Efficient Gauss Elimination for Near-Quadratic Matrices with One Short Random Block per Row, with Applications

Martin Dietzfelbinger
Technische Universität Ilmenau, Germany
martin.dietzfelbinger@tu-ilmenau.de

Stefan Walzer
Technische Universität Ilmenau, Germany
stefan.walzer@tu-ilmenau.de

Abstract

In this paper we identify a new class of sparse near-quadratic random Boolean matrices that have full row rank over $\mathbb{F}_2 = \{0, 1\}$ with high probability and can be transformed into echelon form in almost linear time by a simple version of Gauss elimination. The random matrix with dimensions $n(1-\varepsilon) \times n$ is generated as follows: In each row, identify a block of length $L = O((\log n)/\varepsilon)$ at a random position. The entries outside the block are 0, the entries inside the block are given by fair coin tosses. Sorting the rows according to the positions of the blocks transforms the matrix into a kind of band matrix, on which, as it turns out, Gauss elimination works very efficiently with high probability. For the proof, the effects of Gauss elimination are interpreted as a ("coin-flipping") variant of Robin Hood hashing, whose behaviour can be captured in terms of a simple Markov model from queuing theory. Bounds for expected construction time and high success probability follow from results in this area. They readily extend to larger finite fields in place of $\mathbb{F}_2$.

By employing hashing, this matrix family leads to a new implementation of a *retrieval* data structure, which represents an arbitrary function $f\colon S \to \{0,1\}$ for some set S of $m = (1-\varepsilon)n$ keys. It requires $m/(1-\varepsilon)$ bits of space, construction takes $\mathcal{O}(m/\varepsilon^2)$ expected time on a word RAM, while queries take $\mathcal{O}(1/\varepsilon)$ time and access only one contiguous segment of $\mathcal{O}((\log m)/\varepsilon)$ bits in the representation ($\mathcal{O}(1/\varepsilon)$ consecutive words on a word RAM). The method is readily implemented and highly practical, and it is competitive with state-of-the-art methods. In a more theoretical variant, which works only for unrealistically large S, we can even achieve construction time $\mathcal{O}(m/\varepsilon)$ and query time $\mathcal{O}(1)$, accessing $\mathcal{O}(1)$ contiguous memory words for a query. By well-established methods the retrieval data structure leads to efficient constructions of (static) perfect hash functions and (static) Bloom filters with almost optimal space and very local storage access patterns for queries.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Random Band Matrix, Gauss Elimination, Retrieval, Hashing, Succinct Data Structure, Randomised Data Structure, Robin Hood Hashing, Bloom Filter

Digital Object Identifier 10.4230/LIPIcs.ESA.2019.39

Acknowledgements We are very grateful to Seth Pettie, who triggered this research by asking an insightful question regarding "one block" while discussing the two-block solution from [17]. (This discussion took place at the Dagstuhl Seminar 19051 "Data Structures for the Cloud and External Memory Data".) Thanks are also due to the reviewers, whose comments helped to improve the presentation.

27th Annual European Symposium on Algorithms (ESA 2019).
Editors: Michael A. Bender, Ola Svensson, and Grzegorz Herman; Article No. 39; pp. 39:1–39:18

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

1.1 Sparse Random Matrices

In this paper we introduce and study a new class of sparse random matrices over finite fields, which give rise to linear systems that are efficiently solvable with high probability[1]. For concreteness and ease of notation, we describe the techniques for the field $\mathbb{F}_2 = \{0, 1\}$. (The analysis applies to larger fields as well, as will be discussed below.) A matrix A from this class has n columns and $m = (1 - \varepsilon)n$ rows for some small $\varepsilon > 0$. We always imagine that a right hand side $\vec{b} \in \{0, 1\}^m$ is given and that we wish to solve the system $A\vec{z} = \vec{b}$ for the vector of unknowns $\vec{z}$.

The applications (see Section 1.2) dictate that the rows of A are stochastically independent and are all chosen according to the same distribution $\mathcal{R}$ on $\{0, 1\}^n$. Often, but not always, $\mathcal{R}$ is the uniform distribution on some pool $R \subseteq \{0, 1\}^n$ of admissible rows. The following choices were considered in the literature.

(1) If $R = \{0, 1\}^n$, then A has full row rank whp for any $\varepsilon = \omega(1/n)$. In fact, the probability for full row rank is > 0.28 even for $\varepsilon = 0$, see e.g. [11, 35]. Solving time is $\tilde{\mathcal{O}}(n^3)$.

(2) A popular choice for R is the set of vectors with 1's in precisely k positions, for constant k. Then $\varepsilon = \mathrm{e}^{-\theta(k)}$ is sufficient for solvability whp [34]. Solving time is still $\tilde{\mathcal{O}}(n^3)$ if Gauss elimination is used and $\mathcal{O}(n^2)$ if Wiedemann's algorithm [37] is used, but heuristics exploiting the sparsity of A help considerably [22].

(3) In the previous setting with $k = 3$ and $\varepsilon \geq 0.19$, linear running time can be achieved with a simple greedy algorithm, since then the matrix can be brought into echelon form by row and column *exchanges* alone [7, 24, 32]. Using $k > 3$ is pointless here, as then the required value of ε increases.

(4) Luby et al. [29, 30] study "loss-resilient codes" based on certain random bipartite graphs. Translating their considerations into our terminology shows that at the core of their construction is a distribution on $(1 - \varepsilon)n \times n$-matrices with randomly chosen sparse rows as well. Simplifying a bit, a number $D = \mathcal{O}(1/\varepsilon)$ is chosen and a weight sequence is carefully selected that will give a row at most D many 1's and on average $O(\log D)$ many 1's (in random positions). It is shown in [29, 30] that such matrices not only have full row rank with high probability, but that, as in (3), row and column *exchanges* suffice to obtain an echelon form whp. This leads to a solving time of $\mathcal{O}(n \log(1/\varepsilon))$ for the corresponding linear system.

(5) The authors of the present work describe in a simultaneous paper [18] the construction of sparse $(1-\varepsilon)n \times n$ matrices for very small (constant) ε, with a fixed number of 1's per row, which also allow solving the corresponding system by row and column exchanges. (While behaviour in experiments is promising, determining the behaviour of the construction for arbitrarily small ε is an open problem.)

(6) In a recent proposal [17] by the authors of the present paper, a row $r \sim \mathcal{R}$ contains two *blocks* of $\Theta(\log n)$ random bits at random positions (block-aligned) in a vector otherwise filled with 0's. It turned out that in this case even $\varepsilon = \mathcal{O}((\log n)/n)$ will give solvability with high probability. Solution time is again about cubic (Gauss) resp. quadratic (Wiedemann), with heuristic pre-solvers cushioning the blow partly in practice.

Motivated by the last construction, we propose an even simpler choice for the distribution $\mathcal{R}$: A row r consists of 0's except for one randomly placed block of some length L, which consists of random bits. It turns out that $L = \mathcal{O}((\log n)/\varepsilon)$ is sufficient to achieve solvability

[1] Events occur "with high probability (whp)" if they occur with probability $1 - \mathcal{O}(m^{-1})$.

with high probability. The L-bit block fits into $\mathcal{O}(1)$ memory words as long as ε is constant. Our main technical result (Theorem 2) is that the resulting random matrix has full row rank whp. Moreover, if this is the case then sorting the rows by starting points of the blocks followed by a simple version of Gauss elimination produces an echelon form of the matrix and a solution to the linear system. The expected number of field operations is $\mathcal{O}(nL/\varepsilon)$, which translates into expected running time $\mathcal{O}(n/\varepsilon^2)$ on a word RAM. For the proof, we establish a connection to a particular version of Robin Hood hashing, whose behaviour in turn can be understood by reducing it to a well-known situation in queuing theory. (A detailed sketch of the argument is provided in Section 2.1.)

To our knowledge, this class of random matrices has not been considered before. However, *deterministic* versions of matrices similar to these random ones have been thoroughly studied in the last century, for both infinite and finite fields. Namely, sorting the rows of our matrices yields matrices that with high probability resemble *band matrices*, where the nonzero entries in row i are within a restricted range around column $\lfloor i/(1-\varepsilon)\rfloor$. In the study of band matrices one usually has $\varepsilon = 0$ and assumes that the matrix is nonsingular. Seemingly the best known general upper time bound for the number of field operations needed for solving band quadratic systems with bandwidth L are $\mathcal{O}(nL^{\omega-1}) = \mathcal{O}(n((\log n)/\varepsilon)^{\omega-1})$, where ω is the matrix multiplication exponent, see [20, 23, 33].

1.2 Retrieval

One motivation for studying random systems as described above comes from data structures for solving the *retrieval problem*, which can be described as follows: Some "*universe*" $\mathcal{U}$ of possible keys is given, as is a function $f\colon S \to W$, where $S \subseteq \mathcal{U}$ has finite size m and $W = \{0,1\}^r$ for some $r \geq 1$. A *retrieval data structure* [6, 10, 15, 35] makes it possible to recover $f(x)$ quickly for arbitrary given $x \in S$. We do not care what the result is when $x \notin S$, which makes the retrieval situation different from a dictionary, where the question "$x \in S$?" must also be decided. A retrieval data structure consists of

- an algorithm $\mathsf{construct}$, which takes f as a list of pairs (and maybe some parameters) as input and constructs an object DS_f, and
- an algorithm query, which on input $x \in \mathcal{U}$ and DS_f outputs an element of W, with the requirement that $\mathsf{query}(\mathsf{DS}_f, x) = f(x)$ for all $x \in S$.

The essential performance parameters of a retrieval data structure are:

- the space taken up by DS_f (ideally $(1+\varepsilon)m$ bits of memory for some small $\varepsilon > 0$),
- the running time of $\mathsf{construct}$ (ideally $\mathcal{O}(m)$), and
- the running time of query (ideally a small constant in the worst case, possibly dependent on ε, and good cache behaviour).

In this paper we concentrate on the case most relevant in practice, namely the case of small constant r, in particular on[2] $r = 1$.

A standard approach is as follows [6, 10, 15, 35]. Let $f\colon S \to \{0,1\}$ be given and let $n = m/(1-\varepsilon)$ for some $\varepsilon > 0$. Use hashing to construct a mapping $\mathsf{row}\colon \mathcal{U} \to \{0,1\}^n$ such that $(\mathsf{row}(x))_{x\in S}$ is (or behaves like) a family of independent random variables drawn from a suitable distribution $\mathcal{R}$ on $\{0,1\}^n$. Consider the linear system $(\langle \mathsf{row}(x), \vec{z}\rangle = f(x))_{x\in S}$. In case the vectors $\mathsf{row}(x)$, $x \in S$, are linearly independent, this system is solvable for $\vec{z}$.

[2] Every solution for this case gives a solution for larger r as well, with a slowdown not larger than r. In our case, this slowdown essentially only affects queries, not construction, since the Gauss elimination based algorithm can trivially be extended to simultaneously handle r right hand sides $\vec{b}_1, \ldots, \vec{b}_r$ and produce r solution vectors $\vec{z}_1, \ldots, \vec{z}_r$. This change slows down $\mathsf{construct}$ by a factor of $1 + r/L = 1 + \mathcal{O}(\varepsilon r/\log n)$.

Solve the system and store the bit vector $\vec{z}$ of n bits (and the hash function used) as DS_f. Evaluation is by $\mathsf{query}(\mathsf{DS}_f, x) = \langle \mathsf{row}(x), \vec{z} \rangle$, for $x \in \mathcal{U}$. The running time of construct is essentially the time for solving the linear system, and the running time for query is the time for evaluating the inner product.

A common and well-explored trick for reducing the construction time [5, 16, 35, 22] is to split the key set into "chunks" of size $\Theta(C)$ for some suitable C and constructing separate retrieval structures for the chunks. The price for this is twofold: In queries, one more hash function must be evaluated and the proper part of the data structure has to be located; regarding storage space one needs an array of $\Omega(m/C)$ pointers. In this paper, we first concentrate on a "pure" construction. The theoretical improvements possible by applying the splitting technique will be discussed briefly in Section 4. The splitting technique is also used in experiments for our construction in Section 5 to keep the block length small. In this context it will also be noted the related "split-and-share" technique from [16, 19] can be used to get rid of the assumption that fully random hash functions are available for free.

Our main result regarding the retrieval problem follows effortlessly from the analysis of the new random linear systems (formally stated as Theorem 2).

▶ **Theorem 1.** *Let $\mathcal{U}$ be a universe. Assume the context of a word RAM with oracle access to fully random hash functions on $\mathcal{U}$. Then for any $\varepsilon > 0$ there is a retrieval data structure such that for all $S \subseteq \mathcal{U}$ of size m*

- **(i)** construct *succeeds with high probability.*
- **(ii)** construct *has expected running time $\mathcal{O}(\frac{m}{\varepsilon^2})$.*
- **(iii)** *The resulting data structure DS_f occupies at most $(1+\varepsilon)m$ bits.*
- **(iv)** query *has running time $\mathcal{O}(\frac{1}{\varepsilon})$ and accesses $\mathcal{O}(\frac{1}{\varepsilon})$ consecutive words in memory.*

1.3 Machine Model and Notation

For a positive integer k we denote $\{1, \ldots, k\}$ by $[k]$. The number m always denotes the size of a domain – the number of keys to hash, the size of a function for retrieval or the number of rows of a matrix. A (small) real number $\varepsilon > 0$ is also given. The number n denotes the size of a range. We usually have $m = (1-\varepsilon)n$. In asymptotic considerations we always assume that ε is constant and m and n tend to ∞, so that for example the expression $\mathcal{O}(n/\varepsilon)$ denotes a function that is bounded by cm/ε for a constant c, for all m bigger than some $m(\varepsilon)$. By $\langle \vec{y}, \vec{z} \rangle$ we denote the inner product of two vectors $\vec{y}$ and $\vec{z}$. As our computational model we adopt the word RAM with memory words comprising $\Omega(\log m)$ bits, in which an operation on a word takes constant time. In addition to AC_0 instructions we will need the PARITY of a word as an elementary operation. For simplicity we assume this can be carried out in constant time, which certainly is realistic for standard word lengths like 64 or 128. In any case, as the word lengths used are never larger than $\mathcal{O}(\log m)$, one could tabulate the values of PARITY for inputs of size $\frac{1}{2}\log m$ in a table of size $\mathcal{O}(\sqrt{m}\log m)$ bits to achieve constant evaluation time for inputs comprising a constant number of words.

1.4 Techniques Used

We use *coupling* of random variables X and Y (or of processes $(X_i)_{i\geq 1}$ and $(Y_i)_{i\geq 1}$). By this we mean that we exhibit a single probability space on which X and Y (or $(X_i)_{i\geq 1}$ and $(Y_i)_{i\geq 1}$) are defined, so that there are interesting *pointwise* relations between them, like $X \leq Y$, or $X_i \leq Y_i + a$ for all $i \geq 1$, for a constant a. Sometimes these relations hold only conditioned on some (large) part of the probability space. We will make use of the following

observation. If we have random variables $U_0, \ldots, U_k$ with couplings, i.e. joint distributions, of $U_{\ell-1}$ and U_ℓ, for $1 \le \ell \le k$, then there is a common probability space on which all these random variables are defined and the joint distribution of $U_{\ell-1}$ and U_ℓ is as given.[3]

2 Random Band Systems that Can be Solved Quickly

The main topic of this paper are matrices generated by the following random process. Let $0 < \varepsilon < 1$ and $n \in \mathbb{N}$. For a number $m = (1-\varepsilon)n$ of rows and some number $L \ge 1$ we consider a matrix $A = (a_{ij})_{i\in[m],\, j\in[n+L-1]}$ over the field $\mathbb{F}_2$, chosen at random as follows. For each row $i \in [m]$ a *starting position* $s_i \in [n] = \{1, \ldots, n\}$ is chosen uniformly at random. The entries a_{ij}, $s_i \le j < s_i + L$ form a *block* of fully random bits, all other entries in row i are 0.

In this section we show that for proper choices of the parameters such a random matrix will have full row rank and the corresponding systems $A\vec{z} = \vec{b}$ will be solvable very efficiently whp. Before delving into the technical details, we sketch the main ideas of the proof.

2.1 Proof Sketch

As a starting point, we formulate a simple algorithm, a special version of Gaussian elimination, for solving linear systems $A\vec{z} = \vec{b}$ as just described. We first sort the rows of A by the starting position of their block. The resulting matrix resembles a band matrix, and we apply standard Gaussian elimination to it, treating the rows in order of their starting position. Conveniently, there is no "proliferation of 1's", i.e. we never produce a 1-entry outside of any row's original block. In the round for row i, the entries a_{ij} for $j = s_i, \ldots, s_i + L - 1$ are scanned. If column j has been previously chosen as pivot then $a_{ij} = 0$. Otherwise, a_{ij} is a random bit. While this bit may depend in a complex way on the original entries of rows $1, \ldots, i$ (apart from position (i, j)), for the analysis we may simply imagine that a_{ij} is only chosen now by flipping a fair coin. This means that we consider eligible columns from left to right, and the first j for which the coin flip turns up 1 becomes the pivot column for row i. This view makes it possible to regard choosing pivot columns for the rows as probabilistically equivalent to a slightly twisted version of Robin Hood hashing. Here this means that m keys $x_1, \ldots, x_m$ with random hash values in $\{1, \ldots, n + L - 1\}$ are given and, in order of increasing hash values, are inserted in a linear probing fashion into a table with positions $1, \ldots, n + L - 1$ (meaning that for x_i cells $s_i, s_{i+1}, \ldots$ are inspected). The twist is that whenever a key probes an empty table cell flipping a fair coin decides whether it is placed in the cell or has to move on to the next one. The resulting position of key x_i is the same as the position of the pivot for row i. As is standard in the precise analysis of linear probing hashing we switch perspective and look at the process from the point of view of cells $1, 2, \ldots, n, \ldots, n + L - 1$. Associated with position ("time") j is the set of keys that probe cell j (the "queue"), and the quantity to study is the length of this queue. It turns out that the average queue length determines the overall cost of the row additions, and that the probability for the maximum queue length to become too large is decisive for bounding the success probability of the Gaussian elimination process. The first and routine step in

[3] We do not prove this formally, since arguments like this belong to basic probability theory or measure theory. The principle used is that the pairwise couplings give rise to conditional expectations $\mathbb{E}(U_\ell \mid U_{\ell-1})$. Arguing inductively, given a common probability space for $U_1, \ldots, U_{\ell-1}$ and $\mathbb{E}(U_\ell \mid U_{\ell-1})$, one can obtain a common probability space for $U_1, \ldots, U_\ell$ so that $(U_1, \ldots, U_{\ell-1})$ is distributed as before and $\mathbb{E}(U_\ell \mid U_1, \ldots, U_{\ell-1}) = \mathbb{E}(U_\ell \mid U_{\ell-1})$. – This is practically the same as the standard argument that shows that a sequence of conditional expectations gives rise to a corresponding Markov chain on a joint probability space.

the analysis of the queue length is to "Poissonise" arrivals such that the evolution of the queue length becomes a Markov chain. A second step is needed to deal with the somewhat annoying possibility that in a cell all keys that are eligible for this cell reject it because of their coin flips. We end up with a standard queue (an "M/D/1 queue" in Kendall notation) and can use existing results from queuing theory to read off the bounds regarding the queue length needed to complete the analysis.

The following subsections give the details.

2.2 A Simple Gaussian Solver

We now describe the algorithm to solve linear systems involving the random matrices described above. This is done by a variant or Gauss elimination, which will bring the matrix into echelon form (up to leaving out inessential column exchanges) and then apply back substitution.

Given a random matrix $A = (a_{ij})_{i\in[m],\, j\in[n+L-1]}$ as defined above, with blocks of length L starting at positions s_i, for $i \in [m]$, as well as some $\vec{b} \in \{0,1\}^m$, we wish to find a solution $\vec{z}$ to the system $A\vec{z} = \vec{b}$. Consider algorithm SGAUSS (Algorithm 1). If A has linearly independent rows, it will return a solution $\vec{z}$ and produce intermediate values $(\mathrm{piv}_i)_{i\in[m]}$. (These will be important only in the analysis of the algorithm.) If the rows of A are linearly dependent, the algorithm will fail.

Algorithm 1 A simple Gaussian solver.

Algorithm SGAUSS$(A = (a_{ij})_{i\in[m],\, j\in[n+L-1]}, (s_i)_{i\in[m]}, \vec{b} \in \{0,1\}^m)$**:**
  sort the rows of the system $(A, \vec{b})$ by s_i (in time $\mathcal{O}(m)$)
  relabel such that $s_1 \le s_2 \le \cdots \le s_m$
  $\mathrm{piv}_1, \mathrm{piv}_2, \ldots, \mathrm{piv}_m \leftarrow 0$
  for $i = 1, \ldots, m$ **do**
    for $j = s_i, \ldots, s_i + L - 1$ **do** } // search for leftmost 1 in row i. Can be done
      if $a_{ij} = 1$ **then** } // in time $\mathcal{O}(L/\log m)$ on a word RAM.
        $\mathrm{piv}_i \leftarrow j$
        for i' **with** $i' > i \wedge s_{i'} \le \mathrm{piv}_i$ **do**
          if $a_{i',\mathrm{piv}_i} = 1$ **then**
            $a_{i'} \leftarrow a_{i'} \oplus a_i$ // row addition (= subtraction)
            $b_{i'} \leftarrow b_{i'} \oplus b_i$
        break
    if $\mathrm{piv}_i = 0$ // row i is 0
     then return FAILURE
  // back substitution:
  $\vec{z} \leftarrow \vec{0}$
  for $i = m, \ldots, 1$ **do**
    $z_{\mathrm{piv}_i} \leftarrow \langle \vec{z}, a_i \rangle \oplus b_i$ // note: $a_{ij} = 0$ for j outside of $\{s_i, \ldots, s_i + L - 1\}$
  return $\vec{z}$ // solution to $A\vec{z} = \vec{b}$

Algorithm SGAUSS starts by sorting the rows of the system $(A, \vec{b})$ by their starting positions s_i in linear time, e.g. using counting sort [13, Chapter 8.2]. We suppress the resulting permutation in the notation, assuming $s_1 \le s_2 \le \cdots \le s_m$. Rows are then processed sequentially. When row i is treated, its leftmost 1-entry is found, if possible, and

the corresponding column index is called the *pivot* piv_i of row i. Row additions are used to eliminate 1-entries from column piv_i in subsequent rows. Note that this operation never produces nonzero entries outside of any row's original block, i.e. for no row i are there ever any 1's outside of the positions $\{s_i, \ldots, s_i + L - 1\}$. To see this, we argue inductively on the number of additions performed. Assume $i > 1$ and row i' with $i' < i$ is added to row i. By choice of $\mathrm{piv}_{i'}$ and the induction hypothesis, nonzero entries of row i' can reside only in positions $\mathrm{piv}_{i'}, \ldots, s_{i'} + L - 1$. Again by induction and since row i contains a 1 in position $\mathrm{piv}_{i'}$, we have $s_i \leq \mathrm{piv}_{i'}$; moreover we have $s_{i'} + L - 1 \leq s_i + L - 1$, due to sorting. Thus, row i' contains no 1's outside of the block of row i and the row addition maintains the invariant.

If an all-zero row is encountered, the algorithm fails (and returns FAILURE). This happens if and only if the rows of A are linearly dependent[4]. Otherwise we say that the algorithm succeeds. In this case a solution $\vec{z}$ to $A\vec{z} = \vec{b}$ is obtained by back-substitution.

It is not hard to see that the expected running time of SGAUSS is dominated by the expected cost of row additions.

The proof of the following statement, presented in the rest of this section, is the main technical contribution of this paper.

▶ **Theorem 2.** *There is some $L = \mathcal{O}((\log m)/\varepsilon)$ such that a run of* SGAUSS *on the random matrix $A = (a_{ij})_{i\in[m],\, j\in[n+L-1]}$ and an arbitrary right hand side $\vec{b} \in \{0,1\}^m$ succeeds whp. The expected number of row additions is $\mathcal{O}(m/\varepsilon)$. Each row addition involves entries inside one block and takes time $\mathcal{O}(1/\varepsilon)$ on a word RAM.*

2.3 Coin-Flipping Robin Hood Hashing

Let $\{x_1, \ldots, x_m\} \subseteq \mathcal{U}$ be some set of *keys* to be stored in a hash table T. Each key x_i has a uniformly random *hash value* $h_i \in [n]$. An (injective) placement of the keys in T fulfils the *linear probing* requirement if each x_i is stored in a cell $T[\mathrm{pos}_i]$ with $\mathrm{pos}_i \geq h_i$ and all cells $T[j]$ for $h_i \leq j < \mathrm{pos}_i$ are non-empty. In *Robin Hood hashing* there is the additional requirement that $h_i > h_{i'}$ implies $\mathrm{pos}_i > \mathrm{pos}_{i'}$. Robin Hood hashing is interesting because it minimises the variance of the displacements $\mathrm{pos}_i - h_i$. It has been studied in detail in several papers [9, 14, 25, 27, 36].

Given the hash values $(h_i)_{i\in[m]}$, a placement of the keys obeying the Robin Hood linear probing conditions can be obtained as follows: Insert the keys in the order of increasing hash values, by the usual linear probing insertion procedure, which probes (i.e. inspects) cells $T[h_i], T[h_i + 1], \ldots$ until the first empty cell is found, and places x_i in this cell. We consider a slightly "broken" variation of this method, which sometimes delays placements. In the placing procedure for x_i, when an empty cell $T[j]$ is encountered, it is decided by flipping a fair coin whether to place x_i in cell $T[j]$ or move on to the next cell. (No problem is caused by the fact that the resulting placement violates the Robin Hood requirement and even the linear probing requirement, since the hash table is only used as a tool in our analysis.) For this insertion method we assume we have an (idealised) unbounded array $T[1, 2, \ldots]$. The position in which key x_i is placed is called pos_i. At the end the algorithm itself checks whether any of the displacements $\mathrm{pos}_i - h_i$ is larger than L, in which case it reports FAILURE.[5] Algorithm 2 gives a precise description of this algorithm, which we term CFRH.

[4] Depending on $\vec{b}$, the system $A\vec{z} = \vec{b}$ may still be solvable. We will not pursue this.

[5] The reason we postpone checking for FAILURE until the very end of the execution is that it is technically convenient to have the values $(\mathrm{pos}_i)_{i\in[m]}$ even if failure occurs.

Algorithm 2 The *Coin-Flipping Robin Hood hashing* algorithm. Without the condition "coinFlip() = 1" it would compute a Robin Hood placement with maximum displacement L, if one exists.

```
1  Algorithm CFRH ({x_1, ..., x_m} ⊆ U):
2      sort x_1, ..., x_m by hash value h_1, ..., h_m
3      relabel such that h_1 ≤ ··· ≤ h_m
4      T ← [⊥, ⊥, ...] // empty array, "⊥" means "undefined"
5      pos_1, ..., pos_m ← 0
6      for i = 1, ..., m do
7          for j = h_i, h_i + 1, ... do
8              if T[j] = ⊥ ∧ coinFlip() = 1 ("HEADS") then
9                  pos_i ← j
10                 T[j] ← x_i
11                 break
12     if ∃i ∈ [m] : pos_i − h_i ≥ L then return FAILURE
13     return T
```

2.4 Connection between SGAUSS and CFRH

We now establish a close connection between the behaviour of algorithms SGAUSS and CFRH, thus reducing the analysis of SGAUSS to that of CFRH. The algorithms have been formulated in such a way that some structural similarity is immediate. A run of SGAUSS on a matrix with random starting positions $(s_i)_{i\in[m]}$ and random entries yields a sequence of pivots $(\mathrm{piv}_i)_{i\in[m]}$; a run of CFRH on a key set with random hash values $(h_i)_{i\in[m]}$ performing random coin flips yields a sequence of positions $(\mathrm{pos}_i)_{i\in[m]}$. We will see that the distributions of $(\mathrm{piv}_i)_{i\in[m]}$ and $(\mathrm{pos}_i)_{i\in[m]}$ are essentially the same and that moreover two not so obvious parameters of the two random processes are closely connected. For this, we will show that outside the FAILURE events we can use the probability space underlying algorithm SGAUSS to describe the behaviour of algorithm CFRH. This yields a coupling of the involved random processes.

The first step is to identify $s_i = h_i$ for $i \in [m]$ (both sequences are assumed to be sorted and then renamed). The connection between pos_i and piv_i is achieved by connecting the coin flips of CFRH to certain events in applying SGAUSS to matrix A. We construct this correspondence by induction on i. Assume rows $1, \dots, i-1$ have been treated, $x_1, \dots, x_{i-1}$ have been placed, and $\mathrm{piv}_{i'} = \mathrm{pos}_{i'}$ for all $1 \le i' < i$.

Now row a_i (transformed by previous row additions) is treated. It contains a 0 in columns that were previously chosen as pivots, so possible candidates for piv_i are only indices from $J_i := \{s_i, \dots, s_i + L - 1\} \setminus \{\mathrm{piv}_1, \dots, \mathrm{piv}_{i-1}\}$. For each $j \in J_i$, the initial value of a_{ij} was a random bit. The bits added to a_{ij} in rounds $1, \dots, i-1$ are determined by the original entries of rows $1, \dots, i-1$ alone. We order the entries of J_i as $j^{(1)} < j^{(2)} < \cdots < j^{(|J_i|)}$. Then, conditioned on all random choices in rows $1, \dots, i-1$ of A, the current values $a_{i,j^{(1)}}, \dots, a_{i,j^{(k)}}$ still form a sequence of fully random bits. We use these random bits to run round i of CFRH, in which x_i is placed. Since each cell can only hold one key, and by excluding runs where finally FAILURE is declared, we may focus on the empty cells with indices in $\{h_i, \dots, h_i + L - 1\} \setminus \{\mathrm{pos}_1, \dots, \mathrm{pos}_{i-1}\} = \{s_1, \dots, s_i + L - 1\} \setminus \{\mathrm{piv}_1, \dots, \mathrm{piv}_{i-1}\} = J_i$. We use (the current value) a_{ij} as the value of the coin flip for cell j, for $j = j^{(1)}, j^{(2)}, \dots, j^{(|J_i|)}$. The minimal j in this sequence (if any) with $a_{ij} = 1$ equals piv_i and pos_i. If all these bits are 0, algorithm SGAUSS will fail immediately, and key x_i will be placed in a cell $T[j]$ with $j \ge h_i + L$, so CFRH will eventually fail as well.

Thus we have established that the random variables needed to run algorithm CFRH (outside of FAILURE) can be taken to belong to the probability space defined by $(s_i)_{i\in[m]}$ and the entries in the blocks of A for algorithm SGAUSS, so that (outside of FAILURE) the random variables pos_i and piv_i are the same. In the following lemma we state this connection as Claim (i). In addition, we consider other random variables central for the analysis to follow. First, we define the *height* of position $j \in [n + L - 1]$ in the hash table as

$$H_j := \#\{i \in [m] \mid h_i \le j < \mathrm{pos}_i\}.$$

This is the number of keys probing table cell j without being placed in it, either because the cell is occupied or because it is rejected by the coin flip. Claim (ii) in the next lemma shows that $\sum_{j\in[n+L-1]} H_j$ essentially determines the running time of SGAUSS, so that we can focus on bounding $(H_j)_{j\in\mathbb{N}}$ from here on. Further, with Claim (iii), we get a handle on the question how large we have to choose L in order to keep the failure probability small.

▶ **Lemma 3.** *With the coupling just described, we get*

(i) SGAUSS *succeeds iff* CFRH *succeeds. On success we have* $\mathrm{piv}_i = \mathrm{pos}_i$ *for all* $i \in [m]$.

(ii) *A successful run of* SGAUSS *performs at most* $\sum_{j\in[n+L-1]} H_j$ *row additions.*

(iii) *Conditioned on the event* $\max_{j\in[n]} H_j \le L - 2\log m$*, the algorithms succeed whp.*

Proof. (ii) (Note that a similar statement with a different proof can be found in [26, Lemma 2.1].) Consider the sets $\mathsf{Add} := \{(i, i') \in [m]^2 \mid \text{SGAUSS adds row } i \text{ to row } i'\}$ and $\mathsf{Displ} := \{(i, j) \in [m] \times [n + L - 1] \mid h_i \le j < \mathrm{pos}_i\}$. Since H_j simply counts the pairs $(i, j) \in \mathsf{Displ}$ with $i \in [m]$, we have $|\mathsf{Displ}| = \sum_{j\in[n+L-1]} H_j$. To prove the claim we exhibit an injection from Add into Displ.

Assume $(i, i') \in \mathsf{Add}$. If $\mathrm{pos}_i < \mathrm{pos}_{i'}$, we map (i, i') to (i', pos_i). This is indeed an element of Displ, since $h_{i'} \le \mathrm{piv}_i = \mathrm{pos}_i < \mathrm{pos}_{i'}$ (if piv_i were smaller than $s_{i'}$, row i would not be added to row i'). On the other hand, if $\mathrm{pos}_i > \mathrm{pos}_{i'}$, we map (i, i') to $(i, \mathrm{pos}_{i'})$. This is in Displ since $h_i = s_i \le s'_i \le \mathrm{pos}_{i'} < \mathrm{pos}_i$ (recall that rows are sorted by starting position).

The mapping is injective since from the image of $(i, i') \in \mathsf{Add}$ we can recover the set $\{i, i'\}$ with the help of the injective mapping $i \mapsto \mathrm{pos}_i$, $i \in [m]$. The fact that $i < i'$ fixes the ordering in the pair.

(iii) In CFRH, for an arbitrary $i \in [m]$ consider the state before key x_i probes its first position $j := h_i$. Any previous key $x_{i'}$ with $i' < i$ has a hash value $h_{i'} \le h_i$. Hence it either was inserted in a cell $j' < j$ or it has probed cell j. Since at most H_j keys have probed cell j, at most H_j positions in $T[j, \dots, j + L - 1]$ are occupied and at least $2\log m$ are free. The probability that x_i is not placed in this region is therefore at most $2^{-2\log m} = m^{-2}$. By the union bound we obtain a failure probability of $\mathcal{O}(1/m)$. ◀

2.5 Bounding Heights in CFRH by a Markov Chain

Lemma 3 tells us that we must analyse the heights in the hashing process CFRH. In this subsection, we use "Poissonisation" of the hashing positions to majorise the heights in CFRH by a Markov chain, i.e. a process that is oblivous to the past, apart from the current height. Poissonisation is a common step in the analysis of linear probing hashing, see e.g. [36]. Further, we wish to replace randomized placement by deterministic placement: Whenever a key is available for a position, one is put there (instead of flipping coins for all available keys). By this, the heights may decrease, but only by a bounded amount whp. The details of these steps are given in this subsection.

In analysing CFRH (without regard for the event FAILURE), it is inconvenient that the starting positions h_i are determined by random choices with subsequent sorting. Position j is hit by a number of keys given by a binomial distribution $\mathrm{Bin}(m, \frac{1}{n})$ with expectation $\frac{m}{n} = 1-\varepsilon$, but there are dependencies. We approximate this situation by "Poissonisation" [31, Sect. 5.4]. Here this means that we assume that cell $j \in [n]$ is hit by k_j keys, independently for $j = 1, \dots, m$, where $k_j \sim \mathrm{Po}(1-\varepsilon')$ is Poisson distributed, for $\varepsilon' = \varepsilon/2$. Then the total number $m' = \sum_{j\in[n]} k_j$ of keys is distributed as $m' \sim \mathrm{Po}((1-\varepsilon')n)$. Given $k_1, \dots, k_n$, we can imagine we have m' keys with nondecreasing hash values $(h_i)_{i\in[m']}$, and we can apply algorithm CFRH to obtain key positions $(\mathrm{pos}'_i)_{i\in[m']}$ in $\{1, 2, \dots\}$ and cell heights $(H'_j)_{j\geq 1}$.

Conveniently, with Poissonisation, the heights $(H'_j)_{j\in[n]}$ turn out to form a Markov chain. This can be seen as follows. Recall that H'_{j-1} is the number of keys probing cell $j-1$ without being placed there. Hence the number of keys probing cell j is $H'_{j-1} + k_j$. One of these keys will be placed in cell j, unless $H'_{j-1} + k_j$ coin flips all yield 0, so if $g_j \sim \mathrm{Geom}(\frac{1}{2})$ is a random variable with geometric distribution with parameter $\frac{1}{2}$ (number of fair coin flips needed until the first 1 appears) and b_j is the indicator function $\mathbb{1}_{\{g_j > H'_{j-1}+k_j\}}$, we have $H'_j = H'_{j-1} + k_j - 1 + b_j$. (Note that the case $H'_{j-1} + k_j = 0$ is treated correctly by this description. Conditioned on $H'_{j-1} + k_j$, the value b_j is a Bernoulli variable.) The Markov property holds since H'_j depends only on H'_{j-1} and the two "fresh" random variables k_j and g_j.

The following lemma allows us to shift our attention from (H_j) to (H'_j).

▶ **Lemma 4.** *Let $m = (1-\varepsilon)n$ and $m' \sim \mathrm{Po}((1-\varepsilon')m)$ for $\varepsilon' = \varepsilon/2$. There is a coupling between an ordinary run of CFRH (with m, n and H_j) and a Poissonised run (with m', n and H'_j) such that conditioned on the high probability event $E_{\geq m} = \{m' \geq m\}$ we have $H'_j \geq H_j$ for all $j \in [n + L - 1]$.*

Proof. Because ε and $\varepsilon' = \varepsilon/2$ are constants, the event $E_{\geq m}$ has indeed high probability, as can be seen by well-known concentration bounds for the Poisson distribution (e.g. [31, Th. 5.4]). For $m_0 \geq m$ fixed the distribution of the number of hits in the cells in $T[1, \dots, n]$ conditioned on $\{m' = m_0\}$ is the same as what we get by throwing m_0 balls randomly into n bins [31, Th. 5.6]. Thus, we may assume the Poissonised run has to deal with the m keys of the ordinary run plus $m' - m$ additional keys with random hash values in $[n]$. We apply algorithm CFRH to both inputs. After sorting, the new keys are inserted in some interleaved way with the ordinary keys. Now if one of the ordinary keys x probes an empty cell $T[j]$, we use the same coin flip in both runs to decide whether to place it there; for the probing of the additional keys we use new, independent coin flips. With this coupling it is clear that for all ordinary keys x the displacement "(position of x) − (hash value of x)" in the Poissonized run is at least as big as in the ordinary run. As the additional keys can only increase heights, $H'_j \geq H_j$ follows. ◀

As a further simplification, we eliminate the geometrically distributed variable g_j and the derived variable b_j in the Markov chain $(H'_j)_{j\geq 0}$. For this, let $(X_j)_{j\geq 0}$ be the Markov chain defined as

$$X_0 := 0 \quad \text{and} \quad X_j := \max(0, X_{j-1} + d_j - 1) \quad \text{for } j \geq 1, \tag{1}$$

where $d_j \sim \mathrm{Po}(1 - \varepsilon'/2)$ are independent random variables.

▶ **Lemma 5.** *There is a coupling between $(X_j)_{j\geq 0}$ and $(H'_j)_{j\geq 0}$ such that $X_j + \log(4/\varepsilon') \geq H'_j$ for all $j \in [n + L - 1]$.*

Proof. Assume wlog that $\log(1/\varepsilon')$ is an integer. Let $b'_j \sim \mathrm{Po}(\varepsilon'/2)$ be a random variable on the same probability space as g_j such that $g_j > \log(4/\varepsilon')$ implies $b'_j \geq 1$. This is possible because

$$\Pr[g_j > \log(4/\varepsilon')] = 2^{-\log(4/\varepsilon')} = \varepsilon'/4 \leq 1 - e^{-\varepsilon'/2} = \Pr[b'_j \geq 1].$$

We then define $d_j \coloneqq k_j + b'_j$ which gives $d_j \sim \mathrm{Po}(1 - \varepsilon'/2)$. Proceeding by induction, and using (1), we can define $(X_j)_{j\geq 0}$ and $(H'_j)_{j\geq 0}$ on a common probability space. Then we check $X_j + \log(4/\varepsilon') \geq H'_j$, also by induction: In the case $H'_{j-1} + k_j \leq \log(4/\varepsilon')$ we simply get

$$X_j + \log(4/\varepsilon') \geq \log(4/\varepsilon') \geq H'_{j-1} + k_j \geq H'_{j-1} + k_j + b_j - 1 = H'_j.$$

Otherwise we can use the inequality $b_j = \mathbb{1}_{\{g_j > H'_{j-1}+k_j\}} \leq \mathbb{1}_{\{g_j > \log(4/\varepsilon')\}} \leq b'_j$ to obtain

$$\begin{aligned} X_j + \log(4/\varepsilon') &\geq X_{j-1} + d_j - 1 + \log(4/\varepsilon') \overset{\text{(Ind.Hyp.)}}{\geq} H'_{j-1} + d_j - 1 \\ &= H'_{j-1} + k_j + b'_j - 1 \geq H'_{j-1} + k_j + b_j - 1 = H'_j. \end{aligned}$$

◀

2.6 Enter Queuing Theory

It turns out that, in essence, the behaviour of the Markov chain $(X_j)_{j\geq 0}$ has been studied in the literature under the name "M/D/1 queue", which is Kendall notation [28] for queues with "**M**arkovian arrivals, **D**eterministic service times and **1** server". We will exploit what is known about this simple queuing 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 $\rho = 1 - \varepsilon'/2$ (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 discretisation $(Z_j)_{j\in\mathbb{N}_0}$ of the M/D/1 queue. For $j \geq 1$, the number d_j of arrivals in between two observations Z_{j-1} and Z_j has distribution $d_j \sim \mathrm{Po}(\rho)$, and one customer was served in the meantime if and only if $Z_{j-1} > 0$. We can therefore write

$$Z_j = \begin{cases} d_j & \text{if } Z_{j-1} = 0, \\ Z_{j-1} + d_j - 1 & \text{if } Z_{j-1} > 0. \end{cases}$$

By reusing the variables $(d_j)_{j\geq 1}$ that previously occurred in the definition of $(X_j)_{j\geq 0}$, we already established a coupling between the processes $(X_j)_{j\geq 0}$ and $(Z_j)_{j\geq 0}$. A simple induction suffices to show

$$X_j = \max(0, Z_j - 1) \text{ for all } j \geq 0. \tag{2}$$

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 queuing theory:

▶ **Fact 1.**

(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 [12, Chapter 5.4].)

(ii) *[21, Prop 3.4] We have the following tail bound for the event $\{Z_t > k\}$ for any $k \in \mathbb{N}$:*

$$\Pr[Z_t > k] \leq \lim_{\tau\to\infty} \Pr[Z_\tau > k] = \mathrm{e}^{-k\cdot\Theta(\varepsilon)}, \text{ for all } t \geq 0.$$

2.7 Putting the Pieces Together

We now have everything in place to prove Theorem 2 regarding solving our linear systems.

Proof of Theorem 2. By the observation made in Section 1.4, we may assume that the random variables $(H_j)_{j\in[n+L-1]}$, $(H'_j)_{j\in[n+L-1]}$, $(X_j)_{j\geq 0}$ and $(Z_j)_{j\geq 0}$ and the three corresponding couplings are realized on one common probability space.

By Fact 1(ii) it is possible to choose $L = \Theta((\log m)/\varepsilon)$ while guaranteeing $\Pr[Z_j > L/2] = \mathcal{O}(m^{-2})$ for all $j \geq 0$.

By the choice of L and the union bound, the event $E_{\max Z} = \{\forall j \in [n+L-1]\colon Z_j \leq L/2\}$ occurs whp. Conditioned on $E_{\max Z}$ and the high probability event $E_{\geq m}$ from Lemma 4 we have

$$H_j \overset{\text{Lem. 4}}{\leq} H'_j \overset{\text{Lem. 5}}{\leq} X_j + \log(4/\varepsilon') \overset{\text{Eq. 2}}{\leq} Z_j + \log(4/\varepsilon') \overset{E_{\max Z}}{\leq} L/2 + \log(4/\varepsilon') \leq L - 2\log m.$$

By using Lemma 3(iii) we conclude that SGAUSS succeeds with high probability.

Along similar lines we get, for each $j \in [n+L-1]$:

$$\begin{aligned} E[H_j] &\overset{\text{Lem. 4}}{\leq} \mathbb{E}[H'_j \mid E_{\geq m}] \leq \tfrac{1}{\Pr[E_{\geq m}]}\,\mathbb{E}[H'_j] \overset{\text{Lem. 5}}{\leq} \tfrac{1}{\Pr[E_{\geq m}]}\,\mathbb{E}[X_j + \log(4/\varepsilon')] \\ &\overset{\text{Eq. 2}}{\leq} \tfrac{1}{\Pr[E_{\geq m}]}\,\mathbb{E}[Z_j + \log(4/\varepsilon')] \overset{\text{Fact 1(i)}}{\leq} \tfrac{1}{\Pr[E_{\geq m}]}(\mathcal{O}(1/\varepsilon) + \log(4/\varepsilon')) = \mathcal{O}(1/\varepsilon). \end{aligned}$$

By Lemma 3(ii) the expected number of row additions performed by a successful run of SGAUSS is therefore at most $\mathbb{E}[\sum_{j\in[n+L-1]} H_j] = \mathcal{O}(m/\varepsilon)$. Since unsuccessful runs happen with probability $\mathcal{O}(1/m)$ and can perform at most mL additions (each row can only be the target of L row additions), the overall expected number of additions is not skewed by unsuccessful runs, hence is also in $\mathcal{O}(m/\varepsilon)$. This finishes the proof of Theorem 2. ◀

▶ Remark. The analysis described in this section works in exactly the same way if instead of $\mathbb{F}_2$ a larger finite field $\mathbb{F}$ is used. A row in the random matrix is determined by a random starting position and a block of L random elements from $\mathbb{F}$. A row operation in the Gaussian elimination now consists of a division, a multiplication of a block with a scalar and a row addition. The running time of the algorithm will increase at least by a factor of $\log(|\mathbb{F}|)$ (the bitlength of a field element), and further increases depend on how well word parallelism in the word RAM can be utilized for operations like row additions, scalar multiplications and scalar products. (In [22], efficient methods are described for the field of three elements.) The queue length will become a little smaller, but not significantly, since even the M/D/1 queue with arrivals with a Poisson$(1-\varepsilon)$ distribution will lead to average queue length $\Theta(1/\varepsilon)$.

▶ Remark. An interesting question was raised by a reviewer of the submission: Is anything gained if we fix the first bit of each block to be 1? When checking our analysis for this case we see that this 1-bit need not survive previous row operations. However, such a change does improve success probabilities in the Robin Hood insertion procedure. If a key x_i finds cell h_i empty, it occupies this cell, without a coin being flipped. From the point of view of the queues, we see that now the derived variable b_j in Section 2.5 is 1 if $k_j > 0$ and geometrically distributed only if $k_j = 0$. As in the preceding remark, this brings the process closer to the M/D/1 queue with arrivals with a Poisson$(1-\varepsilon)$ distribution and deterministic service time 1, but the average queue length remains $\Theta(1/\varepsilon)$. Still, it may be interesting to check by experiments if improvements result by this change.

3 A New Retrieval Data Structure

With Theorem 2 in place we are ready to carry out the analysis of the retrieval data structure based on the new random matrices as described in Section 1.2. The proof of Theorem 1 is more or less straightforward.

Proof of Theorem 1. Denote the m elements of S by $x_1, \ldots, x_m$, let $n = \frac{1}{1-\varepsilon}m$, $L = \Theta(\frac{\log m}{\varepsilon})$ the number from Theorem 2 and $h\colon \mathcal{U} \to [n] \times \{0,1\}^L$ a fully random hash function. For construct, we associate the values $(s_i, p_i) := h(x_i)$ with each x_i for $i \in [m]$ and interpret them as a random band matrix $A = (a_{ij})_{i\in[m],\, j\in[n+L-1]}$, where for all $i \in [m]$ row a_i contains the pattern p_i starting at position s_i and 0's everywhere else. Moreover, let $\vec{b} \in \{0,1\}^m$ be the vector with entries $b_i = f(x_i)$ for $i \in [m]$. We call SGAUSS (Algorithm 1) with inputs A and $\vec{b}$, obtaining (in case of success) a solution $\vec{z} \in \{0,1\}^{n+L-1}$ of $A\vec{z} = \vec{b}$. The retrieval data structure is simply $\mathsf{DS}_f = \vec{z}$.

By Theorem 2 construct succeeds whp[6] (establishing (i)) and performs $\mathcal{O}(m/\varepsilon)$ row additions. Since additions affect only $L = \mathcal{O}(\frac{\log m}{\varepsilon})$ consecutive bits, and since a word RAM can deal with $\mathcal{O}(\log m)$ bits at once, a single row addition costs $\mathcal{O}(1/\varepsilon)$ time, leading to total expected running time $\mathcal{O}(m/\varepsilon^2)$ (which establishes (ii)).

The data structure $\mathsf{DS}_f = \vec{z}$ occupies exactly $\frac{1}{1-\varepsilon}m + L - 1 < (1+2\varepsilon)m$ bits. Replacing ε with $\varepsilon/2$ yields the literal result (iii).

To evaluate query(DS_f, y) for $y \in \mathcal{U}$, we compute $(s_y, p_y) = h(y)$ and the scalar product $b_y = \langle \vec{z}\,[s_y \ldots s_y{+}L{-}1], p_y\rangle := \bigoplus_{j=1}^{L} \vec{z}_{s_y+j-1} \cdot p_{yj}$. By construction, this yields $b_i = f(x_i)$ in the case that $y = x_i$. To obtain (iv), observe that the scalar product of two binary sequences of length $L = \mathcal{O}(\log(n)/\varepsilon)$ can be computed using $\mathcal{O}(1/\varepsilon)$ bit parallel AND and XOR operations, as well as a single PARITY operation on $\mathcal{O}(\log m)$ bits, which can be assumed to be available in constant time. ◀

▶ Remark. As the proof of Theorem 2 remains valid for arbitrary fixed finite fields in place of $\mathbb{F}_2$, the same is true for Theorem 1. This is relevant for the compact representation of functions with small ranges like [3], where binary encoding of single symbols implies extra space overhead. Such functions occur in data structures for perfect hash functions [7, 22].

[6] If success with probability 1 is desired, then in case the construction fails with hash function $h_0 = h$, we just restart the construction with different hash functions $h_1, h_2, \ldots$. In this setup, DS_f must also contain the seed $s \in \mathbb{N}_0$ identifying the first hash function h_s that led to success.

4 Input Partitioning

We examine the effect of a simple trick to improve construction and query times of our retrieval data structure. We partition the input into *chunks* using a "first-level hash function" and construct a separate retrieval data structure for each chunk. Using this with chunk size $C = m^\varepsilon$ will reduce the time bounds for construction and query by a factor of ε. The main reason for this is that we can use smaller block sizes L, which in turn makes row additions and inner products cheaper. Note that the idea is not new. Partitioning the input has previously been applied in the context of retrieval to reduce construction times, especially when "raw" construction times are superlinear [15, 22, 35] or when performance in external memory settings is an issue [3, 7]. Partitioning also allows us to get rid of the full randomness assumption, which is interesting from a theoretical point of view [7, 19, 16].

▶ Remark. The reader should be aware that the choice $C = m^\varepsilon$, which is needed to obtain a speedup of $1/\varepsilon$, is unlikely to be a good choice in practice and that this improvement only works for unrealistically large m. Namely, we use that $\frac{\log m}{m^\varepsilon} \ll \varepsilon$ for sufficiently large m. While the left term is indeed $o(1)$ and the right a constant, even for moderate values of ε like 0.05 implausibly large values of m are needed to satisfy the weaker requirement $\frac{\log m}{m^\varepsilon} < \varepsilon$. In this sense, Theorem 6 taken literally is of purely theoretical value. Still, the general idea is sound and it can give improvements in practice when partitioning less aggressively, say with $C \approx \sqrt{m}$. For example, the good running times reported in Section 5 are only possible with this splitting approach.

▶ **Theorem 6.** *The result of Theorem 1 can be strengthened in the following ways.*

- **(i)** *The statements of Theorem 1 continue to hold without the assumption of fully random hash functions being available for free.*
- **(ii)** *The expected construction time is $\mathcal{O}(m/\varepsilon)$ (instead of $\mathcal{O}(m/\varepsilon^2)$).*
- **(iii)** *The expected query time is $\mathcal{O}(1)$ (instead of $\mathcal{O}(1/\varepsilon)$). Queries involve accessing a (small) auxiliary data structure, so technically not all required data is "consecutive in memory".*

Proof Sketch. Let $C = m^\varepsilon$ be the *desired chunk size*. In [7, Section 4] it is described in detail how a splitting function can be used to obtain chunks that have size within a constant factor of C with high probability, and how fully random hash functions on each individual chunk can be provided by a randomized auxiliary structure $\mathcal{H}$ that takes only $o(m)$ space. New functions can be generated by switching to new seeds. (This construction is a variation of what is described in [16, 19].) This fully suffices for our purposes. We construct an individual retrieval data structure for each chunk with $L = \mathcal{O}(\frac{\log C}{\varepsilon}) = \mathcal{O}(\log m)$. Such a construction succeeds in expected time $\mathcal{O}(C/\varepsilon)$ with probability $1 - \mathcal{O}(1/C)$. In case the construction fails for a chunk, it is repeated with a different seed. At the end we save the concatenation of all m/C retrieval data structures, the data structure $\mathcal{H}$ and an auxiliary array. This array contains, for each chunk, the offset of the corresponding retrieval data structure in the concatenation and the seed of the hash function used for the chunk. It is easy to check that the size of all auxiliary data is asymptotically negligible.

The total expected construction time is $\mathcal{O}((m/C) \cdot C/\varepsilon) = O(m/\varepsilon)$, and since $L = \mathcal{O}(\log m)$, a retrieval query can be evaluated in constant time. ◀

▶ Remark. The construction from [30] described in item (4) in the list in Section 1.1 can also be transformed in a retrieval data structure. (This does not seem to have been explored up to now.) The expected running time for construct is $\mathcal{O}(m \log(1/\varepsilon))$ (better than our $\mathcal{O}(m/\varepsilon)$), the expected running time for query is $\mathcal{O}(\log(1/\varepsilon))$, with $\mathcal{O}(\log(1/\varepsilon))$ random accesses in

memory. (Worst case is $\mathcal{O}(1/\varepsilon)$.) In our preliminary experiments, see Section 5, for $m = 10^7$, both construction and query times of our construction seem to be able to compete well with the construction following [30].

5 Experiments

We implemented our retrieval data structure following the approach explained in the proof of Theorem 6, except that we used `MurmurHash3` [1] for all hash functions. This is a heuristic insofar as we depart from the full randomness assumption of Theorem 1. We report[7] running times and space overheads in Table 1, with the understanding that a retrieval data structure occupying N bits of memory in total and accommodating m keys has overhead $\frac{N}{m} - 1$. Concerning the choice of parameters, $L = 64$ has practical advantages on a 64-bit machine and $C = 10^4$ seems to go well with it experimentally. As $\varepsilon \in \{7\%, 5\%, 3\%\}$ decreases, the measured construction time increases as would be expected. This is partly due to the higher number of row additions in successful constructions, but also due to an increased probability for a chunk's construction to fail, which prompts a restart for that chunk with a different seed. Note that, in our implementation, querying an element in a chunk with non-default seed also prompts an additional hash function evaluation.

Competing Implementations. For comparison, we implemented the retrieval data structures from [7, 17, 22] and the one arising from the construction in [29]. (The number D in Table 1 is the maximum number of 1's in a row; the average is then $\Theta(\log D)$.)

In [7], the rows of the linear systems contain three 1's in independent and uniformly random positions. If the number of columns is $n = m/(1-\varepsilon)$ for $\varepsilon > 18.2\%$, the system can be solved in linear time by row and column *exchanges* alone. Compared to that method, we achieve smaller overheads at similar running times.

The approaches from [22] and [17] are different in that they construct linear systems that require cubic solving time with Gaussian elimination. This is counteracted by partitioning the input into chunks as well as by a heuristic *LazyGauss*-phase of the solver that eliminates many variables before the Method of Four Russians [2] is used on what remains. Construction times are higher than ours, but the tiny space overhead achieved in [17] is beyond the reach of our approach. The systems considered in [22] resemble those in [7], except at higher densities. The systems studied in [17] resemble our systems, except with *two* blocks of random bits per row instead of one.

We remark that our approach is easier to implement than those of [17, 22] but more difficult than that of [7].

6 Conclusion

This paper studies the principles of solving linear systems given by a particular kind of sparse random matrices, with one short random block per row, in a random position. The proof works by the point of view from Gaussian elimination to Robin Hood hashing and then to queuing theory. It might be interesting to find a direct, simpler proof for the main

[7] Experiments were performed on a desktop computer with an Intel® Core i7-2600 Processor @ 3.40GHz. Following [22], we used as data set S the first $m = 10^7$ URLs from the `eu-2015-host` dataset gathered by [4] with ≈ 80 bytes per key. As hash function we used MurmurHash3_x64_128 [1]. Reported query times are averages obtained by querying all elements of the data set once and include the evaluation of murmur, which takes about 25 ns on average. The reported figures are medians of 5 executions.

Table 1 Space overhead and running times per key of some practical retrieval data structures.

	Configuration	Overhead	construct [µs/key]	query [ns]
[7]	$\varepsilon = 19\%$	23.5%	0.32	59
⟨NEW⟩	$\varepsilon = 7\%, L = 64, C = 10^4$	8.8%	0.24	52
⟨NEW⟩	$\varepsilon = 5\%, L = 64, C = 10^4$	6.5%	0.27	54
⟨NEW⟩	$\varepsilon = 3\%, L = 64, C = 10^4$	4.3%	0.43	61
[29]	$c = 0.9, D = 12$	11.1%	0.79	94
[29]	$c = 0.99, D = 150$	1.1%	0.87	109
[22]	$\varepsilon = 9\%, k = 3, C = 10^4$	10.2%	1.30	58
[22]	$\varepsilon = 3\%, k = 4, C = 10^4$	3.4%	2.20	64
[17]	$\varepsilon = 0.05\%, \ell = 16, C = 10^4$	0.25%	2.47	56

theorem. Preliminary experiments concerning an application with retrieval data structures are promising. The most intriguing property is that evaluation of a retrieval query requires accessing only one (short) block in memory.

The potential of the construction in practice should be explored more fully and systematically, taking all relevant parameters like block size and chunk size into consideration. Constructions of perfect hash functions like in [7, 22] or Bloom filters that combine perfect hashing with fingerprinting [8, 15] might profit from our construction.

References

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

2 Gregory V. Bard. *Algebraic Cryptanalysis*, chapter The Method of Four Russians, pages 133–158. Springer US, Boston, MA, 2009. `doi:10.1007/978-0-387-88757-9_9`.

3 Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. Cache-oblivious peeling of random hypergraphs. In *Proc. DCC'14*, pages 352–361, 2014. `doi:10.1109/DCC.2014.48`.

4 Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. BUbiNG: Massive crawling for the masses. In *Proc. 23rd WWW'14*, pages 227–228. ACM, 2014. `doi:10.1145/2567948.2577304`.

5 Fabiano C. Botelho, Yoshiharu Kohayakawa, and Nivio Ziviani. A practical minimal perfect hashing method. In *Proc. 4th WEA*, pages 488–500, 2005. `doi:10.1007/11427186_42`.

6 Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proc. 10th WADS*, pages 139–150, 2007. `doi:10.1007/978-3-540-73951-7_13`.

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

8 Andrei Z. Broder and Michael Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 2003. `doi:10.1080/15427951.2004.10129096`.

9 Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin Hood hashing. In *Proc. 26th FOCS*, pages 281–288, 1985. `doi:10.1109/SFCS.1985.48`.

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

11 Colin Cooper. On the rank of random matrices. *Random Struct. Algor.*, 16(2):209–232, 2000. `doi:10.1002/(SICI)1098-2418(200003)16:2<209::AID-RSA6>3.0.CO;2-1`.

12 Robert B. Cooper. *Introduction to Queueing Theory*. Elsevier/North-Holland, 2nd edition, 1981. URL: `http://www.cse.fau.edu/~bob/publications/IntroToQueueingTheory_Cooper.pdf`.

13 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

14 Luc Devroye, Pat Morin, and Alfredo Viola. On worst-case Robin Hood hashing. *SIAM J. Comput.*, 33(4):923–936, 2004. `doi:10.1137/S0097539702403372`.

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

16 Martin Dietzfelbinger and Michael Rink. Applications of a splitting trick. In *Proc. 36th ICALP (1)*, pages 354–365, 2009. `doi:10.1007/978-3-642-02927-1_30`.

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

18 Martin Dietzfelbinger and Stefan Walzer. Dense peelable random uniform hypergraphs. In *Proc. 27th ESA*, pages 38:1–38:16, 2019. `doi:10.4230/LIPIcs.ESA.2019.38`.

19 Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theor. Comput. Sci.*, 380(1-2):47–68, 2007. `doi:10.1016/j.tcs.2007.02.054`.

20 Wayne Eberly. On efficient band matrix arithmetic. In *Proc. 33rd FOCS*, pages 457–463, 1992. `doi:10.1109/SFCS.1992.267806`.

21 Regina Egorova, Bert Zwart, and Onno Boxma. Sojourn time tails in the M/D/1 processor sharing queue. *Probab. Eng. Inf. Sci.*, 20:429–446, 2006. `doi:10.1017/S0269964806060268`.

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

23 Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.

24 George Havas, Bohdan S. Majewski, Nicholas C. Wormald, and Zbigniew J. Czech. Graphs, hypergraphs and hashing. In *Proc. 19th WG*, pages 153–165, 1993. `doi:10.1007/3-540-57899-4_49`.

25 Svante Janson. Individual displacements for linear probing hashing with different insertion policies. *ACM Trans. Algorithms*, 1(2):177–213, 2005. `doi:10.1145/1103963.1103964`.

26 Svante Janson. Individual displacements in hashing with coalesced chains. *Comb. Probab. Comput.*, 17(6):799–814, 2008. `doi:10.1017/S0963548308009395`.

27 Svante Janson and Alfredo Viola. A unified approach to linear probing hashing with buckets. *Algorithmica*, 75(4):724–781, 2016. `doi:10.1007/s00453-015-0111-x`.

28 David G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *Ann. Math. Statist.*, 24(3):338–354, September 1953. `doi:10.1214/aoms/1177728975`.

29 Michael Luby, Michael Mitzenmacher, Mohammad Amin Shokrollahi, and Daniel A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47(2):569–584, 2001. `doi:10.1109/18.910575`.

30 Michael Luby, Michael Mitzenmacher, Mohammad Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical loss-resilient codes. In *Proc. 29th STOC*, pages 150–159, 1997. `doi:10.1145/258533.258573`.

31 Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

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

33 Victor Y. Pan, Isdor Sobze, and Antoine Atinkpahoun. On parallel computations with banded matrices. *Inf. Comput.*, 120(2):237–250, 1995. `doi:10.1006/inco.1995.1111`.

34 Boris Pittel and Gregory B. Sorkin. The satisfiability threshold for k-XORSAT. *Comb. Probab. Comput.*, 25(2):236–268, 2016. doi:10.1017/S0963548315000097.

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

36 Alfredo Viola. Exact distribution of individual displacements in linear probing hashing. *ACM Trans. Algorithms*, 1(2):214–242, 2005. doi:10.1145/1103963.1103965.

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