Published: 13 February 2026
Online AM: 03 January 2026
Accepted: 13 December 2025
Revised: 29 July 2025
Received: 29 July 2025

RESEARCH-ARTICLE

Ribbon: Fast Succinct Static Retrieval and Approximate Membership

MARTIN DIETZFELBINGER, Technical University of Ilmenau, Ilmenau, Thuringen, Germany

PETER C. DILLINGER, Meta, Menlo Park, CA, United States

LORENZ HÜBSCHLE, Karlsruhe Institute of Technology, Karlsruhe, Baden-Wurttemberg, Germany

PETER SANDERS, Karlsruhe Institute of Technology, Karlsruhe, Baden-Wurttemberg, Germany

STEFAN WALZER, Karlsruhe Institute of Technology, Karlsruhe, Baden-Wurttemberg, Germany

Open Access Support provided by:

Karlsruhe Institute of Technology

Meta

Technical University of Ilmenau

Ribbon: Fast Succinct Static Retrieval and Approximate Membership

MARTIN DIETZFELBINGER, Technische Universität Ilmenau, Ilmenau, Germany
PETER C. DILLINGER, Meta Platforms Inc, Seattle, United States
LORENZ HÜBSCHLE, Karlsruhe Institute of Technology, Karlsruhe, Germany and Firebolt Analytics, Munich, Germany
PETER SANDERS, Karlsruhe Institute of Technology, Karlsruhe, Germany
STEFAN WALZER, Karlsruhe Institute of Technology, Karlsruhe, Germany

Given a set $S \subseteq \mathcal{U}$ and a function $f\colon S \longrightarrow \{0,1\}^r$, a static *retrieval* data structure for f supports queries that return $f(x)$ for $x \in S$ and an arbitrary value from $\{0,1\}^r$ for $x \in \mathcal{U} \setminus S$. Retrieval data structures can be used to implement a static *approximate membership query (AMQ)* data structure, i.e., a Bloom filter alternative, with false positive rate 2^{-r}. The information-theoretic space lower bound for both tasks is $r|S|$ bits, and here we aim to use space $r|S|(1+\varepsilon)$ bits for a small *overhead* ε, including *succinct* constructions with $\varepsilon = o(1)$.

A well-known approach to this task associates each key $x \in S$ with a row vector $\vec{h}(x) \in \{0,1\}^m$ and stores a matrix $Z \in \{0,1\}^{m\times r}$ such that $\vec{h}(x) \cdot Z = f(x)$ for every $x \in S$. We propose a new variant where $\vec{h}(x)$ contains a short block of random bits at a random position $s(x)$, and is otherwise zero. Sorting the row vectors by $s(x)$ gives a matrix $A \in \{0,1\}^{n\times m}$ with non-zero entries concentrated in a “ribbon” along a generalized diagonal. This makes a variant of Gaussian elimination particularly efficient at computing Z. We thus obtain simple data structures called *Standard Ribbon Retrieval* and *Homogeneous Ribbon Filter*. We then refine the construction using *bumping* (a variant of backyarding) and *overloading* (using $m < n$) to obtain *bumped ribbon retrieval* (“BuRR”), with overhead $O(\frac{\log w}{rw^2})$, query time $O(1 + \frac{rw}{\log n})$, and expected construction time $O(nw)$, for a tuning parameter $w = O(\log n)$ that opens a trade-off between space and running time.

Our experiments reveal our implementations to be the first to simultaneously achieve small overheads and fast running times in practice, with BuRR achieving overheads well below 1 % while being faster than most competitors, which have larger space overheads. This efficiency, including favorable constants, stems from a combination of simplicity, word parallelism, and high locality.

We offer a unified theoretical perspective on these three ribbon-based data structures, including a nontrivial rigorous analysis of their running times and memory consumption.

Stefan Walzer was funded by the German Research Foundation (DFG) 465963632.
This article includes results from two conference publications [24, 25] and two preprints [26, 27]. It subsumes these works except for some experimental data exclusive to [27].
The code and scripts used for our experiments are available under a permissive license at github.com/lorenzhs/BuRR and github.com/lorenzhs/fastfilter_cpp.

Authors' Contact Information: Martin Dietzfelbinger, Technische Universität Ilmenau, Ilmenau, Thuringia, Germany; e-mail: martin.dietzfelbinger@tu-ilmenau.de; Peter C. Dillinger, Meta Platforms Inc, Seattle, Washington, United States; e-mail: peterd@fb.com; Lorenz Hübschle, Karlsruhe Institute of Technology, Karlsruhe, Germany and Firebolt Analytics, Munich, Germany; e-mail: huebschle@4z2.de; Peter Sanders, Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: sanders@kit.edu; Stefan Walzer (corresponding author), Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: stefan.walzer@kit.edu.

ACM 0004-5411/2026/02-ART7
https://doi.org/10.1145/3785417

CCS Concepts: • **Theory of computation → Data compression**; • **Information systems → Point lookups;**

Additional Key Words and Phrases: AMQ, Bloom filter, dictionary, linear algebra, randomized algorithm, retrieval data structure, static function data structure, succinct data structure, perfect hashing

ACM Reference Format:
Martin Dietzfelbinger, Peter C. Dillinger, Lorenz Hübschle, Peter Sanders, and Stefan Walzer. 2026. Ribbon: Fast Succinct Static Retrieval and Approximate Membership. *J. ACM* 73, 1, Article 7 (February 2026), 45 pages.
https://doi.org/10.1145/3785417

1 Introduction

Retrieval is a basic task in data structures. Assume $\mathcal{U}$ is a set, called the *universe*, containing elements, called *keys*, and R is a finite set. Given a set $S \subseteq \mathcal{U}$ and a function $f \colon S \to R$, we are to build a data structure $\mathcal{R}$ that can be queried with any key $x \in \mathcal{U}$. A query for $x \in S$ must return $f(x)$, while a query for $x \in \mathcal{U} \setminus S$ can give an arbitrary value in R. In this work, we focus on the *static* version of this data structuring task, where f is given at the beginning and never changes. We are interested in three performance characteristics: construction time, query time, and storage space. For retrieval data structures the information theoretic lower bound on storage space is $n \log |R|$ bits, where $n = |S|$, and we strive for data structures that use space close to this bound. (This significantly undercuts the $\Omega((\log |\mathcal{U}| + r)n)$ bits required by a (static) dictionary. The reason for this difference is that a dictionary also solves the membership problem for S, which a retrieval structure need not do.) If the data structure uses $(1 + \varepsilon)n \log |R|$ bits, we say it has *(space) overhead* ε; if $\varepsilon = o(1)$, the data structure is called *succinct*.

For a quick overview we formulate our main results here in a simplified form, making specific choices for a central tuning parameter w, the "ribbon width". The range R is $\{0, 1\}^r$ for some $r \geq 1$; thus we talk about "r-bit data structures". The motivation for words like "ribbon", "bumped", and "homogeneous" in the names of the data structures will become apparent later. The basic new data structure, called *Standard Ribbon*, solves the static retrieval task.

Theorem 1.1 (Preliminary formulation of Theorem 1.5). *For any $\varepsilon > 0$, there is an r-bit Standard Ribbon retrieval data structure with expected construction time $O(n/\varepsilon^2)$, query time $O(r/\varepsilon)$ and overhead $O(\varepsilon)$.*

The basic construction will be enhanced with well-known techniques, like using a backyard data structure, as well as new techniques, like "bumping" in chunks and overloading. The name of the complete data structure is "Bumped Ribbon Retrieval", or *BuRR*. The following is an important special case of our main technical result.

Theorem 1.2 (Special case of Theorem 1.6). *For $r = O(\log n)$ there is an r-bit BuRR data structure with space overhead $O(\frac{\log\log n}{r(\log n)^2})$, query time $O(r)$, and expected construction time $O(n \log n)$.*

Choosing the tuning parameter w differently and employing further (standard) techniques like precomputed tables we can reduce the query time to $O(1)$ and the construction time to $O(n)$, for smallish r.

Corollary 1.3 (Restated below as Theorem 1.7). *There is a variant of BuRR for $r = O(\sqrt{\log n})$ with constant query time, linear expected construction time, and overhead $O(\frac{\log\log n}{r \log n})$.*

Although this result is mostly of theoretical interest, we note that it slightly improves the hitherto best known asymptotic bound of $O(\frac{\log^2 \log n}{r \log n})$ [6] on the overhead for this parameter setting.

The data structures presented here power state of the art constructions for static (*minimal*) *perfect hashing* [44, 45]. But the arguably more important application concerns (static) ***approximate***

membership query (***AMQ***) data structures or *Bloom filter replacements*, where a set S of keys in $\mathcal{U}$ is given and we ask for a (randomized) data structure that answers queries for $x \in \mathcal{U}$ with true or false, as follows: If $x \in S$, the answer is true. For $x \in U \setminus S$, the answer is false, except with a probability of at most φ, called the *false positive probability*. Based on the new retrieval structures like BuRR we obtain, by standard methods [20], static AMQ data structures with the same space overhead and the same query time. A different construction, starting directly from Standard Ribbon, leads to a very simple new AMQ data structure called *Homogeneous Ribbon Filter*, which is not directly based on retrieval. We obtain the following result.

Theorem 1.4 (Preliminary formulation of Theorem 1.8). *For arbitrary $r \geq 1$ and $\varepsilon \in (0, 1/2]$ there is a Homogeneous Ribbon Filter with space overhead ε and false positive probability $(1+O(\varepsilon^2))2^{-r}$. On a word RAM with word size at least $\Omega(\max(r, \log(1/\varepsilon))/\varepsilon)$ the expected construction time is $O(n/\varepsilon)$ and the query time is $O(r)$.*

Of central importance for the work presented in this article is the *engineering aspect*: We strive for algorithms that behave as predicted by the theoretical results already for realistic input sizes, and we utilize insights from empirical observations for algorithmic and theoretical improvements. BuRR is arguably the first succinct retrieval data structure admitting a competitive practical implementation. In our extensive experiments it outperforms previous constructions over a wide range of situations. Since any retrieval data structure can be used as an AMQ data structure (but not vice versa) our experiments compare the wider range of AMQ data structures (some of which are retrieval-based). Roughly speaking, for space overheads between 8% and 30% the Homogeneous Ribbon Filter has a favorable combination of query and construction speed. For smaller space overheads, BuRR-based AMQ structures outperform all competitors.

In the rest of the introduction we give a more detailed overview of the techniques we use, of our results, and of related work.

1.1 Background: Linear Systems

Our constructions utilize a popular basic approach for retrieval data structures, namely, using the solution of a linear system of equations as the data structure. Originally, the idea was developed in the context of the construction of **minimal perfect hash functions (MPHF)** [10, 11, 14, 16, 20, 33, 52, 55].[1] Assume the range R has the structure of an Abelian group, with operation $\oplus$. For $m = (1 + \varepsilon)n$ we associate a set[2] $K_x \subseteq [m]$ with each $x \in \mathcal{U}$, by some pseudo-random (hash) function(s). Then, the system

$$\bigoplus_{i \in K_x} Z_i = f(x), \text{ for } x \in S, \tag{1}$$

of equations with unknowns $Z_1, \ldots, Z_m \in R$ is considered. If this system has a solution, an array containing such a solution can be used as the data structure. Building the data structure amounts to solving system (1); a query for $x \in \mathcal{U}$ requires reading out and adding Z_i for $i \in K_x$. Clearly, then, small overhead means that ε should be small, construction time depends on how fast the system can be solved, and query time depends on the size and structure of the sets K_x. A popular choice for the group, also used in this article, is $R = \{0, 1\}^r$ for some $r \geq 1$, with bitwise XOR as group operation.[3] Let then $\vec{h}(x) \in \{0, 1\}^m$ be the characteristic vector of K_x and $Z \in \{0, 1\}^{m \times r}$

[1]Curiously, a construction for erasure correcting codes [47] can also be interpreted as an algorithm for a retrieval data structure.
[2]In this article, $[k]$ stands for $\{1, \ldots, k\}$ (or sometimes $\{0, \ldots, k-1\}$, depending on the context), and $j..k$ stands for $\{j, j+1, \ldots, k\}$.
[3]The usage of $R = \mathbb{Z}_3$ in [33] is the only exception in the literature that comes to mind.

the matrix whose rows are the Z_i. Then, system (1) can be written as:

$$\vec{h}(x) \cdot Z = f(x), \text{ for } x \in S \qquad \text{or more compactly} \qquad A_S \cdot Z = F_S, \tag{2}$$

where $A_S \in \{0,1\}^{n \times m}$ has $\vec{h}(x)$, $x \in S$, as rows and $F_S \in \{0,1\}^{n \times r}$ has $f(x), x \in S$ as rows (with matching ordering). Note that in (2), the equation really comprises r linear systems with the same matrix A_S, written in parallel. Briefly, the following are the main known methods for choosing K_x resp. $\vec{h}(x)$ that render it likely that a solution Z for (2) exists and can be computed quickly.

(i) Let K_x be a random subset of $[m]$ of some fixed size k [10, 11, 14, 16, 20, 33, 60] or of some fixed expected size k [47]. This gives good control over the (expected) lookup time. The simple case is if matrix A_S can be transformed into row echelon form by row and column *exchanges* alone. Such a system can be solved in time $O(nk)$ by constructing such a reordering and subsequent back-substitution, even if R is an arbitrary Abelian group.

(ii) If (i) does not apply, we have to assume that $R = \mathbb{F}^r$ for a finite field $\mathbb{F}$ and work in the realm of linear algebra over $\mathbb{F}$. (In this article, we only consider $\mathbb{F} = \mathbb{Z}_2$.) A sufficient condition for (2) to be solvable is that A_S has full row rank; then it can be solved by standard methods like Gaussian elimination. In general this could cause construction times up to $O(n^3)$. Even using solving methods that take advantage of the fact that A_S is sparse [61] can improve this only up to $O(kn^2)$, which is impractical for larger n.

(iii) "Sharding": In many situations concerning data structures for sets or functions (see, e.g., [20, 22, 33, 36]) it is useful to split the domain S into pieces by some hash function $h_0 : \mathcal{U} \to [t]$ for some t. This means: Form sets $S_j = S \cap h_0^{-1}(\{j\})$, for $j \in [t]$, and treat these sets ("*shards*") in separate data structures. For example, by using shards of size $C = n^{\Omega(1)}$, the effect of high costs for Gaussian elimination as in (ii) can be mitigated and overall construction times of $O(C^2 n)$ can be achieved.

(iv) As we concentrate on solvability of (2) anyway, it is no longer necessary to think of $\vec{h}(x)$ representing a set K_x. Rather, $\vec{h}(x)$ can be any binary vector chosen from a suitable distribution. This approach was taken in [22], where $\vec{h}(x)$ is built by placing two blocks of $O(\log n)$ random bits in an otherwise empty m-vector. The starting point of the present paper is to try the same with one block, which makes the system of equations much easier to solve.

(v) "Table lookup": If sharding like in (iii) is pushed even further, to shards of sub-logarithmic size, it is possible to employ lookup tables for solving the remaining tiny linear systems, and for processing queries. In the context of retrieval, this has been used in [20, 52]. Because the table size m_j for shard S_j is tiny, this also removes all restrictions on the vectors $\vec{h}(x)$—they might even be fully random. Unfortunately, approaches of this kind do not seem to be too useful in practice.

In this article, we advance the linear algebra approach to the point where space overhead is almost eliminated while keeping or improving the running times of previous constructions.

1.2 First New Idea: One Short Block of Random Bits

From here on, the range is $R = \{0,1\}^r$, and we work over the field $\mathbb{Z}_2$. The basic idea original to the work in this article, first proposed in [24], is to choose the vector $\vec{h}(x)$, for $x \in \mathcal{U}$, as follows: For a parameter w, a block $c(x) \in \{0,1\}^w$ of w bits and a starting position $s(x) \in [m - w + 1]$ are chosen as pseudo-random hash values, and $\vec{h}(x)$ is the vector obtained by placing $c(x)$ in positions $s(x)..s(x) + w - 1$ of a bit vector of length m, with zeros outside the block. We define $A = A_S = (\vec{h}(x))_{x \in S}$ and $F = F_S = (f(x))_{x \in S}$ as before in (2), but now insist that the rows of A_S and F_S should appear in increasing order of starting positions $s(x)$ of the underlying keys (breaking ties according to some arbitrary order on S).

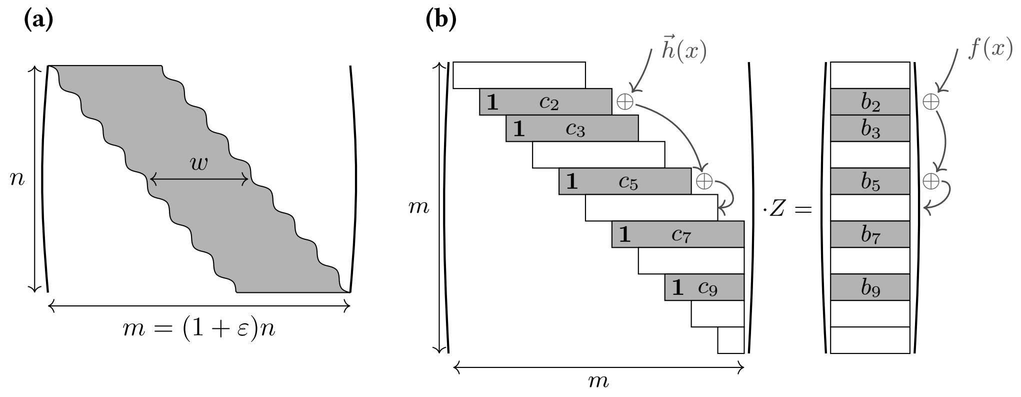

Fig. 1. **(a)** Typical shape of the random matrix A with rows $(\vec{h}(x))_{x \in S}$ sorted by starting positions. The shaded "ribbon" region contains random bits. Gaussian elimination never causes any fill-in outside of the ribbon. **(b)** Shape of the linear system M in row echelon form maintained using Boolean banding on the fly. We visualize the insertion of a key x assuming (i) $\vec{h}(x)$ has its leftmost 1 in position $s(x) = 2$, (ii) after xoring the second row of M to $\vec{h}(x)$, the leftmost 1 is in position 5 and (iii) xoring the fifth row as well, the leftmost 1 is in position 6. The resulting row fills the previously empty sixth row of M and $f(x) \oplus b_2 \oplus b_5$ is added as the sixth row on the right hand side.

While matrix A is then not technically a band matrix, with high probability its non-zeroes lie in a "ribbon" close to a generalized diagonal of (falling) slope $n/m = 1/(1+\varepsilon)$, with a width around w, see Figure 1 (a). This is why we call w the *ribbon width*. Our analysis will identify sufficient conditions for the "ribbon matrix" A to have full row rank with high probability, and it will show that if the rank is full then a solution can be found fast, with high probability. A natural solution method is Gaussian elimination, processing the rows of $A \cdot Z = F$ from top to bottom, i.e., in order of increasing starting position $s(x)$. (This was first described in [24].) This approach obviously prevents a "proliferation of 1s" in the matrix and ensures a swift construction overall.

A variant of this method for solving (2) is called *Ribbon* (for ***R**apid **I**ncremental **B**oolean **B**anding **ON** the fly*). This algorithm, which was first described in [27], is the basis for all constructions discussed in the following. It builds a system equivalent to (2), by iteratively adding one equation from (2) after the other to an initially empty system, in arbitrary order. The system matrix is always quadratic and in echelon form, and for each row of (2), that is processed at most one row in the system is changed. For this algorithm it is convenient to assume that each block $c(x)$ starts with a 1. For an example, see Figure 1 (b).[4] The data structure resulting from this first, simple algorithm is not meant to be succinct, but it lays the ground for the succinct constructions that follow. We call it *Standard Ribbon Retrieval*, and its properties are given in the following theorem.[5]

Theorem 1.5. *For any $\varepsilon > 0$, an r-bit Standard Ribbon Retrieval data structure with ribbon width $w = \frac{\log n}{\varepsilon}$ has expected construction time $O(n/\varepsilon^2)$, query time $O(r/\varepsilon)$ and overhead $O(\varepsilon)$.*

The proof involves techniques as would be used in the analysis of linear probing.[6] We also profit from some results from queueing theory, specifically regarding so-called M/D/1-queues.

[4]We try to capture this kind of behavior by the attribute "on-the-fly". Note, however, that after constructing the full system matrix in echelon form a subsequent round of back-substitution is needed to arrive at the final solution $Z_1, \ldots, Z_m$.
[5]It should be noted that for all theoretical analyses in this article it is assumed that computations are on a word RAM with word size $O(\log n)$ and that hash functions behave like random functions. This is a standard assumption in many papers and can also be justified by standard constructions [21].
[6]For details, see [24] where we have established a tight connection to Robin Hood hashing.

The quantity ε is a parameter that can be given to the construction algorithm as an input, and which describes a tradeoff between overhead and execution times for construction and queries. A problem with Standard Ribbon Retrieval is that our implicit hope that w does not exceed the machine word size is not fulfilled when n is large and ε is small, not only when looking at the theoretical bound $\frac{\log n}{\varepsilon}$, but also in practice. For example, experiments show that for $\varepsilon \leq 3.5\%$ and construction success rate $\geq 50\%$, standard word size $w = 64$ only scales to around $n \leq 10^4$ and a more expensive value $w = 128$ only scales to around $n \leq 10^6$. To some degree this can be mitigated by sharding techniques [59], but in this article we pursue a more ambitious route, described next.

1.3 Second New Idea: (Bumping and) Overloading

A classical strategy for building data structures that store keys in some sense, like dictionaries, retrieval data structures, or AMQ data structures, is the use of *backyard* structures. Under this strategy, keys that cannot be accommodated in the main data structure are "kicked out" (or "*bumped*") and put into a secondary, or "backyard" structure.[7] This may be the same kind of structure, with the method applied recursively, or some simpler structure with a worse overhead. The name is from [1], but the strategy is much older, see, e.g., [13, 20, 30, 51, 52]. In the context of Standard Ribbon, the straightforward strategy would be to bump keys x whose row $\vec{h}(x)$ is linearly dependent of rows handled previously. In the analysis of the overhead one now has to take into account that space is needed (i) for the backyard structure and (ii) for auxiliary information ("*metadata*") indicating which keys have been moved to the backyard. The naive strategy just outlined requires a lot of metadata. A subtler method is more successful, namely bumping keys in chunks of carefully chosen size that have a compact description. This directly aims at reducing the metadata.

Now a new idea comes into play, namely *overloading*. As long as the $n \times m$ matrix in (2) has to have full row rank, there is no way around the restriction $m \geq n$.[8] But with a backyard structure in place it is possible to choose m slightly smaller than n, thus *overloading* the primary data structure.[9] Surprisingly, this trick makes it possible to pack keys so densely in the primary data structure that even with the additional space for the metadata, the overall overhead is reduced.

Thus, a convenient subset $S' \subseteq S$ of keys is handled in the first *layer* of the data structure and the small rest is *bumped*[10] to the backyard. The resulting ***Bumped Ribbon Retrieval*** (**BuRR**) data structure has much smaller overhead than the single-layer version from Theorem 1.5. The result is given in the following theorem. It is the main technical result of this article, and its proof is far from trivial.

Theorem 1.6. *An r-bit BuRR data structure with ribbon width $w = O(\log n)$ and $r = O(w)$ has space overhead $O(\frac{\log w}{rw^2})$, query time $O(1 + \frac{rw}{\log n})$, and expected construction time $O(nw)$.*

To gain some intuition regarding the linear systems relevant in the analysis of this algorithm it is useful to consider two simplifications. First, assume that $\vec{h}(x)$ contains a block of w uniformly random *real* numbers from $[0, 1]$ rather than w random bits. Secondly, assume that we sort the rows by starting position and use Gaussian elimination rather than the construction used in Standard Ribbon Retrieval to produce a row echelon form. The setting is illustrated in Figure 2 **(a)**. The $\times$-marks are the positions where the pivots will be placed and the yellow entries will be eliminated

[7]In this situation the array containing $Z_1, \ldots, Z_m$ together with the hash function $\vec{h}$ is called "primary structure".
[8]Actually, to have any reasonable chance of obtaining a solvable system in the one-block setting, extra space $m - n = \varepsilon n = \Omega((n \log n)/w)$ is unavoidable.
[9]Algorithm engineering worked nicely here: this idea was conceived on the basis of observations made in experiments.
[10]We adopt the term "*bumping*" from the airline industry, which achieves better utilization of planes by overbooking, i.e., by selling more tickets for a flight than there are seats on the plane. This usually works because not all passengers show up for a flight, and if more passengers show up than seats are available, some of them will be *bumped* to later flights.

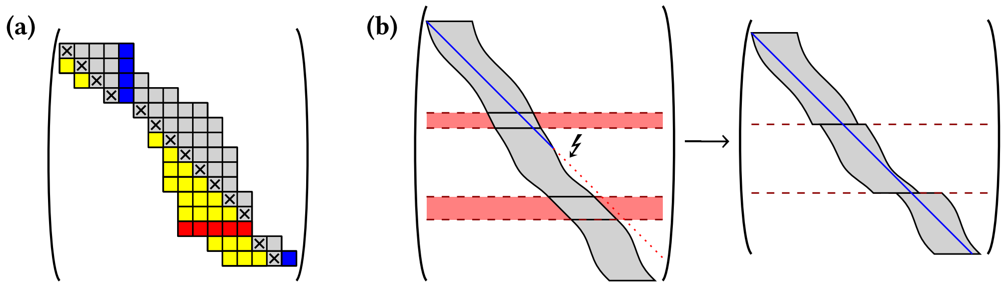

Fig. 2. **(a)** The simplified ribbon diagonal (made up of ×-marks) passing through A.
(b) The idea of BuRR: When starting with an "overloaded" linear system and removing sets of rows strategically, we can often ensure that the ribbon diagonal does not collide with the ribbon border (except possibly at the beginning and at the end).

with one row operation each. Note that while technically a row operation could eliminate more than one entry at the same time, which would lead to a different outcome, the probability of that happening with random real numbers is zero. The ×-marks trace a diagonal through the matrix except that the blue columns and the red row are skipped because the end of the (gray) area of non-zeroes is reached. "Column failures" correspond to free variables and therefore unused space. "Row failures" correspond to linearly dependent equations and therefore failed insertions. This view remains largely intact when handling *Boolean* equations in *arbitrary* order except that the *ribbon diagonal*, which we introduce as an analogue to the trace of pivot positions, has a more abstract meaning and probabilistically incurs row and column failures depending on its *distance* to the ribbon border.

The idea of Standard Ribbon is to give the gray ribbon an expected slope of less than 1 such that row failures are unlikely. BuRR, as illustrated in Figure 2 **(b)**, largely avoids both failure types by using a slope bigger than 1 but removing ranges of rows in strategic positions. We mention in passing that Homogeneous Ribbon Filters (to be sketched in Section 1.5), despite being the simplest of our ribbon-based constructions, have the subtlest analysis (Section 5) as both failure types are allowed to occur. While row failures cannot cause outright construction failure, they are linked to a compromised false positive rate in a non-trivial way.

Note that the ribbon width w is only restricted to be $O(\log n)$ and $\Omega(r)$ in Theorem 2, which means that it describes a tradeoff between overhead and running times in quite a large range. Thus, w can serve as a *tuning parameter* to try and get close to optimal points on that tradeoff curve. If r happens to be too large in comparison with w, one may imagine to use r/w independent copies of the data structure, which results in running times increased by a factor of r/w. In practice, only array Z is given more columns.

On the basis of the BuRR construction we may go another step in a purely theoretical direction, giving up the simplicity, and probably the practicability. In order to obtain a result that beats the best construction for static retrieval up to now from [6], we additionally employ (standard) techniques such as the use of small precomputed tables.

Corollary 1.7 (Restated Theorem 1.3). *There is a variant of BuRR for $r = O(\sqrt{\log n})$ with constant query time, linear expected construction time, and overhead $O(\frac{\log\log n}{r\log n})$.*

1.4 Previous Work on Static Retrieval

For a quick overview, Table 1 summarizes performance parameters of some (not all) previous constructions and of the new constructions of static retrieval data structures. Retrieval data

Table 1. Performance of Various Static r-bit Retrieval Data Structures

	Year	$t_{\text{construct}}$	t_{query}	overhead	shard size	Solver
[47]	2001	$O(nlogk)$	$O(logk)^{\dagger}$	$\frac{1}{k}$	–	peeling
[52]	2009	$O(n)$	$O(1)$	$\boldsymbol{O(\frac{\log\log n}{(\log n)^{1/2}})}$	$\sqrt{\log n}$	lookup table
[11]	2013	$O(n)$	$O(1)$	0.2218	–	peeling
[6]	2013	$O(n)$	$O(1)$	$\boldsymbol{O(\frac{\log^2\log n}{r\log n})}$	$O(\frac{\log^2\log n}{r\log n})$	–
[32]	2016	$O(nC^2)$	$O(1)$	$0.024 + O(\frac{\log n}{C})$	C	structured Gauss
[22]	2019	$O(nC^2)$	$O(r)$	$\boldsymbol{O(\frac{\log n}{C})}$	C	structured Gauss
[60]	2021	$O(nk)$	$O(k)$	$(1+o_k(1))e^{-k}$	–	peeling
Standard Ribbon		$O(n/\varepsilon^2)$	$O(r/\varepsilon)$	ε	–	on-the-fly Gauss
BuRR		$O(nw)$	$O(1+\frac{rw}{\log n})$	$\boldsymbol{O(\frac{\log w}{rw^2})}$	–	on-the-fly Gauss
BuRR (w=$\Theta(\log n)$)		$O(n\log n)$	$O(r)$	$\boldsymbol{O(\frac{\log\log n}{r\log^2 n})}$	–	on-the-fly Gauss
BuRR (w=$\Theta(\sqrt{\log n})$)		$O(n)$	$O(1)$	$\boldsymbol{O(\frac{\log\log n}{r\log n})}$	–	Gauss+lookup table

†Expected query time. The worst case query time is $O(k)$.
The last line assumes $r = O(\sqrt{\log n})$, the line for [6] assumes $r = O(\log\log n)$, other assume $r = O(\log n)$. Expressions for the overhead written in bold indicate that the data structure is (or can be configured to be) succinct with query time $O(1)$ or $O(r)$. The parameters $k \in \mathbb{N}$ and $\varepsilon > 0$ are constants with respect to n. The parameter $C \in \mathbb{N}$ is typically n^{α} for constant $\alpha \in (0,1)$.

structures that are *succinct*, i.e., can achieve overhead $o(1)$, while having query time $O(1)$ or $O(r)$ have their overhead written in bold. Some of the constructions are mentioned in items (i) through (v) in Section 1.1.

Paper [47] by Luby *et al.* treats a problem in coding theory that leads to the same system of equations as our retrieval problem; it solves this system by row and column exchanges (item (i) in 1.1). This is called "peeling" since equations are removed one after the other without any arithmetic. Porat [52] uses sharding and table-lookup techniques (items (iii) and (v) in 1.1) to build a succinct retrieval data structure aimed at implementing a Bloom filter replacement. Bothelo *et al.* [11] use a retrieval construction according to item (i) with $k = 3$ for constructing MPHF, which makes it impossible to achieve an overhead below 0.2217. The query times are very good in practice. Belazzougui and Venturini [6] give a construction with the hitherto smallest asymptotic overhead; they also use sharding and table-lookup. Genuzio *et al.* [32], in a practical study, use sharding, clever linear algebra techniques, and bit-level parallelism to reach practically very good solution techniques for the linear systems obtained with $k = 4$ according to item (ii). The best overhead achievable in this way is about 0.024. Paper [22] is discussed in item (iv); this construction can achieve overhead $n^{-\delta}$ for a constant $\delta > 0$, at the price of a quite high construction time; for the variant involving C see [22, Corollary 2]. Finally, paper [60] uses the approach of item (i) with a carefully chosen distribution ("spatial coupling") on the k-subsets of $[m]$. This makes it possible to use peeling with a much smaller overhead than with fully random subsets.

We shall discuss later what the advantages of tuning by suitable parameters such as C and w are, and which constructions promise good behavior in practical applications.

A Lower Bound. The only approaches that achieve construction time $O(n)$, query time $O(1)$ and multiplicative overhead $o(1/r)$ (i.e., additive overhead $o(n)$ bits) restrict r to values of $o(\log n)$. This has recently been demonstrated to be necessary [38].

Remarks on Value-dynamic and Dynamic Retrieval. In the table we omit constructions based on perfect hashing. Such approaches implement the more general problem of *value-dynamic retrieval* where the key set is static but values can be changed. When used as static retrieval data structures they necessarily have an additive space overhead of $\Omega(n)$ bits [43]. Several perfect hashing techniques would give rise to a row in the table with overhead $O(1/r)$, construction time $O(n)$ and query time $O(1)$, e.g., [36, 50].

Remark on Fully Dynamic Retrieval. Fully dynamic retrieval data structures support insertions and deletions of keys-value pairs. They require $\Omega(n \log\log \frac{u}{n} + nr)$ bits of memory where u is the size of the universe [18].

1.5 A New Data Structure for Approximate Membership

One of the central applications of retrieval data structures is that they immediately lead to *AMQ data structures* (alternative names: *Bloom filter replacement*, *AMQ filter*, or simply *filter*). AMQ filters are randomized data structures that support membership queries for a subset S of $\mathcal{U}$, with some *false positive rate* φ: the answer to a query on some $x \in S$ must be true, for $x \notin S$ we have $\Pr[\text{answer on } x \text{ is true}] \leq \varphi$. An r-bit retrieval data structure $\mathcal{R}$ immediately yields an AMQ with false positive rate 2^{-r} as follows [20]: associate an r-bit random fingerprint $f(x)$ with each key x from $\mathcal{U}$, and store the fingerprints for all $x \in S$ in $\mathcal{R}$; the answer on query x from $\mathcal{U}$ is the truth value $[\text{query}_{\mathcal{R}}(x) = f(x)]$. Thus, Standard Ribbon Retrieval and BuRR immediately yield new implementations for AMQ filters. If (as we assume) fingerprint functions are available for free, the space requirements and the running times for the operations are the same as for the retrieval data structure.

Beyond this standard application of retrieval structures to obtain AMQ filters we will describe and analyze another, even simpler construction, called *Homogeneous Ribbon Filter*, which is not directly based on retrieval and does not employ a fingerprint function. Instead, it uses the following *homogeneous* form of the linear system from Equation (2):

$$\vec{h}(x) \cdot Z = 0^r, \text{ for } x \in S \qquad \text{or more compactly} \qquad A_S \cdot Z = 0^{n \times r}.$$

The system is underdetermined. We show that for a uniformly random solution Z the probability that $\vec{h}(x) \cdot Z = 0$ for $x \in \mathcal{U} \setminus S$ is close to 2^{-r}. We obtain the following result.

THEOREM 1.8. *There exists a constant C such that for arbitrary $r \geq 1$, $\varepsilon \in (0, 1/2]$, for $w = \frac{C \max(r, \log(1/\varepsilon))}{\varepsilon}$ and for any n with $n = \omega(w/\varepsilon)$ and $\varepsilon w = O(\log n)$ the following holds. The Homogeneous Ribbon Filter with ribbon width w and r-bit fingerprints for n keys has space overhead ε and false positive probability $(1 + O(\varepsilon^2))2^{-r}$. On a word RAM with word size at least w the expected construction time is $O(n/\varepsilon)$ and the query time is $O(r)$.*[11]

1.6 Practical Aspects, Tunability, and Experimental Evaluation

The practical performance of our new data structures reflects the good asymptotic behavior asserted by our theorems, i.e., the hidden constants are comparatively small. We demonstrate this experimentally with a rich set of benchmarks of implementations of AMQ data structures, encompassing different approaches from the literature and variants based on BuRR and on Homogeneous Filters. In case of filters that are based on retrieval, the retrieval performance can be inferred from the filter performance.

[11]For w larger than the word size W the running times increase by a factor of w/W, by standard arguments.

Main Results. Which approach is the fastest depends on the permitted space overhead ε. Roughly speaking, BuRR is fastest for $\varepsilon < 4\%$ and Bloom filters and variants are fastest for $\varepsilon > 40\%$. For most intermediate ε, Homogeneous Ribbon has the fastest construction time and XOR-filters and coupled filters have the fastest queries.[12]

Succinctness is not Enough. A data structure is succinct if its space overhead is $o(1)$, i.e., the overhead vanishes as n increases. But *tuning* the space overhead at fixed values of n can also be useful. BuRR involves such a tuning parameter, namely the ribbon width w that trades smaller overhead for increased running time. This is relevant both in theoretical considerations and in practical evaluation. Previous succinct solutions [6, 52] are not tunable and only *barely* succinct with significant overhead in practice. A quick calculation to illustrate: Neglecting the factors hidden by the O-notation, the overheads are $\frac{\log\log n}{(\log n)^{1/2}}$ and $\frac{\log^2\log n}{r\log n}$, which is at least 75% and 7% for $r = 8$ and any $n \le 2^{64}$. A similar estimation for BuRR with $w = \Theta(\log n)$ suggests an overhead of $\frac{\log\log n}{r\log^2 n}$ $\approx 0.1\%$ already for $r = 8$ and $n = 2^{24}$. Only few other constructions have a tunable space overhead, see Table 1. While [22] uses a parameter $C = \omega(\log n)$, this particular approach suffers from comparatively high construction time. Other approaches [47, 60] are tunable via a parameter k but the impact of k on performance is less favorable than for BuRR. In particular, locality of data access in queries deteriorates with increasing k, while for BuRR the data access patterns remain highly local. Refer to Section 1.8 for a more detailed discussion.

Our impression is that the asymptotic property of succinctness alone is lacking in predictive power with respect to the actual space efficiency of a data structure—tunability and a more detailed consideration of the involved tradeoffs seems to be required. Eventually, it is hard to evade analysis of constant factors and/or experiments to come to meaningful results.

The Importance of Tunability. Especially in AMQ applications, retrieval data structures occupy a considerable fraction of RAM in large server farms continuously drawing many megawatts of power. Even small reductions (say 10 %) in their space consumption thus translate into considerable cost savings. Whether or not these space savings should be pursued at the price of increased access costs depends on the frequency of queries. The lower the access frequency, the more worthwhile it is to occasionally spend increased access costs for a permanently lowered memory budget. Since the false-positive rate also has an associated cost (e.g., additional accesses to disk or flash) it is also subject to tuning. The entire set of Pareto-optimal variants with respect to tradeoffs between space, access time, and FP rate is relevant for applications. For instance, sophisticated implementations of LSM-trees use multiple variants of AMQs at once based on known access frequencies [17]. Similar ideas have been used in compressed data bases [49].

1.7 Variants of BuRR

There are several further variants of BuRR that we have considered but not rigorously analyzed, though in some cases we have experimental results. Perhaps most interesting is *bump-once ribbon retrieval (Bu^{1}RR)*, which improves the worst-case query time by guaranteeing that each key can be retrieved from one out of two layers: its *primary layer* or the next one. The primary layer of a keys is a random variable distributed over all layers except for the last. When building the data structure for a layer, the keys bumped from the previous layer are inserted into the system matrix M first, without bumping. Only then the keys with the current layer as their primary layer are inserted, now allowing bumping. The size of the retrieval structures for the layers and the distribution of the primary layer have to be chosen in such a way that on the one hand no bumping is needed for

[12]We remark that a recent preprint [58] improves the construction time of XOR and coupled filters, which changes the picture somewhat.

already bumped keys with high probability and on the other hand that each layer is overloaded. See Section 7.6 for details.

For building large retrieval data structures, *parallel and external memory construction* is important. Doing this naively is difficult for ribbon retrieval since there is no efficient way to parallelize back-substitutions. However, we can partition the equation system into parts that can be solved independently by bumping w consecutive positions at certain positions. If one employs the bumping mechanism, that is present anyway, one can even keep the standard query algorithm. See Section 7.3 for details.

For large r, we may accelerate queries by working with *sparse bit patterns* that set only a small fraction of the w bits in the window used for BuRR. In some sense, we are covering here the middle ground between ribbon and spatial coupling [60]. Experiments indicate that setting 8 out of 64 bits indeed speeds up queries for $r \in \{8, 16\}$ at the price of increased (but still small) overhead. Analysis and further exploration of this middle ground may be an interesting area for future work. See Section 7.5 for details.

1.8 Related Results and Techniques

We mention here some more problem types, results, and techniques relevant in the context of retrieval.

Related Problems. An important application of retrieval data structures besides AMQs (see Section 1.5) is for encoding **perfect hash function (PHF)**, i.e., an injective function $p : S \longrightarrow [(1+\delta)|S|]$ for given $S \subseteq \mathcal{U}$ and some $\delta \geq 0$. Objectives for p are compact encoding, fast evaluation and small δ. A special case of a central result from the area of cuckoo hashing [19, 30, 31, 46] is that given four hash functions $h_1, h_2, h_3, h_4 \colon S \longrightarrow [1.024|S|]$ there exists, with high probability, a choice function $f \colon S \longrightarrow [4]$ such that $x \mapsto h_{f(x)}(x)$ is injective. A 2-bit retrieval data structure for f therefore can be used to represent a PHF with $\delta \approx 0.024$ [11]. Refinements of this basic idea include some of the best known approaches to perfect hashing [44, 45], which profit from the high speed and near optimal space of BuRR.

Retrieval data structures can also be used to directly store compact names of objects, e.g., in column-oriented databases [50]. This takes more space than perfect hashing but makes it possible to encode the ordering of the keys into the names.

In retrieval for AMQs and PHFs the stored values $f(x) \in \{0,1\}^r$ are assumed to be uniformly random. (This assumption is made throughout the present paper.) Work on *compressed static functions* [6, 33, 37] explored the situation where f has a skewed distribution. Then, the space can be reduced to below nr bits. The overhead of the retrieval data structure is measured with respect to the 0*th order empirical entropy* of f. A natural reduction to 1-bit retrieval works as follows. Pick an optimal prefix-free variable-bit-length encoding $\text{enc} \colon \{0,1\}^r \longrightarrow \{0,1\}^*$ of the values, such as a Huffman code. Then, store the kth bit of $\text{enc}(f(x))$ as data to be retrieved for the input tuple (x, k). This can be further improved by storing R 1-bit retrieval data structures where $R = \max_{x \in S} |\text{enc}(f(x))|$. By interleaving these data structures, one can make queries almost as fast as in the case of fixed r. A different approach to dealing with skewed distributions in the context of AMQ data structures is the use of machine learning techniques [57].

We end this section by recalling some otherwise very helpful techniques and giving reasons why we choose not to use them in this article. Some more details on methods used in the experiments are also discussed in Section 8.

Shards. As mentioned in Section 1.1, in data structures for sets and functions often a splitting hash function is used to first divide the input set into smaller sets (called shards or chunks), which can then be handled separately [2, 5, 22, 24, 33, 52]. Direct use of this technique in combination with Standard Ribbon, say, did not yield convincing results. The overhead could not be reduced below $\Omega(1/w)$. The much better overhead of $O\big((\log w)/w^2\big)$ achieved with BuRR comes from using the sharding technique in a "soft" way—keys are assigned to buckets for the purpose of defining bumping information but the ribbon solver may implicitly allocate them to subsequent buckets.

Table Lookup. The first asymptotically efficient succinct retrieval data structure we are aware of [52] uses two levels of sharding to obtain very small shards of size $O(\sqrt{\log n})$ with small asymptotic overhead. It then uses dense random matrices per shard to obtain per-shard retrieval data structures. This can be done in constant time per shard by tabulating the solutions of all possible matrices. This leads to an overhead of $O((\log\log n)/\sqrt{\log n})$. Belazzougui and Venturini [6] use slightly larger shards of size $O((1 + (\log\log n)/r)(\log\log n)/\log n)$. Using carefully designed random lookup tables they show that linear construction time, constant lookup time, and overhead $O\big((\log\log n)^2/\log n\big)$ is possible. We discussed in Section 1.6 why we suspect large overhead for [52] and [6] in practice.

In general, lookup tables are often problematic for compressed data structures in practice, because they cause additional space overhead and cache faults. Even if the table is small and fits into cache, this may yield efficient benchmarks but can still cause cache faults in practical workloads where the data structure is only a small part in a large software system with a large working set.

Cascaded Bumping. Hash tables consisting of multiple shrinking levels are also used in *multilevel adaptive hashing* [13] and *filter hashing* [30]. While similar to BuRR in this sense, they do not maintain bumping information. This is fine for storing key-value pairs because all levels can be searched for a requested key. But it is unclear how the idea would work in the context of retrieval, i.e., without storing keys.

2 Ribbon Insertions

In this section, we describe a linear system solver called *Ribbon* (**R**apid **I**ncremental **B**oolean **B**anding **ON** the fly). It forms the basis for all retrieval and filter data structures we present later.

Ribbon Matrices. The task is to solve a linear system with n equations and m variables, whose right-hand side arises from a function $f\colon S \to \{0,1\}^r$ to be stored, for a set $S \subseteq \mathcal{U}$ of n keys, and whose system matrix A_S is generated by a random experiment, as follows. A parameter $w \in \mathbb{N}$, which we call the *ribbon width*, is given, and two hash functions $s\colon \mathcal{U} \to [m-w+1]$ and $c\colon \mathcal{U} \to \{1\} \times \{0,1\}^{w-1}$ are chosen at random. For $x \in S$ let $\vec{h}(x) = 0^{s(x)-1} \circ c(x) \circ 0^{m-s(x)-w+1} \in \{0,1\}^m$. We call $s(x) \in [m-w+1]$ the *starting position* of x and $c(x)$ the *coefficient vector* of x; it always starts with 1. Note that even though m-bit vectors like $\vec{h}(x)$ are used to simplify mathematical discussion, such vectors can be represented using $\log(m) + w$ bits. The matrix A obtained from $A_S = (\vec{h}(x))_{x\in S}$ by sorting the rows in increasing order of starting positions $s(x)$ (ties are broken by some fixed order on S) has all of its 1-entries in a "ribbon" of width w that randomly passes through the matrix from the top left to the bottom right, as in Figure 1(a). Note that while A has similarities with a band matrix, the ribbon is close to the matrix diagonal only in a probabilistic sense.

Now consider the task of solving $(\vec{h}(x) \cdot Z = f(x))_{x\in S}$ for $Z \in \{0,1\}^{m\times r}$. The equations are processed in arbitrary order.

Boolean Banding on the Fly. The *insertion phase* transforms the given system of equations into row echelon form. The idea of the method, for arbitrary low-density matrices, was described and called *on-the-fly Gaussian elimination* in [8]. It maintains a system M of linear equations in row echelon form as shown in Figure 1(b). Equations are processed one after the other, and each equation is incorporated into M before the next one arrives. System M has m rows, which we also call

ALGORITHM 1: Adding equation $\vec{h}(x) \cdot Z = f(x)$ for key x to the linear system M.

```
(i, c, b) ← (s(x), c(x), f(x))
loop
    if $M.c[i] = 0$ then // slot $i$ of $M$ is empty
        $(M.c[i], M.b[i]) \leftarrow (c, b)$
        return SUCCESS
    $(c, b) \leftarrow (c \oplus M.c[i], b \oplus M.b[i])$
    if $c = 0$ then
        if $b = 0$ then return REDUNDANT
        else return FAILURE
    $j \leftarrow$ number of leading zeroes of $c$
    $i \leftarrow i + j$
    $c \leftarrow c \ll j$ // logical shift to the left by $j$ bits
```

slots. The ith slot contains a w-bit vector $c_i \in \{0,1\}^w$ and an r-bit vector $b_i \in \{0,1\}^r$. Technically, these two components are stored in fields $M.c[i]$ and $M.b[i]$. Conceptually, the ith slot is either empty ($c_i = 0^w$) or it specifies a linear equation $c_i \cdot Z_{[i,i+w)} = b_i$, where c_i starts with a 1. (With $Z_{[i,i+w)} \in \{0,1\}^{w \times r}$ we refer to rows $i, \dots, i+w-1$ of $Z \in \{0,1\}^{m \times r}$. We ensure that $c_i \cdot Z_{[i,i+w)}$ is well-defined even when $i + w - 1 > m$ by observing as an invariant that components of c_i that would select "out of bounds" rows of Z are always 0.)

The details are as follows. We initialize M as the empty system. Then, we consider the equations $\vec{h}(x) \cdot Z = f(x)$ for $x \in S$ one by one, *in arbitrary order* (as is required in Section 6). The round for x calls Algorithm 1 (illustrated in Figure 1 (b)) to try and generate an equation equivalent to $\vec{h}(x) \cdot Z = f(x)$ that can be inserted into M. We explain Algorithm 1 now. Recall that $\vec{h}(x)$ is determined by $s(x)$ and $c(x)$, which starts with a 1. Thus, x's equation is given by $(i, c, b) = (s(x), c(x), f(x))$. If slot i is empty, we can insert (c, b) into this slot. If not, the slot contains another equation, described by $(M.c[i], M.b[i])$. We add $(M.c[i], M.b[i])$ to (c, b). This results in a new pair (c', b') of vectors, where c' has w bits and begins with a 0. We get another description of the same equation by shifting c' to the left by j bits (result c'') and choosing as new starting point $i' = i + j$, where j is the number of leading zeroes in c. Now we try to insert the new triple (i', c'', b') by repeating this procedure, and iterate, if necessary. Once an empty slot is found, the insertion finishes *successfully*. There are two possible irregular outcomes. If at some point the components c'' and b' in the new triple (i', c'', b') are both zero, the newly added equation is *redundant*; it is a linear combination of equations already present in M. If, on the other hand, $c'' = 0$ and $b' \neq 0$, then the newly added equation introduces a contradiction, the system is unsolvable. In this case, we declare the solution attempt a *failure*.[13]

A key's equation may be modified several times before it can be added to M. It is an invariant that the solution space of the system of equations already present in M together with

$$c \cdot Z_{[i,i+w)} = b, \quad \text{for } c \in \{1\} \times \{0,1\}^{w-1}, b \in \{0,1\}^r, \tag{3}$$

(where the jth entry of c is zero if $i + j > m$, i.e., if it is "out of bounds")

always stays the same in the course of these modifications. Note that the initial equation $\vec{h}(x) \cdot Z = f(x)$ of key $x \in S$ satisfies this invariant, with $i = s(x)$, $c = c(x)$ and $b = f(x)$. Termination of

[13]In the context of homogeneous filters (Section 5) the right hand side $(f(x))_{x \in S}$ will be the zero vector. In this case, of course, failures are impossible.

Algorithm 1 is guaranteed since the "looping" case increases the position i of the leftmost 1 entry, which cannot exceed m.

When all rows have been processed successfully (or turned out to be redundant), in the *back-substitution* phase we calculate a solution $Z \in \{0,1\}^{m \times r}$ for the system of equations from the result of Algorithm 1 in the standard way, in one bottom-to-top pass.

"On-the-Fly" and "Incremental". The fact that the insertion phase of Ribbon works *on the fly*, i.e., maintains a row echelon form as keys arrive, allows us to determine the longest prefix $(x_1, \ldots, x_\ell)$ of a sequence $S = (x_1, x_2, x_3, \ldots)$ of keys for which the construction succeeds: Simply insert keys until the first failure. We say the insertion phase is *incremental* since an insertion may lead to a new row in M but does not modify existing ones. This allows us to easily undo the most recent successful insertions by clearing the slots of M that were filled last. These properties will be exploited by BuRR in Section 6, as will be the fact that the insertion order is arbitrary.

3 The Analysis of Ribbon Insertions: Groundwork

Let a set S of n keys and a function $f\colon S \to \{0,1\}^r$ be given. The functions s, c and h on S are chosen as described in Section 2. In this section, we state and prove fundamental properties of the process of inserting the equations in $(\vec{h}(x) \cdot Z = f(x))_{x \in S}$ into the system M using Algorithm 1. These properties, which regard deterministic behavior and probabilities of essential events, will be used later in the analysis of all our ribbon-based data structures.

In particular, we are interested in the number of successful and failed insertions, the number of occupied slots in M, and the total running time. The analysis is simplified by considering redundant equations as failures, so that there are only successes and failures. Then, the right hand side of our linear system is irrelevant, and we can take M to refer to just the $m \times m$ matrix. Recall that $A \in \{0,1\}^{n \times m}$ contains the rows $\vec{h}(x)$ for $x \in S$ sorted by $s(x)$, see Figure 1(a). Our analysis will consider the *ribbon diagonal*, which is a line passing through A.

3.1 Warm-Up: The Simplified Ribbon Diagonal

We begin with an instructive simplification, for which we make the following two assumptions:

(M1) Keys are inserted in the order they appear in A (sorted by $s(x)$ and some order on S for breaking ties). This ensures that the insertion of key $x \in S$ fails or succeeds within the first w steps because no 1-entries can exist in M beyond column $s(x) + w - 1$.

(M2) Inserting $x \in S$ fills the first free slot $i \in [s(x), s(x) + w - 1]$ unless all of these slots are occupied, in which case the insertion fails. This ignores the role of $c(x)$.

Figure 2 **(a)** visualizes the process with an $\times$-mark in position (j, i) if the insertion of the jth key fills slot i of M.[14] These positions trace an approximately diagonal line in A from top left to bottom right, which we call the *simplified ribbon diagonal* d_{simp}. In principle, this line makes one step to the right and one step down for each key, but with two modifications, as observed next. A third observation can be read off from the picture.

(O1) If d_{simp} were to cross the bottom border of the ribbon, it skips a column (shown in blue). Column i is skipped if and only if slot i of M remains empty.

(O2) If d_{simp} were to cross the right border of the ribbon, it skips a row (shown in red). Row j is skipped if and only if the jth key is not inserted successfully.

(O3) The area enclosed between d_{simp} and the left border of the ribbon (shown in yellow) is an upper bound on the number of row operations performed during successful insertions.

[14]This insertion procedure corresponds to a well-known method for creating the key placement in Robin-Hood hashing, see, e.g., [39, Sect. 8].

3.2 The Ribbon Diagonal

The actual analysis of ribbon insertions will salvage much of the intuition from the simplified model. First, we show that assumption (M1) (though not (M2)) can be kept without loss of generality. For an adjusted definition of the ribbon diagonal, we then prove probabilistic versions of (O1), (O2) and (O3). The following notation will be useful.

— $S_i = \{x \in S \mid s(x) \leq i\}$ and $s_i = |S_i|$, for $i \in [m]$.
— $S' \subseteq S$ is the set of keys not inserted successfully. Moreover, $S'_i = S_i \cap S'$ and $s'_i = |S'_i|$.
— r_i, for $i \in [m]$, is the rank of the first i columns of A.
— P_M is the set of slots of M that end up being filled.

On (M1): The order of keys is irrelevant. Regardless of the order in which keys are handled, the eventual version of M arises from A by adding $m-n$ empty rows and then performing row operations. Since these changes do not affect ranks of sets of columns, r_i is the rank of the first i columns of the final version of M. From the form of M (see Figure 1(b)) it is clear that $i \in P_M \Leftrightarrow r_i = r_{i-1} + 1$. Therefore, the set P_M and the number $n - |P_M| = |S'|$ of failed insertions are invariant under the order in which the keys are inserted.

If the insertion of key x is successful, then the number of row operations performed for x is at most the distance from $s(x)$ to the slot $i(x) \in P_M$, that is filled. If we define $S^+ := S \setminus S'$ to be the set of successfully inserted keys then the number of row operations performed during successful insertions is bounded by:

$$\Delta^+ := \sum_{x \in S^+} (i(x) - s(x)) = \sum_{i \in P_M} i - \sum_{x \in S^+} s(x). \tag{4}$$

Unfortunately, S^+ and Δ^+ are not invariant under changes to the insertion order (although $|S^+| = |P_M|$ is). However, we have

$$\sum_{x \in S^+} s(x) = \sum_{j \in [m]} |\{x \in S^+ \mid s(x) \geq j\}| = \sum_{j \in [m]} |S^+ \cap (S \setminus S_{j-1})| = |S^+|m - \sum_{j \in [m]} |S^+ \cap S_{j-1}|.$$

The summand $|S^+ \cap S_{j-1}|$ is maximized and equal to $\text{rank}((\vec{h}(x))_{x \in S_{j-1}})$ if we insert all keys from S_{j-1} before all other keys. Since $|S^+|$ and P_M are invariant, inserting keys sorted by $s(x)$ makes Δ^+ attain its maximum value, which we call Δ. Since we are later interested only in upper bounds on Δ we can assume (M1).

Definition and Properties of d. Given (M1), which ensures that $(s'_i)_{i \in [m]}$ is well-defined, we define the *ribbon diagonal d* as the following set of matrix positions in A.

$$d = \{(d_i, i) \mid i \in [m]\} \text{ where } d_i = r_i + s'_{i-w+1}.$$

We follow the course of the sequence (d_i, i), for $i = 1, \ldots, m$, through A, which in a sense records successful insertions and failures. In the "default case" $r_i = r_{i-1} + 1$ and $s'_{i-w+1} = s'_{i-w}$ we have $d_i = d_{i-1} + 1$, and the ribbon diagonal indeed steps diagonally down and to the right. If $i \notin P_M$, i.e., slot i is empty, the corresponding downward step is left out (due to $r_i = r_{i-1}$); for keys x with $s(x) = i - w + 1$ that fail to be inserted an extra downward step is taken (due to $s'_{i-w+1} > s'_{i-w}$). For an example see Figure 3.

Note that column i of A can have non-zero bits only in rows $s_{i-w} + 1, \ldots, s_i$ (with $s_{i-w} = s_i$ indicating an empty range) and has zeroes in all other rows. We call s_i the bottom ribbon border and $s_{i-w} + 1$ the top ribbon border. Let us check that d essentially runs through this range.[15] We have the following.

[15]If w or more consecutive starting positions have no key assigned to them, the ribbon misses some columns, in which then the "ribbon height" $s_i - s_{i-w}$ is zero. In such a case we permit that the diagonal is one position above the top ribbon border $s_{i-w} + 1$.

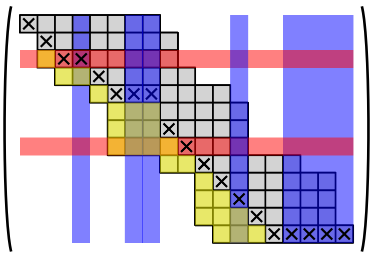

Fig. 3. An example for a ribbon diagonal in the ribbon matrix A, indicated by $\times$-marks. Each column i has an $\times$-mark, i.e., in each step i the ribbon diagonal makes one step to the right, from column $i-1$ to column i. In the standard case, in which a key is inserted in slot i, it also makes one step down. If slot i in M remains empty, this step down is not taken (*blue* columns). If the insertion of a key with starting position $s(x)$ fails because its row does not increase the row rank, this is recorded by the ribbon diagonal in that it makes *an extra step down* when reaching column $s(x)+w-1$ (*red* rows; note the delay, and note that in a blue column such an extra step is "canceled"). The number of yellow cells is the sum of the heights $h_i = d_i - s_i$, which serves as a bound for the number of row operations in successful insertions.

Lemma 3.1. *For all $i \in [m]$, we have $s_i \geq d_i \geq s_{i-w}$.*

Proof. The first inequality holds because

$$d_i = r_i + s'_{i-w+1} \leq r_i + s'_i = |P_M \cap [1,i]| + s'_i \leq s_i,$$

where the last step uses that each key in S_i can fail to be inserted or fill a slot in M, but not both. The second inequality is true because

$$d_i = |P_M \cap [1,i]| + s'_{i-w+1} \geq (s_{i-w+1} - s'_{i-w+1}) + s'_{i-w+1} = s_{i-w+1} \geq s_{i-w}.$$

where the first "$\geq$" uses that the first s_{i-w+1} rows cause $s_{i-w+1} - s'_{i-w+1}$ slots with index at most i to be filled. ◻

The first part of Theorem 3.1 ensures that the *height* $h_i := s_i - d_i$ of the ribbon diagonal above the bottom ribbon border is non-negative. (In Figure 3 the sum of these heights is indicated as the yellow area.) This quantity plays a central role in the precise versions of **(O1)** to **(O3)**, which we prove next. The main adjustment we make is to show that d is probabilistically repelled when *close* to one of the ribbon borders, while d_{simp} only responds to outright collisions.

Lemma 3.2 (Precise version of **(O1)**). *We have* $\Pr[i \notin P_M \mid h_{i-1} = k] \leq 2^{-k}$ *for any $k \in \mathbb{N}_0$. Moreover, this bound remains true conditioned on any shape of the ribbon and the content of the first $i-1$ columns. Formally, we have* $\Pr[i \notin P_M \mid h_{i-1} = k, E] \leq 2^{-k}$ *for any $k \in \mathbb{N}_0$ and any event E relating to $s_1, \ldots, s_m$ and the values $(h_j, s'_{j-w+1}, [j \in P_M], r_j)_{1 \leq j < i}$, assuming* $\Pr[h_{i-1} = k, E] > 0$.

Proof. We ignore E at first. A useful alternative way to think about Algorithm 1 uses language from linear probing: A key x *probes* slots $s(x), s(x)+1, \ldots, s(x)+w-1$ one by one. If slot i is probed and found to be occupied, the coefficient vector of x is possibly changed by a row operation, thus enforcing that coefficient a_i is 0. If slot i is probed and found empty, and the current entry a_i of the coefficient vector of x is 1 then x is inserted into that slot (this certainly happens if $i = s(x)$), otherwise x keeps probing. Here we are interested only in the case $s(x) < i$. Then, a_i is initially random. Since any bits that are added to a_i during row operations are uncorrelated with it, we have $\Pr[a_i = 1] = \frac{1}{2}$ at the moment when x probes slot i. Now consider a fixed slot i. Of the s_{i-1} keys with starting position at most $i-1$, precisely r_{i-1} have been successfully inserted into slots in $[1, i-1]$, and s'_{i-w} insertions have failed without probing slot i. Therefore

$s_{i-1} - s'_{i-w} - r_{i-1} = s_{i-1} - d_{i-1} = h_{i-1}$ such keys probe slot i. Each one of them has probability $\frac{1}{2}$ to be inserted, independently. So conditioned on $h_{i-1} = k$, the probability that none of these keys will be placed in slot i is 2^{-k}. If $s_i > s_{i-1}$, slot i will be filled by a key with starting position i, so overall we get $\Pr[i \notin P_M \mid h_{i-1} = k] \le 2^{-k}$.

Note that the only sources of randomness relevant for this argument are the entries in the ith column of A belonging to keys x with $s(x) < i$. In particular, we may condition on $s_1, \ldots, s_m$ and the entries in the first $i - 1$ columns of A without changing the argument. It is easy to see that $h_j, s'_{j-w+1}, [j \in P_M], r_j$ for $1 \le j < i$ and hence also E are functions of these values. □

Lemma 3.3 (Precise version of (O2)). *Let x be a key with $i = s(x)$ and let occ_x be the number of slots in $\{i, \ldots, i + w - 1\}$ that are occupied when x is inserted. Then, we have the following.*

(a) $\mathrm{occ}_x \le h_i$.
(b) $\Pr[x \in S' \mid h_i = k] \le 2^{-(w-k-1)}$, *for all* $k \in \mathbb{N}_0$ *with* $\Pr[h_i = k] > 0$.

Proof. **(a)** Of the $s_i - 1$ keys that are handled before x, exactly r_{i-1} are inserted before slot i and at least s'_{i-1} are not inserted successfully. The number of slots in $\{i, \ldots, i + w - 1\}$ that are occupied when x is inserted can therefore be bounded as follows:

$$\mathrm{occ}_x \le s_i - 1 - r_{i-1} - s'_{i-1} \le s_i - r_i - s'_{i-w+1} = s_i - d_i = h_i.$$

(b) Assume $h_i = k$. When x is inserted there are $w - \mathrm{occ}_x \ge w - k$ free slots in $\{i, \ldots, i + w - 1\}$. At least $w - k - 1$ of these are beyond column i (where matrix entries are independent of h_i). We can now argue as in the proof of Theorem 3.2 that x has at least $w - k - 1$ chances with success probability $\frac{1}{2}$ to be inserted into an empty slot. □

Recall the definition of Δ as the maximum of all upper bounds Δ^+ as in Equation (4). This maximum is attained when the keys are inserted in order of ascending values $s(x)$.

Lemma 3.4 (Precise version of (O3)). *The bound Δ on the number of row operations performed during* successful *insertions satisfies $\Delta \le n(w-1)$ (trivially) and $\Delta \le \sum_{i\in[m]} h_i$.*

Proof. By the discussion around Equation (4) we may assume that the insertion order is given by the starting positions $s(x)$. Since then $i(x) - s(x) \le w - 1$ holds for all $x \in S^+$, the trivial bound $\Delta \le n(w-1)$ follows.

For the bound in terms of the heights h_i we need to translate between row-wise and column-wise perspectives on the same information. First note that:

$$\sum_{i\in[m]} s_i = |S|(m+1) - \sum_{x\in S} s(x),$$

because each $x \in S$ contributes $m + 1 - s(x)$ to the left sum (by contributing 1 for each $i \in \{s(x), \ldots, m\}$). In a similar way we get

$$\sum_{i\in[m]} s'_{i-w+1} = |S'|(m-w+2) - \sum_{x\in S'} s(x) \qquad \text{and} \qquad \sum_{i\in[m]} r_i = |P_M|(m+1) - \sum_{i\in P_M} i\,.$$

Using these insights, the definitions of h_i and d_i, and the equality $|P_M| + |S'| = |S|$, we get

$$\begin{aligned}\sum_{i\in[m]} h_i &= \sum_{i\in[m]} (s_i - d_i) = \sum_{i\in[m]} (s_i - r_i - s'_{i-w+1}) \\ &= |S|(m+1) - \sum_{x\in S} s(x) - |P_M|(m+1) + \sum_{i\in P_M} i - |S'|(m-w+2) + \sum_{x\in S'} s(x)\end{aligned}$$

ALGORITHM 2: The construction algorithm of standard Ribbon.

```
   Input: $f\colon S \to \{0,1\}^r$ for some $S \subseteq \mathcal{U}$ of size $n$.
   Parameters: $w \in \mathbb{N}$, $\varepsilon > 0$.
1  $m \leftarrow n/(1-\varepsilon) + w - 1$; allocate system $M$ of size $m$
2  pick hash functions $s\colon \mathcal{U} \to [m-w+1]$, $c\colon \mathcal{U} \to \{0,1\}^w$
3  for $x \in S$ do
4      ret $\leftarrow$ insert($x$) // using Algorithm 1
5      if ret = FAILURE then
6          restart
7  $Z \leftarrow 0^{m\times r}$
8  for $i = m$ down to 1 do
9      $Z_i \leftarrow M.b[i] \oplus M.c[i] \cdot Z_{i..i+w-1}$ // back substitution
10 return $(s, c, Z)$
```

ALGORITHM 3: The query algorithm of Standard Ribbon.

```
   Input: $x \in \mathcal{U}$, data structure $(s, c, Z)$.
1  return $c(x) \cdot Z_{s(x)..s(x)+w-1}$
```

$$
\begin{aligned}
&= |S'|(m+1) - |S'|(m-w+2) + \sum_{i\in P_M} i - \sum_{x\in S^+} s(x)\\
&= |S'|(w-1) + \Delta\\
&\geq \Delta.
\end{aligned}
$$

□

4 Standard Ribbon Retrieval

This section is devoted to proving Theorem 1.5, thereby analyzing *standard ribbon*, which is retrieval data structure obtained by applying the ideas laid out in Section 2 in the straightforward way. We sketch an implementation in Algorithm 2.

Given a set S of n keys and given values by $f\colon S \to \{0,1\}^r$, we allocate a system M of size $m = n/(1-\varepsilon) + w - 1$ and try to insert all keys using Algorithm 1. If any insertion fails, the entire construction is restarted with new hash functions. Otherwise, we obtain a solution Z to M in the *back substitution phase*. The rows of Z are obtained from bottom to top. If slot i of M contains an equation then this equation uniquely determines row i of Z in terms of rows of Z below it. If slot i of M is empty, Algorithm 2 will initialize row Z_i to 0^r. We note for later use that actually row i of Z could be initialized arbitrarily in this case.

The expected "slope" of the ribbon is $1-\varepsilon$, giving us reason to hope that the ribbon diagonal will stick to the left ribbon border so that failures are unlikely.

Consider a run of Algorithm 2 that does not restart even when an insertion failure occurs. Then, the symbols s_i, s'_i, d_i, h_i for $i \in [m]$ from Section 3.2 are well-defined.

Lemma 4.1. *The heights $h_i = s_i - d_i$ for $i \in [m]$ satisfy:*

(a) $\mathbb{E}[h_i] = O(1/\varepsilon)$;
(b) $\forall k > 1/\varepsilon\colon\ \Pr[h_i > k] = \exp(-\Omega(\varepsilon k)) + O(n^{-2})$.

Once we have this lemma, we can prove Theorem 1.5, as follows.

Proof of Theorem 1.5. We pick $w = C\frac{\log n}{\varepsilon}$ for some sufficiently large constant C. (The theorem is formulated for $C = 1$, but it is clear that C can be absorbed in the O terms for ε and $1/\varepsilon$.) Further, we choose $m = \frac{n}{1-\varepsilon} + w - 1$, so that the space required for $Z \in \{0,1\}^{m\times r}$ is $mr = (\frac{n}{1-\varepsilon} + w - 1)r = (1 + O(\varepsilon))rn$ bits, giving an overhead of $O(\varepsilon)$ as claimed. Executing a query with Algorithm 3 involves the product of a w-bit vector and a $w \times r$-bit matrix. On a word RAM with word size $\Omega(\log n)$ this takes time $O(\frac{wr}{\log n}) = O(r/\varepsilon)$.

It follows from Theorem 4.1 (b) that $h_i \le w/2$ for all $i \in [m]$ with high probability. Theorem 3.3 (b) then ensures that all keys are inserted successfully with high probability. This implies that the effect of restarts on total construction time is negligible (as is the space required to store the seeds identifying s and c). Now consider the running time of one construction attempt. Combining Theorem 3.4 with Theorem 4.1 (a) shows that the expected number of row operations during successful insertions is $O(n/\varepsilon)$. Each row operation takes time $O(1/\varepsilon)$, which makes for a total contribution of $O(n/\varepsilon^2)$ to construction time. Since construction is aborted on the first failure, there can be at most one failed insertion. Since any unsuccessful insertion turns out to be a failure with probability $1 - 2^{-r} \ge \frac{1}{2}$, the expected number of unsuccessful insertions (including redundant insertions) is at most 2. Each redundant insertion can contribute at most $O(n)$ row operations, which is negligible. □

We devote the rest of the section to the proof of Theorem 4.1, using a combination of the techniques of Poissonization, coupling, and known results from queueing theory.[16]

4.1 Bounding Heights by a Markov Chain

For getting a hold on the rather complicated random variables $(h_i)_{i\in[m]}$ we apply the well-known concept of a *coupling* of two random variables. This is a common probability space on which both these random variables can be defined. A coupling gets its power from the way the two random variables are connected in this particular probability space. (For information about couplings see, e.g., [56] and [48, Ch. 11].)

We will define a coupling of $(h_i)_{i\in[m]}$ and a much simpler Markov chain, which is defined as follows. Let $p_1, p_2, \dots \sim \mathrm{Po}(1 - \varepsilon/2)$ and $g_1, g_2, \dots \sim \mathrm{Geom}(\frac{1}{2})$ be independent random variables with Poisson distribution and geometric distribution, respectively. Then, define

$$h'_0 := 0, \quad \text{and} \quad h'_i := h'_{i-1} + b_i + p_i - 1 \text{ for } i \ge 1, \quad \text{where}^{17}\ b_i = [g_i \ge h'_{i-1}].$$

Lemma 4.2. *There exists a coupling of the random variables $(h_i)_{i\in[m]}$ and $(h'_i)_{i\in[m]}$ in which the following holds: There is an event $E_{\ge n}$ with $\Pr[E_{\ge n}] = 1 - O(n^{-2})$ such that when $E_{\ge n}$ occurs then $h_i \le h'_i$ holds for all $i \in [m]$.*

Proof. The idea is to link the binomially distributed random variable $s_i - s_{i-1} \sim \mathrm{Bin}(n, \frac{1}{m-w+1})$, which captures the number of keys with starting position i, to the Poisson distributed random variable $p_i \sim \mathrm{Po}(1 - \varepsilon/2)$, and the event $\{i \notin P_M\}$ of "skipping" column i to the event $\{b_i = 1\}$. We begin with an alternative probability space for $p_1, \dots, p_m$.

Consider a random experiment where we first sample $n' \sim \mathrm{Po}((1 - \varepsilon/2)(m - w + 1))$ and then throw n' balls randomly into $m - w + 1$ bins. Let p_i for $i \in [m - w + 1]$ be the number of balls landing in bin i and $p_i \sim \mathrm{Po}(1 - \varepsilon/2)$ for $i > m - w + 1$ be additional random variables.

It follows from [48, Thm. 5.6] that $p_1, \dots, p_m$ are then independent random variables with distribution $\mathrm{Po}(1 - \varepsilon/2)$ as introduced earlier. Note that $\mathbb{E}[n'] = (1 - \varepsilon/2)(m - w + 1) = \frac{1-\varepsilon/2}{1-\varepsilon} n = (1 + \Theta(\varepsilon))n$. Standard concentration bounds for Poisson random variables (see, e.g., [48, Thm. 5.4]) imply that the event $E_{\ge n} := \{n' \ge n\}$ occurs with probability $1 - e^{-\Omega(n)} = 1 - O(n^{-c})$ for arbitrary constant $c > 0$. We condition on $E_{\ge n}$ from now on.

[16]For the original version of this analysis see [24].
[17]If ψ is a statement about random variables, the *Iverson bracket* $[\psi]$ denotes the indicator variable of the event $\{\psi\}$.

The numbers $s_i - s_{i-1}$ of keys with starting position i for $i \in [m - w + 1]$ correspond to the numbers of balls in bin i when throwing n balls randomly into $m - w + 1$ bins. The natural coupling where the numbers p_i arise by throwing those n balls and $n' - n$ additional balls ensures $s_i - s_{i-1} \le p_i$ for $i \in [m - w + 1]$. Since columns beyond $m - w + 1$ cannot occur as starting positions of keys we also have $s_i - s_{i-1} = 0 \le p_i$ for $m - w + 1 < i \le m$. To summarize, our coupling ensures:

$$s_i - s_{i-1} \le p_i \text{ for } i \in [m] \text{ conditioned on } E_{\ge n}. \tag{5}$$

Now fix any outcome of $(s_i)_{i\in[m]}$ and $(p_i)_{i\in[m]}$ and consider the process of revealing the values $\Delta_i := (h_i, s'_{i-w+1}, [i \in P_M], r_i)$ as well as b_i for increasing i, one by one. By Theorem 3.2 we have $\Pr[i \notin P_M \mid h_{i-1} = k, s_1, \dots, s_m, \Delta_1, \dots, \Delta_{i-1}] \le 2^{-k}$, for all $k \in \mathbb{N}_0$. On the other hand we have $\Pr[b_i = 1 \mid h'_{i-1} = k] = 2^{-k}$, for all $k \in \mathbb{N}_0$, by definition. A suitable coupling can therefore ensure that $[i \notin P_M] \le b_i$ whenever $h_{i-1} = h'_{i-1}$. A simple implication of this is that:

$$h_{i-1} + [i \notin P_M] \le h'_{i-1} + b_i \text{ whenever } h_{i-1} \le h'_{i-1}, \text{ for } i \in [m]. \tag{6}$$

Together Equations (5) and (6) imply $h_i \le h'_i$ for all $i \in [m]$ by induction, as follows:

$$h_i = s_i - r_i - s'_{i-w+1} = h_{i-1} + (s_i - s_{i-1}) - (r_i - r_{i-1}) - (s'_{i-w+1} - s'_{i-w})$$
$$\overset{\text{Equation}(5)}{\le} h_{i-1} + p_i - [i \in P_M] = h_{i-1} + [i \notin P_M] + p_i - 1 \overset{\text{Equation}(6)}{\le} h'_{i-1} + b_i + p_i - 1 = h'_i. \quad \square$$

As a further simplification, we eliminate the geometrically distributed variable g_i and the derived variable b_i in the Markov chain $(h'_i)_{i\ge0}$. For this, let $(X_i)_{i\ge0}$ be the Markov chain defined by:

$$X_0 := 0 \quad \text{and} \quad X_i := \max(0, X_{i-1} + d_i - 1) \quad \text{for } i \ge 1, \tag{7}$$

where $d_i \sim \mathrm{Po}(1 - \varepsilon/4)$ are independent random variables.

Lemma 4.3. *There is a coupling between $(X_i)_{i\ge0}$ and $(h'_i)_{i\ge0}$ such that $X_i + \log(8/\varepsilon) \ge h'_i$ for all $i \ge 0$.*

Proof. Assume w.l.o.g. that $\log(8/\varepsilon)$ is an integer. Let $b'_i \sim \mathrm{Po}(\varepsilon/4)$ be a random variable on the same probability space as g_i such that:

$$g_i > \log(8/\varepsilon) \text{ implies } b'_i \ge 1. \tag{8}$$

This is possible because

$$\Pr[g_i > \log(8/\varepsilon)] = 2^{-\log(8/\varepsilon)} = \varepsilon/8 \le 1 - \mathrm{e}^{-\varepsilon/4} = \Pr[b'_i \ge 1].$$

We then define $d_i := p_i + b'_i$ which gives $d_i \sim \mathrm{Po}(1 - \varepsilon/4)$. Now that $(X_i)_{i\ge0}$ and $(h'_i)_{i\ge0}$ are defined on a common probability space, we check $X_i + \log(8/\varepsilon) \ge h'_i$ by induction. There are two cases.

Case 1: $h'_{i-1} \le \log(8/\varepsilon) - 1$. Together with $X_{i-1} \ge 0$, $b'_i \ge 0$ and $b_i \le 1$ we get

$$X_i + \log(8/\varepsilon) \ge X_{i-1} + d_i - 1 + \log(8/\varepsilon) \ge p_i + b'_i + h'_{i-1} \ge h'_{i-1} + p_i + b_i - 1 = h'_i.$$

Case 2: $h'_{i-1} \ge \log(8/\varepsilon)$. From Equation (8) we obtain

$$b_i = [g_i > h'_{i-1}] \le [g_i > \log(8/\varepsilon)] \le [b'_i \ge 1] \le b'_i.$$

Together with the induction hypothesis $X_{i-1} + \log(8/\varepsilon) \ge h'_{i-1}$ we get

$$X_i + \log(8/\varepsilon) \ge X_{i-1} + d_i - 1 + \log(8/\varepsilon) \overset{\text{Ind.}}{\ge} h'_{i-1} + d_i - 1$$
$$= h'_{i-1} + p_i + b'_i - 1 \ge h'_{i-1} + p_i + b_i - 1 = h'_i. \quad \square$$

4.2 Enter Queueing Theory

A Markov chain much like $(X_i)_{i\geq 0}$ has been studied in the literature under the name "M/D/1 queue", which is Kendall notation [40] for queues with "**M**arkovian arrivals, **D**eterministic service times and **1** server". We will exploit what is known about this simple queueing situation in order to finish our analysis.

Formally, an M/D/1 queue is a Markov process $(Z_t)_{t\in\mathbb{R}_{\geq 0}}$ in continuous time and discrete space $\mathbb{N}_0 = \{0,1,2,\dots\}$. The random variable Z_t is usually interpreted as the number of *customers* waiting in a FIFO queue at time $t \in \mathbb{R}_{\geq 0}$. Initially the queue is empty ($Z_0 = 0$). Customers arrive independently, i.e., arrivals are determined by a Poisson process with a rate we set to $\rho = 1 - \varepsilon/4$ (which implies that the number of customers arriving in any fixed time interval of length 1 is $\mathrm{Po}(\rho)$-distributed). The *server* requires one time unit to process a customer, which means that if $t \in \mathbb{R}_{\geq 0}$ is the time of the first arrival, then customers will leave the queue at times $t+1, t+2, \dots$ until the queue is empty again.

Now consider the discretization $(Z_i)_{i\in\mathbb{N}_0}$ of the M/D/1 queue. For $i \geq 1$, the number d_i of arrivals between two observations Z_{i-1} and Z_i has distribution $d_i \sim \mathrm{Po}(\rho)$, independently, and one customer is served in $[i-1, i)$ if and only if $Z_{i-1} > 0$. We can therefore write

$$Z_i = \begin{cases} d_i & \text{if } Z_{i-1} = 0, \\ Z_{i-1} + d_i - 1 & \text{if } Z_{i-1} > 0. \end{cases}$$

By identifying the random variables d_i induced by $(Z_t)_{t\in\mathbb{R}_{\geq 0}}$ with the random variables $(d_i)_{i\geq 1}$ that occurred in the definition of $(X_i)_{i\geq 0}$ (Equation (7)), we immediately obtain a coupling between the processes $(X_i)_{i\geq 0}$ and $(Z_i)_{i\geq 0}$. A simple induction suffices to show

$$X_i = \max(0, Z_i - 1), \quad \text{for all } i \geq 0. \tag{9}$$

Intuitively, the server in the X-process is ahead by one customer because customers are processed at integer times "just in time for the observation".

The following results are known in queueing theory:

Lemma 4.4. (i) *The average number of customers in the Z-queue at time $t \in \mathbb{R}_{\geq 0}$ is:*

$$\mathbb{E}[Z_t] \leq \lim_{\tau\to\infty} \mathbb{E}[Z_\tau] = \rho + \tfrac{1}{2}\left(\frac{\rho^2}{1-\rho}\right) = \Theta(1/\varepsilon).$$

(Precise values are known even for general service-time distributions, see [15, Ch. 5.4].)

(ii) *[28, Prop 3.4[18]] For all $t \geq 0$ we have the following tail bound for Z_t:*

$$\Pr[Z_t > k] \leq \lim_{\tau\to\infty} \Pr[Z_\tau > k] = \mathrm{e}^{-k\cdot\Theta(\varepsilon)}, \text{ for any } k \in \mathbb{N}.$$

[18]It should be noted that [28] studies a slightly different setting called "M/D/1 PS" where PS stands for *processor sharing*. This means that at any point in time the single server serves all customers in the queue with correspondingly reduced speed (imagine a CPU being shared by all active processes). In this setting the authors consider the *sojourn time* $S_{\mathrm{PS}}(c)$ of a customer c, i.e., the time c spends in the queue. This is at least the sojourn time $S_{\mathrm{FIFO}}(c)$ of c if the server were to work on jobs in the order in which they arrive. To see this, note that under PS customers still finish in the order in which they arrive and the delay $S_{\mathrm{PS}}(c) - S_{\mathrm{FIFO}}(c)$ of c is precisely the work that, when c leaves the queue, the server has already done for customers who arrived after c. If $Z(c)$ denotes the number of customers already in the queue when customer c arrives then $S_{\mathrm{FIFO}}(c) \in [Z(c), Z(c)+1]$ and by the so-called *PASTA property* ("Poisson Arrivals See Time Averages") $Z(c)$ is distributed like Z_τ assuming we look at the converged distribution in both cases, i.e., a late customer c and a late time τ. Hence the tail bound from [28] on $S_{\mathrm{PS}}(c)$ applies to $Z_\tau \overset{\mathrm{d}}{=} Z(c) \leq S_{\mathrm{FIFO}}(c) \leq S_{\mathrm{PS}}(c)$.

4.3 Putting the Pieces Together

We now have everything in place to prove Theorem 4.1.

Proof of Theorem 4.1. We may assume that the random variables $(h_i)_{i\in[m]}$, $(h'_i)_{i\geq 0}$, $(X_i)_{i\geq 0}$ and $(Z_i)_{i\geq 0}$ reside in the same probability space and that the three couplings we have constructed are realized at the same time. Conditioned on the high probability event $E_{\geq n}$ from Theorem 4.2 we have

$$h_i \overset{\text{Lemma 4.2}}{\leq} h'_i \overset{\text{Lemma 4.3}}{\leq} X_i + \log(8/\varepsilon) \overset{\text{Equation (9)}}{\leq} Z_i + \log(8/\varepsilon).$$

This implies for $k > 1/\varepsilon$

$$\Pr[h_i > k] \leq \Pr[Z_i + \log(8/\varepsilon) > k] + \Pr[E_{\geq n} \text{ fails to occur}]$$
$$\overset{\text{Lemma 4.4(ii)}}{\leq} e^{-(k-\log(8/\varepsilon))\Omega(\varepsilon)} + O(n^{-2}) = e^{-\Omega(\varepsilon k)} + O(n^{-2}).$$

Along similar lines we get, for each $i \in [m]$, using that $(h_i)_{i\in[m]}$ and $E_{\geq n}$ are independent:

$$\mathbb{E}[h_i] = E[h_i \mid E_{\geq n}] \overset{\text{Lemma 4.2}}{\leq} \mathbb{E}[h'_i \mid E_{\geq n}] \leq \tfrac{1}{\Pr[E_{\geq n}]}\mathbb{E}[h'_i] \overset{\text{Lemma 4.3}}{\leq} \tfrac{1}{\Pr[E_{\geq n}]}\mathbb{E}[X_i + \log(8/\varepsilon)]$$
$$\overset{\text{Equation (9)}}{\leq} \tfrac{1}{\Pr[E_{\geq n}]}\mathbb{E}[Z_i + \log(8/\varepsilon)] \overset{\text{Lemma 4.4(i)}}{\leq} \tfrac{1}{\Pr[E_{\geq n}]}(O(1/\varepsilon) + \log(8/\varepsilon)) = O(1/\varepsilon). \qquad \square$$

5 Homogeneous Ribbon Filters

5.1 Homogeneous Ribbon Filter

For the application of ribbon to AMQs, we can also compute a *uniformly random* solution of the *homogeneous* equation system $AZ = 0$, i.e., we compute a retrieval data structure that will retrieve 0^r for all keys of S but is unlikely to produce 0^r for other inputs. Since $AZ = 0$ is always solvable, all insertions in system M will be successful. The crucial point is that the false positive rate is no longer 2^{-r} but higher. We will show, however, that with table size $m = (1+\varepsilon)n$ and $\varepsilon = \Omega(\frac{\max(r,\log w)}{w})$ the difference is negligible, thereby showing Theorem 1.8. In this section, we give a precise description and analysis of Homogeneous Ribbon Filters, which are simpler than filters based on Standard Ribbon but are unsuitable for retrieval. Their disadvantage is that they have higher space overhead than AMQs based on ribbon based retrieval structures like BuRR (see Section 6). Our experiments indicate that together, BuRR based AMQs and homogeneous ribbon AMQs cover a large part of the best tradeoffs for static AMQs.

Recall from Section 1.5 the approach for constructing a filter by picking hash functions $\vec{h}\colon \mathcal{U} \to \{0,1\}^m$ and $f\colon \mathcal{U} \to \{0,1\}^r$, and finding $Z \in \{0,1\}^{m\times r}$ such that all $x \in S$ satisfy $\vec{h}(x)\cdot Z = f(x)$, while most $x \in \mathcal{U} \setminus S$ do not. We now dispose of the fingerprint function f, effectively setting $f(x) = 0^r$ for all $x \in \mathcal{U}$. A filter is then given by a solution Z to the *homogeneous* system $(\vec{h}(x)\cdot Z = 0^r)_{x\in S}$. The false positive rate for Z is $\varphi_Z = \Pr_{a\sim H}[a\cdot Z = 0^r]$ where H is the distribution of $\vec{h}(x)$ for $x \in \mathcal{U}$. An immediate issue with this idea is that $Z = 0^{m\times r}$ is a valid solution but gives $\varphi_Z = 1$. We therefore pick Z *uniformly at random* from all solutions. If $\vec{h}$ has the form $\vec{h}(x) = 0^{s(x)-1}c(x)0^{m-s(x)-w+1}$ from standard ribbon retrieval, we call the resulting filter a *Homogeneous Ribbon Filter*. The full construction is shown in Algorithm 4. Two notable simplifications in comparison to Algorithm 2 are that no function f is needed and that a restart is never required. The main difference is that free variables Z_i, i.e., variables corresponding to empty slots in M, must now be sampled uniformly at random[19] during back substitution.

[19]Our implementation uses simple pseudo-random assignments instead: a free variable in row i is assigned $p\cdot i \bmod 2^r$ for some fixed large odd number p.

ALGORITHM 4: The construction algorithm of homogeneous ribbon filters.

```
Input: $S \subseteq \mathcal{U}$ of size $n$.
Parameters: $r \in \mathbb{N}, w \in \mathbb{N}, \varepsilon > 0$.
$m \leftarrow n/(1-\varepsilon) + w - 1$; allocate system $M$ of size $m$
pick hash functions $s\colon \mathcal{U} \to [m-w+1],\ c\colon \mathcal{U} \to \{1\} \times \{0,1\}^{w-1}$
sort $S$ by $s(x)$ // at least approximately, see proof of Theorem 5.3
for $x \in S$ do
    insert($x$) // using Algorithm 1 with $f \equiv 0$. Cannot fail!
$Z \leftarrow 0^{m \times r}$
for $i = m$ down to 1 do
    if $M.c[i] = 0$ then // slot unused?
        sample $Z_i \sim U(\{0,1\}^r)$ // randomly initialize free variable
    else
        $Z_i \leftarrow M.c[i] \cdot Z_{i..i+w-1}$ // back substitution
return $(s, c, Z)$
```

ALGORITHM 5: The query algorithm of homogeneous ribbon filters.

```
Input: $x \in \mathcal{U}$, data structure $(s, c, Z)$.
return $[c(x) \cdot Z_{s(x)..s(x)+w-1} = 0^r]$
```

The overall false positive rate is $\varphi = \mathbb{E}[\varphi_Z]$ where Z depends on the randomness in $(\vec{h}(x))_{x\in S}$ and the free variables. A complication is that $\varphi = 2^{-r}$ no longer holds, instead there is a gap $\varphi - 2^{-r} > 0$. We show that this gap is negligible if two conditions are satisfied. Firstly, the filter must be *underloaded*, with $\varepsilon \approx \frac{m-n}{m} > 0$, which leads to a memory overhead of $O(\varepsilon)$. Secondly, the ribbon width w must satisfy $w = \Omega(r/\varepsilon)$. The good news is that there is no dependence of w on n (like $w = \Omega(\frac{\log n}{\varepsilon})$ as required in Standard Ribbon) and that no sharding or bumping is required. More precisely, we prove Theorem 1.8, restated here for convenience.

Theorem 5.1 (Restatement of Theorem 1.8). *There exists a constant C such that for arbitrary $r \geq 1$, $\varepsilon \in (0, 1/2]$, for $w = \frac{C\max(r,\log(1/\varepsilon))}{\varepsilon}$ and for any n with $n = \omega(w/\varepsilon)$ and $\varepsilon w = O(\log n)$ the following holds. The Homogeneous Ribbon Filter with ribbon width w and r-bit fingerprints for n keys has space overhead ε and false positive probability $(1 + O(\varepsilon^2))2^{-r}$. On a word RAM with word size at least w the expected construction time is $O(n/\varepsilon)$ and the query time is $O(r)$.*[20]

Note that with $w = \Theta(\log n)$ we can achieve an overhead of $\varepsilon = O(\frac{\max(r,\log\log n)}{\log n})$.

5.2 Chernoff Bounds

The following standard lemma will play a role in this section and in Section 6.

Lemma 5.2. *Let $(X_j)_{j\in[N]}$ be i.i.d. indicator random variables, $X := \sum_{j\in[N]} X_j$ and $\mu := \mathbb{E}[X]$.*

(a) *For $\delta \in [0, 1]$ we have $\Pr[|X - \mu| \geq \delta\mu] \leq 2\exp(-\delta^2\mu/3)$.*

(b) *There exists $C > 0$ such that for any $w \in \mathbb{N}$ and $\mu \leq \frac{2w^2}{C\log w}$ we have $\Pr[|X - \mu| \geq \frac{w}{8}] = O(w^{-5})$.*

Proof. **(a)** This combines standard Chernoff bounds on the probability of the events $\{X \geq (1+\delta)\mu\}$ and $\{X \leq (1-\delta)\mu\}$, as found, for instance, in [48, Chapter 4].

[20]For w larger than the word size W the running times increase by a factor of w/W, by standard arguments.

(b) We set $\delta = \frac{w}{8\mu}$ and apply (a). This gives

$$\Pr[|X - \mu| \geq \tfrac{w}{8}] \leq 2\exp(-\delta^2\mu/3) = 2\exp(-\tfrac{w^2}{192\mu}) \leq 2\exp(-\tfrac{C\log w}{384}) = 2w^{-C/384}.$$

Choosing $C = 1920$ achieves the desired bound.[21] □

5.3 Proof of Theorem 1.8: Running Time

The running time of Algorithm 5 is $O(r)$, if the storage structure for Z is chosen in a suitable way. (For details see Section 7.2 below, in particular Figure 6.) For the construction time, we reuse results for Standard Ribbon Retrieval. While insertions cannot fail, the set of *redundant* keys, i.e., keys for which Algorithm 1 returns "REDUNDANT" rather than "SUCCESS" now demands attention. Seen through the lens of our analytical tools they behave exactly like failed insertions (both do not modify the system M), with the important difference that in the situation considered here many redundant insertions are allowed to occur.

Lemma 5.3. *In the situation of Theorem 1.8 we have:*

(a) *The expected fraction of keys that lead to redundant insertions is* $\exp(-\Omega(\varepsilon w))$.
(b) *The expected number of row additions during construction is* $O(n/\varepsilon)$.

Proof. **(a)** Let $x \in S$ be any key, $S^- := S \setminus \{x\}$ and $i^* \in [m-w+1]$ a uniformly random starting position. Let $(h_i)_{i\in[m]}$ and $(h_i^-)_{i\in[m]}$ be the heights when running the ribbon construction with S and S^-, respectively. We have

$$\Pr[h_{s(x)} \geq w/2+1] \leq \Pr[h_{i^*}^- \geq w/2] \leq \Pr[h_{i^*} \geq w/2] = \exp(-\Omega(\varepsilon w)) + O(n^{-2}).$$

The first two inequalities hold because $s(x)$ and i^* have the same distribution and because the presence of x can affect any height value only by +0 or +1. The last bound follows from Theorem 4.1 (b) (the bound even holds if i^* is fixed).
By Theorem 3.3 (b), the event that x is redundant despite $h_{s(x)} \leq w/2$ has probability at most $2^{-w/2+1}$, which is $\exp(-\Omega(\varepsilon w))$. Overall, the event E_x that x is redundant has probability at most

$$\Pr[E_x] \leq \Pr[h_{s(x)} > w/2] + \Pr[E_x \wedge h_{s(x)} \leq w/2] \leq \exp(-\Omega(\varepsilon w)) + O(n^{-2}).$$

For $w = \Omega(\log(n)/\varepsilon)$ our analysis of Standard Ribbon Retrieval guarantees that there are no redundant insertions with high probability, see the proof of Theorem 1.5 in Section 4. So we may assume $w = o(\log(n)/\varepsilon)$ essentially without loss of generality. In this case, the $O(n^{-2})$ term is dominated by $\exp(-\Omega(\varepsilon w))$.

(b) Combining Theorem 3.4 with Theorem 4.1 (a) shows that the expected number of row operations during *successful* insertions is $O(n/\varepsilon)$. Moreover, since we sort insertions by $s(x)$, the insertion of any key x can take at most w steps, simply because no 1-entries exist beyond column $s(x) + w - 1$ in M. Hence, due to (a), redundant insertions contribute at most $w\exp(-\Omega(\varepsilon w))n$ to the expected number of row additions, which is negligible compared to $O(n/\varepsilon)$.
Note that approximate sorting of keys into buckets of at most $\exp(-\Omega(\varepsilon w))/\varepsilon$ consecutive starting positions is also sufficient for achieving this bound on the number of row operations. □

[21] We do not attempt to optimize C here. Experimental evaluation of homogeneous ribbon filters yields no indication that the constant C needs to be particularly large to achieve small overhead and small false positive probability, see Section 8.

5.4 Proof of Theorem 1.8: False Positive Rate

Now we analyze the false positive rate. We start with the following simple observation.

Lemma 5.4. *Let p be the probability that for $y \in \mathcal{U} \setminus S$ the vector $\vec{h}(y)$ is in the span of $(\vec{h}(x))_{x \in S}$. Then, the false positive rate of the homogeneous ribbon filter is:*

$$\varphi = p + (1-p)2^{-r}.$$

Proof. The first case is that there exists $S' \subseteq S$ with $\vec{h}(y) = \sum_{x \in S'} \vec{h}(x)$, which happens with probability p. In that case

$$\vec{h}(y) \cdot Z = \left(\sum_{x \in S'} \vec{h}(x) \right) \cdot Z = \sum_{x \in S'} (\vec{h}(x) \cdot Z) = 0^r$$

and y is a false positive. The other case, which occurs with probability $1-p$, is that an attempt to add $\vec{h}(y) \cdot Z = 0^r$ to M after all equations for S were added would have resulted in a (non-redundant) insertion in some row i. During back substitution, only one choice for the ith row of Z satisfies $\vec{h}(y) \cdot Z = 0^r$. Since the ith row is initialized randomly we have $\Pr[\vec{h}(y) \cdot Z = 0^r \mid \vec{h}(y) \notin \mathrm{span}((\vec{h}(x))_{x \in S})] = 2^{-r}$. □

Next we derive a simple balls-into-bins lemma. The proof combines standard concentration bounds and couplings; unfortunately it is a little technical.

Lemma 5.5. *Assume n balls are thrown independently and uniformly at random into n bins resulting in a load of L_i balls in bin i for each $i \in [n]$. Let $L'_1 \geq \cdots \geq L'_n$ arise from $L_1, \ldots, L_n$ by sorting. Then with probability $1 - n^{-\omega(1)}$ we have for all $b \in [n]$ that $L'_1 + \cdots + L'_b = O(b(1 + \log \frac{n}{b}))$.*

Proof. We begin by introducing a list of ingredients to be used later.

(a) If $N \sim \mathrm{Po}(2n)$ then $\Pr[N < n] = \exp(-\Omega(n))$ by direct computation.
(b) By Stirling's approximation we have $k! = k^{\Theta(k)}$, which implies $(\log n)! = n^{\Theta(\log\log n)}$.
(c) For $\ell \geq 4$ and $L = \mathrm{Po}(2)$ we have $\Pr[L = \ell + i] \leq \Pr[L = \ell] \cdot 2^{-i}$, which also implies $\Pr[L \geq \ell] \leq 2\Pr[L = \ell]$.
(d) Let $X \sim \mathrm{Po}(2)|_{\geq \ell}$ be a Poisson random variable conditioned on assuming a value of at least ℓ. Using (c), a natural coupling to $G \sim \mathrm{Geom}(1/2)$ ensures $X \leq \ell + G$.
(e) Given independent $G_1, \ldots, G_k \sim \mathrm{Geom}(1/2)$ we have $G_1 + \cdots + G_k \leq 4k$ with probability $1 - \exp(-\Omega(k))$. This is because the opposite can be understood as $4k$ coin-flips giving fewer than k heads, which has probability $\exp(-\Omega(k))$ by Chernoff bounds, see Theorem 5.2.

Now the main argument starts. Since our claim may fail with probability $n^{-\omega(1)}$ we may ignore events that occur with at most such a probability.

The first step is "Poissonization", meaning we sample the number of balls as $N \sim \mathrm{Po}(2n)$. We may ignore the unlikely event that $N < n$ by (a). In other words we throw n balls as before and then $N - n \geq 0$ additional balls, which can only increase the loads of the bins (for which we seek an upper bound). The advantage is that with this change $L_1, \ldots, L_n$ are independent $\mathrm{Po}(2)$-distributed random variables. We have:

$$\Pr[L_1 \geq \log n] \overset{(c)}{\leq} 2\Pr[L_1 = \log n] = 2e^{-2}\frac{2^{\log n}}{(\log n)!} \overset{(b)}{=} n^{-\Theta(\log\log n)}.$$

By a union bound this implies $\Pr[\max\{L_1, \ldots, L_n\} \geq \log n] = n^{-\Theta(\log\log n)}$ as well. Whenever $b \leq n^{1/2}$ we get

$$L'_1 + \cdots + L'_b \leq b \cdot \log n = 2b \log n^{1/2} \leq 2b \log \frac{n}{b}.$$

In the following we may therefore assume $b \geq n^{1/2}$. We consider each value of b individually and use a union bound. The remaining goal is to show for a fixed $b \geq n^{1/2}$ that $L'_1 + \cdots + L'_b = O(b(1 + \log \frac{n}{b}))$ with probability $1 - n^{-\omega(1)}$.

Let ℓ be the smallest number such that $\Pr[L_1 \geq \ell] \leq \frac{b}{2n}$. We have $\ell = O(\log \frac{n}{b})$ by a similar argument as before using (b) and (c). We may imagine that we first reveal for each bin whether it is *heavy*, i.e., contains at least ℓ balls, or *light*, i.e., contains less than ℓ balls, and only afterwards sample L_i either from $\text{Po}(2)|_{\geq \ell}$ or from $\text{Po}(2)|_{<\ell}$, respectively. The expected number of heavy bins is at most $\frac{b}{2}$ by choice of ℓ and by a Chernoff bound and $b \geq n^{1/2}$ we may assume that the actual number of heavy bins is at most b. This means that the sum $L'_1 + \cdots + L'_b$ counts the balls in all heavy bins and some light bins. A natural coupling of $L'_1, \ldots, L'_b$ to $X_1, \ldots, X_b \sim \text{Po}(2)|_{\geq \ell}$ gives $L'_1 + \cdots + L'_b \leq X_1 + \cdots + X_b$. Combined with another coupling of $X_1, \ldots, X_b$ to $G_1, \ldots, G_b \sim \text{Geom}(1/2)$ from (d) and the upper bound from (e) we get as desired

$$L'_1 + \cdots + L'_b \leq X'_1 + \cdots + X'_b \leq b\ell + G_1 + \cdots + G_b \leq b\ell + 4b \leq b \cdot O(1 + \log \tfrac{n}{n}). \qquad \square$$

We now derive an asymptotic bound on p in terms of ε and (appropriately large) w.

Lemma 5.6. *There exists a constant C such that whenever $C\frac{\log w}{w} \leq \varepsilon \leq \frac{1}{2}$ we have $p = \exp(-\Omega(\varepsilon w))$.*

Proof. We may imagine that $S \subseteq \mathcal{U}$ and $y \in \mathcal{U} \setminus S$ are obtained from a set $S^+ \subseteq \mathcal{U}$ of size $n+1$ by picking $y \in S^+$ at random and setting $S = S^+ \setminus \{y\}$. Then, p is simply the expected fraction of keys in S^+ that are contained in some *dependent set*, i.e., in some $S' \subseteq S^+$ with $\sum_{x \in S'} \vec{h}(x) = 0^m$. Clearly, x is contained in a dependent set if and only if it is contained in a *minimal* dependent set. Such a set S' always "touches" a consecutive set of positions, i.e., $\text{pos}(S') := \bigcup_{x \in S'} \{s(x), \ldots, s(x) + w - 1\}$ is an interval.

We call an interval $I \subseteq [m]$ *long* if $|I| \geq w^2$ and *short* otherwise. We call it *overloaded* if $S_I := \{x \in S^+ \mid s(x) \in I\}$ has size $|S_I| \geq |I| \cdot (1 - \varepsilon/2)$. Finally, we call a position $i \in [m]$ *bad* if one of the following is the case:

(b1) i is contained in a long overloaded interval.
(b2) $i \in \text{pos}(S')$ for a minimal dependent set S' with long non-overloaded interval $\text{pos}(S')$.
(b3) $i \in \text{pos}(S')$ for a minimal dependent set S' with short interval $\text{pos}(S')$.

We shall now establish the following

Claim: The number B of bad positions satisfies $\mathbb{E}[B] \leq n \exp(-\Omega(\varepsilon w))$.

For each $i \in [m]$ the contributions from each of the "badness conditions" **(b1)**, **(b2)**, **(b3)** can be bounded separately. In all cases we use the assumption $\varepsilon \geq C\frac{\log w}{w}$. It ensures that $\exp(-\Omega(\varepsilon w))$ is at most $\exp(-c \log w) = w^{-c}$ for a constant $c > 0$ we can choose and hence can "absorb" factors of w in the sense that $w \exp(-c\varepsilon w) = \exp(-\Omega(\varepsilon w))$.

(b1) Let $I \subseteq [m]$ be any interval and let $X_1, \ldots, X_{n+1}$ indicate which of the keys in S^+ have a starting position within I. For $X := \sum_{j \in [n+1]} X_j$ we have

$$\mu := \mathbb{E}[X] \leq \frac{(n+1)|I|}{m - w + 1} = \frac{n+1}{n} \cdot \frac{n|I|}{m - w + 1} = \frac{n+1}{n}(1 - \varepsilon)|I|.$$

Using a Chernoff bound (Theorem 5.2 (a)), the probability for I to be overloaded is (since $n \gg w/\varepsilon \geq 1/\varepsilon^2$)

$$\Pr[X \geq (1 - \varepsilon/2)|I|] \leq \Pr[X \geq (1 + \underbrace{\varepsilon/2}_{\delta})\underbrace{\tfrac{n+1}{n}(1 - \varepsilon)|I|}_{\geq \mu}] \overset{\text{Lemma 5.2}}{\leq} \exp(\tfrac{-\varepsilon^2(1-\varepsilon)|I|}{12}).$$

The probability for $i \in [m]$ to be contained in a long overloaded interval is bounded by the sum of this bound over all lengths $|I| \geq w^2$ and all $|I|$ offsets that I can have relative to i.

The result is of order $\exp(-\Omega(\varepsilon^2 w^2))$ so at most $n\exp(-\Omega(\varepsilon^2 w^2))$ positions are bad for this reason in expectation.

(b2) Consider a long interval I, that is not overloaded, i.e., $|I| \geq w^2$ and $|S_I| \leq (1-\varepsilon/2)|I|$. There are at most $2^{|S_I|}$ sets S' of keys with $\mathrm{pos}(S') = I$ and each is a dependent set with probability $2^{-|I|}$ because each of the $|I|$ positions of I that S' touches imposes one parity condition.
A union bound on the probability for I to support at least one dependent set is therefore $2^{-|I|} \cdot 2^{|S_I|} \leq 2^{-\frac{\varepsilon}{2}|I|} = \exp(-\Omega(\varepsilon|I|))$.
Similarly as in **(b1)** for $i \in [m]$ we can sum this probability over all admissible lengths $|I| \geq w^2$ and all offsets that i can have in I to bound the probability that i is bad due to **(b2)**.

(b3) Let $S_{\mathrm{red}} \subseteq S^+$ be the set of redundant keys. By Theorem 5.3 we have $\mathbb{E}[|S_{\mathrm{red}}|] = n \cdot \exp(-\Omega(\varepsilon w))$.
Now if i is bad due to **(b3)** then $i \in \mathrm{pos}(S')$ for some minimal dependent set S' with short $\mathrm{pos}(S')$. At least one key from S' is redundant (regardless of the insertion order). In particular, i is within short distance ($< w^2$) of the starting position of a redundant key x. Therefore at most $|S_{\mathrm{red}}| \cdot 2w^2$ positions are bad due to **(b3)**, which amounts to $n \cdot \exp(-\Omega(\varepsilon w))$ positions in expectation as desired.

Next we will apply Theorem 5.5 to translate between the number B of bad positions and the number B' of keys $x \in S^+$ for which $s(x)$ is bad. A separate term is needed for a failure event fail of that lemma, which occurs with probability $n^{-\omega(1)}$. If Theorem 5.5 succeeds, we get $B' \leq B \cdot \mathcal{O}(1 + \log \frac{n}{B})$ and use Jensen's inequality for the concave function $x \mapsto x\log(n/x)$ to obtain

$$\begin{aligned}\mathbb{E}[B'] &\leq \mathbb{E}[B' \cdot [\mathrm{fail}]] + \mathbb{E}[B' \cdot (1 - [\mathrm{fail}])] \leq n \cdot \Pr[\mathrm{fail}] + \mathbb{E}[B \cdot \mathcal{O}(1 + \log \tfrac{n}{B})] \\ &\overset{\text{Jensen}}{\leq} n^{-\omega(1)} + \mathbb{E}[B] \cdot \mathcal{O}(1 + \log \tfrac{n}{\mathbb{E}[B]}) \\ &\leq n^{-\omega(1)} + n\exp(-\Omega(\varepsilon w))(1 + \Omega(\varepsilon w)) = n\exp(-\Omega(\varepsilon w)),\end{aligned}$$

where the last step (somewhat sloppily) uses that both occurrences of "$\Omega(\,\cdot\,)$" refer to the same function and that $\varepsilon w = \mathcal{O}(\log n)$.

Now assume that the key $y \in S^+$ we singled out is contained in a minimal dependent set S'. It follows that all of $\mathrm{pos}(S')$ is bad. Indeed, either $\mathrm{pos}(S')$ is a short interval ($\rightarrow$**(b3)**) or it is long. If it is long, then it is overloaded ($\rightarrow$**(b1)**) or not overloaded ($\rightarrow$**(b2)**). In all cases $s(y) \in \mathrm{pos}(S')$ will be bad.

Therefore, the probability p for $y \in S^+$ to be contained in a dependent set is at most the probability for $s(y)$ to be bad. Using that y is uniformly sampled from S^+ we get

$$p \leq \Pr[s(y) \text{ is bad}] = \frac{\mathbb{E}[B']}{n+1} \leq \exp(-\Omega(\varepsilon w)). \qquad \square$$

We are now ready to prove Theorem 1.8.

Proof of Theorem 1.8. We have already dealt with running times in Theorem 5.3.

Our choice $w = \frac{C\max(r, \log(1/\varepsilon))}{\varepsilon}$ ensures that $\varepsilon w > Cr$ and $\varepsilon w > C\log(1/\varepsilon)$ for a constant C of our choosing. Concerning the false positive rate we obtain

$$p \overset{\text{Lemma 5.6}}{\leq} \exp(-\Omega(\varepsilon w)) \leq \exp(-2\log(1/\varepsilon) - r) \leq \varepsilon^2 \mathrm{e}^{-r} \leq \varepsilon^2 2^{-r}$$

$$\text{and hence } \varphi \overset{\text{Lemma 5.4}}{=} p + (1-p)2^{-r} \leq p + 2^{-r} \leq \varepsilon^2 2^{-r} + 2^{-r} = (1+\varepsilon^2)2^{-r},$$

as desired. Concerning the space overhead, recall its definition as $\frac{\textsc{space}}{\textsc{opt}} - 1$ where $\textsc{space}$ is the space usage of the filter and $\textsc{opt} = -\log_2(\varphi)n$ is the information-theoretic lower bound for filters that achieve false positive rate φ. We have:

$$\begin{aligned} \text{OPT} &= -\log_2(\varphi)n \geq -\log_2(\tfrac{1+\varepsilon^2}{2^r})n = (r - \log_2(1+\varepsilon^2))n \geq (r-\varepsilon^2)n \\ \text{and SPACE} &= rm = r(m-w+1) + O(rw) = \tfrac{rn}{1-\varepsilon} + O(rw) \\ \text{which yields } \frac{\text{SPACE}}{\text{OPT}} &= \frac{r}{(1-\varepsilon)(r-\varepsilon^2)} + O(\tfrac{w}{n}) \leq \frac{1}{(1-\varepsilon)(1-\varepsilon^2)} + O(\tfrac{w}{n}) \leq 1 + 3\varepsilon, \end{aligned}$$

where the last step uses $\varepsilon \leq \frac{1}{2}$ and $n \gg \frac{w}{\varepsilon}$. □

6 Bumped Ribbon Retrieval (BuRR)

With this section we return to the topic of retrieval data structures. We describe and analyze *BuRR*, which is the second main technical contribution of this article. It adds several new twists to Standard Ribbon Retrieval from Section 4. The main effect will be that the required ribbon width is reduced to a constant that only depends on the targeted space efficiency.

The following has already been sketched in Section 1.3. The ribbon solving approach manages to insert most rows (representing most keys of S) even when w is small. Thus, by eliminating those rows/keys that cause a linear dependence, we obtain a compact retrieval data structure for a large subset of S. The remaining keys are *bumped*, meaning they will be handled by a fallback or *backyard* data structure, which, by recursion, can be a BuRR data structure again. We will show that only $O(\frac{n \log w}{w})$ keys need to be bumped in expectation. Thus, after a constant number of layers (we use four layers), a less ambitious retrieval data structure can be used to handle the few remaining keys without bumping. *Metadata* are needed to keep track of the bumped keys in each layer, so that queries can be carried out.

6.1 The Central Ideas

BuRR gets off the ground by two ideas beyond the simple "bump to eliminate linear dependencies".

First Idea: Reducing Metadata by Bumping in Ranges. The main challenge is that we need additional metadata to encode which keys are bumped. Bumping and recording in the metadata exactly the keys that create a linear dependency causes too large an overhead. Thus, a crucial observation for BuRR is that bumping should be done with granularity coarser than per-key, which results in bumping more keys, but reducing the metadata. Firstly, we will bump keys based on their starting position and say *position i is bumped* to indicate that all keys with $s(x) = i$ are bumped. Bumping by position will turn out to be sufficient because linear dependencies in A are largely unrelated to the actual bit patterns $c(x)$ but are mostly caused by fluctuations in the number of keys mapped to different positions in the range $[m]$. Secondly, instead of bumping keys from single positions, we choose to bump keys from carefully selected *consecutive ranges* of positions in overcrowded parts of the system. Our analysis will show that we can get by with a very small spectrum of possible bumping ranges, which will drastically reduce the space for the metadata.

Second Idea: Cutting Down on Empty Slots by Overloading. Space overhead results not only from metadata, but also from the $m - n + n_b$ slots of the matrix M that remain free, where n_b is the number of bumped keys. In experiments with implementations based on the ideas just mentioned it could be observed that for $\varepsilon = \frac{m-n}{n} > 0$ the overhead due to such free slots is always $\Omega(1/w)$ and thus will dominate the overhead due to metadata. However, we will show that by choosing $\varepsilon < 0$ (of order $-\varepsilon = O(\frac{\log w}{w})$), i.e., by slightly *overloading* the table, we can almost completely eliminate free slots in M, so that the minuscule amount of metadata becomes the dominant remaining overhead.

Deciding what to Bump, and Insertion Order. There are many ways to decide and encode which keys are bumped. Here, we outline a simple variant, which achieves very good performance in

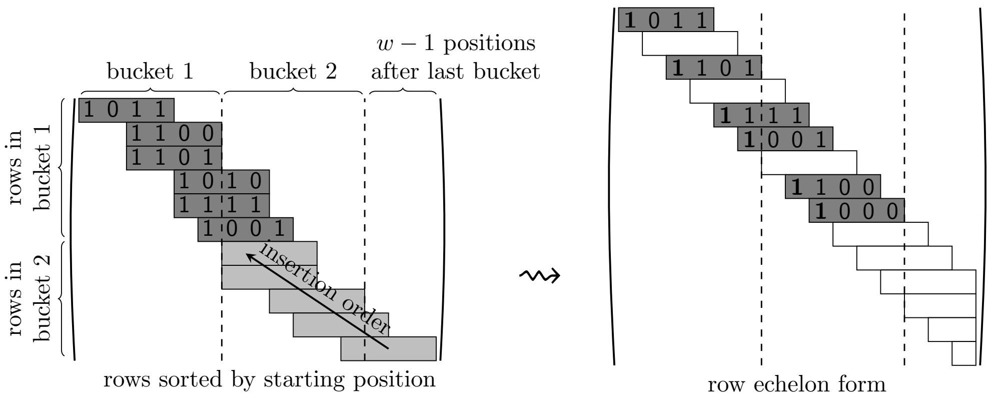

Fig. 4. Illustration of BuRR construction with $n = 11$ keys, $m = 2b + w - 1 = 15$ table positions, ribbon width $w = 4$ and bucket size $b = 6$. Keys of the first bucket were successfully inserted (from right to left) into row echelon form with two insertions "overflowing" into the second bucket. Insertions of the second bucket's rows will be attempted next, in the indicated order.

practice and is a generalization of the approach we will analyze theoretically. It will turn out to be crucial that in Standard Ribbon Retrieval we can choose the insertion order arbitrarily and that we can roll back the insertions carried out last.

The method is as follows: The pairs $(x, f(x))$, $x \in S$, are sorted according to the starting position $s(x)$ (and some order on S to break ties). We use a fast in-place integer sorter for this purpose [3]. We subdivide the possible starting positions in $[m]$ into *buckets* of width $b = O(w^2/\log w)$. For each bucket, it is only allowed to bump nothing, keys from initial ranges of the bucket of certain predefined lengths, or all keys from the bucket. The buckets are processed one after the other *from left to right*, i.e., in increasing order of $s(x)$ values. Within a bucket, however, keys are inserted into matrix M from *right to left*, i.e., in decreasing order. The intuitive reason for this choice is that insertions from the previous bucket may have "spilled over", causing additional load on the left of the bucket—an issue we wish to confront as late as possible. See also Figure 4.

If all keys of a bucket can be successfully inserted, no key of the bucket is bumped. Otherwise, suppose the first failed insertion for a bucket $[i, i + b)$ concerns a key where $s(x) = i + k$ is the $(k + 1)$th position of the bucket. We *could* decide to bump all keys x' of the bucket with $s(x') \le i + k$. This would require storing the *threshold* k using $O(\log w)$ bits and cause an overhead of $O((\log^2 w)/w^2)$ due to metadata. Instead, to reduce this overhead to $O((\log w)/w^2)$, we only allow a constant number of possible threshold values. Then, we find the smallest threshold value $t \ge k$ with t representable by metadata and bump all keys x' with $s(x') \le i + t$ from the bucket. This requires rolling back the insertions of keys x' with $s(x') - i \in [k, t]$. Recall from the end of Section 2 that this can be effected by clearing the most recently filled slots in matrix M. One good compromise between space and speed stores two bits per bucket encoding the threshold values $\{0, \ell, u, b\}$, for suitable ℓ and u. The special case $\ell = u = \frac{3}{8}w$ leads to three threshold values and a particularly clean analysis; we will thus use it below.

With these ingredients, we obtain Theorem 1.6 as stated in Section 1.3 and restated here.

THEOREM 6.1 (RESTATEMENT OF THEOREM 1.6). *An r-bit BuRR data structure with ribbon width $w = O(\log n)$ and $r = O(w)$ has space overhead $O(\frac{\log w}{rw^2})$, query time $O(1 + \frac{rw}{\log n})$, and expected construction time $O(nw)$.*

We remark that our best implementations are configured somewhat differently and many variants are possible, see Section 7.

Running Times. The theorem implies constant query time[22] if $rw = O(\log n)$ and linear construction time if $w \in O(1)$. For wider ribbons, construction time is slightly super-linear. However, in practice this does not necessarily mean that BuRR is slower than other approaches with asymptotically better bounds, as the factor w involves operations with very high locality. An analysis in the external memory model reveals that BuRR construction is possible with a single scan of the input and integer sorting of n objects of size $O(\log n)$ bits; see Section 7.4.

6.2 Proof of Theorem 1.6

Consider Algorithm 6 for construction and Algorithm 7 for queries. In the algorithms, we use the constant C from Theorem 5.2. For n keys, we overload our system M by giving it $m = n/(1 + \frac{C \log w}{8w}) + w - 1$ rows[23]. The $m - w + 1$ possible starting positions are partitioned into *buckets* of size $b = \frac{w^2}{C \log w}$. The first $\frac{3}{8}w$ positions of a bucket are called its *head*, and the larger rest is called its *tail*. Keys implicitly belong to (the head or tail of) a bucket according to their starting position. This partition into buckets with their heads and tails is also transferred to the slots in system M. The keys in S are partitioned into buckets, and head and tail sets within each buckets, in one run through the input. The buckets are treated one after the other, in order of increasing number, trying to insert their keys into the system M. When a bucket is treated, it is possible that some of its keys are bumped. For this, the algorithm has three choices:

(1) No keys belonging to the bucket are bumped.
(2) The keys belonging to the head of the bucket are bumped.
(3) All keys of the bucket are bumped.

These choices are made greedily as follows. When a bucket is treated, we first try to insert the keys belonging to its tail, in arbitrary order. If at least one insertion fails, all previous insertions from the bucket are undone ("rolled back") and the entire bucket is bumped, i.e., Option 3 is used. Otherwise, we also try to insert the keys belonging to the bucket's head. If all these insertions succeed, we choose Option 1, otherwise all insertions of head keys are undone and we choose Option 2. It is easy to undo any number of insertions that have been carried out last, in constant time for each key, by just erasing the entries in M they have created. (Here we exploit the "incremental" nature of Algorithm 1, discussed in Section 2.) Trying to insert a set X of keys and rolling back as soon as the first failure occurs is done by a procedure call insertAll(X), which returns a Boolean value to indicate success or failure, respectively. Which of the three options was chosen for bucket j is taken down in meta[j], for a three-valued array meta[1..#buckets]. Back substitution is carried out for all keys that were successfully inserted. The set of these keys is called $S \setminus S_{\text{bumped}}$, for S_{bumped} the set of bumped keys. Then $|S \setminus S_{\text{bumped}}| = |P_M|$, for P_M the set of occupied slots. The keys in S_{bumped} are inserted in the backyard data structure D_{bumped}. For a query on $x \in \mathcal{U}$, the starting position $s(x)$ together with the information in meta[$\lceil s(x)/b \rceil$] is sufficient to decide whether the primary structure, based on M, or the backyard structure D_{bumped} should be used. The main ingredient in the analysis of the construction algorithm is the following lemma, which we prove later in this section.

[22]It should be noted that the proof invokes a lookup table in one case to speed up the computation of a matrix-vector product. In Section 1.8, we argue that lookup tables should be avoided in practice. Technically, our implementation using *interleaved representation* (see Section 7) has a query time of $O(r)$.
[23]We ignore rounding issues for a clearer presentation and assume that w is large. The latter assumption creates a certain discrepancy between our analysis and practical applications, where concrete values like $w = 32$ are used.

LEMMA 6.2. *The expected fraction of empty slots in M is $O(w^{-3})$.*

If the fraction of empty slots turns out to be significantly higher than expected, we simply restart the construction with new hash functions, and repeat until the fraction of empty slots is below a predetermined bound $O(w^{-3})$ (this is not reflected in Algorithm 6). After back substitution, we obtain a solution vector of mr bits. Additionally, we need to store the choices regarding the bumped ranges that were made by the algorithm. This takes no more than $\lceil \log_2 3 \rceil = 2$ bits of metadata per bucket. The total space SPACE for the primary table is then bounded by $mr + 2\frac{m}{b}$, the optimal space OPT is $|P_M| \cdot r = m \cdot (1 - O(w^{-3})) \cdot r$, which yields the following bound for the space overhead:

$$\frac{\text{SPACE}}{\text{OPT}} - 1 \leq \frac{mr + 2\frac{m}{b}}{|P_M| \cdot r} - 1 \leq \frac{1 + 2\frac{1}{rb}}{1 - O(w^{-3})} - 1 = \left(1 + \tfrac{2}{rb}\right)\left(1 + O(w^{-3})\right) - 1$$
$$= O\left(\tfrac{1}{rb}\right) + O(w^{-3}) = O\left(\tfrac{\log w}{rw^2}\right) + O(w^{-3}) = O\left(\tfrac{\log w}{rw^2}\right).$$

The last step uses the assumption $r = O(w)$. The trivial bound in Theorem 3.4 implies that $O(bw)$ row operations are performed during the *successful* insertions in a bucket. There can be at most one failed insertion for each bucket, which takes $O(b)$ row operations since insertions cannot extend past the next (still empty) bucket. Since $w = O(\log n)$ bits fit into one word of a word RAM, each of these takes constant time, and all of them together take $O(\frac{m}{b} \cdot (bw + b)) = O(nw)$ time. Back substitution has the same complexity as n queries and therefore takes $O(n(1 + \frac{wr}{\log n})) = O(nw)$ time.

A query for a non-bumped key involves computing the product of the w-bit vector $c(x)$ and a block $Z(x)$ of $w \times r$ bits from the solution matrix $Z \in \{0,1\}^{m \times r}$. The wr bit operations can be carried out in $O(1 + \frac{wr}{\log n})$ steps on a word RAM with word size $\Omega(\log n)$. A complication is that if $w, r \in \omega(1) \cap o(\log n)$, we are forced to handle several rows of $Z(x)$ in parallel (Xoring a $c(x)$-controlled selection) or several columns of $Z(x)$ in parallel (bitwise AND with $c(x)$ and popcount). For such numbers "much bigger than 1 and much smaller than $\log n$", which is a concern not appearing in practice, we offer a theoretical resolution (not reflected in our implementation[24]): We resort to the standard technique of tabulating the results of a suitable set of vector-matrix products.

To complete the construction, we still have to deal with the $n - |P_M| = n - m(1 - O(w^{-3})) = O(\frac{n \log w}{w})$ bumped keys. A query can easily identify from the metadata whether a key is bumped, so all we need is another retrieval data structure, that is consulted in this case. We can recursively use BuRR again. In order to avoid compromising worst-case query time we only do this for four levels. Let $S^{(4)}$ be the set of keys bumped four times. We have $|S^{(4)}| = O(n\frac{\log^4 w}{w^4}) = O(n\frac{\log w}{rw^2})$, so we can afford to store $S^{(4)}$ using a retrieval data structure with constant overhead, linear construction time and $O(1)$ worst-case query time, e.g., using MPHF [11].[25]

Improved Construction Time Using Table Lookups. To facilitate a comparison with theoretical results like [52], we now show that BuRR can likewise be improved by using lookup tables, thus proving Theorem 1.7. Assuming $r = O(\sqrt{\log n})$, we can achieve expected construction time $O(n)$, query time $O(1)$ and space overhead $O(\frac{\log \log n}{r \log n})$. Note that this twist is not reflected in our implementation.

Setting $w = \Theta(\sqrt{\log n})$ in Theorem 1.6 immediately gives the desired query time and space overhead. We now sketch how the construction time can be reduced to linear. Consider Algorithm 1

[24]Certain vector instructions such as VPOPCNTDQ in the AVX-512 instruction set, which counts the number of 1's in binary words of 512 bits, may be useful here.
[25]Our implementation is optimized for $w = \Omega(\log n)$ and can simply use ribbon retrieval with an appropriate $\varepsilon > 0$ for the last level.

ALGORITHM 6: The construction algorithm of BuRR.

```
Input: $f: S \to \{0,1\}^r$ for some $S \subseteq \mathcal{U}$ of size $n$.
Parameters: $w \in \mathbb{N}$.
$m \leftarrow n/(1+\frac{C\log w}{8w}) + w - 1$; allocate system $M$ of size $m$ // constant $C$ as in Theorem 5.2
pick hash functions $s: \mathcal{U} \to [m-w+1]$, $c: \mathcal{U} \to \{1\} \times \{0,1\}^{w-1}$
$b \leftarrow \frac{w^2}{C\log w}$, #buckets $\leftarrow \frac{m-w+1}{b}$ // bucket size, number of buckets
for $j \in$ [#buckets] do // partition
    $B_j \leftarrow \{x \in S \mid \lceil s(x)/b \rceil = j\}$ // $j$th bucket
    $H_j \leftarrow \{x \in B_j \mid s(x) - (j-1)b \le \frac{3}{8}w\}$ // head of $j$th bucket
    $T_j = B_j \setminus H_j$ // tail of $j$th bucket
$S_{\text{bumped}} \leftarrow \emptyset$ // set of bumped keys
for $j \in$ [#buckets] do
    // insertAll($X$) inserts all $x \in X$ using Algorithm 1, on first failure everything is rolled back. Return value *true*
       indicates success.
    if insertAll($T_j$) then
        if insertAll($H_j$) then
            meta[$j$] $\leftarrow$ bumpNothing
        else
            $S_{\text{bumped}} \leftarrow S_{\text{bumped}} \cup H_j$
            meta[$j$] $\leftarrow$ bumpHead
    else
        $S_{\text{bumped}} \leftarrow S_{\text{bumped}} \cup B_j$
        meta[$j$] $\leftarrow$ bumpAll
$Z \leftarrow 0^{m\times r}$
for $i = m$ down to 1 do
    $Z_i \leftarrow M.b[i] \oplus M.c[i] \cdot Z_{i..i+w-1}$ // back substitution
$D_{\text{bumped}} \leftarrow$ construct($S_{\text{bumped}}$) // recursive, unless base case reached
return $D = (s, c, Z, \text{meta}, D_{\text{bumped}})$
```

ALGORITHM 7: The query algorithm of BuRR.

```
Input: $x \in \mathcal{U}$, data structure $(s, c, Z, \text{meta}, D_{\text{bumped}})$.
$j \leftarrow \lceil s(x)/b \rceil$, offset $\leftarrow s(x) - (j-1)b$ // using bucket size $b$ as in Algorithm 6
if meta[$j$] = bumpNothing $\vee$ (meta[$j$] = bumpHead $\wedge$ offset $> \frac{3}{8}w$) then
    return $c(x) \cdot Z_{s(x)..s(x)+w-1}$
else
    return query($x, D_{\text{bumped}}$)
```

at the start of its loop. In order to execute many steps simultaneously, i.e., using bit-parallelism, we collect relevant data in a single machine word X. Concretely, X should encode the values c and b for k keys as well as $M.c[i..i+k]$ and $M.b[i..i+k]$, for $k = \Theta(\sqrt{\log n})$. By choosing the constants in a suitable way, the length of X can be made to be at most $\frac{\log n}{2}$ bits. There is a function f such that $Y = f(X)$ encodes the state of Algorithm 1, that is as far in the future as the data contained in X permits. The options are:

(1) $Y = (\text{ONGOING}, i' - i, c', b')$. Execution is back at the beginning of the loop with updated content i', c', b' in the local variables and $i' - i = \Omega(\sqrt{\log n})$. (Here, further values from $M.c$ and $M.b$, not contained in X, would be needed to continue.)

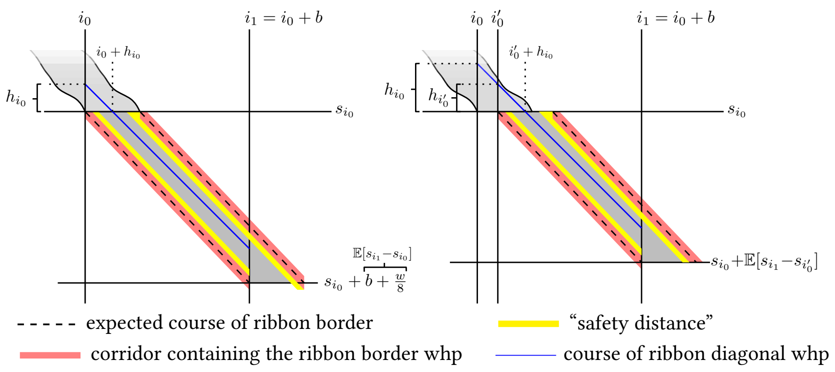
i_0
$i_1 = i_0 + b$
$i_0 + h_{i_0}$
h_{i_0}
s_{i_0}
$\mathbb{E}[s_{i_1} - s_{i_0}]$
$s_{i_0} + b + \frac{w}{8}$
i_0 i'_0
$i'_0 + h_{i_0}$
$h_{i'_0}$
$s_{i_0} + \mathbb{E}[s_{i_1} - s_{i'_0}]$
expected course of ribbon border
corridor containing the ribbon border whp
"safety distance"
course of ribbon diagonal whp

Fig. 5. Situation in Cases 1 (left) of 2 (right) of the proof of Claim 1.

(2) $Y = (\textsc{success}, i' - i, c', b')$. A suitable insertion slot has been found, i.e., line 4 has been reached.
(3) $Y = \textsc{redundant}$ or $Y = \textsc{failure}$. The corresponding return statements have been reached.

Tabulating f requires $\widetilde{O}(2^{\log(n)/2}) = o(n)$ space and time. Each access to f allows us to either reach the end of Algorithm 1 immediately or it allows us to advance by $i' - i = \Omega(\sqrt{\log n})$ inserted rows in a single step, in the sense used in our upper bound Equation (4). The corresponding speedup is transferred to Theorem 3.4 and to Theorem 1.6 for an overall construction time of $O(nw/\sqrt{\log n}) = O(n)$ as desired. We omit the details.

6.3 Proof of Lemma 6.2

We give an induction-like argument showing that "most" buckets satisfy two properties:

(P1) All slots of the bucket are filled.
(P2) The height $h_i := s_i - d_i$ of the ribbon diagonal over the lower ribbon border at the last position i of the bucket satisfies $h_i \geq \frac{w}{4}$.[26]

Claim 1. *If* **(P2)** *holds for a bucket* B_0 *then* **(P1)** *and* **(P2)** *hold for the following bucket* B_1 *with probability* $1 - O(w^{-3})$.

Proof. Let i_0 and i_1 be the last positions of buckets B_0 and B_1, respectively. By **(P2)** for B_0 we have $h_{i_0} \geq \frac{w}{4}$.

Case 1: $h_{i_0} < \frac{5}{8}w$. We claim that with probability $1 - O(w^{-3})$ all keys belonging to B_1 (head and tail) can be inserted and that **(P1)** and **(P2)** are fulfilled afterwards. The situation is illustrated in Figure 5 on the left.
The dashed black lines show the expected trajectories of the ribbon borders for bucket B_1. The lower expected border travels in a straight line from (s_{i_0}, i_0) to the point, that is $b = i_1 - i_0$ positions to the right and $\mathbb{E}[|\{x \in S \mid s(x) \in B_1\}|] = \frac{n}{m-w+1}b = (1+\frac{C\log w}{8w})b = b + \frac{w}{8}$ positions below. The actual position of the border randomly fluctuates around the expectation. At each point the (vertical) deviation exceeds $\frac{w}{8}$ with probability at most $O(w^{-5})$ by Theorem 5.2 **(b)**. A union bound shows that it is at most $\frac{w}{8}$ *everywhere in the bucket* (and thus within the region shaded red in Figure 5 on the left) with probability at least $1 - O(w^{-3})$. In other words,

[26]The key set underlying the definitions of s_i and d_i excludes the bumped keys.

we have

$$s_i = s_{i_0} + \tfrac{i-i_0}{b} \cdot (b + \tfrac{w}{8}) + \mathrm{err}_i \text{ for } i \in B_1, \text{where } \mathrm{err}_i \in [-\tfrac{w}{8}, \tfrac{w}{8}]. \tag{10}$$

The region shaded yellow represents a (vertical) "safety distance" of another $\frac{w}{8}$ that we wish to keep from the ribbon borders. Finally, the blue line $\tilde{d}$ is a perfect diagonal starting from (d_{i_0}, i_0) with vertical position $\tilde{d}_i = d_{i_0} + i - i_0$ in column i.
The height $\tilde{h}_i = s_i - \tilde{d}_i$ of the blue line above the bottom ribbon border satisfies

$$\tilde{h}_i = s_i - \tilde{d}_i = s_i - d_{i_0} + i - i_0 \overset{\text{Equation (10)}}{=} h_{i_0} + \tfrac{i-i_0}{b} \cdot \tfrac{w}{8} + \mathrm{err}_i. \tag{11}$$

Now our arguments nicely interlock. As long as no insertion fails and no slot remains empty, d proceeds along a diagonal path and coincides with $\tilde{d}_i$. Conversely, as long as $d_i = \tilde{d}_i$ holds we have, using Equation (11) and $h_{i_0} \in \{\frac{w}{4}, \dots, \frac{5w}{8}\}$

$$h_i = \tilde{h}_i = h_{i_0} + \tfrac{i-i_0}{b} \cdot \tfrac{w}{8} + \mathrm{err}_i \in \{\tfrac{w}{8}, \dots, \tfrac{7w}{8}\}.$$

Then, due to $h_i \geq \frac{w}{8}$ we see no empty slots by Theorem 3.2 and due to $h_i \leq \frac{7}{8}w$ we see no failed insertions by Theorem 3.3 **(b)**, unless corresponding events with probability $2^{-w/8}$ occur.
This establishes **(P1)** with probability $1 - O(w^{-3})$. Then **(P2)** follows easily: The extreme case is when both $h_{i_0} = \frac{w}{4}$ and $|\{x \in S \mid s(x) \in B_1\}| = b$ take the minimum permitted values. In that case we have $h_{i_1} = \frac{w}{4}$, so in general $h_{i_1} \geq \frac{w}{4}$ follows.

Case 2: $h_{i_0} \geq \frac{5}{8}w$. We claim that with probability $1 - O(w^{-3})$ all keys belonging to the tail of B_1 can be inserted and that afterwards **(P1)** and **(P2)** are fulfilled. In case the keys in the head of B_1 can also be successfully inserted this cannot hurt **(P1)** or **(P2)** because the set of filled slots can only be extended and the height can only increase due to the additional keys.[27]
The situation is illustrated in Figure 5 on the right. We only consider the keys in the tail of B_1 which starts at position $i_0' + 1$ where $i_0' = i_0 + \frac{3}{8}w$. Note that for $i \in [i_0, i_0']$ the ribbon diagonal (d_i, i) follows an ideal diagonal trajectory with probability $1 - O(2^{-\Omega(w)})$ since keys from B_0 are successfully inserted and the distance to the bottom border is always at least $\frac{1}{4}w$. This implies that all slots in the head of B_1 are filled by keys from B_0 and $h_{i_0'} = h_{i_0} - \frac{3}{8}w$. Since $h_{i_0} \in [\frac{5}{8}w, w]$ we have $h_{i_0'} \in [\frac{1}{4}w, \frac{5}{8}w]$, which allows us to reason as in Case 1 to show that all slots in the tail of bucket B_1 are filled and $h_{i_1} \geq \frac{w}{4}$ with probability $1 - O(w^{-3})$. □

Handling failures and the first bucket. The following claim is needed to deal with the (rare) cases where Claim 1 does not apply.

Claim 2. *Assume B_1 is either the first bucket or is preceded by a bucket B_0 for which* **(P2)** *does not hold. Then with probability $1 - O(w^{-3})$ all keys of B_1 (head and tail) are successfully inserted.*

Proof. The ribbon diagonal d starts at a height $h_{i_0} < \frac{w}{4}$, which is lower than desired, and might hit the lower ribbon border within B_1. However, d avoids the right ribbon border, because otherwise (recycling ideas from Case 1 of Claim 1) d would have to pierce the diagonal starting at the desired height $\frac{w}{4}$ first and would afterwards stay on that diagonal with probability $1 - O(w^{-3})$. This situation might cause some slots in B_1 to remain empty but it implies all keys are successfully inserted with probability $1 - O(w^{-3})$. □

[27]Note that our analysis suggests that B_1 is already full after the keys from the tail of B_1 are inserted, which means that the keys from the head can only be inserted if they all "overflow" into the next bucket.

We now classify the buckets. Consider a bucket B_1. Note that either Claim 1 or Claim 2 is applicable. If the corresponding event with probability $1 - O(w^{-3})$ fails to occur or if B_1 contains fewer than b keys (this happens with probability $O(w^{-5})$), then B_1 is a *bad bucket*. Otherwise, if Claim 1 applies, then B_1 is a *good bucket* and if Claim 2 applies then B_1 is a *recovery bucket*. A *recovery sequence* is a maximal contiguous sequence of recovery buckets. Since such a sequence cannot be preceded by a good bucket, the number of recovery sequences is at most the number of bad buckets plus 1 (the first bucket is always a recovery bucket or bad). Only bad buckets contain fewer than b keys, so a recovery sequence of k buckets contains at least kb keys, all of which are inserted successfully by Claim 2. At most $w - 1$ of these insertions fill slots after the sequence so there are at most $w - 1$ empty slots within a recovery sequence. With x denoting the number of bad buckets, the number $m - |P_M|$ of empty slots in total is:

$$m - |P_M| \leq xb + (x + 1)(w - 1) + w - 1,$$

where the last $w-1$ accounts for slots $[m-w+1, m]$ that do not belong to any bucket. The dominating term is xb so using $\mathbb{E}[x] = O(\frac{m}{b}w^{-3})$ we obtain $\mathbb{E}[\frac{m-|P_M|}{m}] = O(w^{-3})$, which completes the proof of Theorem 6.2.

7 Details and Variants of BuRR

BuRR as analyzed in Section 6 is a relatively simple data structure. However, the actual design space of BuRR is much larger, and we explored a considerable part of it before arriving at the simple approach presented above. There was a fruitful back and forth between design ideas, experimental observations, and theoretical considerations much in the spirit of algorithm engineering [54]; see [26] for more details.

In the following, we only highlight a few refinements and implementation details as well as variants that may be useful when applying the data structure in various situations. We do not attempt to formally justify any of these details and variants.

7.1 Threshold-Based Bumping

The BuRR variant analyzed in Section 6 assumes a threshold with three possible values $(0, t, b)$ to be stored with each bucket. Our implementation looks at several generalizations of this approach with two standing out as a good tradeoff between performance and space consumption:

2-bit Thresholds. The obvious and fast way to store a three-way threshold encodes this into 2 bits. This gives room for four threshold values $(b, u, \ell, 0)$ without additional space overhead. Indeed, this enables lower space consumption as it allows for larger buckets or higher overloading. Refer to the TR [26] for a discussion how to choose the threshold values.

1^+*-Bit Thresholds.* The analysis suggests that the case of bumping the entire bucket is a rare case so that a variable-bit-length encoding of the threshold variable suggests itself to achieve best space efficiency. Our implementation stores a single bit for every bucket encoding the threshold values 0 and t. Additionally, a small hash table stores buckets that require a threshold value in $\{t + 1, \dots, b\}$, allowing arbitrary values from that set.

7.2 Table Representation

The natural way of storing the solution matrix $Z \in \{0, 1\}^{m \times r}$ is row-wise, i.e., using m words of r bits each. We call this *contiguous* storage. However, we propose to use *interleaved* storage in most cases, where the matrix is horizontally sliced into m/w sub-matrices of size $w \times r$, which are stored column-wise; see Figure 6. This organization allows the extraction of one retrieved bit from two machine words (that are r positions apart) using population-count instructions. Interleaved representation

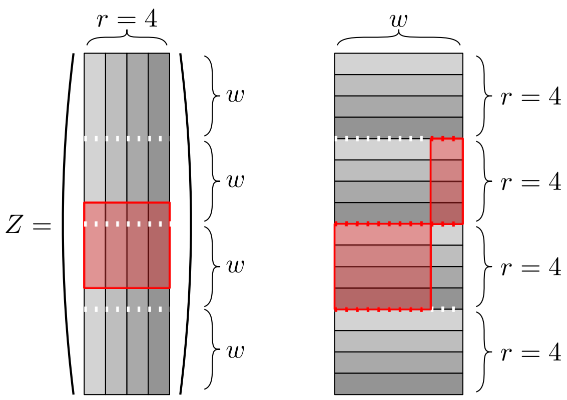

Fig. 6. A solution matrix (left) and w-bit interleaved column-major layout (right) of that matrix. The red shaded region shows the bits used in a query crossing a boundary.

is advantageous for uses of BuRR as an AMQ data structure since a negative query (i.e., for a key not in the AMQ) only has to extract two bits in expectation. Moreover, the implementation directly works for any value of r.

Contiguous storage, despite its conceptual simplicity, is more difficult to implement, in particular when r is not a power of two. On the other hand, we expect it to be more efficient when all r bits need to be retrieved, in particular when r is large and when the implementation makes careful use of available bit-manipulation[28] and SIMD instructions. This is particularly true when the sparse bit patterns from Section 7.5 are used.

In the TR [26] we also outline how to further reduce cache faults by keeping metadata and retrieval information in the same memory blocks.

7.3 Parallel Construction

As a static data structure, queries are trivial to parallelize on a shared-memory system. Parallelizing construction can use sharding, i.e., subdividing the data structure into pieces that can be built independently. An interesting property of BuRR is that sharding can be done transparently in a way that does not affect query implementation and thus avoids performance overheads for queries. To subdivide the data structure, we set the shard boundaries and bumping information such all keys with equations spanning a shard boundary are bumped. The equation system can then be solved independently for each shard.

Note that the analysis from Section 6 suggest to choose bucket heads of size $\Theta(w)$. Together with our 2-bit threshold encoding variant, one can choose w as one possible threshold, allowing particularly seamless parallelization. When using 1^+-Bit thresholds, one is free to choose appropriate thresholds anyway. In the mean time this has also been implemented [4].

We can hope for a time-efficient and space efficient parallelization on p threads provided that $n/p \gg \max(w^2/\log w, \log n)$—each shard should have at least a constant number of buckets and needs $\Omega(\log n)$ elements to achieve good load balance during construction.

7.4 External Memory Construction

For large BuRRs that fill a significant part of main memory, space consumption during construction can be a major limitation. Once more, sharding is a simple way out—by constructing the data

[28]For example, the BMI2 bit manipulation operations PDEP and PEXT in newer x86 processors look useful.

structure one shard at a time, the temporary space overhead for construction is only proportional to the size of a shard rather than to the size of the entire data structure.

An alternative is to consider a "proper" external memory algorithm. The input is a file F of n key-value pairs. The output is a file containing the layers of the BuRR data structure. A difficulty here is that keys could be much larger than $\log n$ bits and that random accesses to keys is much more expensive than just scanning or sorting. We therefore outline an algorithm that, with high probability, reads the key file only once. The trick is to first substitute keys by ***master hash codes*** (**MHCs**) with $c \log n$ bits for a constant $c > 2$. Using the well-known calculations for the birthday problem, this size suffices that the MHCs are unique with high probability. If a collision should occur, the entire construction is restarted.[29] Otherwise, the keys are never accessed again—all further hash bits are obtained by applying secondary hash functions to the MHC.[30]

Construction then begins by scanning the input file F and producing a stream F_1 of pairs (MHC, function value)[31]. Construction at layer i amounts to reading a stream F_i of MHC-value pairs. This stream F_i is then sorted by a bucket ID, that is derived from the MHCs. A collision is detected when two pairs with the same MHC show up. These are easy to detect since they will be sorted to the same bucket and the same column within that bucket. Constructing the row echelon form (REM) then amounts to simultaneously scanning F_i, the REM and the right-hand side. At no time during this process do we need to keep more than two buckets worth of data in main memory. Bumped MHC-value pairs are output as a stream F_{i+1}, that is the input for construction of the next layer. Back-substitution amounts to a backward scan of the REM and the right-hand side—producing the table for layer i as an output.

Overall, the I/O complexity is the I/Os for scanning n keys plus sorting $O(n)$ items consisting of a constant number of machine words ($O(\log n)$ bits). The fact that there are multiple layers contributes only a small constant factor to the sorting term since these layers are shrinking geometrically with a rather large shrinking factor.

7.5 Sparse Bit Patterns

A query to a BuRR data structure as described so far needs to process $\Omega(rw)$ bits of data to produce r bits of output. Despite the opportunity for bit parallelism, this can be a considerable cost. It turns out that BuRR can also operate with significantly sparser bit patterns. We can split w bits into k groups and set just one randomly chosen bit per group. The downside of sparse patterns is that they lead to more linear dependencies. This will induce more bumped keys and possibly more empty slots. Our experiments indicate that the compromise is quite interesting. For example, for $w = 64$ and $k = 8$ we can reduce the expected number of 1-bits by a factor of four and observe faster queries for large r at the cost of a slight increase in space overhead. Experiments in the TR [27] indicate that our implementation of sparse coefficient BuRR is indeed a good choice for $r \in \{8, 16\}$.

7.6 Bu1RR: Accessing Only Two Layers in the Worst Case

BuRR as described so far has worst-case constant access time when the number of layers is constant (our analysis and experiments use four layers). However, for real-time applications, the resulting worst case might be a limitation. Here, we describe how the worst-case number of accessed layers can be limited to two. We call the approach *bump-once ribbon retrieval*, stylised as "Bu^{1}RR". The

[29]For use in AMQs, restarts are not needed since duplicate MHCs lead to identical equations that will be ignored as redundant by the ribbon solver.

[30]We use a (very fast) linear congruential mapping [27, 42] that, with some care, (multiplier congruent 1 modulo 4 and an odd summand) even defines a bijective mapping [42]. We also tried linear combinations of two parts of the MHC [41] which did not work well, however, for a 64 bit MHC.

[31]When used as an AMQ, the function value need not be stored since it can be derived from the MHC.

idea is quite simple: Rather than mapping all keys to the first layer, we map the keys to all layers using a distribution still to be determined. We now guarantee that a key originally mapped to layer i is either retrieved there or from layer $i + 1$. A query for key x now proceeds as follows. First the primary layer $i(x)$ for that key is determined. Then, the bumping information of layer i is consulted to find out whether x is bumped. If not, x is retrieved from layer i, otherwise it is retrieved from layer $i + 1$ *without* consulting the bumping information for layer $i + 1$.

For constructing layer i, the input consists of keys E'_i bumped from layer $i - 1$ ($E'_0 = \emptyset$) and keys E_i having layer i as their primary layer ($E_i = \emptyset$ for the last layer). First, the bumped keys E'_i are inserted bucket by bucket. In this phase, a failure to insert a key causes construction to fail. Then, the keys E_i are processed bucket by bucket as before, recording bumped keys in E'_{i+1}.

The size of layer i can be chosen as $(1 + \varepsilon)(|E'_i| + |E_i|)$ to achieve the same overloading effect as in basic BuRR. An exception is the last layer i^* where we choose the size large enough so that construction succeeds with high probability. Note that if $|E_i|$ shrinks geometrically, we can choose i^* such that $|E_{i^*}|$ is negligible.

Construction of a layer can only fail due to keys bumped from the previous layer, which is a small fraction even for practical w. In the unlikely[32] case of construction failing, construction of that layer can be retried with more space. Tests suggest that increasing space by a factor of $\frac{w+1}{w}$ has similar or better construction success probability than a fresh hash function.

A simpler Bu^{1}RR construction has layers of predetermined sizes, all in one linear system. Construction reliability and/or space efficiency are reduced slightly because of reduced adaptability. For moderate $n \approx w^3$, layers of uniform size can work well, especially if the last layer is of variable size.

Our implementation of Bu^{1}RR, tested to scale to billions of keys, uses layers with sizes shrinking by a factor of two, each with a power of two number of buckets. To a first approximation, the primary layer $i(x)$ of a key is simply the number of leading zeros in an appropriate hash value, up to the maximum layer. To consistently saturate construction with expected overload of $\alpha = -\varepsilon$, this is modified with a bias for the first layer. A portion (α) of values with a leading 0 (not first layer) are changed to a leading 1 (re-assigned to first layer), so $|E_0| \approx (1 + \alpha)2^{-1}m$ and other $|E_i| \approx (1 - \alpha)2^{-(i+1)}m$. With bumped entries, $|E'_i| \approx \alpha 2^{-i}m$, expected overload is consistent through the layers: $|E_i| + |E'_i| \approx (1 + \alpha)2^{-(i+1)}m$.

8 Experiments

We performed extensive experiments to evaluate our ribbon-based data structures and competitors. We refer to the TR [27] for detailed data and discussion and only give a summary here.

Implementation Details. We implemented BuRR and homogeneous ribbon filters in C++ using template parameters that allow us to navigate a large part of the design space. We use sequential construction using 64-bit MHCs so that the input keys themselves are hashed only once. Linear congruential mapping is used to derive more pseudo-random bits from the MHC. When not otherwise mentioned, our default configuration is BuRR with left-to-right processing of buckets, right-to-left insertion within a bucket, threshold-based bumping, interleaved storage of the solution Z, and separately stored metadata. The data structure has four layers, the last of which uses $w' := \min(w, 64)$ and $\varepsilon \geq 0$, where ε is increased in increments of 0.05 until no keys are bumped. For 1^+-bit thresholds, we choose $t := \lceil -2\varepsilon b + \sqrt{b/(1 + \varepsilon)}/2 \rceil$ and $\varepsilon := -2/3 \cdot w/(4b + w)$. For 2-bit thresholds, parameter tuning showed that $\ell := \lceil (0.13 - \varepsilon/2)b \rceil, u := \lceil (0.3 - \varepsilon/2)b \rceil$, and $\varepsilon := -3/w$ work well for $w = 32$; for $w \geq 64$, we use $\ell = \lceil (0.09 - 3\varepsilon/4)b \rceil$, $u = \lceil (0.22 - 1.3\varepsilon)b \rceil$, and $\varepsilon := -4/w$.

[32]We do not have a complete analysis of this case but believe that our analysis in Section 6 can be adapted to show that the construction process will succeed with high probability for $w = \Omega(\log n)$.

Table 2. Competitors used in our Experimental Comparison

Name	operations	note	paper
Bloom	qi	classical Bloom filters	[9]
BlBloom	qi	blocked Bloom filters	[53]
Cuckoo	qidc	Cuckoo filters	[29]
Morton	qidc	Morton filters	[12]
QF	qidc	Quotient filters	[7]
CQF	qidc	Quotient filters	[7]
Xor	q	Xor filters	[23, 35]
LMSS	q	coding based	[47]
Coupled	q	coupled hypergraph peeling	[60]
Standard	q	standard ribbon	Algorithm 2
Homog	q	homogeneous ribbon	Algorithm 4
Bu^{1}RR	q	bump once ribbon	Section 7.6
BuRR	q	bumped ribbon retrieval	Section 6

Supported operations are indicated as q/query, i/insertion, d/deletion, and c/count.

In addition, there is a prototypical implementation of Bu^{1}RR; see Section 7.6. Both BuRR and Bu^{1}RR build on the same software for ribbon solving. For validation we extend the experimental setup used for Cuckoo and Xor filters [34], with our code available at https://github.com/lorenzhs/fastfilter_cpp and https://github.com/lorenzhs/BuRR.

Experimental Setup. Since every retrieval data structure can be used as a AMQ / filter but not vice versa, our experiments are for **filters**, which admits a larger number of competitors. All experiments were run on a machine with an AMD EPYC 7702 processor with 64 cores, a base clock speed of 2.0 GHz, and a maximum boost clock speed of 3.35GHz. The machine is equipped with 1 TiB of DDR4-3200 main memory and runs Ubuntu 20.04. We use the `clang++` compiler in version 11.0 with optimization flags `-O3 -march=native`.

Figure 7 compares different AMQs for $n = 10^8$ and $\varphi \approx 2^{-8}$ (i.e., $r = 8$ for BuRR).[33] See Table 2 for a summary of the competitors. We make a number of observations for this setting and then generalize with reference to broader measurements reported in the TR [27]. Most techniques allow different configurations with different space-performance tradeoffs. We plot those that are not dominated by other configurations of the same technique.

We see that BuRR dominates all other techniques for overheads below 2 %. Homogeneous ribbon filters have slightly faster queries and construction at the price of overheads around 10 %. Filters based on hypergraph peeling (Coupled and Xor) have faster queries but slower construction and overhead between 4 and 30 %. Even at the high end of this overhead range, the queries are only a factor 2 faster than BuRR. We view this as perhaps the most interesting practical result—BuRR reduce space overhead to almost nothing at a rather moderate price in query time and achieves high construction throughput.

Blocked Bloom filters [53] achieve the fastest queries but at the price of about 50 % space overhead. The "two block" approach [22] (as mentioned in Section 1.4 (iv)), while offering fast queries and very small overhead, is only implemented for $r = 1$. It is not included here, and experiments in [27] show that its construction time is more than an order of magnitude higher than most ribbon-based approaches. The other tried filters are only competitive if their expanded set of supported operations is needed.

[33]Small deviations of parameters are necessary because not all filters support arbitrary parameter choices.

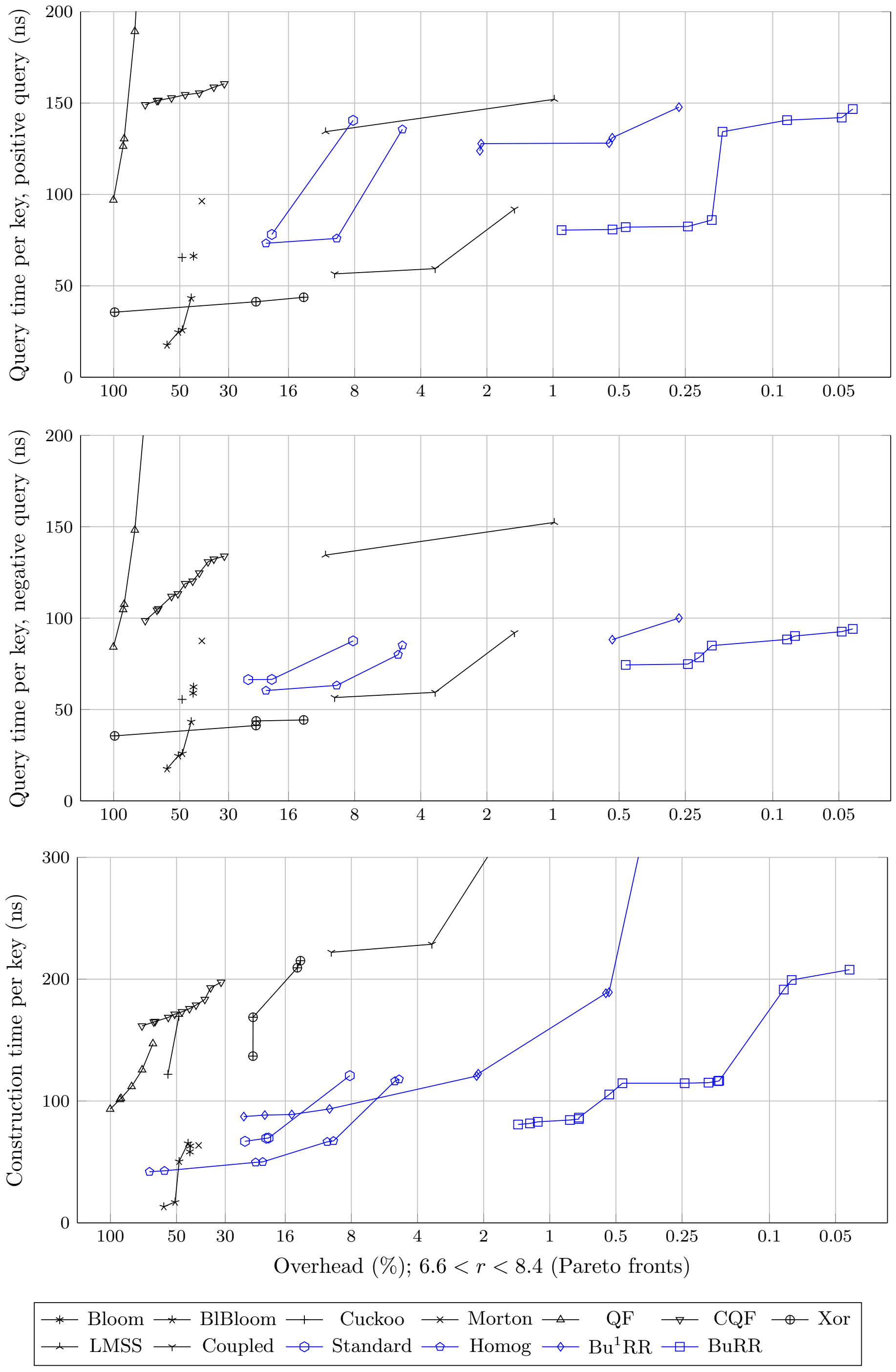

Fig. 7. Time–overhead trade-off for positive queries (i.e., the key is in the AMQ), negative queries (i.e., the key is not in the AMQ) and construction time (bottom). False-positive rate between 0.3 % and 1 % for different AMQs and $n = 10^8$ different inputs.

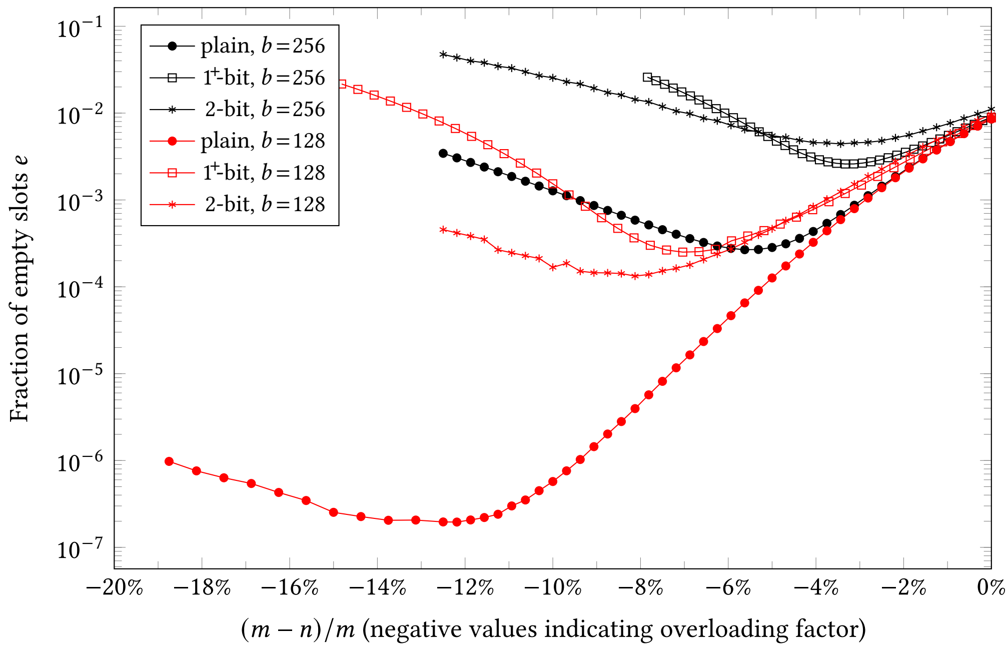

Fig. 8. Fraction of empty slots for various configurations of BuRR with $w = 64$, depending on the overloading factor. "plain" stands for metadata allowing arbitrary bumping thresholds.

Further Observations.

- **Good ribbon widths are $w = 32$ and $w = 64$.** Ribbon widths as small as $w = 16$ can achieve small overhead but at least on 64-bit processors, $w \in \{32, 64\}$ seems most sensible. The case $w = 32$ is only 15–20 % faster than $w = 64$ while the latter has about four times less overhead. Thus, the case $w = 64$ seems the most favorable one. This confirms that the linear dependence of the construction time on w is to some extent hidden behind the cache faults, which are similar for both values of w (this is in line with our analysis in the external memory model; see Section 7.4).
- **Small r favor Ribbon.** In particular, for $r = 1$ queries get considerably faster while construction time remains similar.
- **Large r favor Peeling.** Here, Coupled and Xor have an advantage regarding query time since they perform less computations. In sequential benchmarks, they can also access their required data in parallel. There are also useful approaches between ribbon and peeling. In particular for $r \in \{8, 16\}$, the sparse bit patterns mentioned in Section 7.5 perform well.
- **Ribbon excels in parallel settings.** On modern server processors, many threads typically compete for resources of the memory subsystem. In this situation, peeling based retrieval data structures suffer considerable performance degradation compared to a sequential setting, where a single thread can perform multiple memory accesses in parallel.
- **The 1+-bit variant of BuRR is smaller but slower** than the variant with 2-bit metadata per bucket, as expected, though not by a large margin.
- **Bloom filters and ribbon are fast for *negative queries*** where, on average, only two bits need to be retrieved to prove that a key is not in the set.

The Effect of Overloading. Figure 8 plots the fraction e of empty slots of BuRR for $w = 64$ and several combinations of bucket size b and different threshold compression schemes. Similar plots are given in the TR [26] for $w = 32$, $w = 128$, and for $w = 64$ with sparse coefficients. Note that (for an infinite number of layers), the space per element s is about $s = \frac{\frac{\mu}{b}+r}{1-e}$ where r is the number of retrieved bits and μ is the number of metadata bits per bucket.[34] Hence, at least when μ is constant, space consumption is a monotonic function in e and thus minimizing e also minimizes space.

We see that for small $|\varepsilon|$, e decreases exponentially. For sufficiently small b, e can get almost arbitrarily small. For fixed $b > w$, e eventually reaches a local minimum because with threshold-based compression, a large overload enforces large thresholds ($> w$) and thus empty regions of buckets. Which actual configuration to choose depends primarily on r. Roughly, for larger r, more and more metadata bits (i.e., small b, higher resolution of threshold values) can be invested to reduce e. For fixed b and threshold compression scheme, one can choose ε to minimize e. One can often choose a larger ε to get slightly better performance due to less bumping with little impact on o. Perhaps the most delicate tuning parameters are the thresholds to use for 2-bit and 1^+-bit compression (see Section 7.1).

9 Conclusion and Future Work

BuRR improves the state of the art of retrieval and static approximate membership data structures both in theory and in practice. *In theory*, among constructions with linear construction time and constant query time, we reduce the multiplicative space overhead by a $\log\log n$ factor to $O(\frac{\log\log n}{r\log n})$, assuming a small number of retrieved bits $r = O(\sqrt{\log n})$. More importantly perhaps, this overhead is *tunable* and can, for instance, be further reduced to $O(\frac{\log\log n}{r\log^2 n})$ at the price of moderately increased construction time of $O(n\log n)$ and query time $O(r)$. *In practice*, BuRR is faster than widely used data structures with much larger overhead and is reasonably simple to implement. Our results further strengthen the success of linear algebra based solutions to the problem. Our on-the-fly approach shows that Gauss-like solvers can be superior to peeling-based greedy solvers even with respect to speed.

While the wide design space of BuRR leaves room for further practical improvements, we see the main open problems for large r. In practice, peeling based solvers (e.g., [60]) might outperform BuRR if faster construction algorithms can be found—perhaps using ideas like overloading and bumping. In theory, existing succinct data structures (e.g., [6, 52]) allow constant query time but have high space overhead for realistic input sizes. Combining constant cost per element for large r with small (preferably tunable) space overhead therefore remains a theoretical promise yet to be convincingly redeemed in practice.

Acknowledgements

We thank engineers at Meta for their supporting roles, including Jay Zhuang, Siying Dong, Shrikanth Shankar, Affan Dar, and Ramkumar Vadivelu. We thank Seth Pettie for triggering the first steps of this research by an insightful question in a discussion at Dagstuhl Seminar 19051 “Data Structures for the Cloud and External Memory Data”.

References

[1] Yuriy Arbitman, Moni Naor, and Gil Segev. 2010. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Proceedings of the 51th FOCS*. 787–796. DOI: https://doi.org/10.1109/FOCS.2010.80

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

[34]Choosing a primary table of size m we have expected total space $\overbrace{\mu m/b}^{\text{metadata}} + \overbrace{rm}^{\text{main table}} + \overbrace{s\cdot(n-m(1-e))}^{\text{bumped data}}$. Thus, $sn = \mu m/b + rm + s\cdot(n-m(1-e))$. Solving for s yields $s = \frac{\frac{\mu}{b}+r}{1-e}$.

[3] Michael Axtmann, Daniel Ferizovic, Peter Sanders, and Sascha Witt. 2022. Engineering in-place (shared-memory) sorting algorithms. *ACM Trans. Parallel Comput.* 9, 1 (2022), 2:1–2:62.

[4] Matthias Becht, Hans-Peter Lehmann, and Peter Sanders. 2024. Brief Announcement: Parallel Construction of Bumped Ribbon Retrieval. *Computing Research Repository.* arXiv:2411.12365. Retrieved from https://arxiv.org/abs/2411.12365

[5] Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. 2014. Cache-oblivious peeling of random hypergraphs. In *Proceedings of the DCC.* 352–361. DOI: https://doi.org/10.1109/DCC.2014.48

[6] Djamal Belazzougui and Rossano Venturini. 2013. Compressed static functions with applications. In *Proceedings of the 24th SODA*, Sanjeev Khanna (Ed.). SIAM, 229–240. DOI: https://doi.org/10.1137/1.9781611973105.17

[7] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't thrash: How to cache your hash on flash. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1627–1637. DOI: https://doi.org/10.14778/2350229.2350275

[8] Valerio Bioglio, Marco Grangetto, Rossano Gaeta, and Matteo Sereno. 2009. On the fly gaussian elimination for LT codes. *Commun. Lett., IEEE* 13, 12 (12 2009), 953 – 955. DOI: https://doi.org/10.1109/LCOMM.2009.12.091824

[9] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426. DOI: https://doi.org/10.1145/362686.362692

[10] Fabiano Cupertino Botelho, Rasmus Pagh, and Nivio Ziviani. 2007. Simple and space-efficient minimal perfect hash functions. In *Proceedings of the 10th WADS.* 139–150. DOI: https://doi.org/10.1007/978-3-540-73951-7_13

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

[12] Alex D. Breslow and Nuwan Jayasena. 2020. Morton filters: Fast, compressed sparse cuckoo filters. *VLDB J.* 29, 2-3 (2020), 731–754. DOI: https://doi.org/10.1007/s00778-019-00561-0

[13] Andrei Z. Broder and Anna R. Karlin. 1990. Multilevel adaptive hashing. In *Proceedings of the 1st SODA*, David S. Johnson (Ed.). SIAM, 43–53. Retrieved from http://dl.acm.org/citation.cfm?id=320176.320181

[14] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the 15th SODA.* SIAM, 30–39. Retrieved from http://dl.acm.org/citation.cfm?id=982792.982797

[15] Robert B. Cooper. 1990. *Introduction to Queueing Theory* (3rd ed.). George Washington University.

[16] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. 1992. An optimal algorithm for generating minimal perfect hash functions. *Inf. Process. Lett.* 43, 5 (1992), 257–264. DOI: https://doi.org/10.1016/0020-0190(92)90220-P

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17).* Association for Computing Machinery, New York, NY, USA, 79–94. DOI: https://doi.org/10.1145/3035918.3064054

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

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

[20] Martin Dietzfelbinger and Rasmus Pagh. 2008. Succinct data structures for retrieval and approximate membership (extended abstract). In *Proceedings of the 35th ICALP (1).* 385–396. DOI: https://doi.org/10.1007/978-3-540-70575-8_32

[21] Martin Dietzfelbinger and Michael Rink. 2009. Applications of a splitting trick. In *Proceedings of the 36th ICALP (1).* 354–365. DOI: https://doi.org/10.1007/978-3-642-02927-1_30

[22] Martin Dietzfelbinger and Stefan Walzer. 2019. Constant-time retrieval with $O(\log m)$ extra bits. In *Proceedings of the 36th STACS.* 24:1–24:16. DOI: https://doi.org/10.4230/LIPIcs.STACS.2019.24

[23] Martin Dietzfelbinger and Stefan Walzer. 2019. Dense peelable random uniform hypergraphs. In *Proceedings of the 27th ESA.* 38:1–38:16. DOI: https://doi.org/10.4230/LIPIcs.ESA.2019.38

[24] Martin Dietzfelbinger and Stefan Walzer. 2019. Efficient gauss elimination for near-quadratic matrices with one short random block per row, with applications. In *Proceedings of the 27th ESA.* 39:1–39:18. DOI: https://doi.org/10.4230/LIPIcs.ESA.2019.39

[25] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. 2022. Fast succinct retrieval and approximate membership using ribbon. In *Proceedings of the 20th SEA (LIPIcs)*, Christian Schulz and Bora Uçar (Eds.), Vol. 233. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:20. DOI: https://doi.org/10.4230/LIPIcs.SEA.2022.4

[26] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. 2021. Fast succinct retrieval and approximate membership using ribbon. *Computing Research Repository.* arXiv:2109.01892. Retrieved from https://arxiv.org/abs/2106.12270

[27] Peter C. Dillinger and Stefan Walzer. 2021. Ribbon filter: Practically smaller than bloom and xor. *Computing Research Repository.* arXiv:2103.02515. Retrieved from https://arxiv.org/abs/2103.02515

[28] Regina Egorova, Bert Zwart, and Onno Boxma. 2006. Sojourn time tails in the M/D/1 processor sharing queue. *Probab. Eng. Inf. Sci.* 20, 3 (2006), 429–446. DOI: https://doi.org/10.1017/S0269964806060268

[29] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. Cuckoo filter: Better than bloom. *;login:* 38, 4 (2013), 5. Retrieved from https://www.usenix.org/publications/login/august-2013-volume-38-number-4/cuckoo-filter-better-bloom

[30] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. 2005. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.* 38, 2 (2005), 229–248. DOI: https://doi.org/10.1007/s00224-004-1195-x

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

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

[33] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. 2020. Fast scalable construction of ([compressed] static | minimal perfect hash) functions. *Inf. Comput.* 273 (2020), 104517. DOI: https://doi.org/10.1016/J.IC.2020.104517

[34] Thomas Mueller Graf and Daniel Lemire. 2019. fastfilter_cpp. (2019). Retrieved from https://github.com/FastFilter/fastfilter_cpp

[35] Thomas Mueller Graf and Daniel Lemire. 2020. Xor filters: Faster and smaller than bloom and cuckoo filters. *ACM J. Exp. Algorithmics* 25, 1, Article 1.5 (2020), 1–16. DOI: https://doi.org/10.1145/3376122

[36] Torben Hagerup and Torsten Tholey. 2001. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18st STACS*. 317–326. DOI: https://doi.org/10.1007/3-540-44693-1_28

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

[38] Yang Hu, William Kuszmaul, Jingxun Liang, Huacheng Yu, Junkai Zhang, and Renfei Zhou. 2025. Static Retrieval Revisited: To Optimality and Beyond. In *Proceedings of the 66th FOCS*.

[39] Svante Janson. 2005. Individual displacements for linear probing hashing with different insertion policies. *ACM Trans. Algorithms* 1, 2 (2005), 177–213. DOI: https://doi.org/10.1145/1103963.1103964

[40] David G. Kendall. 1953. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *Ann. Math. Statist.* 24, 3 (09 1953), 338–354. DOI: https://doi.org/10.1214/aoms/1177728975

[41] Adam Kirsch and Michael Mitzenmacher. 2008. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms* 33, 2 (2008), 187–218. DOI: https://doi.org/10.1002/rsa.20208

[42] D. E. Knuth. 1981. *The Art of Computer Programming — Seminumerical Algorithms* (2nd ed.). Vol. 2. Addison Wesley.

[43] William Kuszmaul and Stefan Walzer. 2024. Space lower bounds for dynamic filters and value-dynamic retrieval. In *Proceedings of the 56th STOC*. 1153–1164. DOI: https://doi.org/10.1145/3618260.3649649

[44] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. 2023. SicHash – small irregular cuckoo tables for perfect hashing. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2023*. SIAM, 176–189. DOI: https://doi.org/10.1137/1.9781611977561.CH15

[45] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. 2024. ShockHash: Towards optimal-space minimal perfect hashing beyond brute-force. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2024*. SIAM, 194–206. DOI: https://doi.org/10.1137/1.9781611977929.15

[46] Marc Lelarge. 2012. A new approach to the orientation of random hypergraphs. In *Proceedings of the 23rd SODA*. SIAM, 251–264. DOI: https://doi.org/10.1137/1.9781611973099.23

[47] Michael Luby, Michael Mitzenmacher, Mohammad Amin Shokrollahi, and Daniel A. Spielman. 2001. Efficient erasure correcting codes. *IEEE Trans. Inf. Theory* 47, 2 (2001), 569–584. DOI: https://doi.org/10.1109/18.910575

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

[49] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive string dictionary compression in in-memory column-store database systems. In *Proceedings of the 17th EDBT*. 283–294. DOI: https://doi.org/10.5441/002/edbt.2014.27

[50] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. 2014. Retrieval and perfect hashing using fingerprinting. In *Proceedings of the 14th SEA*. 138–149. DOI: https://doi.org/10.1007/978-3-319-07959-2_12

[51] W. W. Peterson. 1957. Addressing for random-access storage. *IBM J. Res. Dev.* 1, 2 (1957), 130–146. DOI: https://doi.org/10.1147/rd.12.0130

[52] Ely Porat. 2009. An optimal bloom filter replacement based on matrix solving. In *Proceedings of the 4th CSR*. 263–273. DOI: https://doi.org/10.1007/978-3-642-03351-3_25

[53] Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient bloom filters. *ACM J. Exp. Algorithmics* 14, Article 4.4 (2009), 18. DOI: https://doi.org/10.1145/1498698.1594230

[54] P. Sanders. 2009. Algorithm engineering – an attempt at a definition. In *Proceedings of the Efficient Algorithms*. LNCS, Vol. 5760. Springer, 321–340.

[55] Steven S. Seiden and Daniel S. Hirschberg. 1994. Finding succinct ordered minimal perfect hash functions. *Inf. Process. Lett.* 51, 6 (1994), 283–288. DOI: https://doi.org/10.1016/0020-0190(94)00108-1

[56] Hermann Thorisson. 1995. Coupling methods in probability theory. *Scand. J. Stat.* 22, 2 (1995), 159–182. Retrieved from http://www.jstor.org/stable/4616351

[57] Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. 2021. Partitioned learned bloom filter. In *International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=6BRLOfrMhW

[58] Sebastiano Vigna. 2025. ϵ-Cost sharding: Scaling hypergraph-based static functions and filters to trillions of keys. *Computing Research Repository*. arXiv:2503.18397. Retrieved from https://arxiv.org/abs/2503.18397

[59] Stefan Walzer. 2020. *Random Hypergraphs for Hashing-Based Data Structures*. Ph.D. Dissertation. Technische Universität Ilmenau. Retrieved from https://www.db-thueringen.de/receive/dbt_mods_00047127

[60] Stefan Walzer. 2021. Peeling close to the orientability threshold: Spatial coupling in hashing-based data structures. In *Proceedings of the 32nd SODA*. SIAM, 2194–2211. DOI: https://doi.org/10.1137/1.9781611976465.131

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

Received 29 July 2025; revised 29 July 2025; accepted 13 December 2025