On Hashing by (Random) Equations

Martin Dietzfelbinger

Technische Universität Ilmenau

ESA 2023, Amsterdam, September 4, 2023

Thanks to collaborators (over the time)

Thanks to collaborators (over the time)

- Stefan Walzer

Thanks to collaborators (over the time)

- Stefan Walzer
- Rasmus Pagh

Thanks to collaborators (over the time)

- Stefan Walzer
- Rasmus Pagh
- Peter Dillinger
- Andreas Goerdt
- Lorenz Hübschle-Schneider
- Michael Mitzenmacher
- Michael Rink
- Peter Sanders

1. Retrieval

Given: $\mathcal{U}$, set of all possible keys.

$S \subseteq \mathcal{U}$, set of size n, and mapping $f\colon S \to G$, for a set G.

Want: Data structure $\mathcal{R}_f$ for computing f.

1. Retrieval

Given: $\mathcal{U}$, set of all possible keys.

$S \subseteq \mathcal{U}$, set of size n, and mapping $f\colon S \to G$, for a set G.

Want: Data structure $\mathcal{R}_f$ for computing f.

Construction algorithm BUILD gets f and builds $\mathcal{R} = \mathcal{R}_f$.

1. Retrieval

Given: $\mathcal{U}$, set of all possible keys.

$S \subseteq \mathcal{U}$, set of size n, and mapping $f\colon S \to G$, for a set G.

Want: Data structure $\mathcal{R}_f$ for computing f.

Construction algorithm $\textsc{build}$ gets f and builds $\mathcal{R} = \mathcal{R}_f$.

Evaluation algorithm $\textsc{query}$ gets $x \in \mathcal{U}$, returns $\textsc{query}(x, \mathcal{R}) \in G$ such that

$$\textsc{query}(x, \mathcal{R}) = f(x) \text{ for all } x \in S.$$

Nothing is required for $x \notin S$.

1. Retrieval

Example: $G = \{\mathsf{m}, \mathsf{f}, \mathsf{u}\}$, S is a set of first names, f maps names to their gender.

$f(\text{Albert}) = \mathsf{m}$, $f(\text{Bertha}) = \mathsf{f}$, $f(\text{Carol}) = \mathsf{u}$, ... (u = undecided.)

1. Retrieval

Example: $G = \{\mathsf{m}, \mathsf{f}, \mathsf{u}\}$, S is a set of first names, f maps names to their gender.

$f(\text{Albert}) = \mathsf{m}$, $f(\text{Bertha}) = \mathsf{f}$, $f(\text{Carol}) = \mathsf{u}$, ... (u = undecided.)

Goals:

- Fast BUILD (ideal: time "linear in" n)

1. Retrieval

Example: $G = \{\mathsf{m}, \mathsf{f}, \mathsf{u}\}$, S is a set of first names, f maps names to their gender.

$f(\text{Albert}) = \mathsf{m}$, $f(\text{Bertha}) = \mathsf{f}$, $f(\text{Carol}) = \mathsf{u}$, ... (u = undecided.)

Goals:

- Fast BUILD (ideal: time "linear in" n)
- Fast QUERY (ideal: "constant" time **and** very few random accesses into storage area that holds $\mathcal{R}$)
- Compactness/Conciseness: "small" space for $\mathcal{R}$.

1. Retrieval

Example: $G = \{\mathsf{m}, \mathsf{f}, \mathsf{u}\}$, S is a set of first names, f maps names to their gender.

$f(\text{Albert}) = \mathsf{m}$, $f(\text{Bertha}) = \mathsf{f}$, $f(\text{Carol}) = \mathsf{u}$, ... (u = undecided.)

Goals:

- Fast BUILD (ideal: time "linear in" n)
- Fast QUERY (ideal: "constant" time **and** very few random accesses into storage area that holds $\mathcal{R}$)
- Compactness/Conciseness: "small" space for $\mathcal{R}$.

Listing n values alone (does not solve the problem but) takes space $n \log_2 3$.
Can't be beaten (information theory).

1. Retrieval

Simplistic: Static **dictionary** for f.

1. Retrieval

Simplistic: Static **dictionary** for f.

Operations: $\textsc{build}(f)$, $\textsc{query}(x)$ gives $f(x)$ for $x \in S$ and $\perp$ otherwise.

1. Retrieval

Simplistic: Static **dictionary** for f.

Operations: $\textsc{build}(f)$, $\textsc{query}(x)$ gives $f(x)$ for $x \in S$ and $\perp$ otherwise.

Overkill, and waste of space: essentially $n(\log|\mathcal{U}| + \log|G|)$ bits.

1. Retrieval

Simplistic: Static **dictionary** for f.

Operations: $\textsc{build}(f)$, $\textsc{query}(x)$ gives $f(x)$ for $x \in S$ and $\perp$ otherwise.

Overkill, and waste of space: essentially $n(\log|\mathcal{U}| + \log|G|)$ bits.

Information theory lower space bound for retrieval:

$$\mathbf{min}_{G,n} := n \log |G|.$$

1. Retrieval

Simplistic: Static **dictionary** for f.

Operations: $\textsc{build}(f)$, $\textsc{query}(x)$ gives $f(x)$ for $x \in S$ and $\perp$ otherwise.

Overkill, and waste of space: essentially $n(\log|\mathcal{U}| + \log|G|)$ bits.

Information theory lower space bound for retrieval:

$$\mathbf{min}_{G,n} := n \log |G|.$$

Desired space bounds: $(1+\varepsilon)\min_{G,n}$ for "small" **overhead** ε.

"concise": $\varepsilon = o(1)$,

1. Retrieval

Simplistic: Static **dictionary** for f.

Operations: $\textsc{build}(f)$, $\textsc{query}(x)$ gives $f(x)$ for $x \in S$ and $\perp$ otherwise.

Overkill, and waste of space: essentially $n(\log|\mathcal{U}| + \log|G|)$ bits.

Information theory lower space bound for retrieval:

$$\mathbf{min}_{G,n} := n \log |G|.$$

Desired space bounds: $(1+\varepsilon)\min_{G,n}$ for "small" **overhead** ε.

"concise": $\varepsilon = o(1)$, "compact" in [Nav16]: any reasonable notion of $\min_{G,n}$ + "little".

1. Retrieval

Simplistic: Static **dictionary** for f.

Operations: $\textsc{build}(f)$, $\textsc{query}(x)$ gives $f(x)$ for $x \in S$ and $\perp$ otherwise.

Overkill, and waste of space: essentially $n(\log|\mathcal{U}| + \log|G|)$ bits.

Information theory lower space bound for retrieval:

$$\mathbf{min}_{G,n} := n \log |G|.$$

Desired space bounds: $(1+\varepsilon)\min_{G,n}$ for "small" **overhead** ε.

"concise": $\varepsilon = o(1)$, "compact" in [Nav16]: any reasonable notion of $\min_{G,n}$ + "little".

Details of the $o(\dots)$ term are interesting!

Overview

1. Retrieval
2. Warmup: Equations, Peeling

Overview

1. Retrieval
2. Warmup: Equations, Peeling
3. Orientability
4. Solvability
5. Peeling up to the orientability threshold

Overview

1. Retrieval
2. Warmup: Equations, Peeling
3. Orientability
4. Solvability
5. Peeling up to the orientability threshold
6. Helpful: Sharding/Splitting

Overview

1. Retrieval
2. Warmup: Equations, Peeling
3. Orientability
4. Solvability
5. Peeling up to the orientability threshold
6. Helpful: Sharding/Splitting
7. Two blocks
8. One block: Sorted solving
9. One block: Ribbon
10. Bumping, batch bumping, overloading

Disclaimers and caveats

- Not (always) "best" implementations, using all types of handles and tricks, but focus on an interesting technology, giving raise to nice mathematical arguments.
- Focus on **retrieval**.

Disclaimers and caveats

- Not (always) "best" implementations, using all types of handles and tricks, but focus on an interesting technology, giving raise to nice mathematical arguments.
- Focus on **retrieval**.
- Focus on the static problem.

Disclaimers and caveats

- Not (always) "best" implementations, using all types of handles and tricks, but focus on an interesting technology, giving raise to nice mathematical arguments.
- Focus on **retrieval**.
- Focus on the static problem.
- Related problems: Perfect hashing, simulation of fully random hash functions, (static) filters,. . . : Omitted here.

Disclaimers and caveats

- Not (always) "best" implementations, using all types of handles and tricks, but focus on an interesting technology, giving raise to nice mathematical arguments.
- Focus on **retrieval**.
- Focus on the static problem.
- Related problems: Perfect hashing, simulation of fully random hash functions, (static) filters,. . . : Omitted here.
- **(This is serious!)** Recently: A lot of developments in direction of **dynamic** retrieval data structures and of course filters, which I won't touch.

Disclaimers and caveats

- Not (always) "best" implementations, using all types of handles and tricks, but focus on an interesting technology, giving raise to nice mathematical arguments.
- Focus on **retrieval**.
- Focus on the static problem.
- Related problems: Perfect hashing, simulation of fully random hash functions, (static) filters,. . . : Omitted here.
- **(This is serious!)** Recently: A lot of developments in direction of **dynamic** retrieval data structures and of course filters, which I won't touch.

(Slides on homepage.)

2. Warmup: Equations, Peeling

(*)

2. Warmup: Equations, Peeling

To store f "implicitly", G needs to have structure: Let $(G, \oplus)$ be an abelian group.

Example: Identify $G = \{\mathsf{m}, \mathsf{f}, \mathsf{u}\}$ with $\{0, 1, 2\}$, let $\oplus$ be addition modulo 3.

Choose $m \geq n$. Assume a mapping

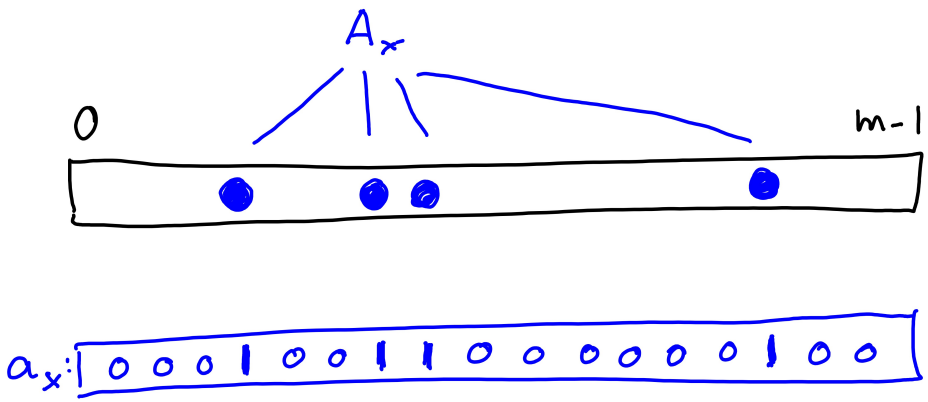
$$H : \mathcal{U} \ni x \mapsto A_x \subseteq [m]$$

is given. Alternative: $a_x = ([j \in A_x])_{j \in [m]} \in \{0,1\}^m$, the characteristic vector of A_x.

(Regard $H : x \mapsto A_x$ resp. $h : x \mapsto a_x$ as a *hash function*.)

Seek a vector $Z[0..m-1]$ over G with

$$f(x) = \bigoplus_{j \in A_x} Z[j], \text{ for } x \in S. \qquad (*)$$


A_x
0
m-1
a_x: 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 0

2. Warmup: Equations, Peeling

$$f(x) = \bigoplus_{j \in A_x} Z[j], \text{ for } x \in S. \qquad (*)$$

Then $Z[0..m-1]$ can be used as data structure $\mathcal{R}_f$ [SH94].

Example: $m = 4$, $A_{\text{Albert}} = \{1, 2\}$, $A_{\text{Bertha}} = \{0, 2\}$, $A_{\text{Carol}} = \{1, 3\}$.
$Z = [1, 0, 0, 2] = [\text{f}, \text{m}, \text{m}, \text{u}]$ does the job.

2. Warmup: Equations, Peeling

$$f(x) = \bigoplus_{j \in A_x} Z[j], \text{ for } x \in S. \qquad (*)$$

2. Warmup: Equations, Peeling

$$f(x) = \bigoplus_{j \in A_x} Z[j], \text{ for } x \in S. \qquad (*)$$

Questions:

- What is the cost/space for H resp. h?

2. Warmup: Equations, Peeling

$$f(x) = \bigoplus_{j \in A_x} Z[j], \text{ for } x \in S. \qquad (*)$$

Questions:

- What is the cost/space for H resp. h?
- When (for which $m = (1+\varepsilon)n$) can we hope that solutions Z exist?

2. Warmup: Equations, Peeling

$$f(x) = \bigoplus_{j \in A_x} Z[j], \text{ for } x \in S. \qquad (*)$$

Questions:

- What is the cost/space for H resp. h?
- When (for which $m = (1+\varepsilon)n$) can we hope that solutions Z exist?
- What is the cost of finding solution Z?

2. Warmup: Equations, Peeling

$$f(x) = \bigoplus_{j \in A_x} Z[j], \text{ for } x \in S. \qquad (*)$$

Questions:

- What is the cost/space for H resp. h?
- When (for which $m = (1+\varepsilon)n$) can we hope that solutions Z exist?
- What is the cost of finding solution Z?
- What is the query time?

2. Warmup: Equations, Peeling

$$f(x) = \bigoplus_{j \in A_x} Z[j], \text{ for } x \in S. \qquad (*)$$

Questions:

- What is the cost/space for H resp. h?
- When (for which $m = (1+\varepsilon)n$) can we hope that solutions Z exist?
- What is the cost of finding solution Z?
- What is the query time?

Last question first:

Group operations in constant time $\rightarrow$ $O(|A_x|) = O(\|a_x\|)$ query time.

2. Warmup: Equations, Peeling

About H and h we assume they are given for free, including all randomness possibly involved, and can be evaluated in time $O(|A_x|)$ (unless stated otherwise).
Pretty steep assumption; can be justified here in a sense ("Split-and-Share" [DR09]).

$$f(x) = \bigoplus_{j \in A_x} Z[j], \text{ for } x \in S. \qquad (*)$$

A solution Z always exists (for arbitrary G) if and only if $(A_x)_{x \in S}$ is **peelable**, i.e. if one can arrange S as $x_1, \ldots, x_n$ such that

$$A_{x_i} - \bigcup_{\ell > i} A_{x_\ell} \neq \emptyset, \quad \text{for all } i.$$

Equivalent: The $n \times m$-matrix $\boldsymbol{A_{S,h}} := (a_x)_{x \in S}$ can be brought into **row echelon form** by exchanging rows and exchanging columns. We also say: $A_{S,h}$ is peelable.

$A_{s,h}$, permute rows and columns

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \longrightarrow \begin{array}{c} x_2 \\ x_1 \\ x_4 \\ x_3 \end{array} \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \longrightarrow$$

$$\longrightarrow \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} z_1 \\ z_3 \\ z_0 \\ z_5 \\ z_2 \\ z_4 \end{pmatrix} = \begin{pmatrix} f(x_2) \\ f(x_1) \\ f(x_4) \\ f(x_3) \end{pmatrix}$$

Back-Substitution!

2. Warmup: Equations, Peeling

Good for fast query times: $|A_x|$ is a small constant k, at least "on average".

2. Warmup: Equations, Peeling

Good for fast query times: $|A_x|$ is a small constant k, at least "on average".

Great about peelability: If solution exists, it can be found in linear time:
Find peeling order (by standard data structures), use back-substitution.

2. Warmup: Equations, Peeling

Good for fast query times: $|A_x|$ is a small constant k, at least "on average".

Great about peelability: If solution exists, it can be found in linear time:
Find peeling order (by standard data structures), use back-substitution.

Small, constant $k = |A_x|$ [MWHC96] (rougher version re-discovered in [CKRT04]):
For random sets A_x of size k there is a threshold c_k^0 such that
$(A_x)_{x \in S}$ is peelable w.h.p. (roughly) for $m > c_k^0 n$.

2. Warmup: Equations, Peeling

Good for fast query times: $|A_x|$ is a small constant k, at least "on average".

Great about peelability: If solution exists, it can be found in linear time:
Find peeling order (by standard data structures), use back-substitution.

Small, constant $k = |A_x|$ [MWHC96] (rougher version re-discovered in [CKRT04]):
For random sets A_x of size k there is a threshold c_k^0 such that
$(A_x)_{x \in S}$ is peelable w.h.p. (roughly) for $m > c_k^0 n$.

k	2	3	4	5	6
c_k^0	2	1.222	1.295	1.425	1.570

2. Warmup: Equations, Peeling

Good for fast query times: $|A_x|$ is a small constant k, at least "on average".

Great about peelability: If solution exists, it can be found in linear time:
Find peeling order (by standard data structures), use back-substitution.

Small, constant $k = |A_x|$ [MWHC96] (rougher version re-discovered in [CKRT04]):
For random sets A_x of size k there is a threshold c_k^0 such that
$(A_x)_{x \in S}$ is peelable w.h.p. (roughly) for $m > c_k^0 n$.

k	2	3	4	5	6
c_k^0	2	1.222	1.295	1.425	1.570

One can show: $c_k^0 \nearrow \infty$ for $k \geq 3$.

2. Warmup: Equations, Peeling

Good for fast query times: $|A_x|$ is a small constant k, at least "on average".

Great about peelability: If solution exists, it can be found in linear time:
Find peeling order (by standard data structures), use back-substitution.

Small, constant $k = |A_x|$ [MWHC96] (rougher version re-discovered in [CKRT04]):
For random sets A_x of size k there is a threshold c_k^0 such that
$(A_x)_{x \in S}$ is peelable w.h.p. (roughly) for $m > c_k^0 n$.

k	2	3	4	5	6
c_k^0	2	1.222	1.295	1.425	1.570

One can show: $c_k^0 \nearrow \infty$ for $k \geq 3$.

So $\mathcal{R}_f$ constructed in this way will take space no less than $1.222 \log |G|$.

Too bad!

2. Warmup: Equations, Peeling

Good for fast query times: $|A_x|$ is a small constant k, at least "on average".

Great about peelability: If solution exists, it can be found in linear time:
Find peeling order (by standard data structures), use back-substitution.

Small, constant $k = |A_x|$ [MWHC96] (rougher version re-discovered in [CKRT04]):
For random sets A_x of size k there is a threshold c_k^0 such that
$(A_x)_{x \in S}$ is peelable w.h.p. (roughly) for $m > c_k^0 n$.

k	2	3	4	5	6
c_k^0	2	1.222	1.295	1.425	1.570

One can show: $c_k^0 \nearrow \infty$ for $k \geq 3$.

So $\mathcal{R}_f$ constructed in this way will take space no less than $1.222 \log |G|$.

Too bad! (Not end of story, see below . . .)

3. Orientability

3. Orientability

With each $x \in S$ associate a (uniformly) random set $A_x \subseteq [m]$ of size k.

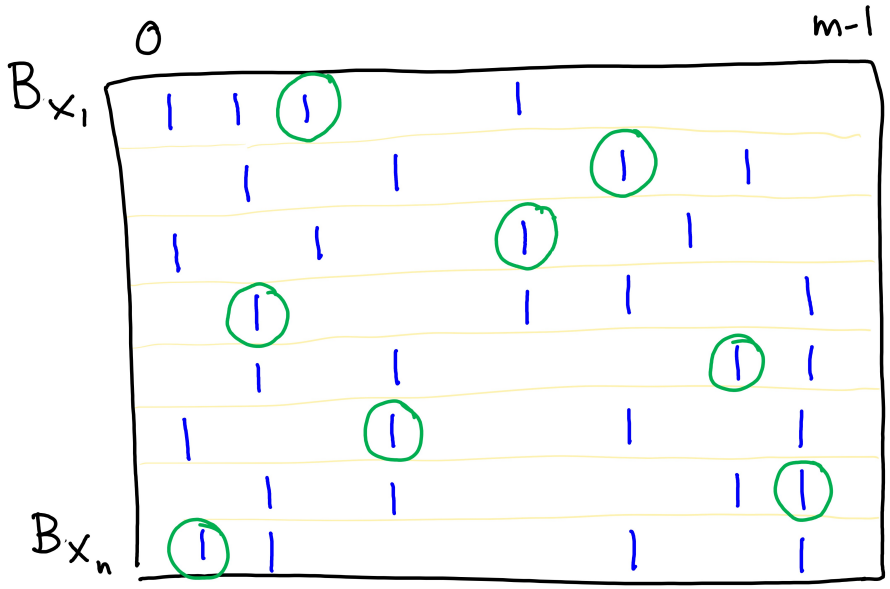
$(A_x)_{x \in S}$ is called **orientable** if there is a one-to-one mapping $\tau \colon S \to [t]$ such that $\tau(x) \in A_x$ for $x \in S$.

Notes. (1) $(A_x)_{x \in S}$ is an order-k random hypergraph with node set $[m]$, and this orientability notion is standard.
(2) Orientability gets (k-ary) cuckoo hashing going [FPSS05] (not our focus).

[FP10, FM12, DGM$^+$10] established *orientability thresholds* c_k, $k \geq 2$, so that (roughly) for $m \geq c_k n$ a random set $(A_x)_{x \in S}$ is orientable w.h.p., but not for smaller m.

k	2	3	4	5	6
c_k	2	1.089	1.024	1.0076	1.0026

One can show: $c_k - 1 \sim e^{-k}$.
Much more pleasant than c_k^0!


0
m-1
B_{x_1}
B_{x_n}

4. Solvability

Switch to $G = \{0,1\}^r$ with $\oplus =$ bitwise XOR on r bits. Our field: $\mathbb{Z}_2 = \{0,1\}$.

(Recall your linear algebra!)

As before: $A_S = (a_x)_{x \in S}$, an $n \times m$-matrix. Order of rows: irrelevant.

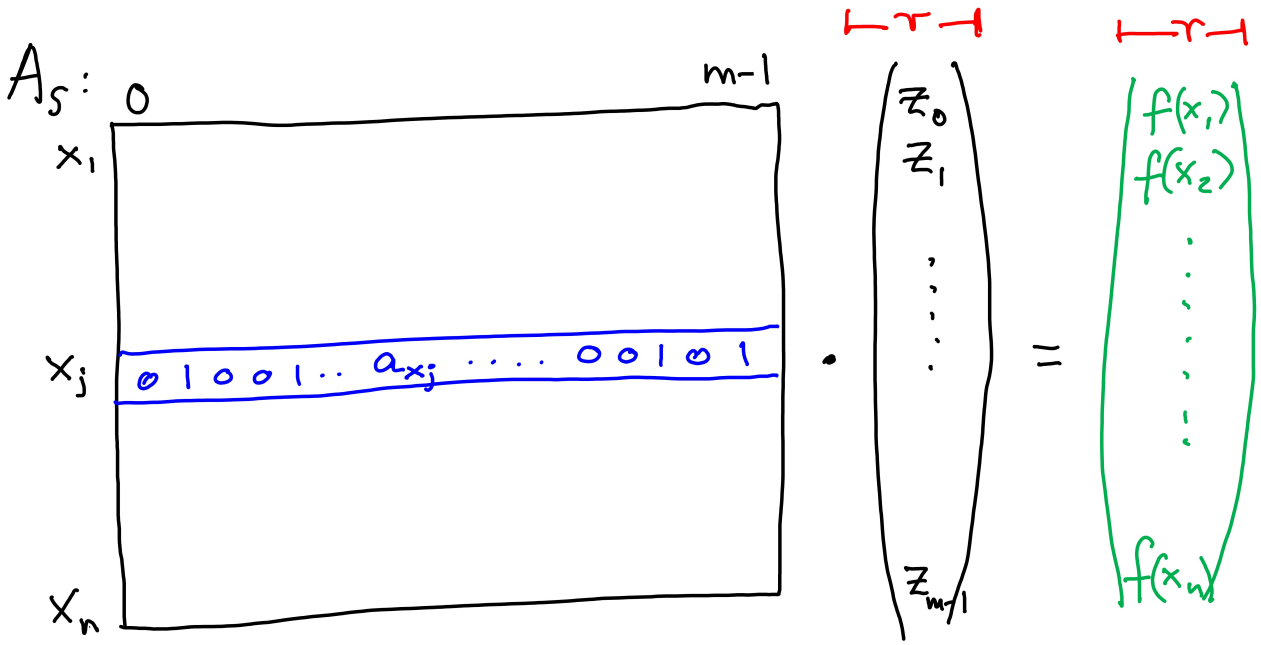
For retrieval: BUILD needs to **solve** the linear system

$$A_S \cdot z = f,$$

where $f = (f(x))_{x \in S} \in (\{0,1\}^r)^n$ is given and $z \in (\{0,1\}^r)^m$ is unknown.

(Actually, these are r linear systems over $\mathbb{Z}_2$, treated simultaneously. May focus on $r = 1$.)

Clear: A_S has linearly independent rows (i.e. row rank n) $\Rightarrow$ solution always exists.



$\Uparrow$

A_S needs to have full row rank

4. Solvability

Classic scenario: A_x, $x \in S$, is a fully random k-subset of $[m]$.

4. Solvability

Classic scenario: A_x, $x \in S$, is a fully random k-subset of $[m]$.

Surprisingly, it turned out (claims in [DM02, DGM+10], full proof in [PS16]) that solvability of resulting matrices A_S **has the same thresholds as orientability**.

4. Solvability

Classic scenario: A_x, $x \in S$, is a fully random k-subset of $[m]$.

Surprisingly, it turned out (claims in [DM02, DGM+10], full proof in [PS16]) that solvability of resulting matrices A_S **has the same thresholds as orientability**.

So we have solvability for $m \geq c_k n$ (roughly), where $c_k - 1 \sim e^{-k}$.

4. Solvability

Classic scenario: A_x, $x \in S$, is a fully random k-subset of $[m]$.

Surprisingly, it turned out (claims in [DM02, DGM+10], full proof in [PS16]) that solvability of resulting matrices A_S **has the same thresholds as orientability**.

So we have solvability for $m \geq c_k n$ (roughly), where $c_k - 1 \sim e^{-k}$.

Means QUERY gets by with time $O(k)$ and k random accesses into array $Z[0..m-1]$.

4. Solvability

Classic scenario: A_x, $x \in S$, is a fully random k-subset of $[m]$.

Surprisingly, it turned out (claims in [DM02, DGM+10], full proof in [PS16]) that solvability of resulting matrices A_S **has the same thresholds as orientability**.

So we have solvability for $m \geq c_k n$ (roughly), where $c_k - 1 \sim e^{-k}$.

Means QUERY gets by with time $O(k)$ and k random accesses into array $Z[0..m-1]$.
For solving: Gaussian elimination ($O(n^3)$).
Wiedemann's algorithm [Wie86] avoids the proliferation of 1's: with k many 1's per row in the original system it has running time $O(n^2 k)$.

4. Solvability

Classic scenario: A_x, $x \in S$, is a fully random k-subset of $[m]$.

Surprisingly, it turned out (claims in [DM02, DGM+10], full proof in [PS16]) that solvability of resulting matrices A_S **has the same thresholds as orientability**.

So we have solvability for $m \geq c_k n$ (roughly), where $c_k - 1 \sim e^{-k}$.

Means QUERY gets by with time $O(k)$ and k random accesses into array $Z[0..m-1]$.
For solving: Gaussian elimination ($O(n^3)$).
Wiedemann's algorithm [Wie86] avoids the proliferation of 1's: with k many 1's per row in the original system it has running time $O(n^2 k)$.

Detailed study of improvements for Gaussian elimination in sparse systems, by word parallelism and clever reduction techniques, extensive experimental evaluation: [GOV16, GOV20].

5. Peeling up to the orientability threshold

5. Peeling up to the orientability threshold

Want to save peeling with $\approx k$ many 1s per row to higher densities.

Linear construction time $+ O(k)$ query time!

5. Peeling up to the orientability threshold

Want to save peeling with $\approx k$ many 1s per row to higher densities.

Linear construction time $+ O(k)$ query time!

[LMSS01] (context: erasure correcting codes) use $m = (1 + \frac{1}{k})n$, a special distribution on the set sizes $|A_x|$ with expectation $O(k) = O(\log D)$ and maximum D.

5. Peeling up to the orientability threshold

Want to save peeling with $\approx k$ many 1s per row to higher densities.

Linear construction time $+ O(k)$ query time!

[LMSS01] (context: erasure correcting codes) use $m = (1 + \frac{1}{k})n$, a special distribution on the set sizes $|A_x|$ with expectation $O(k) = O(\log D)$ and maximum D.

Expected time for BUILD is $O(n^2 k)$ [Wie86], **expected** query time is $O(k)$, worst case query time is $O(D)$ (not so good).

5. Peeling up to the orientability threshold

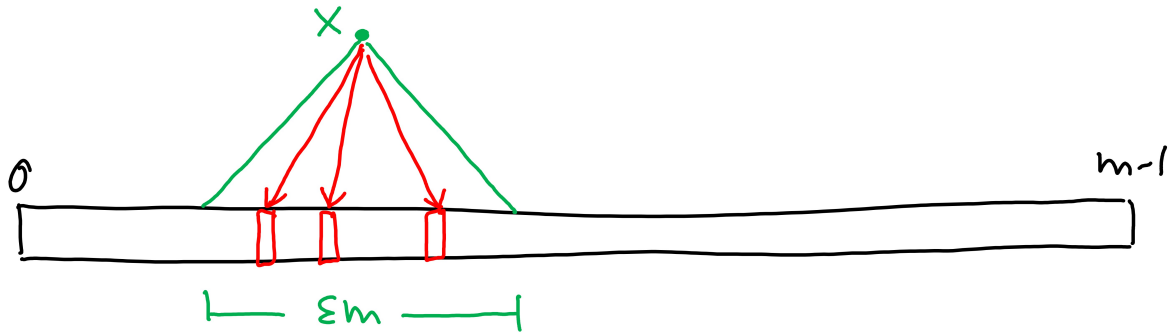
Different approach: **"Spatial coupling"**, described in [DW19b], fully analyzed in [Wal21], using machinery developed in the context of coding theory [KRU15].

A_x is chosen at random in two stages:

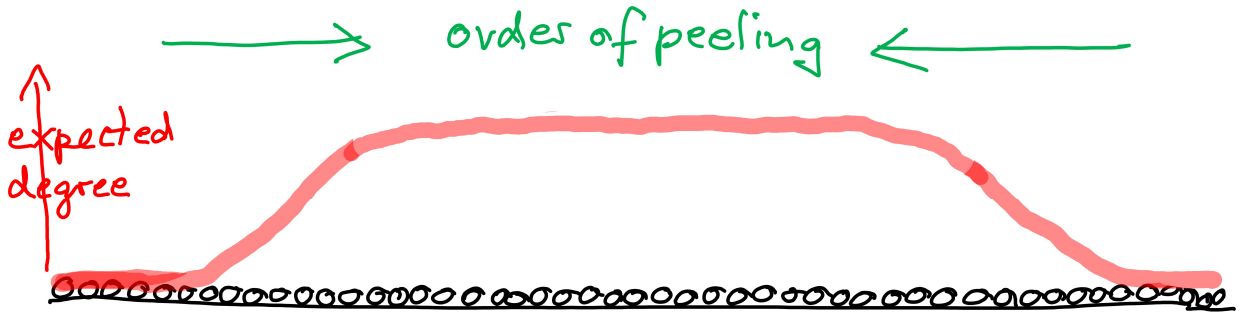
- A "window" W of width εm with random position in $[m]$ is chosen.
- A_x is a random k-size subset of the window.

Theorem [Wal21]
Given $c > c_k$ (the *orientability* threshold), one can choose $\varepsilon > 0$ such that for n large enough the system $(A_x)_{x \in S}$ with $m = cn$ allows *peeling* w.h.p.

What mechanism is behind this? Roughly, the peeling process runs "from the outside in".
Close to the borders the average degree of a point is smaller than the overall average, and there is always a high probability to have nodes of degree 1.



Peeling process:



6. Helpful: Sharding/Splitting

6. Helpful: Sharding/Splitting

Old idea, used often in theoretical constructions.

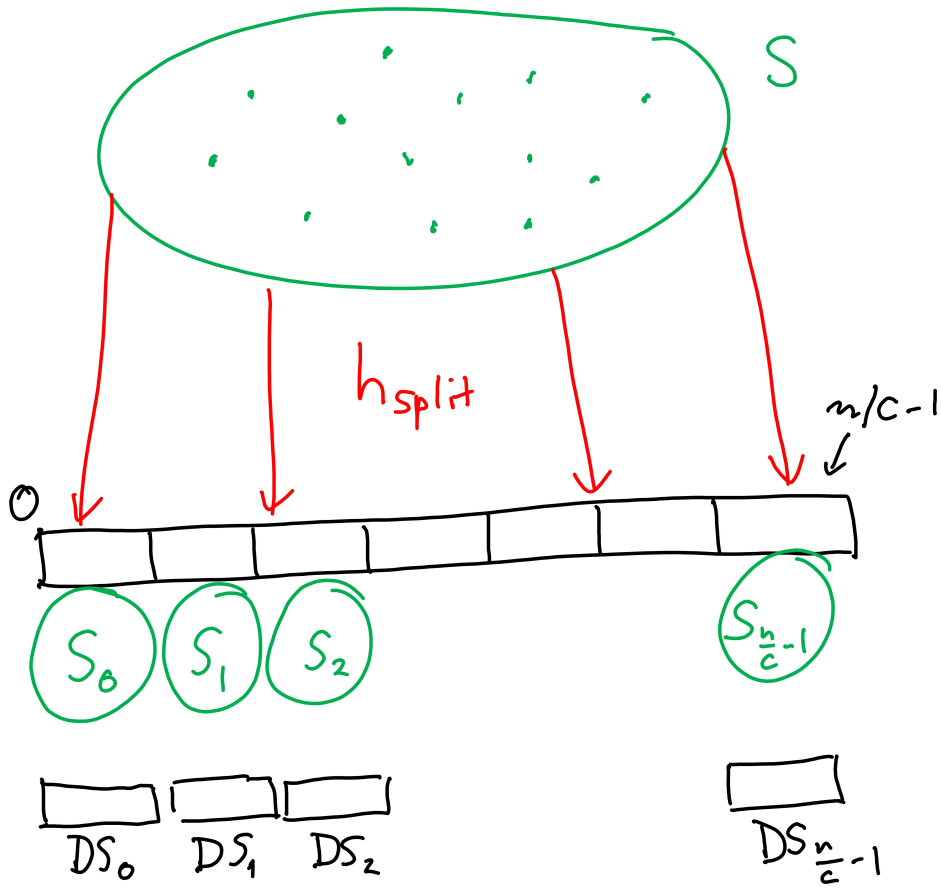
Most useful in practical tuning of implementations (e.g. [BPZ13, GOV16, GOV20]) and in justifying "full randomness assumption" [DR09].

Given S with $|S| = n$, use hash function $h_{\text{split}} \colon \mathcal{U} \to [0..n/C]$ to split S into pieces ("shards") $S_u = S \cap h_{\text{split}}^{-1}(u)$, for $u \in [n/C]$.

Treat the S_u separately. *Expected* shard size: C.

Version 1: Use bound n' on $|S_u|$ that is kept with high probability by all shards. (Extra space overhead due to random fluctuation (underflow!). May have special treatment for overflowing shards.)

Version 2: Calculate $n_u = |S_u|$, for each u, allocate space correspondingly. (Extra space overhead for storing offsets.)


S
h_{split}
$n/c-1$
0
S_0
S_1
S_2
$S_{\frac{n}{c}-1}$
DS_0
DS_1
DS_2
$DS_{\frac{n}{c}-1}$

6. Helpful: Sharding/Splitting

Options:

- Give special treatment to shards that are too large or to overflowing keys (e.g. [Por09, ANS10, PBC+23]).

6. Helpful: Sharding/Splitting

Options:

- Give special treatment to shards that are too large or to overflowing keys (e.g. [Por09, ANS10, PBC+23]).
- In case construction for S_u fails, start over with new seed for randomness. (Adds small overhead for storing seed.)

6. Helpful: Sharding/Splitting

Options:

- Give special treatment to shards that are too large or to overflowing keys (e.g. [Por09, ANS10, PBC+23]).
- In case construction for S_u fails, start over with new seed for randomness. (Adds small overhead for storing seed.)

If Gaussian elimination takes time $O(n^3)$, then shard size $n^\theta \leq C \leq n$ leads to overall solution time $O(n^2 C)$.

6. Helpful: Sharding/Splitting

Options:

- Give special treatment to shards that are too large or to overflowing keys (e.g. [Por09, ANS10, PBC+23]).
- In case construction for S_u fails, start over with new seed for randomness. (Adds small overhead for storing seed.)

If Gaussian elimination takes time $O(n^3)$, then shard size $n^\theta \leq C \leq n$ leads to overall solution time $O(n^2 C)$.

With Wiedemann's algorithm [Wie86] (running time $O(n^2 k)$) and sharding we get construction time $O(knC)$.

6. Helpful: Sharding/Splitting

Options:

- Give special treatment to shards that are too large or to overflowing keys (e.g. [Por09, ANS10, PBC+23]).
- In case construction for S_u fails, start over with new seed for randomness. (Adds small overhead for storing seed.)

If Gaussian elimination takes time $O(n^3)$, then shard size $n^\theta \leq C \leq n$ leads to overall solution time $O(n^2 C)$.

With Wiedemann's algorithm [Wie86] (running time $O(n^2 k)$) and sharding we get construction time $O(knC)$.

Watch out: "High probability" in C might not be so large after all.

7. Two blocks

7. Two blocks

a_x consists of two w-bit blocks of random bits, aligned to grid of width w.

Theorem ([DW19a], simplified.)
Let $w = 4 \log n$. Then $m = n + O(\log n)$ is sufficient to guarantee that A_S has full row rank with probability $1 - n^{-\delta}$ for some $\delta > 0$.

QUERY is fast: Just access two blocks of width w in Z. Time on RAM: $O(wr/\log n)$.

For BUILD use Gauss elimination ($O(n^3)$ (amenable to speed-up tricks like the Four Russians algorithm, word parallelism, etc.) or Wiedemann's algorithm with a running time of $O(n^2 \log n)$.

Sharding gives a tradeoff between construction time and space overhead.

"Sweet line": Construction time $O(nC)$ and additive space overhead $\Theta(\frac{n \log n}{C})$, for shard size $n^\theta \leq C \leq n$.

$m = n + O(\log n)$, tiniest overhead

8. One block: Sorted solving

8. One block: Sorted solving

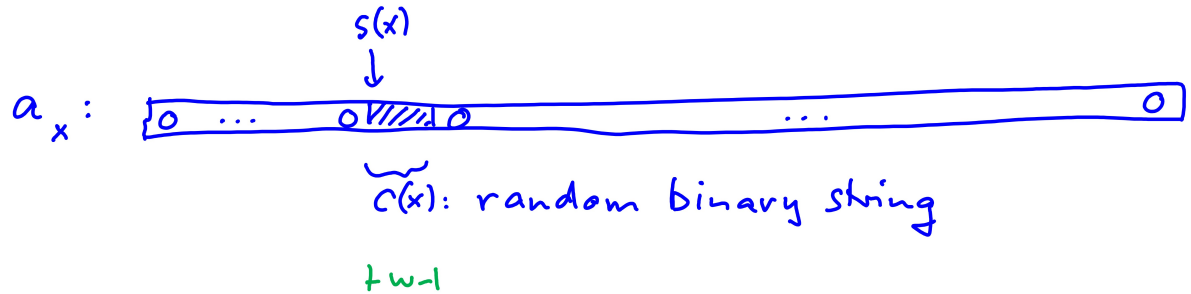
Row a_x is described by the binary string

$$0^{s(x)}c(x)0^{m-s(x)-w},$$

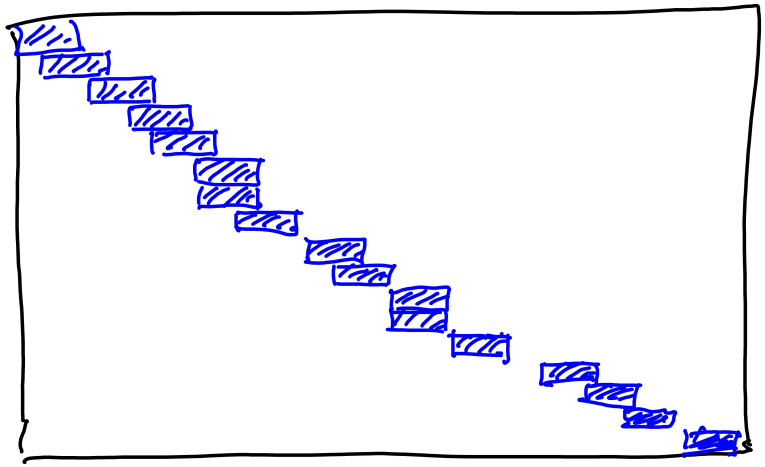
where $s(x) \in [0..m-w]$ is random and $c(x) \in \{0,1\}^w$ is random.

Theorem [DW19c]
With $w = O((\log n)/\varepsilon)$ sufficiently large the one-block construction leads to a retrieval data structure with space overhead ε, construction time $O(n/\varepsilon^2)$, and query time $O(r/\varepsilon)$, with a query costing one random memory access. The construction succeeds with high probability.

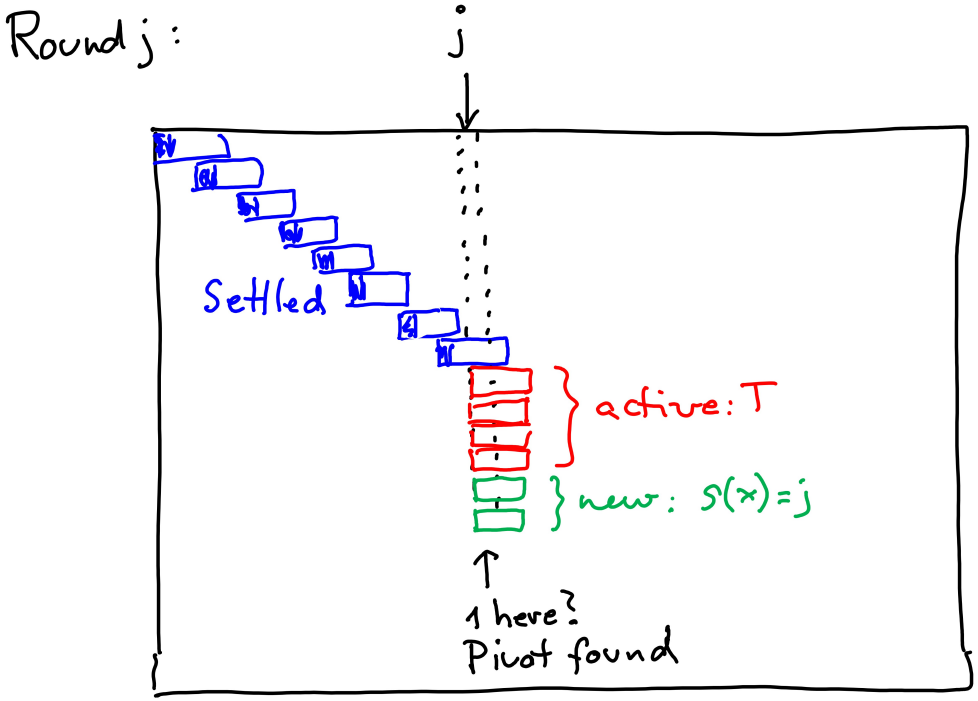
Query time is easy: Look up $w = O((\log n)/\varepsilon)$ bit vectors of length r and XOR them. At first glance this gives time $O((\log n)r/\varepsilon)$. Improvement: Store $Z[0..m-1]$ locally column-wise, use bitwise XOR on words of length $\log n$.



A_S, rows a_x sorted by $s(x)$:



approximate band matrix


Round j :
j
Settled
active: T
new: S(x)=j
1 here?
Pivot found

8. One block: Sorted solving

BUILD: Gaussian elimination "from left to right".

8. One block: Sorted solving

BUILD: Gaussian elimination "from left to right".

Walk through columns $j = 0, \ldots, m-1$.
Always have set $T \subseteq S$ of "active" keys, initially $T = \emptyset$.

8. One block: Sorted solving

BUILD: Gaussian elimination "from left to right".

Walk through columns $j = 0, \ldots, m-1$.
Always have set $T \subseteq S$ of "active" keys, initially $T = \emptyset$.

Each active key x has a "current row" (a'_x, f'_x) in which all entries outside the block $[j..j+w-1]$ are 0.

8. One block: Sorted solving

BUILD: Gaussian elimination "from left to right".

Walk through columns $j = 0, \ldots, m-1$.
Always have set $T \subseteq S$ of "active" keys, initially $T = \emptyset$.

Each active key x has a "current row" (a'_x, f'_x) in which all entries outside the block $[j..j+w-1]$ are 0.

Round j: All x with $s(x) = j$ are added to T, with $(a'_x, f'_x) = (a_x, f(x))$ (extended row).

8. One block: Sorted solving

BUILD: Gaussian elimination "from left to right".

Walk through columns $j = 0, \ldots, m-1$.
Always have set $T \subseteq S$ of "active" keys, initially $T = \emptyset$.

Each active key x has a "current row" (a'_x, f'_x) in which all entries outside the block $[j..j+w-1]$ are 0.

Round j: All x with $s(x) = j$ are added to T, with $(a'_x, f'_x) = (a_x, f(x))$ (extended row).

If there is an active x so that a'_x has a 1 in position j, then choose the first (according to $s(x)$ and some order in S) such x, set $p(x) \leftarrow j$ (we pivot on row x and column j), and declare x to be finished.

For all other active y for which a'_y has a 1 in position j, add (i.e., XOR) (a'_x, f'_x) onto (a'_y, f'_y).

8. One block: Sorted solving

BUILD: Gaussian elimination "from left to right".

Walk through columns $j = 0, \ldots, m-1$.
Always have set $T \subseteq S$ of "active" keys, initially $T = \emptyset$.

Each active key x has a "current row" (a'_x, f'_x) in which all entries outside the block $[j..j+w-1]$ are 0.

Round j: All x with $s(x) = j$ are added to T, with $(a'_x, f'_x) = (a_x, f(x))$ (extended row).

If there is an active x so that a'_x has a 1 in position j, then choose the first (according to $s(x)$ and some order in S) such x, set $p(x) \leftarrow j$ (we pivot on row x and column j), and declare x to be finished.

For all other active y for which a'_y has a 1 in position j, add (i.e., XOR) (a'_x, f'_x) onto (a'_y, f'_y).

Finally: Back substitution with $(f'_x)_{x \in S}$ to find $Z[0..m-1]$.

8. One block: Sorted solving

We want to argue that w.h.p. we have

- $|T| = O((\log n)/\varepsilon)$ always, and

8. One block: Sorted solving

We want to argue that w.h.p. we have

- $|T| = O((\log n)/\varepsilon)$ always, and
- procedure ends with $T = \emptyset$ for some $j \in [m - w..m - 1]$

8. One block: Sorted solving

$H_j := |T|$ at the end of round j.

8. One block: Sorted solving

$H_j := |T|$ at the end of round j.

Increase(round j): $B_j = |\{x \in S \mid s(x) = j\}|$, approx. $\textsc{Poisson}(1/(1+\varepsilon))$-distrib.

8. One block: Sorted solving

$H_j := |T|$ at the end of round j.

Increase(round j): $B_j = |\{x \in S \mid s(x) = j\}|$, approx. $\textsc{Poisson}(1/(1+\varepsilon))$-distrib.

Decrease(round j):
$\Pr(\text{not all } (a'_x)_j,\ x \in T, \text{ are } 0) = \Pr(\textsc{Geom}(\frac{1}{2}) \le H_{j-1} + B_j) = 1 - 2^{H_{j-1}+B_j}$.

(The relevant bits in position j in T are random – as long as keys don't stay in T too long.)

8. One block: Sorted solving

$H_j := |T|$ at the end of round j.

Increase(round j): $B_j = |\{x \in S \mid s(x) = j\}|$, approx. $\textsc{Poisson}(1/(1+\varepsilon))$-distrib.

Decrease(round j):
$\Pr(\text{not all } (a'_x)_j,\ x \in T, \text{ are } 0) = \Pr(\textsc{Geom}(\frac{1}{2}) \le H_{j-1} + B_j) = 1 - 2^{H_{j-1}+B_j}$.

(The relevant bits in position j in T are random – as long as keys don't stay in T too long.)

As soon as $|T| > \log(1/\varepsilon) + 2$, we have an overall negative drift.

The rest is queuing theory
((M/M/1)-queue with arrival rate $1 - \varepsilon$, service rate $\approx 1 - \varepsilon/2$.)

8. One block: Sorted solving

$H_j := |T|$ at the end of round j.

Increase(round j): $B_j = |\{x \in S \mid s(x) = j\}|$, approx. $\textsc{Poisson}(1/(1+\varepsilon))$-distrib.

Decrease(round j):
$\Pr(\text{not all } (a'_x)_j,\ x \in T, \text{ are } 0) = \Pr(\textsc{Geom}(\frac{1}{2}) \le H_{j-1} + B_j) = 1 - 2^{H_{j-1}+B_j}$.

(The relevant bits in position j in T are random – as long as keys don't stay in T too long.)

As soon as $|T| > \log(1/\varepsilon) + 2$, we have an overall negative drift.

The rest is queuing theory
((M/M/1)-queue with arrival rate $1 - \varepsilon$, service rate $\approx 1 - \varepsilon/2$.)

Cumulative queue length is $O(n/\varepsilon)$, maximum queue length is $O((\log n)/\varepsilon)$ w.h.p.

Cumulative queue length = total number of vector additions.

Each vector addition costs $O(1/\varepsilon)$ word operations.

9. One block: Ribbon

[DHSW22] Sorted solving requires sorting by $s(x)$ first, so we start from an an approximate band matrix ("ribbon"). Curious: This is irrelevant.

Assume key-value pairs $(x, f(x))$, $x \in S$ arrive in some order:

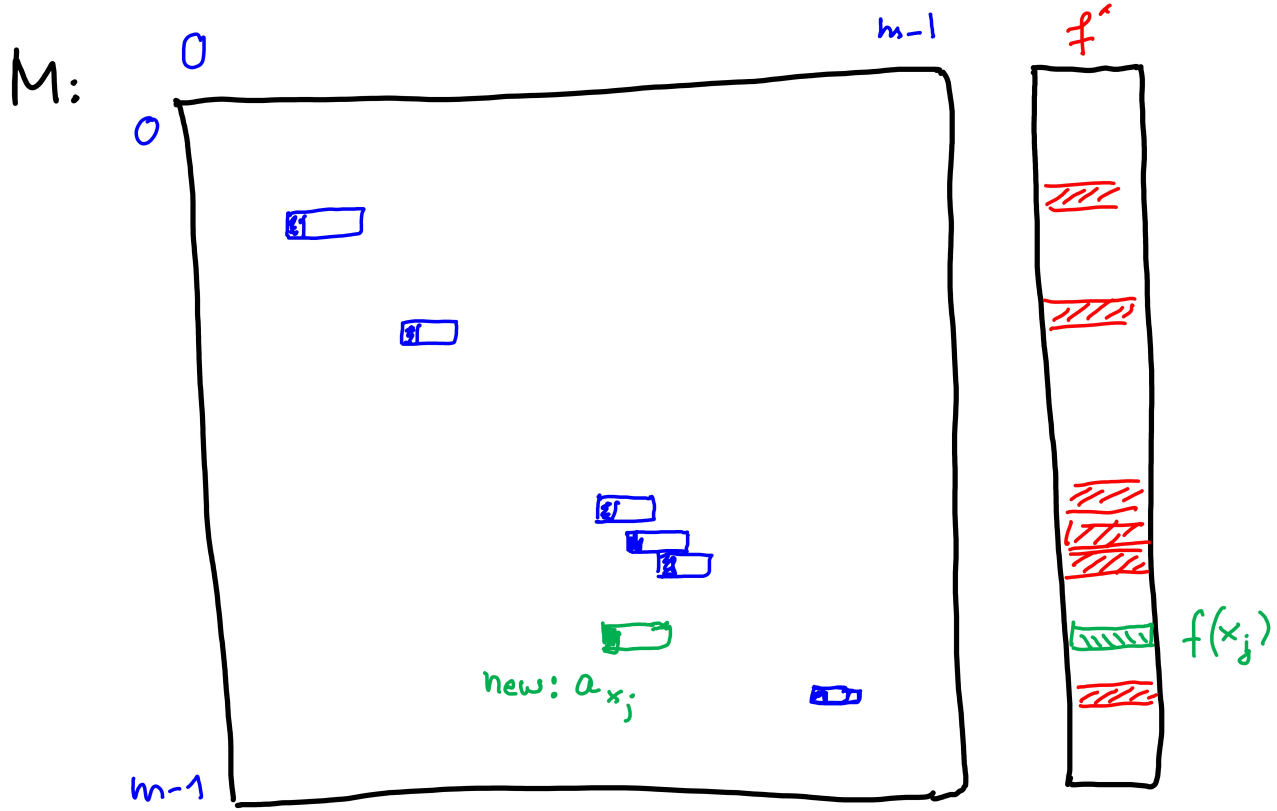
$$(x_1, f(x_1)), \ldots, (x_n, f(x_n)).$$

Build an $m \times m$ *echelon* matrix M "on the fly" with a right hand side $F[0..m-1]$, incrementally.

After round j, matrix $M \cdot z = F[0..m-1]$ is equivalent to the system $(a_{x_i} \cdot z) = f(x_i)$, $i = 1, \ldots, j$.

Since M is in echelon form, can find solution Z for $M \cdot Z = F$ by back substitution.

Random **I**ncremental **B**inary**B**anding**On** the Fly.


M:
0
0
m-1
m-1
f*
f(x_j)
new: a_{x_j}

9. One block: Ribbon

Initialization: M is the zero matrix, $F[0..m-1]$ is the zero vector (entries from $\{0,1\}^r$). We have rounds $j = 1, \ldots, n$.

9. One block: Ribbon

Initialization: M is the zero matrix, $F[0..m-1]$ is the zero vector (entries from $\{0,1\}^r$). We have rounds $j = 1, \ldots, n$.

Round j: Pair $(x_j, f(x_j))$ arrives.

9. One block: Ribbon

Initialization: M is the zero matrix, $F[0..m-1]$ is the zero vector (entries from $\{0,1\}^r$). We have rounds $j = 1, \ldots, n$.

Round j: Pair $(x_j, f(x_j))$ arrives.

$a' := a_{x_j}$, $f' := f(x_j)$, done := *false*. // New equation: $a' \cdot z = f'$.

9. One block: Ribbon

Initialization: M is the zero matrix, $F[0..m-1]$ is the zero vector (entries from $\{0,1\}^r$). We have rounds $j = 1, \ldots, n$.

Round j: Pair $(x_j, f(x_j))$ arrives.

$a' := a_{x_j}$, $f' := f(x_j)$, done := *false*. // New equation: $a' \cdot z = f'$.

while not done **and** there is s such that $(a')_s = 1$:

9. One block: Ribbon

Initialization: M is the zero matrix, $F[0..m-1]$ is the zero vector (entries from $\{0,1\}^r$). We have rounds $j = 1, \ldots, n$.

Round j: Pair $(x_j, f(x_j))$ arrives.

$a' := a_{x_j}$, $f' := f(x_j)$, done := *false*. // New equation: $a' \cdot z = f'$.

while not done **and** there is s such that $(a')_s = 1$:

pick smallest such s

if row s of M is $b \neq 0$, **then** $(a', f') := (a' \oplus b, f' \oplus F[s])$

// row transformation, moves first 1 in a' to the right

9. One block: Ribbon

Initialization: M is the zero matrix, $F[0..m-1]$ is the zero vector (entries from $\{0,1\}^r$). We have rounds $j = 1, \ldots, n$.

Round j: Pair $(x_j, f(x_j))$ arrives.

$a' := a_{x_j}$, $f' := f(x_j)$, done := *false*. // New equation: $a' \cdot z = f'$.

while not done **and** there is s such that $(a')_s = 1$:

pick smallest such s

if row s of M is $b \neq 0$, **then** $(a', f') := (a' \oplus b, f' \oplus F[s])$

// row transformation, moves first 1 in a' to the right

else // row s of M is zero

enter (a', F) as row s in the system (M, F); done := *true*

9. One block: Ribbon

Initialization: M is the zero matrix, $F[0..m-1]$ is the zero vector (entries from $\{0,1\}^r$). We have rounds $j = 1, \ldots, n$.

Round j: Pair $(x_j, f(x_j))$ arrives.

$a' := a_{x_j}$, $f' := f(x_j)$, done := *false*. // New equation: $a' \cdot z = f'$.

while not done **and** there is s such that $(a')_s = 1$:
  pick smallest such s
  if row s of M is $b \neq 0$, **then** $(a', f') := (a' \oplus b, f' \oplus F[s])$
  // row transformation, moves first 1 in a' to the right
  else // row s of M is zero
    enter (a', F) as row s in the system (M, F); done := *true*
if not done **then return** "dependence at j".

After finishing all rounds: **return** (M, F).

9. One block: Ribbon

What? This is all?

9. One block: Ribbon

What? This is all? – Observations:

- Only row transformations $\Rightarrow$ $M \cdot z = F$ is equivalent to $(a_{x_i} \cdot z) = f(x_i)$, $i = 1, \ldots, n$.

9. One block: Ribbon

What? This is all? – Observations:

- Only row transformations $\Rightarrow$ $M \cdot z = F$ is equivalent to $(a_{x_i} \cdot z) = f(x_i)$, $i = 1, \ldots, n$.
- Get "dependence at j" $\Leftrightarrow$ j is minimal with $x_1, \ldots, x_j$ is linearly dependent.

9. One block: Ribbon

What? This is all? – Observations:

- Only row transformations $\Rightarrow$ $M \cdot z = F$ is equivalent to $(a_{x_i} \cdot z) = f(x_i)$, $i = 1, \ldots, n$.
- Get "dependence at j" $\Leftrightarrow$ j is minimal with $x_1, \ldots, x_j$ is linearly dependent.
- The overall cost is the same as for sorted solving (not too hard to see).

9. One block: Ribbon

What? This is all? – Observations:

- Only row transformations $\Rightarrow$ $M \cdot z = F$ is equivalent to $(a_{x_i} \cdot z) = f(x_i)$, $i = 1, \ldots, n$.
- Get "dependence at j" $\Leftrightarrow$ j is minimal with $x_1, \ldots, x_j$ is linearly dependent.
- The overall cost is the same as for sorted solving (not too hard to see).
- The solutions are the same, if we set the nonpivot entries $Z[s]$ to zero.

9. One block: Ribbon

What? This is all? – Observations:

- Only row transformations $\Rightarrow M \cdot z = F$ is equivalent to $(a_{x_i} \cdot z) = f(x_i)$, $i = 1, \ldots, n$.
- Get "dependence at j" $\Leftrightarrow j$ is minimal with $x_1, \ldots, x_j$ is linearly dependent.
- The overall cost is the same as for sorted solving (not too hard to see).
- The solutions are the same, if we set the nonpivot entries $Z[s]$ to zero.
- (Unfortunately:) This is not really an online or incremental algorithm for retrieval, because of back-substitution at the end.

9. One block: Ribbon

What? This is all? – Observations:

- Only row transformations $\Rightarrow$ $M \cdot z = F$ is equivalent to $(a_{x_i} \cdot z) = f(x_i)$, $i = 1, \ldots, n$.
- Get "dependence at j" $\Leftrightarrow$ j is minimal with $x_1, \ldots, x_j$ is linearly dependent.
- The overall cost is the same as for sorted solving (not too hard to see).
- The solutions are the same, if we set the nonpivot entries $Z[s]$ to zero.
- (Unfortunately:) This is not really an online or incremental algorithm for retrieval, because of back-substitution at the end.
- Backtracking: It is easy to *undo* insertion of the last entry $(x_j, f(x_j))$ into (M, F): just zero out the last row that was added. (Can be iterated.)

10. Bumping

10. Bumping

[DHSW21, DHSW22]

Following an algorithm engineering trail leads to a new theoretical result.

Old technique: **"Bumping"** keys: Kick out keys that do not fit, treat elsewhere (called "backyard" in [ANS10]). Price to pay: Extra access into memory.

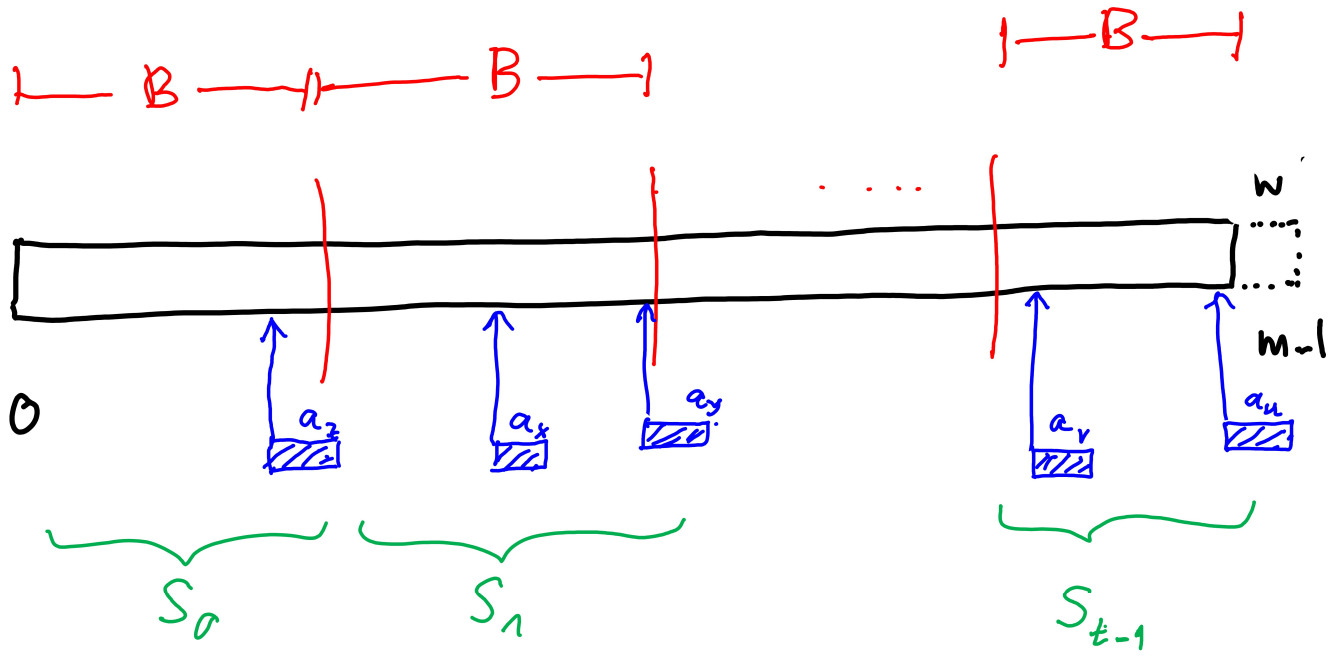
Start with a version of *sharding*: Keys are split into **buckets** using $s(x)$, the starting position in a_x.

Subdivide the range $[m - w]$ of into segments of length B.

Bucket S_u: set of keys whose starting positions $s(x)$ fall into segment u.

Buckets are treated in increasing order of segments, left to right.

Unconventional: **No gaps** between segments. So for x in S_u vector a_x may have nonzero bits in segment $u + 1$. Keys from bucket S_u will mainly placed in positions from segment u in M, but there may be a some overspill into segment $u + 1$.


B
B
B
w
m-1
0
a_z
a_x
a_y
a_v
a_u
S_0
S_1
S_{t-1}

10. Bumping

Some keys may be "bumped", i.e., taken out of the system $M \cdot z = F$ of equations.

10. Bumping

Some keys may be "bumped", i.e., taken out of the system $M \cdot z = F$ of equations.

First idea: Bump keys whose a_x ruins linear independence.

10. Bumping

Some keys may be "bumped", i.e., taken out of the system $M \cdot z = F$ of equations.

First idea: Bump keys whose a_x ruins linear independence.

Good: The truly minimum number of keys. Bad: Must store them in a dictionary manner, at cost of

$$\approx \beta \log |\mathcal{U}| \text{ bits, for } \beta = n - \dim(\text{span}\{a_x \mid x \in S\}), \text{ the deficiency,}$$

causing significant overhead.

10. Bumping

Some keys may be "bumped", i.e., taken out of the system $M \cdot z = F$ of equations.

First idea: Bump keys whose a_x ruins linear independence.

Good: The truly minimum number of keys. Bad: Must store them in a dictionary manner, at cost of

$$\approx \beta \log |\mathcal{U}| \text{ bits, for } \beta = n - \dim(\text{span}\{a_x \mid x \in S\}), \text{ the deficiency,}$$

causing significant overhead.

Next idea: Bump whole bucket as soon as there is a linear dependency in it.

10. Bumping

Some keys may be "bumped", i.e., taken out of the system $M \cdot z = F$ of equations.

First idea: Bump keys whose a_x ruins linear independence.

Good: The truly minimum number of keys. Bad: Must store them in a dictionary manner, at cost of

$$\approx \beta \log |\mathcal{U}| \text{ bits, for } \beta = n - \dim(\text{span}\{a_x \mid x \in S\}), \text{ the deficiency,}$$

causing significant overhead.

Next idea: Bump whole bucket as soon as there is a linear dependency in it.

Gives small overhead (1 Bit/bucket) but bumps many keys.

10. Bumping

Some keys may be "bumped", i.e., taken out of the system $M \cdot z = F$ of equations.

First idea: Bump keys whose a_x ruins linear independence.

Good: The truly minimum number of keys. Bad: Must store them in a dictionary manner, at cost of

$$\approx \beta \log |\mathcal{U}| \text{ bits, for } \beta = n - \dim(\text{span}\{a_x \mid x \in S\}), \text{ the deficiency,}$$

causing significant overhead.

Next idea: Bump whole bucket as soon as there is a linear dependency in it.

Gives small overhead (1 Bit/bucket) but bumps many keys.

Compromise: Only three options: Bump **nothing**, bump all keys with **smallish** $s(x)$ in segment u, or bump the **whole bucket**. – Overhead: $\log_2 3$ bits/bucket.

10. Bumping, batch bumping

Some parameters:

w is the block length in a_x, a parameter we will play around with.

$B = \frac{w^2}{\log w}$, almost squarish in w, is the segment length.

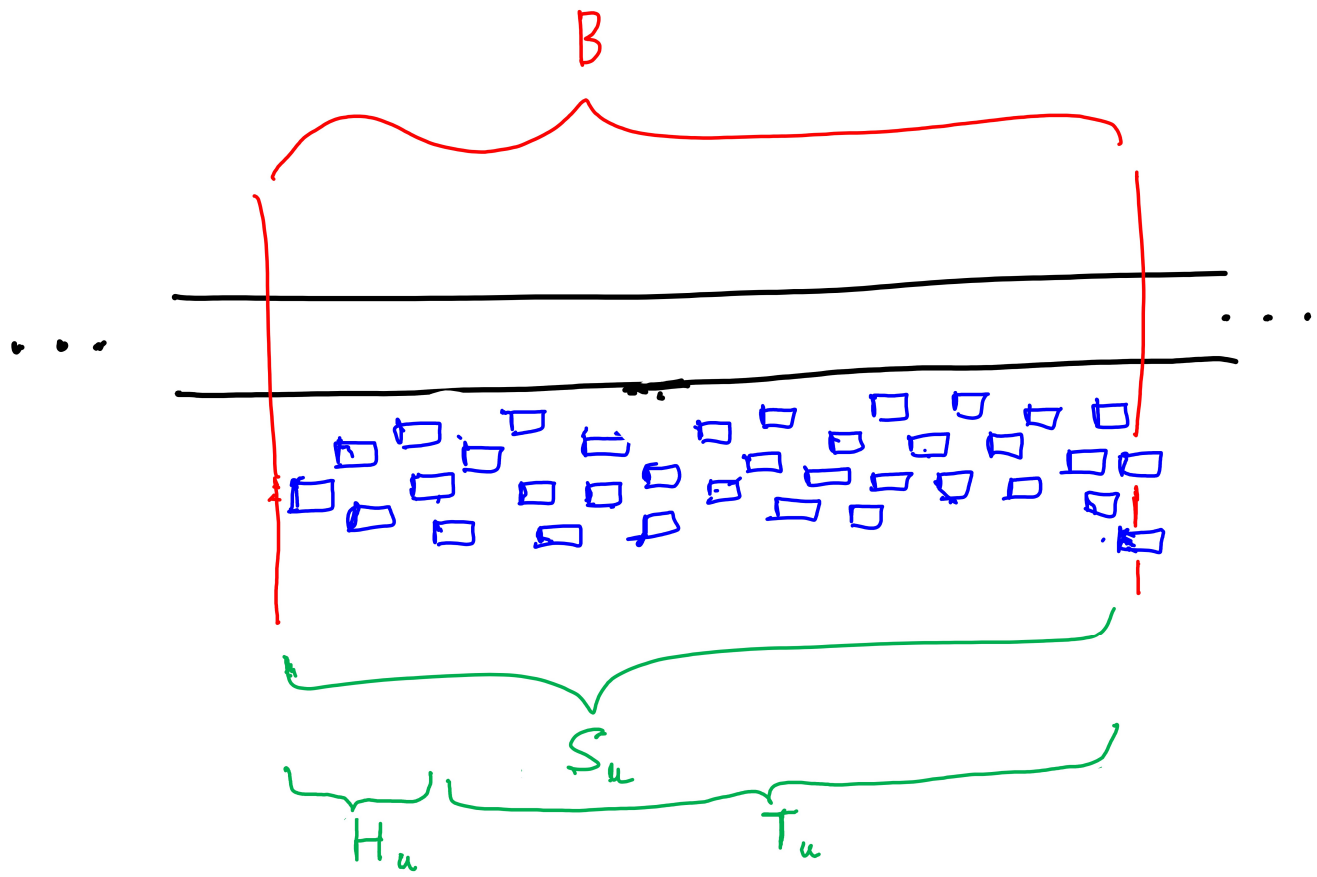
$S_u = \{x \in S \mid s(x) \text{ is in segment } u\}$, for $u = 0, \ldots, (m-w)/B - 1$.

$H_u = \{x \in S \mid s(x) \text{ is among the smallest } 3w/8 \text{ values in segment } u\}$, ("head")

$T_u = S_u - H_u$ (keys with larger s-values, "tail")

Options for bucket u: Bump nothing, bump keys in H_u, bump all of S_u.

All bumped keys are treated in a "secondary" data structure, which could be of the same type again (recursion for a constant number of levels), or use some other tricks of the trade. We do not worry about them.


B
S_u
H_u
T_u

10. Bumping, batch bumping

algorithm BumpedRibbonRetrieval (BuRR)

Build square matrix M and right hand side (f'_x), treating one S_u after the other.

10. Bumping, batch bumping

algorithm BumpedRibbonRetrieval (BuRR)

Build square matrix M and right hand side (f'_x), treating one S_u after the other.

For keys x in S_u:

10. Bumping, batch bumping

algorithm BumpedRibbonRetrieval (BuRR)

Build square matrix M and right hand side (f'_x), treating one S_u after the other.

For keys x in S_u:

In M consider rows and columns in segment u.
Some of them may already be occupied by keys from segment $u-1$ ("overspill"), we expect a smallish fraction of w many.

10. Bumping, batch bumping

algorithm BUMPEDRIBBONRETRIEVAL (BURR)

Build square matrix M and right hand side (f'_x), treating one S_u after the other.

For keys x in S_u:

In M consider rows and columns in segment u.
Some of them may already be occupied by keys from segment $u-1$ ("overspill"), we expect a smallish fraction of w many.

(1) Try to insert all keys from T_u. // No conflict with overspill!

10. Bumping, batch bumping

algorithm BumpedRibbonRetrieval (BuRR)

Build square matrix M and right hand side (f'_x), treating one S_u after the other.

For keys x in S_u:

In M consider rows and columns in segment u.
Some of them may already be occupied by keys from segment $u-1$ ("overspill"), we expect a smallish fraction of w many.

(1) Try to insert all keys from T_u. // No conflict with overspill!
if this fails: bump all of S_u. // Helpful: Can easily undo last changes to M

10. Bumping, batch bumping

algorithm BUMPEDRIBBONRETRIEVAL (BURR)

Build square matrix M and right hand side (f'_x), treating one S_u after the other.

For keys x in S_u:

In M consider rows and columns in segment u.
Some of them may already be occupied by keys from segment $u-1$ ("overspill"), we expect a smallish fraction of w many.

(1) Try to insert all keys from T_u. // No conflict with overspill!
if this fails: bump all of S_u. // Helpful: Can easily undo last changes to M

(2) Try to insert all keys from H_u.
if this fails: bump all of H_u.

10. Bumping, batch bumping

algorithm BUMPEDRIBBONRETRIEVAL (BURR)

Build square matrix M and right hand side (f'_x), treating one S_u after the other.

For keys x in S_u:

In M consider rows and columns in segment u.
Some of them may already be occupied by keys from segment $u-1$ ("overspill"), we expect a smallish fraction of w many.

(1) Try to insert all keys from T_u. // No conflict with overspill!
if this fails: bump all of S_u. // Helpful: Can easily undo last changes to M

(2) Try to insert all keys from H_u.
if this fails: bump all of H_u.

If both (1) and (2) are successful, nothing is bumped from S_u.

10. Bumping, batch bumping

algorithm BUMPEDRIBBONRETRIEVAL (BURR)

Build square matrix M and right hand side (f'_x), treating one S_u after the other.

For keys x in S_u:

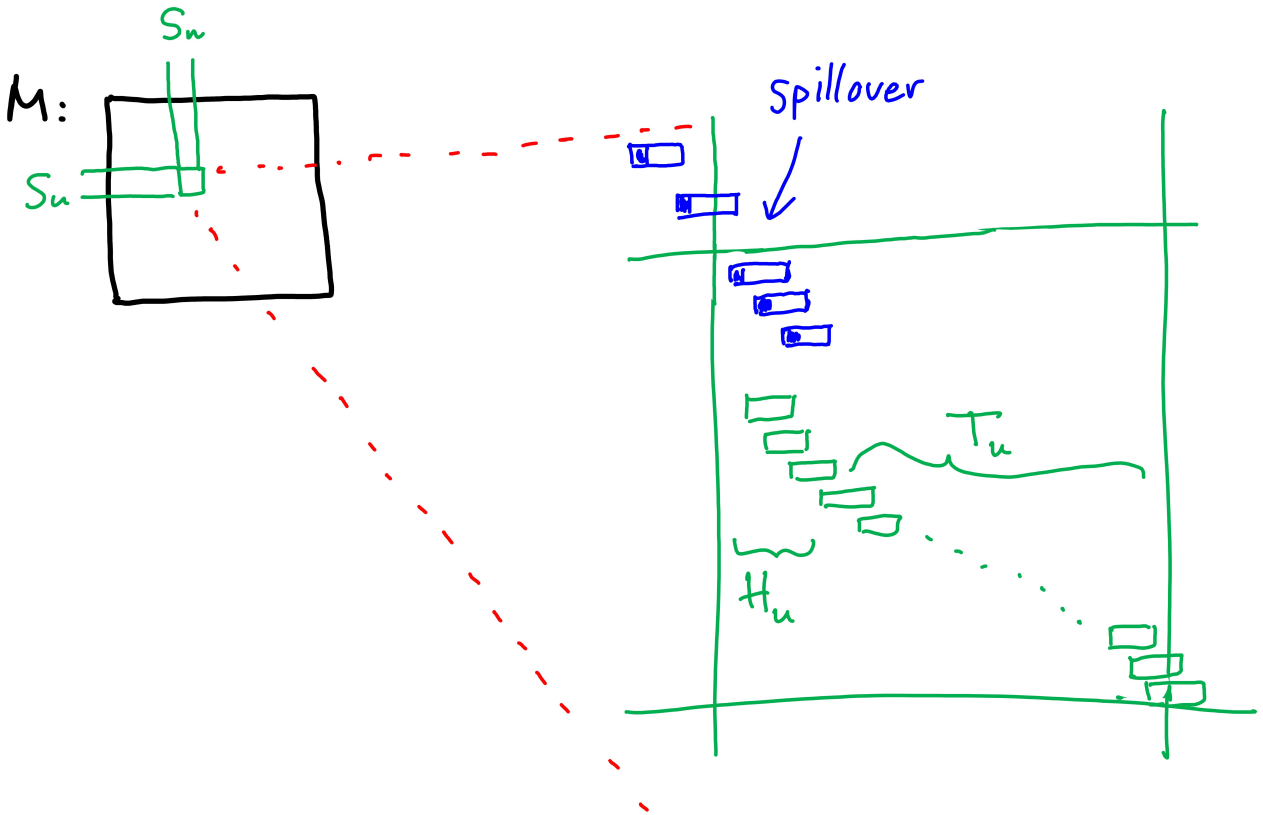
In M consider rows and columns in segment u.
Some of them may already be occupied by keys from segment $u-1$ ("overspill"), we expect a smallish fraction of w many.

(1) Try to insert all keys from T_u. // No conflict with overspill!
if this fails: bump all of S_u. // Helpful: Can easily undo last changes to M

(2) Try to insert all keys from H_u.
if this fails: bump all of H_u.

If both (1) and (2) are successful, nothing is bumped from S_u.

Finally: Back substitution for all keys *that are not bumped*.


M:
S_u
S_u
Spillover
T_u
H_u

10. Bumping, batch bumping

Bumping information for each bucket S_u is part of the data structure, giving overhead of

$$\frac{m}{B} \cdot \log_2 3 \text{ bits.}$$

$\textsc{query}(x)$:

$s(x)$ combined with bumping information of its bucket tells us if x is bumped or not.

Accordingly, get answer from Z or from the backyard data structure.

10. Bumping, batch bumping, **overloading**

Parameters to play around with: w and $\varepsilon = \frac{m}{n} - 1$.

10. Bumping, batch bumping, overloading

Parameters to play around with: w and $\varepsilon = \frac{m}{n} - 1$.

Last engineering twist: **Overloading**.

10. Bumping, batch bumping, overloading

Parameters to play around with: w and $\varepsilon = \frac{m}{n} - 1$.

Last engineering twist: **Overloading**.

Background: *Experiments showed* that for $m = n(1+\varepsilon)$ no exceptional behavior appeared for smaller and smaller ε, even $\varepsilon = 0$ was o.k.

10. Bumping, batch bumping, overloading

Parameters to play around with: w and $\varepsilon = \frac{m}{n} - 1$.

Last engineering twist: **Overloading**.

Background: *Experiments showed* that for $m = n(1+\varepsilon)$ no exceptional behavior appeared for smaller and smaller ε, even $\varepsilon = 0$ was o.k.

Explanation: Bumping defuses situations where single buckets overflow.

10. Bumping, batch bumping, overloading

Parameters to play around with: w and $\varepsilon = \frac{m}{n} - 1$.

Last engineering twist: **Overloading**.

Background: *Experiments showed* that for $m = n(1+\varepsilon)$ no exceptional behavior appeared for smaller and smaller ε, even $\varepsilon = 0$ was o.k.

Explanation: Bumping defuses situations where single buckets overflow.

Even "negative ε" worked: m is chosen a little smaller than n.

10. Bumping, batch bumping, overloading

Parameters to play around with: w and $\varepsilon = \frac{m}{n} - 1$.

Last engineering twist: **Overloading**.

Background: *Experiments showed* that for $m = n(1+\varepsilon)$ no exceptional behavior appeared for smaller and smaller ε, even $\varepsilon = 0$ was o.k.

Explanation: Bumping defuses situations where single buckets overflow.

Even "negative ε" worked: m is chosen a little smaller than n.

Some keys are bumped anyway. If these are $\Theta(n/w)$ many, say, we could just as well throw in $(1+1/w)m$ keys, this does not change the "pressure into the backyard" much.

10. Bumping, batch bumping, overloading

Parameters to play around with: w and $\varepsilon = \frac{m}{n} - 1$.

Last engineering twist: **Overloading**.

Background: *Experiments showed* that for $m = n(1+\varepsilon)$ no exceptional behavior appeared for smaller and smaller ε, even $\varepsilon = 0$ was o.k.

Explanation: Bumping defuses situations where single buckets overflow.

Even "negative ε" worked: m is chosen a little smaller than n.

Some keys are bumped anyway. If these are $\Theta(n/w)$ many, say, we could just as well throw in $(1+1/w)m$ keys, this does not change the "pressure into the backyard" much.

Effect: In M, rows/col's in buckets tend to be used completely, **no gaps!**

10. Bumping, batch bumping, overloading

Theorem [DHSW21, DHSW22]

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

10. Bumping, batch bumping, overloading

Theorem [DHSW21, DHSW22]

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

w can be $o(\log n)$! E.g., $w = 64$ is an interesting choice.

10. Bumping, batch bumping, overloading

Theorem [DHSW21, DHSW22]

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

w can be $o(\log n)$! E.g., $w = 64$ is an interesting choice.

Very promising experimental results [DHSW22].

All details in [DDHSSW2?], forthcoming.

10. Bumping, batch bumping, overloading

Theorem [DHSW21, DHSW22]

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

w can be $o(\log n)$! E.g., $w = 64$ is an interesting choice.

Very promising experimental results [DHSW22].

All details in [DDHSSW2?], forthcoming.

Also illuminating and accessible, for the topic of this talk, and more: [Wal23] by S. Walzer (Bull. EATCS).

Omitted

Retrieval can be used to build . . .

- . . . small perfect hashing data structures [BPZ13].
- . . . small (static) filters (observed in [DP08]).
- . . . data structures for simulating fully random hash functions [DR09].

Conclusion

Retrieval by equations:

One simple concept – many variants – multiple methods and insights, including constructions interesting for practical use.

- Give more precise bounds/thresholds for sorted solving.
- Explore other uses for overloading (whenever there is a backyard structure . . .).

Conclusion

Retrieval by equations:

One simple concept – many variants – multiple methods and insights, including constructions interesting for practical use.

Problems:

- Are equations ever useful in a dynamic setting?
- Give more precise bounds/thresholds for sorted solving.
- Explore other uses for overloading (whenever there is a backyard structure . . .).

Thank you.

A. Orientability + Retrieval $\rightarrow$ Perfect Hashing

[CKRT04, BPZ13]

Choose $r > c_k n$ and B_x for $x \in S$; can assume orientability of $(B_x)_{x \in S}$.

A. Orientability + Retrieval $\rightarrow$ Perfect Hashing

[CKRT04, BPZ13]

Choose $r > c_k n$ and B_x for $x \in S$; can assume orientability of $(B_x)_{x \in S}$.

Find $\tau \colon S \stackrel{1-1}{\rightarrow} [t]$ such that $\tau(x) \in B_x$ for $x \in S$.

(This is a matching problem. Algorithms that work in linear time w.h.p. are known [KA19].)

A. Orientability + Retrieval $\rightarrow$ Perfect Hashing

[CKRT04, BPZ13]

Choose $r > c_k n$ and B_x for $x \in S$; can assume orientability of $(B_x)_{x \in S}$.

Find $\tau\colon S \stackrel{1-1}{\rightarrow} [t]$ such that $\tau(x) \in B_x$ for $x \in S$.

(This is a matching problem. Algorithms that work in linear time w.h.p. are known [KA19].)

With $B_x = \{h_0(x), \ldots, h_{k-1}(x)\}$ we then have $\tau(x) = h_{\sigma(x)}(x)$ for some $\sigma\colon S \rightarrow [k]$.

A. Orientability + Retrieval $\to$ Perfect Hashing

[CKRT04, BPZ13]

Choose $r > c_k n$ and B_x for $x \in S$; can assume orientability of $(B_x)_{x \in S}$.

Find $\tau\colon S \overset{1-1}{\to} [t]$ such that $\tau(x) \in B_x$ for $x \in S$.

(This is a matching problem. Algorithms that work in linear time w.h.p. are known [KA19].)

With $B_x = \{h_0(x), \ldots, h_{k-1}(x)\}$ we then have $\tau(x) = h_{\sigma(x)}(x)$ for some $\sigma\colon S \to [k]$.

Then

$$h\colon \mathcal{U} \to [t] \text{ defined by } h(x) = h_{\sigma(x)}(x)$$

is one-to-one on S.

A. Orientability + Retrieval $\rightarrow$ Perfect Hashing

[CKRT04, BPZ13]

Choose $r > c_k n$ and B_x for $x \in S$; can assume orientability of $(B_x)_{x \in S}$.

Find $\tau\colon S \stackrel{1-1}{\rightarrow} [t]$ such that $\tau(x) \in B_x$ for $x \in S$.

(This is a matching problem. Algorithms that work in linear time w.h.p. are known [KA19].)

With $B_x = \{h_0(x), \ldots, h_{k-1}(x)\}$ we then have $\tau(x) = h_{\sigma(x)}(x)$ for some $\sigma\colon S \rightarrow [k]$.

Then

$$h\colon \mathcal{U} \rightarrow [t] \text{ defined by } h(x) = h_{\sigma(x)}(x)$$

is one-to-one on S.

h can be evaluated in time $O(k)$ by using a retrieval data structure $\mathcal{R}_\sigma$ for σ.

A. Orientability + Retrieval $\rightarrow$ Perfect Hashing

Worked out in full detail, with experimentation, in [BPZ13].

A. Orientability + Retrieval $\rightarrow$ Perfect Hashing

Worked out in full detail, with experimentation, in [BPZ13].

In the retrieval structure, use $|A_x| = 3$, can use larger k for the B_x.

Range of h is $[1.025n]$, space for $\mathcal{R}_\sigma$ is $1.23(\log_2 3)n \approx 1.95n$ [Bits].

Beware: Perfect hashing is another topic with many extra tricks, and new developments!

(Not our focus.)

B. Solvability with $n = m$

A question aside: When can we have $m = n$?

B. Solvability with $n = m$

A question aside: When can we have $m = n$?

Fact [Folklore]

If the entries of an $n \times n$ matrix M are chosen randomly from $\{0, 1\}$, the probability that M is regular is > 0.288 (but not much larger for large n).

B. Solvability with $n = m$

A question aside: When can we have $m = n$?

Fact [Folklore]

If the entries of an $n \times n$ matrix M are chosen randomly from $\{0, 1\}$, the probability that M is regular is > 0.288 (but not much larger for large n).

Fact [Coo00]

If the rows of an $n \times n$ matrix M are chosen randomly from the set of all vectors of weight $C \log n$, for C large enough, the probability that M is regular is > 0.25.

B. Solvability with $n = m$

A question aside: When can we have $m = n$?

Fact [Folklore]

If the entries of an $n \times n$ matrix M are chosen randomly from $\{0, 1\}$, the probability that M is regular is > 0.288 (but not much larger for large n).

Fact [Coo00]

If the rows of an $n \times n$ matrix M are chosen randomly from the set of all vectors of weight $C \log n$, for C large enough, the probability that M is regular is > 0.25.

For vectors of weight $o(\log n)$ the respective probability goes to 0 for $n \to \infty$.

Not good for retrieval, since suddenly QUERY needs logarithmic time.

B. Solvability with $n = m$

A question aside: When can we have $m = n$?

Fact [Folklore]

If the entries of an $n \times n$ matrix M are chosen randomly from $\{0, 1\}$, the probability that M is regular is > 0.288 (but not much larger for large n).

Fact [Coo00]

If the rows of an $n \times n$ matrix M are chosen randomly from the set of all vectors of weight $C \log n$, for C large enough, the probability that M is regular is > 0.25.

For vectors of weight $o(\log n)$ the respective probability goes to 0 for $n \to \infty$.

Not good for retrieval, since suddenly QUERY needs logarithmic time.

([Por09] used these facts in combination with (among others) a table lookup technique to obtain very good retrieval structures.)

C. More applications: Static filters, Full randomness

C. More applications: Static filters, Full randomness

A simple application of retrieval: Static filters with multiplicative overhead $\frac{m}{n} - 1$.

(Observed in [DP08].)

Let $h\colon \mathcal{U} \to \{0,1\}^r$ be a random hash function.

$\textsc{build}_{\mathsf{F}}(S)$: Build retrieval structure $\mathcal{R}_{h\restriction S}$ for $(h(x))_{x\in S}$.

F-$\textsc{query}(x)$: Evaluate $v := h(x)$, return $[v = \mathcal{R}\text{-}\textsc{query}(x)]$.

Easy to see: False positive probability is 2^{-r}, space is mr.

Lower space bound for this false positive probability is (essentially) nr.

C. More applications: Static filters, **Full randomness**

Given: S. We wish to have a data structure for a function $g\colon \mathcal{U} \to \{0,1\}^r$ that on S behaves fully randomly.

Using the random mapping $x \mapsto a_x$ as before, we initialize $Z[0..m-1]$ with random entries from $\{0,1\}^r$.

On input x, we return $\textsc{query}_Z(x)$.

(If A_S has full row rank, this is fully random on S.
Construction can be carried out without knowing S.)

Space overhead: $\frac{m}{n} - 1$ with m from retrieval structure.

Hey! We use randomness in a_x, for $x \in S$, to simulate fully random values on S?

Be assured: We can get by without any randomness "from outside",
by sharding techniques. Increases overhead. Details: [DR09].

References

[ANS10] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 787–796. IEEE Computer Society, 2010.

[BPZ13] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Inf. Syst.*, 38(1):108–131, 2013.

[CKRT04] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In J. Ian Munro, editor, *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 30–39. SIAM, 2004.

[Coo00] Colin Cooper. On the rank of random matrices. *Random Struct. Algorithms*, 16(2):209–232, 2000.

[DGM+10] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In

Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I*, volume 6198 of *Lecture Notes in Computer Science*, pages 213–225. Springer, 2010.

[DHSW21] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership using ribbon. *CoRR*, abs/2109.01892, 2021.

[DHSW22] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership using ribbon. In Christian Schulz and Bora Uçar, editors, *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany*, volume 233 of *LIPIcs*, pages 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[DM02] Olivier Dubois and Jacques Mandler. The 3-xorsat threshold. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings*, pages 769–778. IEEE Computer Society, 2002.

[DP08] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and

approximate membership (extended abstract). In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 2008.

[DR09] Martin Dietzfelbinger and Michael Rink. Applications of a splitting trick. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, and Sotiris E. Nikoletseas andDBLP:conf/icalp/DietzfelbingerR09 Wolfgang Thomas, editors, *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I*, volume 5555 of *Lecture Notes in Computer Science*, pages 354–365. Springer, 2009.

[DW19a] Martin Dietzfelbinger and Stefan Walzer. Constant-time retrieval with o(log m) extra bits. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019, March 13-16, 2019, Berlin, Germany*, volume 126 of *LIPIcs*, pages 24:1–24:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[DW19b] Martin Dietzfelbinger and Stefan Walzer. Dense peelable random uniform hypergraphs. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 38:1–38:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[DW19c] Martin Dietzfelbinger and Stefan Walzer. Efficient gauss elimination for near-quadratic matrices with one short random block per row, with applications. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, volume 144 of *LIPIcs*, pages 39:1–39:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[FM12] Alan M. Frieze and Páll Melsted. Maximum matchings in random bipartite graphs and the space utilization of cuckoo hash tables. *Random Struct. Algorithms*, 41(3):334–364, 2012.

[FP10] Nikolaos Fountoulakis and Konstantinos Panagiotou. Orientability of random hypergraphs and the power of multiple choices. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata,*

Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I, volume 6198 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2010.

[FPSS05] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005.

[GOV16] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In Andrew V. Goldberg and Alexander S. Kulikov, editors, *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, volume 9685 of *Lecture Notes in Computer Science*, pages 339–352. Springer, 2016.

[GOV20] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of ([compressed] static | minimal perfect hash) functions. *Inf. Comput.*, 273:104517, 2020.

[KA19] Megha Khosla and Avishek Anand. A faster algorithm for cuckoo insertion and bipartite matching in large graphs. *Algorithmica*, 81(9):3707–3724, 2019.

[KRU15] Shrinivas Kudekar, Thomas J. Richardson, and Rüdiger L. Urbanke. Wave-like solutions of general 1-d spatially coupled systems. *IEEE Trans. Inf. Theory*, 61(8):4117–4157, 2015.

[LMSS01] Michael Luby, Michael Mitzenmacher, Mohammad Amin Shokrollahi, and Daniel A. Spielman. Improved low-density parity-check codes using irregular graphs. *IEEE Trans. Inf. Theory*, 47(2):585–598, 2001.

[MWHC96] Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. A family of perfect hashing methods. *Comput. J.*, 39(6):547–554, 1996.

[Nav16] Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[PBC+23] Prashant Pandey, Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, Guido Tagliavini, and Rob Johnson. Iceberght: High performance hash tables through stability and low associativity. *Proc. ACM Manag. Data*, 1(1):47:1–47:26, 2023.

[Por09] Ely Porat. An optimal bloom filter replacement based on matrix solving. In Anna E. Frid, Andrey Morozov, Andrey Rybalchenko, and Klaus W. Wagner, editors, *Computer Science - Theory and Applications, Fourth International Computer Science Symposium*

in Russia, CSR 2009, Novosibirsk, Russia, August 18–23, 2009. Proceedings, volume 5675 of *Lecture Notes in Computer Science*, pages 263–273. Springer, 2009.

[PS16] Boris G. Pittel and Gregory B. Sorkin. The satisfiability threshold for k-xorsat. *Comb. Probab. Comput.*, 25(2):236–268, 2016.

[SH94] Steven S. Seiden and Daniel S. Hirschberg. Finding succinct ordered minimal perfect hash functions. *Inf. Process. Lett.*, 51(6):283–288, 1994.

[Wal21] Stefan Walzer. Peeling close to the orientability threshold - spatial coupling in hashing-based data structures. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10–13, 2021*, pages 2194–2211. SIAM, 2021.

[Wal23] Stefan Walzer. What if we tried less power? - lessons from studying the power of choices in hashing-based data structures. *CoRR*, abs/2307.00644, 2023.

[Wie86] Douglas H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trans. Inf. Theory*, 32(1):54–62, 1986.