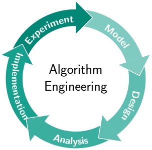
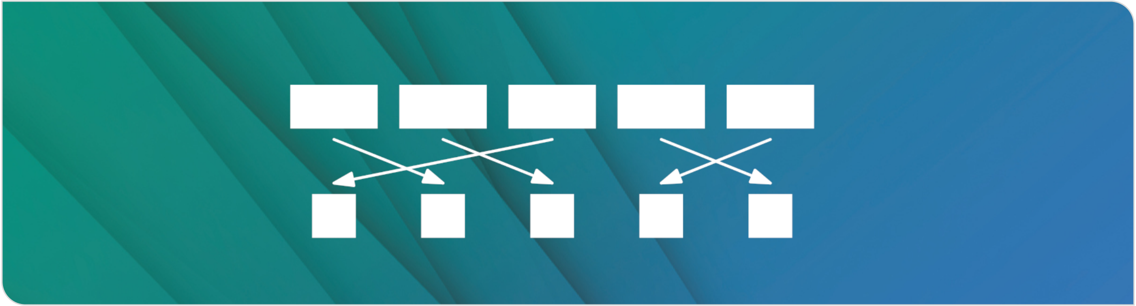
Karlsruhe Institute of Technology



Fast and Space-Efficient Perfect Hashing

Doctoral Defense

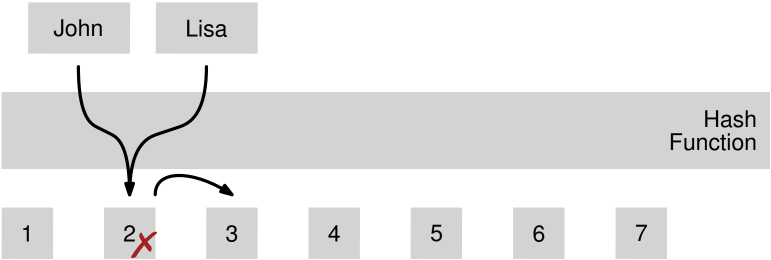
Hans-Peter Lehmann | October 24, 2024



Hash Tables

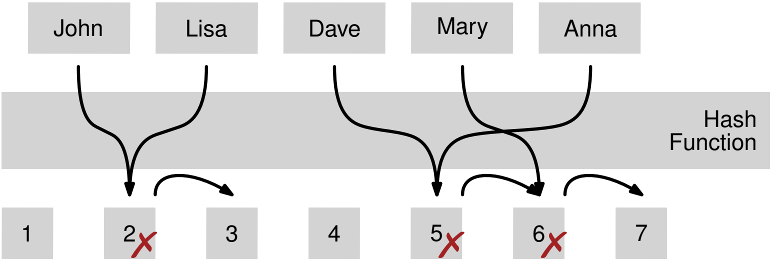
- Hash function maps input to integers
- Uniform mapping leads to ✗ collisions

Hash Tables



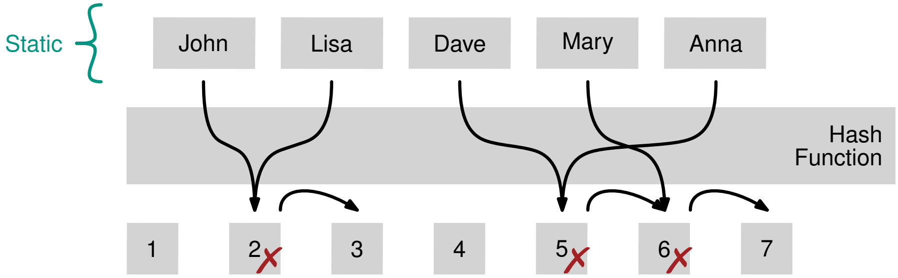
- Hash function maps input to integers
- Uniform mapping leads to ✗ collisions

Hash Tables



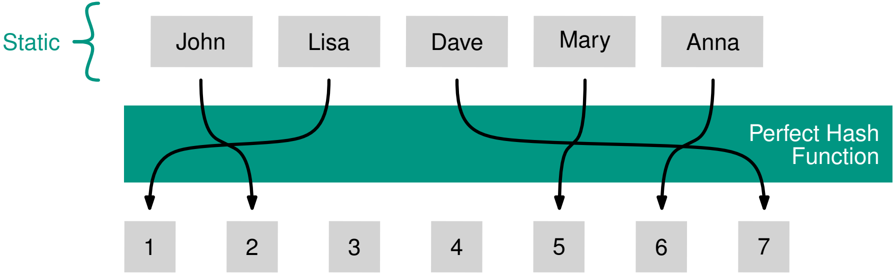
- Hash function maps input to integers
- Uniform mapping leads to ✗ collisions

Hash Tables



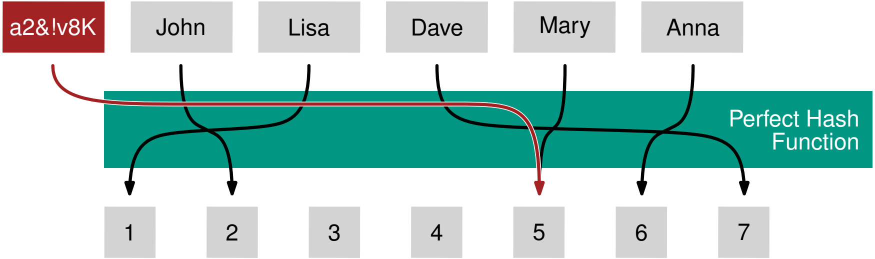
- Hash function maps input to integers
- Uniform mapping leads to ✗ collisions

Perfect Hashing



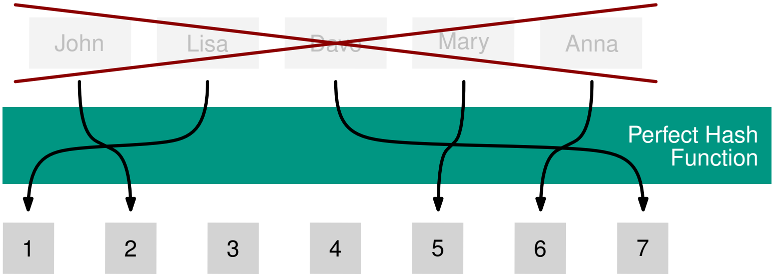
- Data structure to map keys to integers without collisions

Perfect Hashing



- Data structure to map keys to integers without collisions

Perfect Hashing



- Data structure to map keys to integers without collisions

Applications

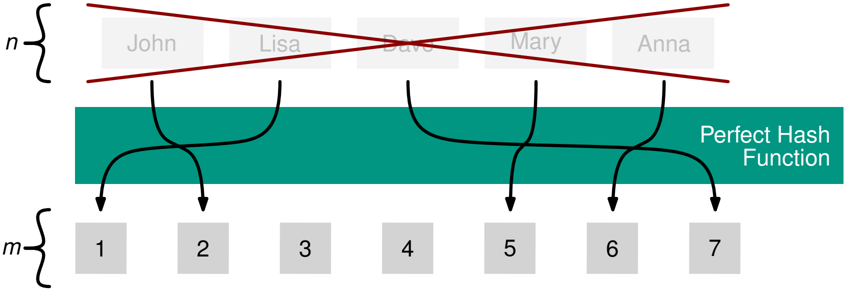
Databases
- Enums
- Updatable retrieval

Text indexing
- Alphabet reduction
- Trie navigation

Bioinformatics
- Index data
- Union-Find

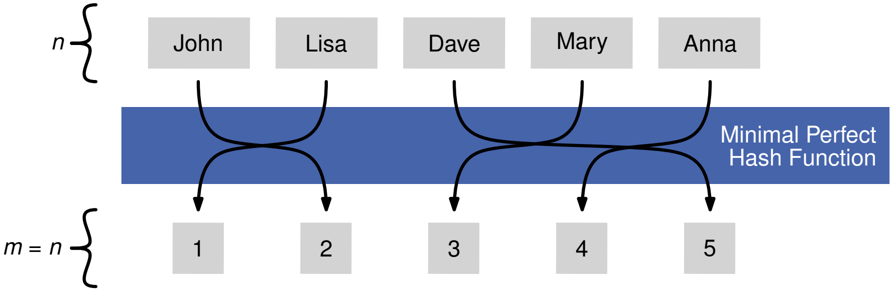
Representatives
- Handle large objects

Load Factor



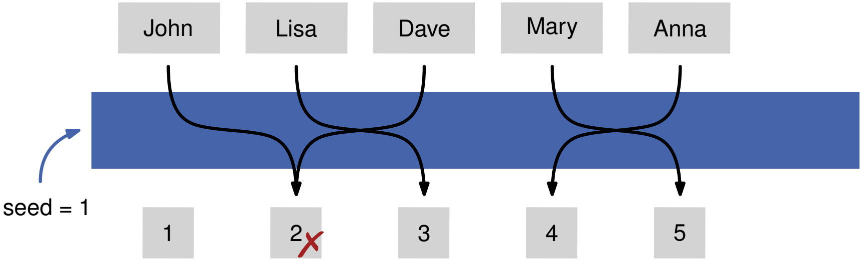
- n/m is the load factor

Minimal Perfect Hashing



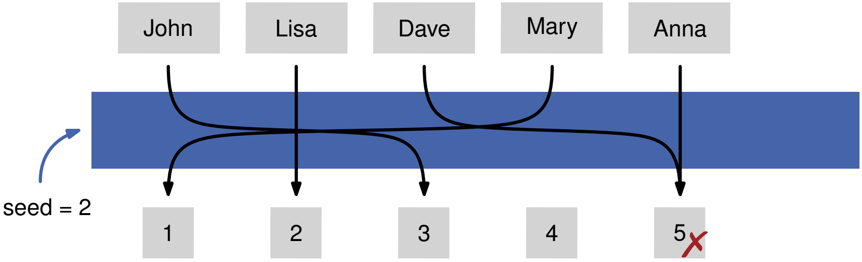
- Bijection between keys and $[n]$

Simple Brute-Force Construction [Meh84]



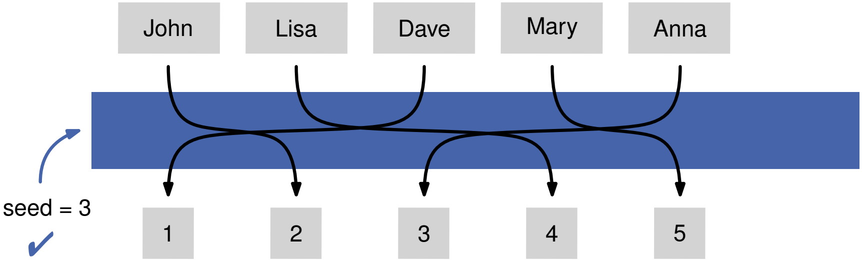
Function hash(x, seed)
  x = x ^ seed
  x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9
  x = (x ^ (x >> 27)) * 0x94d049bb133111eb
  return (x ^ (x >> 31)) mod n

Simple Brute-Force Construction [Meh84]



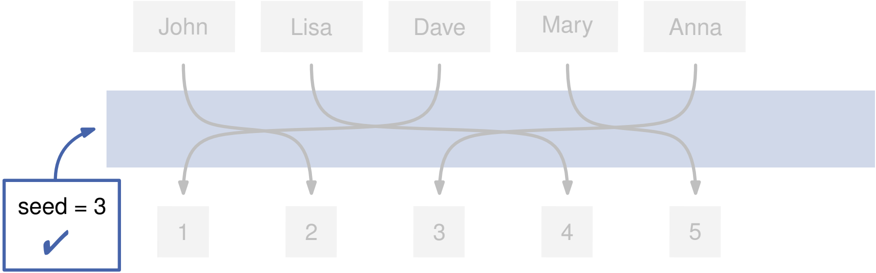
Function hash(x, seed)
x = x ^ seed
x = (x ^ (x >> 30)) ∗ 0xbf58476d1ce4e5b9
x = (x ^ (x >> 27)) ∗ 0x94d049bb133111eb
return (x ^ (x >> 31)) mod n

Simple Brute-Force Construction [Meh84]



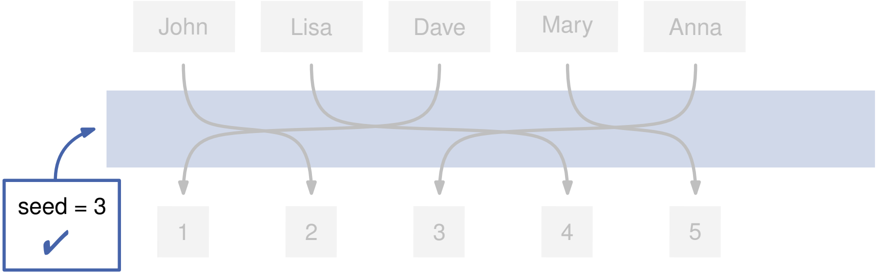

```
Function hash(x, seed)
    x = x ^ seed
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb
    return (x ^ (x >> 31)) mod n
```

Simple Brute-Force Construction [Meh84]



Function hash(x, seed)
    x = x ^ seed
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb
    return (x ^ (x >> 31)) mod n

Simple Brute-Force Construction [Meh84]




```
Function hash(x, seed)
    x = x ^ seed
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb
    return (x ^ (x >> 31)) mod n
```

- $\mathbb{E}(\text{seed}) = \frac{n^n}{n!} \approx e^n$
- $\mathbb{E}(\text{space}) \approx \log_2(e^n) \approx 1.44n$ bits

Retrieval Data Structures [DHSW22]

x	$f(x)$
John	🤓
Lisa	😀
Dave	😀
Mary	🤓

- Store static function $f : S \to \{0, 1\}^r$, arbitrary for $x \notin S$
- BuRR [DHSW22] close to *rn* bits

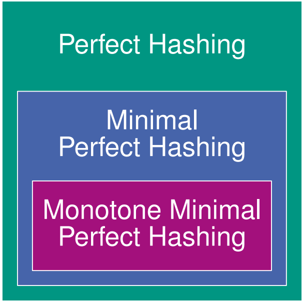
Retrieval Data Structures [DHSW22]

Karlsruhe Institute of Technology

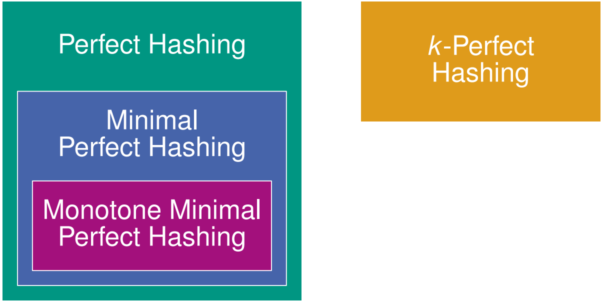
x	$f(x)$
John	🤓
Lisa	😀
Dave	😀
Mary	🤓

- Store static function $f : S \to \{0,1\}^r$, arbitrary for $x \notin S$
- BuRR [DHSW22] close to rn bits

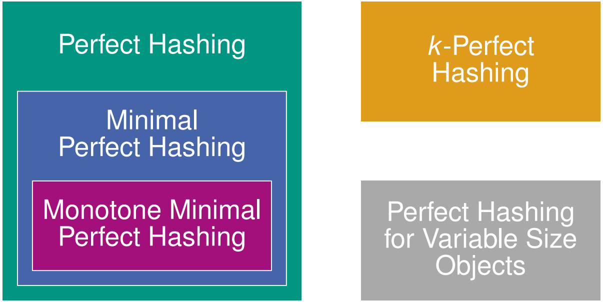
Main Results



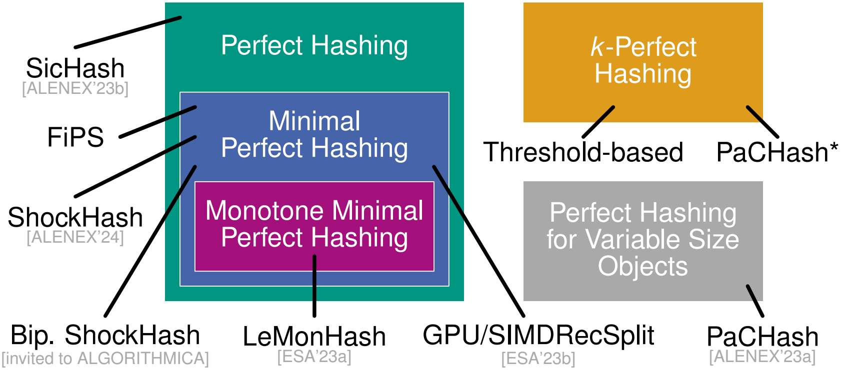
Main Results



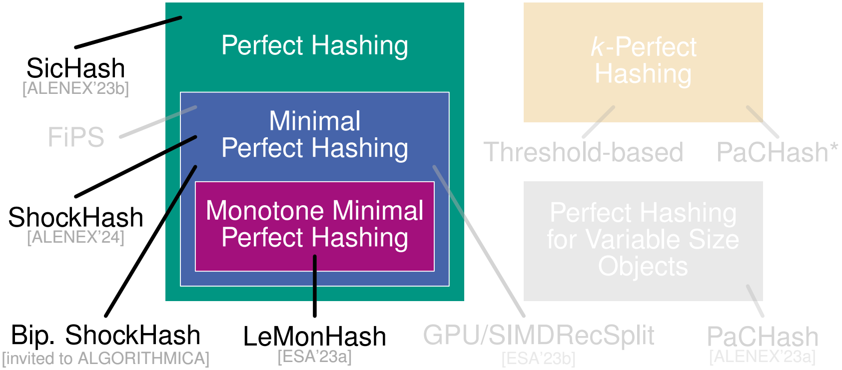
Main Results



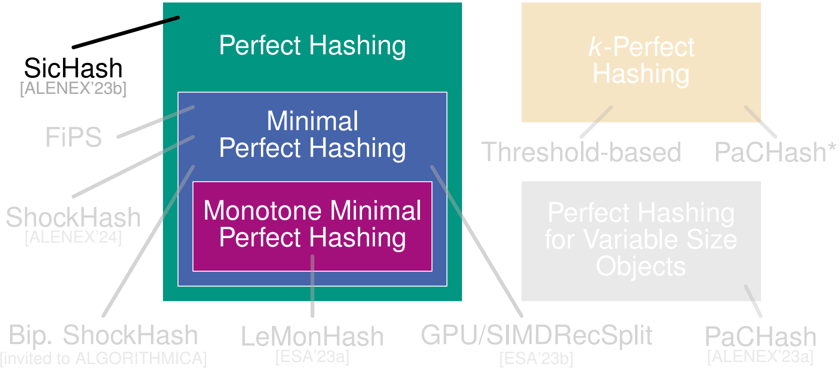
Main Results



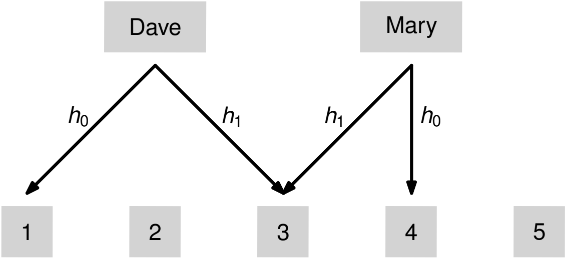
Main Results



Main Results

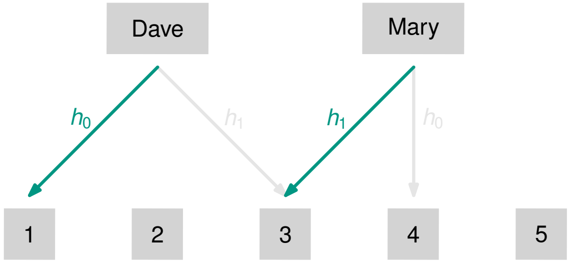


Perfect Hashing Through Retrieval [BPZ07, DHSW22]



- Store hash function index for each key in a retrieval data structure

Perfect Hashing Through Retrieval [BPZ07, DHSW22]

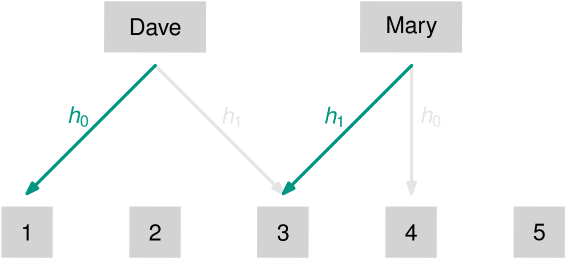


1-bit retrieval:

Dave	0
Mary	1

- Store hash function index for each key in a retrieval data structure

Perfect Hashing Through Retrieval [BPZ07, DHSW22]



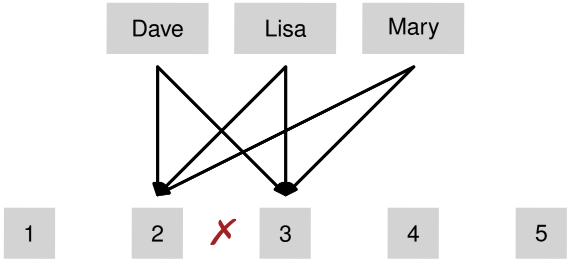
1-bit retrieval:

Dave	0
Mary	1

PHF(Mary) = h_1(Mary) = 3

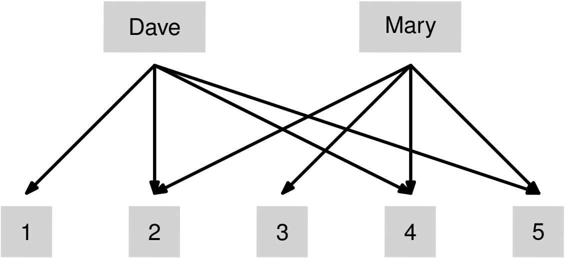
- Store hash function index for each key in a retrieval data structure

Perfect Hashing Through Retrieval [BPZ07, DHSW22]



- Store hash function index for each key in a retrieval data structure

Perfect Hashing Through Retrieval [BPZ07, DHSW22]

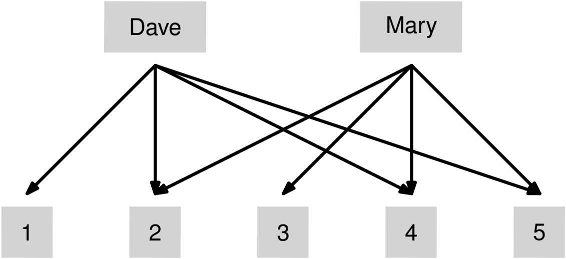


2-bit retrieval:

Dave	01
Mary	11

- Store hash function index for each key in a retrieval data structure

Perfect Hashing Through Retrieval [BPZ07, DHSW22]

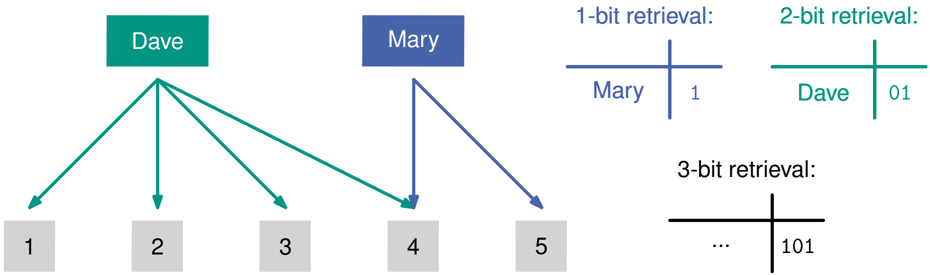


2-bit retrieval:

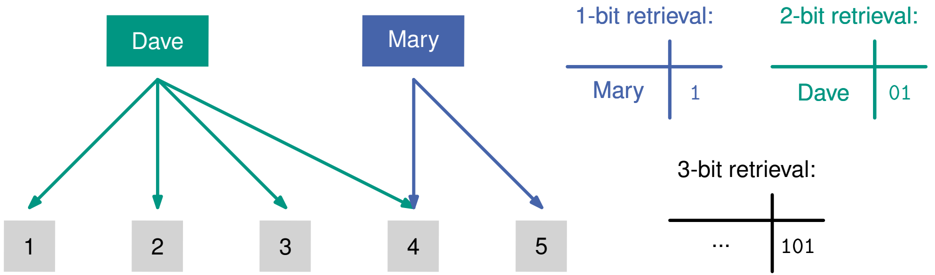
Dave	01
Mary	11

- Store hash function index for each key in a retrieval data structure
- Remap to make minimal perfect

SicHash [ALENEX'23b]

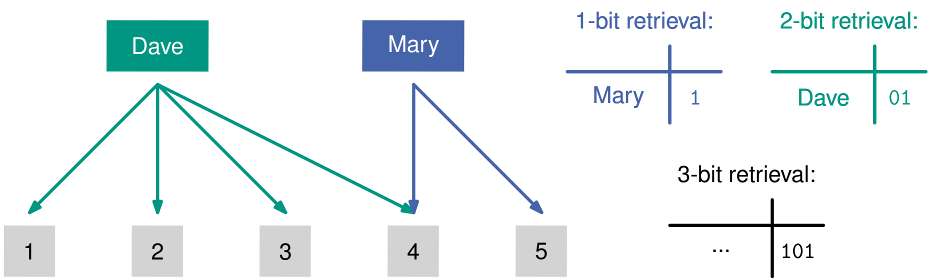


SicHash [ALENEX'23b]



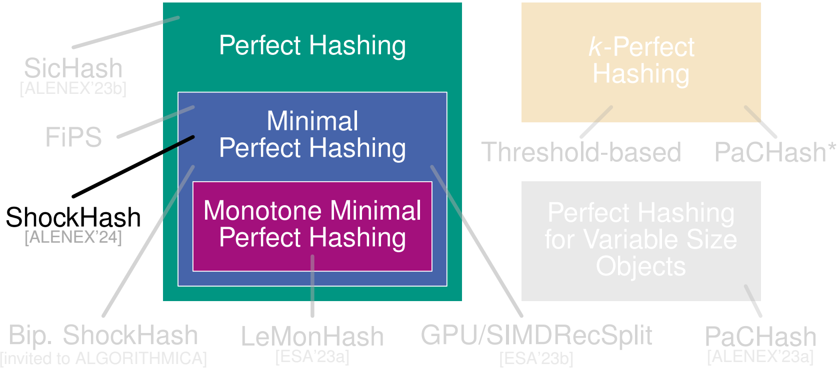
- Before (100% 2-bit): remapping takes 0.18 bits/key

SicHash [ALENEX'23b]

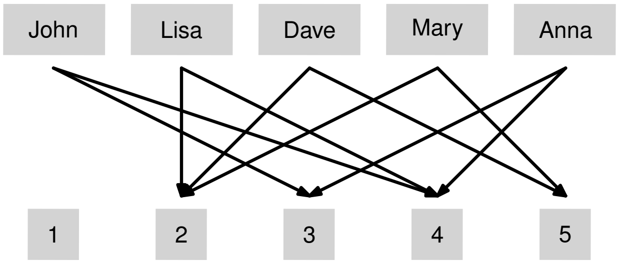


- Before (100% 2-bit): remapping takes 0.18 bits/key
- SicHash (50% 1-bit, 50% 3-bit): remapping takes 0.07 bits/key

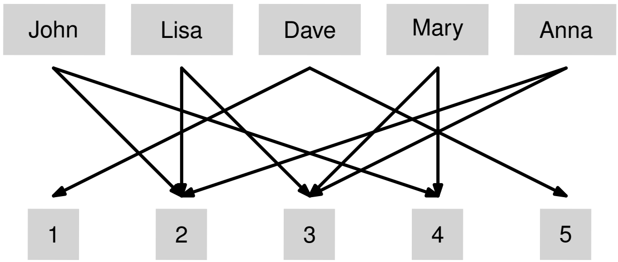
Main Results



ShockHash [ALENEX’24]



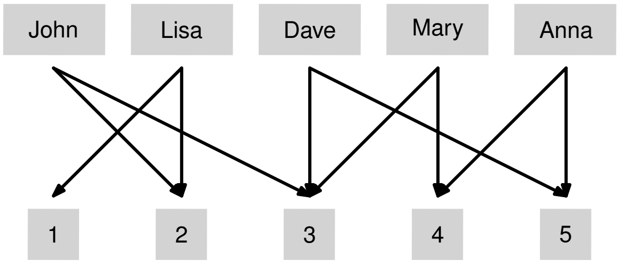
ShockHash [ALENEX'24]



seed = 2

- Retry $\approx (e/2)^n$ seeds

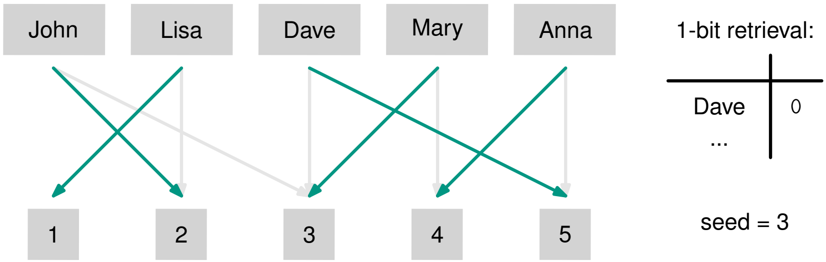
ShockHash [ALENEX'24]



seed = 3

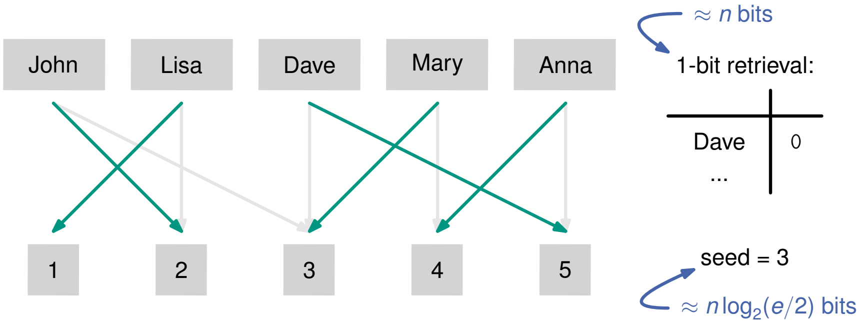
- Retry $\approx (e/2)^n$ seeds

ShockHash [ALENEX'24]



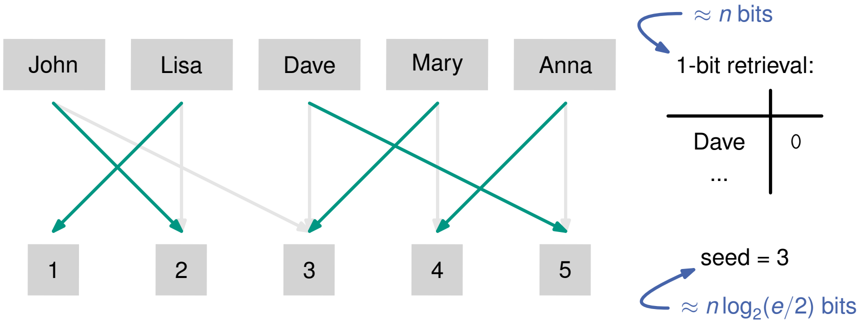
- Retry $\approx (e/2)^n$ seeds

ShockHash [ALENEX'24]



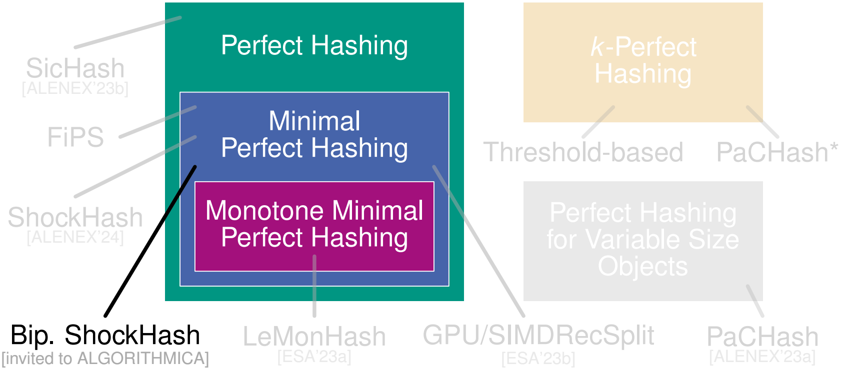
- Retry $\approx (e/2)^n$ seeds

ShockHash [ALENEX'24]

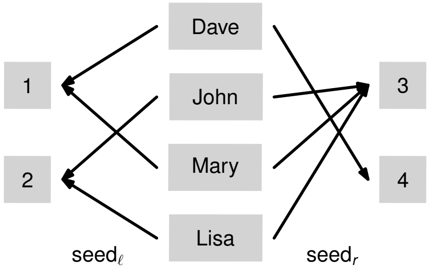


- Retry $\approx (e/2)^n$ seeds
- $\approx 2^n$ times faster than brute-force

Main Results

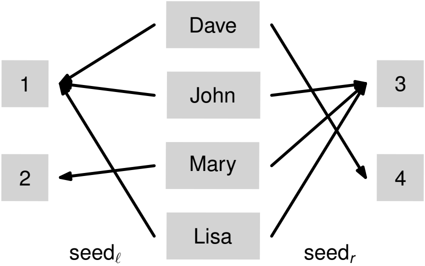


Bipartite ShockHash [invited to ALGORITHMICA]



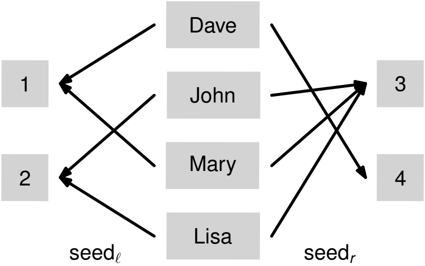
- Partition output values
- Store two seeds and retrieval data structure

Bipartite ShockHash [invited to ALGORITHMICA]



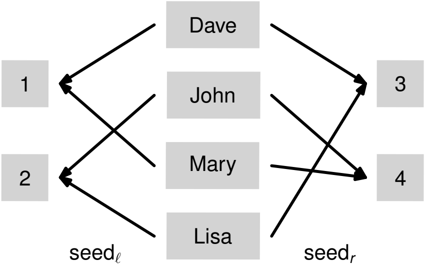
- Partition output values
- Store two seeds and retrieval data structure

Bipartite ShockHash [invited to ALGORITHMICA]



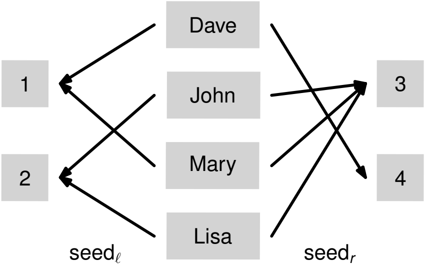
- Partition output values
- Store two seeds and retrieval data structure

Bipartite ShockHash [invited to ALGORITHMICA]



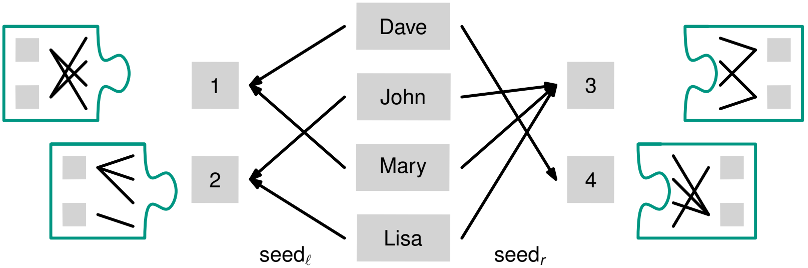
- Partition output values
- Store two seeds and retrieval data structure

Bipartite ShockHash [invited to ALGORITHMICA]



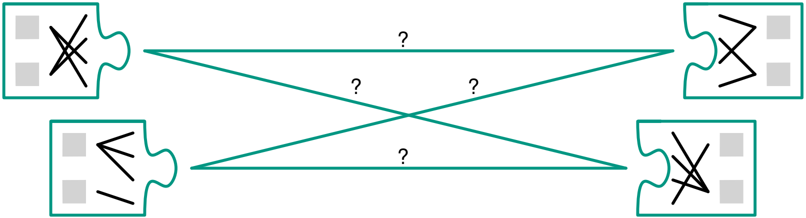
- Partition output values
- Store two seeds and retrieval data structure

Bipartite ShockHash [invited to ALGORITHMICA]



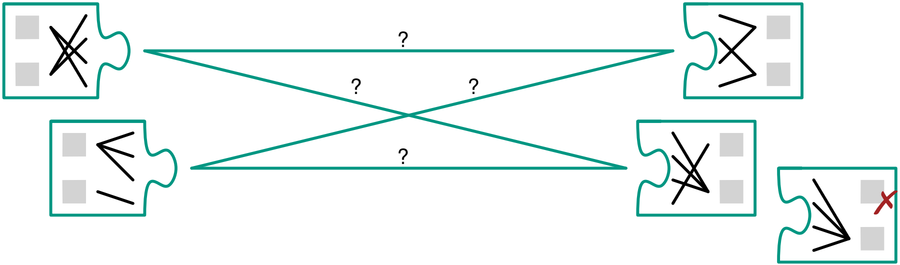
- Build pool of $\sqrt{(e/2)^n} \approx 1.165^n$ seeds

Bipartite ShockHash [invited to ALGORITHMICA]



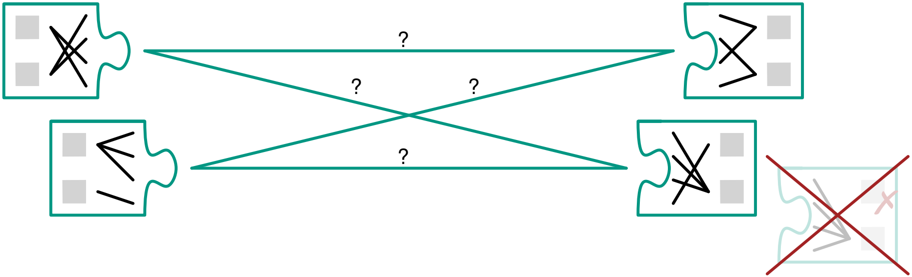
- Build pool of $\sqrt{(e/2)^n} \approx 1.165^n$ seeds

Bipartite ShockHash [invited to ALGORITHMICA]



- Build pool of $\sqrt{(e/2)^n} \approx 1.165^n$ seeds

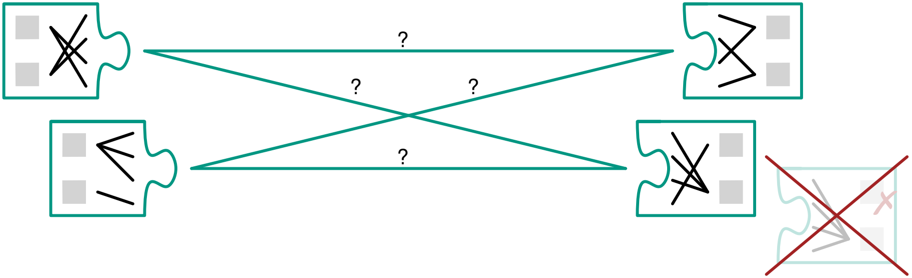
Bipartite ShockHash [invited to ALGORITHMICA]



- Build pool of $\sqrt{(e/2)^n} \approx 1.165^n$ seeds
- Filter seeds before combining
- Only $\sqrt{(e/2)^n} \cdot 0.836^{n/2}$ combinations to test

Bipartite ShockHash [invited to ALGORITHMICA]

Karlsruhe Institute of Technology



- Build pool of $\sqrt{(e/2)^n} \approx 1.165^n$ seeds
- Filter seeds before combining
- Only $\left(\sqrt{(e/2)^n} \cdot 0.836^{n/2}\right)^2 \approx 1.136^n$ combinations to test

Brute-Force Techniques

Simple Brute-Force: [Meh84]
$e^n \approx 2.718^n$ seeds

Brute-Force Techniques

Karlsruhe Institute of Technology

ShockHash:
$(e/2)^n \approx 1.359^n$ seeds

Simple Brute-Force: [Meh84]
$e^n \approx 2.718^n$ seeds

Brute-Force Techniques

Karlsruhe Institute of Technology

Bipartite ShockHash:
Pool of $\sqrt{(e/2)^n} \approx 1.165^n$ seeds

ShockHash:
$(e/2)^n \approx 1.359^n$ seeds

Simple Brute-Force: [Meh84]
$e^n \approx 2.718^n$ seeds

Brute-Force Techniques

Karlsruhe Institute of Technology

4 milliseconds

Bipartite ShockHash:
Pool of $\sqrt{(e/2)^n} \approx 1.165^n$ seeds

ShockHash:
$(e/2)^n \approx 1.359^n$ seeds

Simple Brute-Force: [Meh84]
$e^n \approx 2.718^n$ seeds

Brute-Force Techniques

4 milliseconds

Bipartite ShockHash:
Pool of $\sqrt{(e/2)^n} \approx 1.165^n$ seeds

6 hours

ShockHash:
$(e/2)^n \approx 1.359^n$ seeds

Simple Brute-Force: [Meh84]
$e^n \approx 2.718^n$ seeds

Brute-Force Techniques

Karlsruhe Institute of Technology

4 milliseconds

Bipartite ShockHash:
Pool of $\sqrt{(e/2)^n} \approx 1.165^n$ seeds

6 hours

ShockHash:
$(e/2)^n \approx 1.359^n$ seeds

$10^{17} \times$ age of universe

Simple Brute-Force: [Meh84]
$e^n \approx 2.718^n$ seeds

Brute-Force Techniques

4 milliseconds

Bipartite ShockHash:
Pool of $\sqrt{(e/2)^n} \approx 1.165^n$ seeds

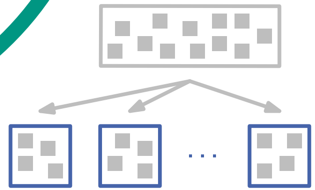
6 hours

ShockHash:
$(e/2)^n \approx 1.359^n$ seeds

$10^{17} \times$ age of universe

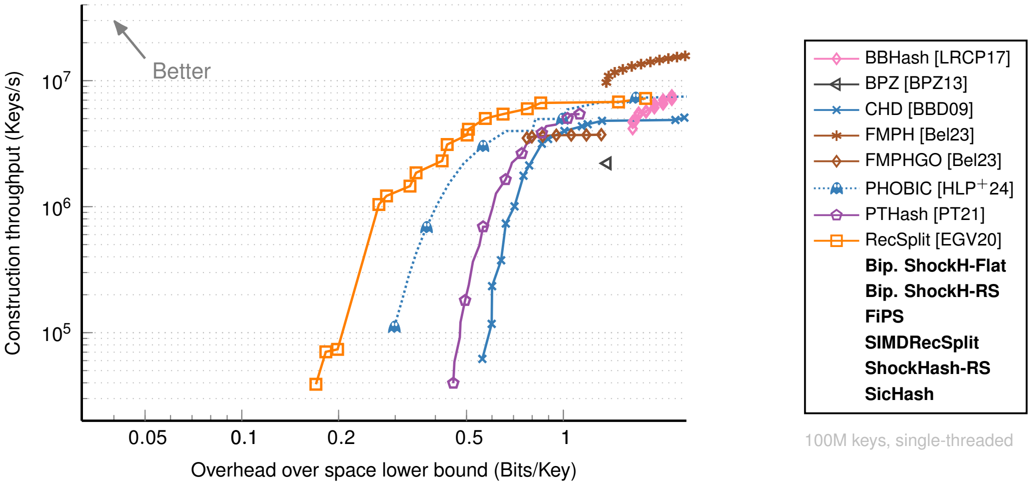
Simple Brute-Force: [Meh84]
$e^n \approx 2.718^n$ seeds

Practice:



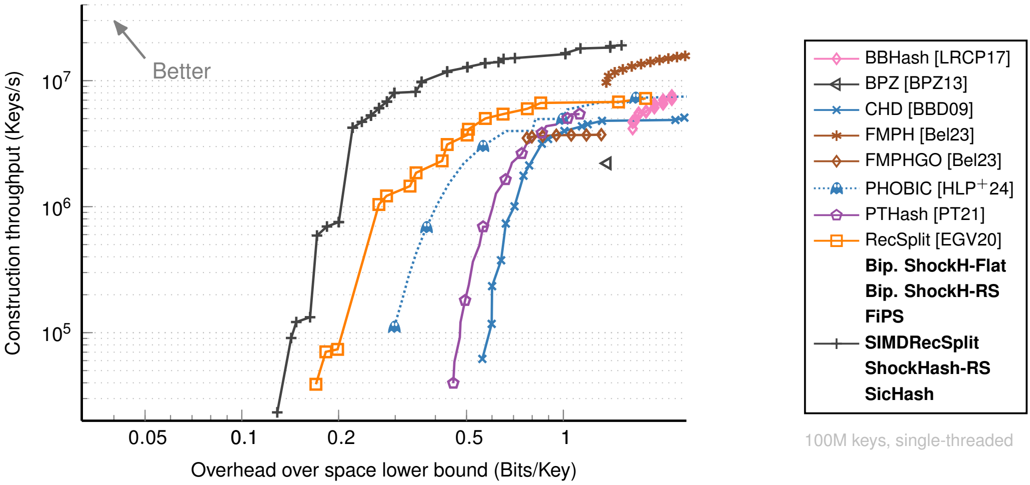
Base-case: Brute-Force

Experiments: Construction



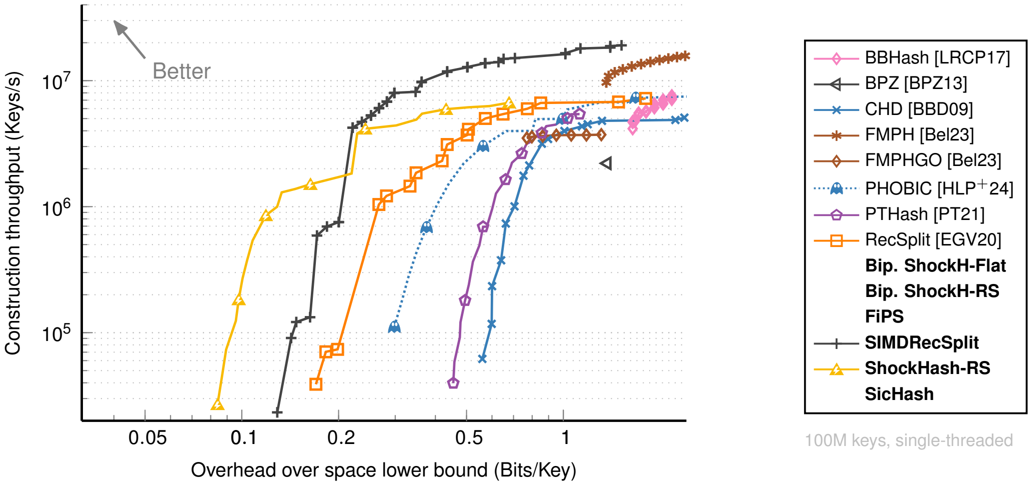
100M keys, single-threaded

Experiments: Construction



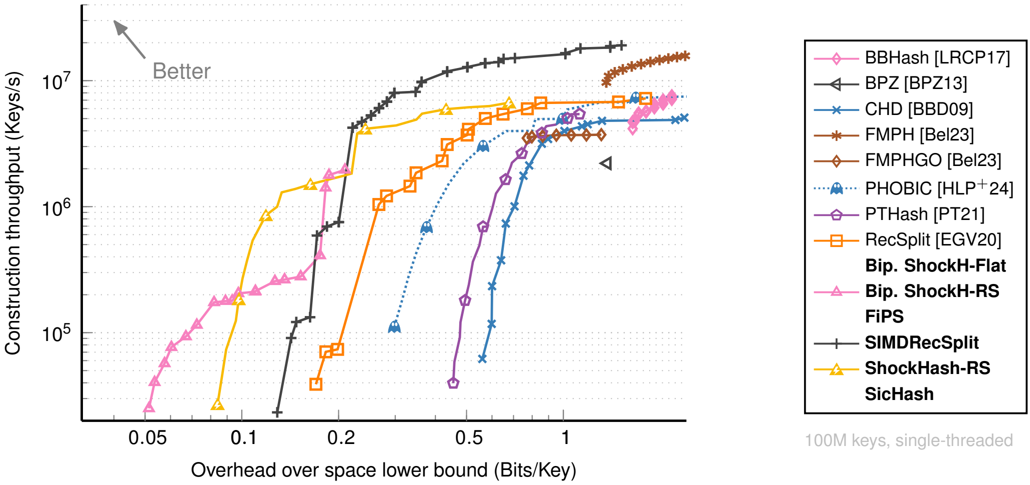
100M keys, single-threaded

Experiments: Construction



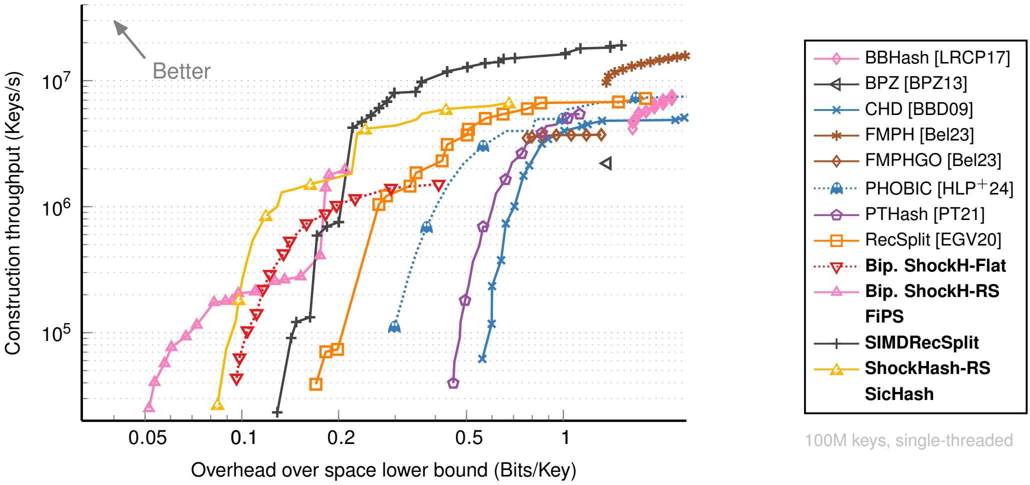
100M keys, single-threaded

Experiments: Construction

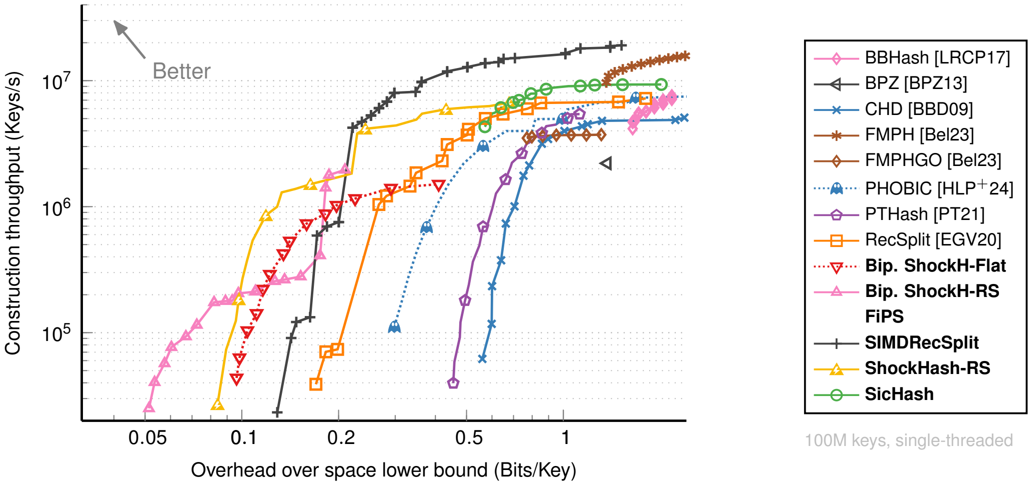


100M keys, single-threaded

Experiments: Construction

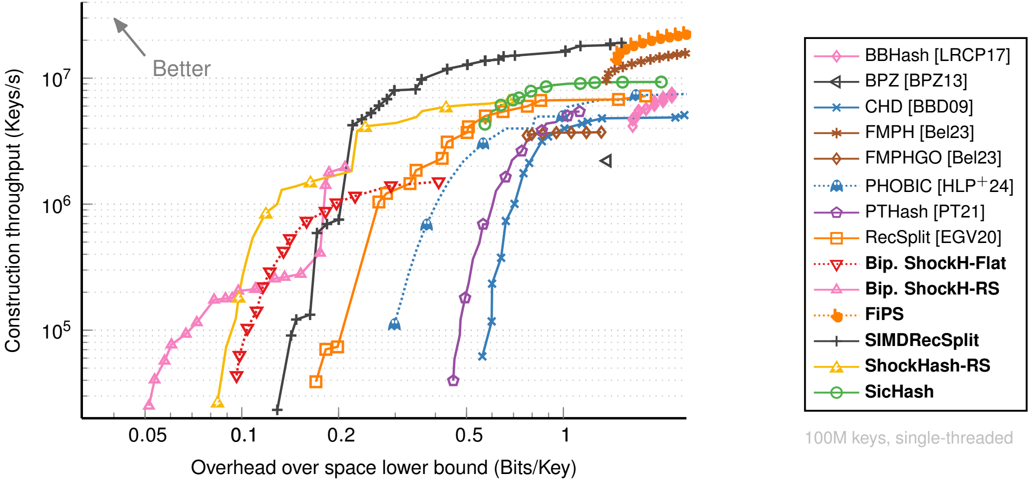


Experiments: Construction



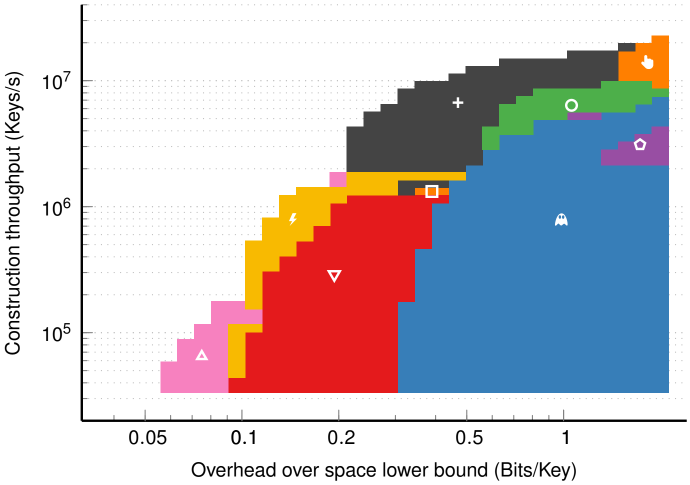
100M keys, single-threaded

Experiments: Construction



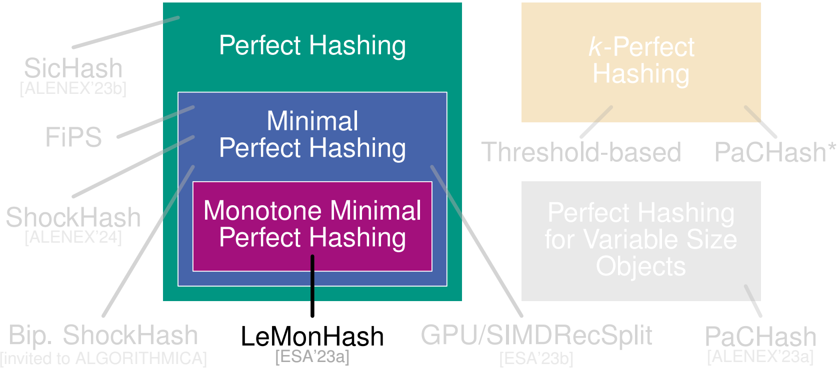
100M keys, single-threaded

Experiments: Queries

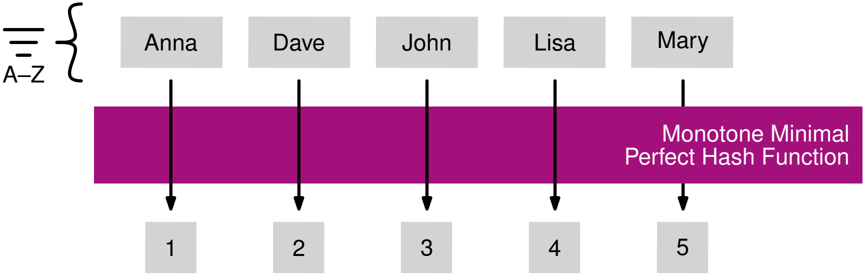


100M keys, single-threaded

Main Results

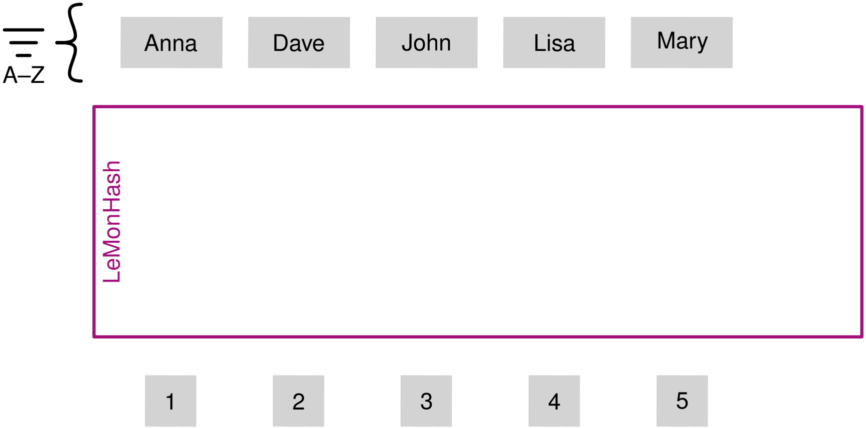


Monotone Minimal Perfect Hashing

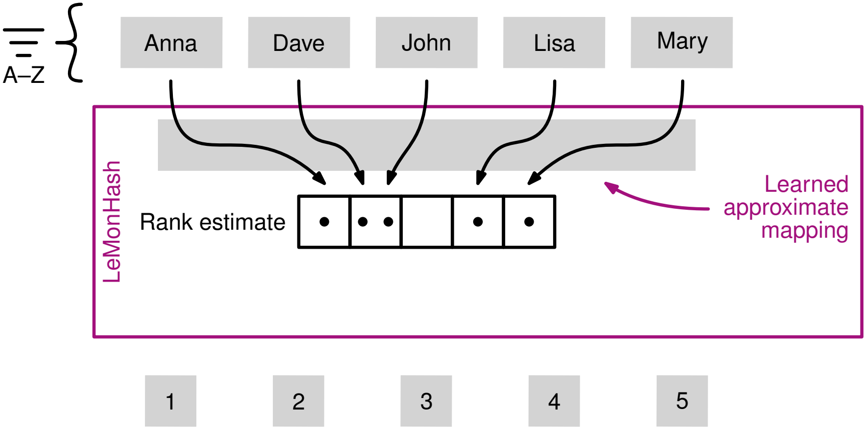


- Retain natural order of the input keys
- Rank queries

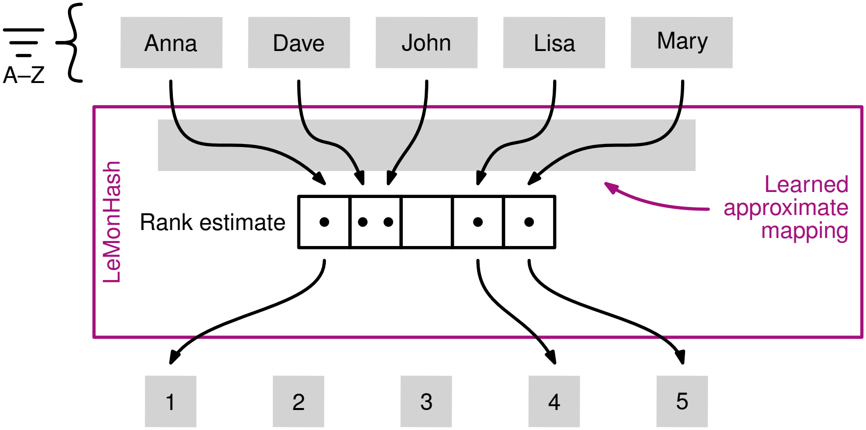
LeMonHash [ESA'23a]



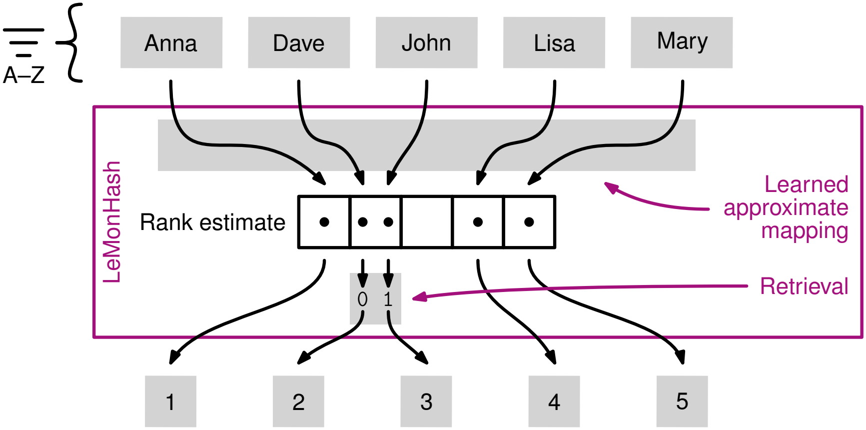
LeMonHash [ESA'23a]



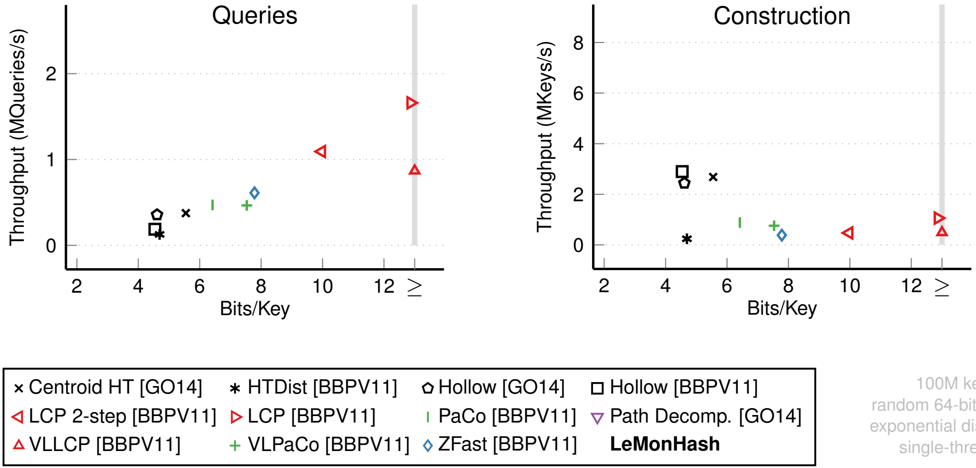
LeMonHash [ESA'23a]



LeMonHash [ESA’23a]

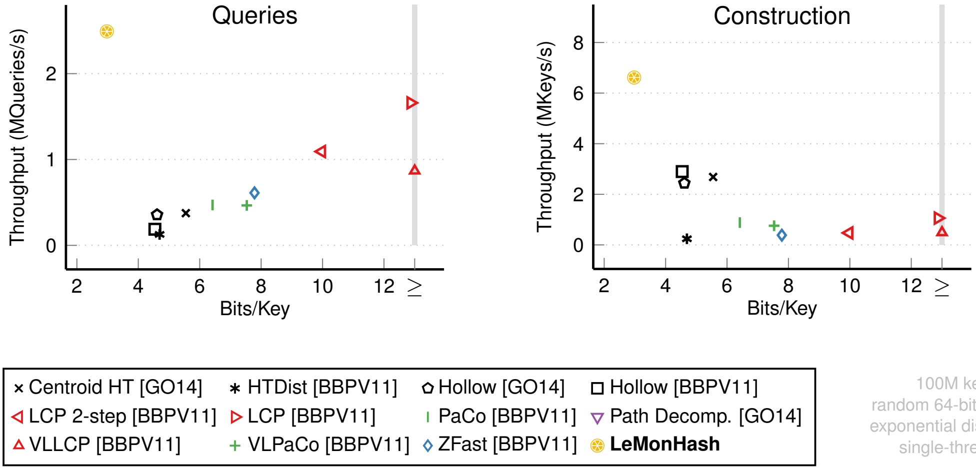


Monotone MPHF: Experiments



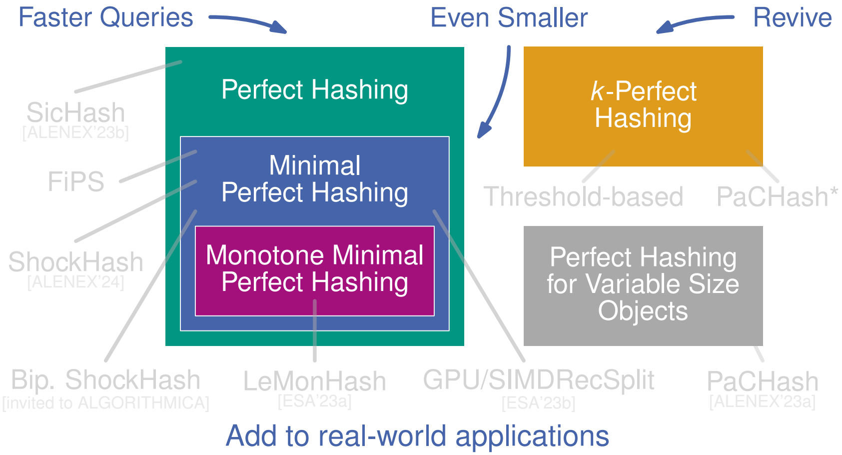
100M keys,
random 64-bit integers,
exponential distribution,
single-threaded

Monotone MPHF: Experiments



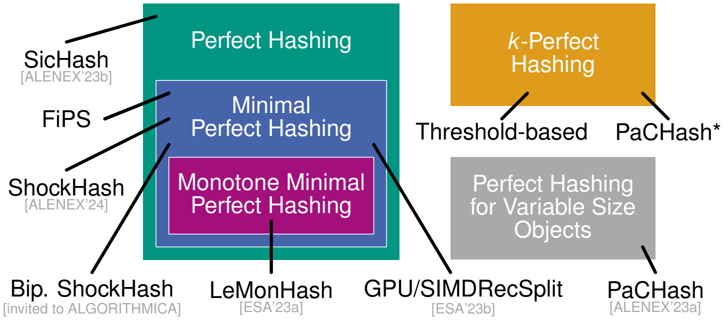
100M keys,
random 64-bit integers,
exponential distribution,
single-threaded

Future Work



Faster Queries
Even Smaller
Revive
Perfect Hashing
Minimal
Perfect Hashing
Monotone Minimal
Perfect Hashing
k-Perfect
Hashing
Perfect Hashing
for Variable Size
Objects
SicHash
[ALENEX'23b]
FiPS
ShockHash
[ALENEX'24]
Threshold-based
PaCHash*
Bip. ShockHash
[invited to ALGORITHMICA]
LeMonHash
[ESA'23a]
GPU/SIMDRecSplit
[ESA'23b]
PaCHash
[ALENEX'23a]
Add to real-world applications

Summary



- Fast and space-efficient perfect hashing
- More than 2^n asymptotic speedup
- Implementation up to 76 000 times faster than previous state of the art

References I

[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.

[BBPV11] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna.
Theory and practice of monotone minimal perfect hashing.
ACM J. Exp. Algorithmics, 16, 2011.

[Bel23] Piotr Beling.
Fingerprinting-based minimal perfect hashing revisited.
ACM J. Exp. Algorithmics, 28:1.4:1–1.4:16, 2023.

[BPZ07] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani.
Simple and space-efficient minimal perfect hash functions.
In *WADS*, volume 4619 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2007.

References II

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

[DHSW22] 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.

[GO14] Roberto Grossi and Giuseppe Ottaviano.
Fast compressed tries through path decompositions.
ACM J. Exp. Algorithmics, 19(1), 2014.

References III

[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.
In *ESA*, volume 308 of *LIPIcs*, pages 69:1–69:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 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.

[Meh84] Kurt Mehlhorn.
Data structures and algorithms, vol. 1: Sorting and searching.
EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1984.

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