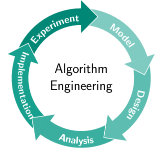
Karlsruhe Institute of Technology



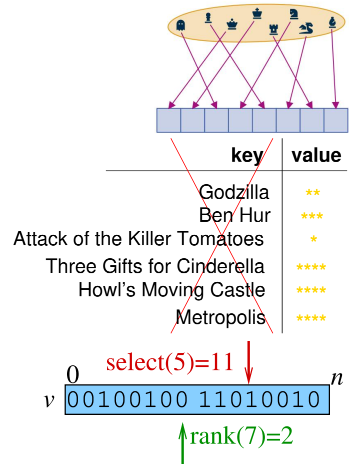
Engineering Compressed Data Structures

Peter Sanders | April 13, 2026

Oosom @en.wikipedia

Overview

- Motivation and Methodology
- Hash-table Variants
  - **Perfect Hashing Deep Dive**
  - **Static Function Retrieval**
  - Approximate Membership Query (AMQ) Filter aka Bloom F.
- **Bit vectors with rank and select**
- Summary



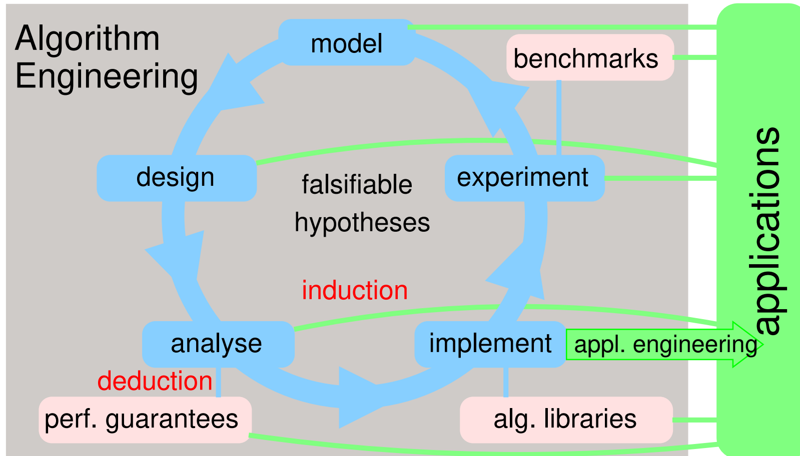
Why is this a Fastcode Talk?

Karlsruhe Institute of Technology

- Fast compressed data structures can help in

  performance engineering many applications
- Interesting case studies in algorithm engineering and performance engineering

Algorithm Engineering [San09]



Why to
Bring Theory and Practice Together?

Karlsruhe Institute of Technology

Perspective

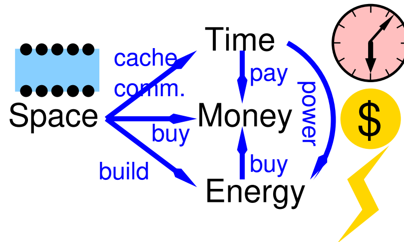
Theorist	Practitioner
you want to be relevant?	robustness
find the right models	better algorithms
new questions,	theory guides tuning
e.g., constant factors,	
smoothed analysis	

Recurring theme in this talk:
Seemingly evasive space lower bounds
turn out to be reachable and useful.

Why Compressed Data Structures

Karlsruhe Institute of Technology

Here, concretely:
Static data structures,
space ↔ construction time ↔ query time

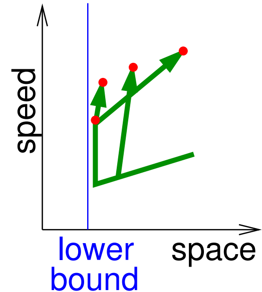


Applications:

- databases
- bioinformatics
- search engines
- brute-force search
- ...

An Emerging Design Pattern

1. Design a space-optimal data structure
2. Performance engineering
3. Relax space fixation
4. More performance engineering
5. Diversify to achieve a Pareto front with different tradeoffs



Overview

Karlsruhe Institute of Technology

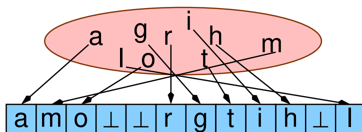
- Motivation and Methodology
- **Hash-table Variants**
  - Perfect Hashing Deep Dive
  - Static Function Retrieval
  - Approximate Membership Query (AMQ) Filter aka Bloom Filter
- Bit vectors with rank and select
- Summary

Hash-Tables and Variants

Karlsruhe Institute of Technology

Perhaps the most frequently used nontrivial data structure.

- + Rich set of constant time operations – insert, delete, isMember query, retrieve and update associated information, . . .
- – Considerable space for keys, associated information, pointers, empty cells,. . .
- – Poor locality $\rightsquigarrow$ often performance bottleneck



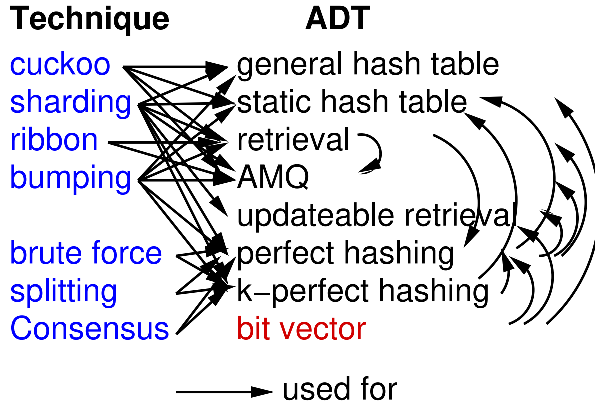
$\rightsquigarrow$ specialized data structures that are more compact (and faster)

$n \times u$ bit keys and r bits associated information

Name	operations	space lower bound
hash table	all	$u + r - \log n$
static function	retrieve associated information	r
AMQ/Bloom filter	isMember (false pos. prob. f)	$\log \frac{1}{f}$
perfect hashing	map keys to $\mathbb{Z}_n$	1.44

Hash-Tables, Variants, and Techniques

Karlsruhe Institute of Technology



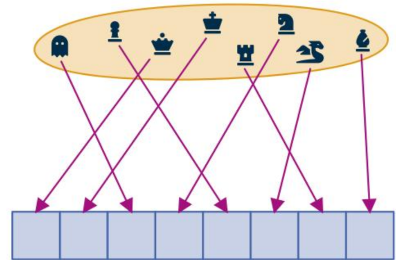
Overview

Karlsruhe Institute of Technology

- Motivation and Methodology
- Hash-table Variants
  - **Perfect Hashing Deep Dive**
  - Static Function Retrieval
  - Approximate Membership Query (AMQ) Filter aka Bloom Filter
- Bit vectors with rank and select
- Summary

Perfect Hashing

A perfect hash function h PHF) maps
$S = \{s_1, \ldots, s_n\}$ injectively to $1..m$.
minimal PHF (MPHF): $m = n$
Most work focusses on MPHFs

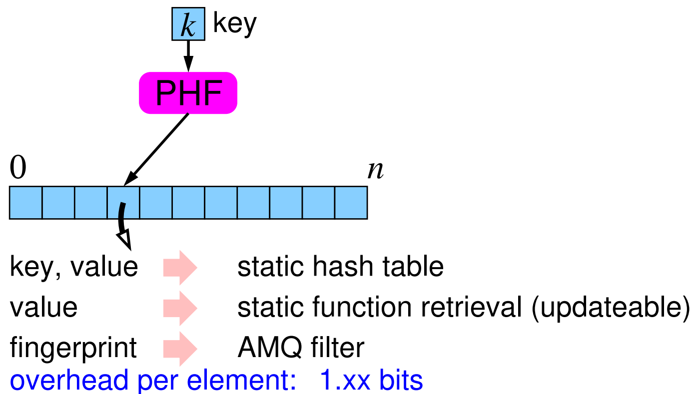


Variant: k-perfect hashing: $\forall i \in 1..m : |h^{-1}(i)| \leq k$

Information-theoretical **lower bound:** $n \log e \approx 1.44n$ bits.

Applications of Perfect Hashing

Databases, bioinformatics, static hash tables,. . .
Recent survey paper has 170 references [LMP+26].



In a columnstore database: dictionary encoding of attributes.

Brute Force Search for PHFs – Space-Optimal yet Preposterously Slow (?)

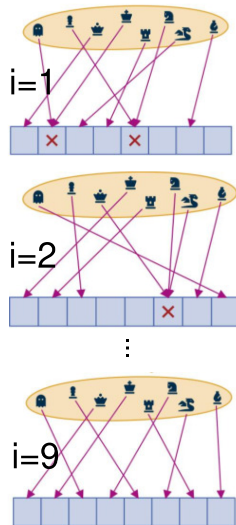
Consider a sequence $h_1, h_2, \ldots$ of random hash functions.

for $i := 1$ **to** ∞ **do** **if** $|h_i(S)| = |S|$ **then** break loop
store i (* variable bitlength encoding *)

$$p := \mathbf{P}[\text{success}] = \frac{n!}{n^n} \approx e^{-n}$$

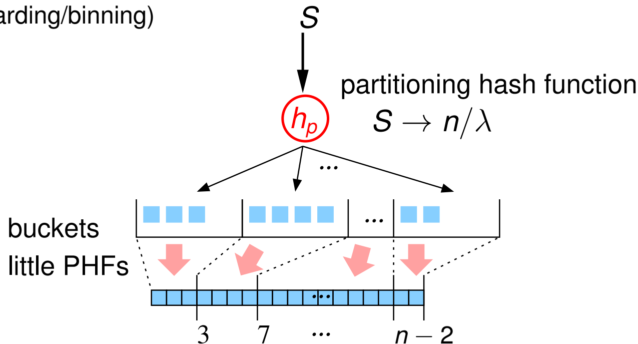
i has geometric distribution with parameter p
Its entropy is $\log 1/p + \mathrm{O}(1) = n \log e + \mathrm{O}(1) \approx 1.44n$.

- $+$ Optimal space
- $-$ Exponential construction time
- $+$ Perhaps good as a base case?
- $-$ Variable bitlength encoding



Random Partitioning + Brute Force?

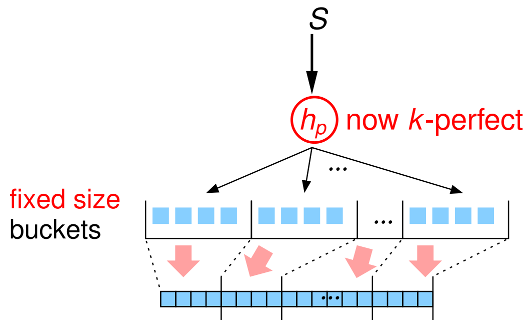
(aka bucketing/sharding/binning)



Tuning parameter: expected bucket size λ

- Space overhead $\approx \frac{\mathrm{O}(1) + \log \lambda}{\lambda}$
- Construction time $\mathrm{O}\left(n \cdot 5.58^{\lambda}\right)$
- Query time $\mathrm{O}(1)$. Not super favorable constants – Elias-Fano decoding of prefix sum of bucket sizes $+$ variable-bitlength seed decoding

k-Perfect Partitioning + Brute Force

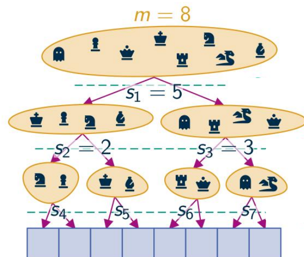


Lower bound indicates $\mathrm{O}\left(\frac{\log k}{k}\right)$ space for k-PHF.

$\approx$ space for prefix sums of bucket sizes.

- ne^k construction time
- \+ Practical with recent fast, compact k-PHFs [HKL+25]
- \+ Greatly simplifies the RecSplit and CONSENSUS approaches discussed next.

Splitting

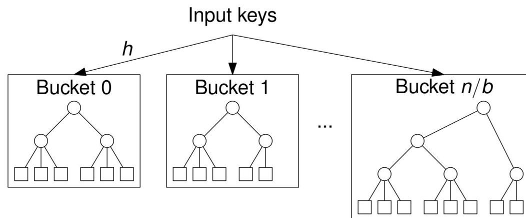


Similar to brute force but with a simpler task:
Use a splitting hash function $h : S \to \{0, 1\}$ to split S exactly into two halves.
(Assuming $n = |S|$ is even. Can be generalized for other splitting degrees)

- $\mathrm{O}(\sqrt{n})$ trials and $\frac{\log n}{2} + \mathrm{O}(1)$ bits of space.
- Observation [EGV20]: This is information-theoretically "efficient" – the space spent is saved on space needed for solving the subproblems (remaining overhead is $\mathrm{O}(1)$ bits per split).

RecSplit: Bucketing + Splitting + Brute Force? [EGV20]

Karlsruhe Institute of Technology



+ Exponential now only in size of small, fixed size ℓ leafs.
  Construction time now sth like $n \cdot e^{\ell}$
– Query time grows
  compared to non-splitting approaches
– Somewhat complicated

RecSplit: Performance Engineering [BKLS23]

Karlsruhe Institute of Technology

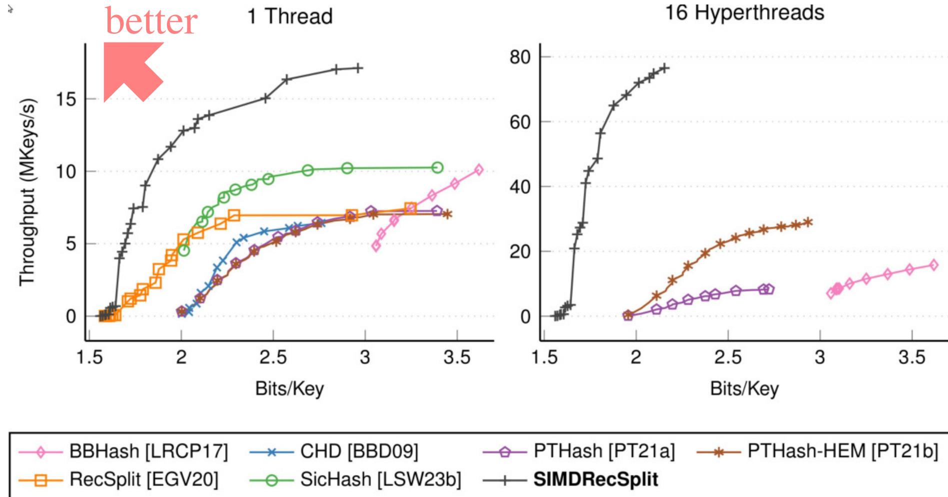
CPU:

- Threads: Parallelize over buckets
- SIMD-lanes: Parallelize over hash functions
- Bit-parallelism: Rotation fitting
  on additional, fixed 2-way split and rotation of one part

GPU: (NVIDIA CUDA)

- Stream of Kernels for different bucket sizes
- Kernels collect similar tasks (e.g., split $42 \rightarrow 21 + 21$)
- One block of threads for each bucket
- One thread for each hash function tried

RecSplit: Performance [BKLS23]



RecSplit: Fast Leaf-Construction

Karlsruhe Institute of Technology

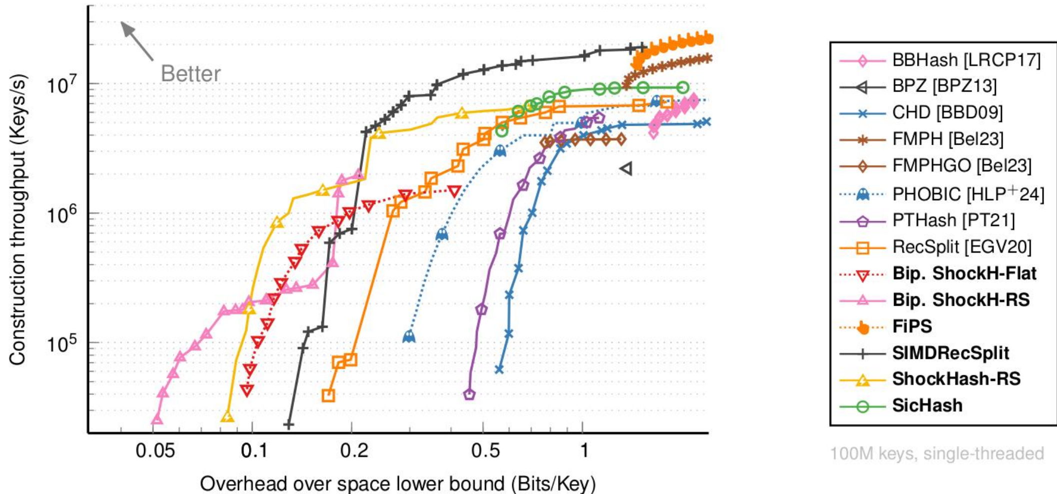
Basic RecSplit has time $\approx 2.71^{\ell}$ per leaf

- ShockHash [LSW24]: Combine with 1-bit retrieval 1.359^{ℓ}
- BipartiteShockHash [LSW25]: Build from pieces, Filter,... 1.165^{ℓ}
- MorphisHash [Her25]: specialized retrieval

Still – to achieve overhead ϵ, you need construction time $\Omega\left(n\alpha^{1/\epsilon}\right)$ for some constant $\alpha > 1$

Problem: $\Omega(1)$-bit space overhead for each of the many random construction tasks (splitting and base cases) we have to solve.

More Construction Performance Data



100M keys, single-threaded

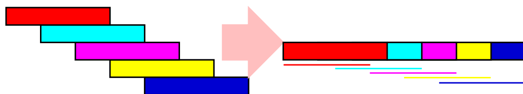
Consensus [LSWZ25]

Consider a sequence of K random construction tasks with success probability p.
Find sequence of succeeding seeds.

Classical Approach: Variable bitlength encodings of K geometric random deviates.

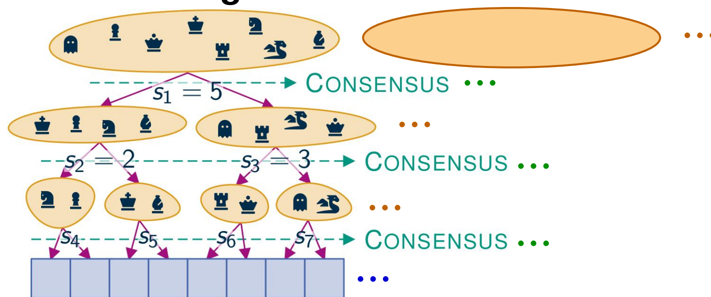
Brute Force: Store a single master seed.

Consensus: Allocate $(1+\epsilon)\log\frac{1}{p}$ seed bits for each experiment.
Actual seed used comprises also "enough" of the previous bits.
Simple random-walk like algorithm. Advance when succcessful. Otherwise backtrack.



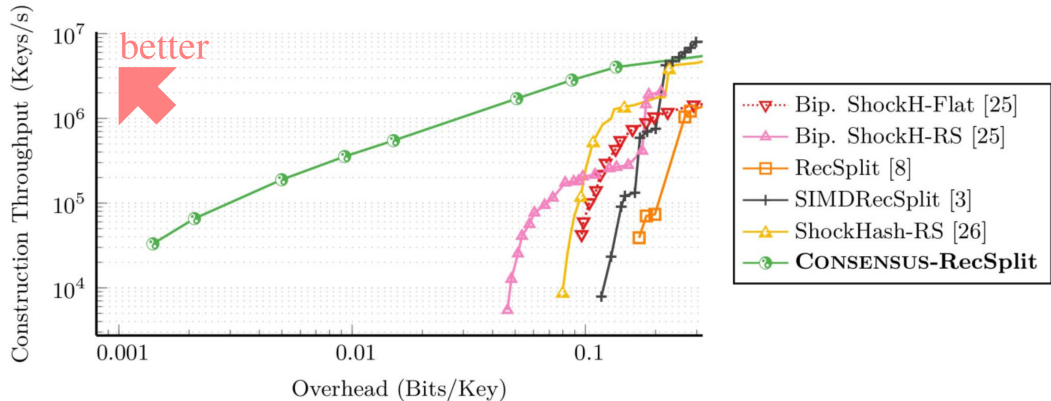
Approach	# of trials	space
Classical	$\frac{K}{p}$	$K\log\frac{1}{p} + \mathrm{O}(K)$
Brute force	$(\frac{1}{p})^K$	$K\log\frac{1}{p} + \mathrm{O}(1)$
Consensus	$\frac{K}{p\epsilon}$	$K\log\frac{1}{p}\cdot(1+\epsilon)$

Consensus [LSWZ25] RecSplit [EGV20] Minimal Perfect Hashing



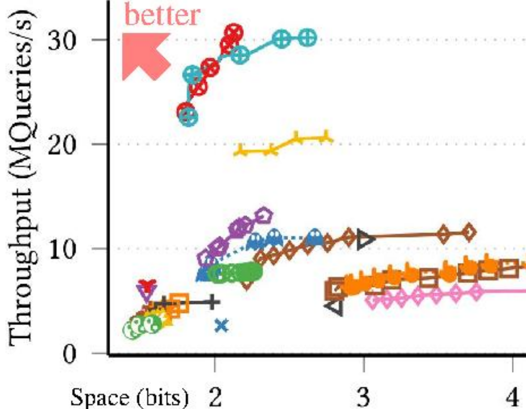
- Start with 2^c-perfect hashing (e.g., [HKL+25])
- Use brute force to guess perfect splitting seeds at c levels
- seed $s_i = b_{i-k} \cdots b_{i-1} b_i$ – share seeds in a left-looking window
- yields space $(1 + \epsilon) \log_2(e) n$ and construction time $\mathrm{O}(n/\epsilon)$

Space Efficient MPHFs



But What About
Query Time?

Query throughput versus
Bits per key
From [LMP+26]



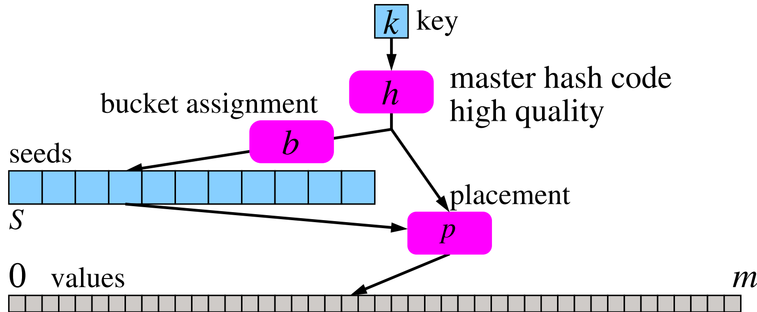
Bucket Placement
- FCH [66]
- CHD [11]
- PTHash [139]
- PHOBIC [90]
- PHast [16]
- PHast+ [16]
- PtrHash [83]

Fingerprinting
- BBHash [114]
- FMPH [15]
- FMPHGO [15]
- FiPS [106]

Multiple Choice
- BPZ [25]
- SicHash [108]
- ShockHash-RS [110]
- Bip. ShockH-Flat [109]
- Bip. ShockH-RS [109]
- MorphisHash-Flat [89]
- MorphisHash-RS [89]

Recursive Splitting
- RecSplit [62]
- SIMDRecSplit [19]
- CONSENSUS-RecSplit [104]

Bucket Placement PHFs

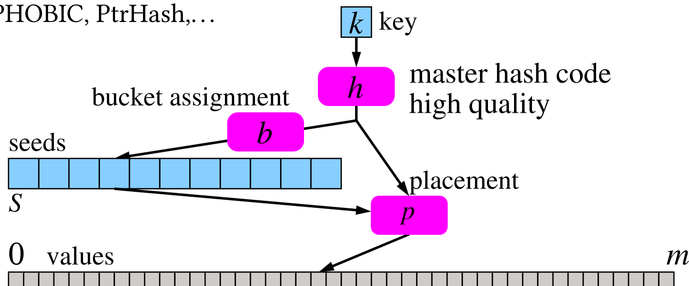


Query: $k \to p(h(k), S[b(h(k))])$

- $+$ Single memory access
- $+$ One high-quality hash function (h) $+$
  2 simple functions – b and p

Basic Bucket Placement Construction

CHD, PTHash, PHOBIC, PtrHash,…



values$[0..m-1] := \langle 0, \ldots, 0 \rangle$ -- occupied values

assign keys to buckets using h -- ≈ integer sorting by random keys

foreach bucket B by decreasing $|B|$ **do** -- most difficult first

  for $s := 0$ **to** ∞ **do** -- try seeds by brute force

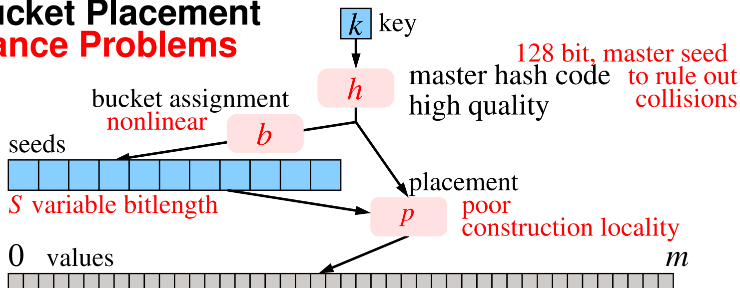
    if $\forall k \in B$: values$[p(h(k), s)] = 0 \wedge$ no intra-bucket collisions **then** -- success

      $S[B] := s$ -- remember seed

      foreach $k \in B$ **do** values$[p(h(k), s)] := 1$ -- block values

      break loop -- next bucket

Basic Bucket Placement
Performance Problems



values$[0..m-1] := \langle 0, \ldots, 0 \rangle$ -- occupied values

assign keys to buckets -- $\approx$ integer sorting by random keys

foreach bucket B by decreasing $|B|$ **do** -- parallelization requires sharding

  for $s := 0$ **to** ∞ **do** -- try seeds by brute force

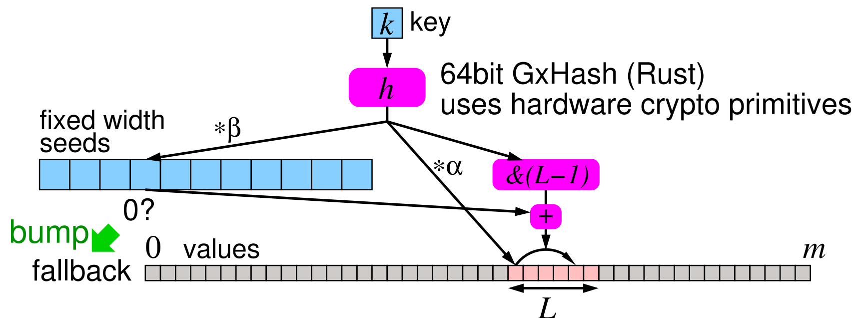
    if $\forall k \in B$: values$[p(h(k), s)] = 0 \wedge$ no intra-bucket collisions **then** -- poor locality

      $S[B] := s$

      foreach $k \in B$ **do** values$[p(h(k), s)] := 1$

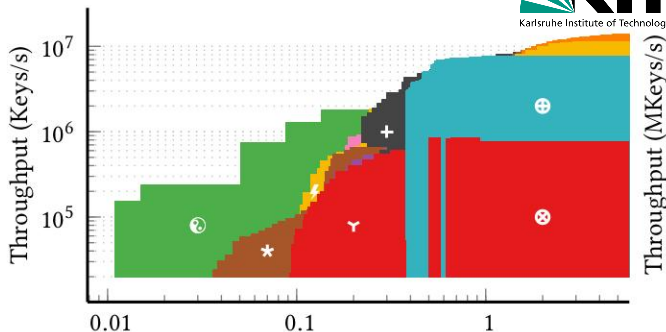
      break loop

PHast: Perfect Hashing Made Fast [BS26]



- Bumping allows few hash bits, fixed width seeds, and parallelization transparent to the query
- Moving window enables local construction and linear bucket assignment
  (success probabilities remain the same throughout construction)
- Additive seeds allow bit-parallel construction (in PHast+)

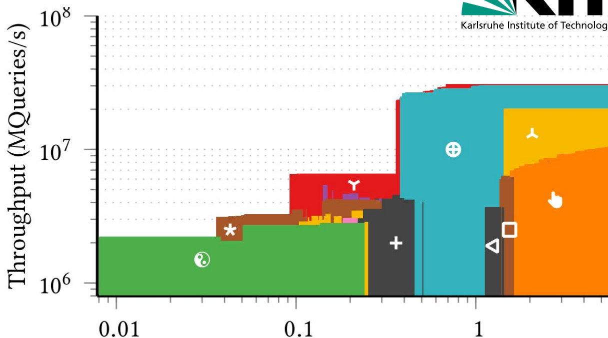
Minimal Perfect Hashing – Fastest Query



x: Space overhead in bits per key

Bucket Placement	Fingerprinting	Multiple Choice	Recursive Splitting
FCH [69]	BBHash [119]	BPZ [27]	RecSplit [64]
CHD [12]	FMPH [16]	SicHash [112]	SIMDRecSplit [21]
PTHash [144]	FMPHGO [16]	ShockHash-RS [114]	CONSENSUS-RecSplit [109]
PHOBIC [94]	FiPS [111]	Bip. ShockH-Flat [113]	
PHast [17]		Bip. ShockH-RS [113]	
PHast⁺ [17]		MorphisHash-Flat [93]	
PtrHash [87]		MorphisHash-RS [93]	

Minimal Perfect Hashing – Fastest Construction



x: Space overhead in bits per key

Bucket Placement	Fingerprinting	Multiple Choice	Recursive Splitting
FCH [69]	BBHash [119]	BPZ [27]	RecSplit [64]
CHD [12]	FMPH [16]	SicHash [112]	SIMDRecSplit [21]
PTHash [144]	FMPHGO [16]	ShockHash-RS [114]	CONSENSUS-RecSplit [109]
PHOBIC [94]	FiPS [111]	Bip. ShockH-Flat [113]	
PHast [17]		Bip. ShockH-RS [113]	
PHast+ [17]		MorphisHash-Flat [93]	
PtrHash [87]		MorphisHash-RS [93]	

Perfect Hashing: Summary and Future Work

Karlsruhe Institute of Technology

PHast:

- Query as fast as it gets
- Construction almost as fast as competitors that need more space and more query time.
  (for PHast$^+$, bit-parallelism almost eliminates brute-force aspects)
  $\rightarrow$also bridge the remaining gap?
- Theoretical analysis is open.
  does it approach the lower bounds for large buckets?
- Combination with CONSENSUS

Splitting + CONSENSUS:

- Approaching the lower bound becomes practical
- Slow queries
- Faster ways to use CONSENSUS?
  In particular, based on bucket placement?

More open problems:

- Space-optimal k-perfect hashing (also non-minimal)?
- Reconsider applications: static hash tables (e.g. for DB join?), updateable retrieval (e.g. for counting Bloom Filters),. . .

Overview

Karlsruhe Institute of Technology

- Motivation and Methodology
- Hash-table Variants
  - Perfect Hashing Deep Dive
  - **Static Function Retrieval**
  - Approximate Membership Query (AMQ) Filter aka Bloom Filter
- Bit vectors with rank and select
- Summary

Static Function Retrieval

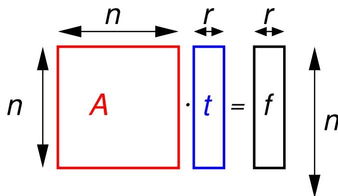
For $S = \{s_1, \ldots, s_n\}$ allow evaluating $f : S \to \{0,1\}^r$.

key	value
Godzilla	**
Ben Hur	***
Attack of the Killer Tomatoes	*
Three Gifts for Cinderella	****
Howl's Moving Castle	****
Metropolis	****

$\approx rn$ bits space, $O(1)$ query, $O(n)$ construction?

Brute Force: Random Matrix

Consider a hash function $h : S \to \{0,1\}^n$.
Row i of A is $h(s_i)$. Postulate: $A_i \cdot t = f(s_i)$



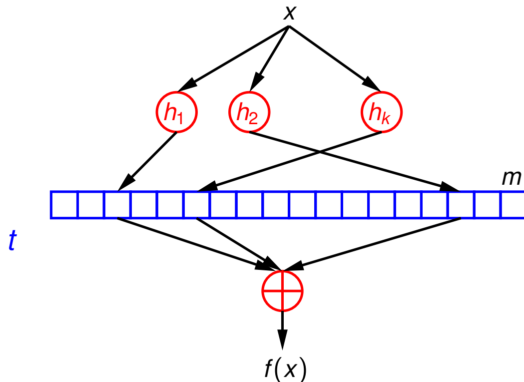
- A is a random $n \times n$ matrix

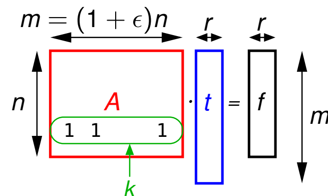
+ A has full rank with constant probability (store a succeeding hash seed)
+ Space $rn + O(\log n)$ bits
− Cubic construction time (solve equation system)
− Linear query time (product of t with a row of A)

Sparse Matrices (aka XOR-Filters) [BPZ13, GL20]

Most well known: $k \in 3..7$ random nonzeroes per row.

+ Linear time construction heuristics (peeling) for $k = 3$ and $m > 1.21n$
− Otherwise slow construction
− Bad locality for query and construction



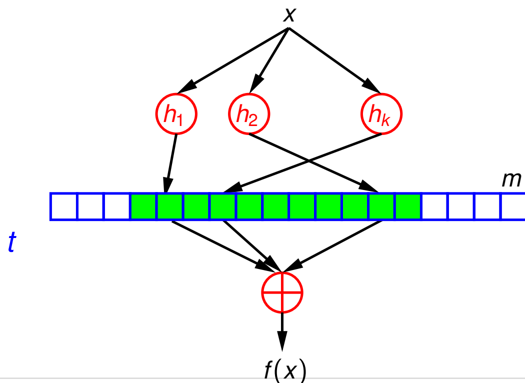


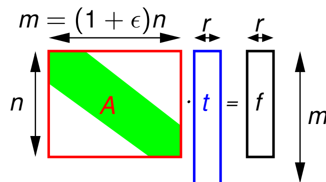
Coupling (aka Fuse Filters)

Karlsruhe Institute of Technology

[Wal25, GL22] $k \in 3..6$ random nonzeroes in a narrowish band $w = n^{\alpha}$, $\alpha < 1$

+ Peeling now useful for all k and almost as good as solving the equation system
+ Low space overhead $k = 3 \to 9\%$, $k = 4 \to 2.4\%$, $k = 5 \to 0.8\%$, as $n \to \infty$
+ Some locality for construction
− Still bad locality for query
+ Fastest known query for large r, single thread

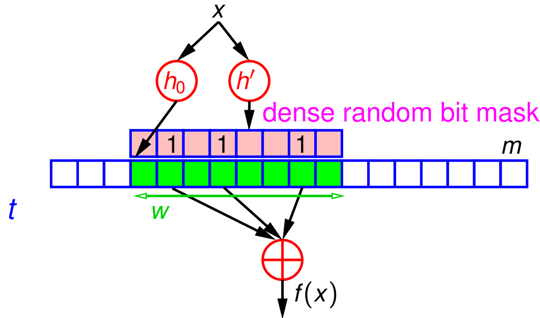


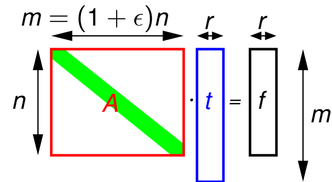


Ribbon

[DW21, DDH+26] Access one small, dense block (size $w = \Theta\left(\frac{\log n}{\epsilon}\right)$)

+ Solvable in near linear time (ribbon solving)
+ High locality
– Requires sharding to achieve practical w
– $\Theta(wr)$ bit operations per query.

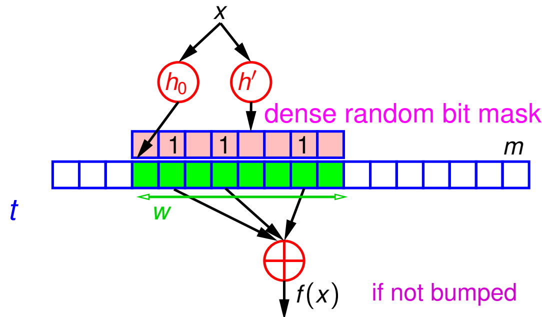


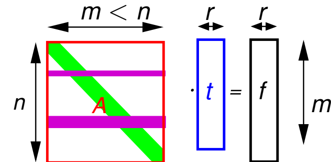


BuRR: Bumped Ribbon Retrieval

[DDH+26] enforce max. rank by bumping (i.e., removing) offending rows.
Overloading ($\epsilon < 0$) largely eliminates redundancy.

- + Very little metadata – 1–2 bits per $\mathrm{O}\left(\frac{w^2}{\log w}\right)$ keys.
- + Small word size w (32–128) OK, no sharding needed
- + Space overhead below 1% becomes practical





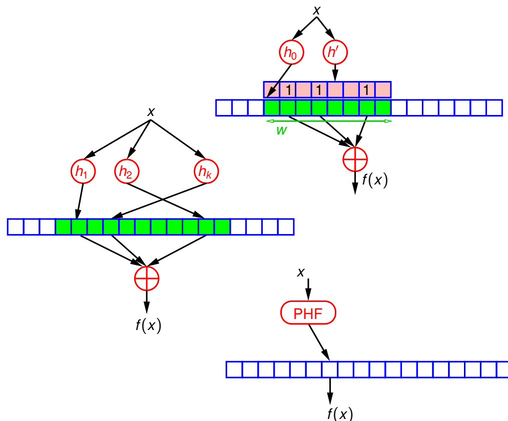
Discussion: Static Retrieval Methods of Choice?

Karlsruhe Institute of Technology

stay tuned for some experiments

- BuRR is better for small r, low overhead
- Coupling is better for large r, higher overhead, single-threaded
- For large r, multi-threaded/batched recent fast perfect hashing might also be useful, (its also updateable) stay tuned

Various further ideas possible, e.g.,
BuRR with more sparse blocks.
k-perfect hashing + tiny dense matrices + Consensus?



Overview

- Motivation and Methodology
- Hash-table Variants
  - Perfect Hashing Deep Dive
  - Static Function Retrieval
  - **Approximate Membership Query (AMQ) Filter aka Bloom Filter**
- Bit vectors with rank and select
- Summary

AMQs: Approximate Membership Query Filters

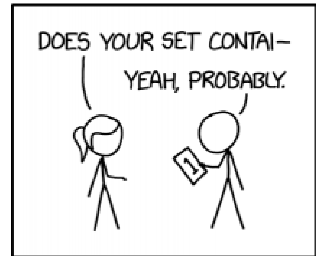
For $S = \{s_1, \ldots, s_n\}$
support approximate isMember$(x) \in \{0, 1\}$ operation

Case $x \in S$, result 1: true positive query

Case $x \notin S$, result 0: true negative query

Case $x \notin S$, result 1: false positive query with false positive rate f

Lower space bound $n \log \frac{1}{f}$

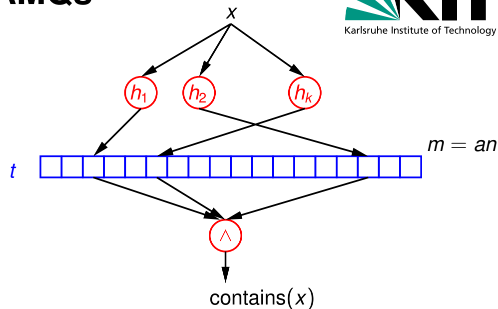


xkcd.com/2934

Applications:
data bases, cybersecurity, content delivery networks, systematic search, blockchains, bioinformatics, content recommendations, ... [GA13, BM04]

Bloom Filters – Simple Dynamic AMQs

[Blo70]≈11 000 citations



Consider bit vector $b[1..an]$ and
hash functions $h_1, \ldots, h_k$ with range $1..an$.

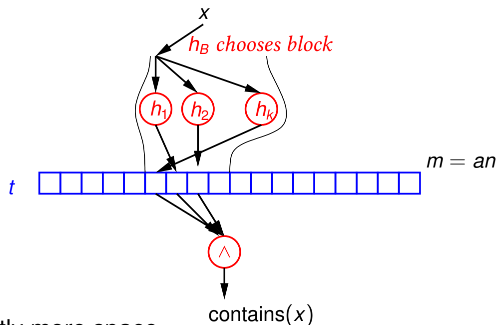
Inserting x: set $b[h_1(x)], \ldots, b[h_k(x)]$.

contains$(x) = b[h_1(x)] \wedge \cdots \wedge b[h_k(x)]$.

- $+$ supports insertion (but with poor adaptivity)
- $+$ simple
- $-$ space overhead about 44 %
- $-$ poor locality, in particular for small f

Blocked Bloom Filters [PSS10]

Set k random bits in a block (cache line)

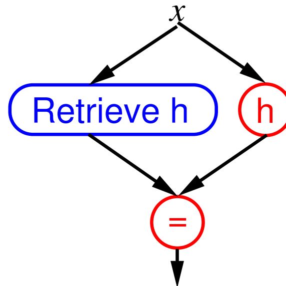


Much better locality but slightly more space.

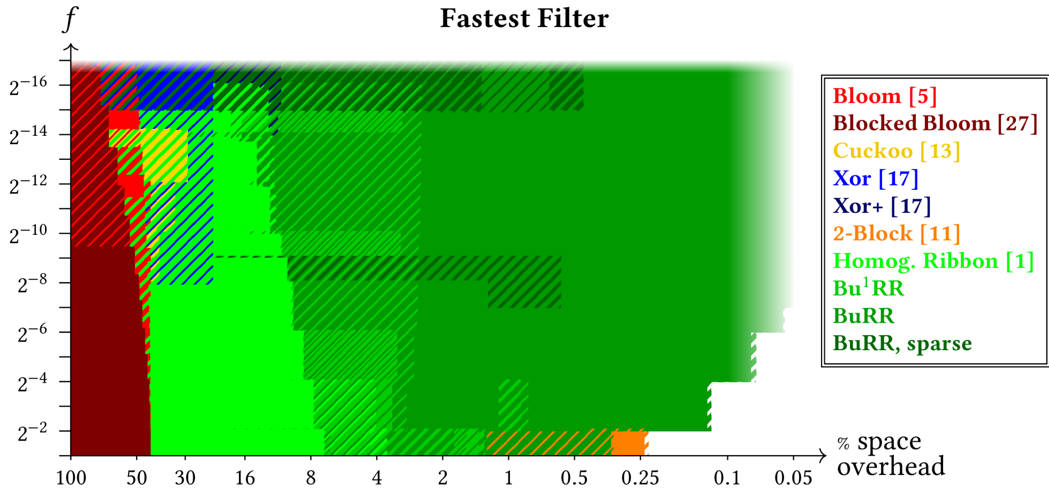
Static Retrieval Based AMQs

static function stores a random fingerprint $h(x)$ of each key x.

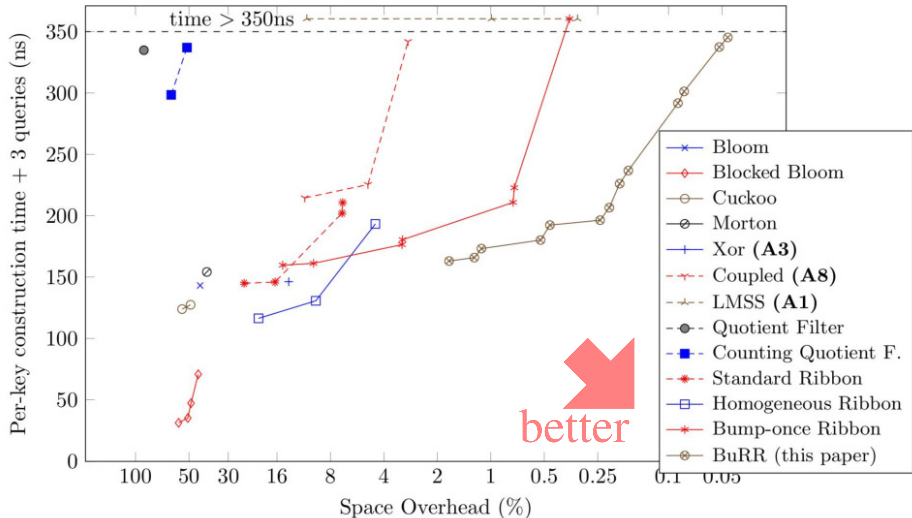
- Better space than Bloom for XOR-filters, Coupling/Fuse, Cuckoo, Ribbon, PHF,. . .
- Succinct solution using BuRR



Tradeoff Speed, Space, f (PHFs missing yet)



AMQs with False Positive Rate $\approx 1/100$



Overview

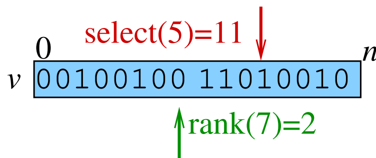
Karlsruhe Institute of Technology

- Motivation and Methodology
- Hash-table Variants
  - Perfect Hashing Deep Dive
  - Static Function Retrieval
  - Approximate Membership Query (AMQ) Filter aka Bloom Filter
- **Bit vectors with rank and select**
- Summary

Bit Vectors

Store an array v of n bits.
Operations:

- **access**(i): $v[i]$
- **rank**(i): $|\{j \leq i : v[j] = 1\}|$ (count ones)
- **select**$_{0/1}(i)$: position of i-th 0/1 in v (default is select$_1$)



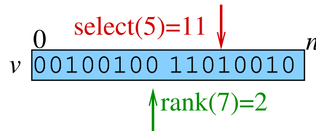
Versatile and fundamental building block of compressed/succinct data structures.
Applications: Elias-Fano encoding of sorted sequences, trees (2 bits per node), wavelet trees, range-minima, AMQ filters, perfect hashing, static function retrieval,. . .

Succinct Bit Vectors

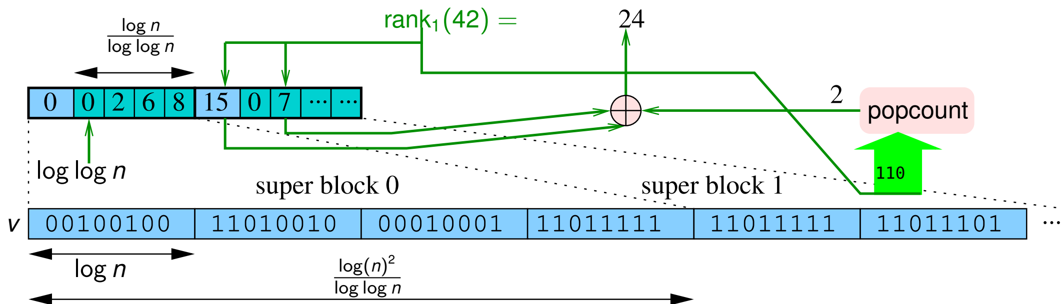
Assume the n bits themselves are not compressed/compressible.
Additional space to support rank and or select in constant time on a RAM:

$$\Theta\left(n\frac{\log\log n}{\log n}\right)$$

Is this practical?



Rank is Practical and Highly Engineered



Performance Engineering: Cache aligned blocks. Careful superblock layout. SIMD horizontal add versus precomputed prefix sums? Interleave metadata? 2 or 3 levels? Depends on n and sweet spots of design space.

$\approx$ 3 % space overhead for 512-bit cache lines. 1–2 parallel memory probes.

Select in Practice? That was less Clear

Karlsruhe Institute of Technology

- Constant time query needed $\omega(n\frac{\log\log n}{\log n})$ space
- $\omega(1)$ worst case query time with $\mathrm{O}\left(n\frac{\log\log n}{\log n}\right)$ space
- Theorectical asymptotically optimal constructions are not directly promising (likely neither fast nor space efficient compared to practical solutions)

select(5)=11

v 00100100 11010010

Select: Theory Meets Practice

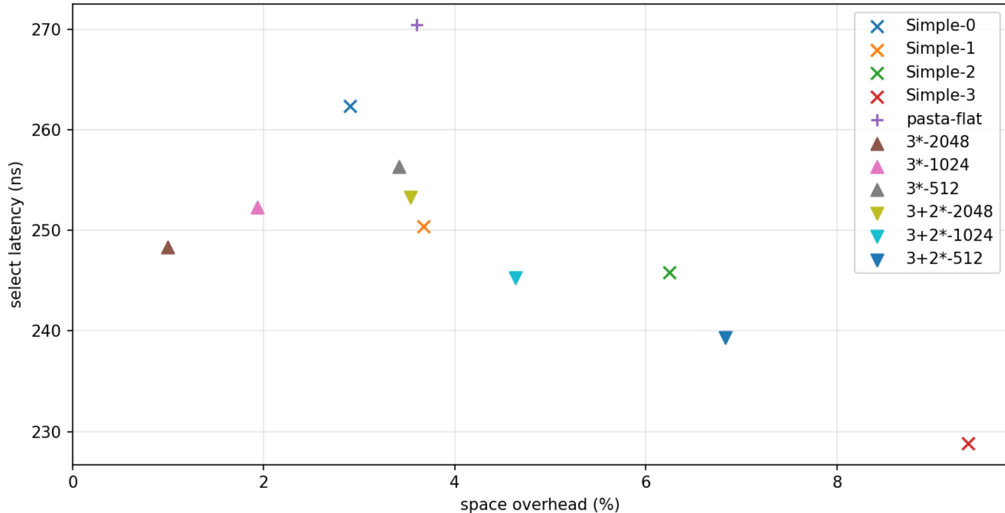
Karlsruhe Institute of Technology

- Reuse rank data structure (but slight deviations can have noticable impact)
- Span design space “summary tree” levels $\times$ “sample tree” levels $\times$ sample compression

$\Rightarrow$ $o(n\frac{\log\log n}{\log n})$ space for select metadata

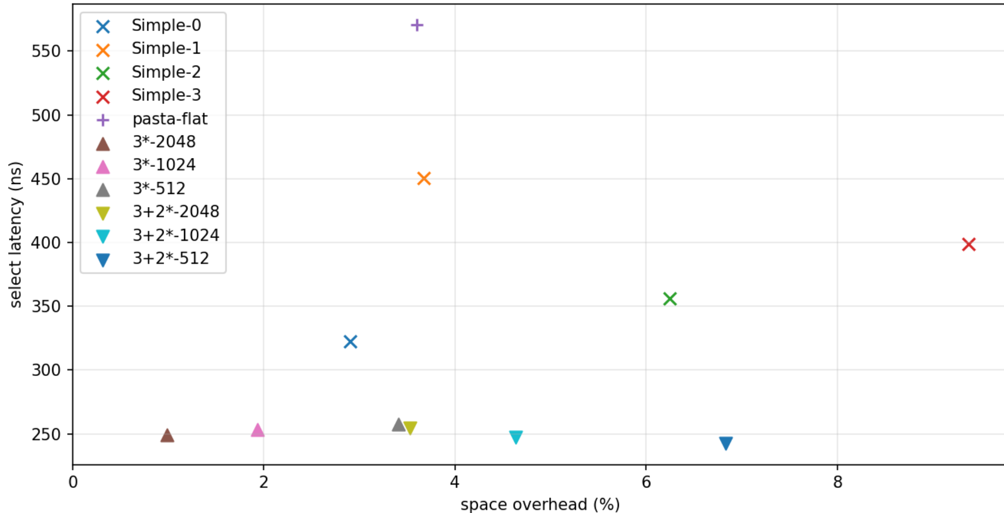
- Indeed $\ll$ rank data in practice
- Some space and time overhead only for inputs where previous implementations get slow

Select: Performance



Select: Performance

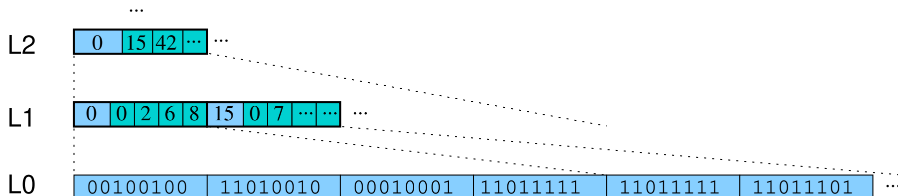
Karlsruhe Institute of Technology



Select: Summary Trees

Karlsruhe Institute of Technology

Continue construction of summaries beyond the 2 levels needed for rank.
An Li block summarizes some L$i-1$ blocks.
A select query now descends this tree (similar to a static B-Tree)

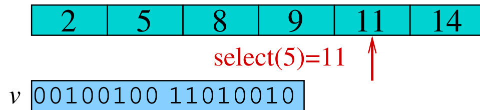


+ negligibly more space than the 2 bottom levels needed for rank
− logarithmic query time
+ useful ingredient of more complex designs

Select: Tabulation

Precompute all results.

- $+$ $O(1)$ query time
- $-$ $\Theta(n \log n)$ bits space



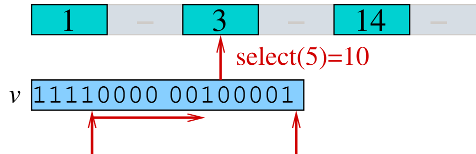
Select: Sampling

Karlsruhe Institute of Technology

Precompute position of every k-th 1-bit.
Scan/binary search in range select$(k\lfloor i/k \rfloor)$–select$(k\lceil i/k \rceil)$

- Space $\frac{n \log n}{k}$
- Average case query time $\mathrm{O}(\log k)$

– Worst case query time $\Omega(\log n)$ (at large gaps)

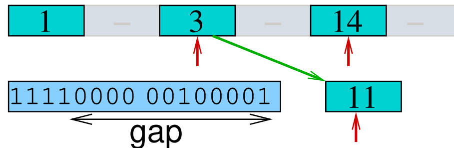


Select: Sample Tree

Karlsruhe Institute of Technology

Idea: Use denser samples at large gaps – possibly recursively.

+ Can achieve asymptotically optimal space and time
– Previous theoretical results are not promising for practice – [Gol07] needs 4 levels, complex bit compression, and still substantial overhead
+ Can be configured to be almost identical to previous approach regarding the average case
+ We derive variants that work well in practice, also in the worst case. Using 2–3 levels of sample tree.



Outlining the Design Space for Select

Karlsruhe Institute of Technology

- ℓ levels of sample tree
- How aggressively is the sample tree compressed?
  - uncompressed
  - bit-compression
  - Elias-Fano encoding (information-theoretically near optimal)
- k levels summary tree as entry points from the sample tree

The RAM model implies word size $\Theta(\log n)$ this yields asymptotic results on space overhead for the select data structure.

Overhead $\mathrm{O}\left(\mathrm{polyloglog}(n)/\log^{\mathrm{tableentry}} n\right)$

Karlsruhe Institute of Technology

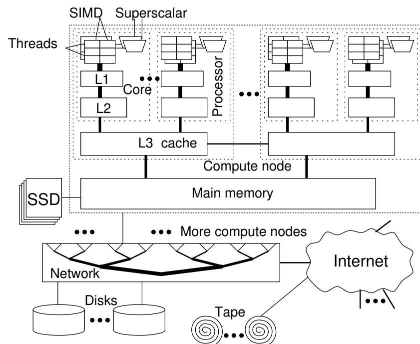
	ℓ	k: 0	1	2	3	4	5	≥ 7
uncompressed	1	**-1**	—	—	—	—	—	—
	2	—	**-0.5**	**0**	**0.5**	1	1.5	—
	3	—	-0.33	0.33	**1**	**1.66**	—	—
	4	—	—	0.5	1.25	—	—	—
	5	—	—	0.6	—	—	—	—
compressed	1	**0**	—	—	—	—	—	—
	2	—	**0**	**0.5**	**1**	**1.5**	—	1 [RRS07]
	3	—	—	**1**	**1.66**	—	—	—
	4	—	—	—	1 [Gol07]	—	—	—
EF compr.	1	**0**	—	—	—	—	—	—
	2	—	**0.5**	**1**	**1.5**	—	—	—
	3	—	0.66	1.33	—	—	—	—

Outlining the Analysis Space

Karlsruhe Institute of Technology

- ℓ levels of sample tree
- How aggressively is the sample tree compressed?
- k levels summary tree
- Input size $n \rightsquigarrow$ word size $\mathrm{O}(\log n)$ in the RAM model
- Cache line size B
- SIMD word size w
- Cache sizes $M_1, M_2, M_3, \ldots$
- Parallel versus data dependent access versus multithreading

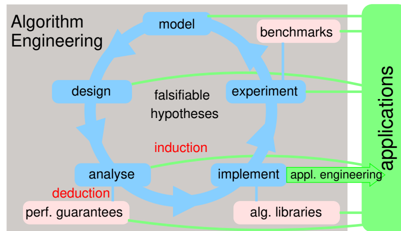
Could be part of a more complex analysis but might get messy.

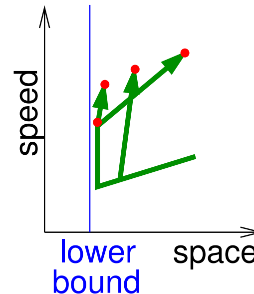


Performance Engineering Insights

Karlsruhe Institute of Technology

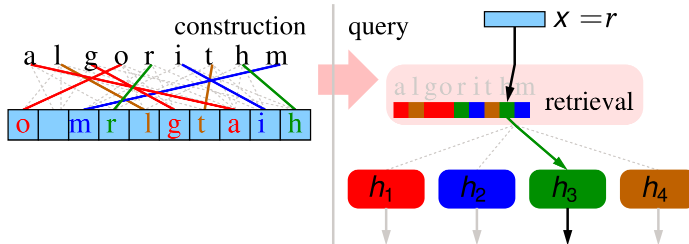
- Recent fast compressed data structures likely to get more and more relevant for performance engineering
- Theoretical analysis gives qualitative and quantitative insights that help design and tuning
- Even seemingly idiotic extreme approaches are a good starting point
- Spanning large design spaces yields a spectrum of Pareto optimal solutions





Retrieval Based Perfect Hashing

One of the leading approaches until recently – with a boost through BuRR [LSW23].
For example: for each $s \in S$ store $r = 2$ bits indicating which of 4 hash functions to use. Use cuckoo-hashing/bipartite matching for construction (possible in linear epected time). Yields a PHF for $m/n \geq 1.024$ whp.
Using another 0.2 bits per element, this can “repaired” to a MPHF. See next slide.



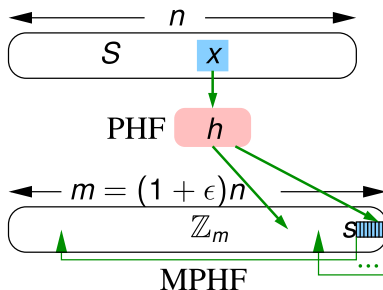
Good but suboptimal space. Somewhat slow compared to the currently best approaches.

Repairing a PHF to obtain an MPHF

Map keys with $h(x) > n$ to unused slots.
Using Elias-Fano encoding (and thus bit vectors) this takes space about

$$(m-n)(2+\log\frac{n}{m-n}).$$

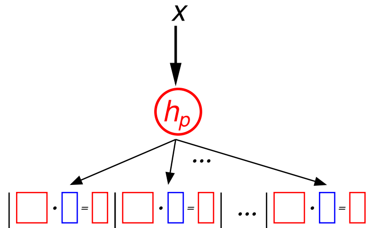
If $m - n \ll n$ overhead is paid only by few keys.



Sharding/Partitioning – Again

Assume retrieval size $r = \mathrm{O}(1)$.

- Partitioning hash function h_p maps elements to shards of size $\Theta(\log n)$
- Constant time row operations using word parallelism
- $\frac{n}{\log n} \times \frac{\log^3 n}{\log n} = n \log n$ construction time
- $\mathrm{O}(1)$ query time using word parallelism
- Space $(1 + o(1))rn$
  (assuming space efficient encoding of offsets and seeds)

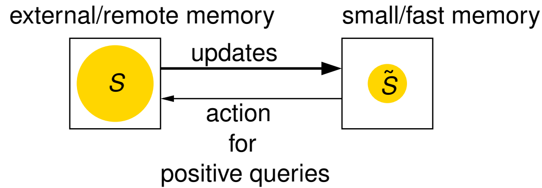


More on Sharding

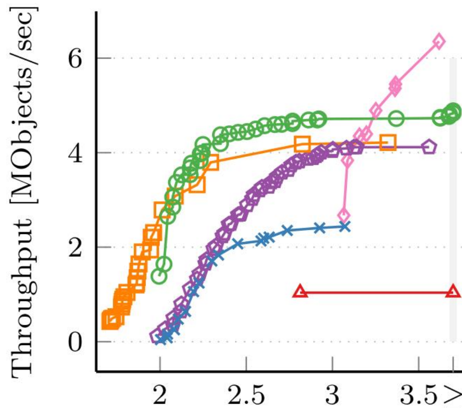
Karlsruhe Institute of Technology

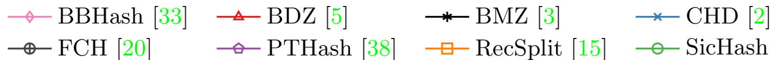
- Similar techniques use in many hashing related data structures, AMQs, perfect hashing,...
- Large shards are a standard way to support parallel construction but may imply some space and query time overhead.
- Theoretical solutions (e.g. [Por09]) use very small buckets plus large lookup tables to achieve theoretically optimal results. But calculations show that this does not yield really small overheads for realistic n.

Typical Application of AMQs

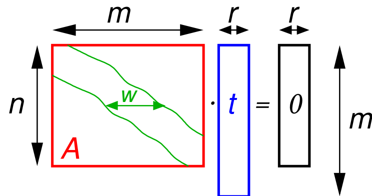


Homogeneous Ribbon Filter

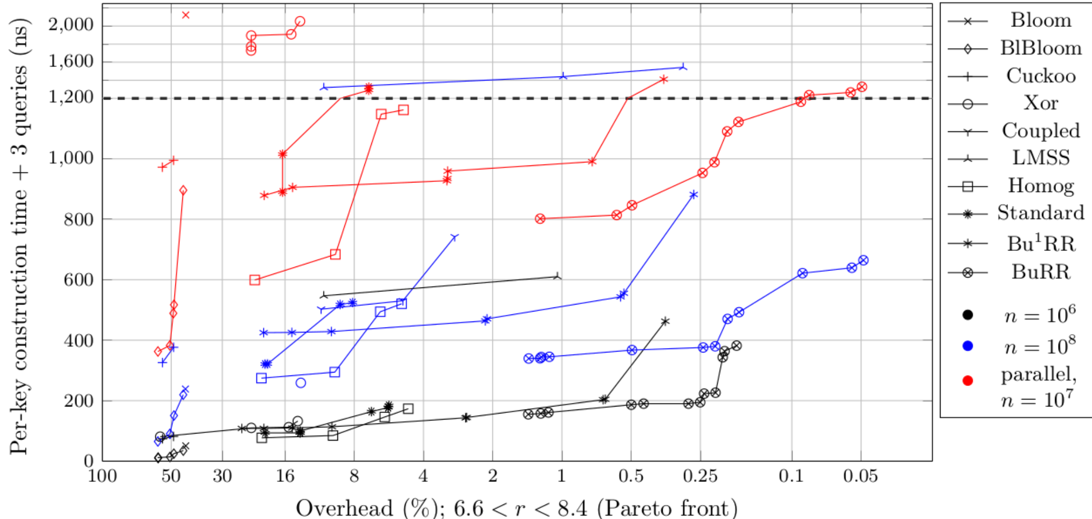




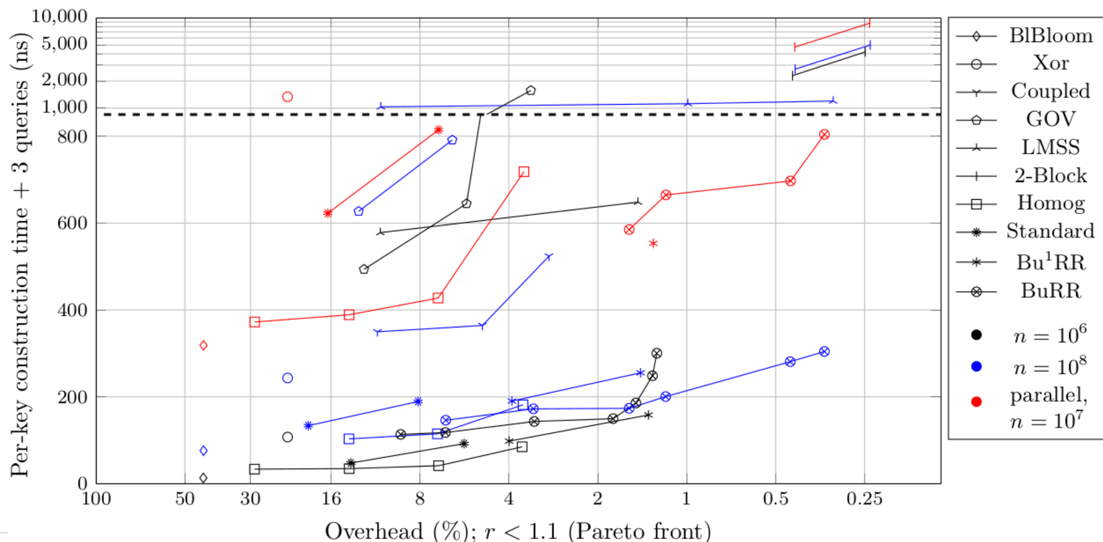
Solve a homogenous system of equations.
⇒ always solvable.
Take a random solution.



Tradeoff Speed, Space, f



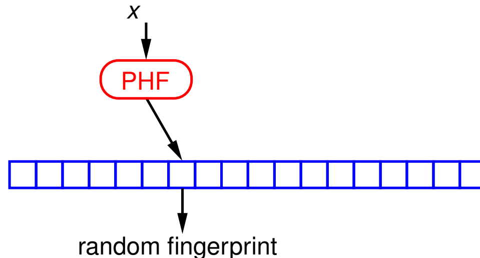
Tradeoff for Small r



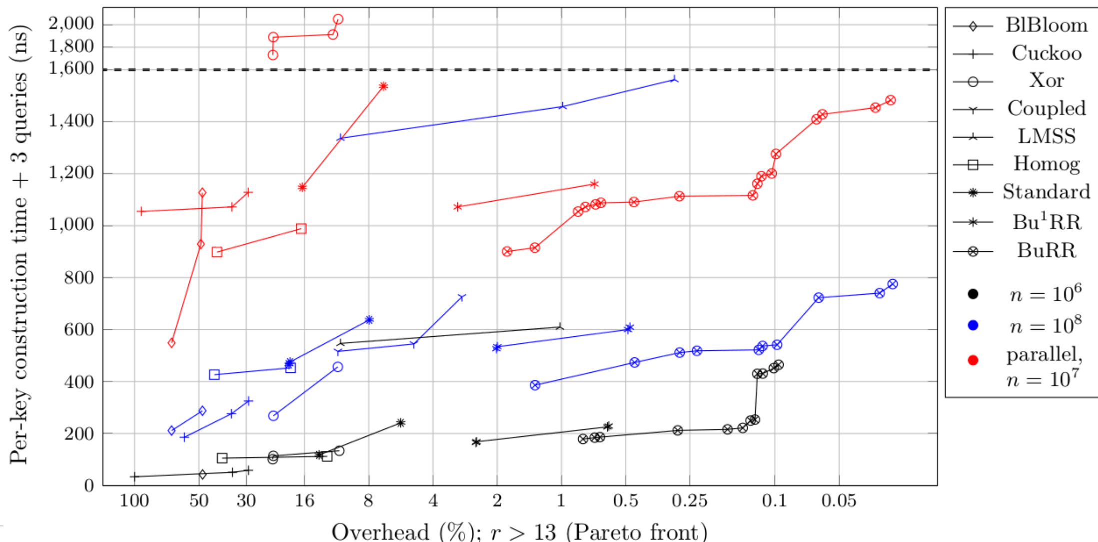
Perfect Hashing Based AMQs

PHF indexes an array with fingerprints.
Use fast PHFs like PHast (stay tuned)

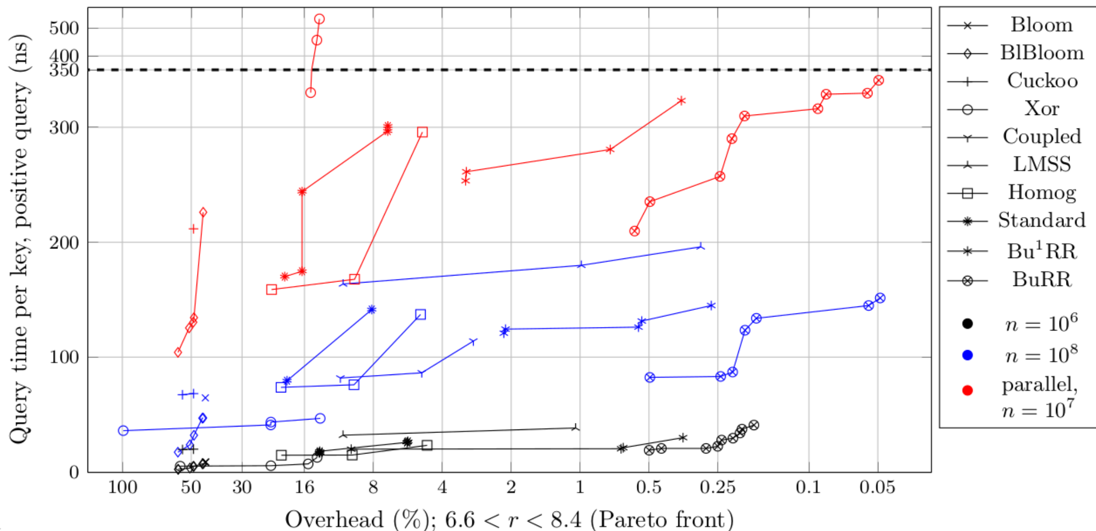
- Low overhead ($\approx 2/r$) for large r
- Faster than Ribbon/BuRR for large r
- Faster than XOR/Coupling when memory in congested (parallel/batched)



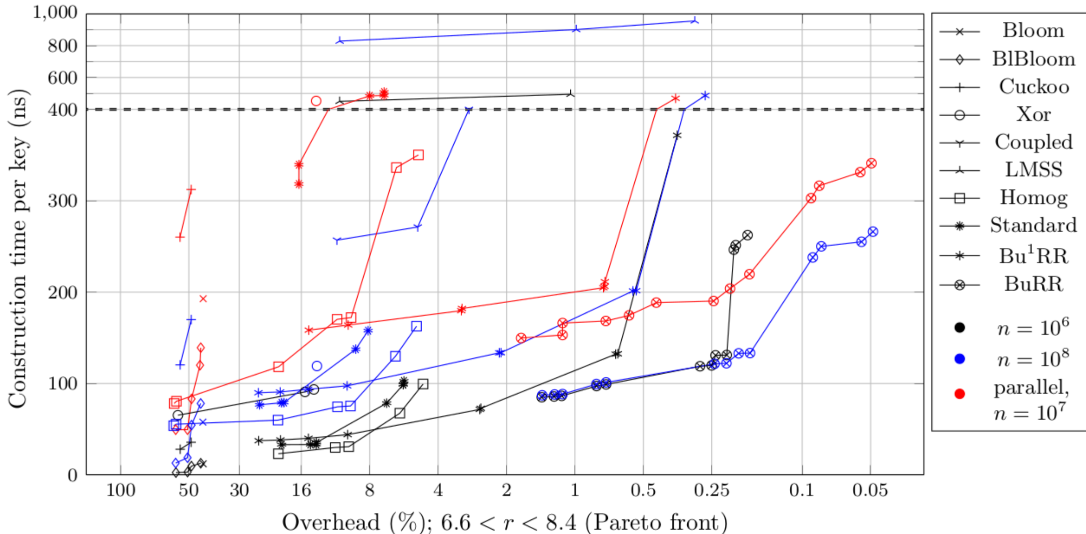
Tradeoff for Large r



Tradeoff Query Time – Space ($r = 8$)



Tradeoff Constr. T. – Space ($r = 8$)



References I

Karlsruhe Institute of Technology

[BKLS23] Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders.
High performance construction of RecSplit based minimal perfect hash functions.
In *31st European Symposium on Algorithms (ESA)*, volume 274 of *LIPIcs*, pages 19:1–19:16, 2023.

[Blo70] Burton H Bloom.
Space/time trade-offs in hash coding with allowable errors.
Communications of the ACM, 13(7):422–426, 1970.

[BM04] Andrei Broder and Michael Mitzenmacher.
Network applications of Bloom filters: A survey.
Internet mathematics, 1(4):485–509, 2004.

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

[BS26] Piotr Beling and Peter Sanders.
PHast – perfect hashing with fast evaluation.
In *Symposium on Algorithm Engineering and Experiments, (ALENEX)*. SIAM, 2026.

[DDH+26] Martin Dietzfelbinger, Peter C. Dillinger, Lorenz Hübschle, Peter Sanders, and Stefan Walzer.
Ribbon: Fast succinct static retrieval and approximate membership.
Journal of the ACM, 2026.

[DW21] Peter C. Dillinger and Stefan Walzer.
Ribbon filter: practically smaller than Bloom and xor, 2021.

[EGV20] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna.
RecSplit: Minimal perfect hashing via recursive splitting.
In *22nd Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 175–185. SIAM, 2020.

[GA13] Shahabeddin Geravand and Mahmood Ahmadi.
Bloom filter applications in network security: A state-of-the-art survey.
Computer Networks, 57(18):4047–4064, 2013.

References II

Karlsruhe Institute of Technology

[GL20] Thomas Mueller Graf and Daniel Lemire.
Xor filters.
ACM Journal of Experimental Algorithmics, 25(1):1–16, Nov 2020.

[GL22] Thomas Mueller Graf and Daniel Lemire.
Binary fuse filters: Fast and smaller than xor filters.
Journal of Experimental Algorithmics (JEA), 27(1):1–15, 2022.

[Gol07] Alexander Golynski.
Optimal lower bounds for rank and select indexes.
Theoretical Computer Science, 387(3):348–359, 2007.

[Her25] Stefan Hermann.
Morphishash: Improving space efficiency of shockhash for minimal perfect hashing.
In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms, ESA 2025, Warsaw, Poland, September 15-17, 2025*, volume 351 of *LIPIcs*, pages 9:1–9:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[HKL+25] Stefan Hermann, Sebastian Kirmayer, Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer.
Engineering minimal k-perfect hash functions.
In *33rd European Symposium on Algorithms (ESA)*, volume 351 of *LIPIcs*, pages 99:1–99:18, 2025.

[LMP+26] Hans-Peter Lehmann, Thomas Mueller, Rasmus Pagh, Giulio Ermanno Pibiri, Peter Sanders, Sebastiano Vigna, and Stefan Walzer.
Modern minimal perfect hashing: A survey.
ACM Computing Surveys, 2026.

[LSW23] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer.
SicHash – small irregular cuckoo tables for perfect hashing.
In *Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 176–189, 2023.

[LSW24] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer.
ShockHash: Towards optimal-space minimal perfect hashing beyond brute-force.
In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, page 194–206, 2024.

References III

Karlsruhe Institute of Technology

[LSW25] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer.
Shockhash: Near optimal-space minimal perfect hashing beyond brute-force.
Algorithmica, 87(11):1620–1668, 2025.

[LSWZ25] Hans-Peter Lehmann, Peter Sanders, Stefan Walzer, and Jonatan Ziegler.
Combined search and encoding for seeds, with an application to minimal perfect hashing.
In *33rd European Symposium on Algorithms (ESA)*, volume 351 of *LIPIcs*, pages 109:1–109:18, 2025.

[Por09] Ely Porat.
An optimal Bloom filter replacement based on matrix solving.
In *Computer Science Symposium in Russia (CSR)*, volume 5675 of *LNCS*, pages 263–273. Springer, 2009.

[PSS10] F. Putze, P. Sanders, and J. Singler.
Cache-, hash- and space-efficient Bloom filters.
ACM Journal on Experimental Algorithmics, 14:4:4.4–4:4.18, January 2010.

[RRS07] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti.
Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets.
ACM Transanction on Algorithms, 3(4):43, 2007.

[San09] P. Sanders.
Algorithm engineering – an attempt at a definition.
In *Efficient Algorithms*, volume 5760 of *LNCS*, pages 321–340. Springer, 2009.

[Wal25] Stefan Walzer.
Peeling close to the orientability threshold – spatial coupling in hashing-based data structures.
ACM Trans. Algorithms, January 2025.
also arxiv 2001.10500, 2020 and SODA 2021.