot \end{smallmatrix}\right) \end{array} \rightarrow \begin{array}{c} \\ 5\\3\\2\\4\\1 \end{array}\!\!\begin{array}{c} 4\,5\,6\,7\,2\,3\,1 \\ \left(\begin{smallmatrix} 1&1&\cdot&\cdot&1&\cdot&\cdot\\ \cdot&1&1&\cdot&\cdot&\cdot&1\\ \cdot&\cdot&1&1&\cdot&1&\cdot\\ \cdot&\cdot&\cdot&1&1&1&1\\ \cdot&\cdot&\cdot&\cdot&1&1&1 \end{smallmatrix}\right) \end{array}$$

The row and column of the highlighted 1 is swapped to the front to create an echelon form.

After finding the peeling order of equations and variables in **linear time**, we can find a solution with back substitution in linear time. The setup with peeling thresholds close to 1 from Section 1.1.3 reconciles this approach with close to optimal space efficiency both in theory [67] and in practice [40].

Quadratic Time Solver using Wiedemann's Algorithm. If $\mathbb{F}$ is a finite field, $A \in \mathbb{F}^{n\times n}$ a regular matrix with ψ non-zero entries and $b \in \mathbb{F}^n$ then a solution z to $A\cdot$

$z = b$ can be computed in $O(n\psi)$ field operations using **Wiedemann's algorithm** [70]. For sparse matrices with $\psi = O(n)$ this gives a **quadratic time** solver that can be generalised to non-square matrices A. Exploiting this for retrieval has been tried [4, 66], but it did not perform convincingly in experiments.

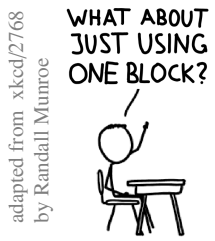
Bringing out the Big Gauss Rifles. A line of papers starting with Genuzio et al [37] cooked up ideas that made biting the bullet of performing Gaussian elimination seem like an almost appetising prospect.

The most important realisation is that we can **partition the input set** into many *shards* of expected size C using a hash function and construct a data structure for each of these shards. This reduces the running time from $O(n^3)$ to $O(\frac{n}{C} \cdot C^3) = O(nC^2)$. The price is an additional level of indirection during queries and a few bits of metadata for each of the $\frac{n}{C}$ shards.[9] This can be combined with **bit-level parallelism**, a **structured Gaussian elimination** that tries to keep the matrix as sparse as possible for as long as possible, and the **method of four Russians** [2], which eliminates entries from $O(\log n)$ columns at the same time. This yields fairly efficient solvers that are trivial to **parallelise**.

Part 2: The Power of *less than two* Choices

2.1 Ribbon: Choice is overrated

After a talk on the retrieval data structure using two blocks per matrix row (see (6) on the right) a listener asked a question.



Seth Pettie to my coauthor Martin Dietzfelbinger at Dagstuhl-Seminar 19051.

Supposedly he had this in mind:

$$\begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 0 & 0 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 0 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 0 & 1 & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 0 & 0 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 0 & 0 & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & 0 & 1 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 & 0 & 0 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & 0 & 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 0 & 1 & 1 & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \tag{7}$$

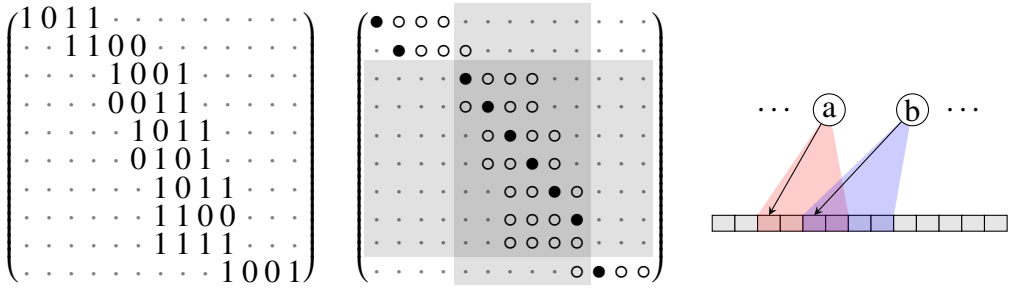
In the suggested scheme there is a block size w and each key is associated with a *starting position* $i \in [m - w + 1]$ and a coefficient vector $c \in \{0, 1\}^w$ that together

[9] In theory, a construction with two levels of indirection and shards of size $C = \sqrt{\log n}$ can lead to a succinct data structure with linear construction time [61]. However, astronomical input sizes are required for the asymptotic behaviour to kick in.

lead to a row in A with zeroes everywhere except for **one randomly placed block of w random bits**.

This blasphemous single-block-per-key suggestion throws the power of choices paradigm completely out the window. Could this work?[10] More precisely, for which values of m, n and w is such a matrix likely to have independent rows?

Simplification: Bounded Linear Probing. First, let us **sort the rows** by starting position. Second, let us simplify the problem by ignoring the patterns of zeroes and ones and just **consider the placement problem** where we wish to select one position within the block of each row without selecting the same column twice. Our question is now simply: In a linear probing hash table where each key hashes to a starting position, **can all keys be placed within $w - 1$ cells of their starting positions?**



rows sorted by starting position

simplified task: select one position in the block of each row using distinct columns

bounded linear probing

A greedy algorithm suffices to decide this question: Go through the keys in ascending order of starting position and place each key as far left as possible. The filled dots above indicate where the keys are placed. If we fail to place a key this way, as we do in the example, then this is witnessed by a range of N positions such that at least $N + 1$ keys have their block completely contained within the range (shaded grey with $N = 6$).

Connection to queuing theory. There is an equivalent way to describe the greedy placement algorithm just discussed (cf. [21]). We maintain a **FIFO queue** Q of keys and go through the table cells from left to right. When handling cell i we add the keys with starting position i to the back of Q and then place the first key in Q (if any) into position i and remove it from Q. This procedure places every key within its block if and only if the size of Q never exceeds w.[11] We therefore analyse the size q_i of Q after step i. If x_i is the number of keys with starting position i

[10] Spoiler alert: Yes, and it kicked off an entire line of papers for us [21, 25, 24].

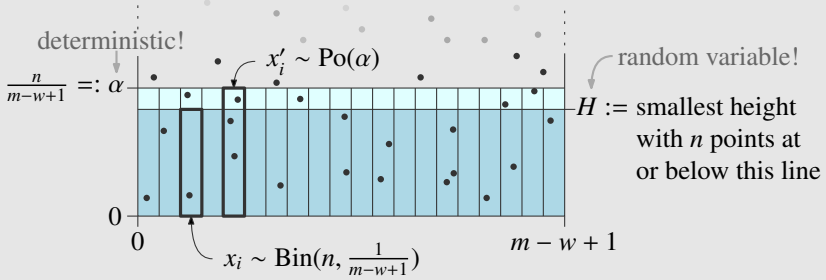
[11] Can you see why?

then

$$q_i = \max(0, q_{i-1} + x_i - 1) \text{ for } i \geq 1, \text{ and } q_0 = 0. \tag{8}$$

Each x_i has Binomial distribution $\mathrm{Bin}(n, \frac{1}{m-w+1})$ because each of the n keys has an independent chance of $\frac{1}{m-w+1}$ to have i as its starting position. Since we know $\sum_i x_i = n$, there is a slight but annoying negative correlation between the x_i. A common approximation in such a situation **replaces** the family $(x_i)_{i\in[m-w+1]}$ of **Binomial** random variables with an *independent* family $(x'_i)_{i\in[m-w+1]}$ of **Poisson** distributed random variables with the same expectation $\mathbb{E}[x'_i] = \alpha := \frac{n}{m-w+1}$. See Digression 3 for an explanation.

***Digression 3**: Poissonisation and random coupling (see also [54, Chapter 5.4]).*



The picture shows a strip of width $m - w + 1$ that is open to the top and contains points randomly with a *rate* of 1 point per unit of area (known as a *Poisson point process*). Within the strip we consider $m - w + 1$ disjoint sub-strips of height α as shown. The number of points x'_i within the i-th strip has distribution $\mathrm{Po}(\alpha)$. Let now H be the smallest vertical position such that exactly n points fall at or below H. Note that H is a random variable. Because each of the points below H is within any of the vertical strips with the same probability, the number x_i of points below H in the i-th strip has distribution $\mathrm{Bin}(n, \frac{1}{n-w+1})$.
Technically, we have constructed a **coupling** between the two families $(x_i)_{i\in[m-w+1]}$ and $(x'_i)_{i\in[m-w+1]}$ of random variables, i.e. we have embedded them in the same probability space. The outcomes of $(x_i)_{i\in[m-w+1]}$ and $(x'_i)_{i\in[m-w+1]}$ are now tightly linked. If $H \geq \alpha$ then we have $x_i \geq x'_i$ for all i and if $H \leq \alpha$ then we have $x_i \leq x'_i$ for all i, and both happen with probability roughly $1/2$. By changing α very slightly we can ensure that one of the two cases occurs whp. Relying on some monotonicity property of our surrounding problem such as "additional keys can only reduce the probability that all keys can be placed" allows us to **transfer a result that holds for the Poisson model to the Binomial setting**.

With this change, the values $q_0, q_1, q_2, \ldots$ from (8) are the states of a so-called **M/D/1** queue, which is a **Markov chain** that can be illustrated like this:

adapted from xkcd/779 by Randall Munroe

Markovian arrivals, i.e. people arrive randomly with a *rate* of α

state of the process: number q of people waiting

Deterministic server handling **1** person per time unit

served people leave

If $\alpha < 1$, then the customers arrive, on average, slower than they can be served and there is a **stable distribution** for the number q^* of waiting customers in the long run. **Queuing theory** literature promises an average queue length of $\mathbb{E}[q^*] = O(1/(1-\alpha))$ [15] and the tail bound $\Pr[q^* \geq w] = \mathrm{e}^{-\Theta(w/(1-\alpha))}$ [27, Prop 3.4]. This means we have to pick $w = \Omega(\log(n)/(1-\alpha))$ to ensure that the size of Q never exceeds w within $O(n)$ steps whp, which means all keys are validly placed in bounded linear probing.

So is this any good? We have essentially reinvented a **linear probing** hash table with the twist that the **maximum probe length** is guaranteed to not exceed $w = O(\log(n)/(1-\alpha))$ when the load factor is $\frac{n}{m} \approx \frac{n}{m-w+1} = \alpha$. This is strongly related to *Robin Hood hashing* [9] and not too exciting at first.

The arguments just given can be strengthened to show that the matrix from (7) with block length w has independent rows whp.[12] Moreover, a corresponding linear system can be solved with $O(n/(1-\alpha))$ row operations in expectation using **Gaussian elimination** because after sorting the rows by starting position the matrix is **already close to being in echelon form** (as already seen above). The number of row operations per column is linked to the average length of Q. Assuming we can handle $O(\log n)$ bits at a time using bit parallelism, the scalar product to be carried out by a query takes $O(1/(1-\alpha))$ operations. A **query** is very cache efficient because it reads w **contiguous bits from memory**.

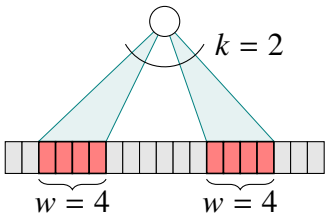
Overall we get a retrieval data structure with a load factor close to 1 that needs to access one contiguous sequence of bits from memory per query, with no bullet biting concerning expensive linear system solvers. My coauthor Martin and I were quite pleased with this solution [21], which has since been dubbed *ribbon retrieval* [24], because it improved upon the state of the art in 2019. Then we got an email from a database engineer who made it even better. [22]

[12] The corresponding variant of the M/D/1 queue involves customers that randomly pay attention only half the time when they are in the queue. At every step, the left-most customer that pays attention is served, if any. When the queue is long, the speed at which customers are served is hardly affected. The additional time a customer spends in the queue is $O(\log(n))$ whp.

2.2 Bumped Ribbon: The Power of 1 + ε Choices

The following ideas were intended for retrieval, but here I continue in the simpler hash table setting, where they also apply.

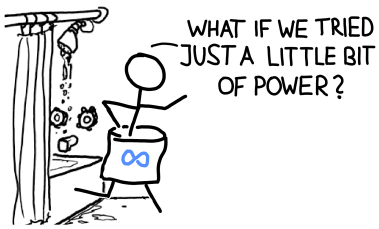
More than one, less than two. Let's take a step back. Consider a cuckoo hashing setting where each key is associated with k random starting positions $s_1, \dots, s_k \in [m-w+1]$ and may be placed in cells with an index in $\bigcup_{i\in[k]}\{s_i, \dots, s_i + w - 1\}$, i.e. within one of **$k$ randomly placed blocks of size w**.



For $k = 1$ we obtain the ribbon scheme discussed in Section 2.1 and for $k \geq 2$ a scheme by Lehman and Panigrahy briefly mentioned in Section 1.1.3. Let us now ask: What is the **smallest $w = w(k)$** such that all keys can be placed whp when the load factor is, say, $\alpha = 0.99$? We find:

k	1	2	3	4	5	6	7	…
$w(k)$	$\Theta(\log n)$	3	2	2	1	1	1	…

The point is: There is a **qualitative difference** between having just one choice of a block (if you even want to call it "choice") and the power of two or more choices. Could there be an interesting **middle ground between 1 and 2**?



adapted from xkcd/1715 by Randall Munroe

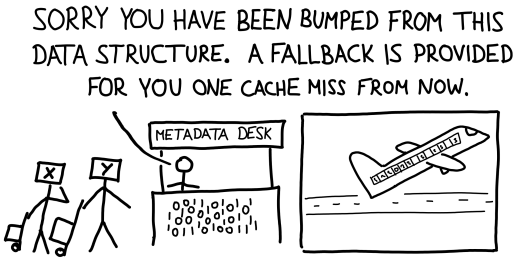
How I imagine Peter Dillinger came up with a great idea.

Could there be a power of $k = 1.5$ choices? What would this even mean? Well, a key could have 2 choices with probability 50% and only 1 choice with probability 50%. One might argue that this is 1.5 choices per key on average. I would nitpick and say that choices tend to aggregate multiplicatively (selecting among 2 choices and then among 3 choices gives 6 choices overall) so taking the geometric mean

and speaking of $\sqrt{2} \approx 1.41$ choices per key is more natural.[13] Either way, the problem is that placing the $\approx 50\%$ of keys that end up with 1 choice (and ignoring the others) still requires $w = \Omega(\log n)$ with no improvement over $k = 1$.

But here is a different kind of compromise. We partition the key set into groups of expected size b and offer **c choices for handling a group of keys** as a whole. This (arguably) corresponds to $k = c^{1/b}$ independent choices per key. Such a k might well be strictly between 1 and 2. Dillinger et al. [24] dubbed a concrete setup of this kind *bumped ribbon*.

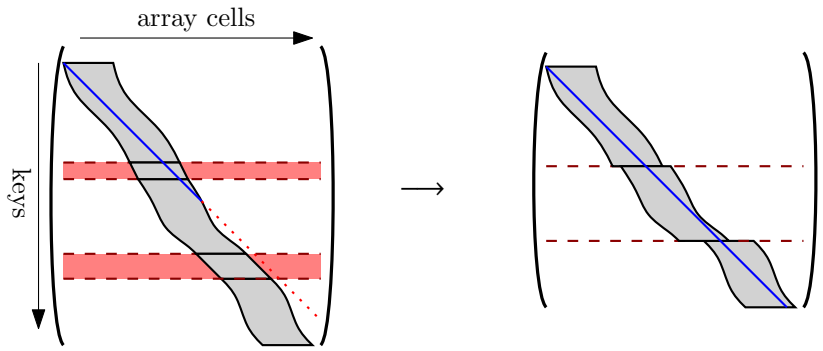
Bumped Ribbon. Like before, each key is associated with one block of w consecutive positions. Keys are partitioned into groups with consecutive starting positions. For each group, there are two choices. The keys are either stored normally or the entire group is ***bumped***, meaning the keys are moved to a recursively constructed **fallback data structure**.[14]



To get some intuition, consider the following matrix-like visualisations were rows correspond to the keys sorted by starting position and columns to array cells. Grey shading in position (i, j) indicates that the ith key may go into array slot j (imagine we have zoomed out so the border of the grey area looks like a continuous curve). We'd like for the **matrix diagonal** to run through the shaded region because then the ith key can go into the ith slot so (i) every key would be placed and (ii) every slot would be put to use. We can ensure (i) and *mostly* ensure (ii) with two measures. We start with a **slightly overloaded** data structure (a load factor $\alpha > 1$) and then **bump groups of keys in strategic positions**.

[13] This is the right view when considering the *average* amount of *information* that has to be recorded per key to encode the choices. This information in bits is the $\log_2$ of the number of choices.

[14] To bring this more in line with the previous setting, we can consider the fallback data structure to be a special segment in the primary data structure rather than separate and ensure that that bumped keys are not bumped *again* by the fallback data structure. In [23] such a variant is called Bu^{1}RR, but introducing it here would be a distraction from our main point.



an overloaded data structure where not all keys could be placed

after bumping some keys, the remaining keys perfectly fill all cells

Details on the exact proposal are given in Digression 4. For any $\varepsilon > 0$, bumped ribbon has $1+\varepsilon$ choices per key in the mean (for each *group* of $O(1/\varepsilon)$ keys, there are 3 choices: bump none, some, or all keys from the group). The block size is $w = \Omega(\log(1/\varepsilon)/\sqrt{\varepsilon})$. The memory overhead (the fraction of wasted space taking into account unused cells, metadata and fallback data structures) is of order $O(\varepsilon)$. In this sense, we are using a **power of $1+\varepsilon$ choices**! Framed in terms of the previous table:

k	1	$\mathbf{1+\varepsilon}$	2	3	4	5	6	7	…
$w(k)$	$\Theta(\log n)$	$\boldsymbol{O(\frac{\log(1/\varepsilon)}{\sqrt{\varepsilon}})}$	3	2	2	1	1	1	…

While we have not evaluated the performance of bumped ribbon as a static *dictionary*, the corresponding static *retrieval* data structure has significantly improved upon the state of the art by marrying three useful properties: First, courtesy

***Digression 4**: Some details on bumped ribbon.*
We are actually bumping keys by starting position, meaning a key is bumped if and only if its starting position is marked as bumped. The starting positions are partitioned into groups of size $O(w^2/\log w)$. For each group, we store 2 bits of metadata that indicate which positions are bumped: (i) none, (ii) the first $\frac{3}{8}w$ positions, or (iii) all positions, where (iii) is a rarely used emergency option. The initial load factor (before bumping) is set to be $\alpha = 1 + \Theta(\frac{\log w}{w})$. The metadata can be set such that:

1. Each non-bumped key is placed within w cells of its starting position.
2. Only a $w^{-\Omega(1)}$ fraction of the cells remains empty.
3. The overall memory overhead is dominated by the metadata and thus $O(\frac{\log w}{w^2})$ bits per key.

of $1 + \varepsilon$ choices, the **block size w need not grow with n**. Second, there is a good chance of answering a query with a **single memory access** since most keys are not bumped. Third, we get **close to linear time construction** since the relevant linear system is already close to echelon form.

~~TL;DR~~ Conclusion

Our journey began with a discussion of cuckoo hash tables that have worst-case access times of $O(1)$ and a load factor of $\alpha = \frac{1}{2} - \varepsilon$. We considered various approaches for increasing the load factor and argued that increasing the number of independent choices that every key has is not the most attractive option due to the increased number of cache misses incurred by each query. While two choices per key is qualitatively different from a single choice (i.e. "no choice"), the step to more than two choices is comparatively underwhelming. What's more, when considering static retrieval data structures closely related to cuckoo hash tables we find that the flexibility afforded by multiple choices can come at the price of expensive construction algorithms.

Wondering if "two" is really the least amount of choice one can have per key motivates *bumped ribbon*. When used as a hash table it is much like linear probing, except that the maximum probe length is guaranteed to never exceed a constant w. Keys can be marked as "bumped", meaning they are placed in a fallback data structure, if their part of the hash table is too crowded. Superficially, there are two choices per key: bumped or not bumped. However, there is more rigidity in two important ways: First, only few keys are bumped so the entropy in any key's choice is much less than one bit. Second, bumping decisions are correlated with large groups of keys being bumped as a whole.

In a sense we have harnessed the power of $1 + \varepsilon$ choices per key. The qualitative power of choices is present, namely the ability to defuse certain worst-case constellations, but the price that flexibility brings is significantly attenuated.

References

[1] Austin Appleby. Murmurhash3, 2012. URL: `https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp`.

[2] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk SSSR, 194*, 1970.

[3] Jean-Philippe Aumasson and Daniel J. Bernstein. Siphash: A fast short-input PRF. In *Proc. 13th INDOCRYPT*, pages 489–508, 2012. `doi:10.1007/978-3-642-34931-7_28`.

[4] Martin Aumüller, Martin Dietzfelbinger, and Michael Rink. Experimental variations of a theoretically good retrieval data structure. In *Proc. 17th ESA*, pages 742–751, 2009. doi:10.1007/978-3-642-04128-0_66.

[5] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999. doi:10.1137/S0097539795288490.

[6] Djamal Belazzougui and Rossano Venturini. Compressed static functions with applications. In *Proc. 24th SODA*, pages 229–240. SIAM, 2013. doi:10.1137/1.9781611973105.17.

[7] Fabiano Cupertino Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Inf. Syst.*, 38(1):108–131, 2013. doi:10.1016/j.is.2012.06.002.

[8] Julie Anne Cain, Peter Sanders, and Nicholas C. Wormald. The random graph threshold for k-orientiability and a fast algorithm for optimal multiple-choice allocation. In *Proc. 18th SODA*, pages 469–476, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283433.

[9] Pedro Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, Canada, 1986.

[10] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proc. 15th SODA*, pages 30–39. SIAM, 2004. URL: http://dl.acm.org/citation.cfm?id=982792.982797.

[11] Amin Coja-Oghlan, Mihyun Kang, Lena Krieg, and Maurice Rolvien. The k-XORSAT threshold revisited. *CoRR*, abs/2301.09287, 2023. arXiv:2301.09287.

[12] Yann Collet. xxhash - extremely fast hash algorithm, 2020. URL: https://github.com/Cyan4973/xxHash.

[13] Colin Cooper. On the rank of random matrices. *Random Structures & Algorithms*, 16(2):209–232, 2000. doi:10.1002/(SICI)1098-2418(200003)16:2<209::AID-RSA6>3.0.CO;2-1.

[14] Colin Cooper. The cores of random hypergraphs with a given degree sequence. *Random Struct. Algorithms*, 25(4):353–375, 2004. doi:10.1002/rsa.20040.

[15] Robert B. Cooper. *Introduction to queueing theory*. George Washington University, 3rd edition, 1990.

[16] Erik D. Demaine, Friedhelm Meyer auf der Heide, Rasmus Pagh, and Mihai Pătraşcu. De dictionariis dynamicis pauco spatio utentibus (*lat.* on dynamic dictionaries using little space). In *LATIN*, pages 349–361, 2006. doi:10.1007/11682462_34.

[17] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In *Proc. 37th ICALP (1)*, pages 213–225, 2010. doi:10.1007/978-3-642-14165-2_19.

[18] Martin Dietzfelbinger, Michael Mitzenmacher, and Michael Rink. Cuckoo hashing with pages. In *Proc. 19th ESA*, pages 615–627, 2011. doi:10.1007/978-3-642-23719-5_52.

[19] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In *Proc. 35th ICALP (1)*, pages 385–396, 2008. doi:10.1007/978-3-540-70575-8_32.

[20] Martin Dietzfelbinger and Stefan Walzer. Constant-time retrieval with $O(\log m)$ extra bits. In *Proc. 36th STACS*, pages 24:1–24:16, 2019. doi:10.4230/LIPIcs.STACS.2019.24.

[21] Martin Dietzfelbinger and Stefan Walzer. Efficient Gauss elimination for near-quadratic matrices with one short random block per row, with applications. In *Proc. 27th ESA*, pages 39:1–39:18, 2019. doi:10.4230/LIPIcs.ESA.2019.39.

[22] Peter Dillinger. Ribbon: A practical and near-optimal static bloom alternative for rocksdb, 2020. URL: https://www.youtube.com/watch?v=XfwxUBL8xT8&t=1h16m10s.

[23] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership using ribbon. *CoRR*, abs/2109.01892, 2021. arXiv:2109.01892.

[24] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership using ribbon. In *20th SEA*, volume 233, pages 4:1–4:20, 2022. doi:10.4230/LIPIcs.SEA.2022.4.

[25] Peter C. Dillinger and Stefan Walzer. Ribbon filter: practically smaller than Bloom and Xor. *CoRR*, 2021. arXiv:2103.02515.

[26] Olivier Dubois and Jacques Mandler. The 3-XORSAT threshold. In *Proc. 43rd FOCS*, pages 769–778, 2002. doi:10.1109/SFCS.2002.1182002.

[27] Regina Egorova, Bert Zwart, and Onno Boxma. Sojourn time tails in the M/D/1 processor sharing queue. *Probability in the Engineering and Informational Sciences*, 20:429–446, 2006. doi:10.1017/S0269964806060268.

[28] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 1960. URL: http://www.renyi.hu/~p_erdos/1961-15.pdf.

[29] Daniel Fernholz and Vijaya Ramachandran. The k-orientability thresholds for $g_{n,p}$. In *Proc. 18th SODA*, pages 459–468, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283432.

[30] 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.

[31] Nikolaos Fountoulakis, Megha Khosla, and Konstantinos Panagiotou. The multiple-orientability thresholds for random hypergraphs. *Combinatorics, Probability & Computing*, 25(6):870–908, 2016. doi:10.1017/S0963548315000334.

[32] Nikolaos Fountoulakis and Konstantinos Panagiotou. Sharp load thresholds for cuckoo hashing. *Random Struct. Algorithms*, 41(3):306–333, 2012. doi:10.1002/rsa.20426.

[33] Nikolaos Fountoulakis, Konstantinos Panagiotou, and Angelika Steger. On the insertion time of cuckoo hashing. *SIAM J. Comput.*, 42(6):2156–2181, 2013. doi:10.1137/100797503.

[34] Alan M. Frieze and Tony Johansson. On the insertion time of random walk cuckoo hashing. *Random Struct. Algorithms*, 54(4):721–729, 2019. doi:10.1002/rsa.20808.

[35] Alan M. Frieze and Páll Melsted. Maximum matchings in random bipartite graphs and the space utilization of cuckoo hash tables. *Random Struct. Algorithms*, 41(3):334–364, 2012. doi:10.1002/rsa.20427.

[36] Alan M. Frieze, Páll Melsted, and Michael Mitzenmacher. An analysis of random-walk cuckoo hashing. *SIAM J. Comput.*, 40(2):291–308, 2011. doi:10.1137/090770928.

[37] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In *Proc. 15th SEA*, pages 339–352, 2016. doi:10.1007/978-3-319-38851-9_23.

[38] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of ([compressed] static | minimal perfect hash) functions. *Information and Computation*, 2020. doi:10.1016/j.ic.2020.104517.

[39] Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than Bloom and cuckoo filters. *ACM J. Exp. Algorithmics*, 25:1–16, 2020. doi:10.1145/3376122.

[40] Thomas Mueller Graf and Daniel Lemire. Binary fuse filters: Fast and smaller than xor filters. *ACM J. Exp. Algorithmics*, 27:1.5:1–1.5:15, 2022. doi:10.1145/3510449.

[41] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973. doi:10.1137/0202019.

[42] Jóhannes B. Hreinsson, Morten Krøyer, and Rasmus Pagh. Storing a compressed function with constant time access. In *Proc. 17th ESA*, pages 730–741, 2009. doi:10.1007/978-3-642-04128-0_65.

[43] Svante Janson and Malwina J. Luczak. A simple solution to the k-core problem. *Random Struct. Algorithms*, 30(1-2):50–62, 2007. doi:10.1002/rsa.20147.

[44] Megha Khosla. Balls into bins made faster. In *Proc. 21st ESA*, pages 601–612, 2013. doi:10.1007/978-3-642-40450-4_51.

[45] William Kuszmaul. Brief announcement: Fast concurrent cuckoo kick-out eviction schemes for high-density tables. In *28th SPAA*, pages 363–365. ACM, 2016. doi:10.1145/2935764.2935814.

[46] M. Leconte, M. Lelarge, and L. Massoulié. Convergence of multivariate belief propagation, with applications to cuckoo hashing and load balancing. In *Proc. 24th SODA*, pages 35–46, 2013. URL: http://dl.acm.org/citation.cfm?id=2627817.2627820.

[47] Mathieu Leconte. Double hashing thresholds via local weak convergence. In *Proc. 51st Allerton*, pages 131–137, 2013. doi:10.1109/Allerton.2013.6736515.

[48] Eric Lehman and Rina Panigrahy. 3.5-way cuckoo hashing for the price of 2-and-a-bit. In *Proc. 17th ESA*, pages 671–681, 2009. doi:10.1007/978-3-642-04128-0_60.

[49] Marc Lelarge. A new approach to the orientation of random hypergraphs. In *Proc. 23rd SODA*, pages 251–264. SIAM, 2012. doi:10.1137/1.9781611973099.23.

[50] Tobias Maier, Peter Sanders, and Stefan Walzer. Dynamic space efficient hashing. *Algorithmica*, 81(8):3162–3185, 2019. doi:10.1007/s00453-019-00572-x.

[51] Marc Mezard and Andrea Montanari. *Information, Physics, and Computation*. Oxford University Press, Inc., USA, 2009.

[52] Michael Mitzenmacher, Konstantinos Panagiotou, and Stefan Walzer. Load thresholds for cuckoo hashing with double hashing. In *Proc. 16th SWAT*, pages 29:1–29:9, 2018. doi:10.4230/LIPIcs.SWAT.2018.29.

[53] Michael Mitzenmacher and Justin Thaler. Peeling arguments and double hashing. In *Proc. 50th Allerton*, pages 1118–1125, 2012. doi:10.1109/Allerton.2012.6483344.

[54] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, New York, NY, USA, 2nd edition, 2017.

[55] Michael David Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, Harvard University, 1996. URL: http://www.eecs.harvard.edu/~michaelm/postscripts/mythesis.pdf.

[56] Michael Molloy. Cores in random hypergraphs and Boolean formulas. *Random Struct. Algorithms*, 27(1):124–135, 2005. doi:10.1002/rsa.20061.

[57] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and perfect hashing using fingerprinting. In *Proc. 14th SEA*, pages 138–149, 2014. doi:10.1007/978-3-319-07959-2_12.

[58] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. doi:10.1016/j.jalgor.2003.12.002.

[59] Boris Pittel and Gregory B. Sorkin. The satisfiability threshold for k-XORSAT. *Combinatorics, Probability & Computing*, 25(2):236–268, 2016. doi:10.1017/S0963548315000097.

[60] Boris Pittel, Joel Spencer, and Nicholas C. Wormald. Sudden emergence of a giant k-core in a random graph. *J. Comb. Theory, Ser. B*, 67(1):111–151, 1996. doi:10.1006/jctb.1996.0036.

[61] Ely Porat. An optimal Bloom filter replacement based on matrix solving. In *Proc. 4th CSR*, pages 263–273, 2009. doi:10.1007/978-3-642-03351-3_25.

[62] Ely Porat and Bar Shalem. A cuckoo hashing variant with improved memory utilization and insertion time. In *Proc. 22nd DCC*, pages 347–356, 2012. doi:10.1109/DCC.2012.41.

[63] Michael Rink. *Thresholds for Matchings in Random Bipartite Graphs with Applications to Hashing-Based Data Structures*. PhD thesis, Technische Universität Ilmenau, Germany, 2015. URL: http://www.db-thueringen.de/servlets/DocumentServlet?id=25985.

[64] Mikkel Thorup. High speed hashing for integers and strings. *CoRR*, 2015. arXiv:1504.06804.

[65] Stefan Walzer. Load thresholds for cuckoo hashing with overlapping blocks. In *Proc. 45th ICALP*, pages 102:1–102:10, 2018. doi:10.4230/LIPIcs.ICALP.2018.102.

[66] Stefan Walzer. *Random Hypergraphs for Hashing-Based Data Structures*. PhD thesis, Technische Universität Ilmenau, 2020. URL: https://www.db-thueringen.de/receive/dbt_mods_00047127.

[67] Stefan Walzer. Peeling close to the orientability threshold: Spatial coupling in hashing-based data structures. In *Proc. 32nd SODA*, pages 2194–2211. SIAM, 2021. doi:10.1137/1.9781611976465.131.

[68] Stefan Walzer. Insertion time of random walk cuckoo hashing below the peeling threshold. In *30th ESA*, volume 244 of *LIPIcs*, pages 87:1–87:11, 2022. doi:10.4230/LIPIcs.ESA.2022.87.

[69] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981. doi:10.1016/0022-0000(81)90033-7.

[70] Douglas H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trans. Inf. Theory*, 32(1):54–62, 1986. doi:10.1109/TIT.1986.1057137.

[71] Jens Zentgraf and Sven Rahmann. Fast gapped k-mer counting with subdivided multi-way bucketed cuckoo hash tables. In *22nd WABI*, volume 242 of *LIPIcs*, pages 12:1–12:20, 2022. doi:10.4230/LIPIcs.WABI.2022.12.