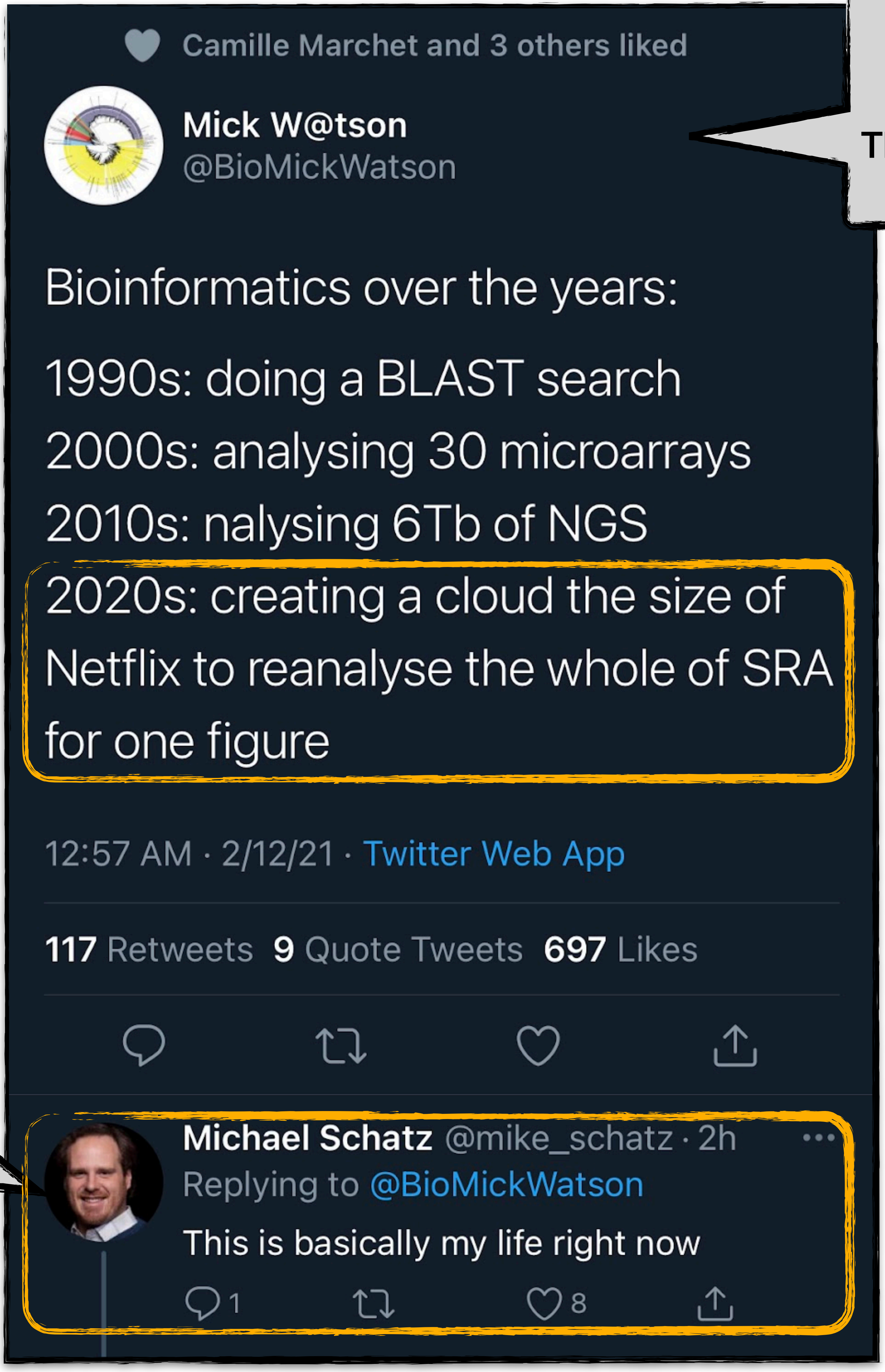
From Filters to Hash Tables

Rethinking Core Data Structures for Scalable Performance

Prashant Pandey, Northeastern University, Boston USA
https://prashantpandey.github.io/

Scalability challenges ft. Twitter

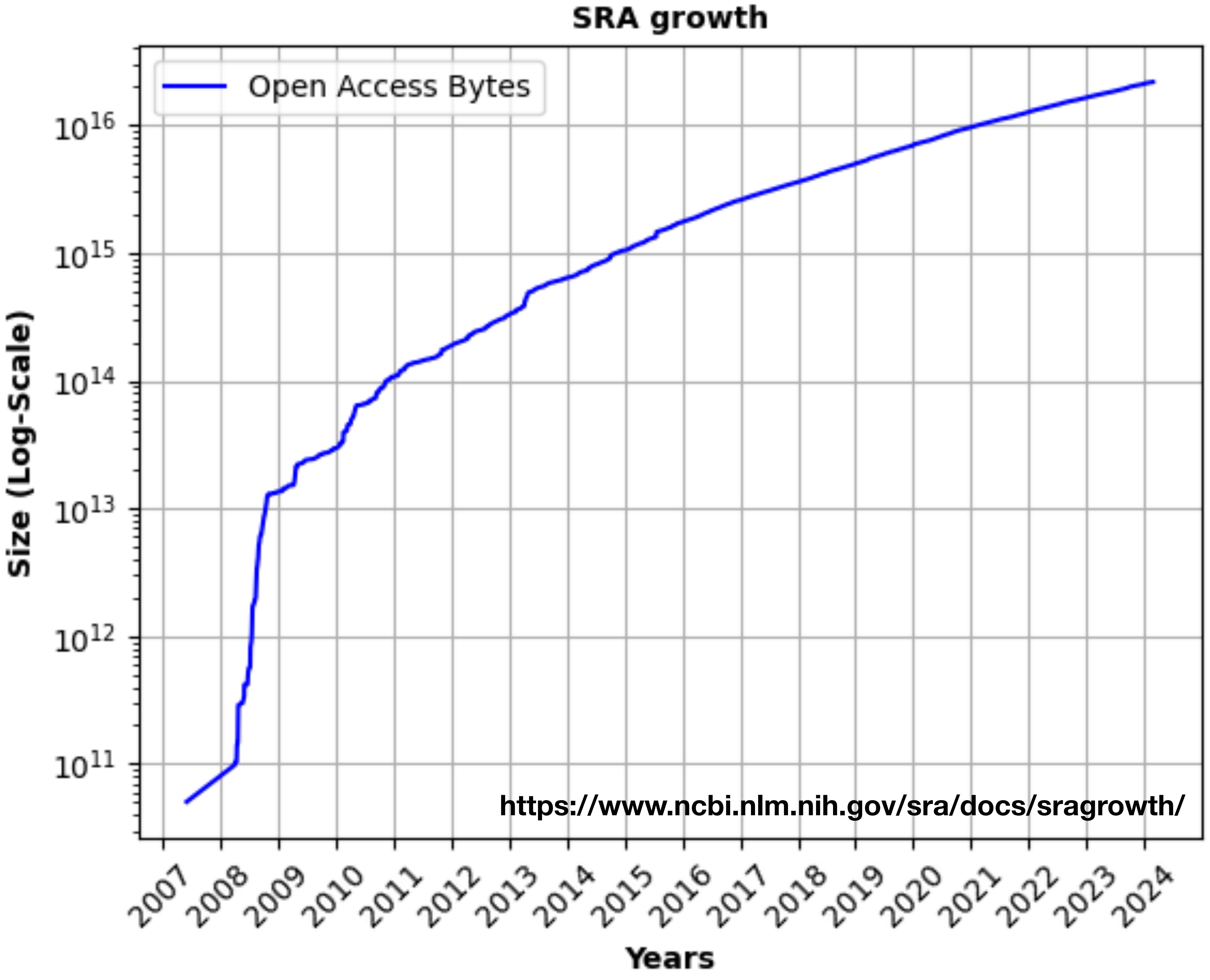


Professor Bioinformatics and Comp Bio.
The University of Edinburgh

Professor Computer Science and Comp Bio.
Johns Hopkins University

Sequence Read Archive (SRA) is growing rapidly

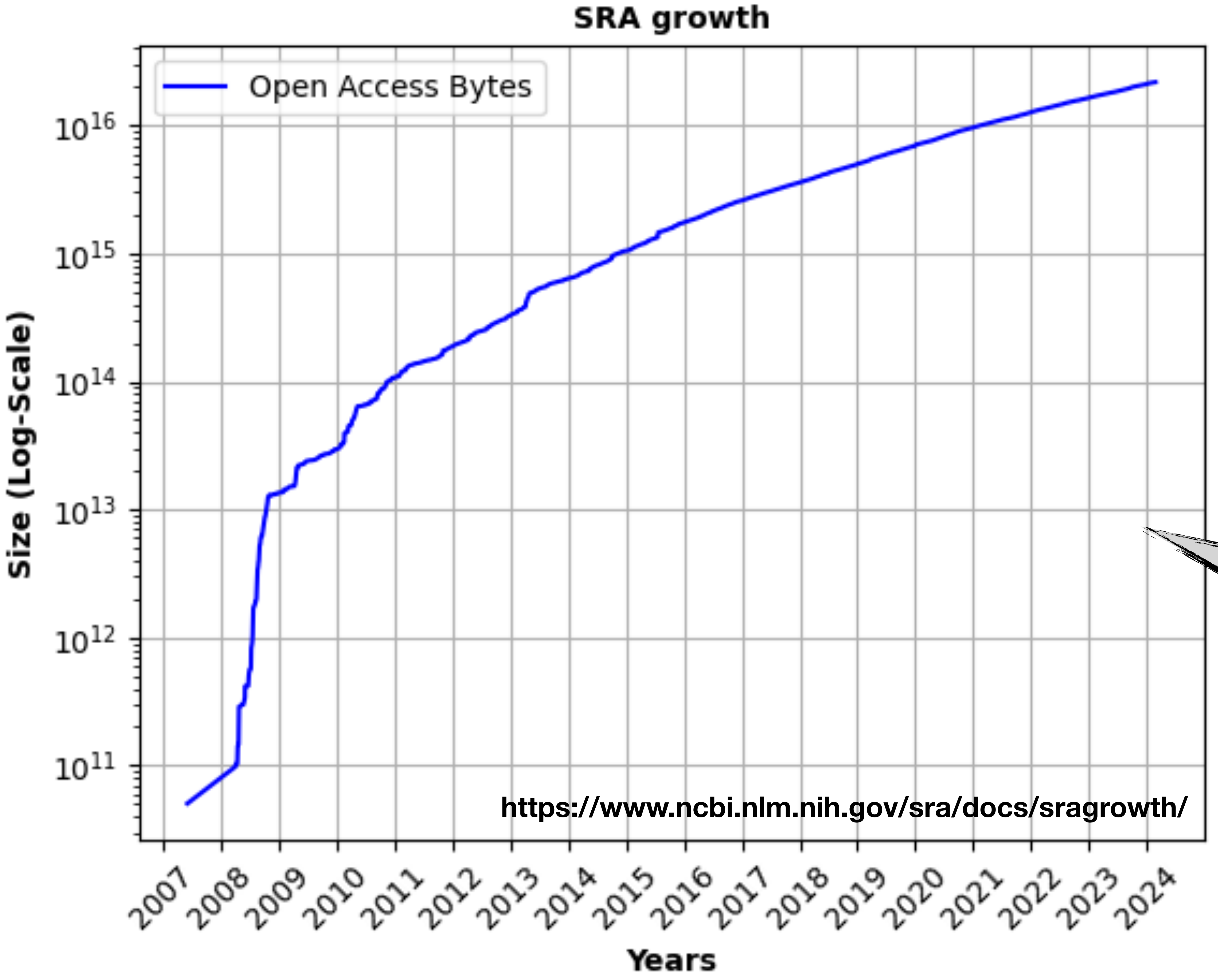
SRA contains a lot of biological diversity information



Q: What if I find, e.g., a new disease-related gene, and want to see if it appeared in other experiments?

Scalability is a critical bottleneck in data science

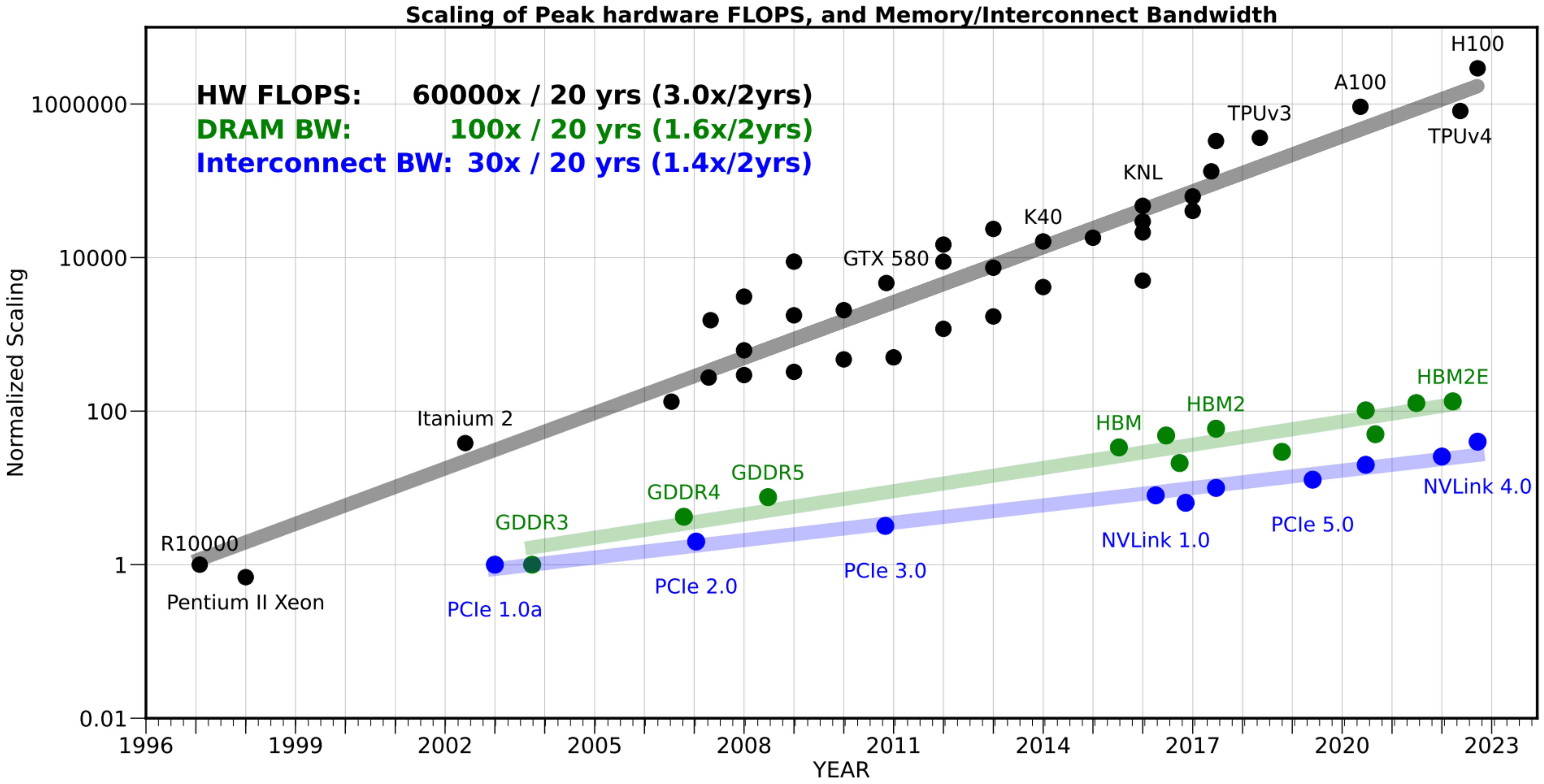
SRA contains a lot of biological diversity information



Current search indexes
~100 TB

This renders what is otherwise an immensely valuable public resource ***largely inert***!

Efficient scaling needs efficient data movement

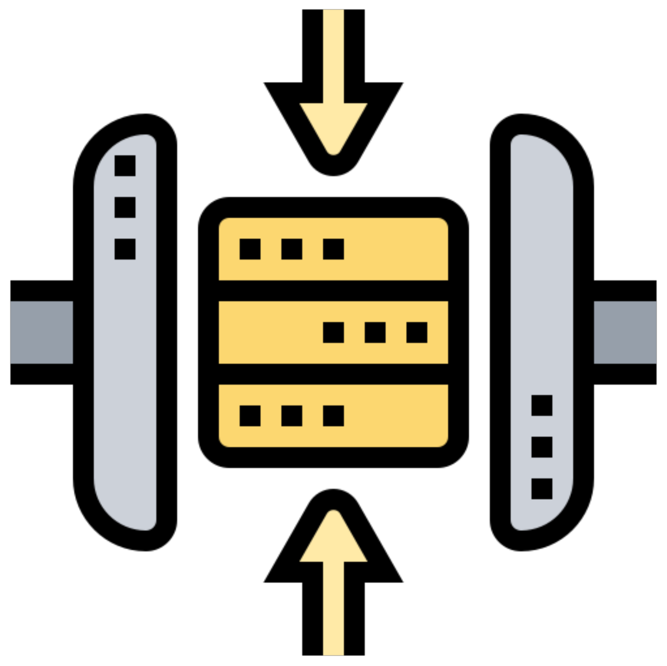


Memory wall

Gholami, Yao, Kim, Hooper, Mahoney, Keutzer IEEE Micro 2024

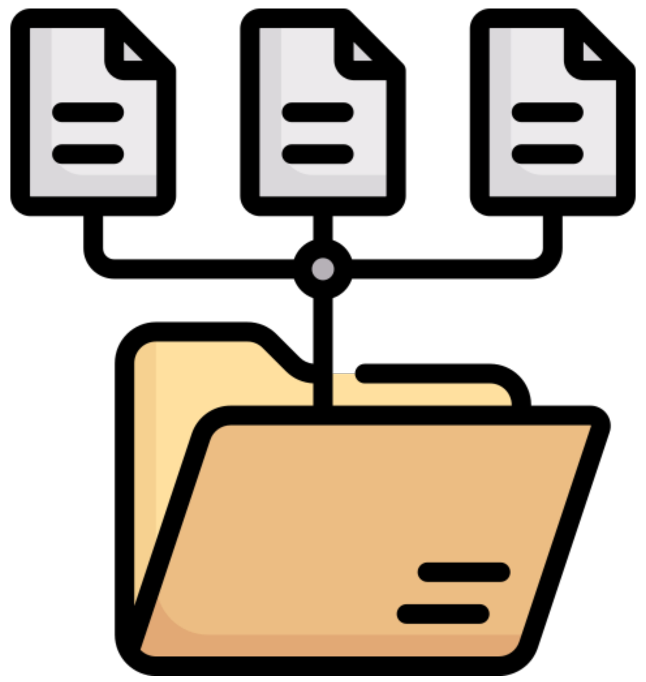
My goal as a researcher is to build scalable data systems with strong theoretical guarantees

Three approaches to build scalable data systems



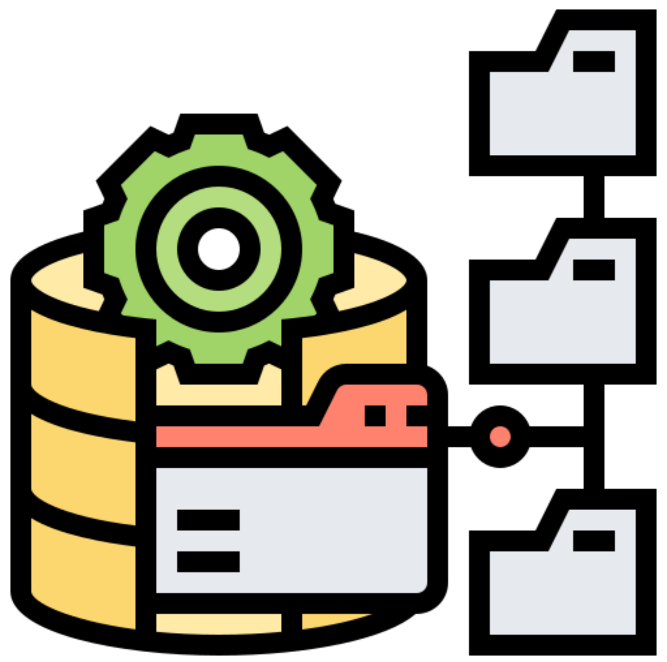
Compress it

Goal: make data smaller to fit inside fast memory



Organize it

Goal: organize data in a I/O friendly way



Distribute it

Goal: distribute data & reduce inter-node communication

Vertically integrated research

COMPRESS ORGANIZE DISTRIBUTE

Applications

- **Genomic Data Processing Toolchain (Squeakr, deBGR, Mantis, Rainbowfish)**
  BIOINFORMATICS 17,18
  ISMB 17, RECOMB 18, 19
  Cell Systems 18
- **LSM-Mantis**
  BIOINFORMATICS 22
- **Variation Graph VariantStore**
  Genome Biology 21
- **Metagenomic Assembler (MetaHipMer*)**
  IPDPS 21, ACDA 23
- **Distributed Graph Learning (RDM)**
  IPDPS 23

Systems

- **File System (B^{e}trFS)**
  FAST 15, 16,
  TOS 16, 17
- **Anomaly detection (LERT)**
  SIGMOD 20, TODS 21
- **Graph system (Terrace, BYO)**
  SIGMOD 21, VLDB 24
- **Distributed KV (IONIA)**
  FAST 24

Data structures

- **Filters**
  SIGMOD 17, 21, 24
- **Sketches**
  ESA 18, APOCS 23, ISMB 19
- **Hash tables**
  SIGMOD 23, 25
- **B-trees**
  VLDB 23, SPAA 19, TOPC 21
- **GPU DS**
  PPoPP 23, 24
- **Learned Indexes**
  ACDA 25
- **Adaptive Filters**
  SIGMOD 25

Dictionary data structures

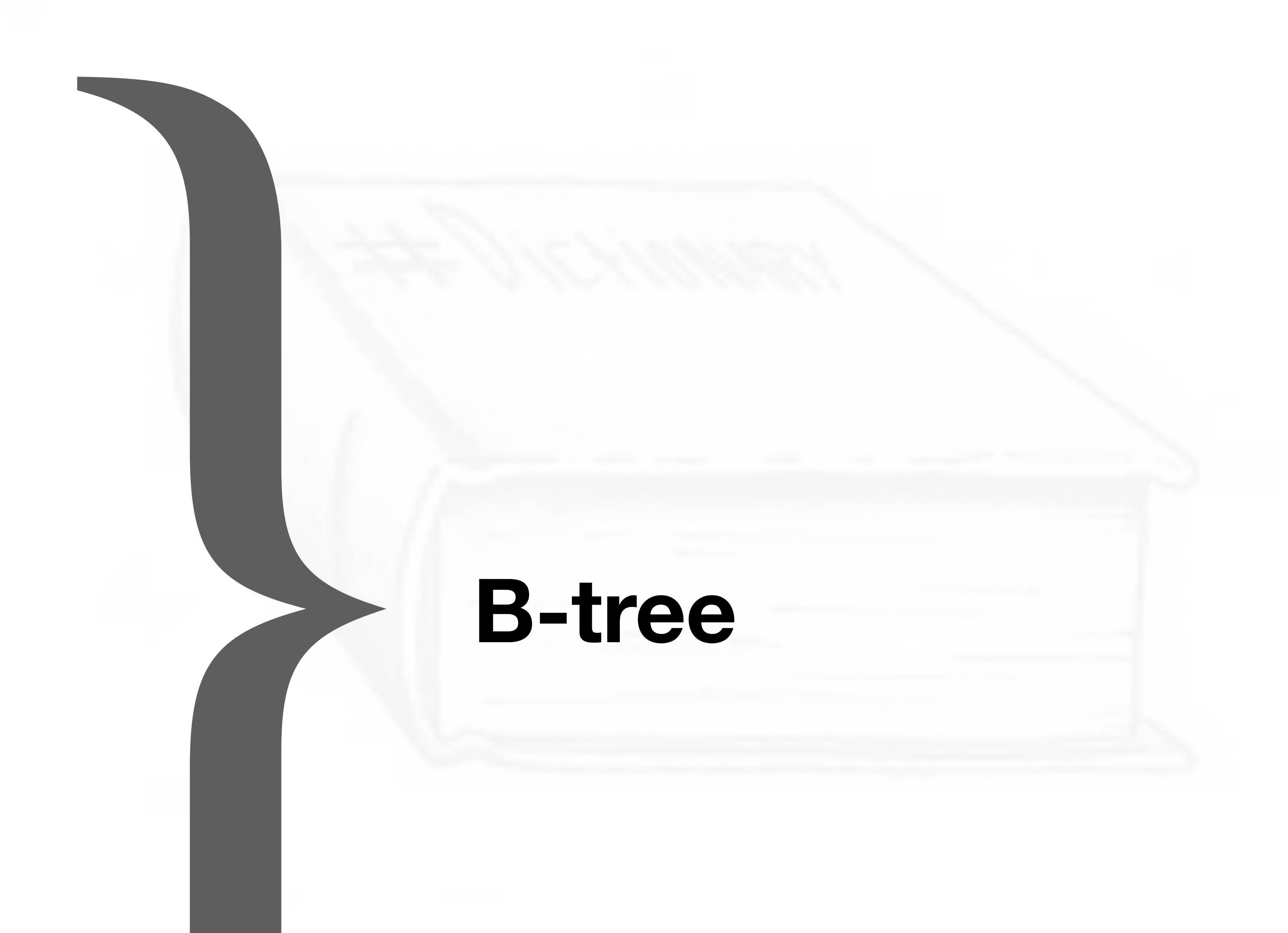
- Queries
  - Predecessor/Successor
  - Range queries
  - Membership
- Updates
  - Insertions
  - Deletions

Dictionary data structures

- Queries
  - Predecessor/Successor
  - Range queries
  - Membership
- Updates
  - Insertions
  - Deletions

Hash table

B-tree



IcebergHT [SIGMOD 2023]

Pandey, Bender, Conway, Farch-Colton, Kuszmaul, Tagliavini, Johnson

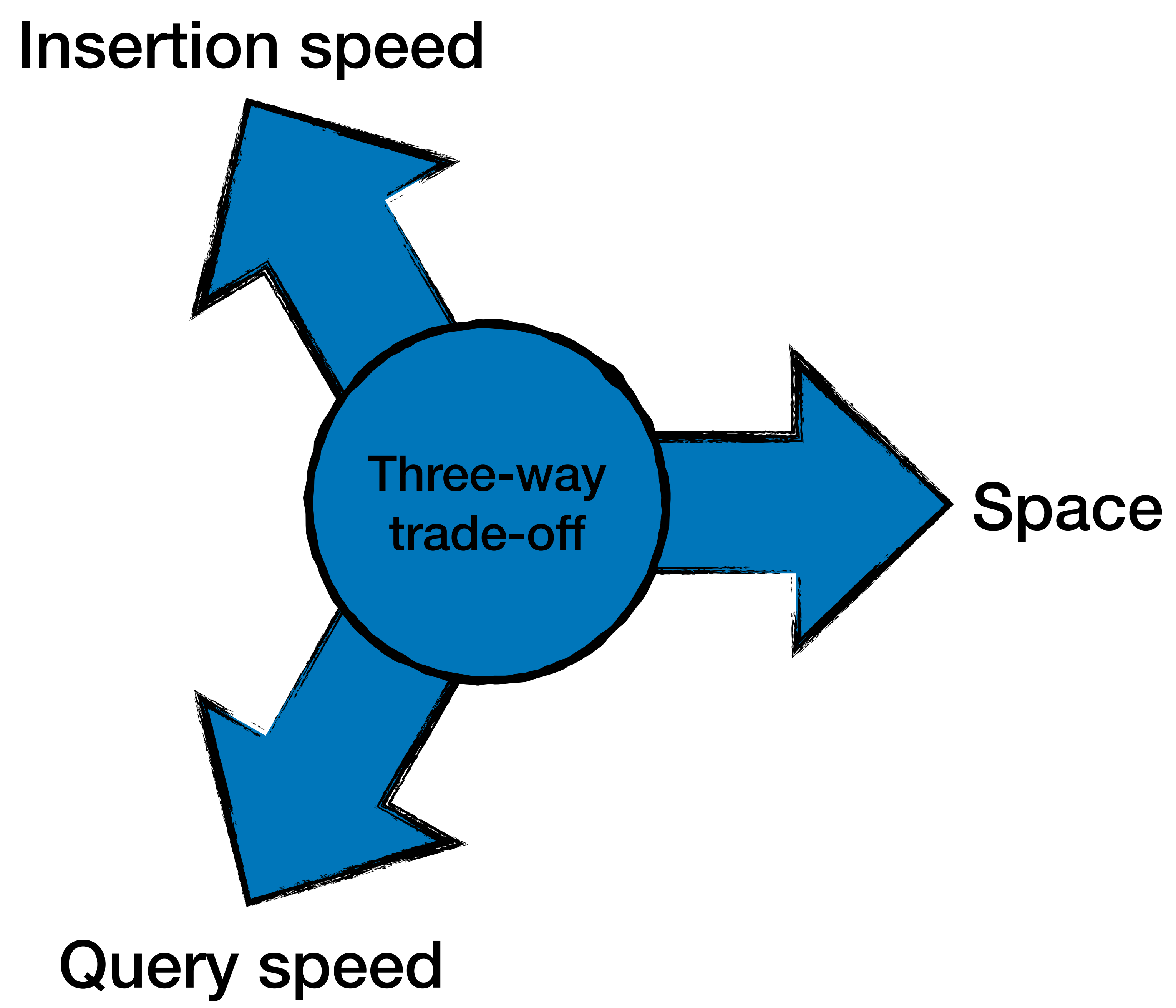
Hash tables are everywhere!

Built into many languages…

Built into many software packages…

And performance is critical to many applications

Hash table performance criteria



Hash table performance has a three-way trade off between insertion speed, query speed, and space

Hash table design mechanism

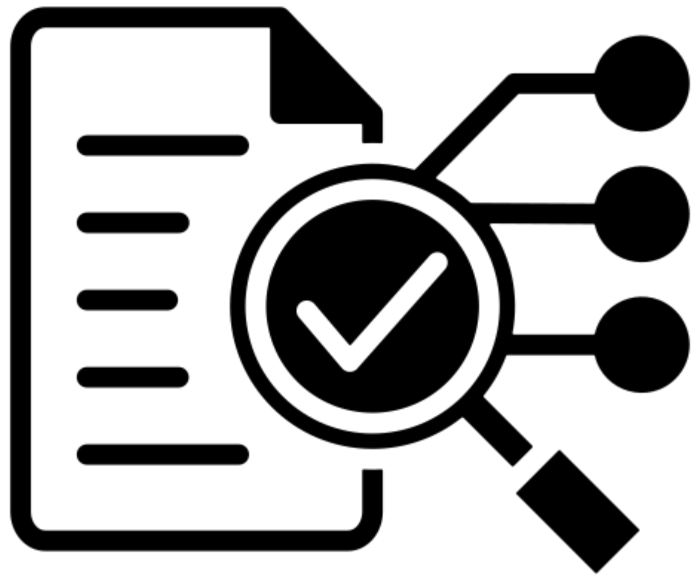
Stability

Items don’t move after insertion

Fast insertion

Low associativity

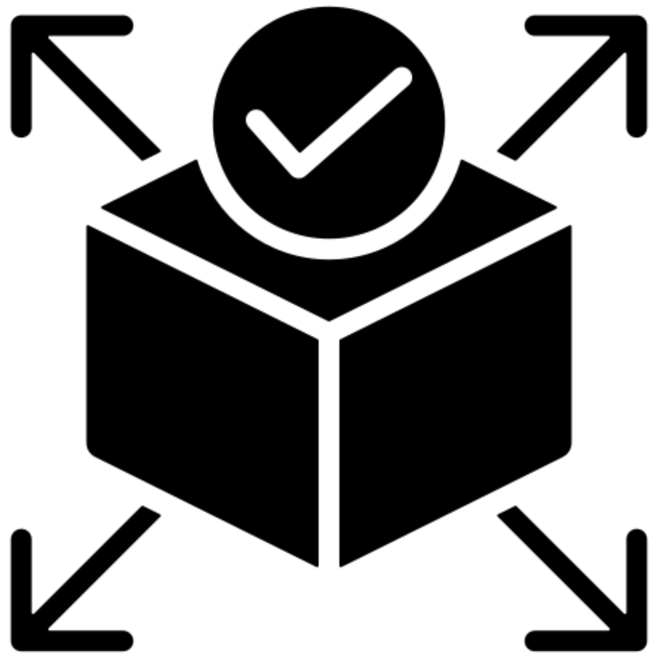
Map each item to a small number of locations



Fast queries

Space efficiency

Minimum overhead from pointers or over provisioning

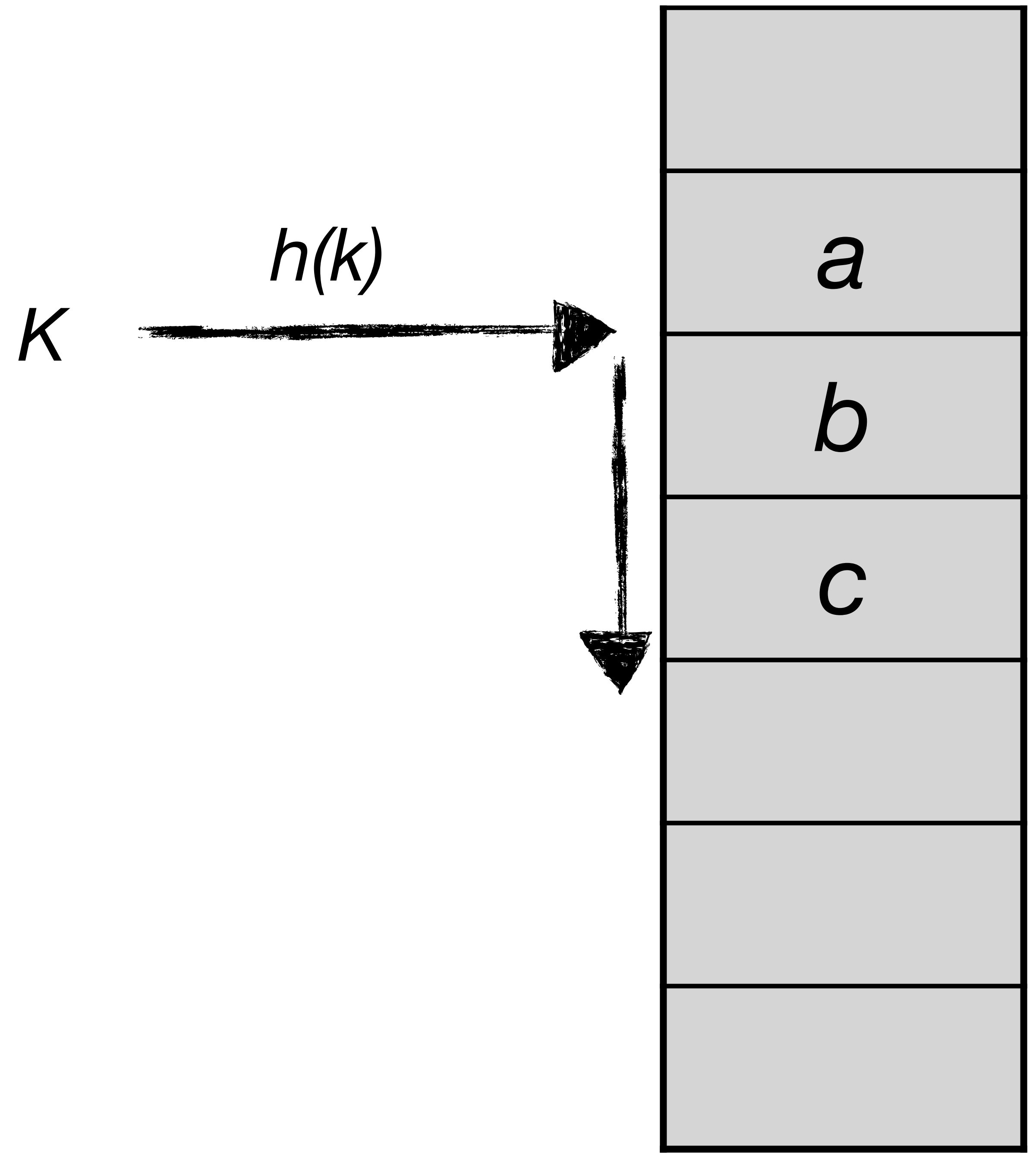


Low space

Achieving all three is a long-standing open problem in hash table design

For example: linear probing

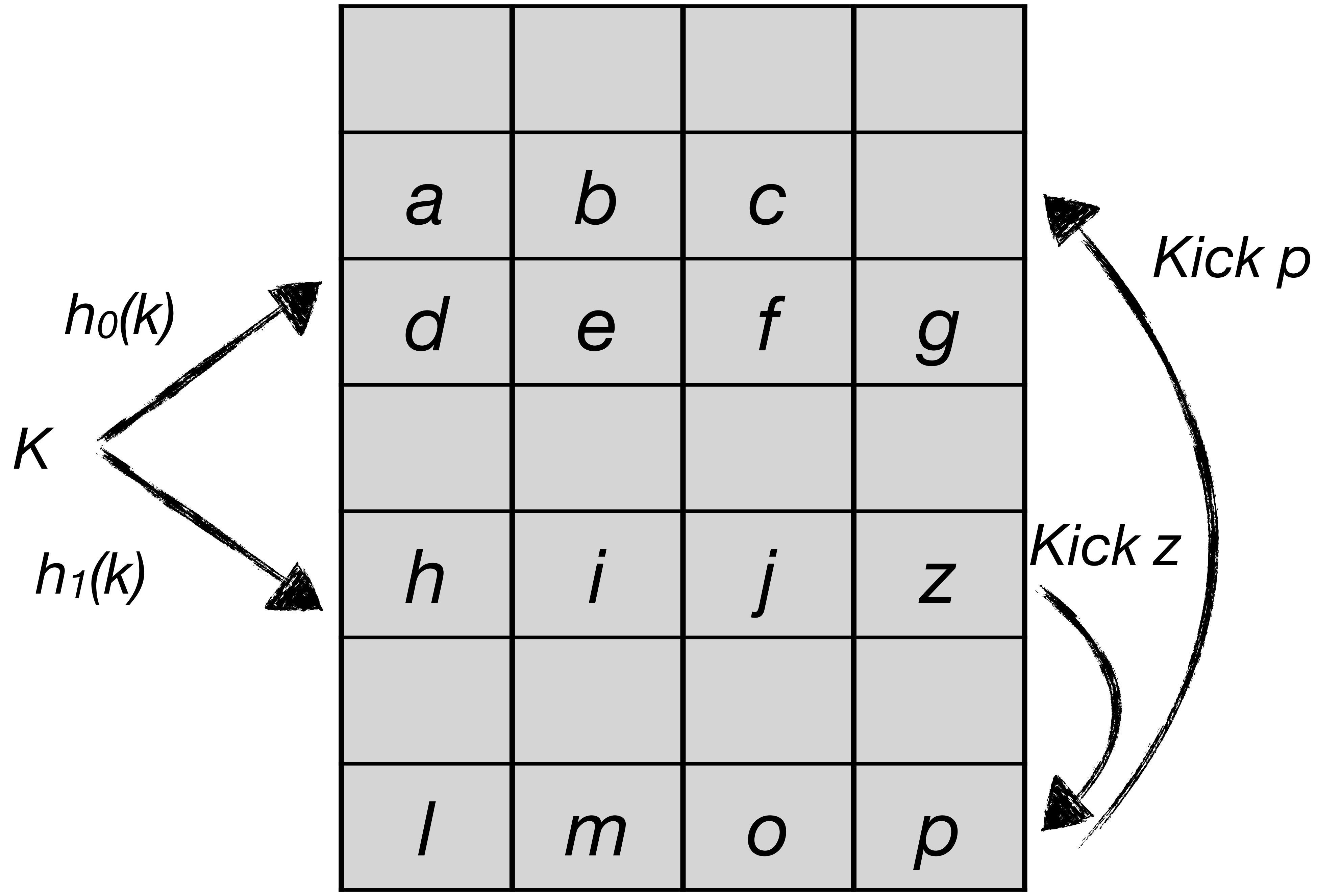
- Stable
- Associativity $\approx \dfrac{\log N}{(1-\alpha)^2}$ (α = load factor)
- E.g., N = 1Billion, α = 95%, associativity = 12000



Must choose between low associativity and space efficiency

For example: cuckoo hashing

- Low associativity: queries must check only 2 cache lines
- Space efficient, load factor > 95%
- But not stable

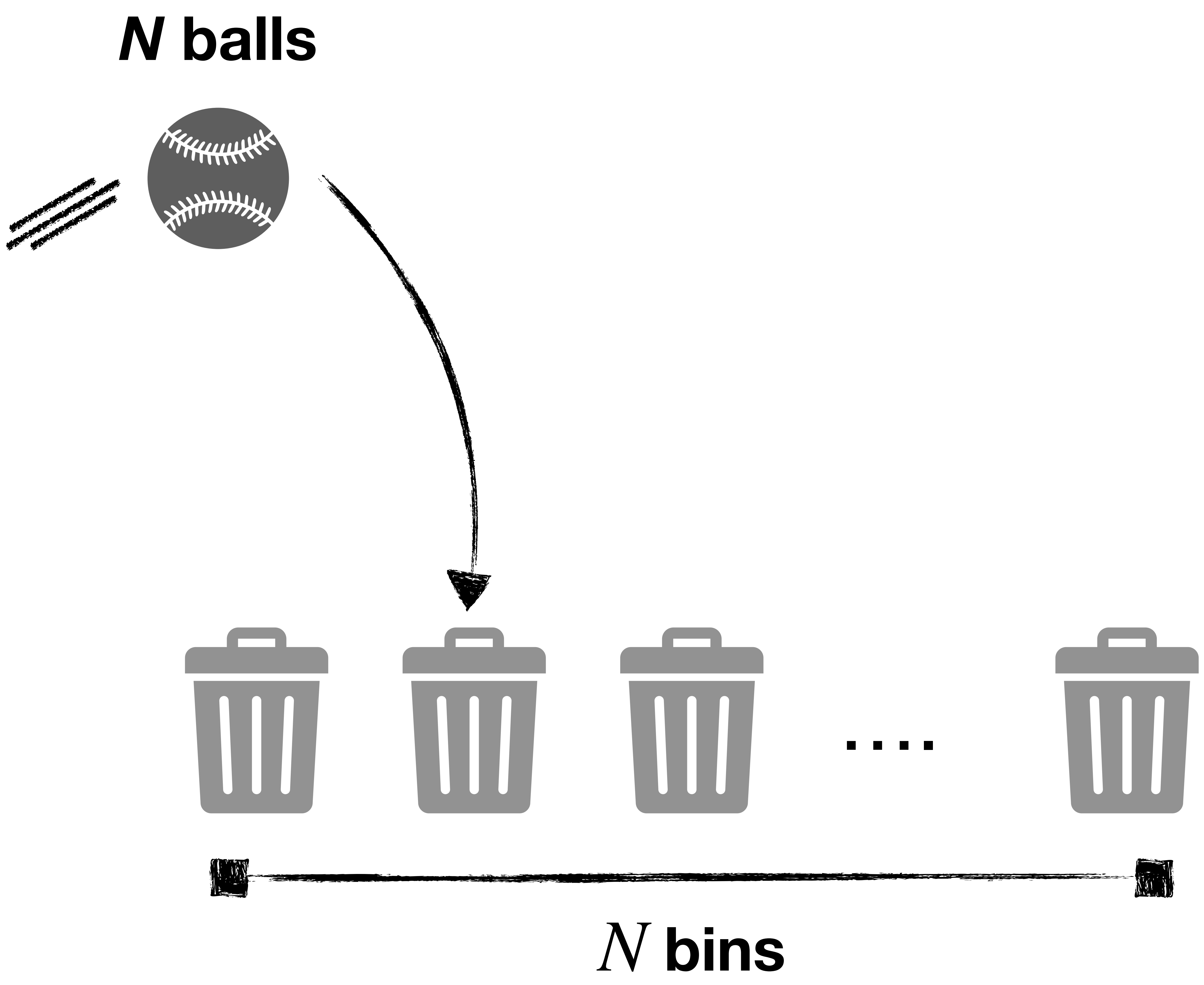


Insertion performance drops significantly due to excessive kicking at high load factors

Other hashing schemes:

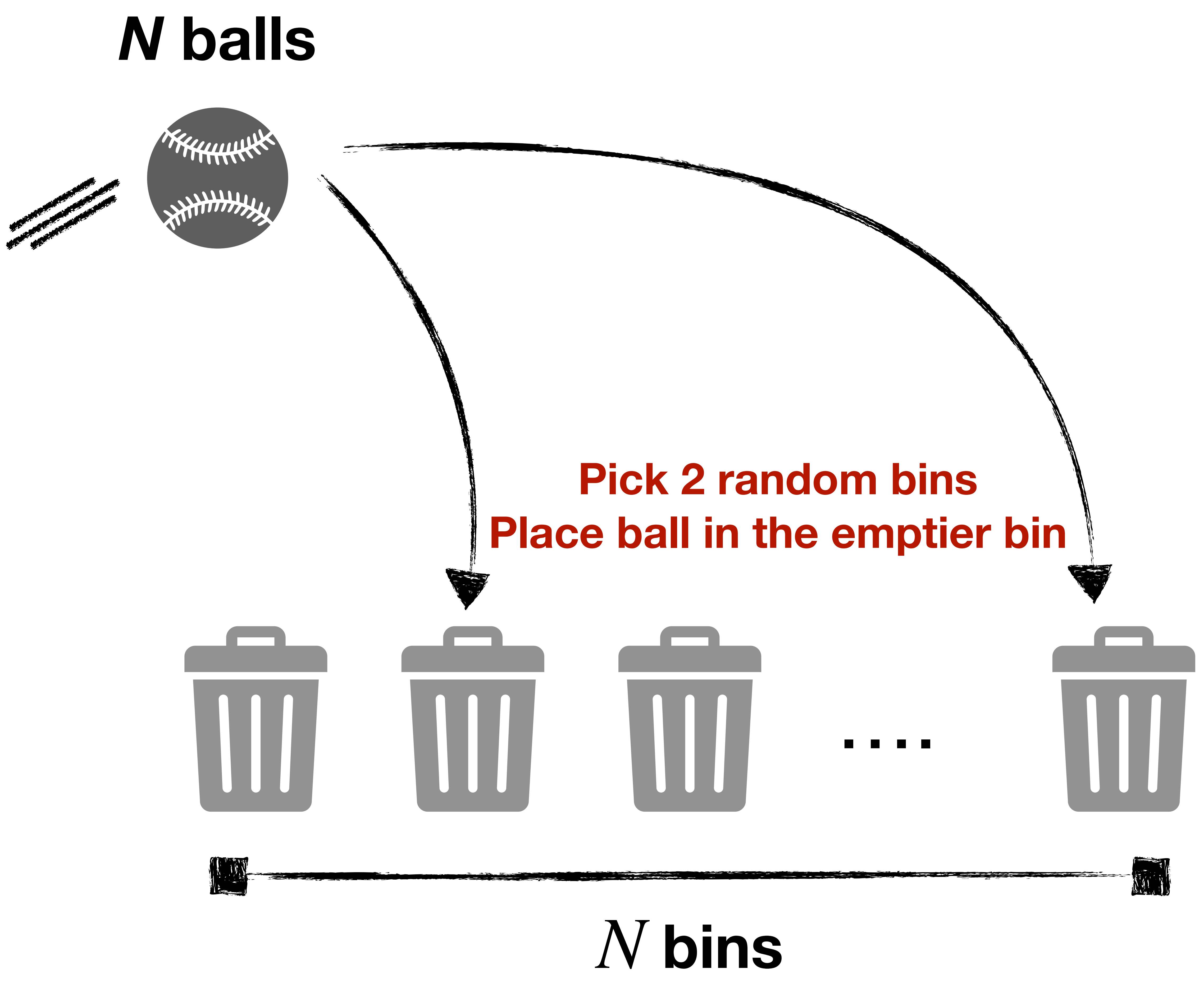
- Other hashing schemes also lack one or more of these properties
- **Chaining**: not low associativity
- **Robin hood**: not stable and not low associativity at high load factors
- **Hopscotch**: not stable
- **Quadratic probing**: not stable and not low associativity at high load factors

Single choice hashing (Balls n Bins)



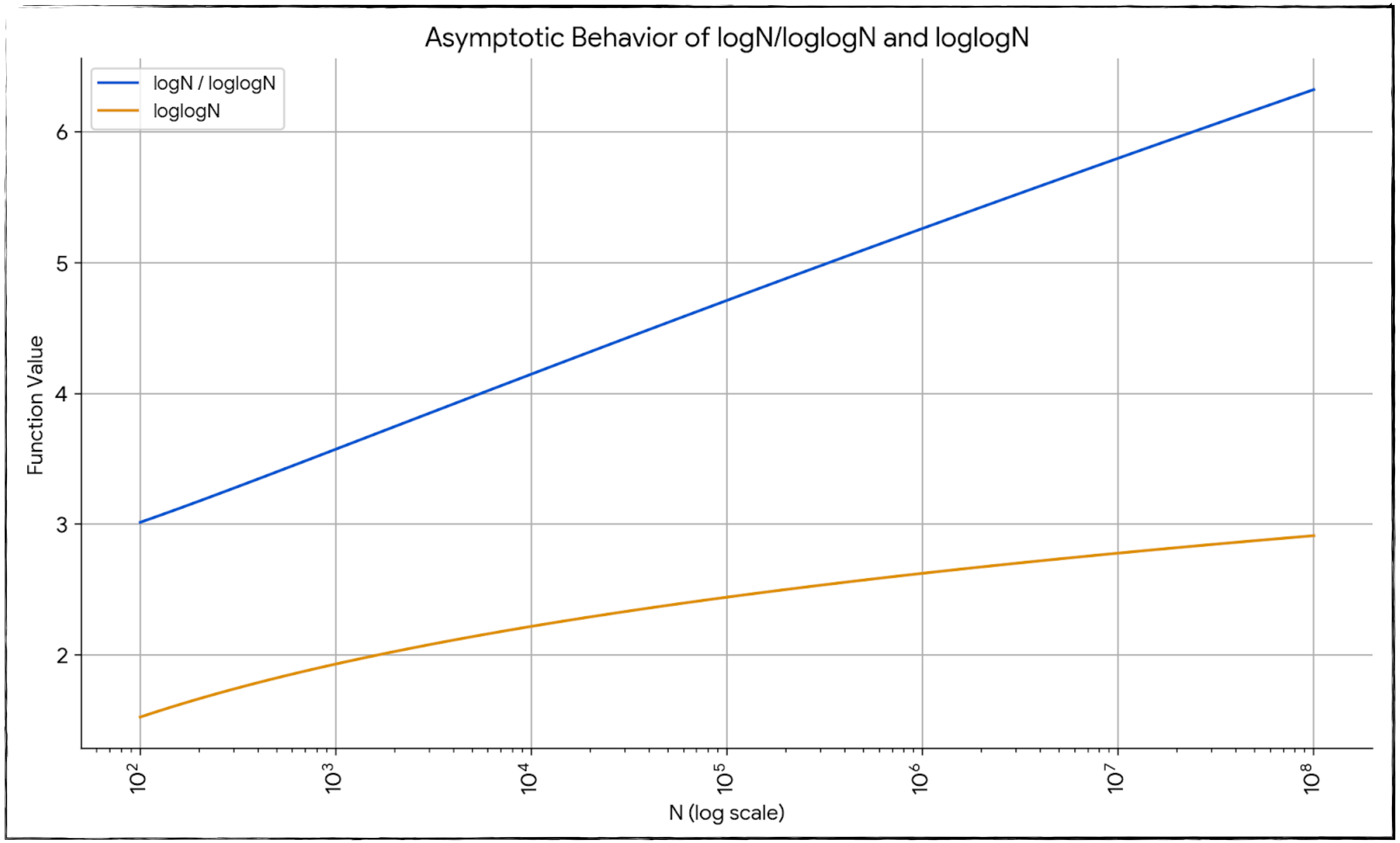
Theorem: if you throw N balls into N bins, the fullest bin will have $\Theta(\log N / \log \log N)$ balls W.H.P.

Two choice hashing (Balls n Bins)

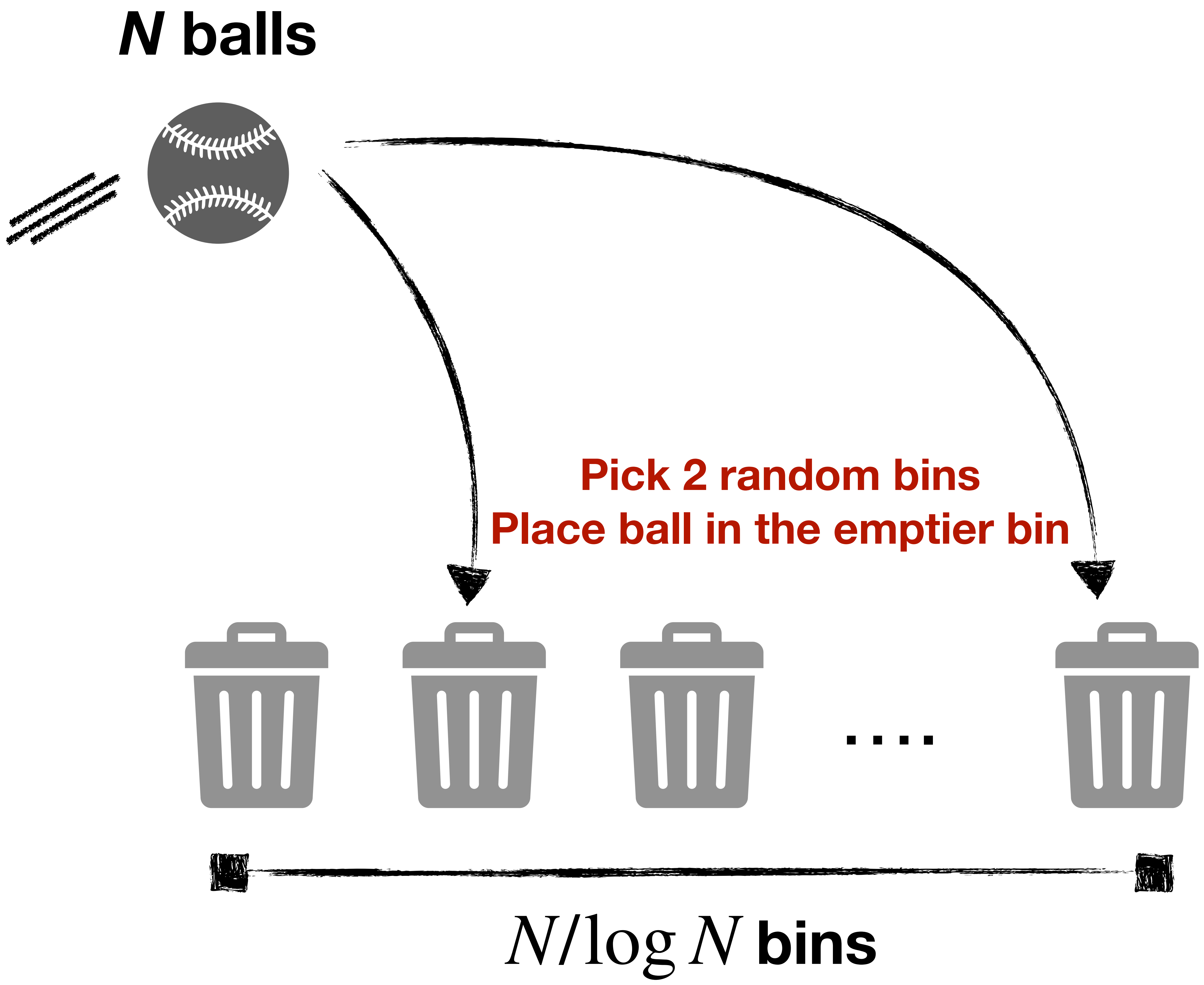


Theorem: if you throw *N* balls into N bins using minimum of two choices, the fullest bin will have $\Theta(\log \log N)$ balls W.H.P.

Two choice hashing provides asymptomatic improvement



Two choice hashing for hash tables

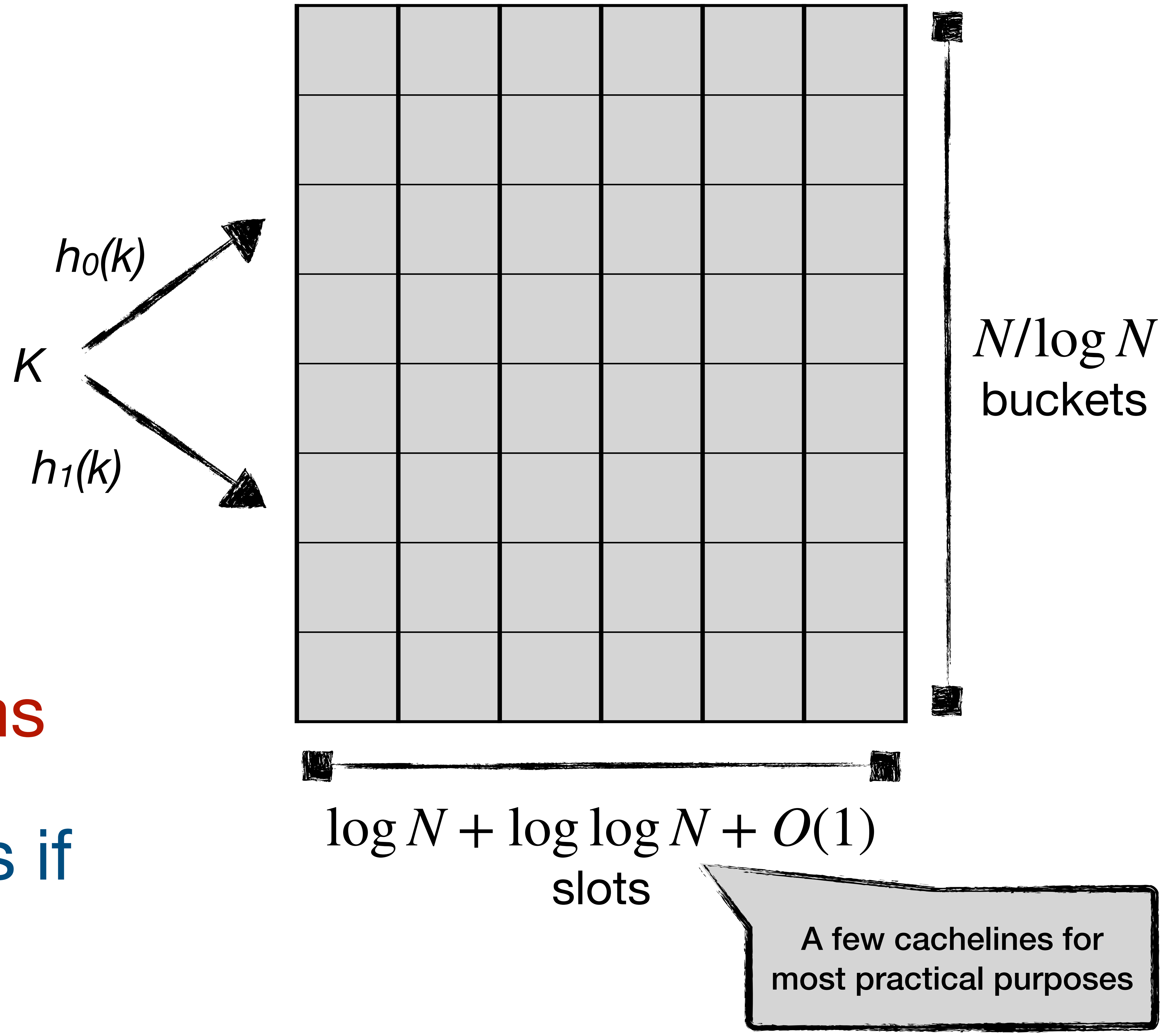


Theorem: if you throw *N* balls into $N/\log N$ bins using minimum of two choices, the fullest bin will have $\log N + \log\log N + O(1)$ balls W.H.P.

- By Berenbrink, Czumaj, Steger, Vöcking 2000

An almost solution: two choice hashing

- **2-choice hashing**: hash to two buckets and put item in emptier bucket
- Stable: no kicking
- Low associativity: $O(\log N)$
- Space efficient: load factor $1 - o(1)$



Problem: it does not hold when we delete items

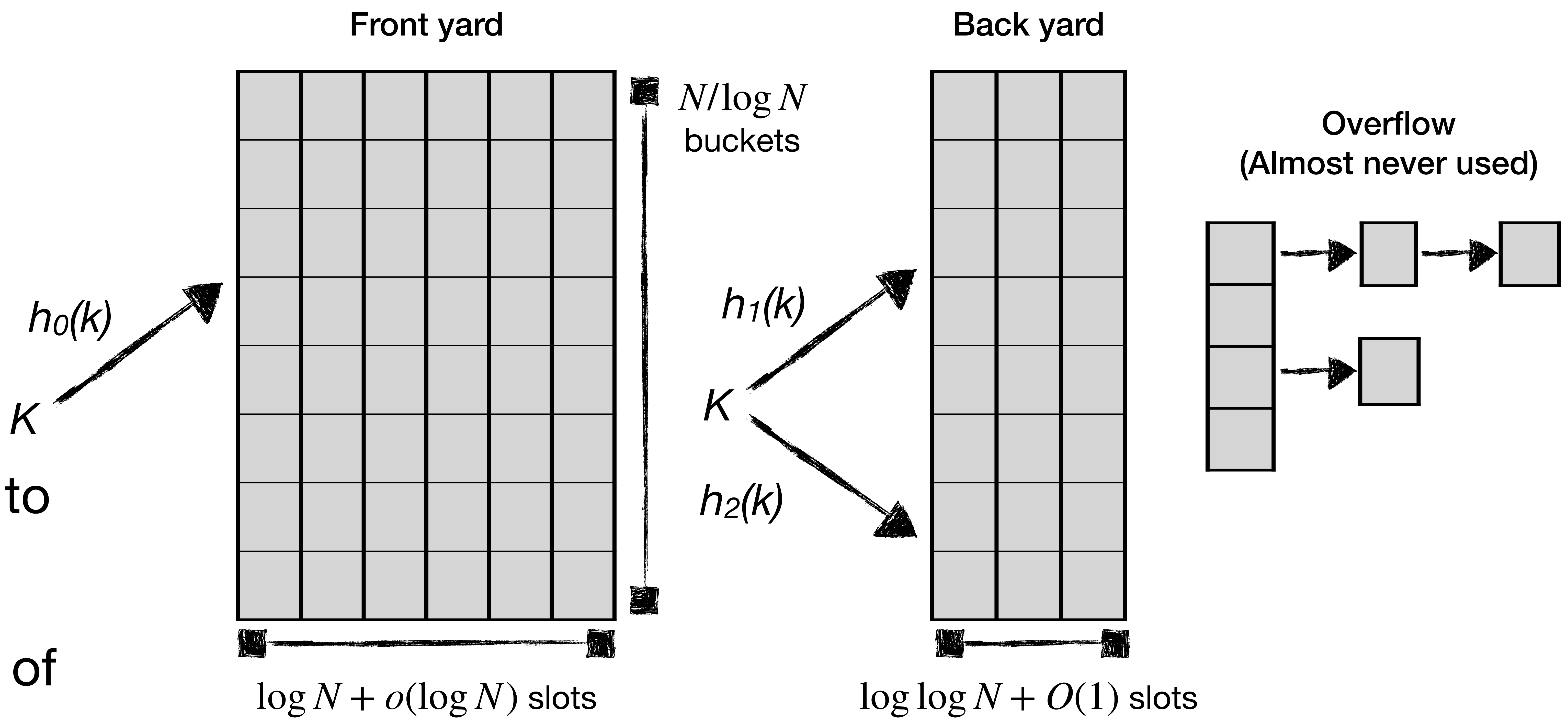
Opportunity: theorem does hold with deletions if average bucket occupancy is $O(1)$

Iceberg hashing (Single + Two choice hashing)

- **Iceberg theorem**: if you throw N balls into $N/\log N$ bins of size $\log N + o(\log N)$, the number of overflow balls will be $O(N/\log N)$

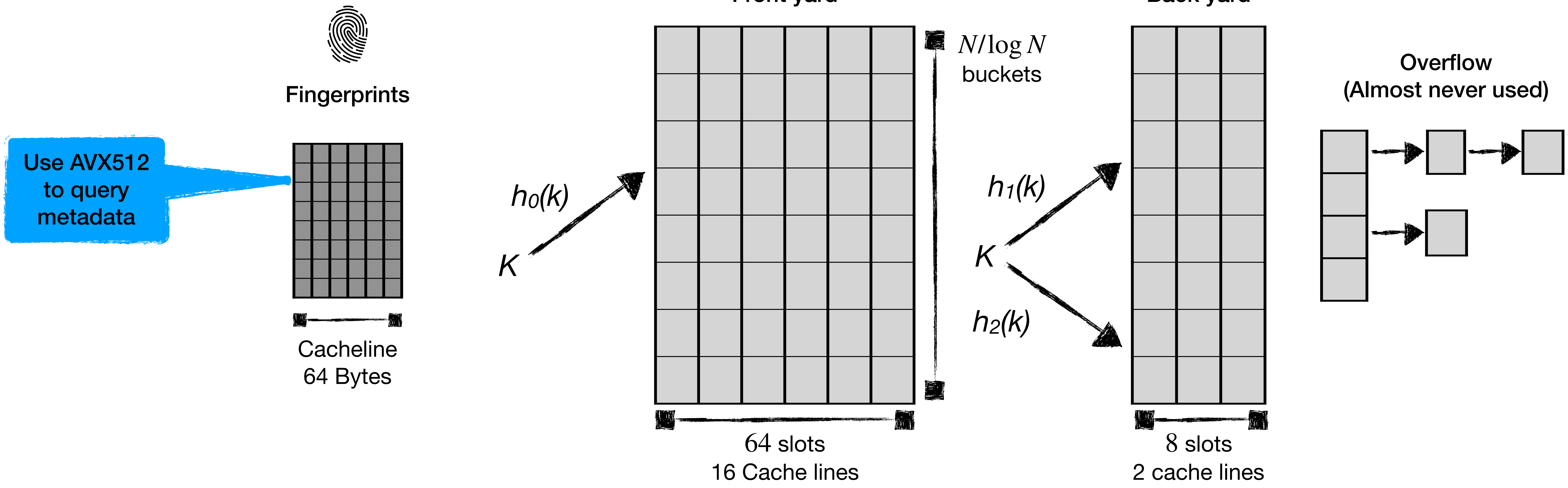
- **Idea**: use single-choice front yard to absorb most items

- Backyard has average occupancy of $O(1)$



Problem: buckets in the front yard span many cache lines, so queries must load many cache lines.

Iceberg hashing: metadata to reduce associativity



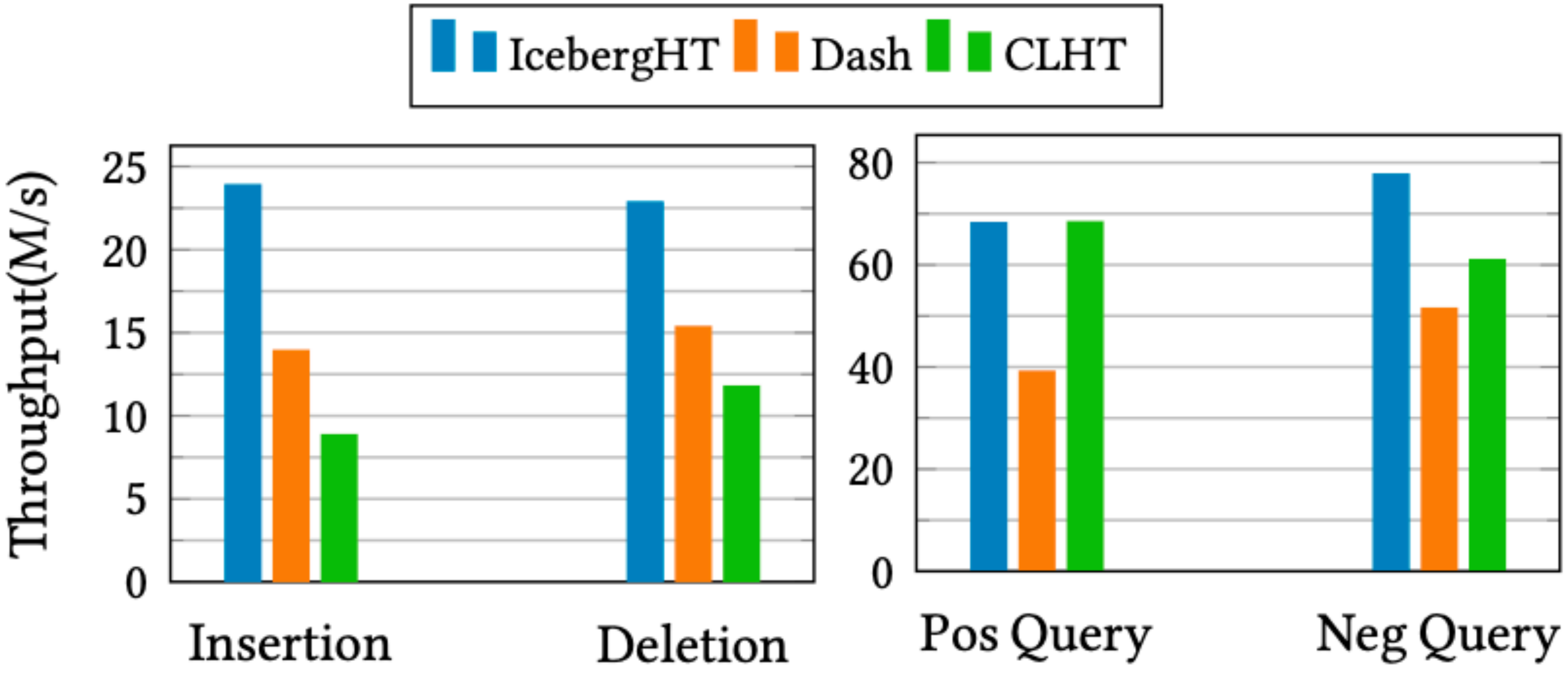
Problem: buckets in the front yard span many cache lines, so queries must load many cache lines.

Solution: store a fingerprint table.

IcebergHT implementation

- Highly concurrent operations
- IcebergHT supports in-place resizing; reduces peak memory usage
    - Multi-threaded resizes are implemented using distributed reader-writer locks
- Crash safety is trivial
    - Using CLWB; no need for a fence between key & value writes
    - Metadata stays in DRAM and is reconstructed during recovery

PMEM performance: operation throughput



Performance using 16 threads for PMEM hash tables.

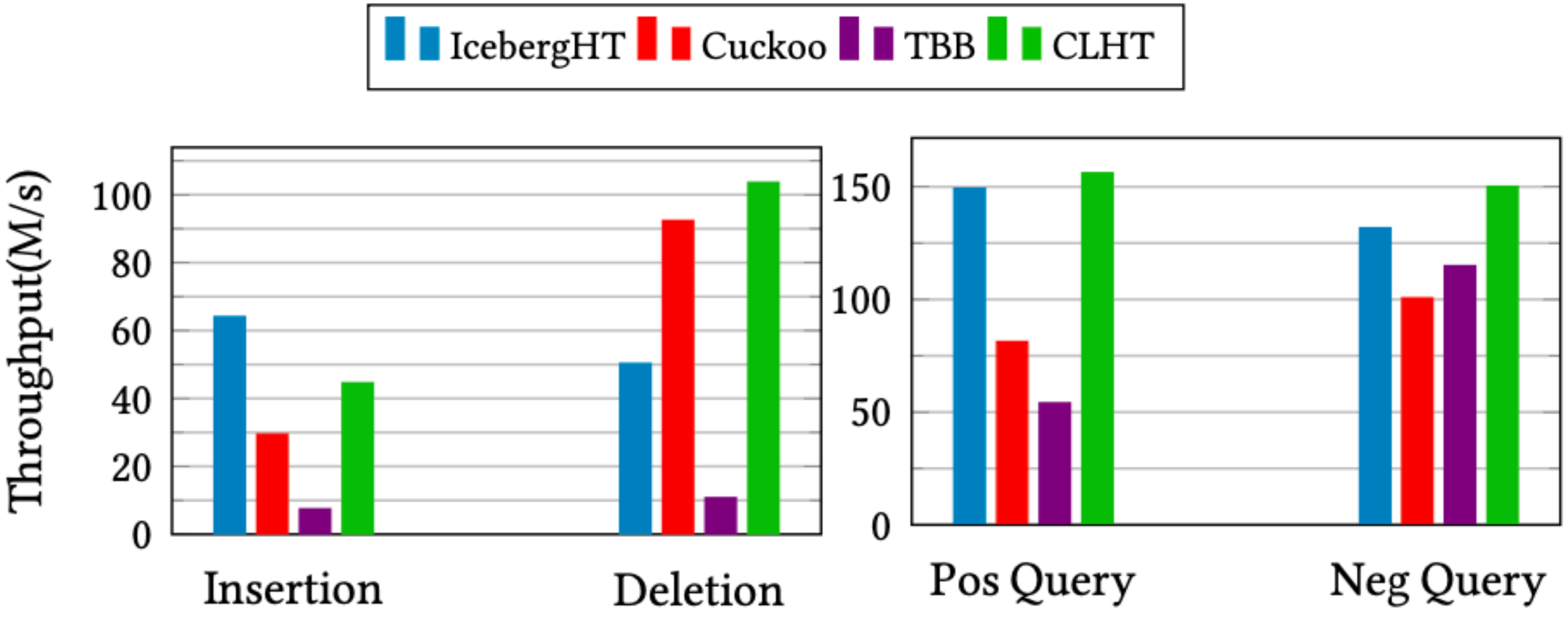
Iceberg outperforms state-of-the-art hash tables across all operations.

PMEM performance: space efficiency

Hash tables	Space efficiency
IcebergHT	85%
Dash	69%
CLHT	33%

IcebergHT offers higher space efficiency compared to Dash (extendible) and CLHT (chaining) hash tables.

DRAM performance: operation throughput

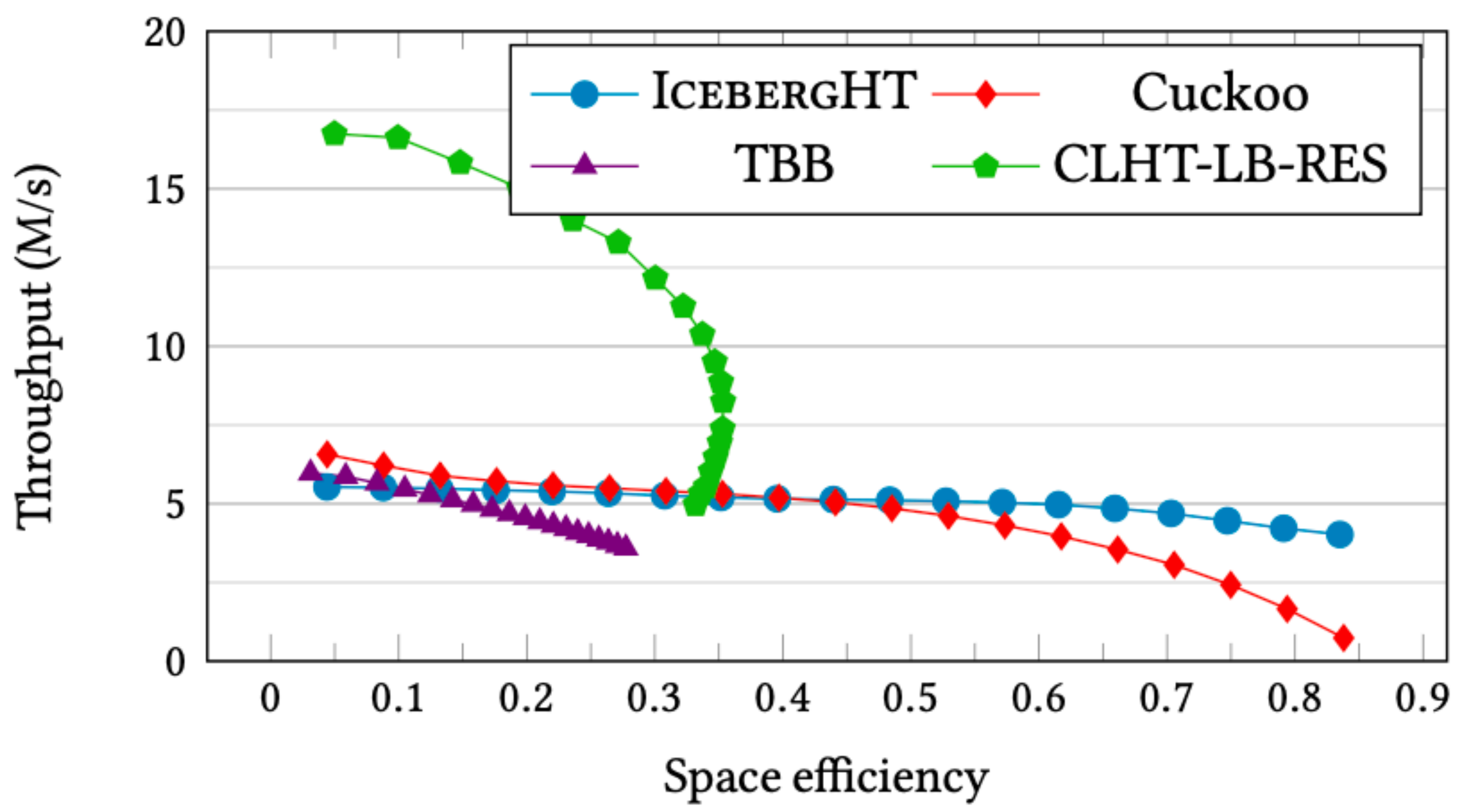


Performance using 16 threads for DRAM hash tables.

Iceberg outperforms state-of-the-art hash tables for insertions and offers similar performance to CLHT for queries.

IcebergHT deletes are slower.

DRAM performance: space efficiency



IcebergHT can achieve high space efficiency and maintain insertion throughput.

CLHT space efficiency drops quickly.

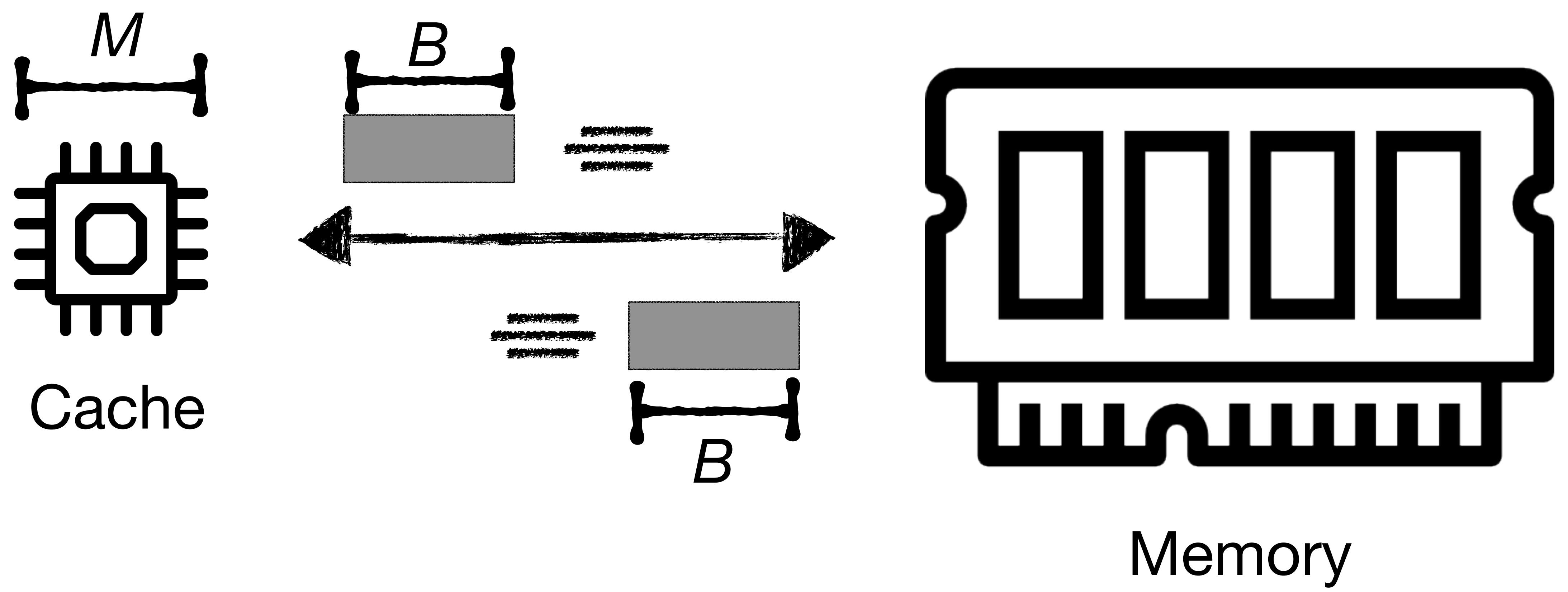
CuckooHT insertion throughput drops at high load factor.

BP-Tree [VLDB 2023]

Xu, Li, Wheatman, Marneni, Pandey

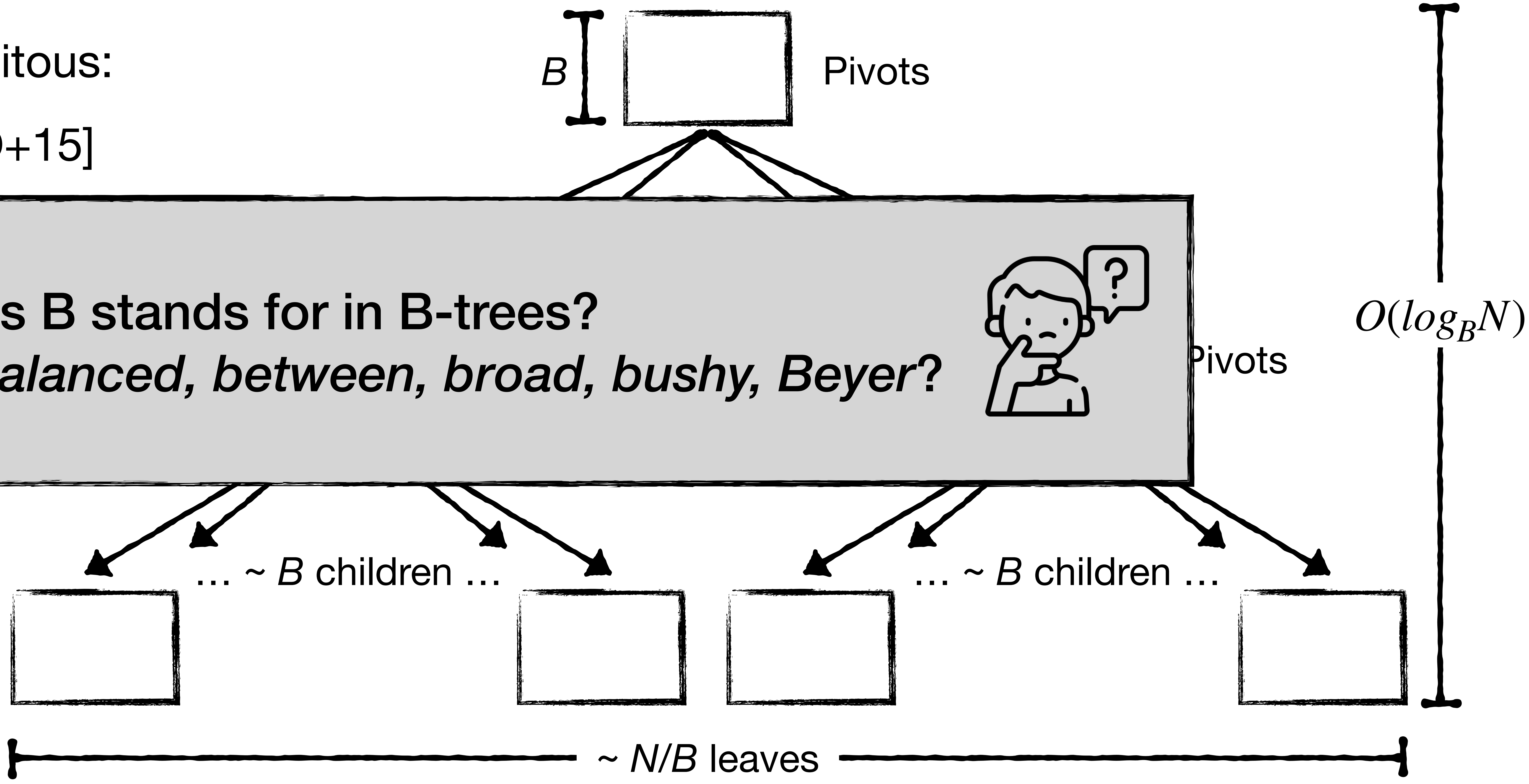
External memory model for dictionaries

- **How computations work [AV88]:**
  - Data is transferred in blocks between levels
  - The number of block transfers dominate the running time
- **Goal: minimize number of block transfers**
  - Performance bounds are parameterized by block size *B*, memory size *M*, and data size *N*



B-tree: a classic dictionary data structure

- B/B+-trees [BM72] are ubiquitous:
  - In memory indexing [ZCO+15]
  - Databases [K9
  - Filesystems [F



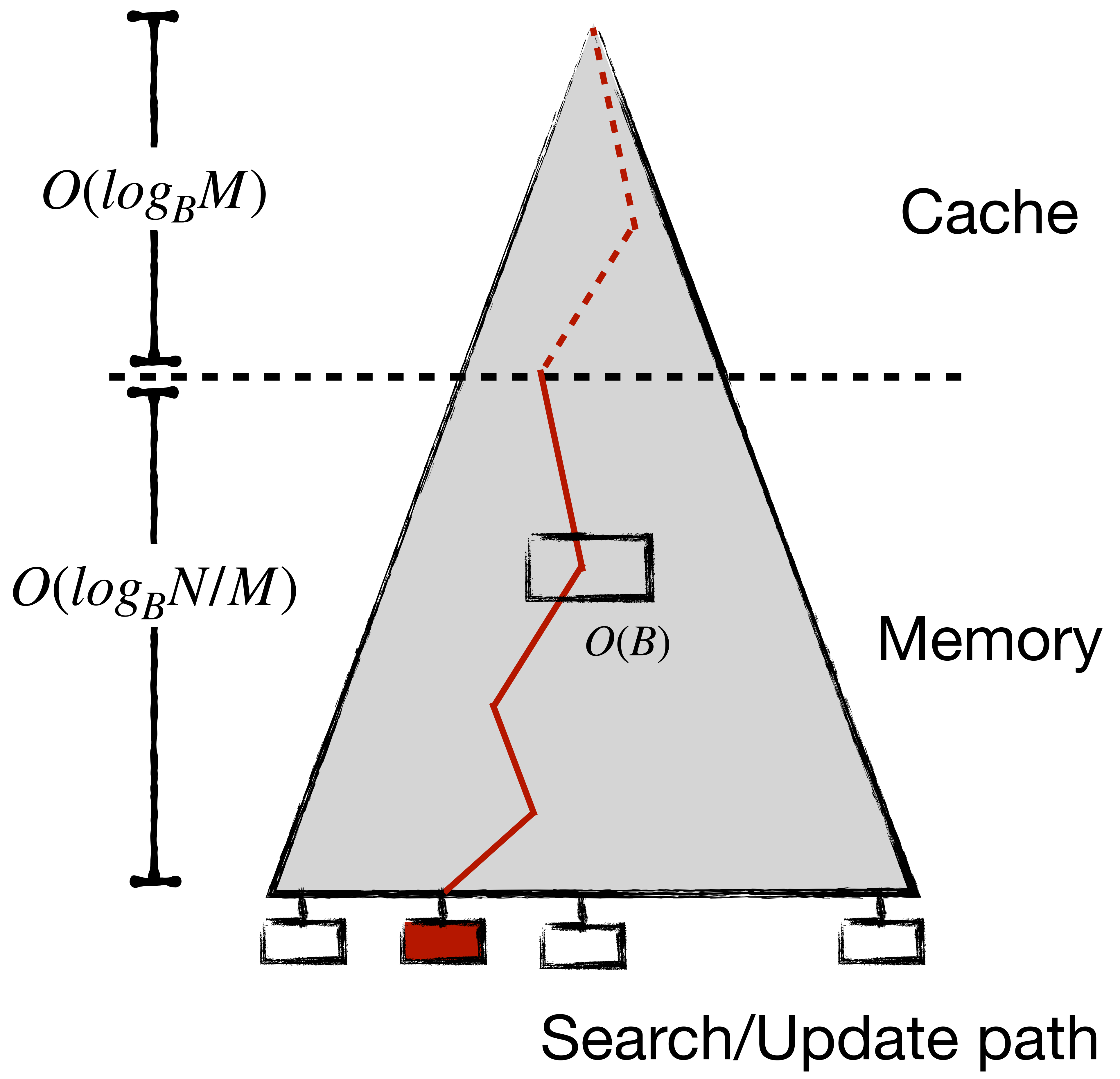
What does B stands for in B-trees?
***Boeing, balanced, between, broad, bushy, Beyer*?**

Cost of operations in B-trees

Insert

Search

$\}$ $O(log_B N/M)$ I/Os

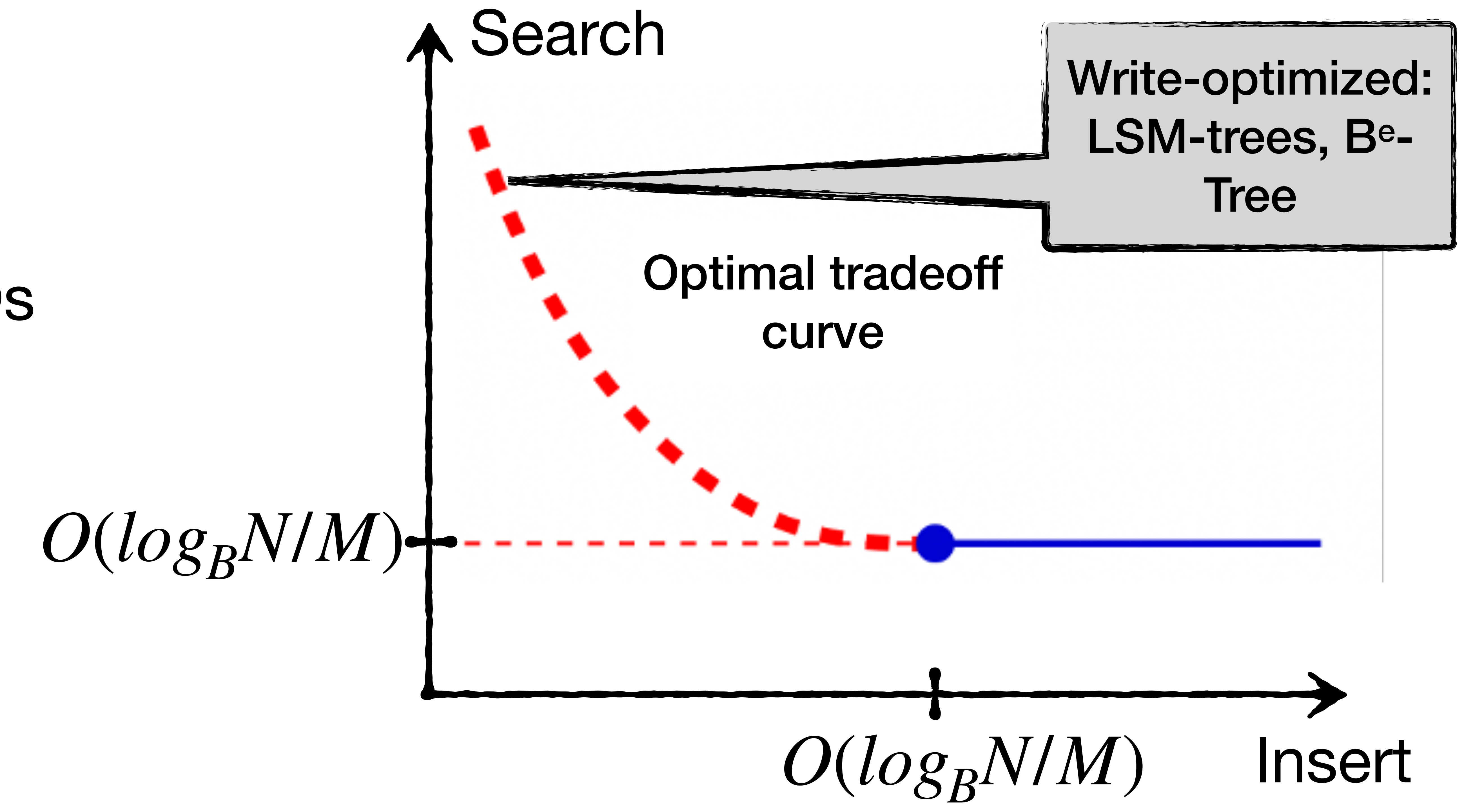


B-trees: trade-off between search and inserts

Insert

Search

$\left.\begin{matrix}\text{Insert}\\ \text{Search}\end{matrix}\right\} O(log_B N/M)$ I/Os

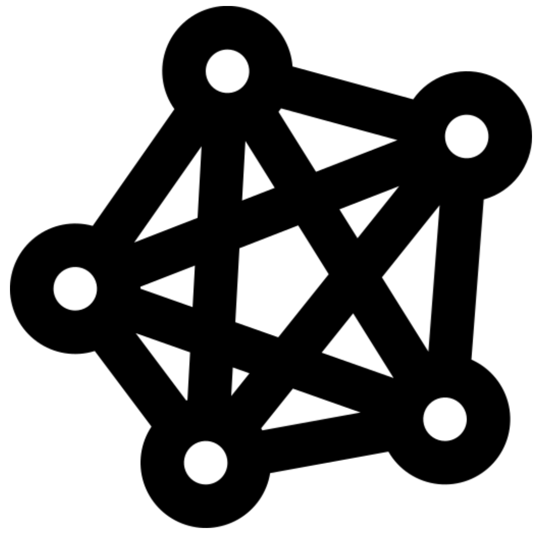


B-trees are asymptotically optimal for point operations [BF03]

In this talk: trade-off between point and range operations in in-memory B-trees

Long range scans are critical in applications

Real-time analytics
[PTPH12]



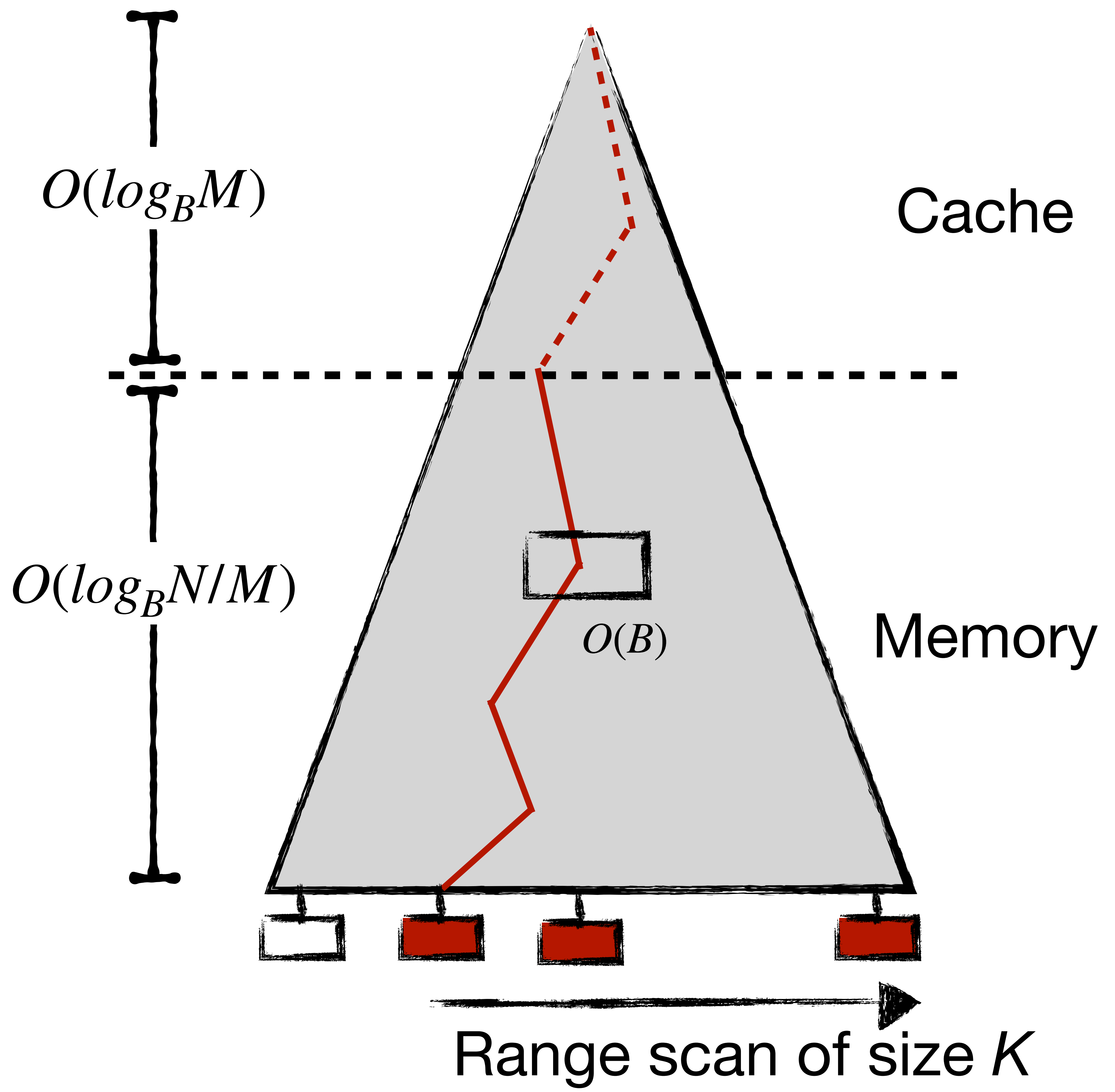
Graph processing
[DBGS22, PWXB21]

Range scan in a B-tree

Range scan $O(log_B N/M + K/B)$ I/Os

Dominates for short ranges

Dominates for long ranges

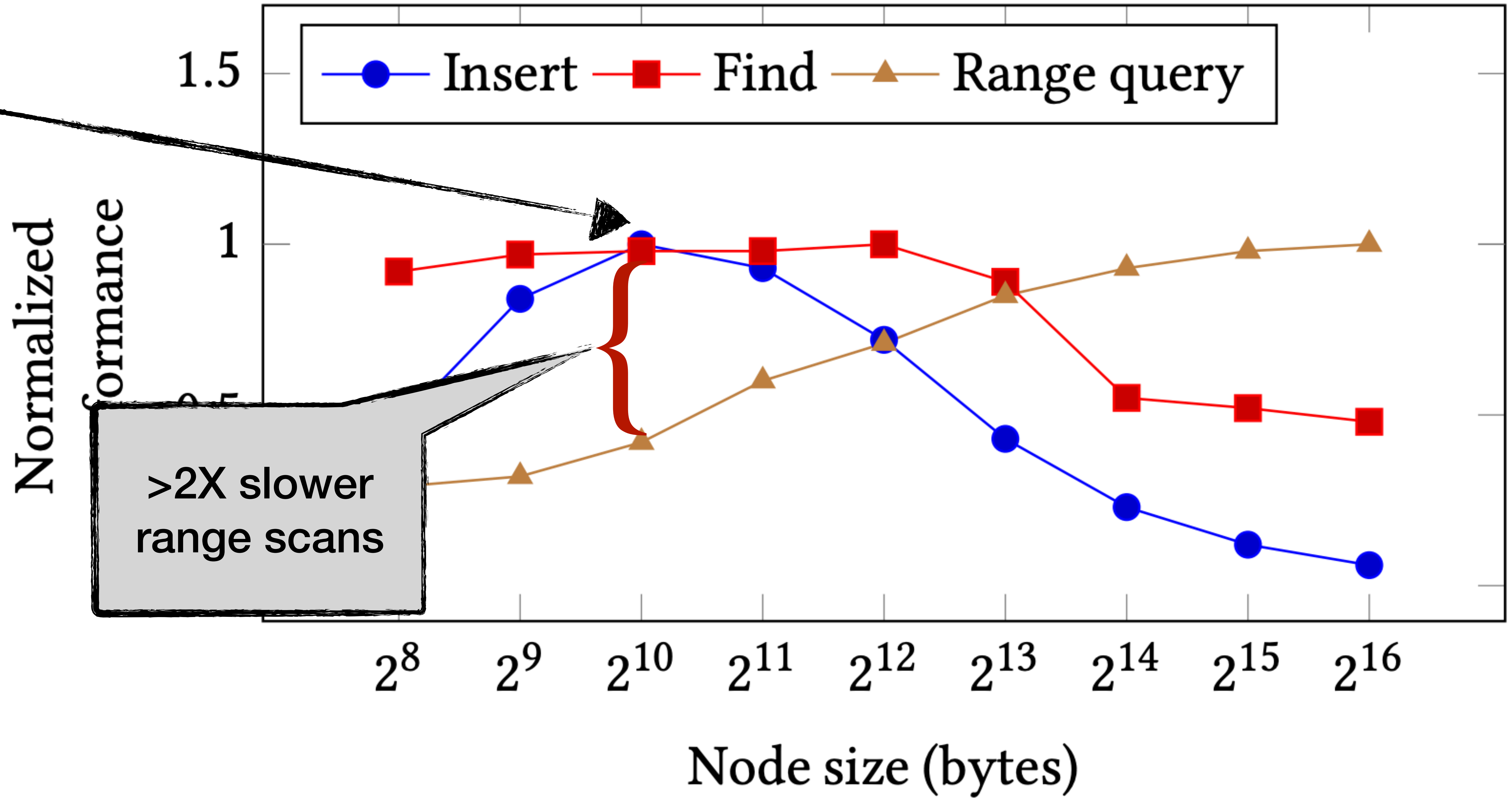


Range scan is optimal in the external memory model

How to choose the node size?

B-trees show a trade-off in point-range operations

Default node size:
1024 Bytes



Large nodes speed up range scans at the cost of point inserts

Supporting fast range scans without sacrificing point update/query performance is a long-standing open problem in B-tree design

Our results: BP-tree [VLDB 2023]

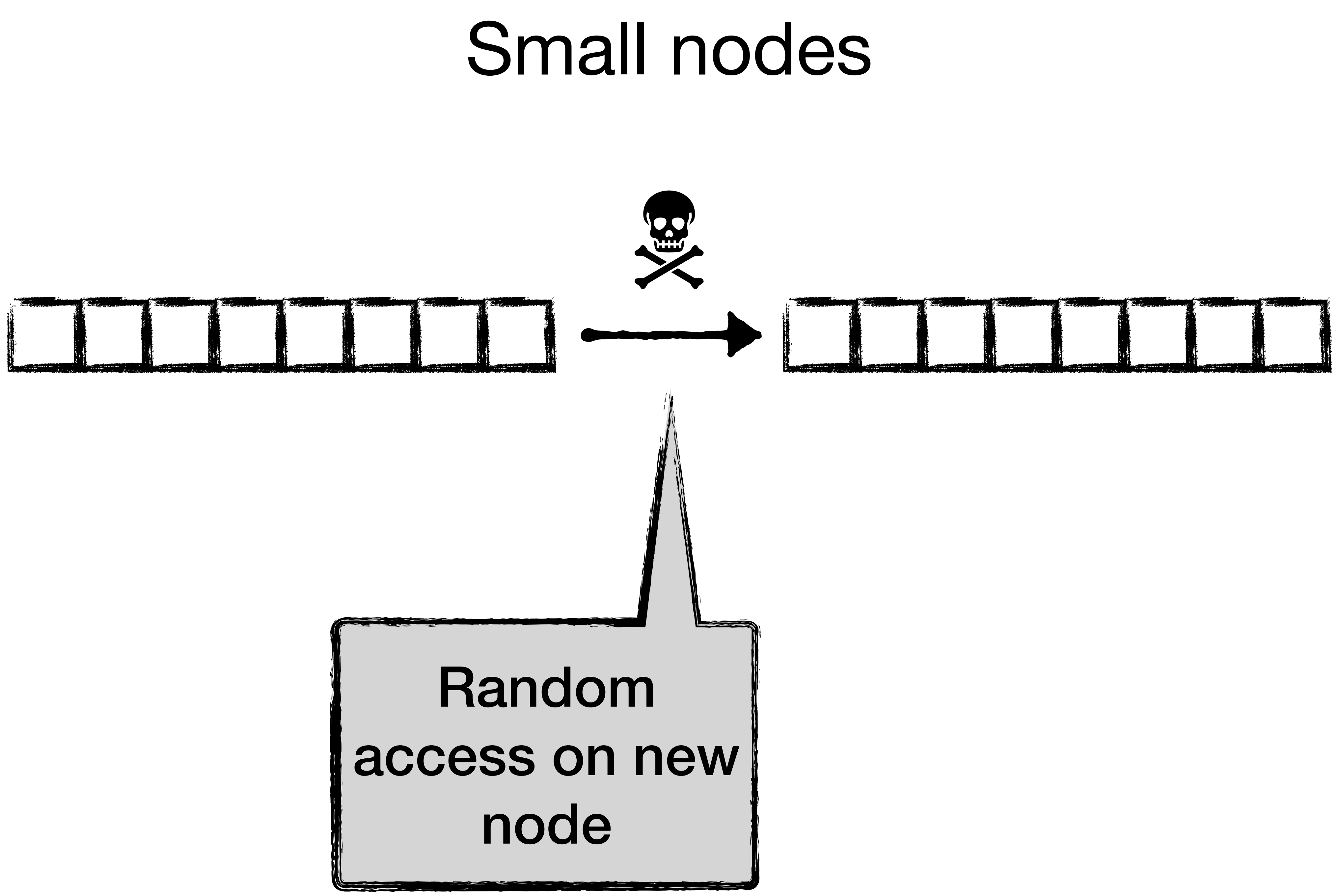
Concurrent C++ implementation

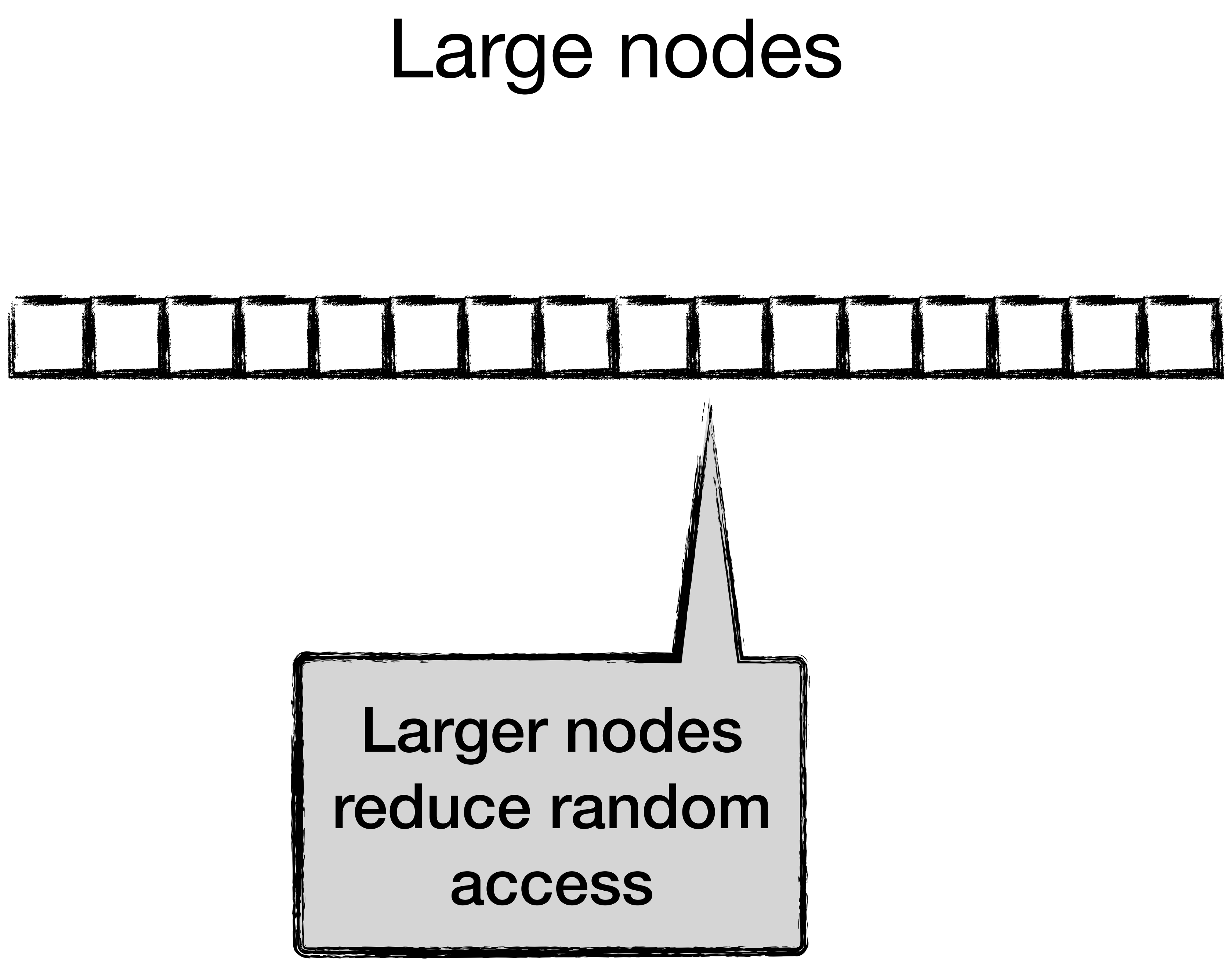
Empirical evaluation using YCSB [CST+10] workloads

Extended YCSB to include long range scans

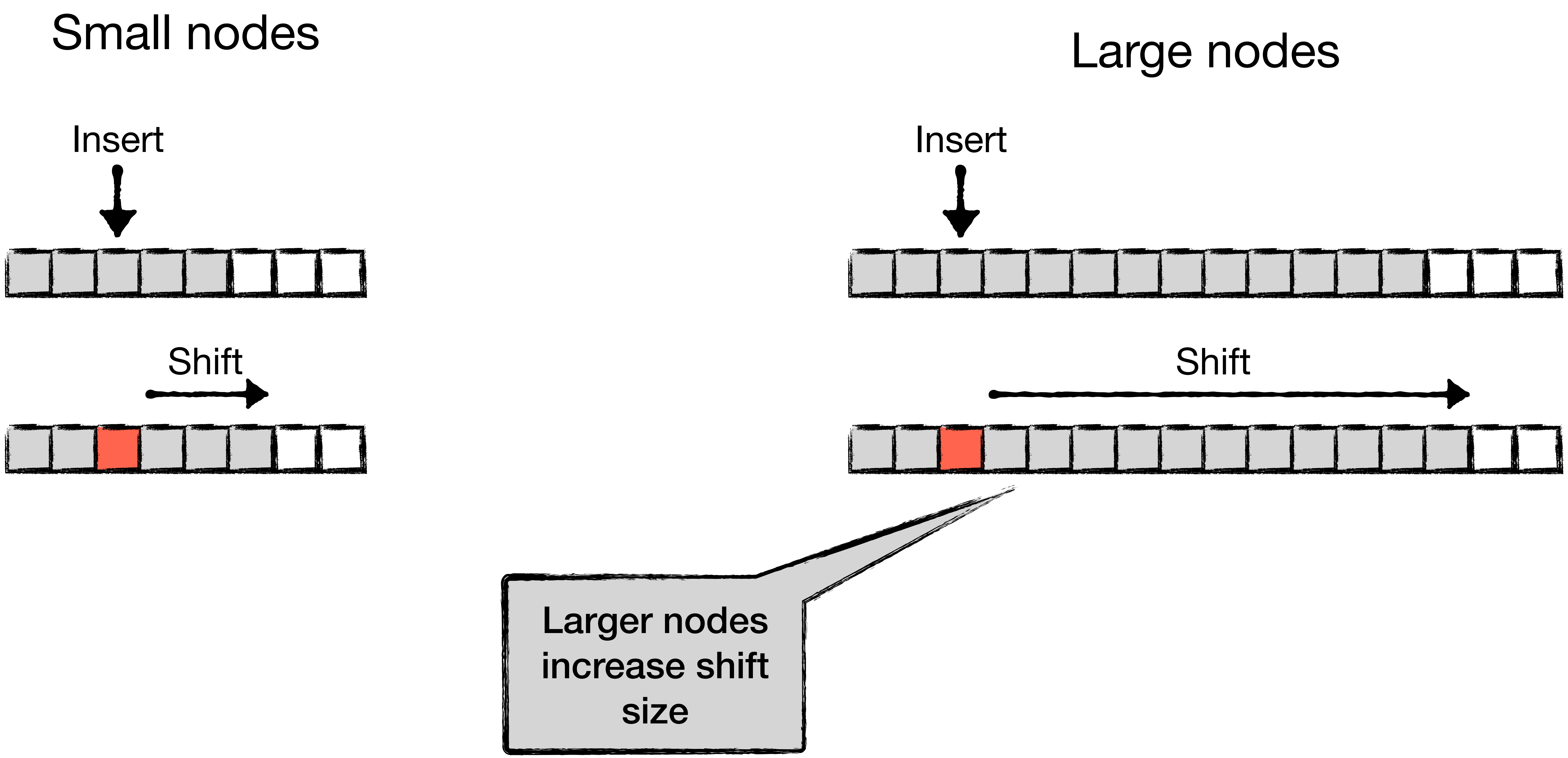
	Point operations	**Range operations**
TLX B-tree [Bingman18]	0.95X — 1.2X faster	1.3X faster
Masstree [MKM12]	0.94X — 7.4X faster	30X faster
OpenBW Tree [WPL+18]	1.2X — 1.6X faster	2.5X faster

Larger nodes improve range scan performance

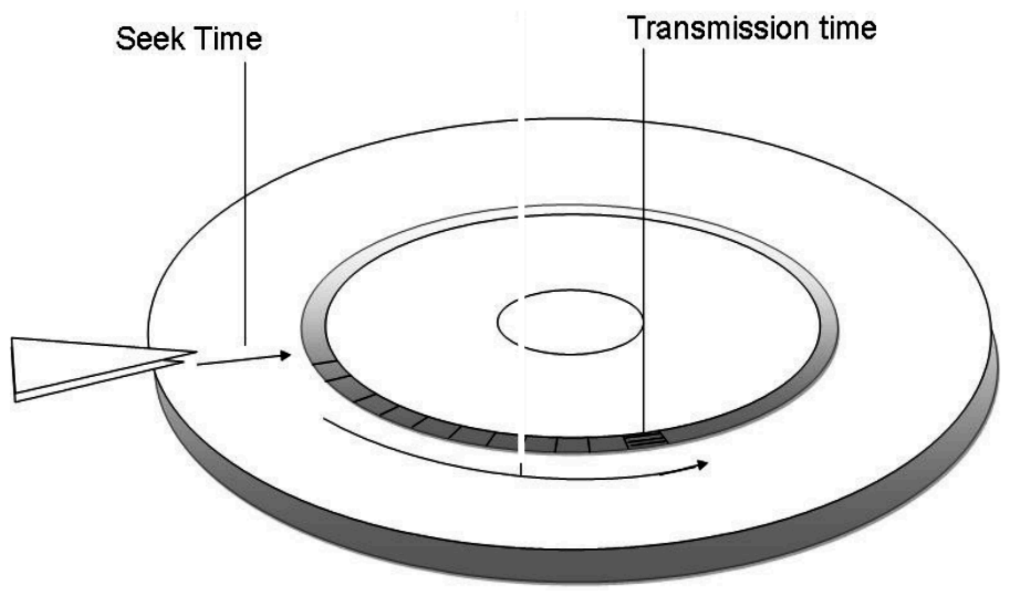




Larger nodes cause overhead to maintain order

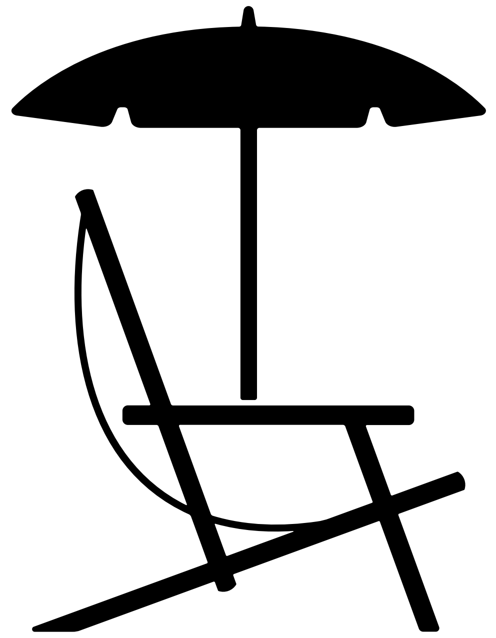


BP-tree design principles



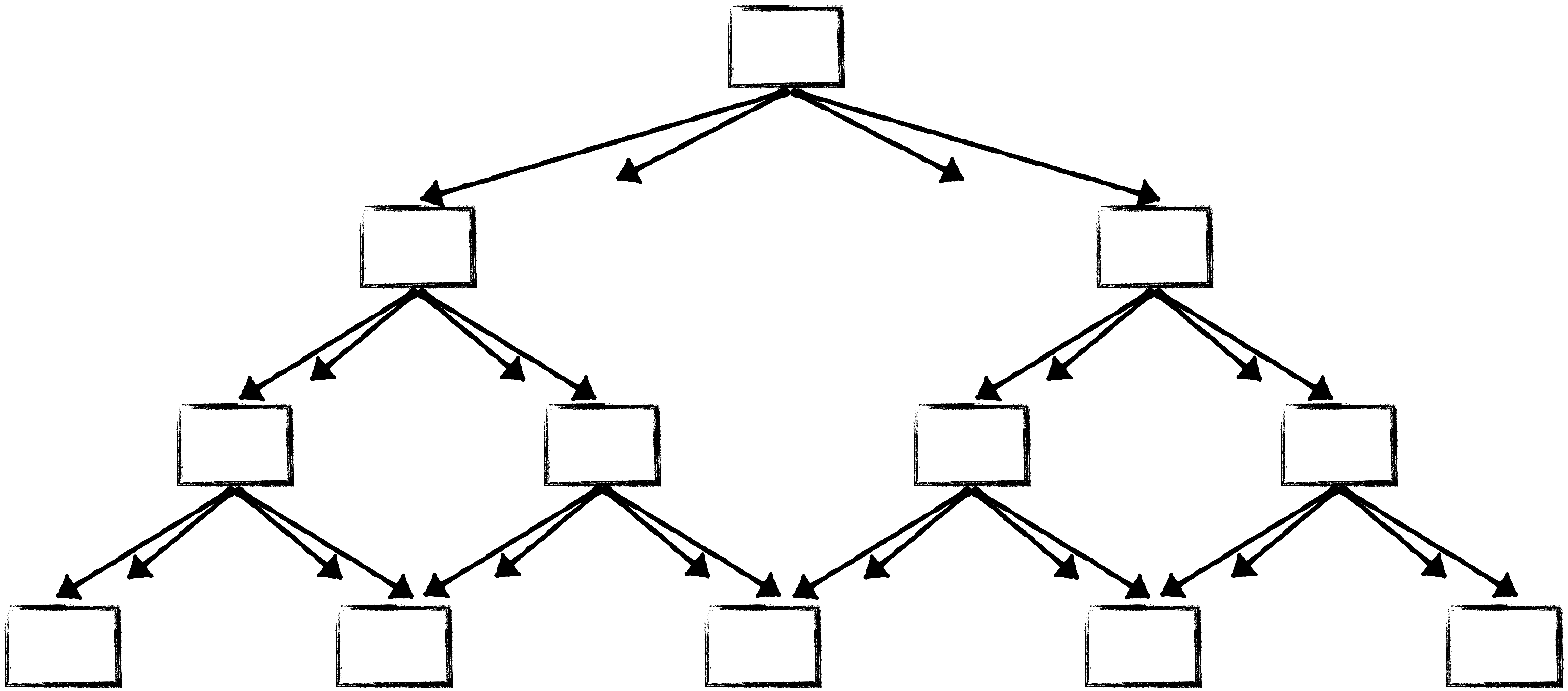
Affine model for performance [BCF+19]

Large leaf nodes

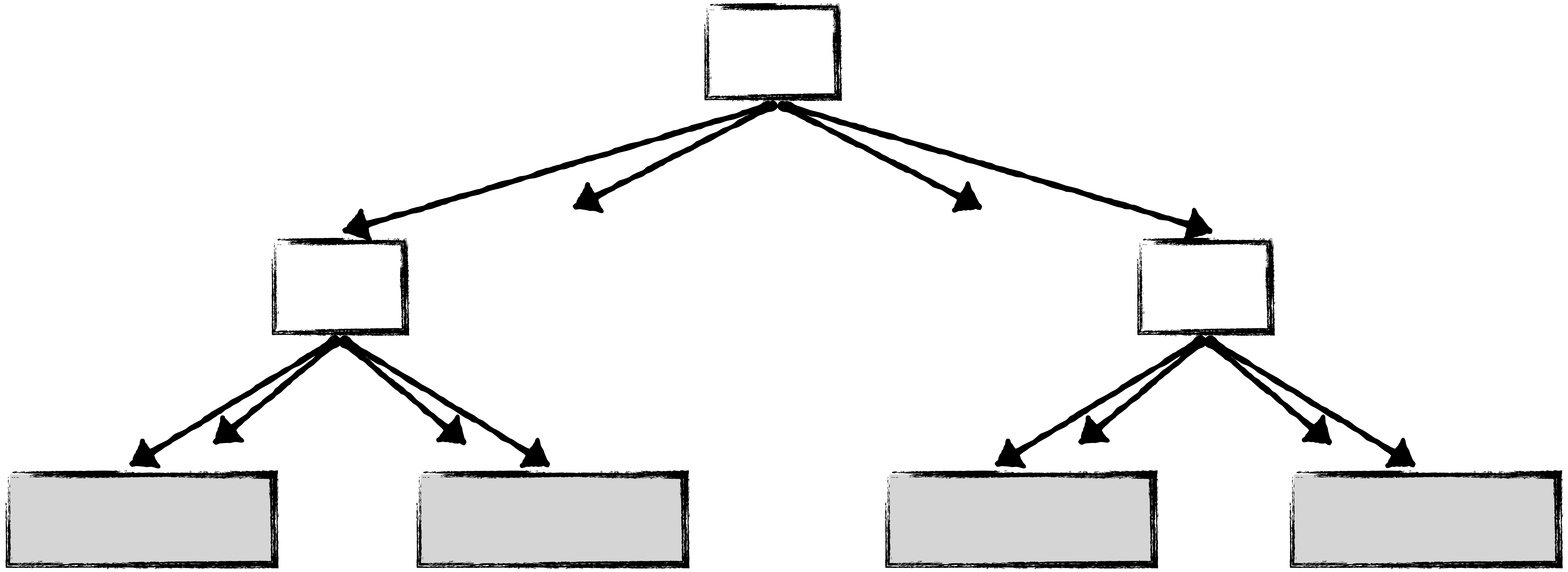


Lazy ordering in leaf nodes

BP-tree design

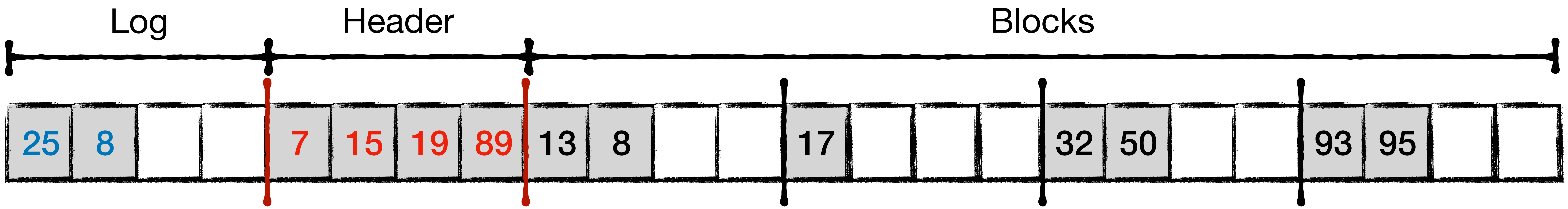


BP-tree design

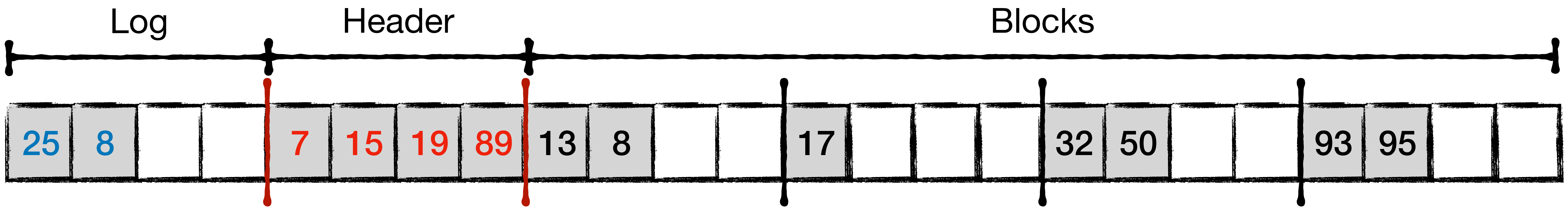


Buffered Partitioned Array:
a special data structure for leaves

Buffered Partitioned Array

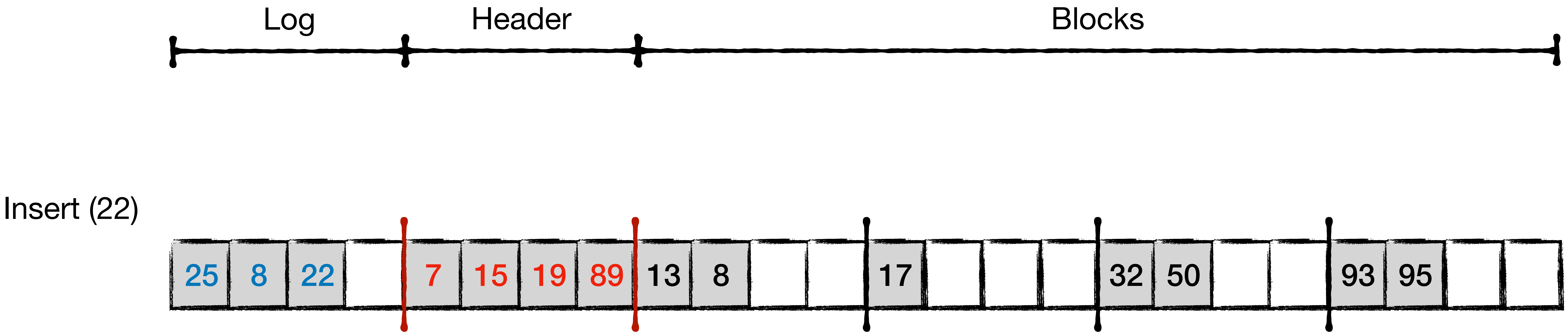


Buffered Partitioned Array

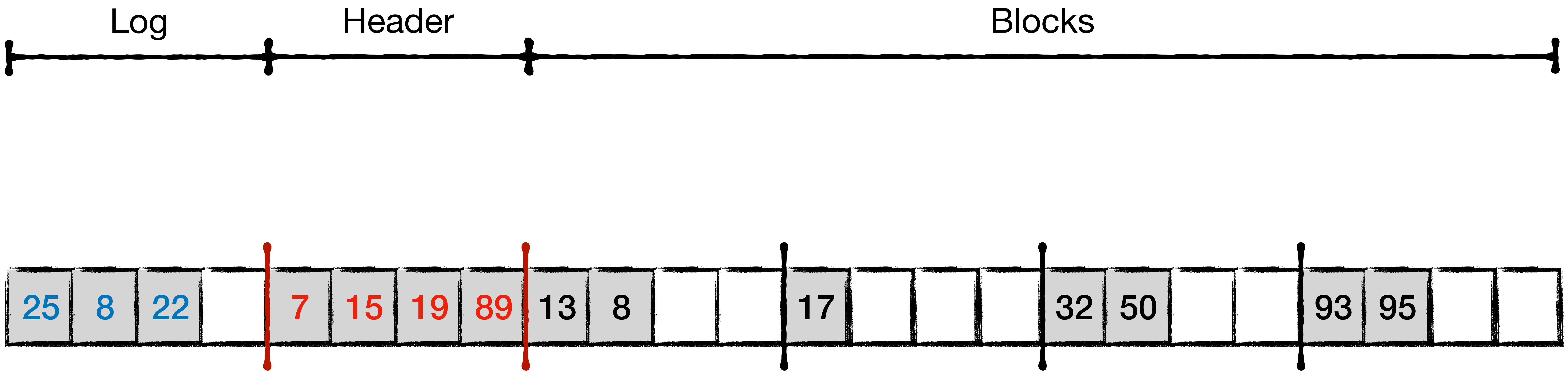


Insert (22)

Buffered Partitioned Array

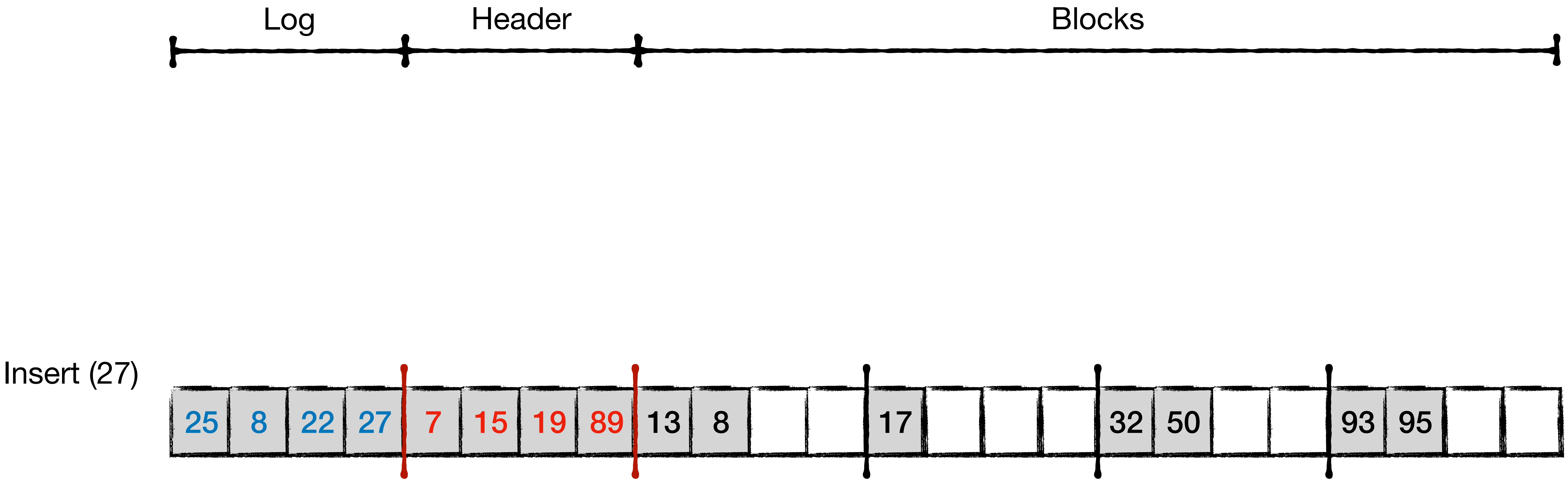


Buffered Partitioned Array

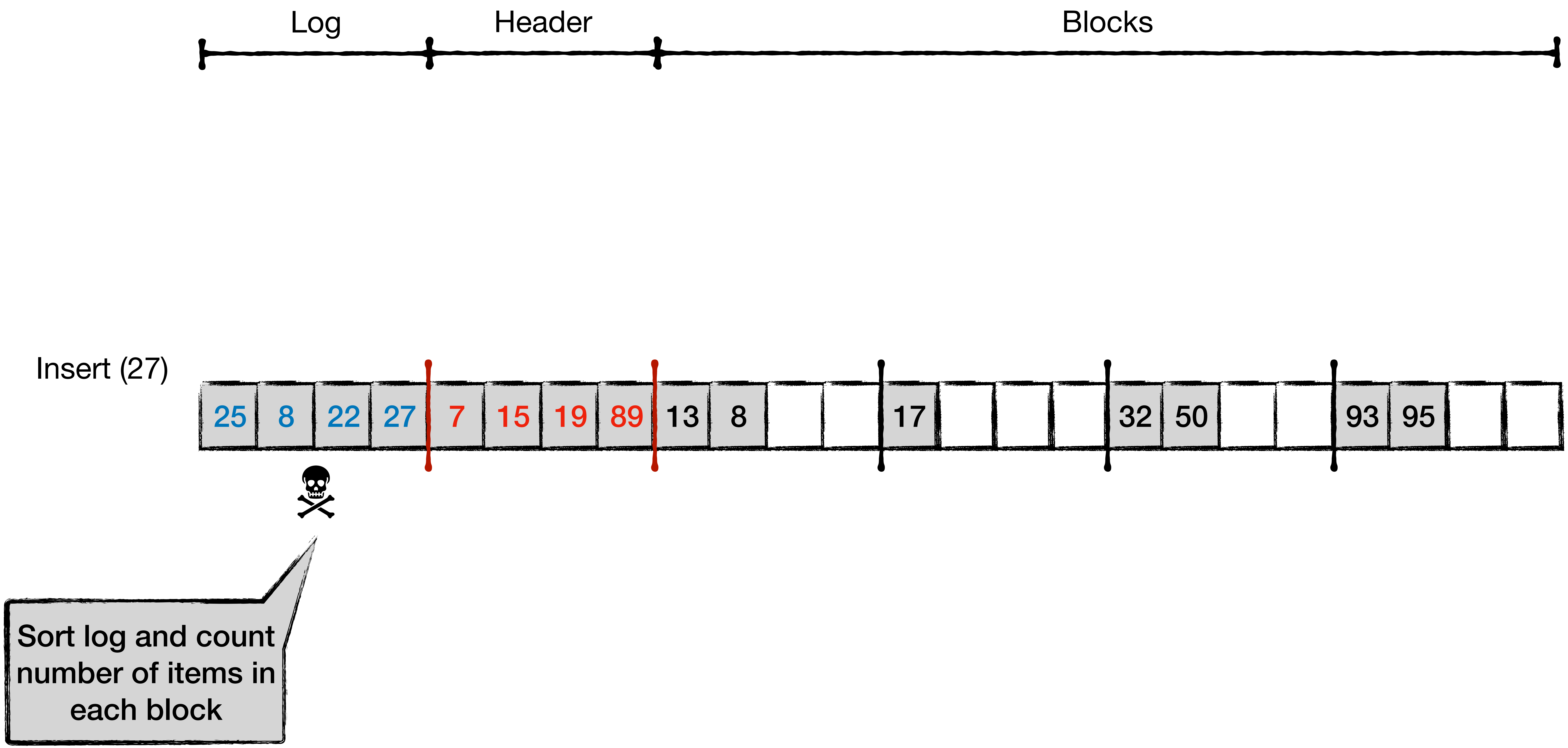


Insert (27)

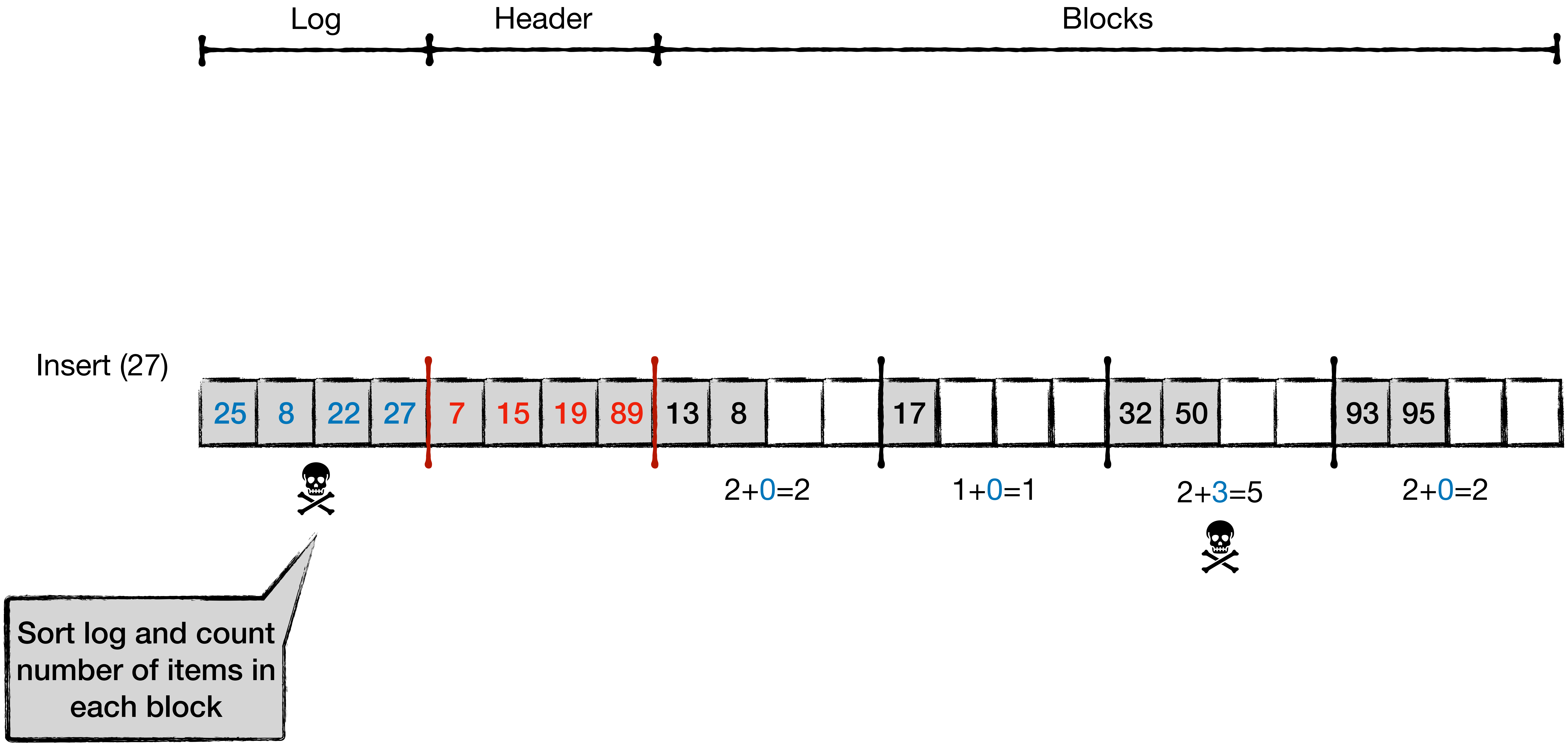
Buffered Partitioned Array



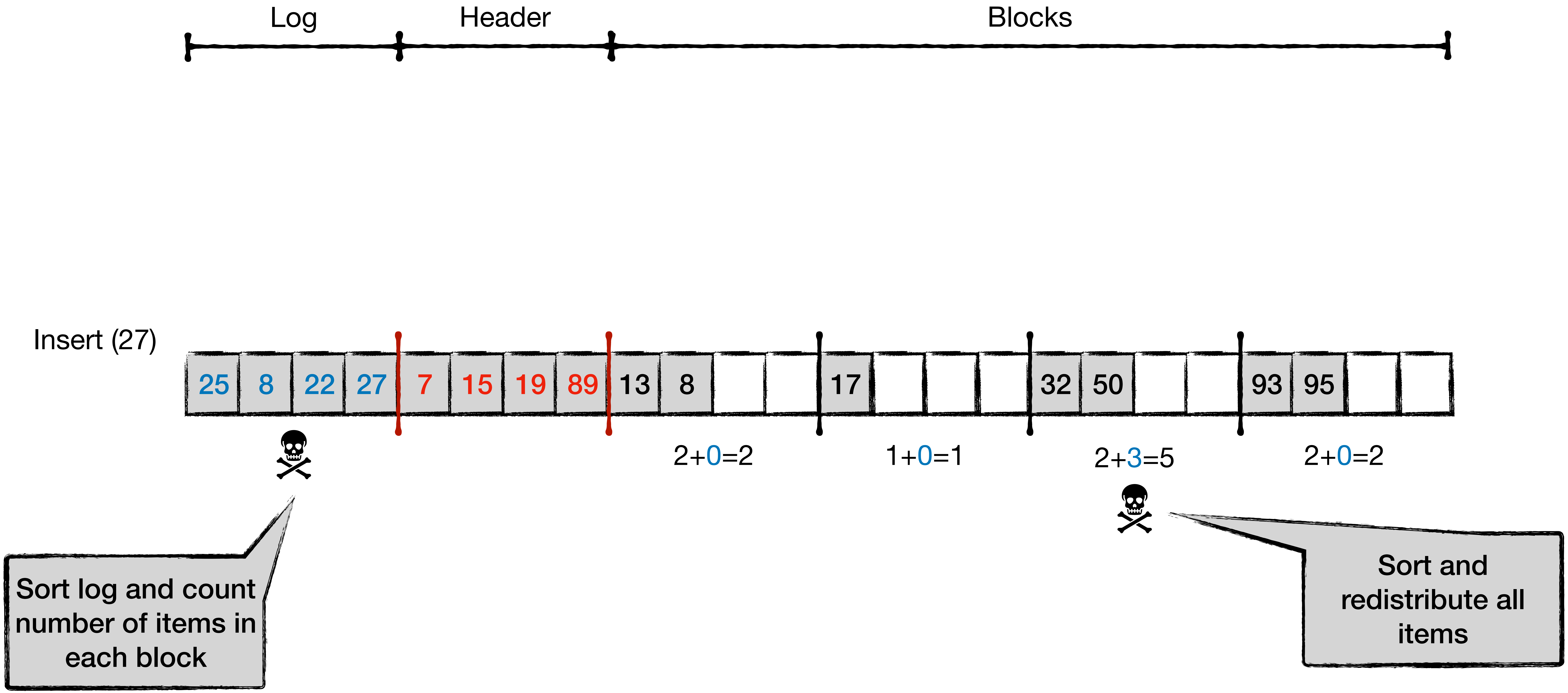
Buffered Partitioned Array



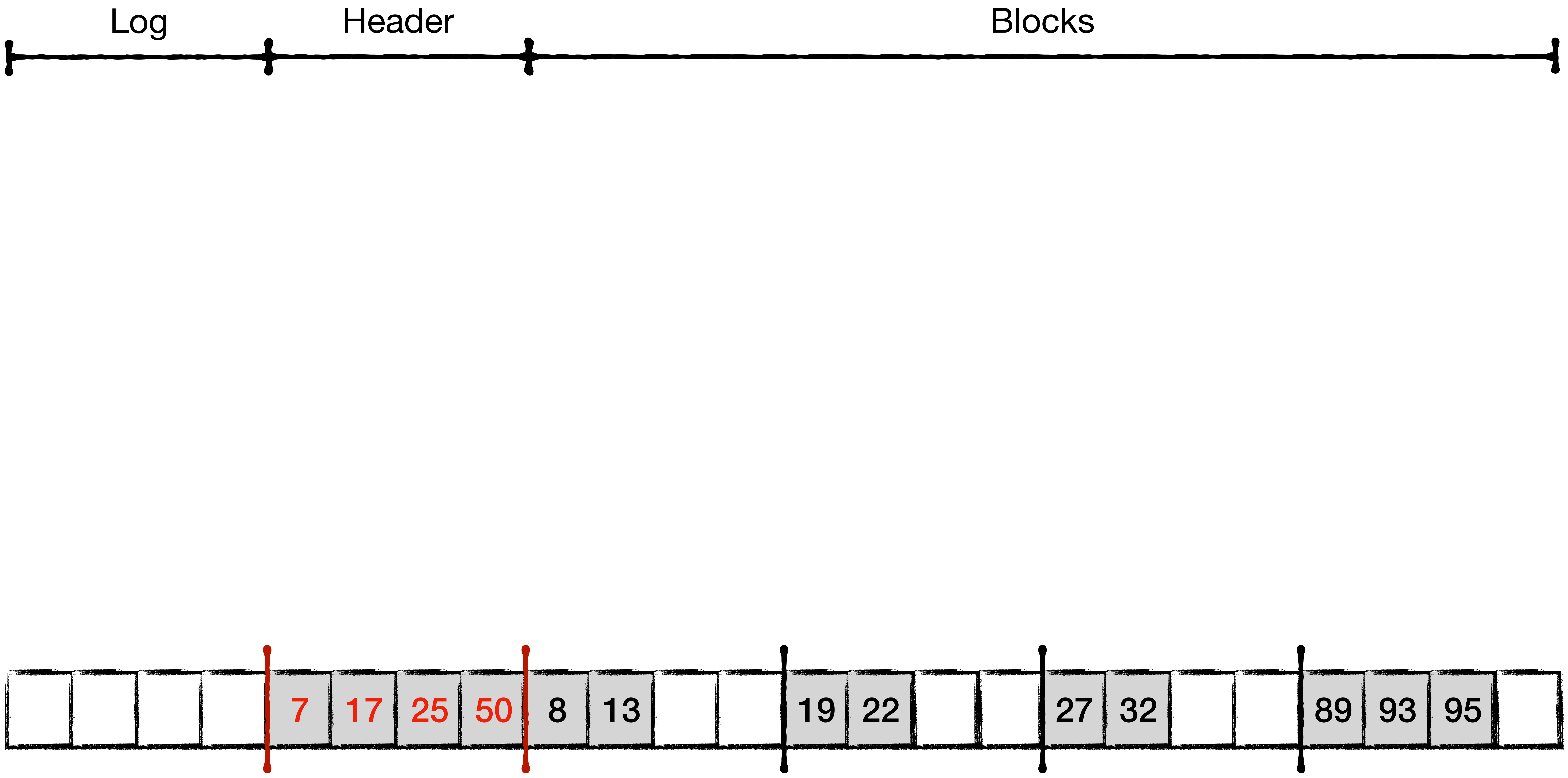
Buffered Partitioned Array



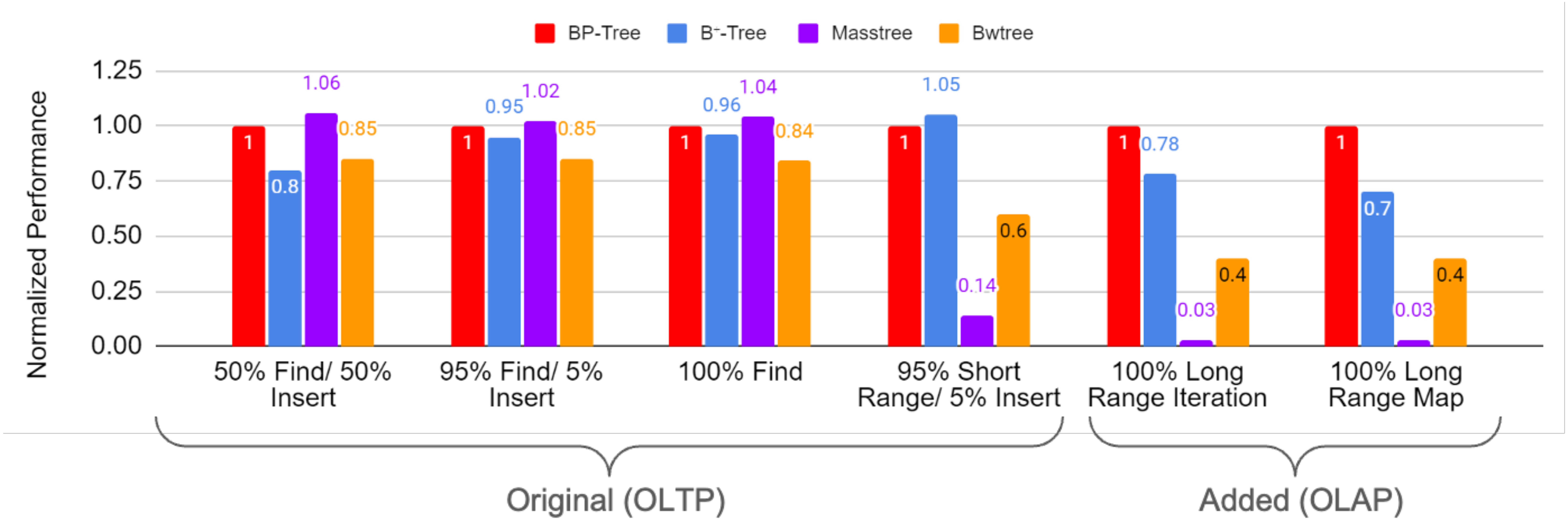
Buffered Partitioned Array



Buffered Partitioned Array



Performance YCSB workloads



BP tree matches on point operations while being 2X faster for range scans

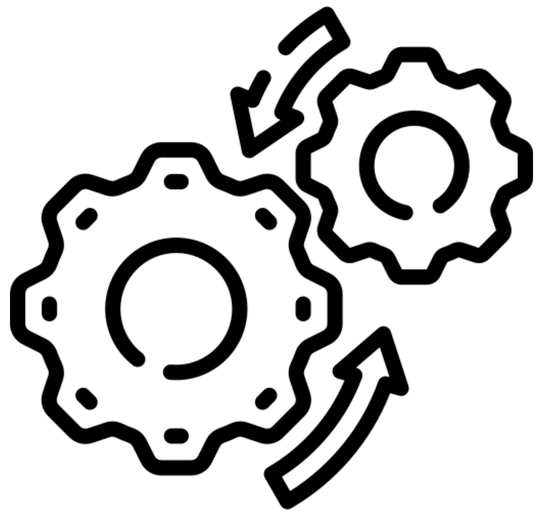
BP-trees takeaways

- I/O models (External memory and Affine) apply to in-memory indexes
- Relaxing ordering constraint in leaf nodes can help overcome traditional tradeoffs
- BP-tree supports fast range scans (OLAP) an optimal point updates/ queries (OLTP)

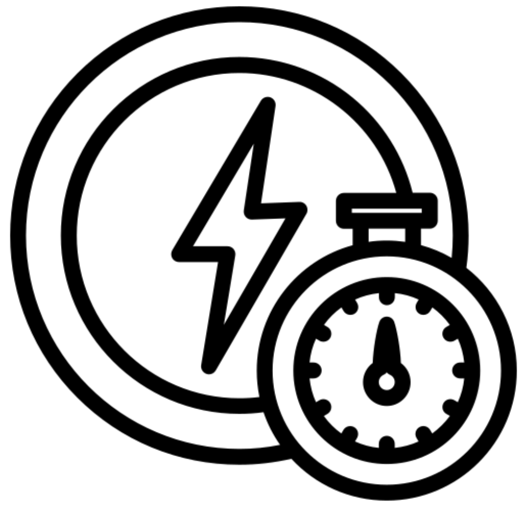
Source code:https://github.com/wheatman/BP-Tree

Ongoing research:

GPU-accelerated
vector databases



Dynamic/Scalable GPU
memory management



Energy efficient data
management

Concluding remarks

- We need to develop new algorithmic paradigms to better leverage modern hardware
- Data systems backed by strong theoretical guarantees are key to tackle future data analyses challenges

https://prashantpandey.github.io/

Acknowledgements:

National Science Foundation

U.S. DEPARTMENT OF ENERGY