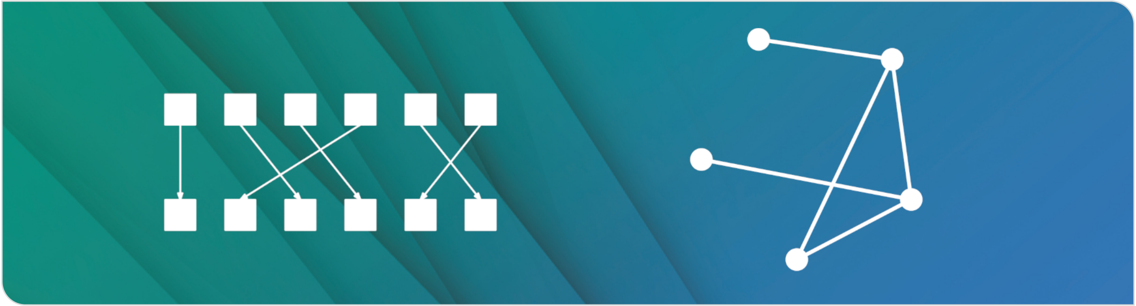
Karlsruhe Institute of Technology

Mining Perfect Hash Functions

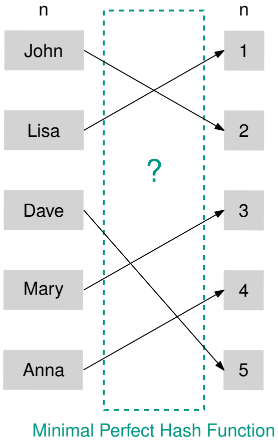
SPAA Workshop

Hans-Peter Lehmann, Stefan Hermann, **Peter Sanders**, Stefan Walzer | Jun 17, 2024

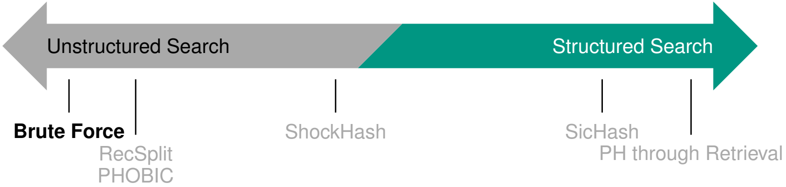


Minimal Perfect Hash Function (MPHF)

- Static set of n keys
- Data structure to injectively/bijectively map keys to the first n integers
- MPHF: Lower bound $1.44n$ bits $\ll$ space of input keys
- Goal: Near minimal space, constant time query, linear construction time
- Applications: databases, hash tables, AMQ, retrieval, replace pointers

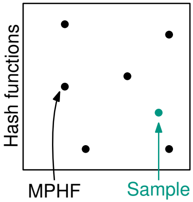


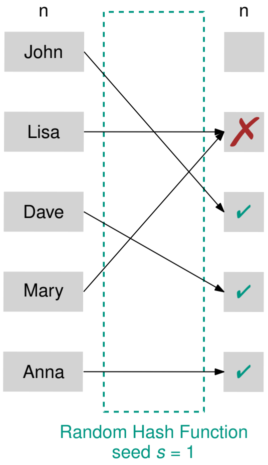
Overview


Unstructured Search
Structured Search
Brute Force
RecSplit
PHOBIC
ShockHash
SicHash
PH through Retrieval

Brute-Force Construction

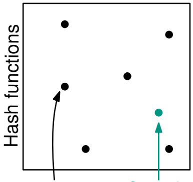
- Given a hash function
  try seeds 1, 2, 3, . . .
- Perfect hash function data structure:
  store successful seed s
- Expected tries: $n^n/n! \approx e^n$
  $\Rightarrow \approx n \log e \approx 1.44n$ bits (this is optimal)
- But exponential construction time, ≈linear query time

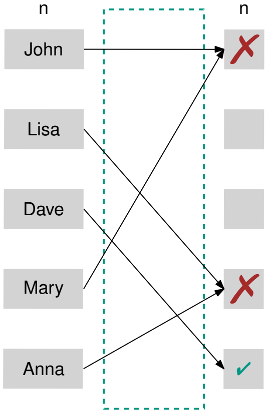




Brute-Force Construction

- Given a hash function
  try seeds 1, 2, 3, . . .
- Perfect hash function data structure:
  store successful seed s
- Expected tries: $n^n/n! \approx e^n$
  $\Rightarrow \approx n \log e \approx 1.44n$ bits (this is optimal)
- But exponential construction time, $\approx$linear query time

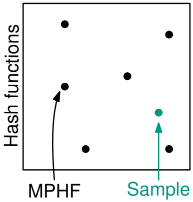


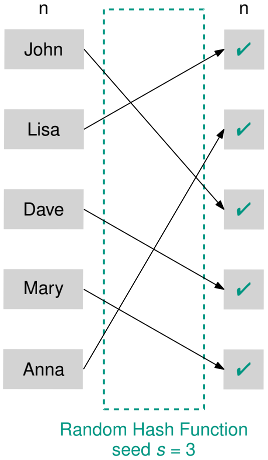


Brute-Force Construction

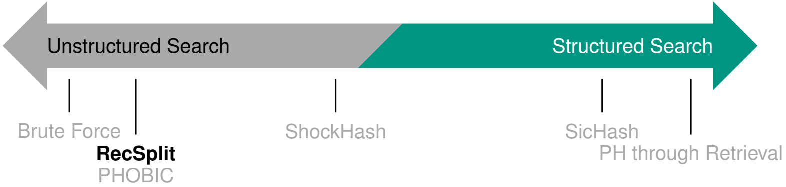
Karlsruhe Institute of Technology

- Given a hash function
  try seeds 1, 2, 3, ...
- Perfect hash function data structure:
  store successful seed s
- Expected tries: $n^n/n! \approx e^n$
  $\Rightarrow \approx n \log e \approx 1.44n$ bits (this is optimal)
- But exponential construction time, $\approx$linear query time



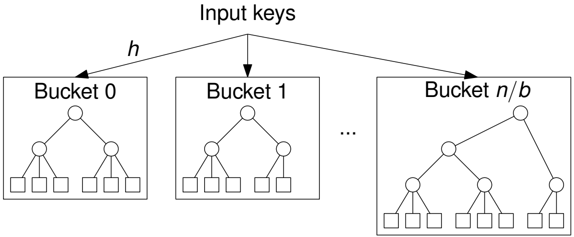


Overview


Unstructured Search
Structured Search
Brute Force
RecSplit
PHOBIC
ShockHash
SicHash
PH through Retrieval

RecSplit [EGV20, BKLS23a]

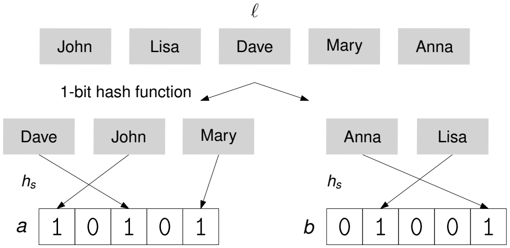
- Randomly hash keys to buckets
  Store prefix sum of bucket sizes
  using Elias-Fano coding
- Tree structure within buckets
  - Brute-force search for splitting hash function
  - Specific shape depending only on bucket size
- Small leaves
  - Brute-force search for bijection hash function
  - Practicable for $\ell \leq 16$



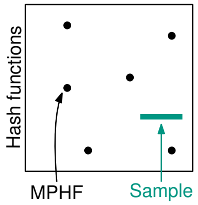
Store seeds for leaves and inner nodes (variable-bitlength, geometrically distributed).
Overall: optimal $+\mathrm{O}(1)$ bits per node.

Bijections: Rotation Fitting

Karlsruhe Institute of Technology

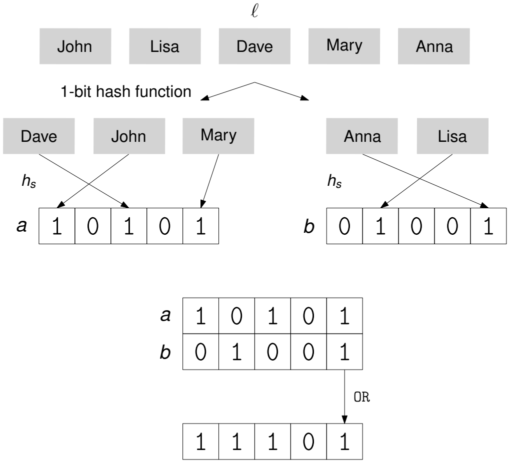


- Split keys into two subsets
- Determine function values independently
- Cyclically “rotate” word b
- Store seed and rotation $s \cdot \ell + r$
- Test $\approx \ell$ times fewer seeds
- Can use lookup tables



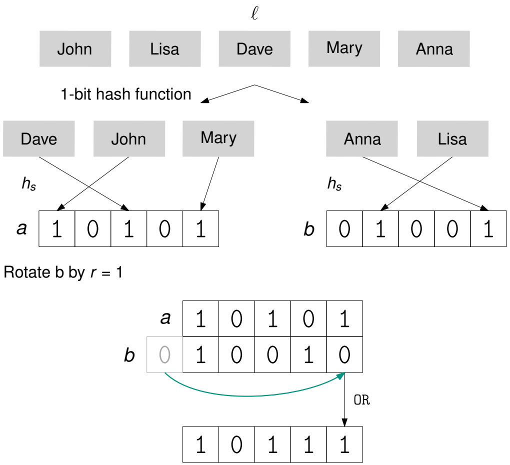
Bijections: Rotation Fitting

- Split keys into two subsets
- Determine function values independently
- Cyclically "rotate" word b
- Store seed and rotation $s \cdot \ell + r$
- Test $\approx \ell$ times fewer seeds
- Can use lookup tables



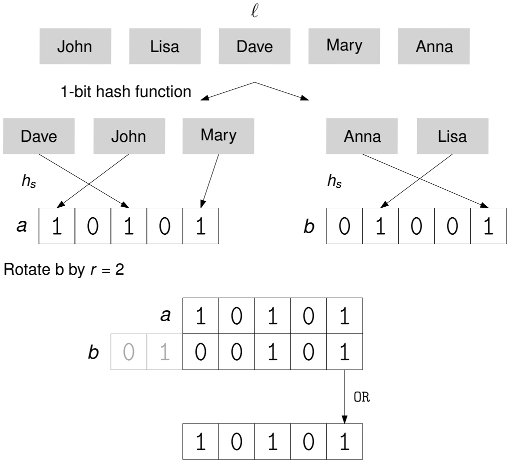
Bijections: Rotation Fitting

- Split keys into two subsets
- Determine function values independently
- Cyclically "rotate" word b
- Store seed and rotation $s \cdot \ell + r$
- Test $\approx \ell$ times fewer seeds
- Can use lookup tables



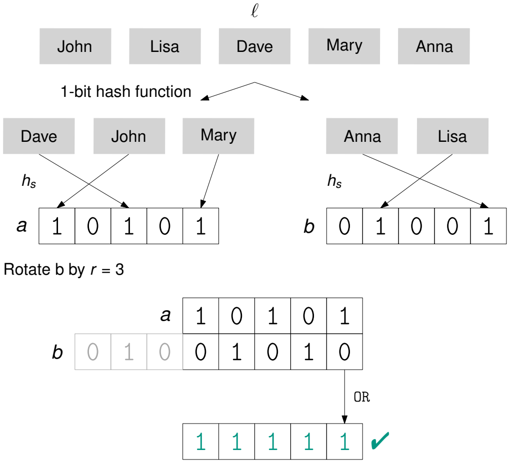
Bijections: Rotation Fitting

- Split keys into two subsets
- Determine function values independently
- Cyclically “rotate” word b
- Store seed and rotation $s \cdot \ell + r$
- Test $\approx \ell$ times fewer seeds
- Can use lookup tables



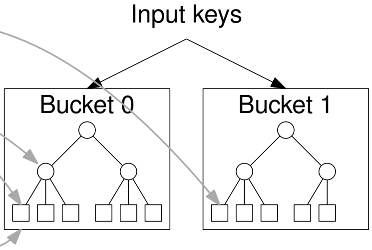
Bijections: Rotation Fitting

- Split keys into two subsets
- Determine function values independently
- Cyclically “rotate” word b
- Store seed and rotation $s \cdot \ell + r$
- Test $\approx \ell$ times fewer seeds
- Can use lookup tables



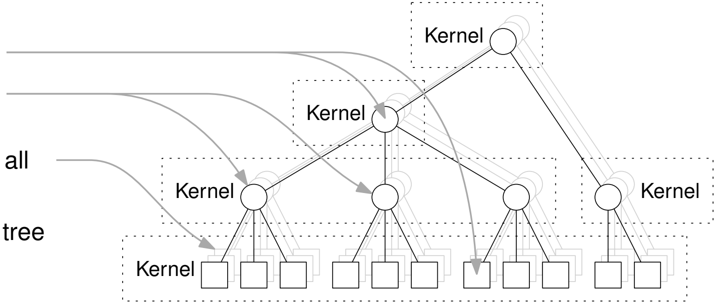
CPU Parallelization

- Bit parallelism
  - Bit operations rotate all keys of a leaf
- SIMD parallelism
  - Each lane tries a different hash function seed
- Multi-Threaded parallelism
  - Calculate different buckets in parallel

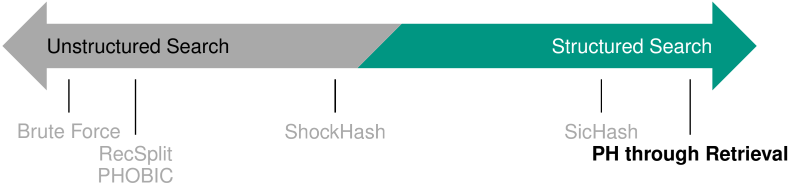


GPU Parallelization

- Threads try different seeds
- Groups of threads work on different tree nodes
- 2D grid of groups to calculate all trees with same shape
- Streams to calculate different tree shapes in parallel

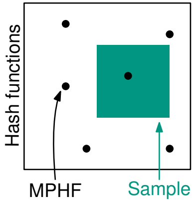


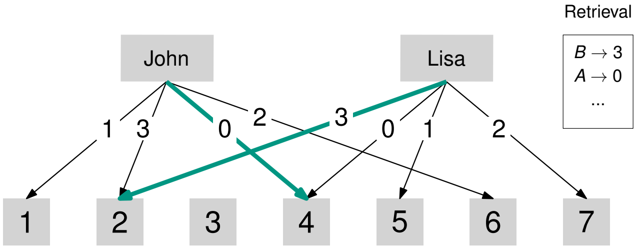
Overview


Unstructured Search
Structured Search
Brute Force
RecSplit
PHOBIC
ShockHash
SicHash
PH through Retrieval

Perfect Hashing by Retrieval [DHSW22a]

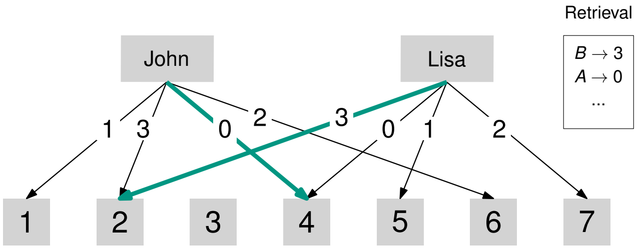
- Each object has 2^k choices
- Find collision-free mapping through perfect matching or cuckoo hashing
- Store static function $S \to \{0,1\}^k$ in retrieval data structure
- Space: $kn + o(n)$ bits



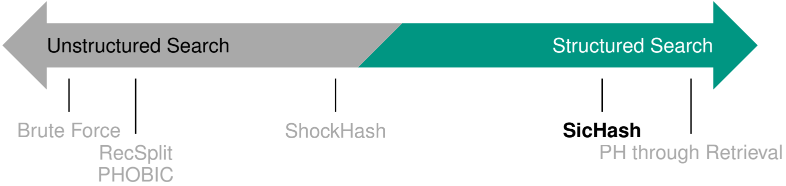


Perfect Hashing by Retrieval [DHSW22a]

- $k = 1$ bits: yields PHF with range $1..2n$
- $k = 2$ bits: yields PHF with range $1..1.024n$
  can be modified to an MPHF [PT21].
  Overall: $\approx$ 2.15 **bits per key**



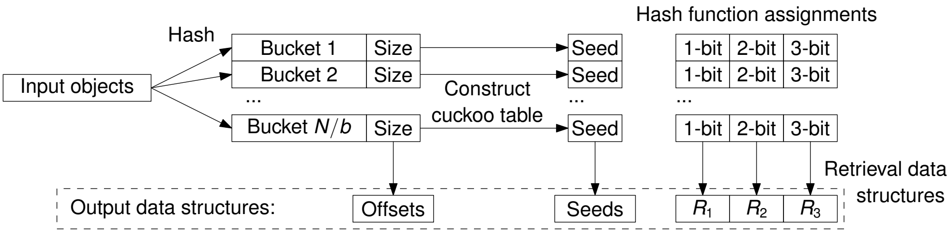
Overview


Unstructured Search
Structured Search
Brute Force
RecSplit
PHOBIC
ShockHash
SicHash
PH through Retrieval

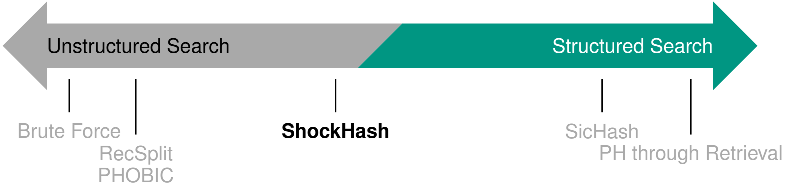
SicHash

Karlsruhe Institute of Technology

- Mix of 1/2/3-bit retrieval [DGM$^+$10] + Partitioning + Retries
- Around **2 bits per key** for MPHF



Overview



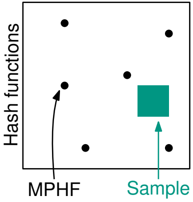
Unstructured Search
Structured Search
Brute Force
RecSplit
PHOBIC
ShockHash
SicHash
PH through Retrieval

ShockHash

- Hybrid between RecSplit and 1-bit SicHash-MPHF
- Sample random graphs
- Store choice between two candidates [DHSW22a, LSW23c]

- Problem: Unlikely to work for $> n/2$ edges [PR04], here we use n
- **ShockHash**: Do it anyway, try **many** seeds

- Orientability check?
  Success probability?
  Space usage?



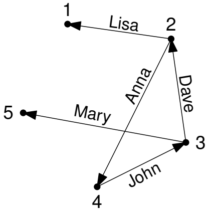
ShockHash

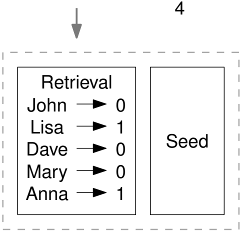
- Hybrid between RecSplit and 1-bit SicHash-MPHF
- Sample random graphs
- Store choice between two candidates [DHSW22a, LSW23c]

- Problem: Unlikely to work for $> n/2$ edges [PR04], here we use n
- **ShockHash**: Do it anyway, try **many** seeds

- Orientability check?
  Success probability?
  Space usage?

x	$h_0(x)$	$h_1(x)$
John	③	4
Lisa	2	①
Dave	②	3
Mary	⑤	3
Anna	2	④

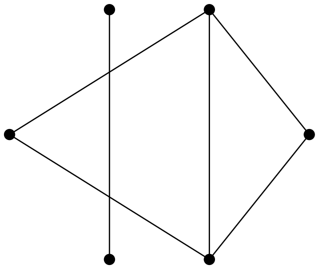




ShockHash data structure

Orientability Check

- 1-orientable if each component contains no more edges than nodes
  - Here: Tree with one additional edge
- Can be checked in linear time using connected components algorithms
- $\Rightarrow$ We check the 2^n different states of the retrieval data structure in linear time



Success Probability

$$\mathbb{P} \geq \frac{}{n^{2n}}$$

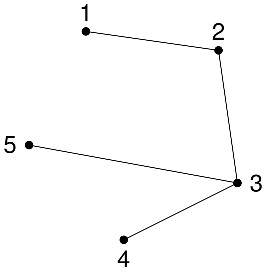
x	$h_0(x)$	$h_1(x)$
John	?	?
Lisa	?	?
Dave	?	?
Mary	?	?
Anna	?	?

Success Probability

- Labeled trees
  (Cayley's formula)

$$\mathbb{P} \geq \frac{n^{n-2}}{n^{2n}}$$

x	$h_0(x)$	$h_1(x)$
John	3	4
Lisa	2	1
Dave	2	3
Mary	5	3
Anna	?	?

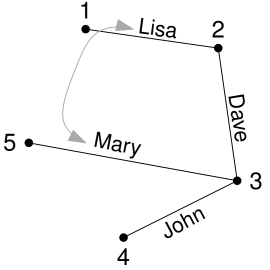


Success Probability

- Labeled trees (Cayley's formula)
- Table rows can be in any order

$$\mathbb{P} \geq \frac{n^{n-2} \; (n-1)!}{n^{2n}}$$

x	$h_0(x)$	$h_1(x)$
John	3	4
Lisa	2	1
Dave	2	3
Mary	5	3
Anna	?	?

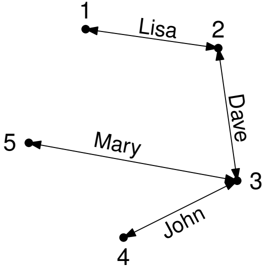


Success Probability

- Labeled trees (Cayley's formula)
- Table rows can be in any order
- Hash values can be in any order

$$\mathbb{P} \geq \frac{n^{n-2}\ (n-1)!\ 2^{n-1}}{n^{2n}}$$

x	$h_0(x)$	$h_1(x)$
John	3	4
Lisa	2	1
Dave	2	3
Mary	5	3
Anna	?	?

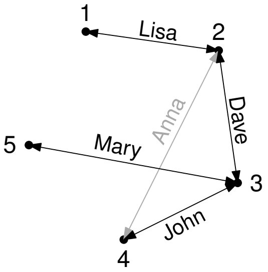


Success Probability

- Labeled trees (Cayley's formula)
- Table rows can be in any order
- Hash values can be in any order
- Last edge can be anything

$$\mathbb{P} \geq \frac{n^{n-2} \; (n-1)! \; 2^{n-1} \; n^2}{n^{2n}}$$

x	$h_0(x)$	$h_1(x)$
John	3	4
Lisa	2	1
Dave	2	3
Mary	5	3
Anna	?	?



Success Probability

Karlsruhe Institute of Technology

- Labeled trees (Cayley's formula)
- Table rows can be in any order
- Hash values can be in any order
- Last edge can be anything

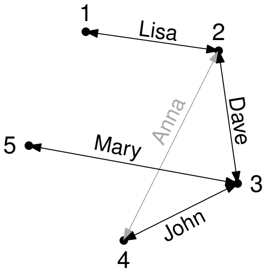
$$\mathbb{P} \geq \frac{n^{n-2} \; (n-1)! \; 2^{n-1} \; n^2}{n^{2n}}$$

$$= \frac{n!}{n^n} \cdot \frac{2^{n-1}}{n}$$

Brute force

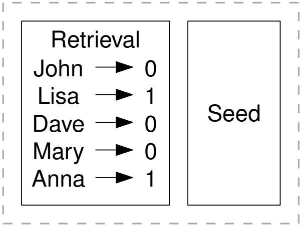
Almost 2^n times higher probability

x	$h_0(x)$	$h_1(x)$
John	3	4
Lisa	2	1
Dave	2	3
Mary	5	3
Anna	?	?



Space Usage

- $n + o(n)$ bits for 1-bit retrieval. Practice: $1.007n$ bits using BuRR [DHSW22b]
- Expected space for seed: $\approx \log\left(\frac{n}{2^{n-1}} \cdot \frac{n^n}{n!}\right) \approx n \log e - n$ bits
- Together: $\approx n \log e$ bits (optimal!)

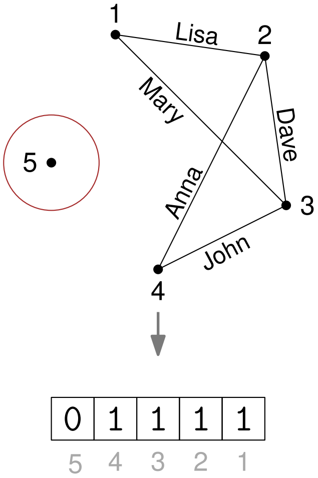


ShockHash data structure

Similar space as brute-force but nearly 2^n times faster!

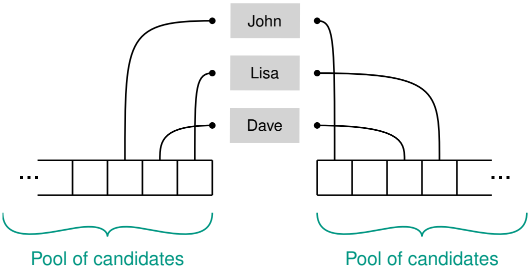
Filtering

- First implementation dominated by orientability check
  $\Rightarrow$ Filter seeds that can't work
- Efficiently in registers
- Filter passed with probability only 0.84^n
- Main ingredient for making ShockHash feasible in practice



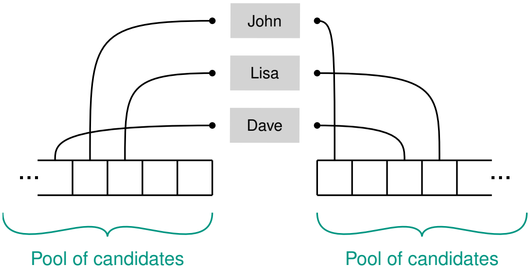
Bipartite ShockHash

- Extension of ShockHash
- Test all combinations from a set of seed candidates
- Filter halves before combining them
- Brute-force: $e^n \approx 2.72^n$
  $\Rightarrow$ ShockHash: 1.36^n
  $\Rightarrow$ Bipartite ShockHash: 1.166^n



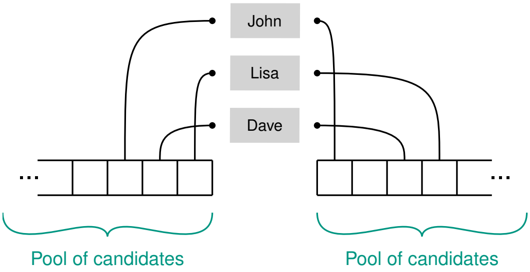
Bipartite ShockHash

- Extension of ShockHash
- Test all combinations from a set of seed candidates
- Filter halves before combining them
- Brute-force: $e^n \approx 2.72^n$
  $\Rightarrow$ ShockHash: 1.36^n
  $\Rightarrow$ Bipartite ShockHash: 1.166^n



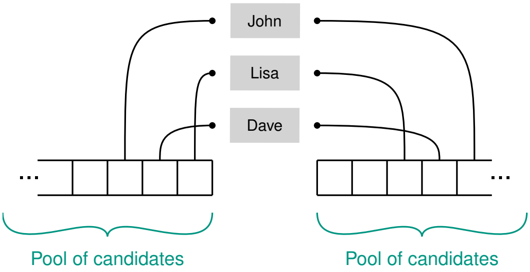
Bipartite ShockHash

- Extension of ShockHash
- Test all combinations from a set of seed candidates
- Filter halves before combining them
- Brute-force: $e^n \approx 2.72^n$
  $\Rightarrow$ ShockHash: 1.36^n
  $\Rightarrow$ Bipartite ShockHash: 1.166^n



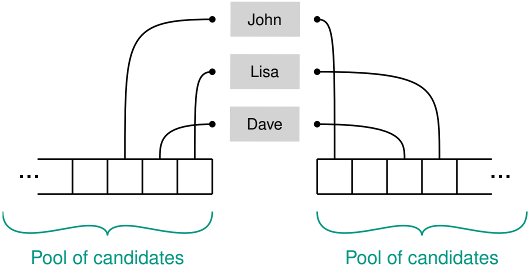
Bipartite ShockHash

- Extension of ShockHash
- Test all combinations from a set of seed candidates
- Filter halves before combining them
- Brute-force: $e^n \approx 2.72^n$
  $\Rightarrow$ ShockHash: 1.36^n
  $\Rightarrow$ Bipartite ShockHash: 1.166^n



Bipartite ShockHash

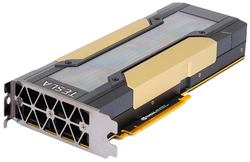
- Extension of ShockHash
- Test all combinations from a set of seed candidates
- Filter halves before combining them
- Brute-force: $e^n \approx 2.72^n$
  $\Rightarrow$ ShockHash: 1.36^n
  $\Rightarrow$ Bipartite ShockHash: 1.166^n



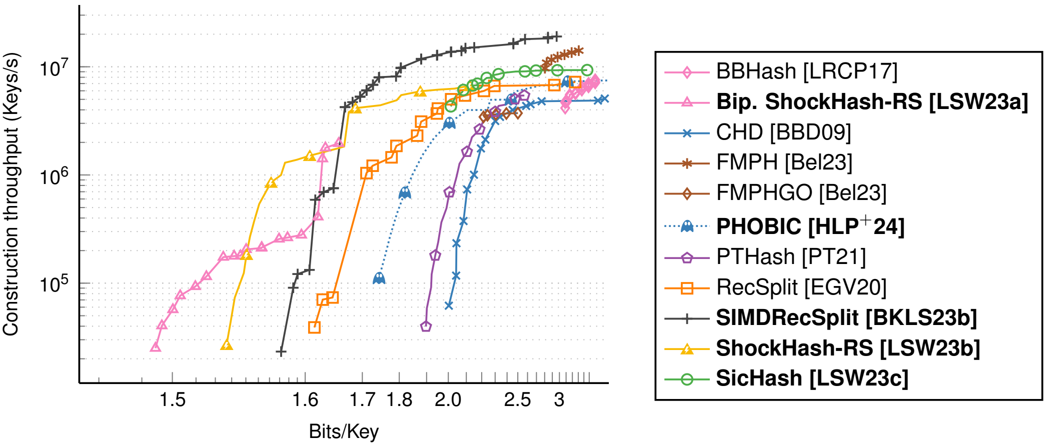
ShockHash on the GPU

Karlsruhe Institute of Technology

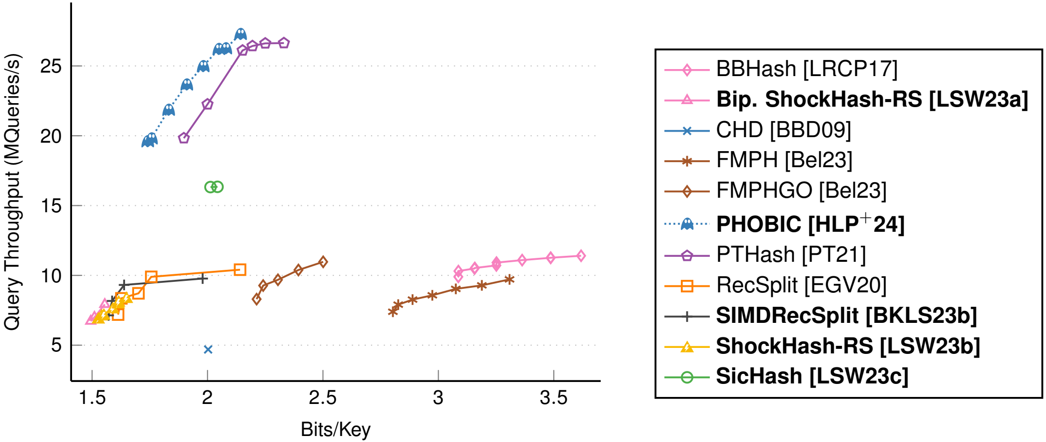
- Cuckoo hashing hard because of irregular memory access
- Filtering is easy and dominates asymptotically
- Hybrid implementation planned
  - Filtering and bit fiddling on the GPU
  - Cuckoo hashing on the CPU



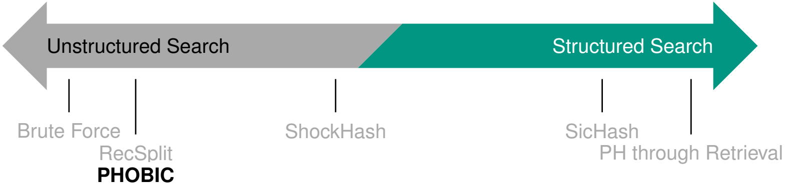
Evaluation (100M keys, single threaded)



Evaluation (100M keys, single threaded)



Overview


Unstructured Search
Structured Search
Brute Force
RecSplit
PHOBIC
ShockHash
SicHash
PH through Retrieval

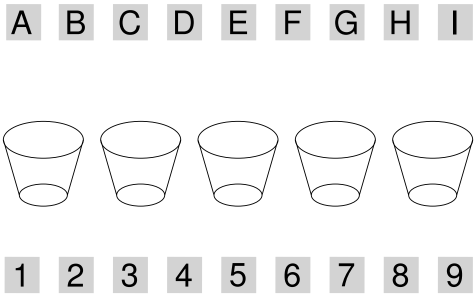
State-of-the-art [BBD09, FCH92, PT21]

A	B	C	D	E	F	G	H	I

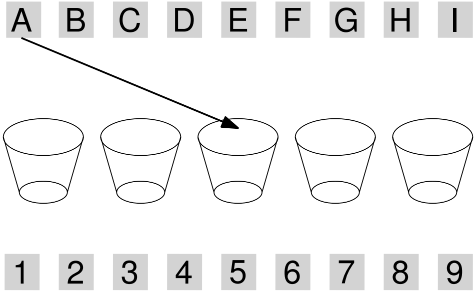
1	2	3	4	5	6	7	8	9

State-of-the-art [BBD09, FCH92, PT21]

- Choose a number of buckets proportional to the number of keys



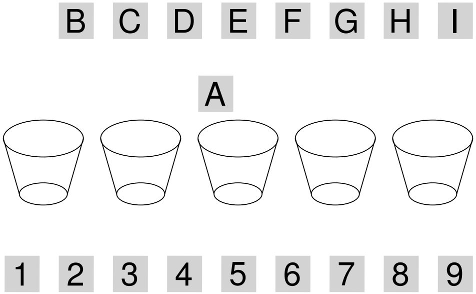
State-of-the-art [BBD09, FCH92, PT21]



- Choose a number of buckets proportional to the number of keys
- Distribute keys to buckets using random hash function

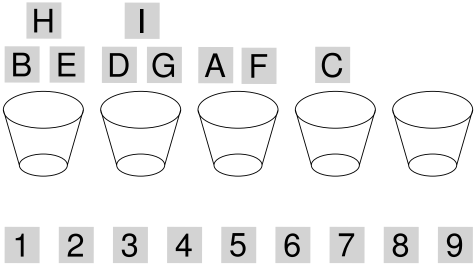
State-of-the-art [BBD09, FCH92, PT21]

- Choose a number of buckets proportional to the number of keys
- Distribute keys to buckets using random hash function



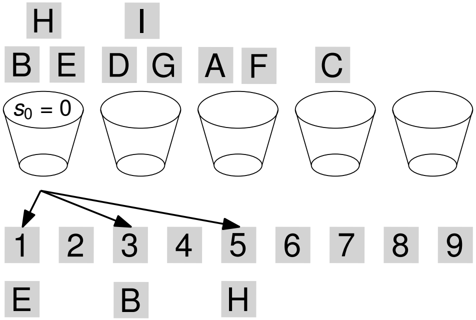
State-of-the-art [BBD09, FCH92, PT21]

- Choose a number of buckets proportional to the number of keys
- Distribute keys to buckets using random hash function

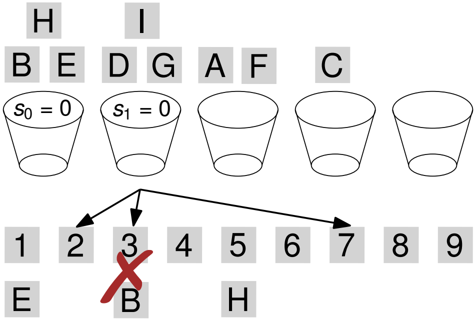


State-of-the-art [BBD09, FCH92, PT21]

- Choose a number of buckets proportional to the number of keys
- Distribute keys to buckets using random hash function
- For each bucket: Search for a seed such that there are no collisions

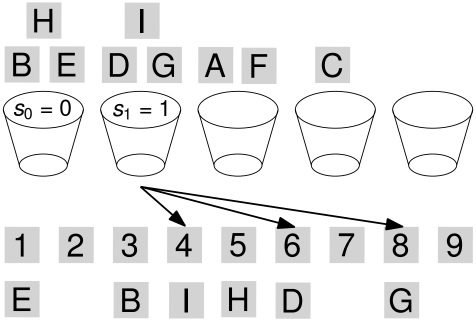


State-of-the-art [BBD09, FCH92, PT21]



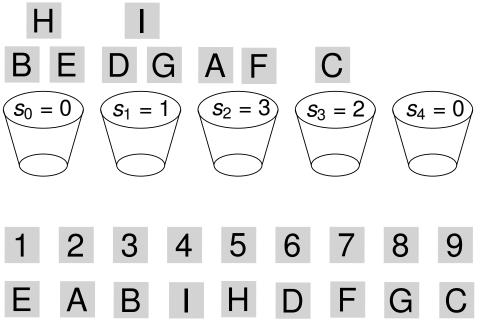
- Choose a number of buckets proportional to the number of keys
- Distribute keys to buckets using random hash function
- For each bucket: Search for a seed such that there are no collisions

State-of-the-art [BBD09, FCH92, PT21]



- Choose a number of buckets proportional to the number of keys
- Distribute keys to buckets using random hash function
- For each bucket: Search for a seed such that there are no collisions

State-of-the-art [BBD09, FCH92, PT21]

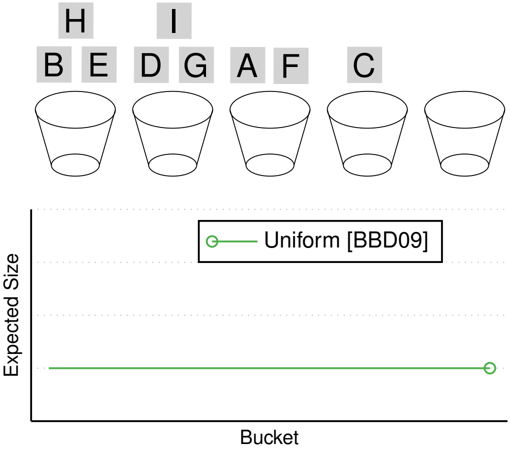


- Choose a number of buckets proportional to the number of keys
- Distribute keys to buckets using random hash function
- For each bucket: Search for a seed such that there are no collisions

State-of-the-art [BBD09, FCH92, PT21]

- The first buckets are easier to insert
- Therefore insert in non-increasing size

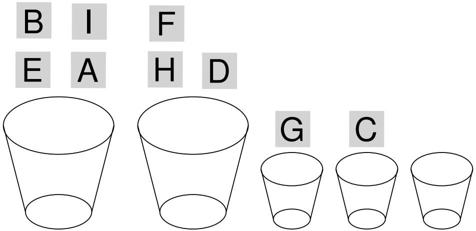
idea: buckets should have the same success probability in expectation

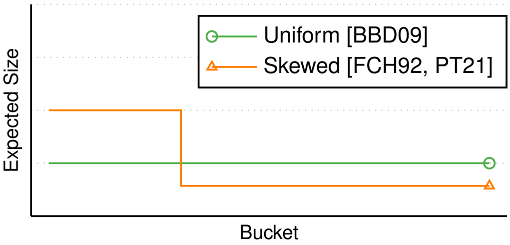


State-of-the-art [BBD09, FCH92, PT21]

- The first buckets are easier to insert
- Therefore insert in non-increasing size
- Exaggerating this effect by intentionally making some buckets even larger is helpful
- Previous state-of-the-art was to make 30% of the buckets larger

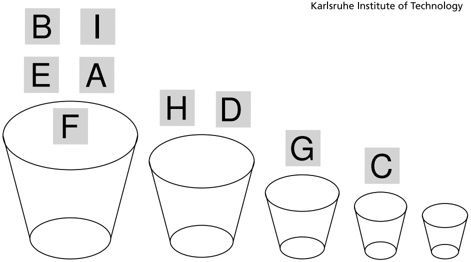
idea: buckets should have the same success probability in expectation

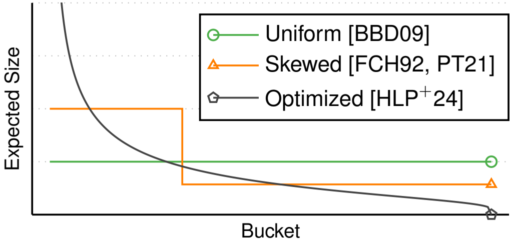




PHOBIC [HLP+24]

- The first buckets are easier to insert
- Therefore insert in non-increasing size
- Exaggerating this effect by intentionally making some buckets even larger is helpful
- Previous state-of-the-art was to make 30% of the buckets larger
- We determined optimal bucket sizes idea: buckets should have the same success probability in expectation

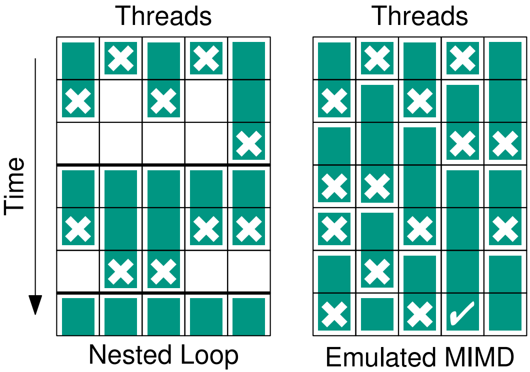




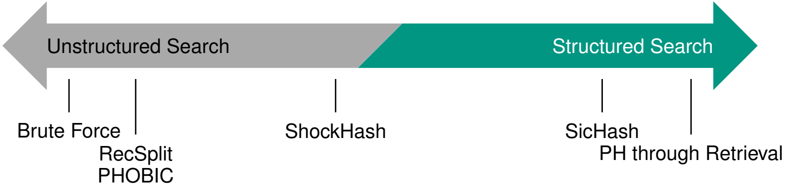
$$-\frac{1}{ln(1-x)}$$

PHOBIC on GPU

- Threads try different seeds
- Groups of threads work on different partitions
- MIMD on SIMD emulation [San94]
- 62x Speedup



Conclusion



- Tradeoff between space efficient brute-force and larger linear time algorithms
- Engineering wide range of tradeoffs
- Supported by GPUs and parallelization
- Future work
  - Combine ShockHash and PHOBIC
  - Hybrid ShockHash GPU implementation
  - How close can we get to the optimal space without construction time deteriorating?
  - More use of data structures to accelerate search
  - Hardness proofs for achieving lower bound $(+O(1))$?
  - Better performance (construction and query) when more bits are allowed (3–8)

References I

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).

[BBD09] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger.
Hash, displace, and compress.
In *ESA*, volume 5757 of *Lecture Notes in Computer Science*, pages 682–693. Springer, 2009.

[Bel23] Piotr Beling.
Fingerprinting-based minimal perfect hashing revisited.
ACM Journal of Experimental Algorithmics, 2023.

[BKLS23a] Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders.
High performance construction of RecSplit based minimal perfect hash functions.
In *ESA*, volume 274 of *LIPIcs*, pages 19:1–19:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[BKLS23b] Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders.
High performance construction of recsplit based minimal perfect hash functions.
In *ESA*, volume 274 of *LIPIcs*, pages 19:1–19:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[DGM+10] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink.
Tight thresholds for cuckoo hashing via XORSAT.
In *ICALP (1)*, volume 6198 of *Lecture Notes in Computer Science*, pages 213–225. Springer, 2010.

[DHSW22a] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer.
Fast succinct retrieval and approximate membership using ribbon.
In *SEA*, volume 233 of *LIPIcs*, pages 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

References II

[DHSW22b] Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer.
Fast succinct retrieval and approximate membership using ribbon.
In *SEA*, volume 233 of *LIPIcs*, pages 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[EGV20] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna.
RecSplit: Minimal perfect hashing via recursive splitting.
In *ALENEX*, pages 175–185. SIAM, 2020.

[FCH92] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath.
A faster algorithm for constructing minimal perfect hash functions.
In *SIGIR*, pages 266–273. ACM, 1992.

[HLP+24] Stefan Hermann, Hans-Peter Lehmann, Giulio Ermanno Pibiri, Peter Sanders, and Stefan Walzer.
Phobic: Perfect hashing with optimized bucket sizes and interleaved coding.
arXiv preprint arXiv:2404.18497, 2024.

[LRCP17] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo.
Fast and scalable minimal perfect hashing for massive key sets.
In *SEA*, volume 75 of *LIPIcs*, pages 25:1–25:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[LSW23a] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer.
Bipartite ShockHash: Pruning ShockHash search for efficient perfect hashing.
CoRR, abs/2310.14959, 2023.

[LSW23b] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer.
ShockHash: Towards optimal-space minimal perfect hashing beyond brute-force.
CoRR, abs/2308.09561, 2023.

[LSW23c] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer.
Sichash - small irregular cuckoo tables for perfect hashing.
In *ALENEX*, pages 176–189. SIAM, 2023.

References III

[PR04] Rasmus Pagh and Flemming Friche Rodler.
Cuckoo hashing.
J. Algorithms, 51(2):122–144, 2004.

[PT21] Giulio Ermanno Pibiri and Roberto Trani.
PTHash: Revisiting FCH minimal perfect hashing.
In *SIGIR*, pages 1339–1348. ACM, 2021.

[San94] P. Sanders.
Emulating MIMD behavior on SIMD machines.
In *International Conference Massively Parallel Processing Applications and Development*, pages 313–321, Delft, 1994. Elsevier.