BP-tree

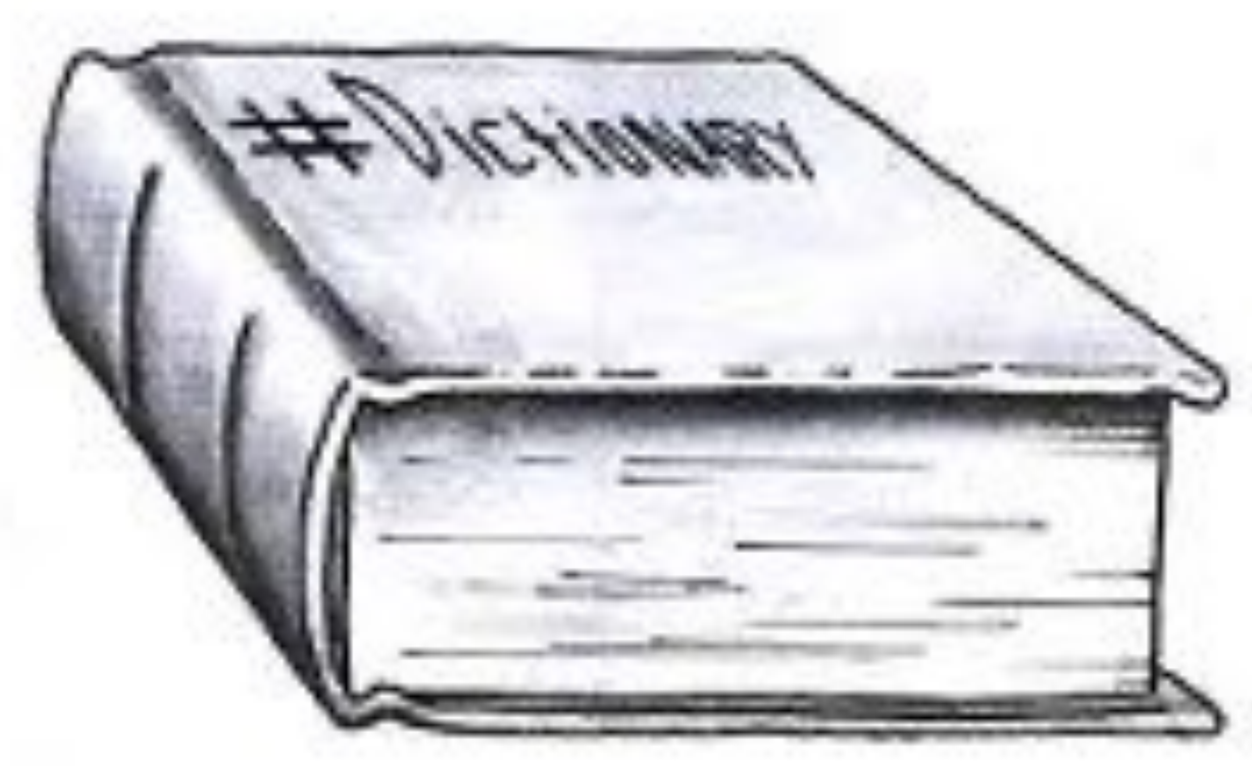
Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees

VLDB 2023

Prashant Pandey, University of Utah
Joint work with: Helen Xu (Georgia Tech), Amanda Li (MIT), Brian Wheatman (JHU), Manoj Marneni (Utah)

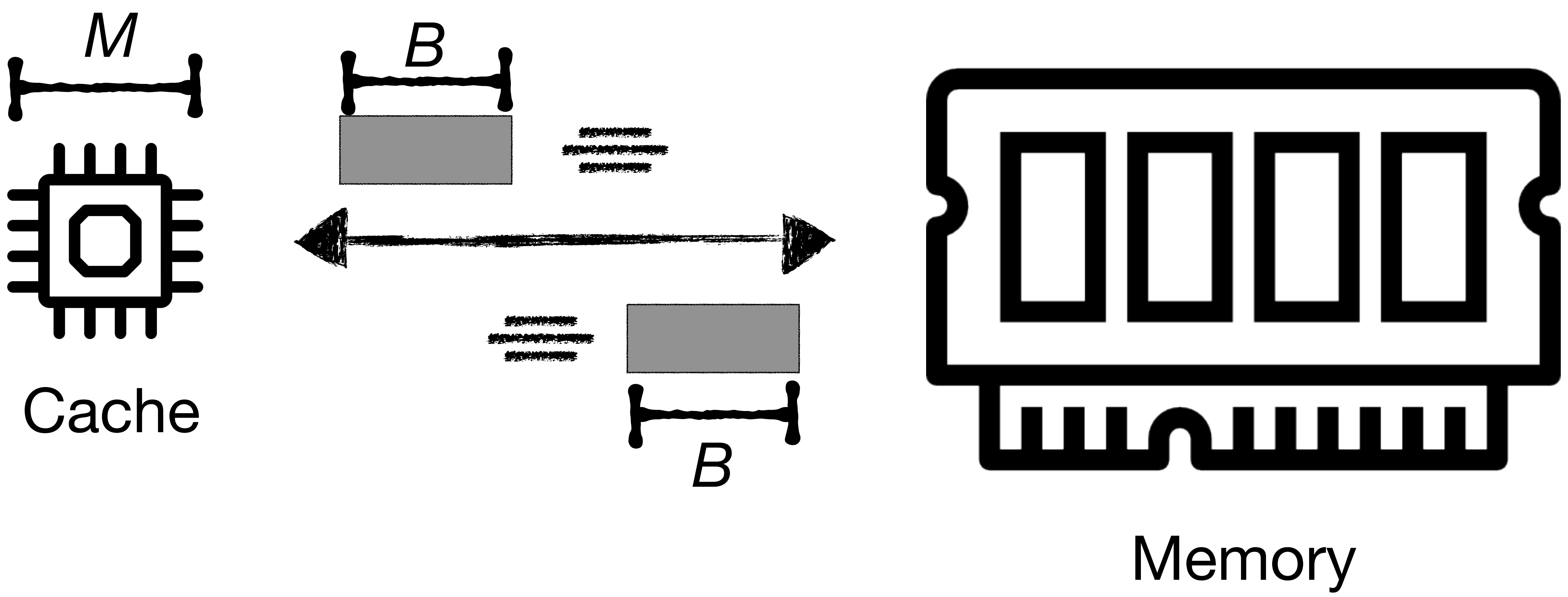
Dictionary data structures

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



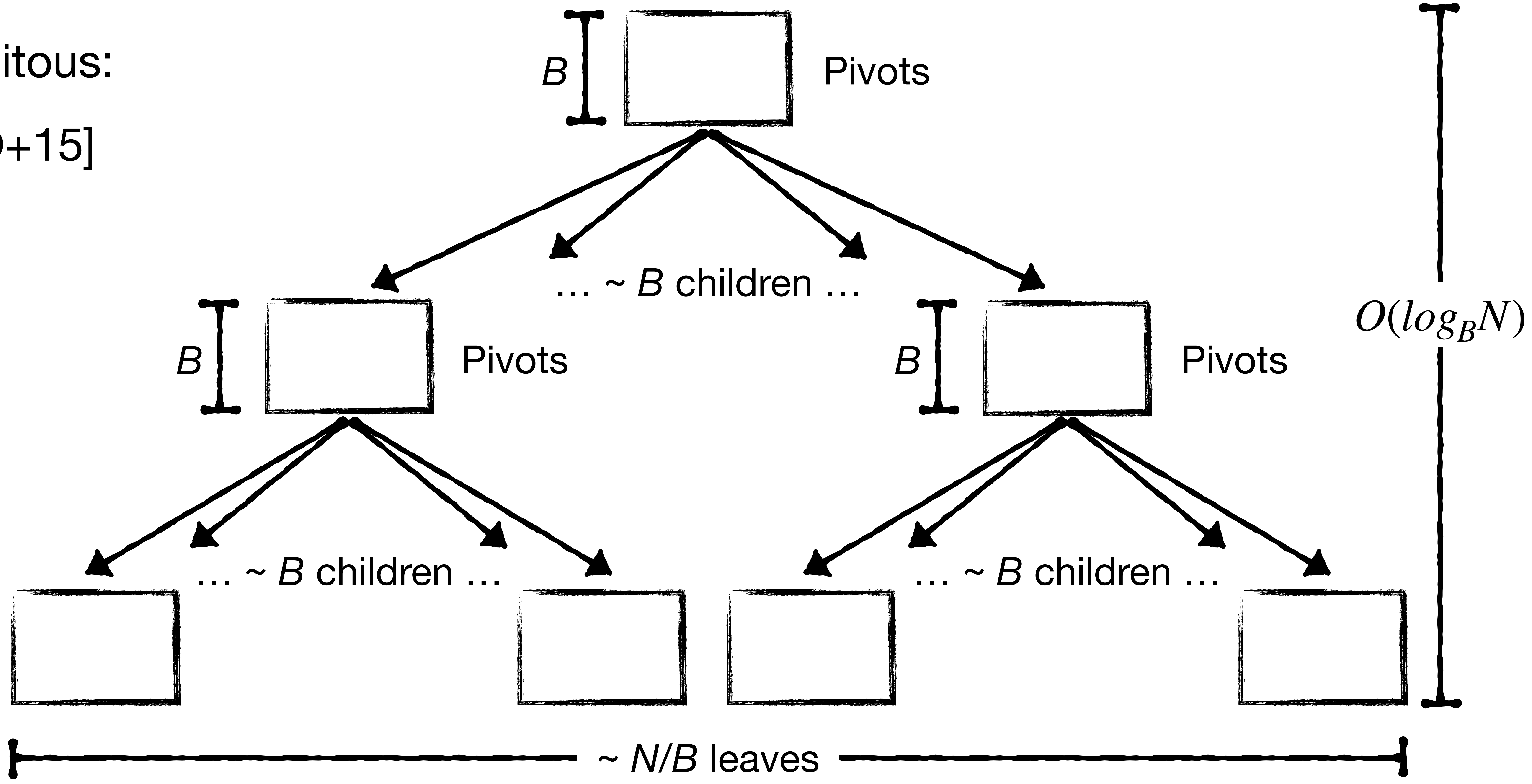
External memory model for dictionaries

- **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*



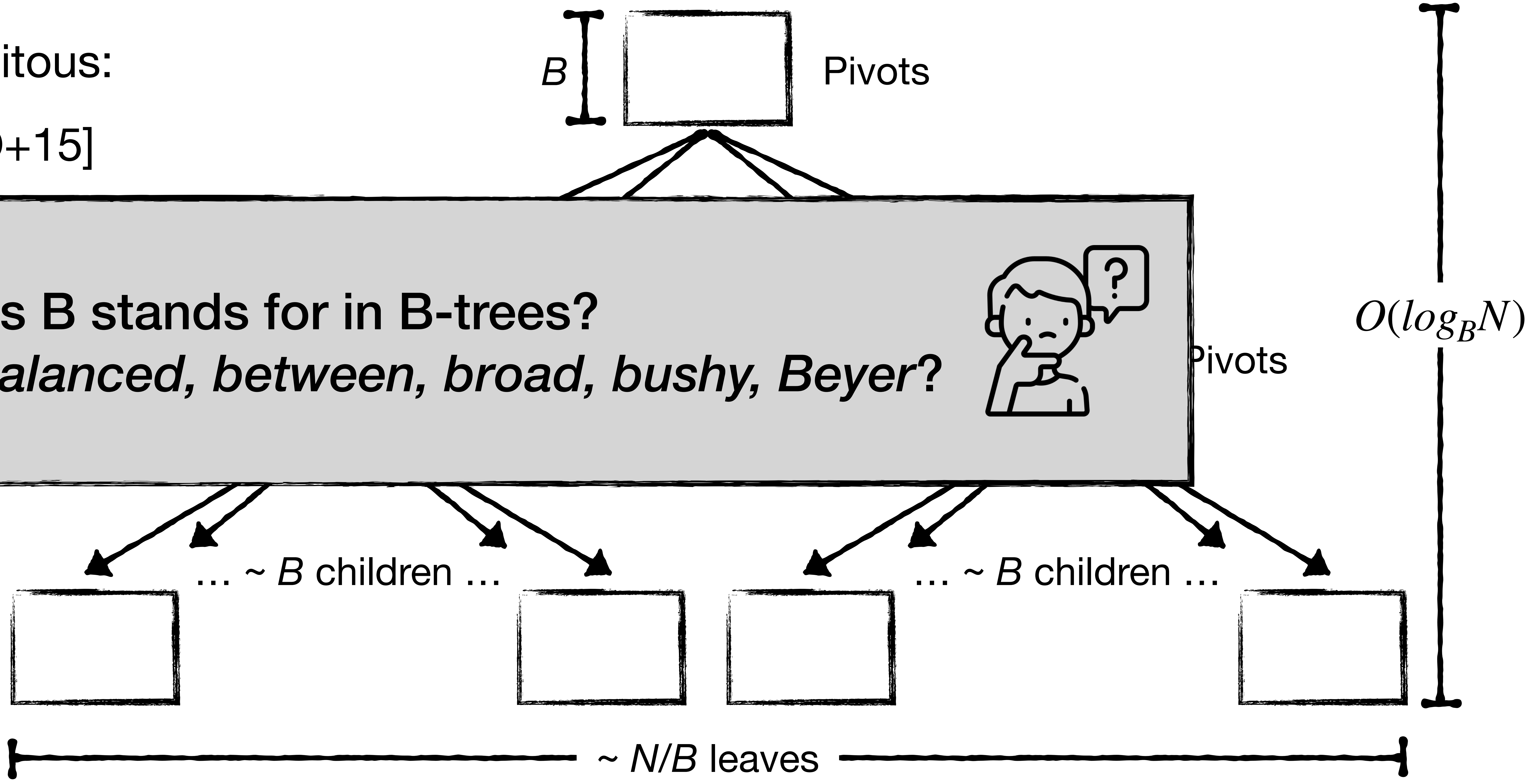
B-tree: a classic dictionary data structure

- B/B+-trees [BM72] are ubiquitous:
  - In memory indexing [ZCO+15]
  - Databases [K98]
  - Filesystems [RBM13]



B-tree: a classic dictionary data structure

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



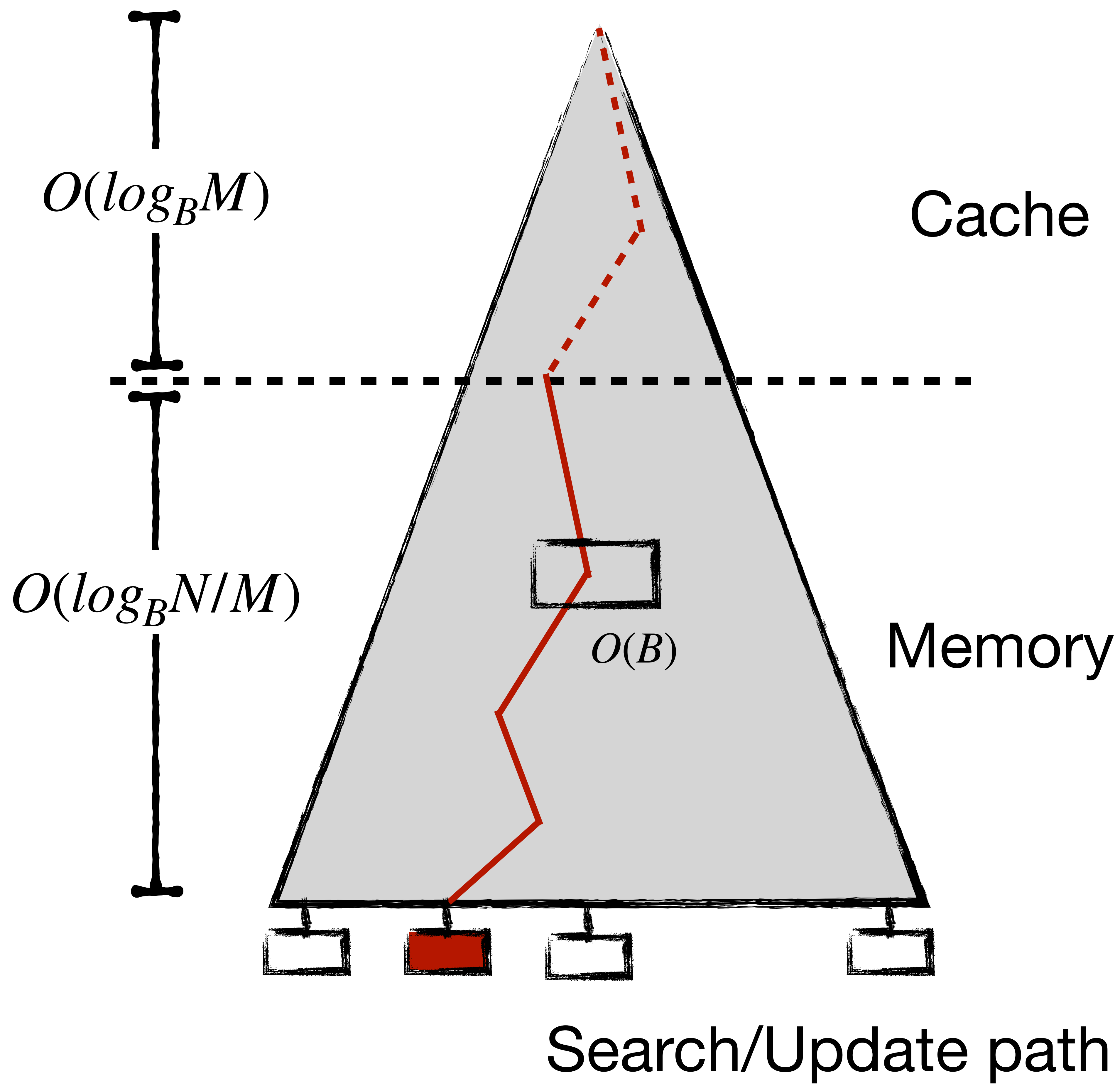
What does B stands for in B-trees?
***Boeing, balanced, between, broad, bushy, Beyer*?**

Cost of operations in B-trees

Insert

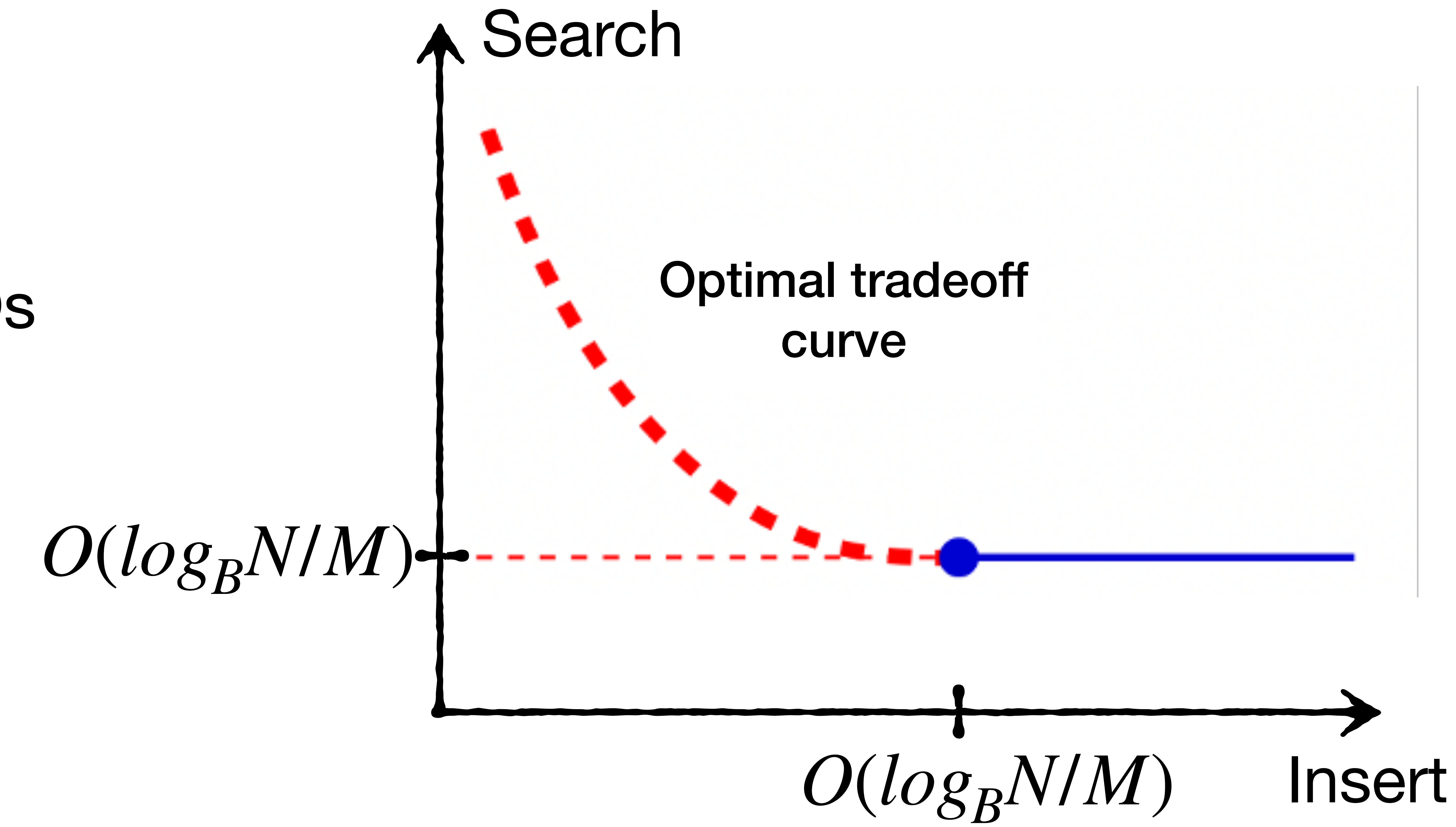
Search

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



B-trees: tradeoff between search and inserts

Insert
Search
} $O(log_B N/M)$ I/Os



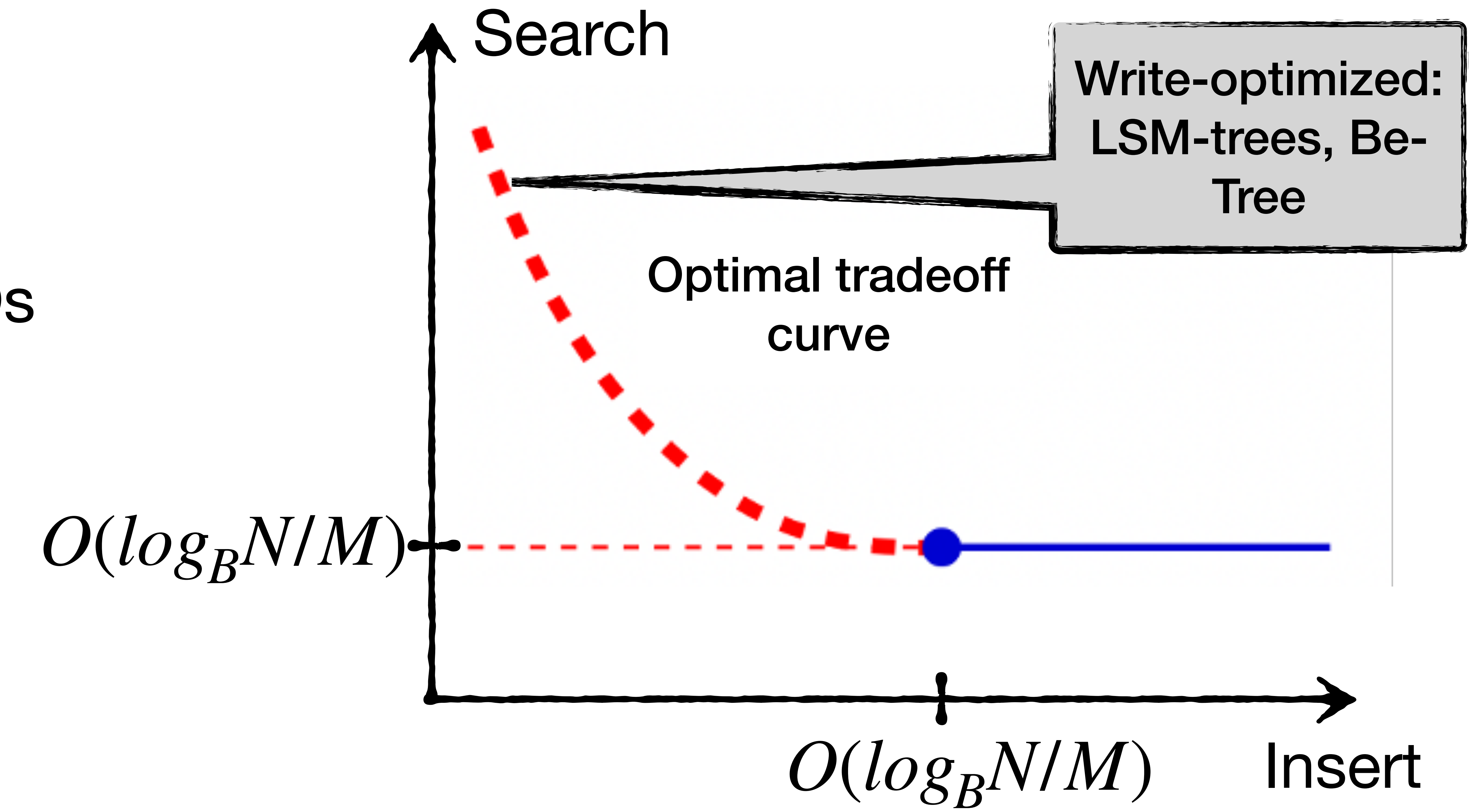
B-trees are asymptotically optimal for point operations [BF03]

B-trees: tradeoff between search and inserts

Insert

Search

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

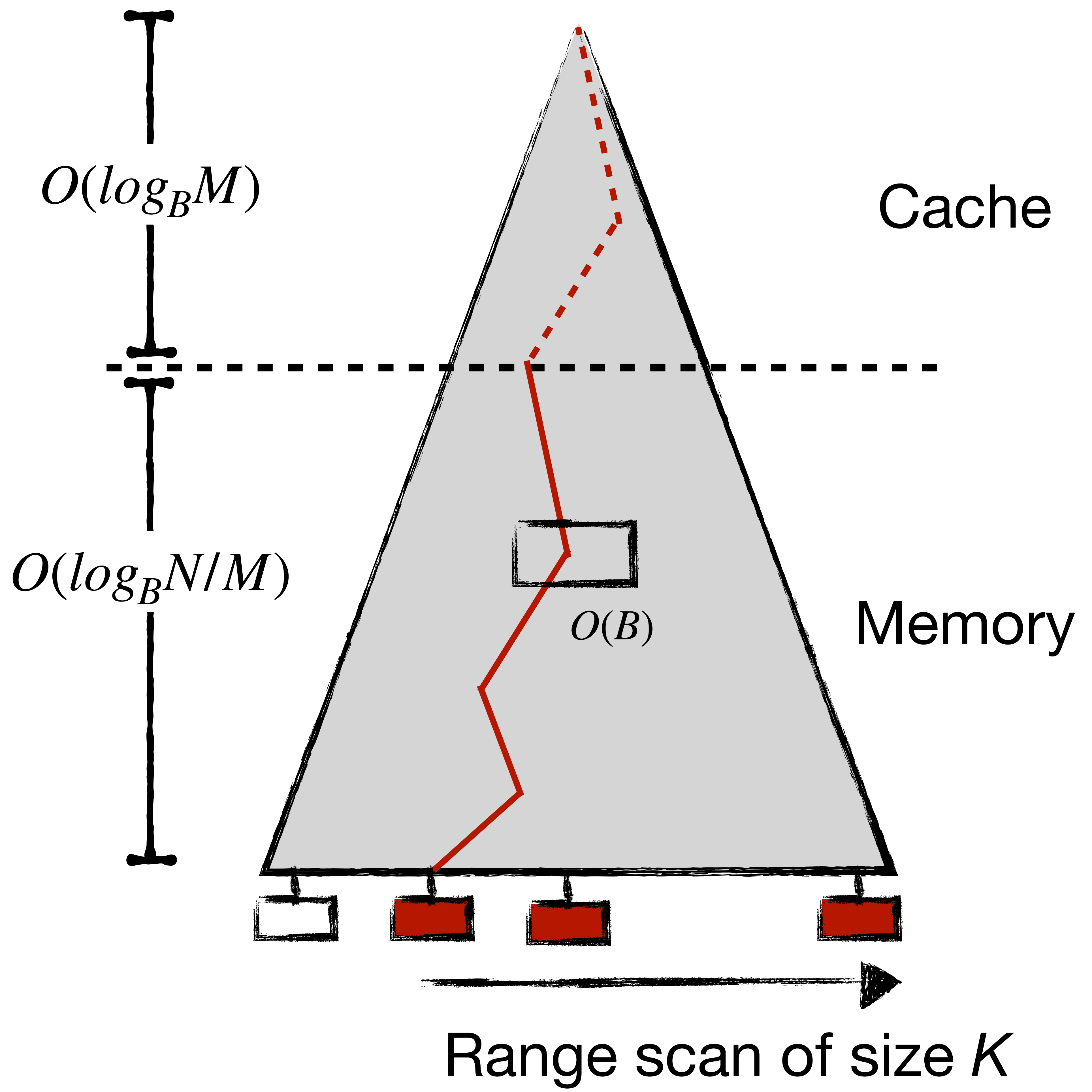


B-trees are asymptotically optimal for point operations [BF03]

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

Range scan in a B-tree

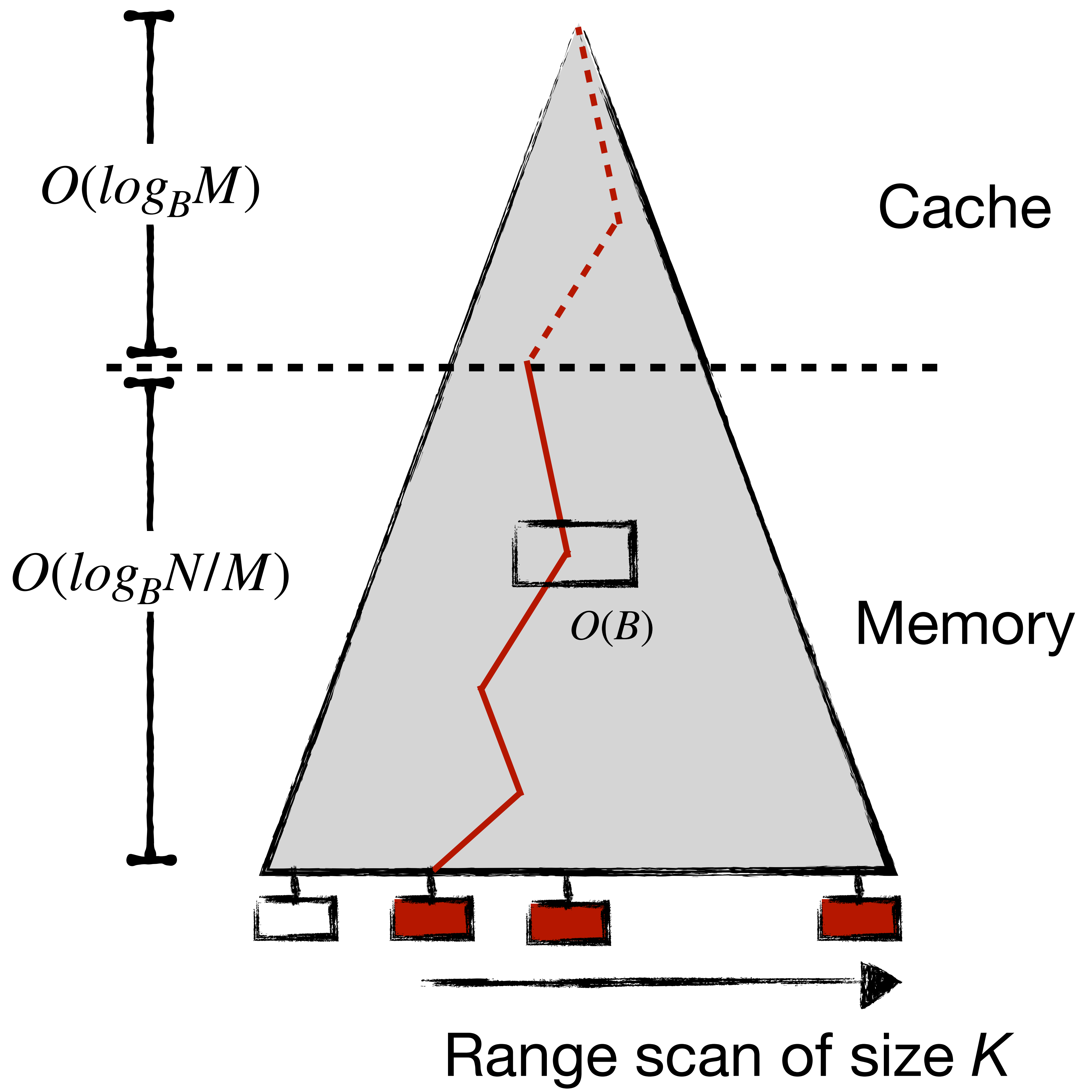
Range scan $O(log_B N/M + K/B)$ I/Os



Range scan in a B-tree

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

Dominates for short ranges

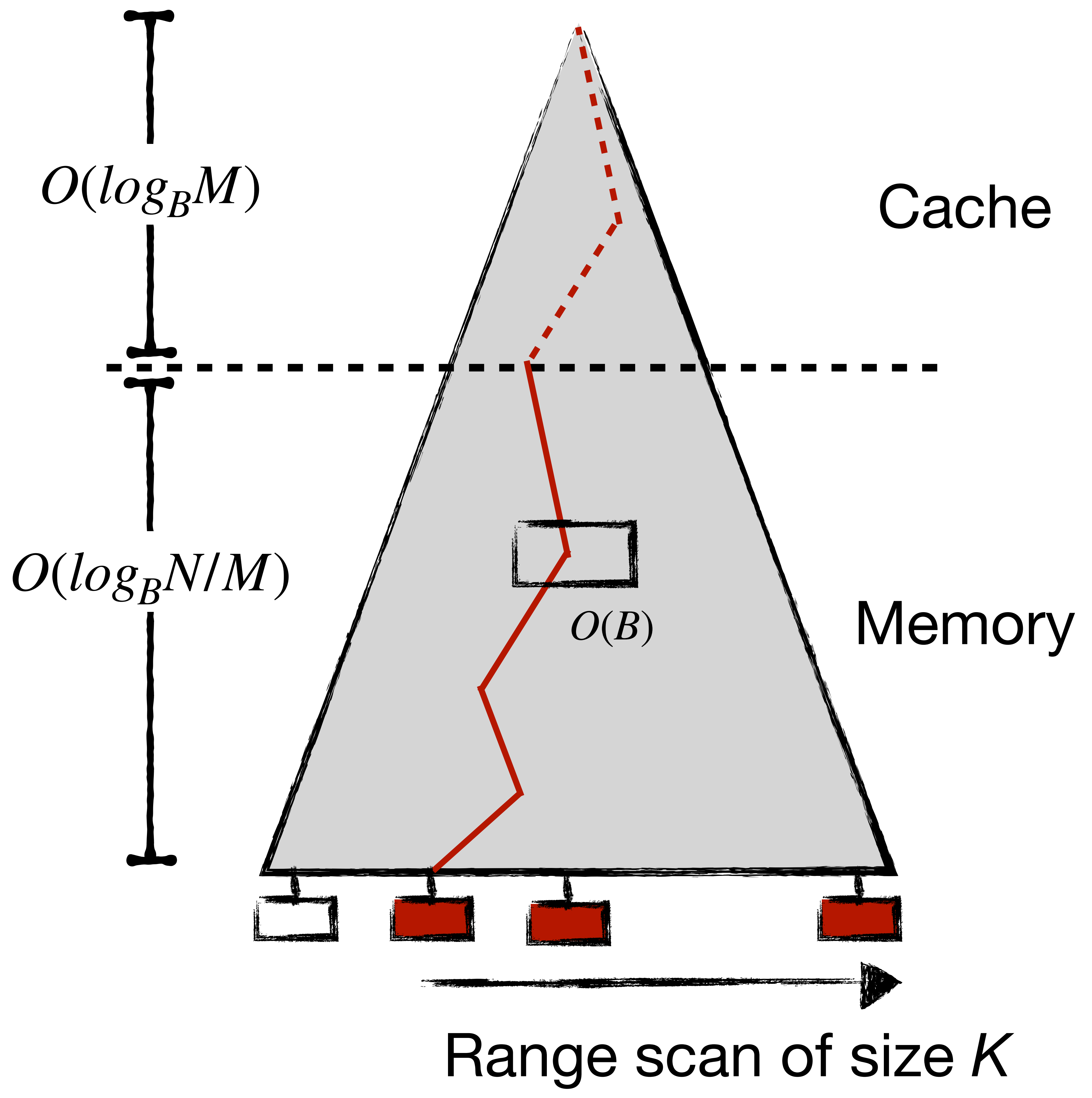


Range scan in a B-tree

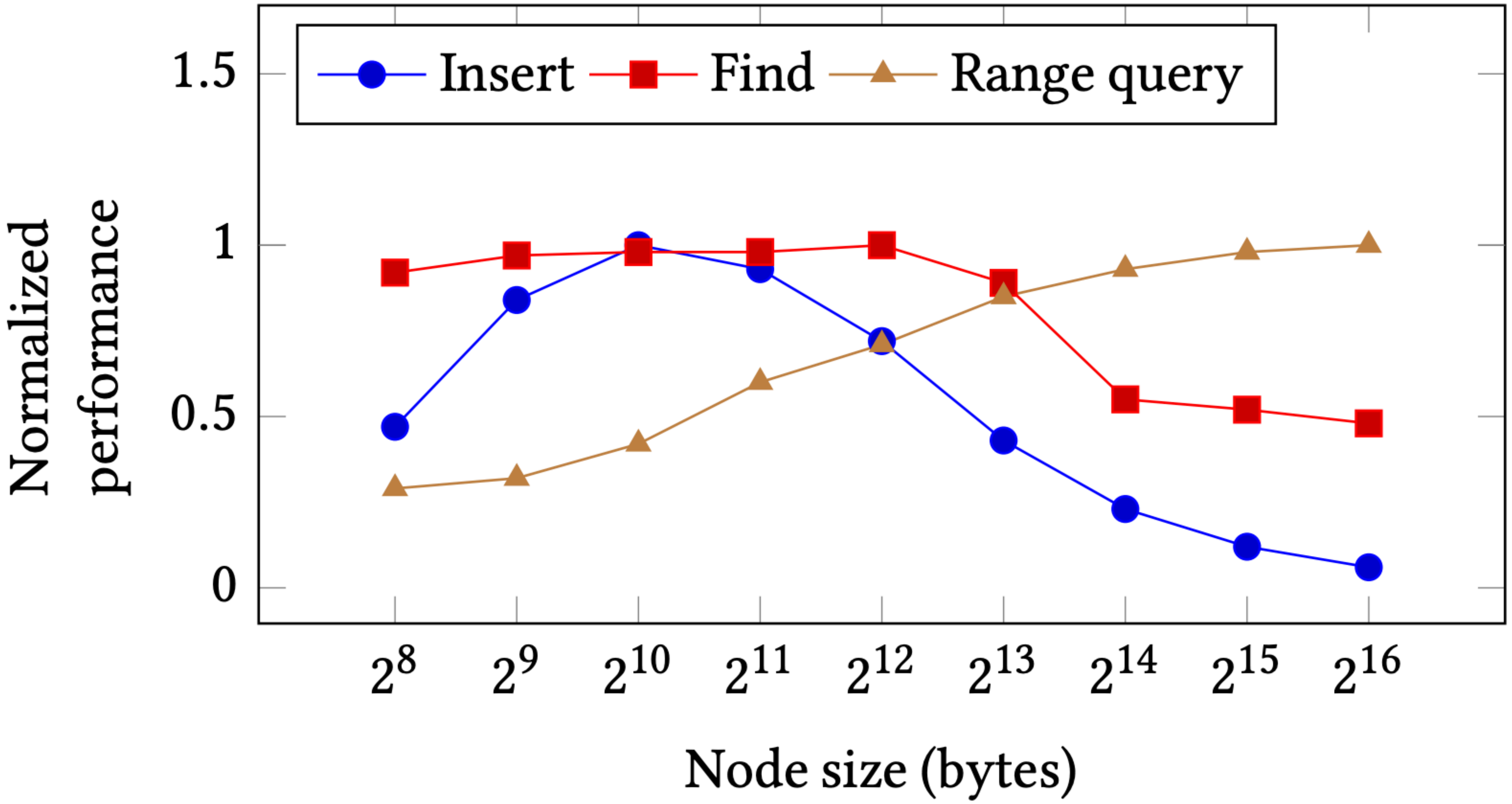
Range scan $O(log_B N/M + K/B)$ I/Os

Dominates for short ranges

Dominates for long ranges



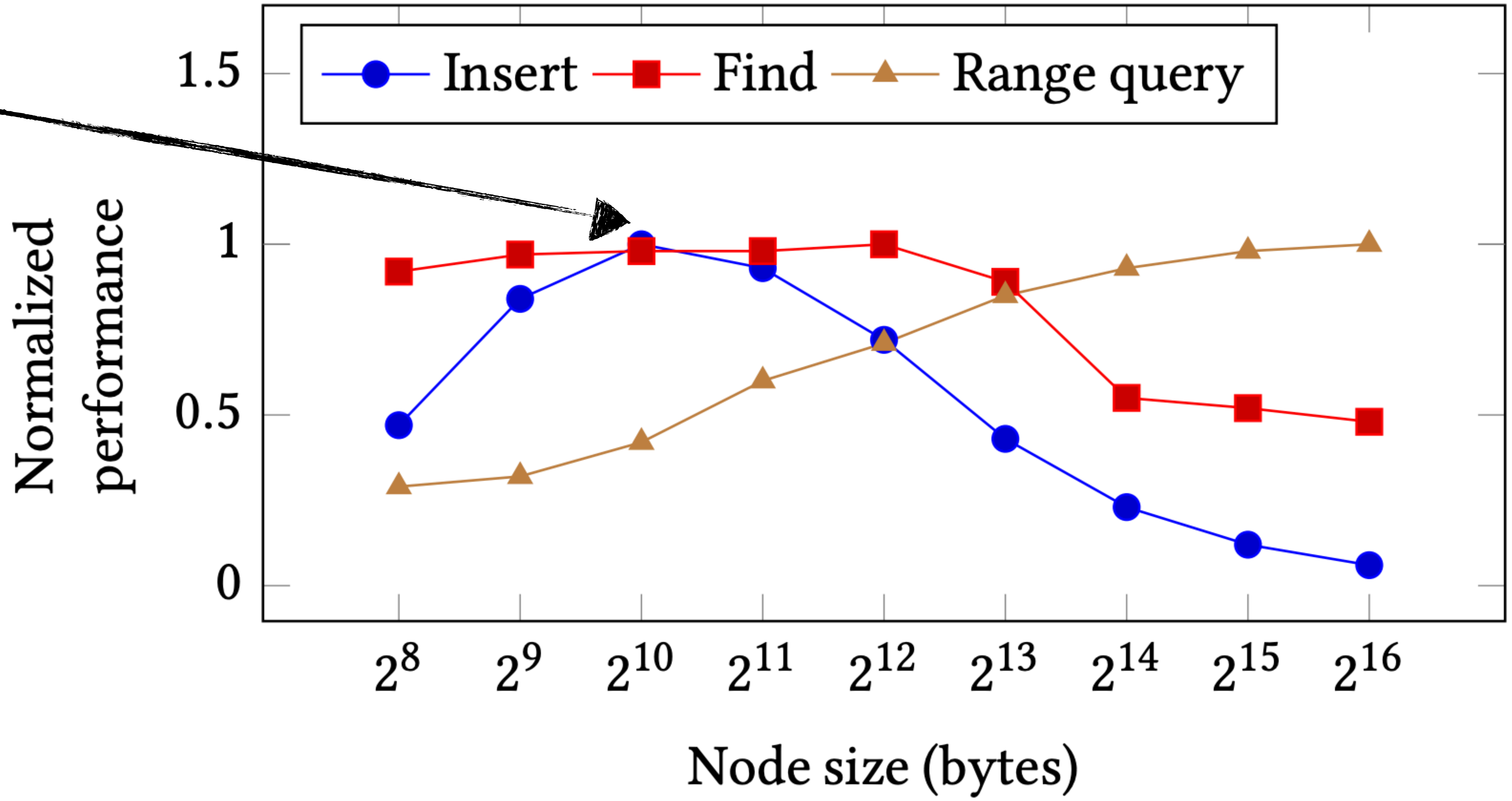
B-trees show a tradeoff in point-range operations



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

B-trees show a tradeoff in point-range operations

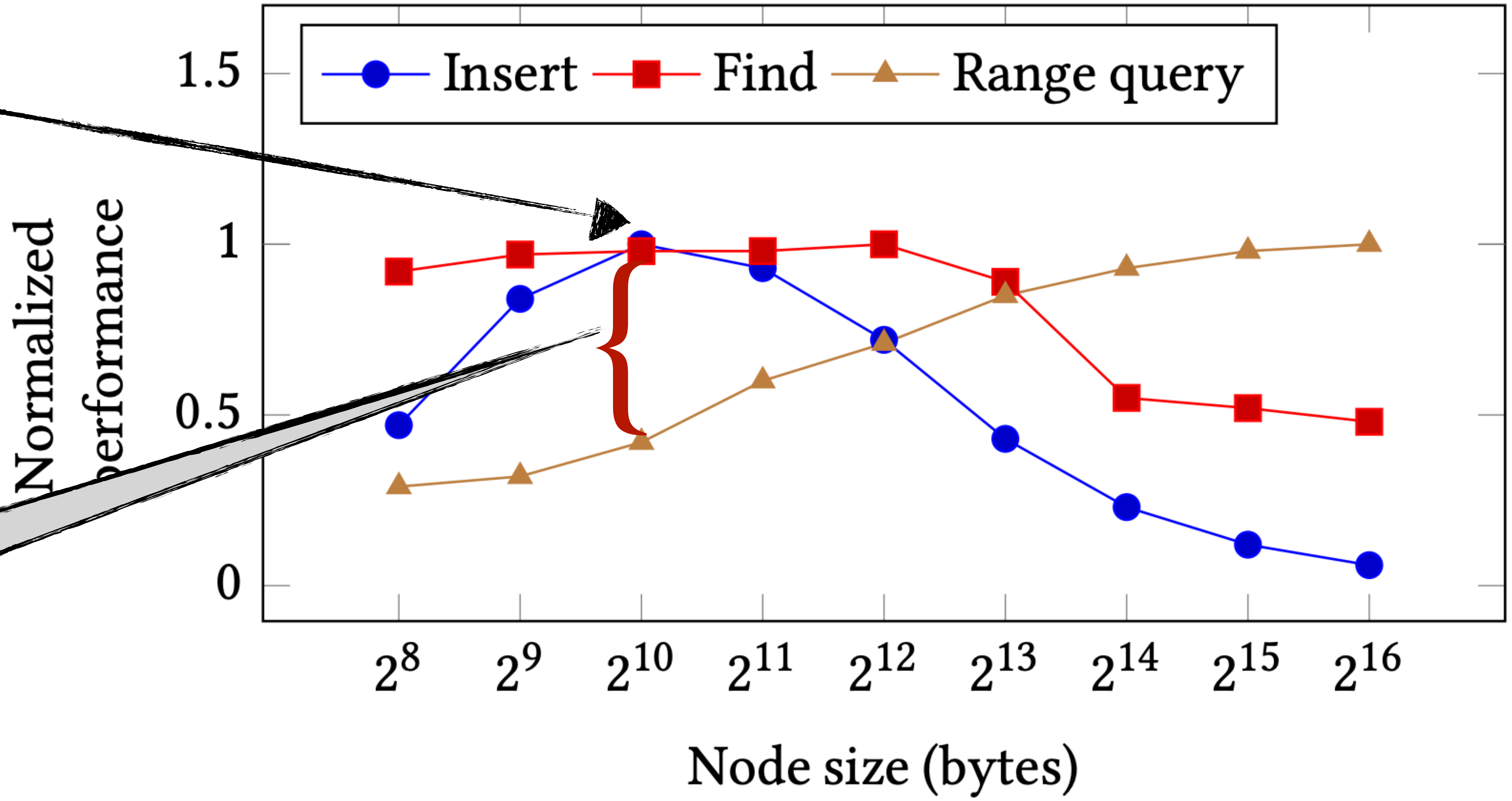
Default node size:
1024 Bytes



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

B-trees show a tradeoff in point-range operations

Default node size:
1024 Bytes

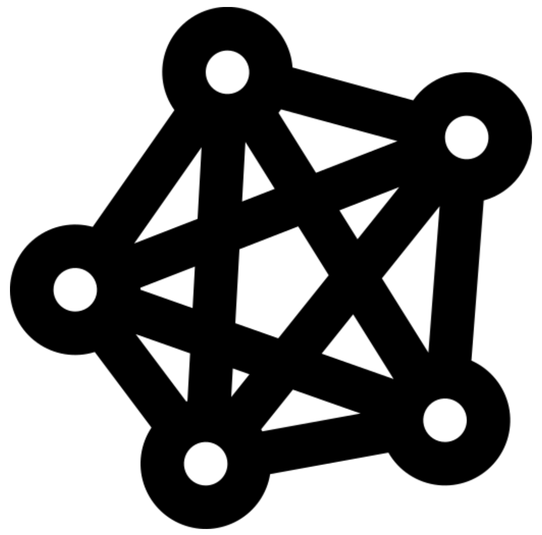


>2X slower range scans

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

Long range scans are critical in applications

Real-time analytics
[PTPH12]



Graph processing
[DBGS22, PWXB21]

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

Our results: BP-tree

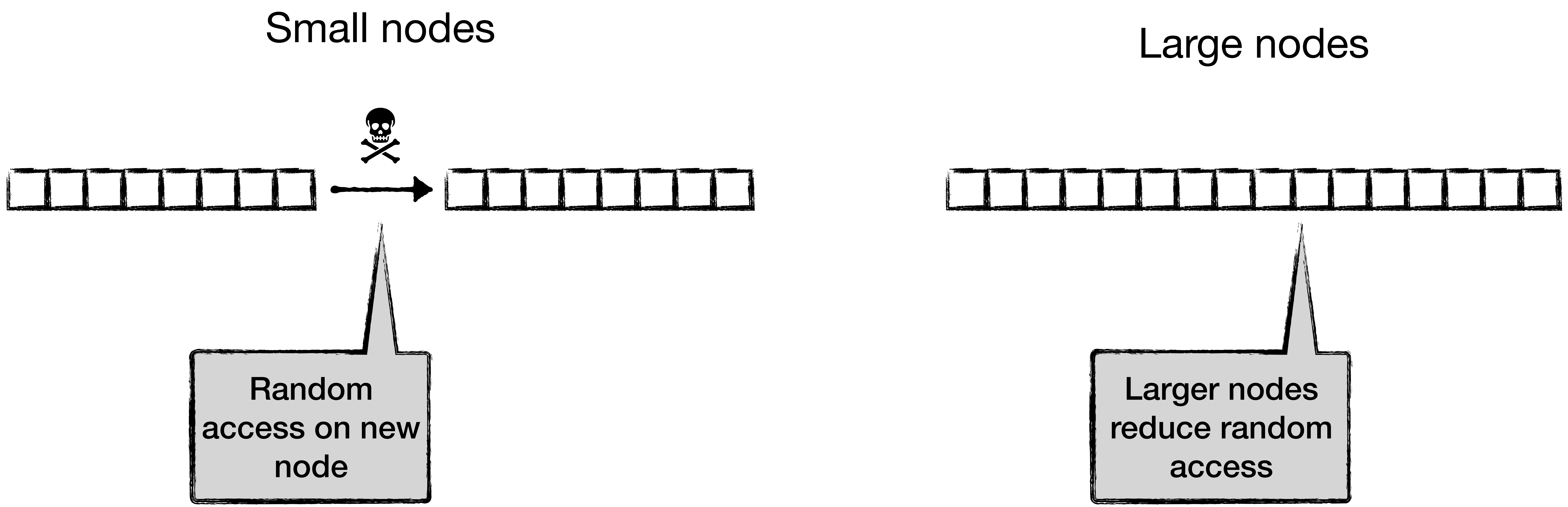
Concurrent C++ implementation

Empirical evaluation using YCSB [CST+10] workloads

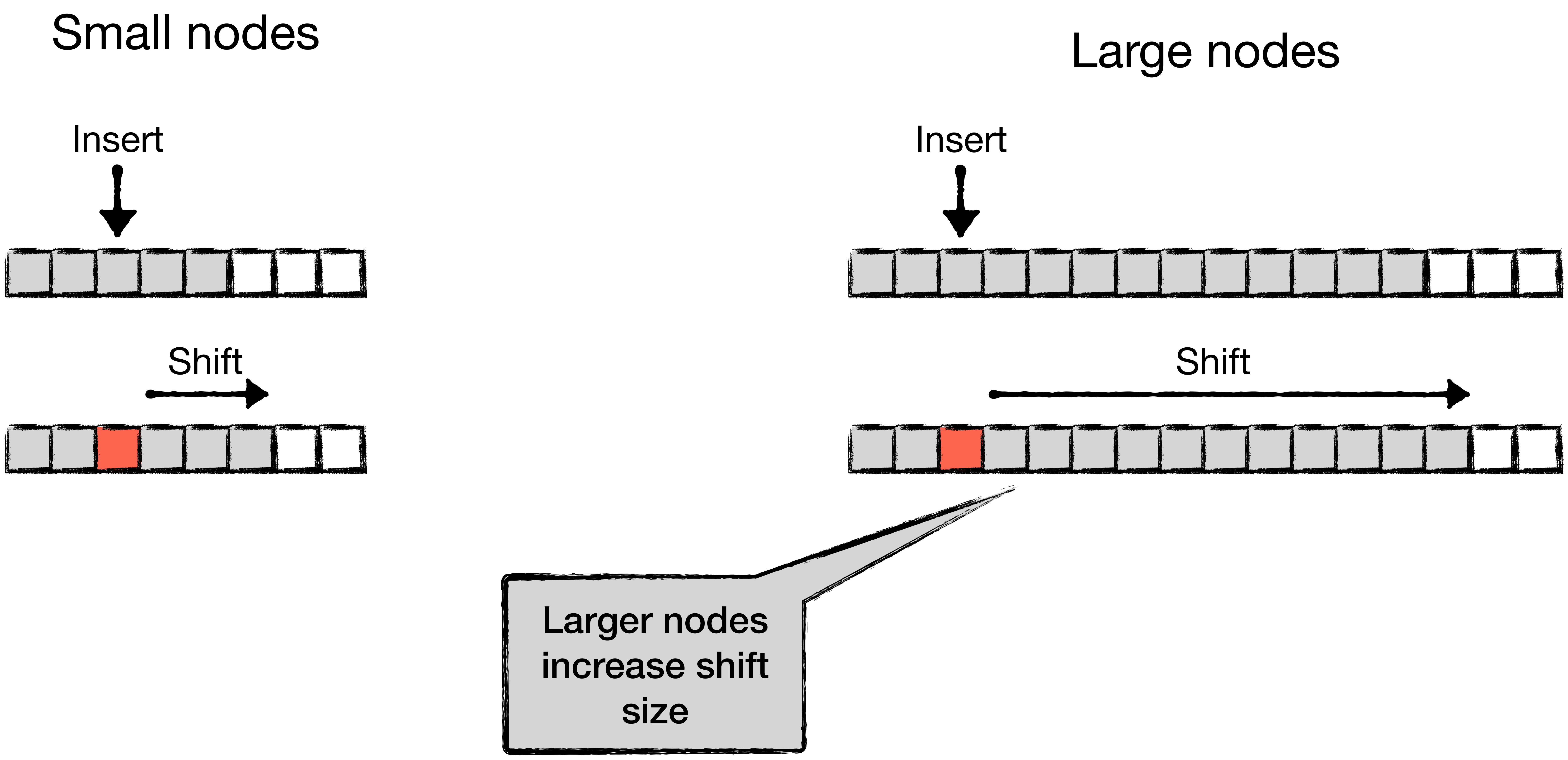
Extended YCSB to include long range scans

	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

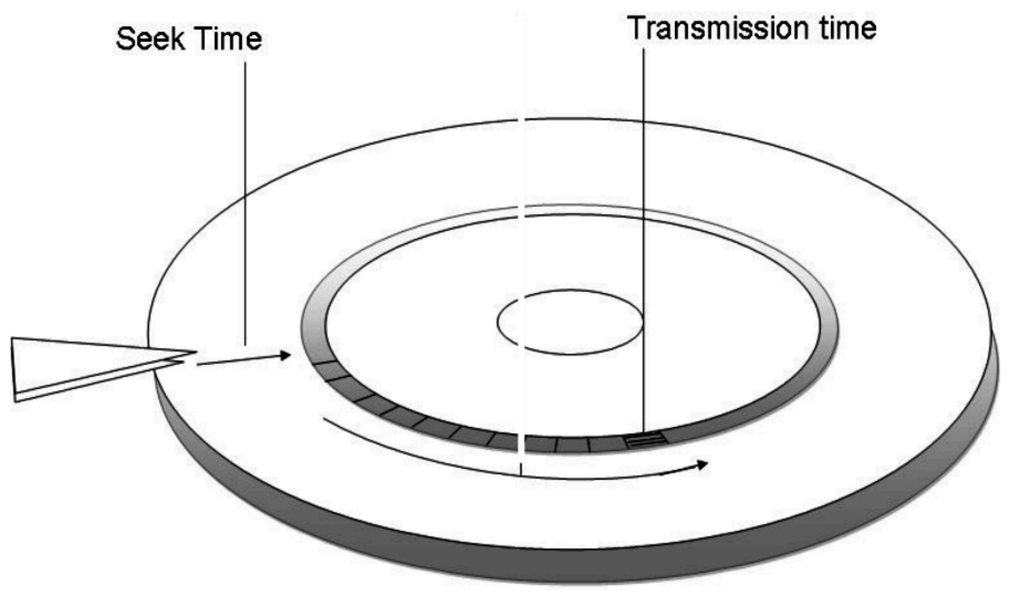
Larger nodes improve range scan performance



Larger nodes cause overhead to maintain order

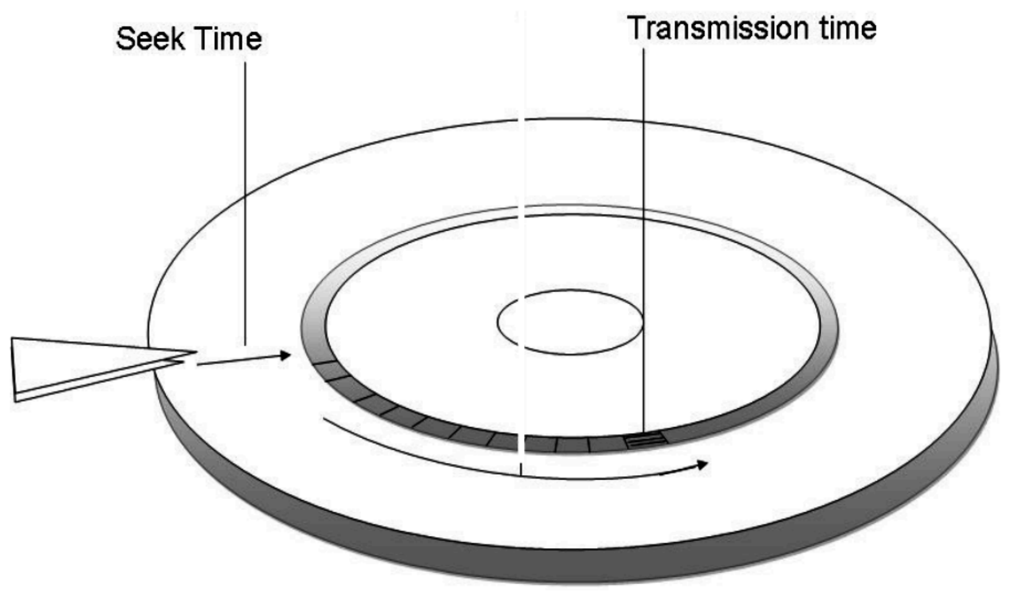


BP-tree design principles



Affine model for performance

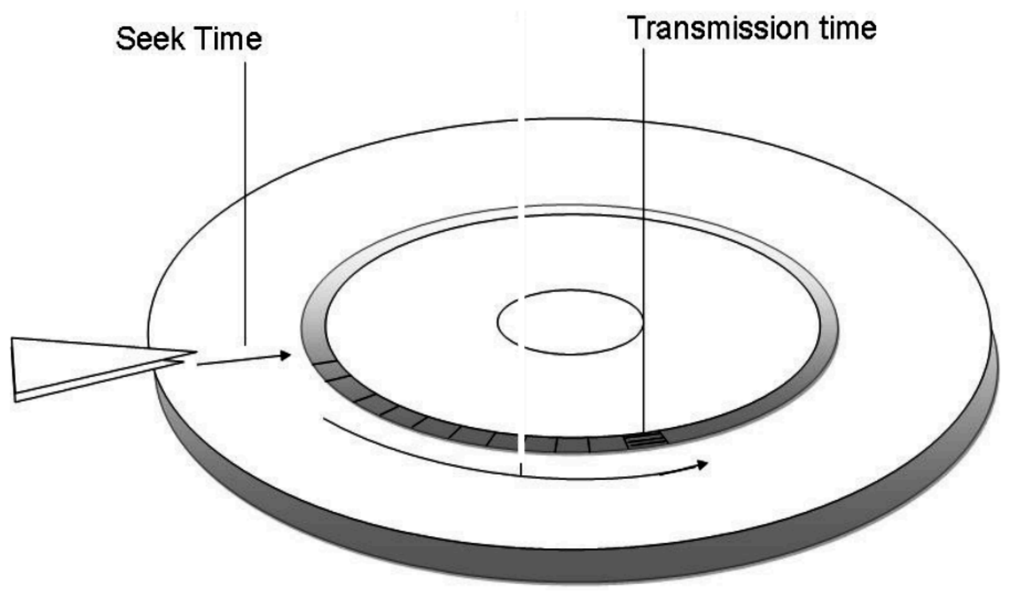
BP-tree design principles



Affine model for performance

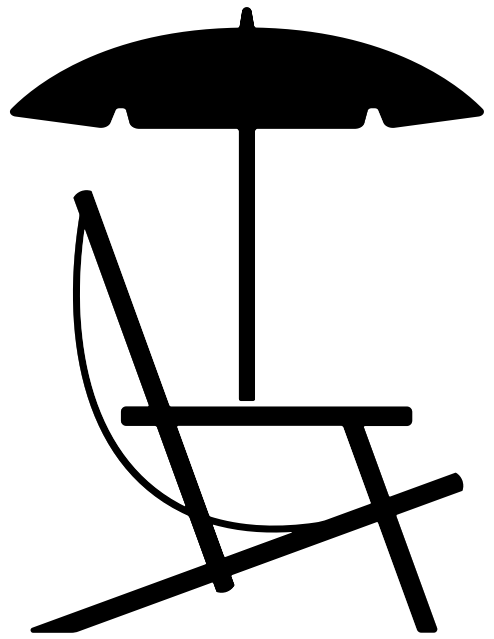
Large leaf nodes

BP-tree design principles



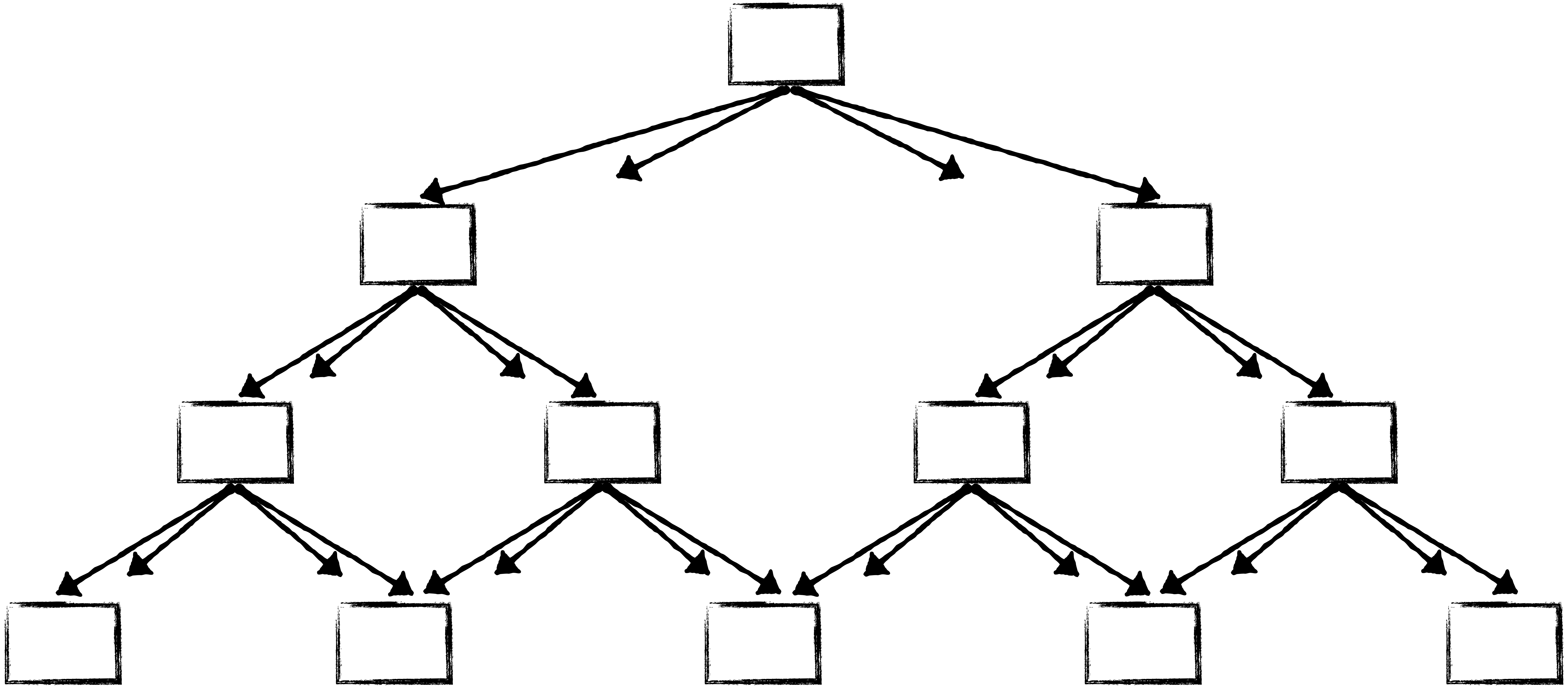
Affine model for performance

Large leaf nodes

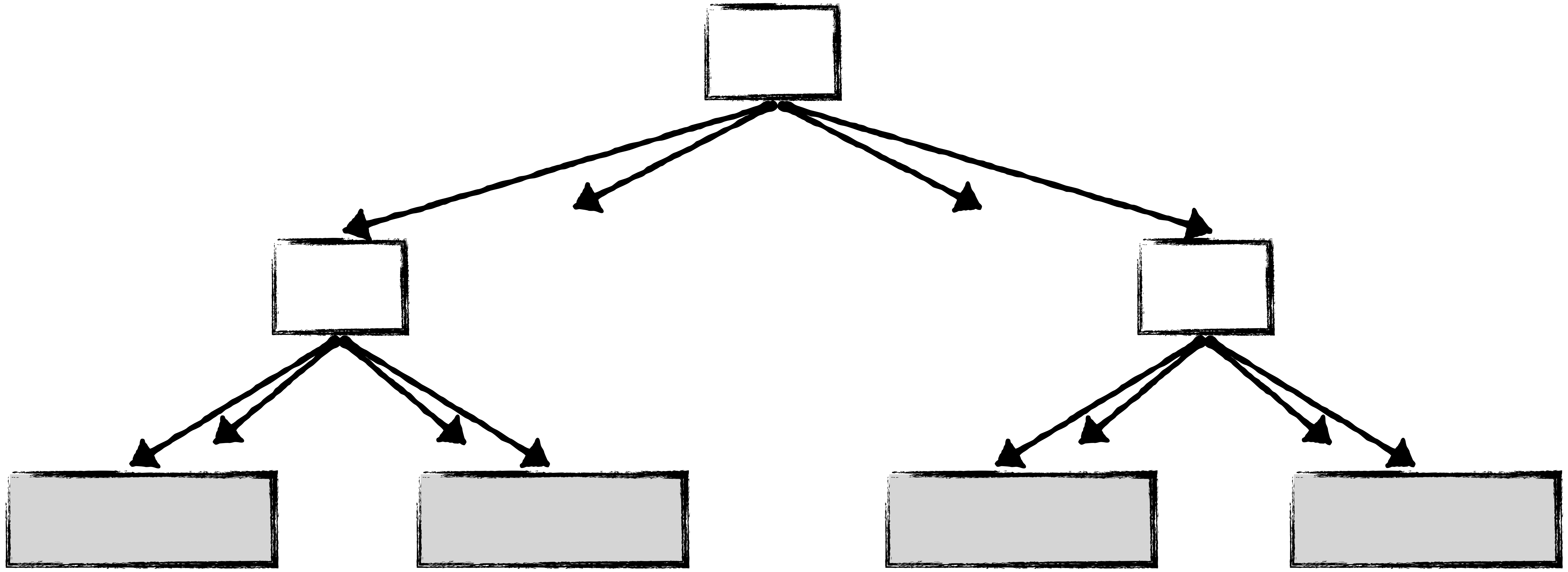


Lazy ordering in leaf nodes

BP-tree design

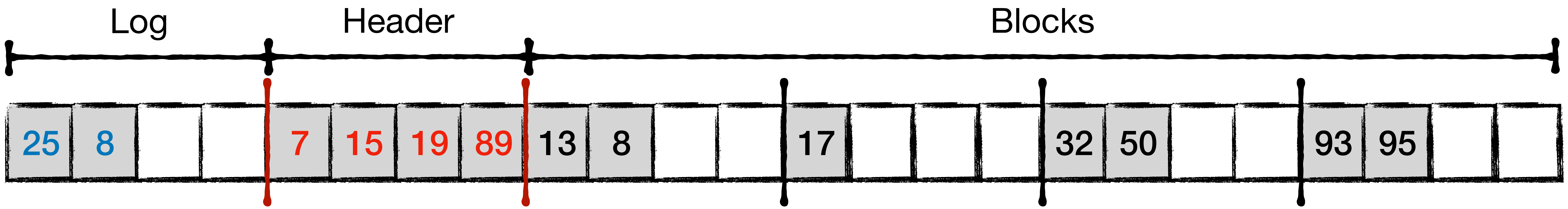


BP-tree design

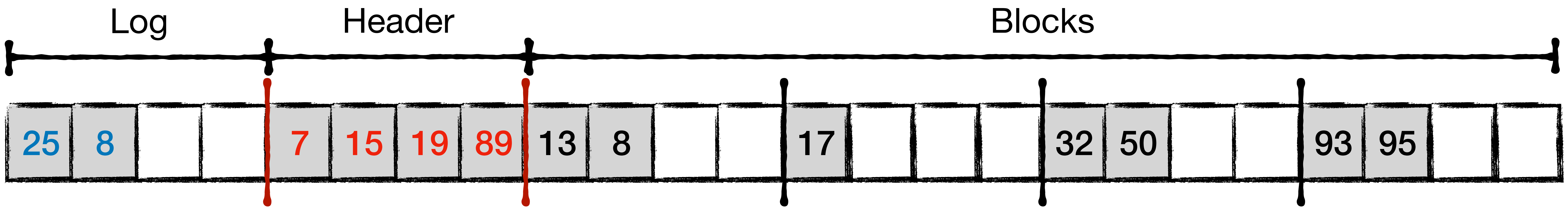


Buffered Partitioned Array:
a special data structure for leaves

Buffered Partitioned Array

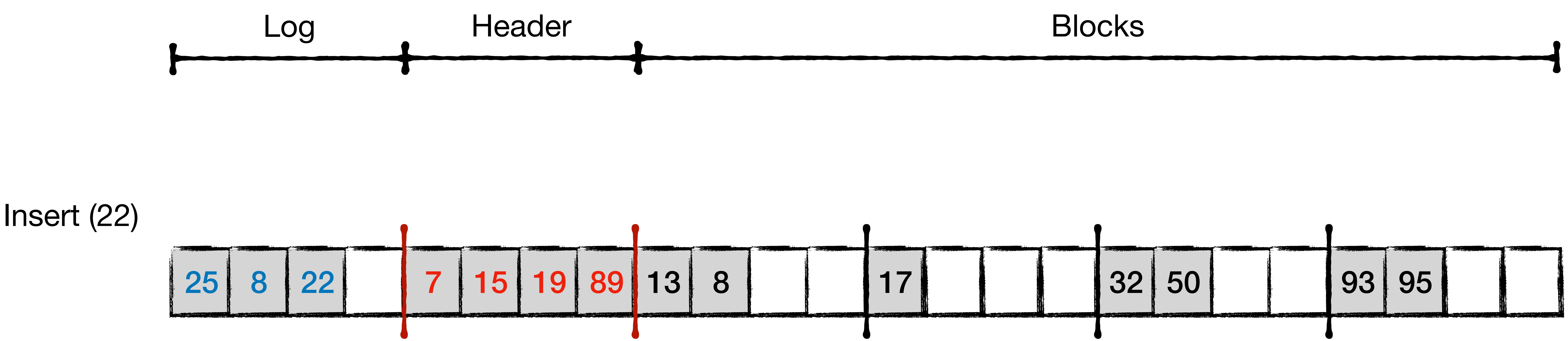


Buffered Partitioned Array

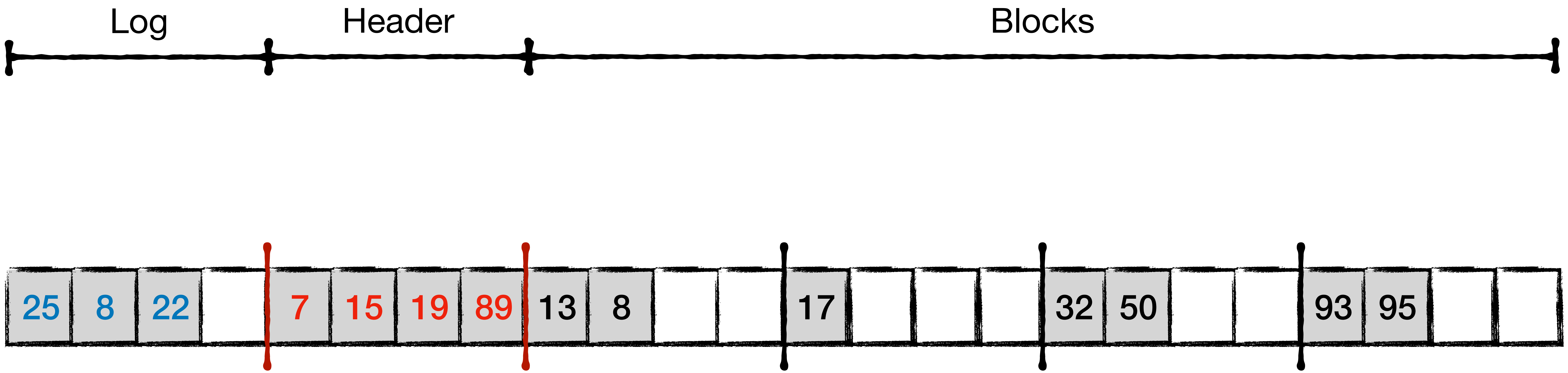


Insert (22)

Buffered Partitioned Array

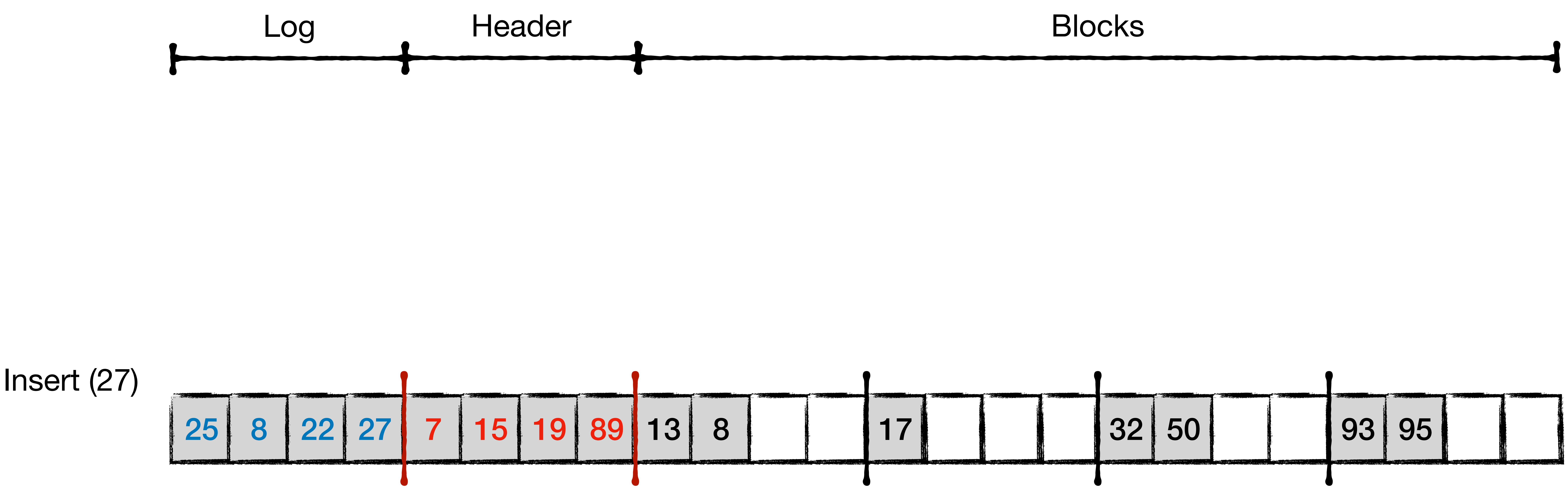


Buffered Partitioned Array

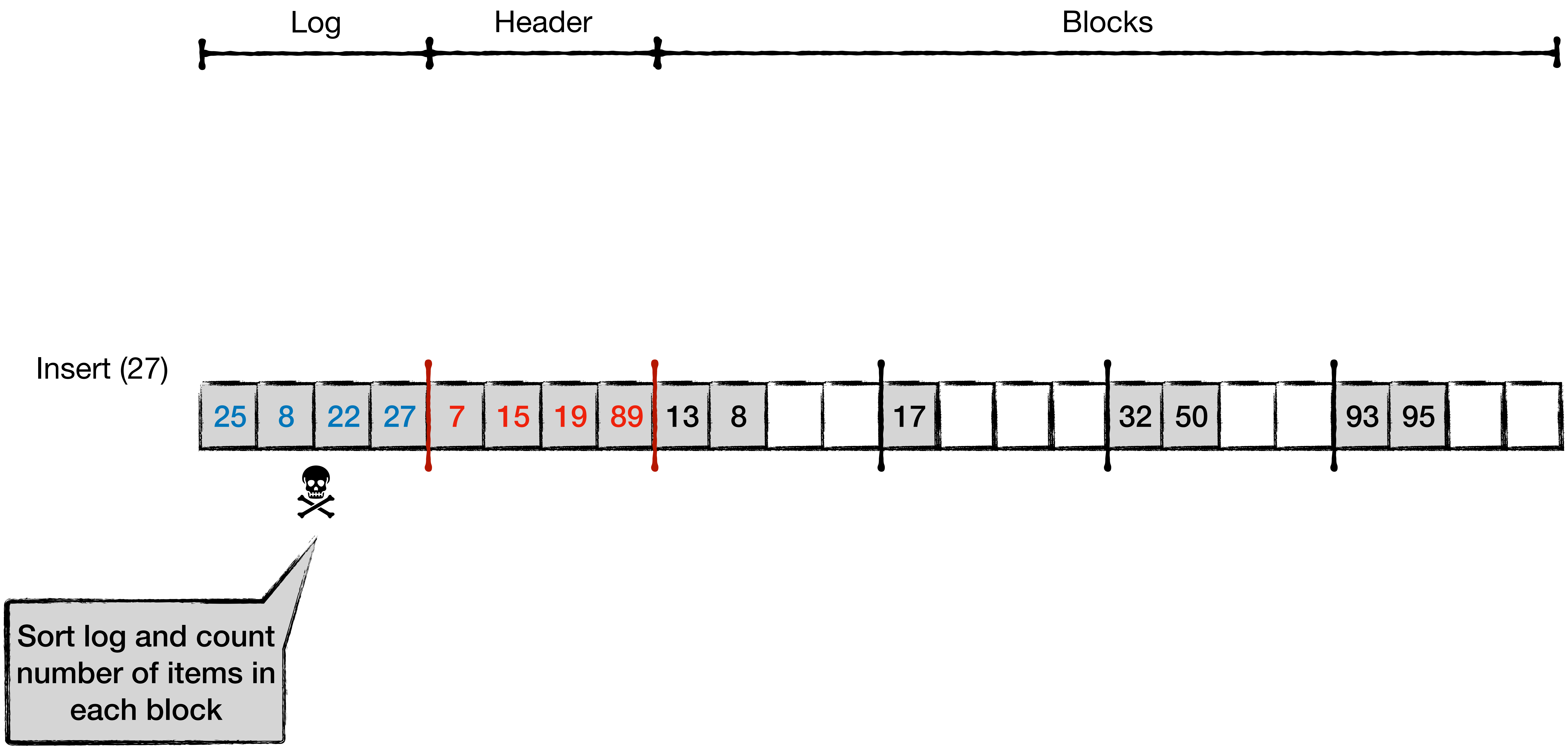


Insert (27)

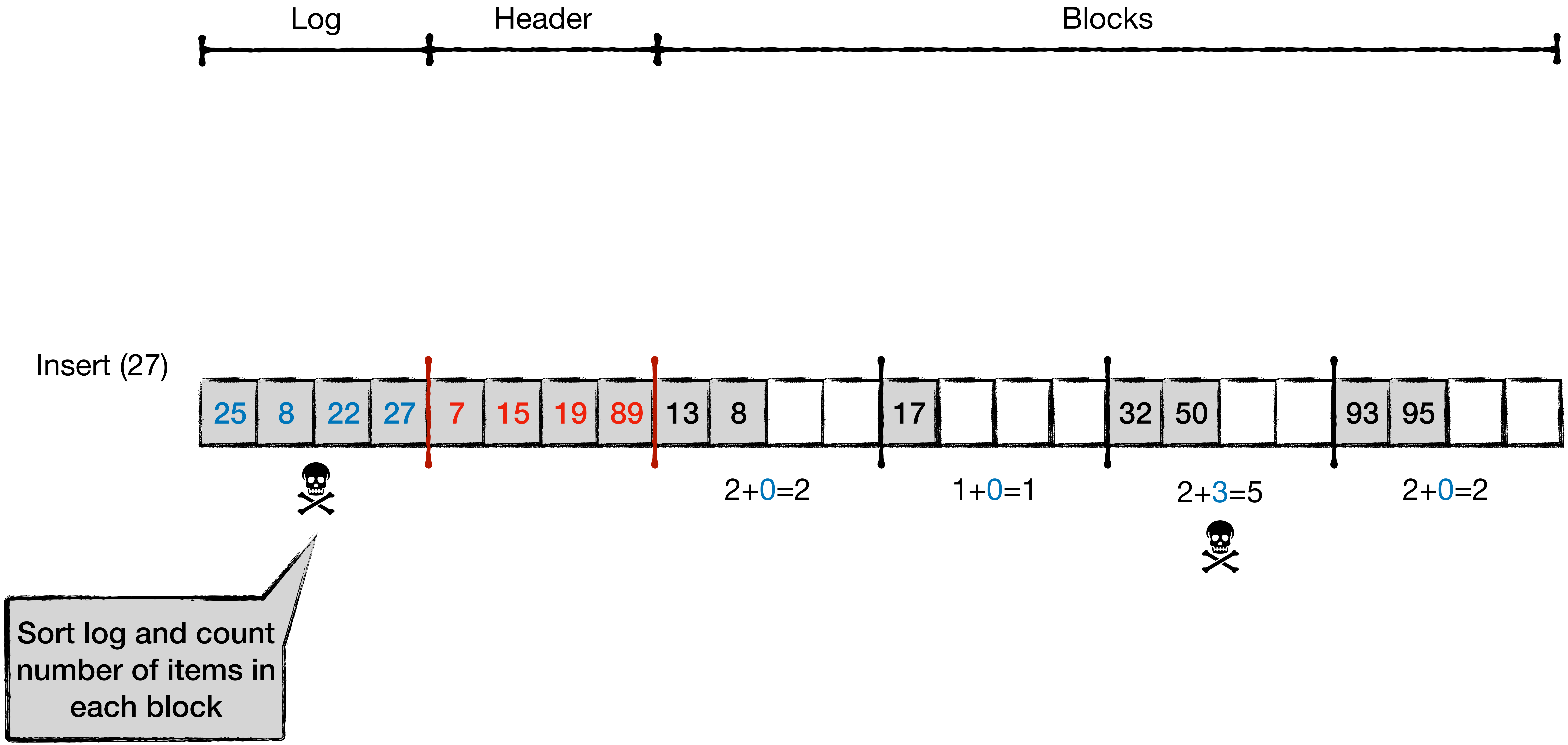
Buffered Partitioned Array



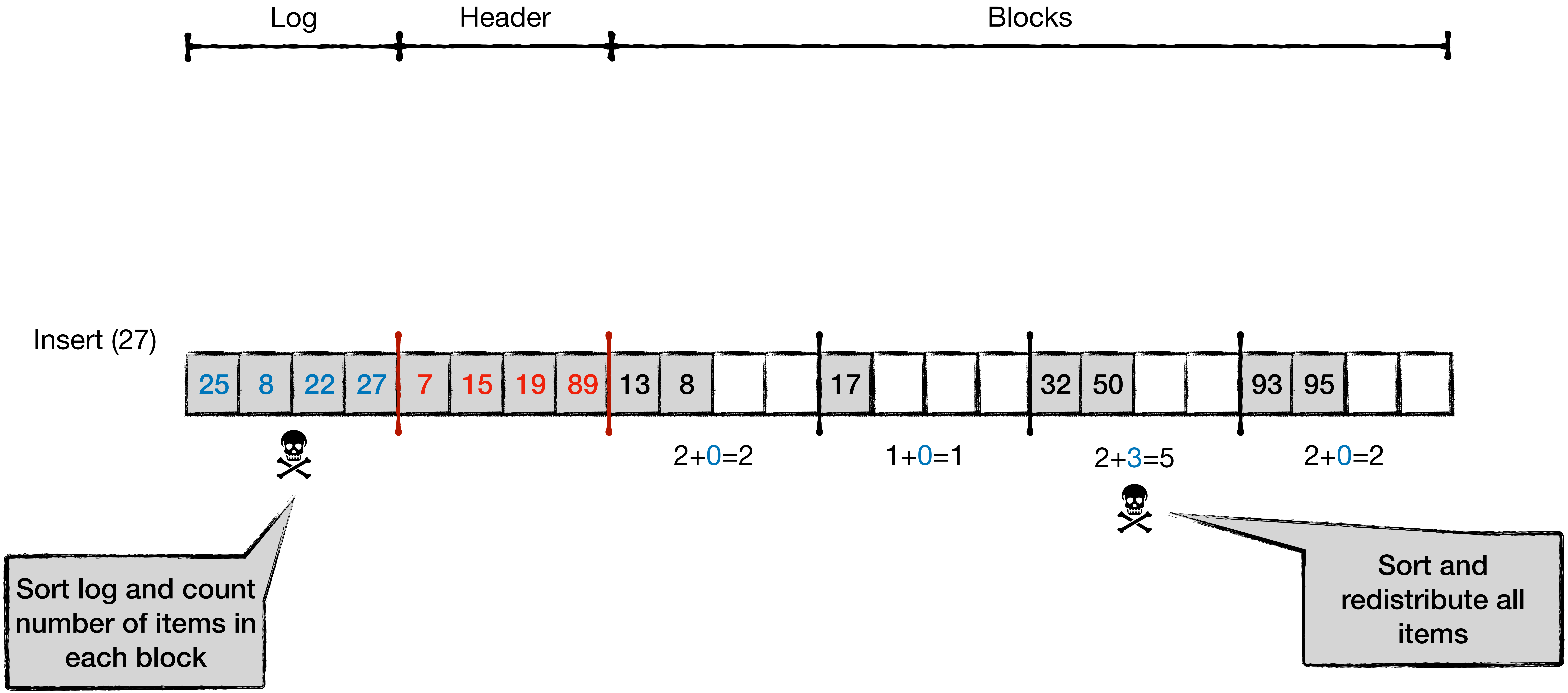
Buffered Partitioned Array



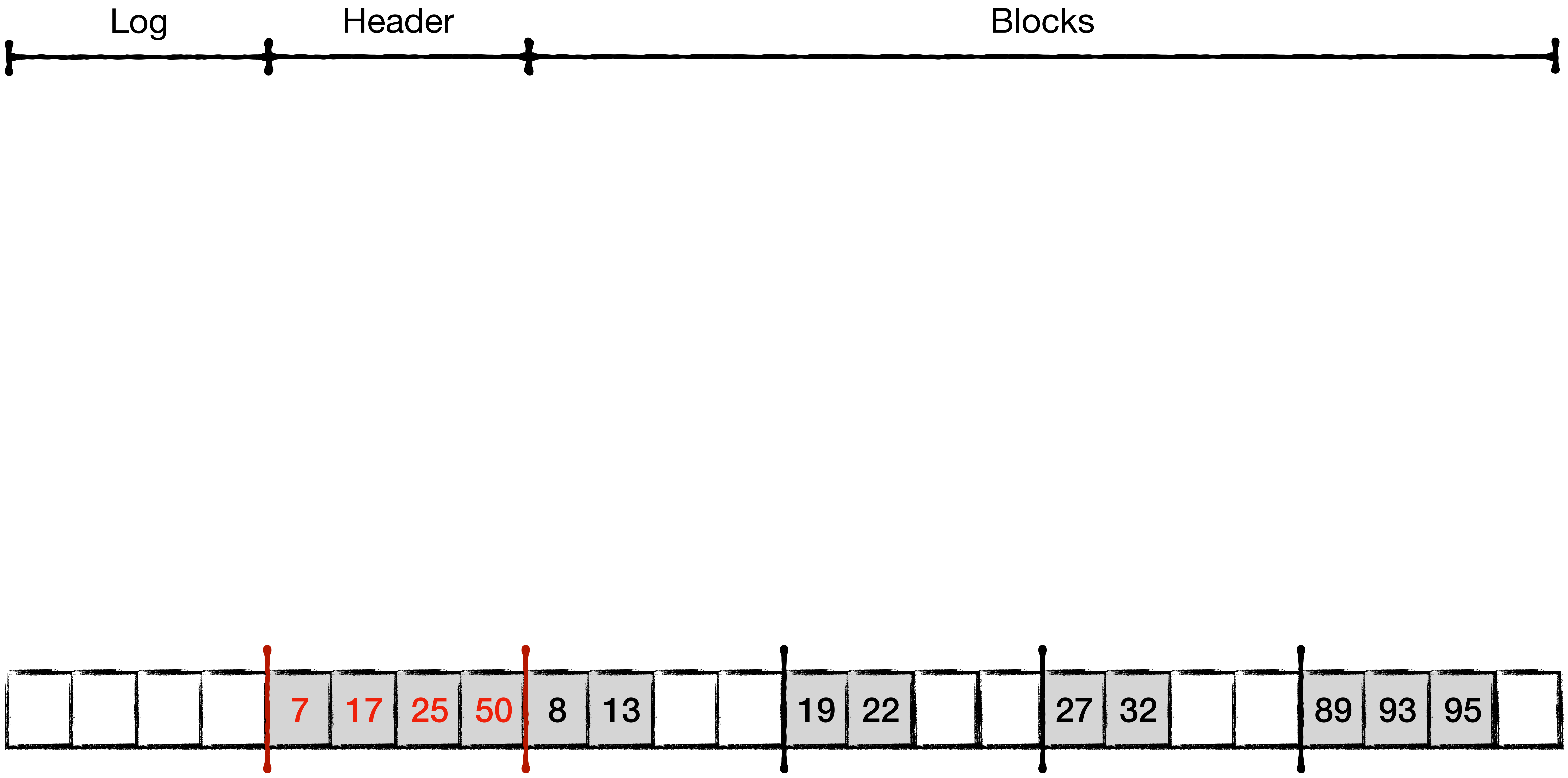
Buffered Partitioned Array



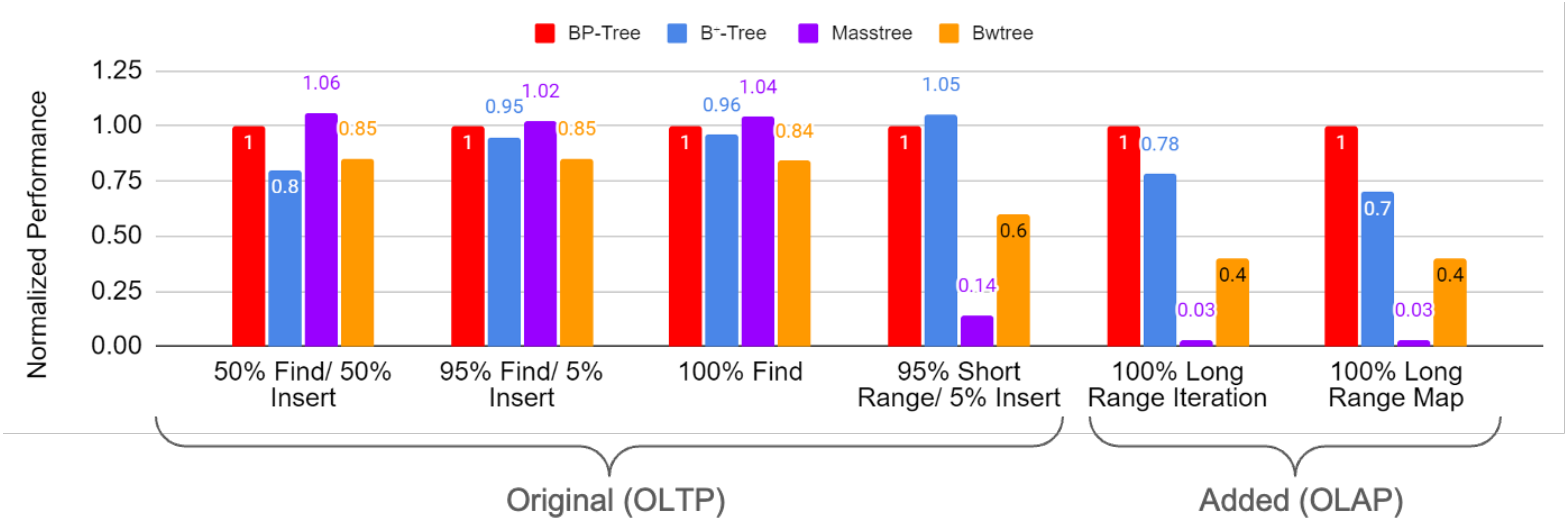
Buffered Partitioned Array



Buffered Partitioned Array



Performance YCSB workloads



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

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)

Code: https://github.com/wheatman/concurrent-btrees

- 48-core 2-way hyperthreaded Intel® Xeon® Platinum 8275CL CPU @ 3.00GHz
- Cache
  - 1.5MiB of L1 cache,
  - 48 MiB of L2 cache,
  - 71.5 MiB of L3 cache across all of the cores
- 189 GB of memory
- all experiments on a single socket with 24 physical cores and 48 hyperthreads
- All times are the median of 5 trials after one warm-up trial

Table 1: Throughput (thr., in operations per second) and normalized performance of point operations in the B-tree and BP-tree. Point operation throughput is reported in operations/s. We use N.P. to denote the normalized performance in the B-tree (BP-tree) compared to the best B-tree (BP-tree) configuration for that operation (1.0 is the best possible value).

B-tree					*BP-tree*						
	Insert		*Find*					*Insert*		*Find*	
Node size (bytes)	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Header size (elts)*	*Block size (elts)*	*Total size (bytes)*	*Thr.*	*N.P.*	*Thr.*	*N.P.*
256	8.72E6	0.47	2.66E7	0.92	4	4	384	1.05E7	0.54	2.96E7	0.94
512	1.56E7	0.84	2.81E7	0.97	4	8	640	1.42E7	0.73	2.96E7	0.94
1024	1.86E7	1	2.86E7	0.98	8	8	1280	1.63E7	0.84	3.05E7	0.96
2048	1.74E7	0.93	2.84E7	0.98	8	16	2304	1.83E7	0.94	3.09E7	0.98
4096	1.34E7	0.72	2.91E7	1	16	16	4608	1.87E7	0.97	3.16E7	1.00
8192	8.04E6	0.43	2.60E7	0.89	16	32	8704	1.87E7	0.97	3.12E7	0.99
16384	4.27E6	0.23	1.59E7	0.55	32	32	17408	1.94E7	1.00	3.02E7	0.96
32768	2.20E6	0.12	1.50E7	0.52	32	64	33792	1.84E7	0.95	2.97E7	0.94
65536	1.12E6	0.06	1.40E7	0.48	64	64	67584	1.73E7	0.89	1.73E7	0.55

Table 2: Throughput (thr., in expected elements per second) of range queries of varying maximum lengths (`max_len`) in the B-tree and BP-tree. We also report the normalized performance (N.P.) compared to the best-case performance for each operation (up to 1.0).

B-tree									*BP-tree*										
	Short (`max_len` = 100)				*Long* (`max_len` = 100,000)							*Short* (`max_len` = 100)				*Long* (`max_len` = 100,000)			
	Map		*Iterate*		*Map*		*Iterate*					*Map*		*Iterate*		*Map*		*Iterate*	
Node size (bytes)	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Header size (elts)*	*Block size (elts)*	*Total size (bytes)*	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Thr.*	*N.P.*	*Thr.*	*N.P.*
256	8.56E8	0.77	9.48E8	0.72	1.88E9	0.25	2.16E9	0.29	4	4	384	4.76E8	0.53	7.15E8	0.59	7.32E8	0.14	1.20E9	0.22
512	9.58E8	0.86	1.05E9	0.80	2.12E9	0.28	2.43E9	0.32	4	8	640	6.86E8	0.76	8.93E8	0.73	1.32E9	0.25	1.71E9	0.32
1024	1.01E9	0.91	1.13E9	0.85	2.69E9	0.36	3.13E9	0.42	8	8	1280	7.91E8	0.88	9.45E8	0.78	1.72E9	0.32	1.85E9	0.35
2048	1.08E9	0.97	1.20E9	0.91	4.23E9	0.56	4.51E9	0.60	8	16	2304	8.98E8	1.00	1.07E9	0.88	2.46E9	0.46	2.54E9	0.47
4096	1.11E9	1.00	1.26E9	0.95	5.18E9	0.69	5.33E9	0.71	16	16	4608	8.99E8	1.00	1.13E9	0.93	3.17E9	0.59	3.22E9	0.60
8192	1.10E9	0.99	1.28E9	0.97	5.97E9	0.80	6.36E9	0.85	16	32	8704	8.86E8	0.99	1.22E9	1.00	4.19E9	0.78	4.25E9	0.79
16384	1.08E9	0.98	1.29E9	0.98	6.60E9	0.88	7.00E9	0.93	32	32	17408	8.14E8	0.91	1.17E9	0.96	4.75E9	0.89	4.75E9	0.89
32768	1.08E9	0.97	1.30E9	0.98	7.18E9	0.96	7.36E9	0.98	32	64	33792	6.73E8	0.75	1.05E9	0.87	5.21E9	0.97	5.16E9	0.96
65536	1.09E9	0.98	1.32E9	1.00	7.50E9	1.00	7.49E9	1.00	64	64	67584	5.74E8	0.64	9.83E8	0.81	5.35E9	1.00	5.35E9	1.00

Table 3: Throughput (in operations/s) of the BP-tree (BPT), B-tree (B^+T), Masstree (MT), and OpenBw-tree (BWT) on uniform random and zipfian workloads from YCSB.

		Uniform							*Zipfian*						
Workload	Description	*BPT*	*B^+T*	*B^+T/ BPT*	*MT*	*MT/ BPT*	*BWT*	*BWT/ BPT*	*BPT*	*B^+T*	*B^+T/ BPT*	*MT*	*MT/ BPT*	*BWT*	*BWT/ BPT*
A	50% finds, 50% inserts	2.91E7	2.33E7	0.80	3.07E7	1.06	2.47E7	0.85	3.00E7	2.78E7	0.93	3.20E7	1.07	2.56E7	0.85
B	95% finds, 5% inserts	4.70E7	4.46E7	0.95	4.79E7	1.02	3.98E7	0.85	5.63E7	4.84E7	0.86	5.82E7	1.03	4.74E7	0.84
C	100% finds	4.99E7	4.81E7	0.96	5.18E7	1.04	4.21E7	0.84	6.01E7	5.99E7	1.00	6.40E7	1.06	5.10E7	0.85
E	95% short range iterations (`max_len` = 100), 5% inserts	2.58E7	2.71E7	1.05	3.49E6	0.14	1.54E7	0.60	3.25E7	3.35E7	1.03	3.96E6	0.12	1.70E7	0.52
X	100% long range iterations (`max_len` = 10,000)	8.89E5	6.90E5	0.78	2.74E4	0.03	3.60E5	0.40	1.05E6	7.96E5	0.76	2.76E4	0.03	3.65E5	0.35
Y	100% long range maps (`max_len` = 10,000)	9.18E5	6.45E5	0.70	2.74E4	0.03	3.63E5	0.40	1.08E6	7.44E5	0.69	2.76E4	0.03	3.71E5	0.34