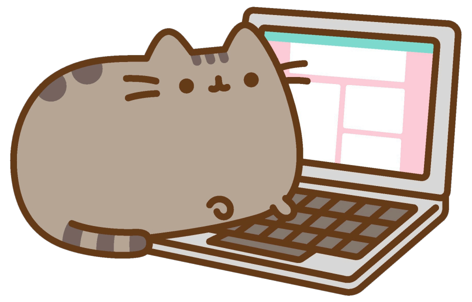
CSE 6230:
HPC Tools and Applications

+



Lecture 21: BP-tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees

Helen Xu

hxu615@gatech.edu

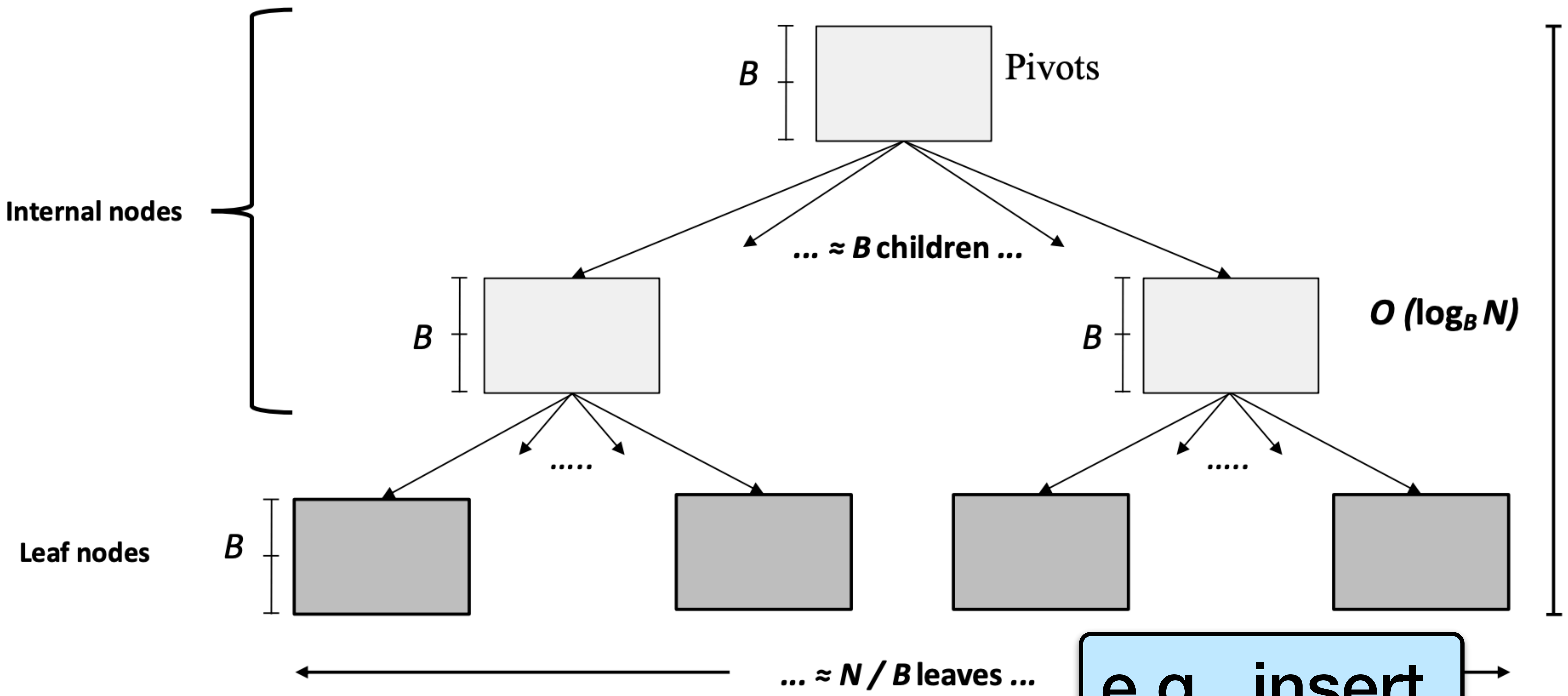
Georgia Tech College of Computing
School of Computational Science and Engineering

from Xu, Li, Wheatman, Marneni, Pandey - VLDB 23

Recall: B-trees are classical indexing structures

B/B+-trees are used everywhere

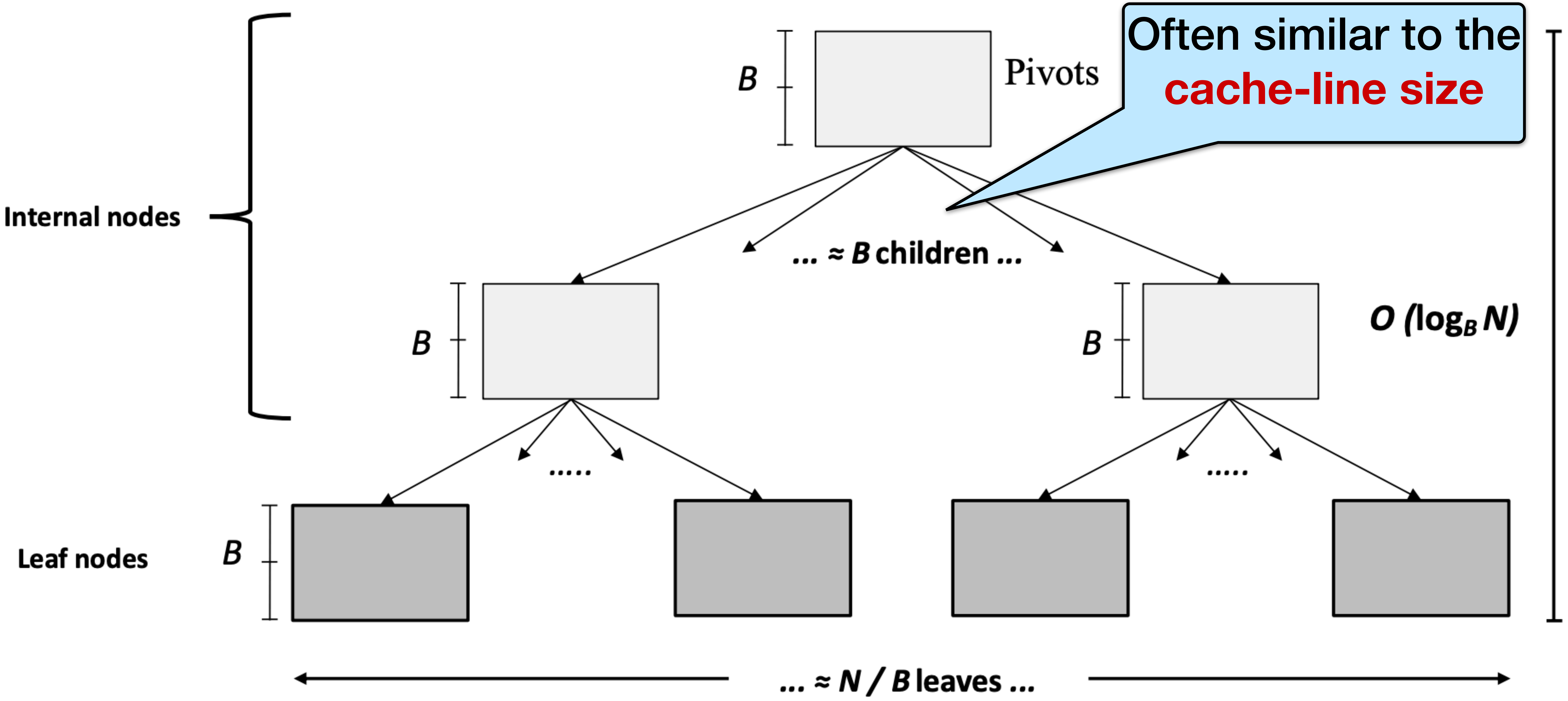
- In-memory indexing
- Databases
- Filesystems



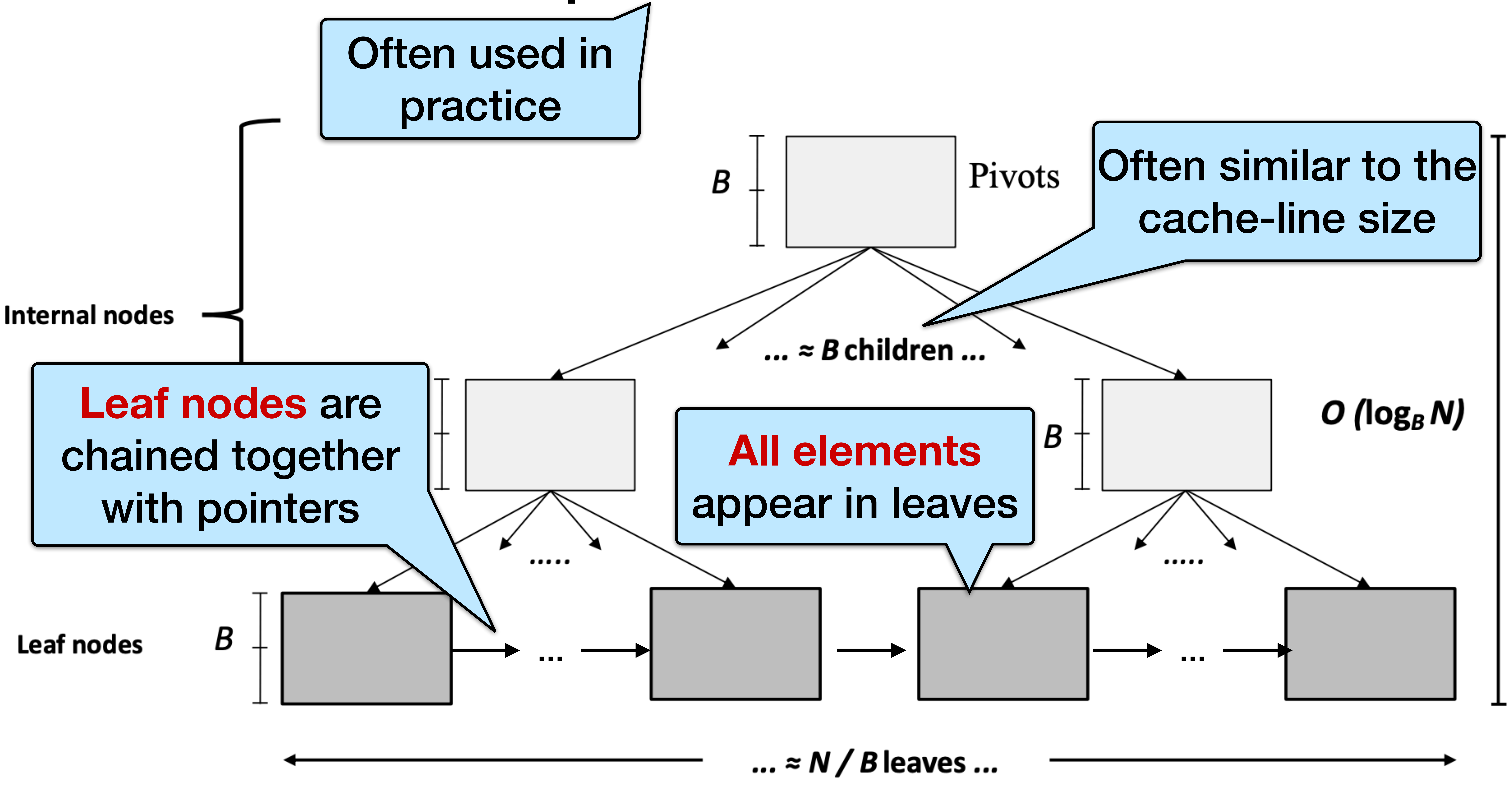
e.g., insert, find

B-trees are asymptotically optimal for point operations

Recap: B-tree structure



Recap: B+-tree structure



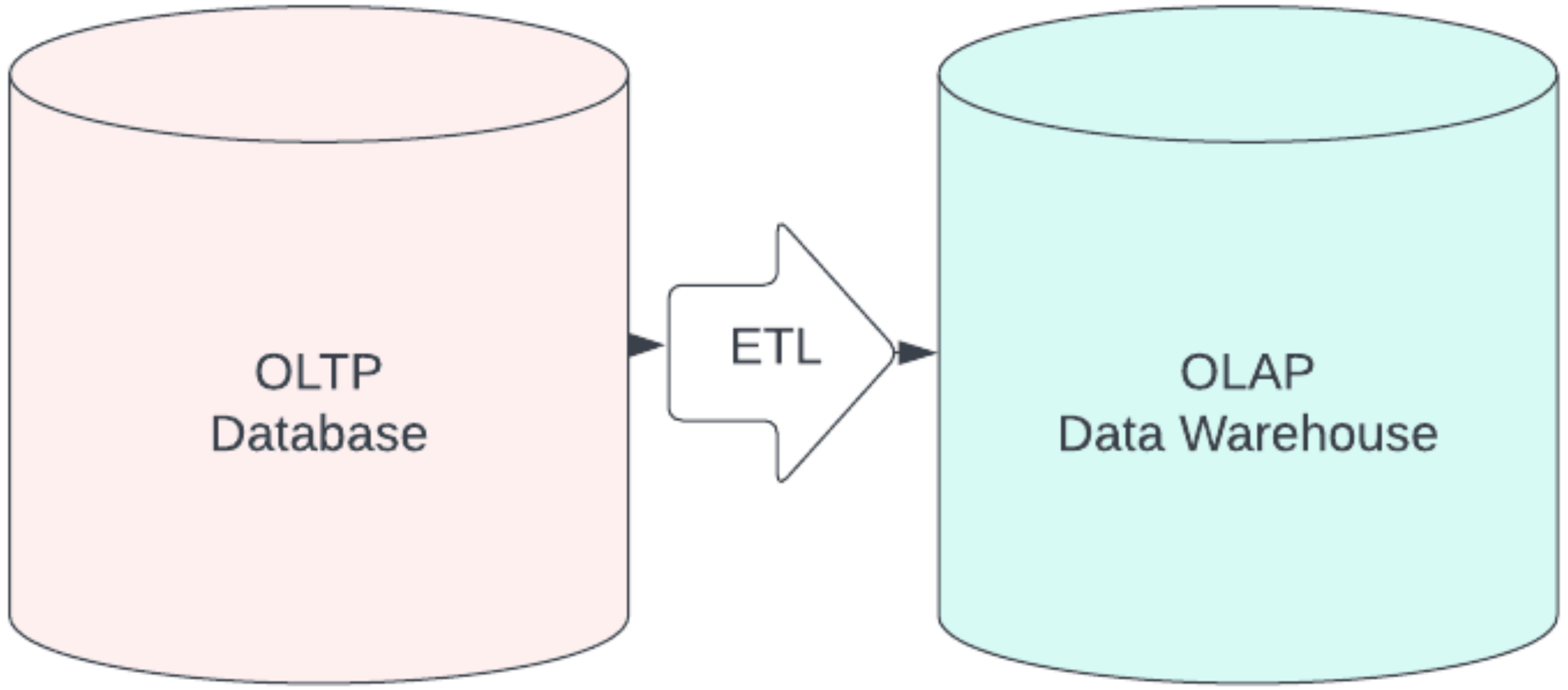
source

OLAP vs OLTP Workloads

- **Online analytical processing** (OLAP) and **online transaction processing** (OLTP) are two different use cases for data-processing systems.

- OLAP is optimized for **complex data analysis and reporting**, while OLTP is optimized for transactional processing and **real-time updates**.

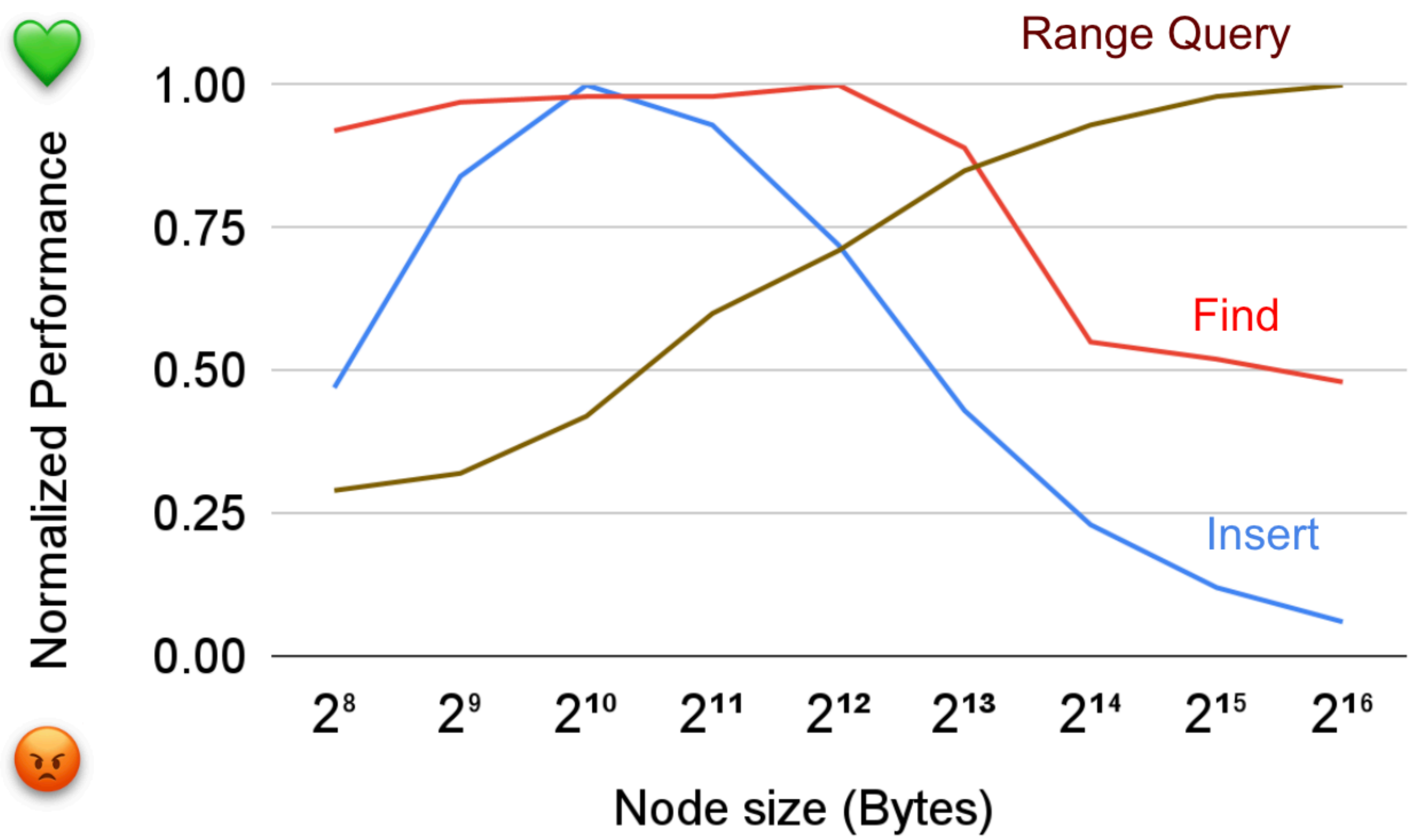
- Traditionally, systems are **optimized for one or the other**, but recently there has been exploration into **combining both functionalities** into one system.



https://dev.to/alexmercedcoder/introduction-to-the-world-of-data-oltp-olap-data-warehouses-data-lakes-and-more-2me7

Problem: B-tree insert-range tradeoff

- B-trees exhibit a **tradeoff** between point inserts (OLTP) and long range queries (OLAP) **as a function of node size**.
- Long range queries are critical for real-time analytics [PTPH12] and graph processing [DBGS22, PGK21, PWXB21].



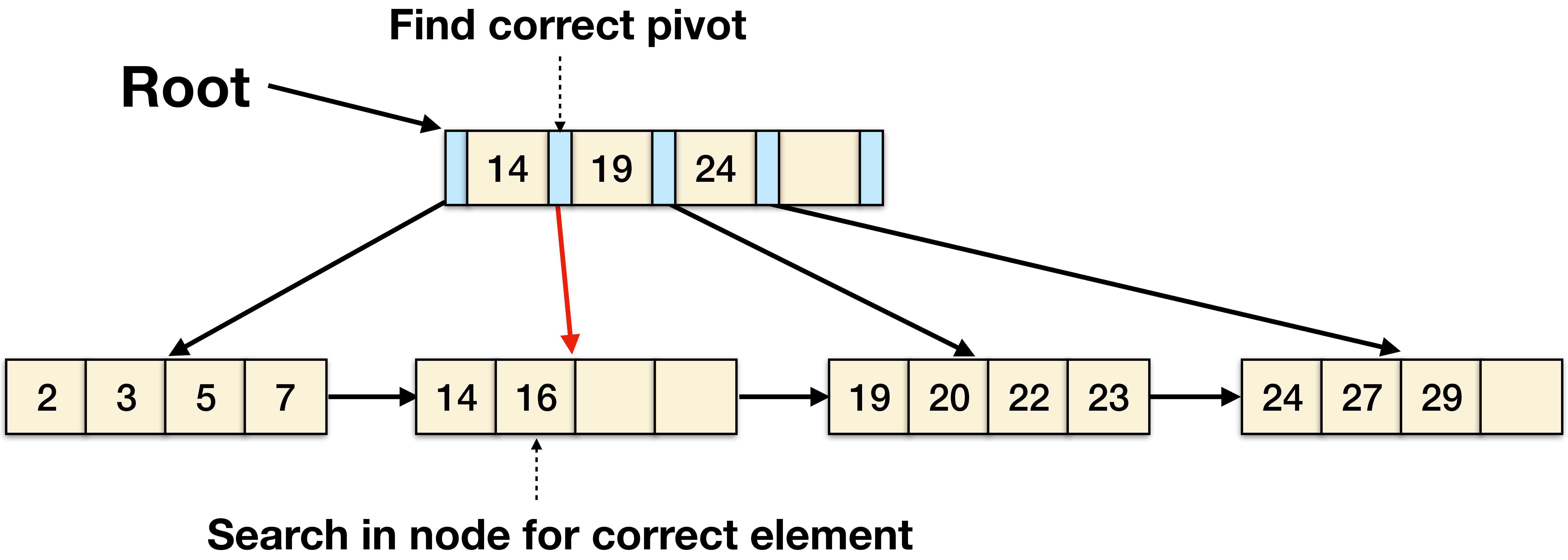
Large nodes speed up range scans at the cost of point inserts

YCSB Point Operations

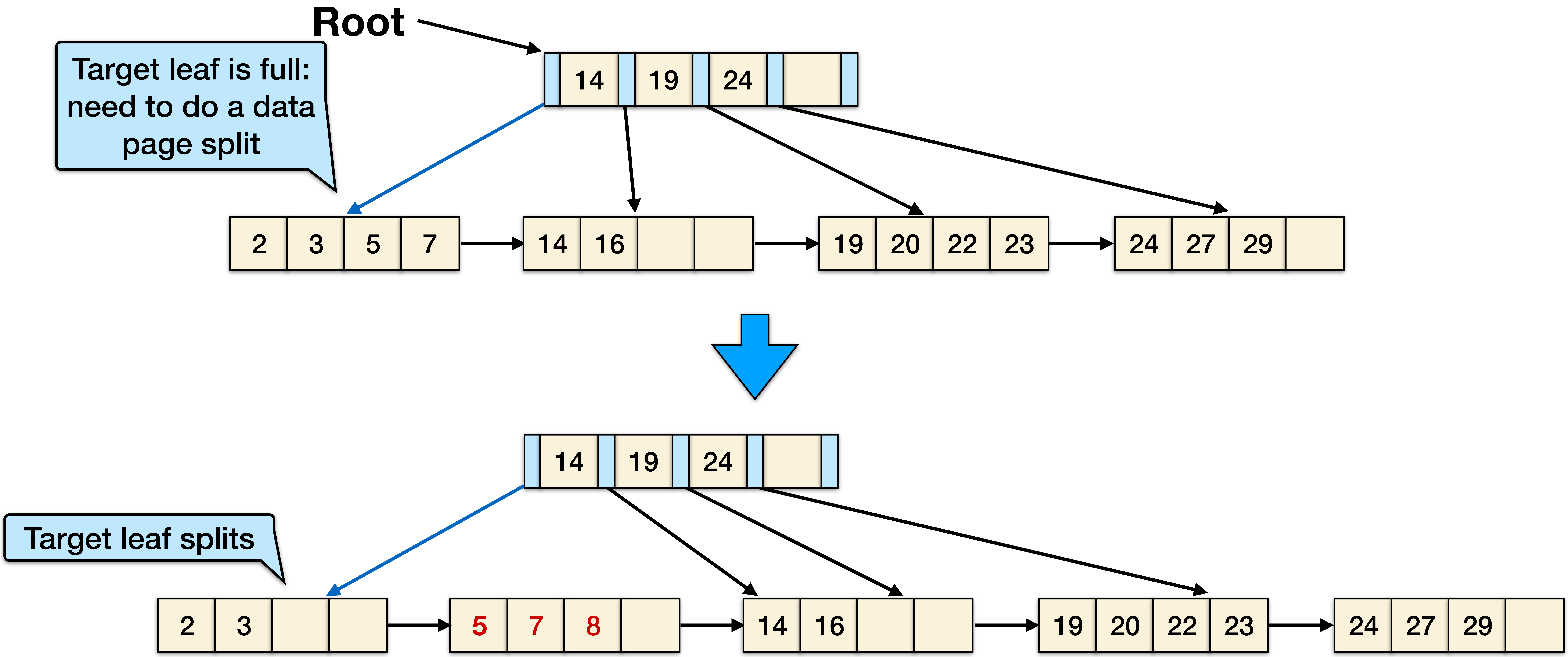
Point operations:

- `Insert(k, v)`: insert a key-value pair (`k, v`)
- `Find(k)`: return a pointer to the element with the smallest key that is at least k

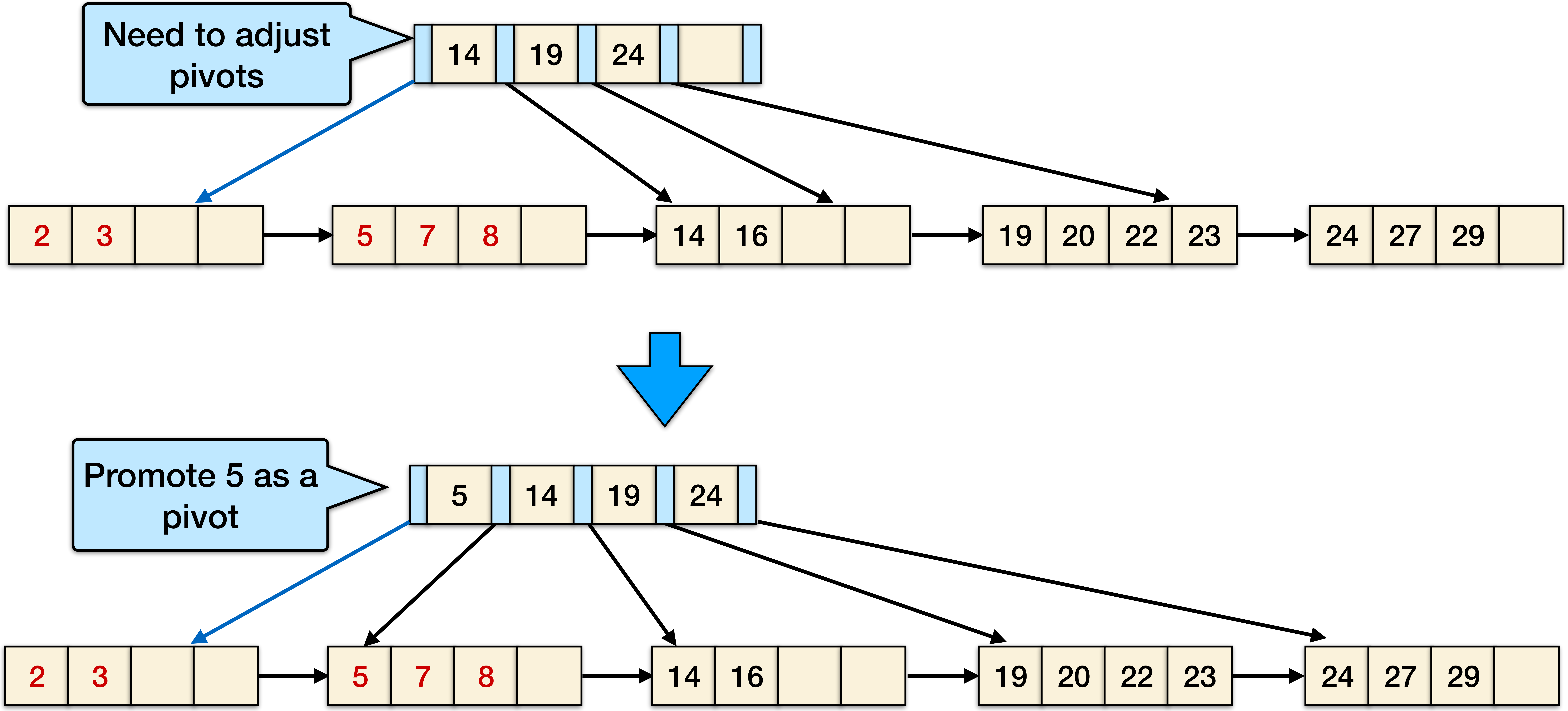
Example: `Find(15)`



Recap example: Insert 8 into B+-tree



Recap example: Insert 8 into B+-tree



Ordered Range Operations

The importance of ordered iteration in range operations (scans) depends on the **use case**.

For example, the YCSB requires **range iteration** (in sorted order) to simulate an application of threaded conversations:

`Iterate_range(start, length, f)`: applies the function `f` to `length` elements in order (by key) starting with the elements with the smallest key that is at least `start`

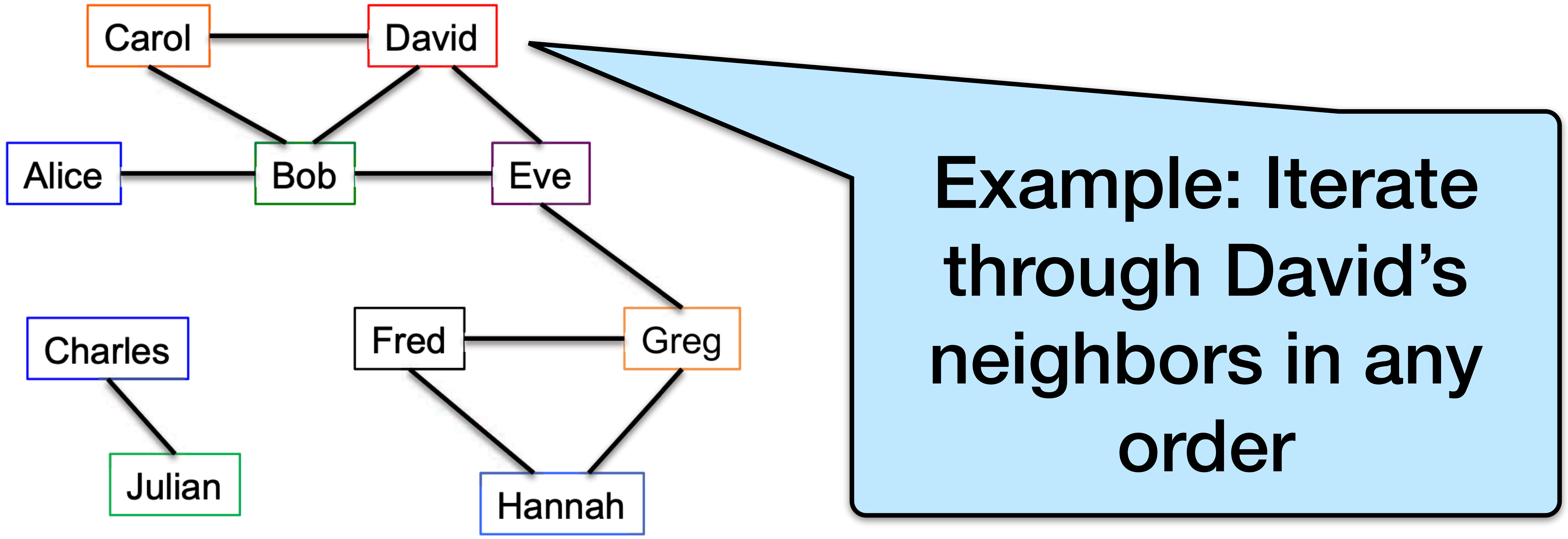
Example: Load the first 50 messages on some date

Unordered Range Operations

On the other hand, some applications **may not necessarily need access to the keys in order**.

For example: graph processing, feature storage in machine learning, file system metadata management.

Therefore, we consider another primitive not in YCSB:
`Map_range(start, end, f)`: applies the function `f` to all elements with keys in the range `[start, end)`

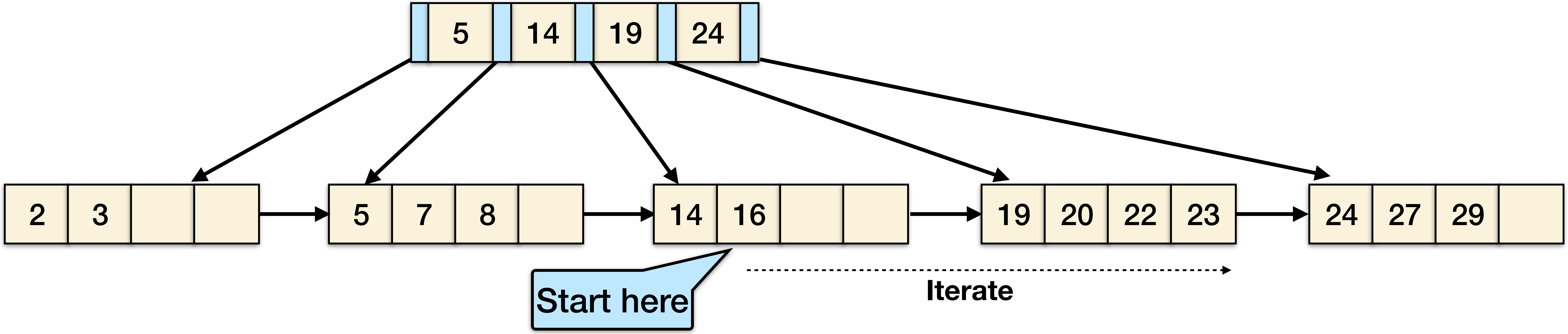


Recall: Range operations in B+-trees

Example: Get me 5 elements in sorted order with min key 15.

Step 1: Do a `find` for the element with the smallest key at least 15

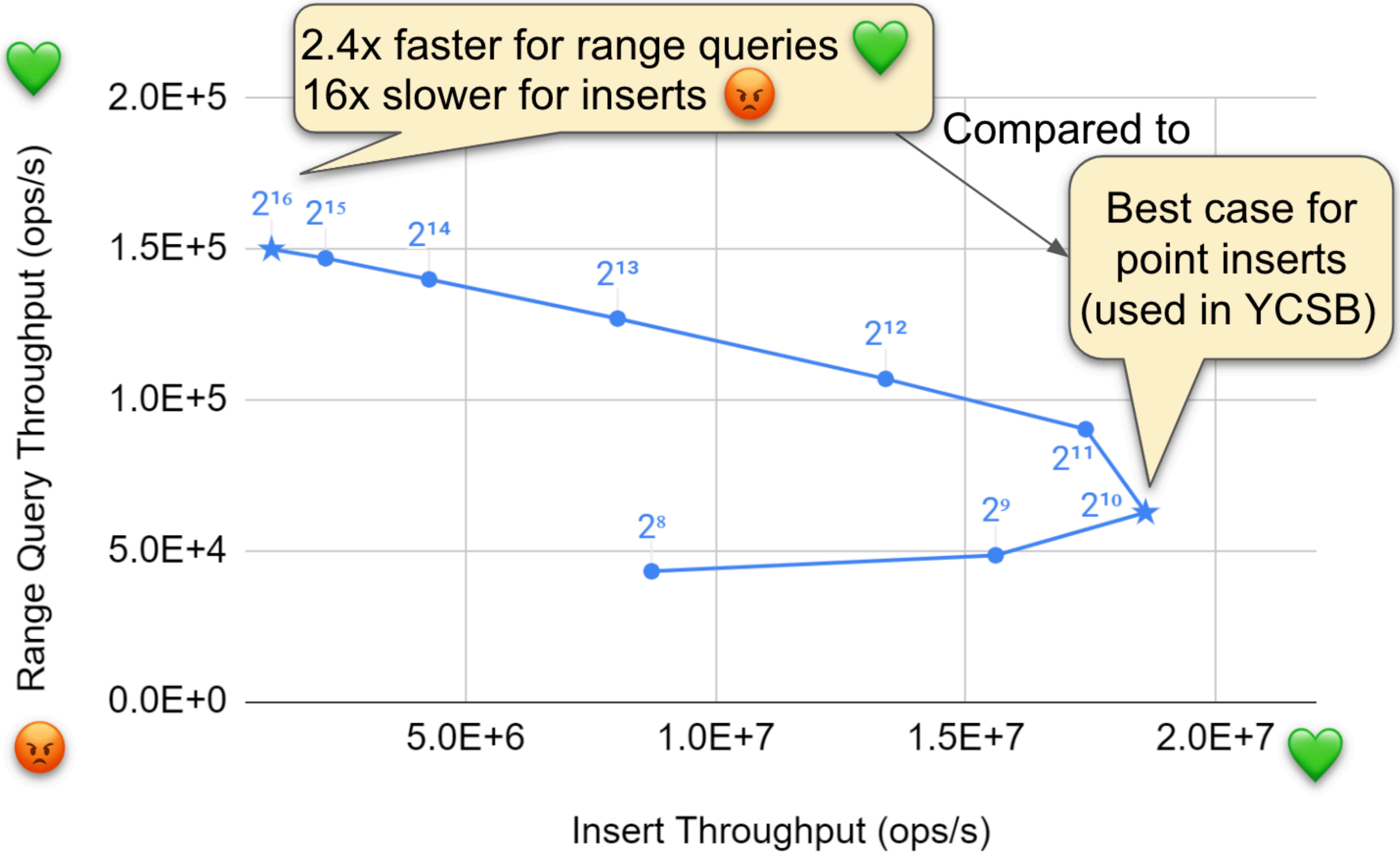
Step 2: Iterate forward 5 steps or until the end, whichever comes first



We can use a similar method for the other range API of [start, end) by just modifying the end condition.

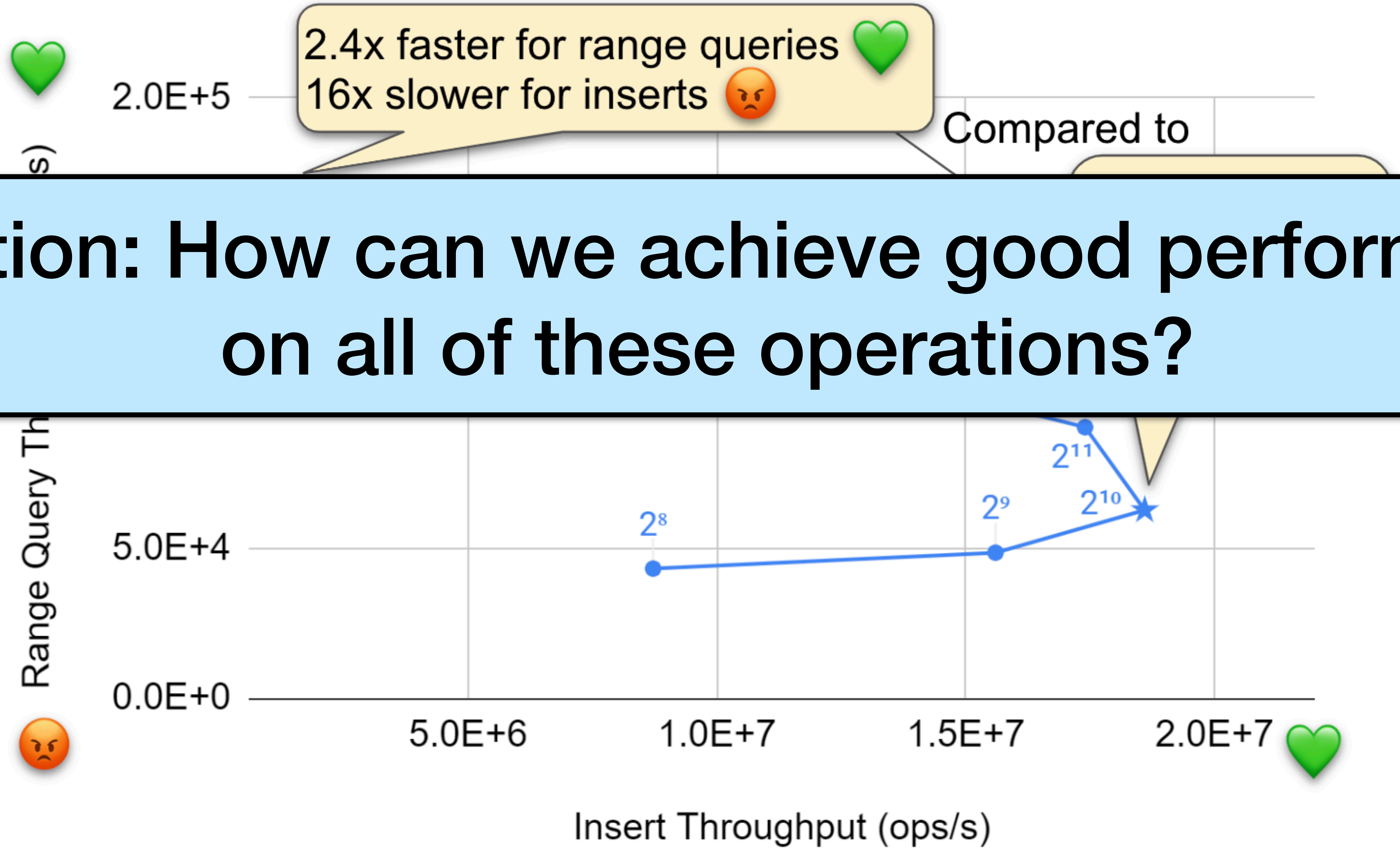
B-tree insert/range query trade-off

There is **no one best node size for all operations** - large node sizes improve range query throughput, but slow down inserts.



B-tree insert/range query trade-off

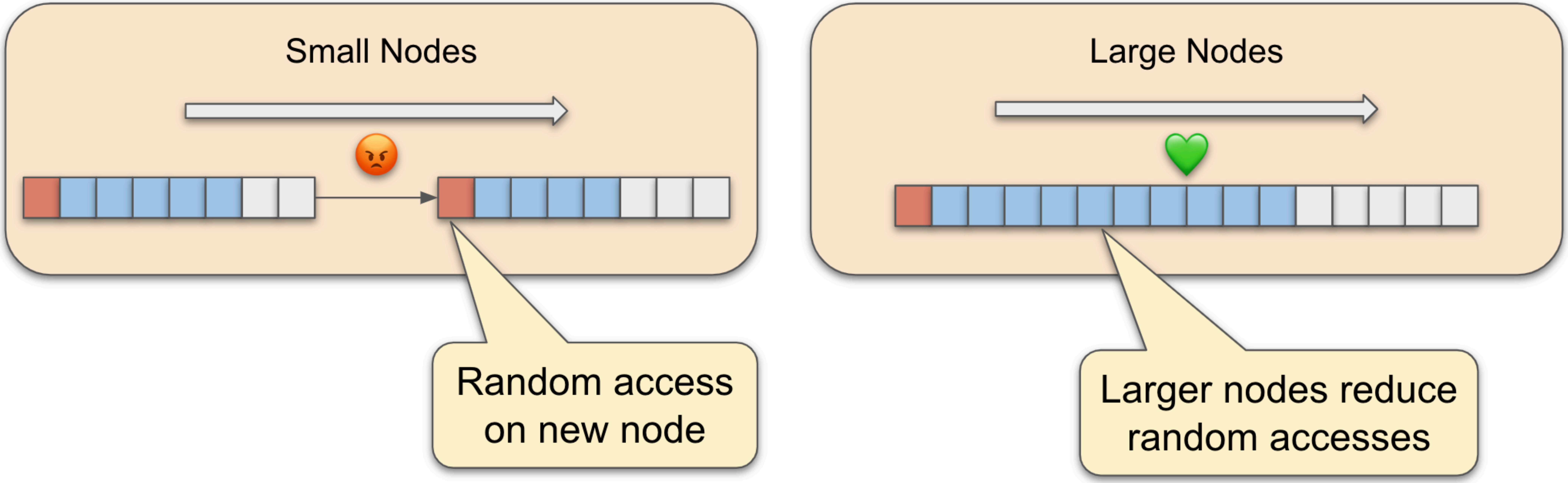
There is **no one best node size for all operations** - large node sizes improve range query throughput, but slow down inserts.



Question: How can we achieve good performance on all of these operations?

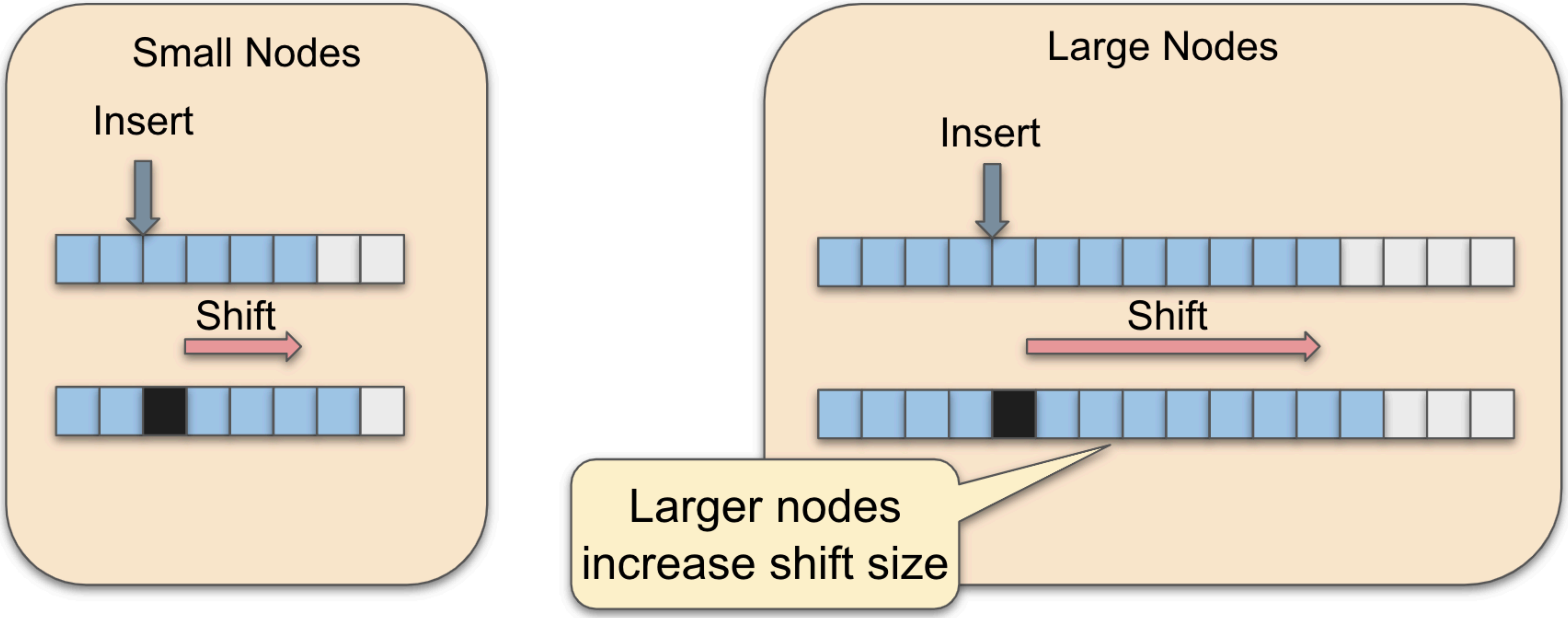
Larger nodes improve range query performance

Increasing the size of nodes decreases the number of nodes accessed during long range queries and thus the number of **random memory accesses**.



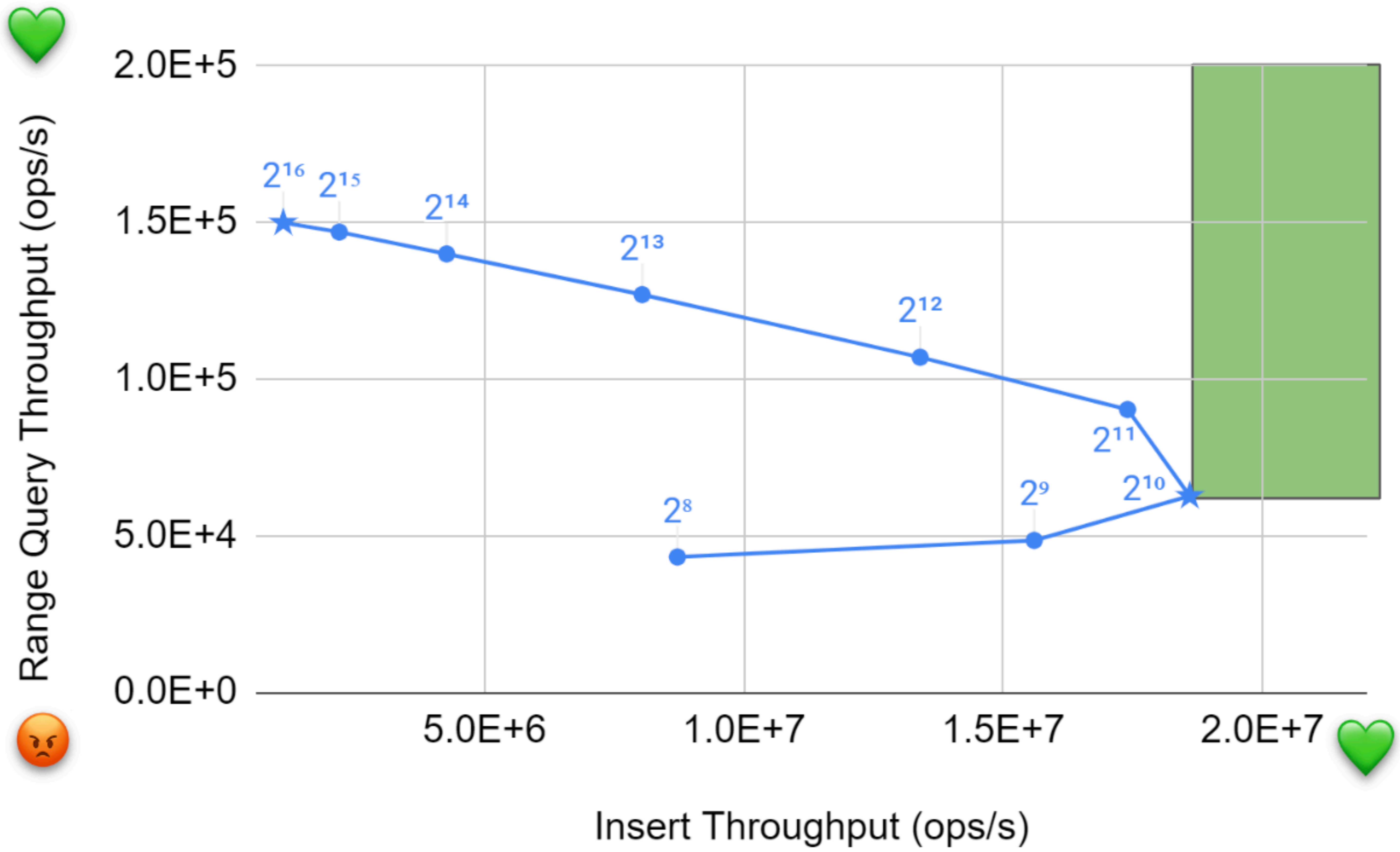
But larger nodes require more shifting on every insert

- However, simply increasing the node size does not solve the problem because larger nodes **require more work to maintain during inserts**
- Traditionally, B-trees (and B+-trees) use a **sorted array** to maintain elements in the nodes



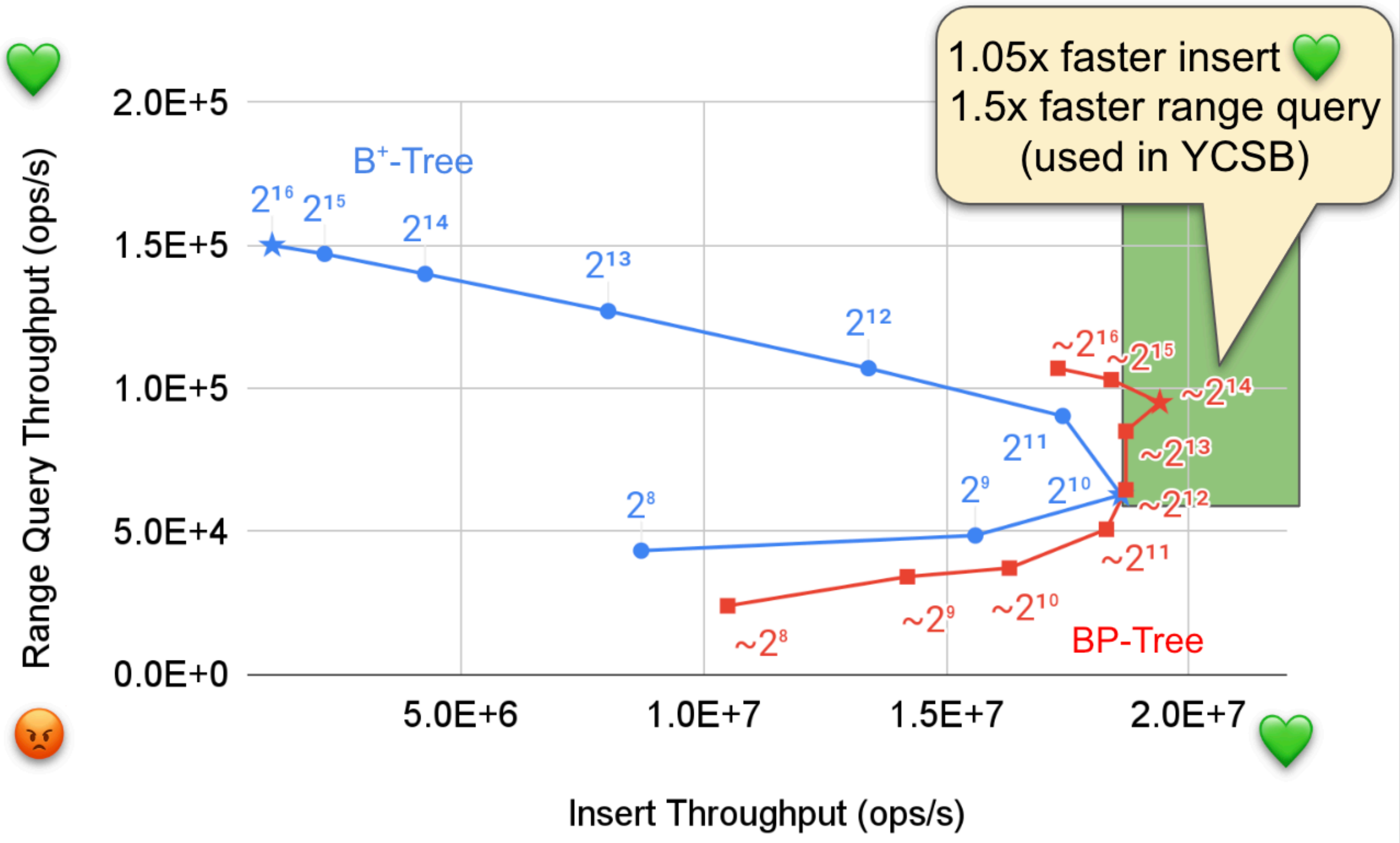
B-tree insert/range query tradeoff

How can we improve performance overall despite the insert/range tradeoff?



BP-tree: Overcoming the insert/range query tradeoff

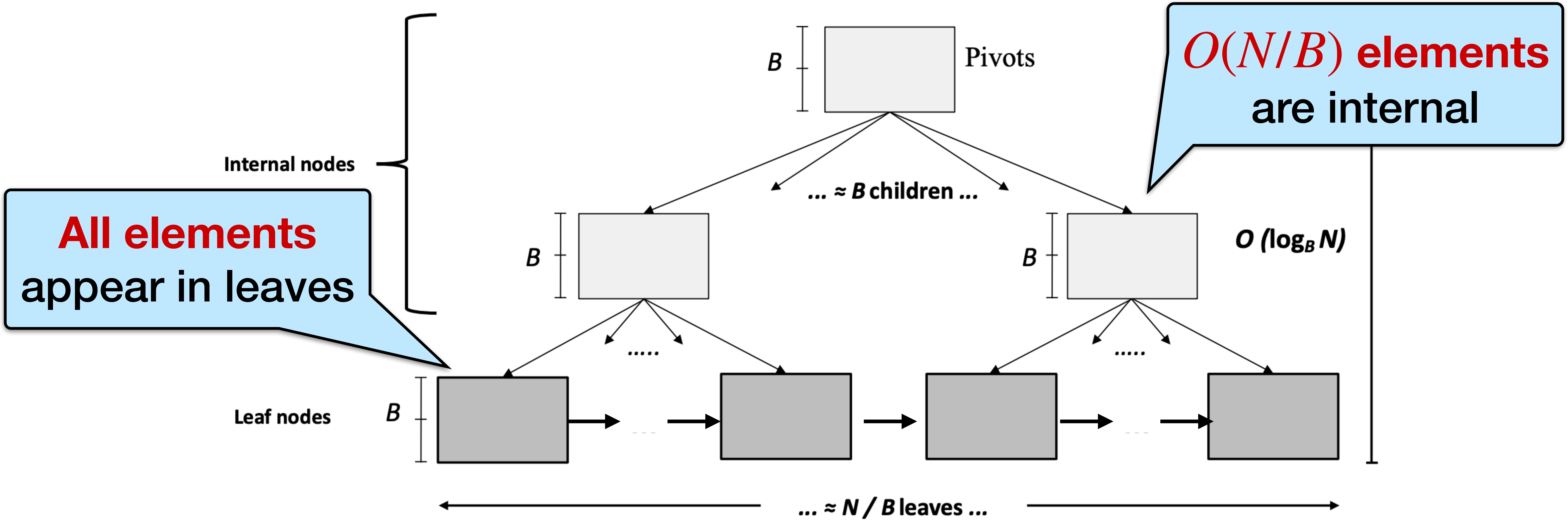
The BP-tree can improve long ranges without sacrificing point operations



BP-tree design

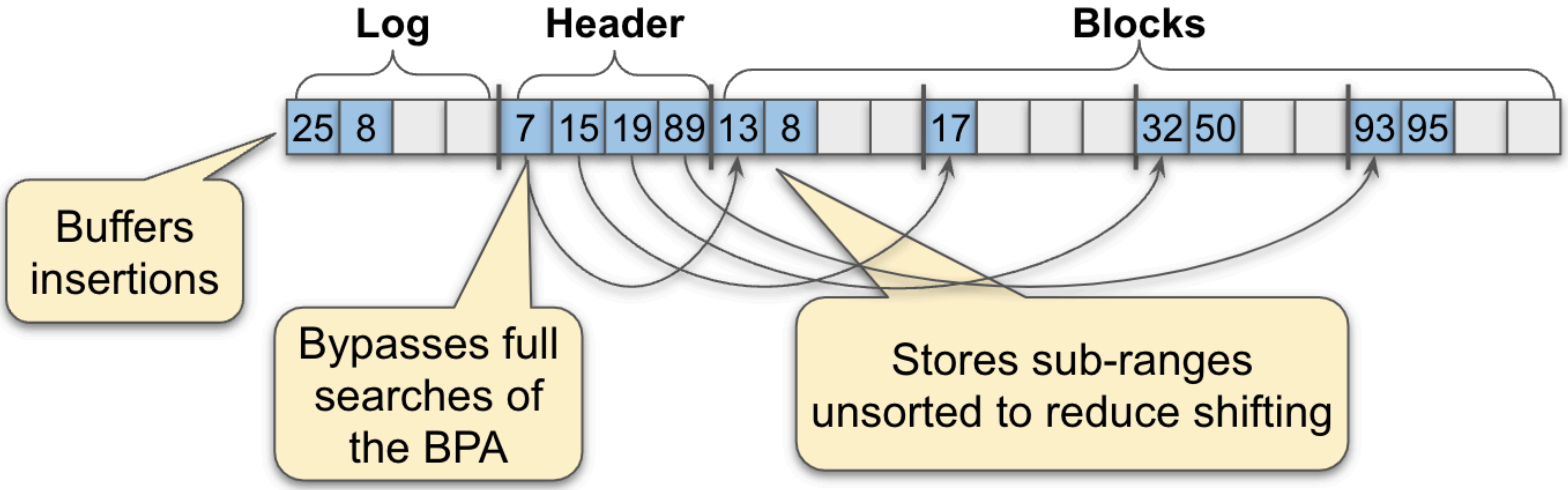
Motivation: Leaf nodes are the hotspots in B-tree variants

- Every insert will modify at least one leaf.
- Only one in every $O(B)$ inserts will affect the internal nodes.

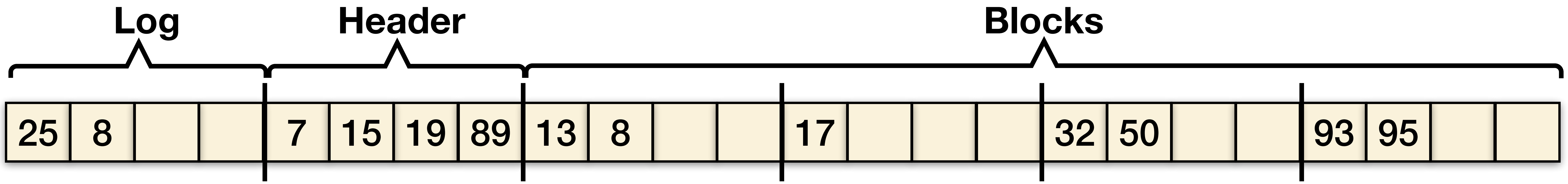


Buffered Partitioned Array (BPA) Design

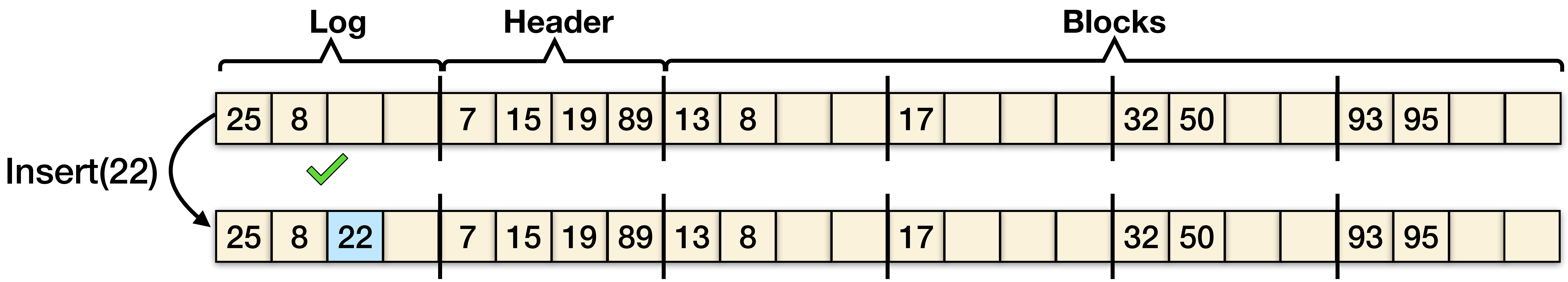
- The BP-tree overcomes the insert-range tradeoff by using large **nodes with an insert-optimized data structure in the leaves** called the Buffered Partitioned Array (BPA).
- One way to think about the BPA is like **collapsing the last two levels of a B-tree** into one insert-optimized array-like data structure.



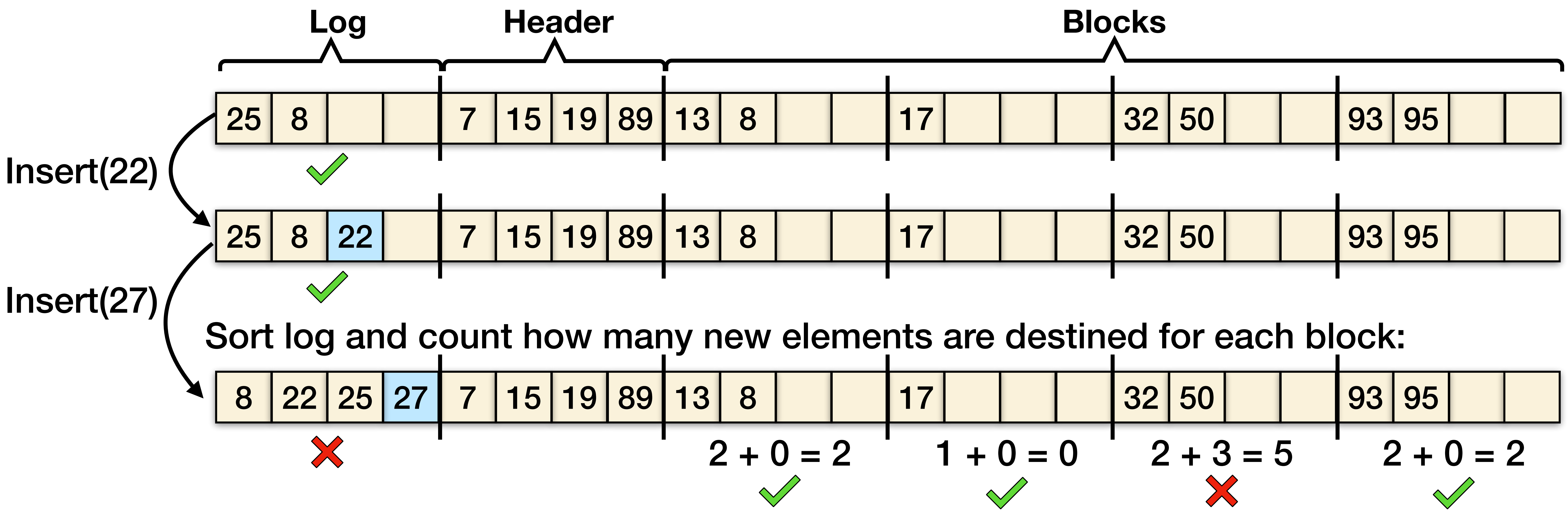
Example: Insertions in a BPA



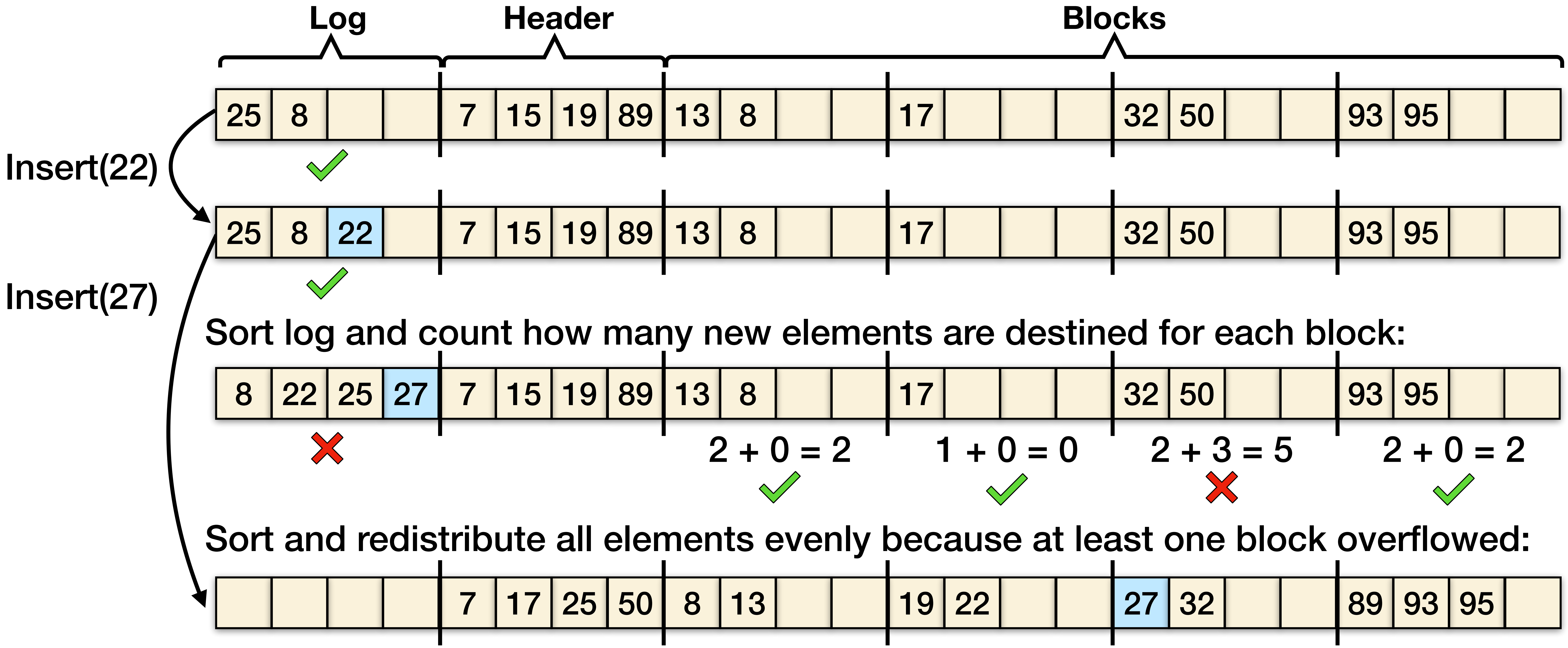
Example: Insertions in a BPA



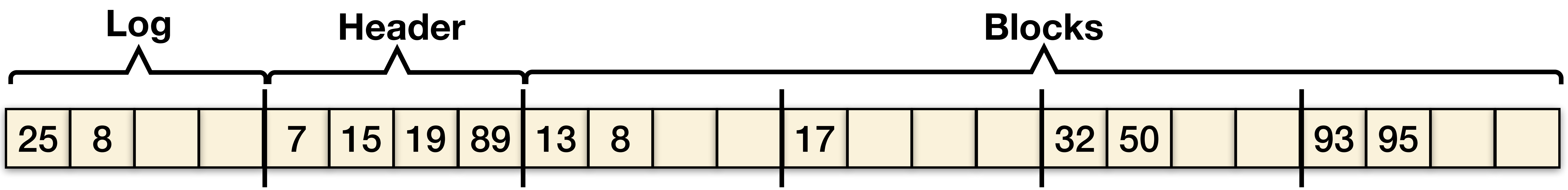
Example: Insertions in a BPA



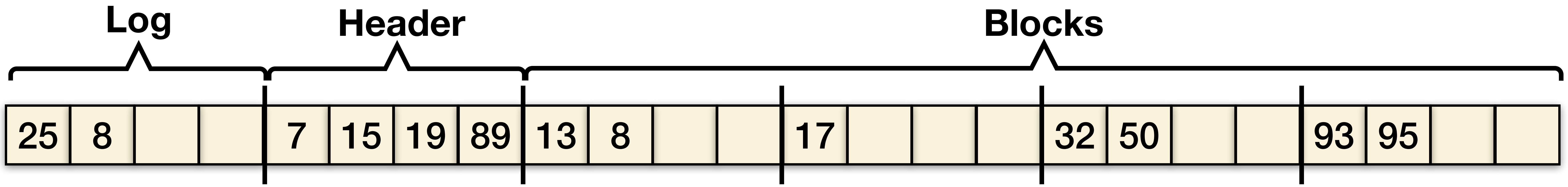
Example: Insertions in a BPA



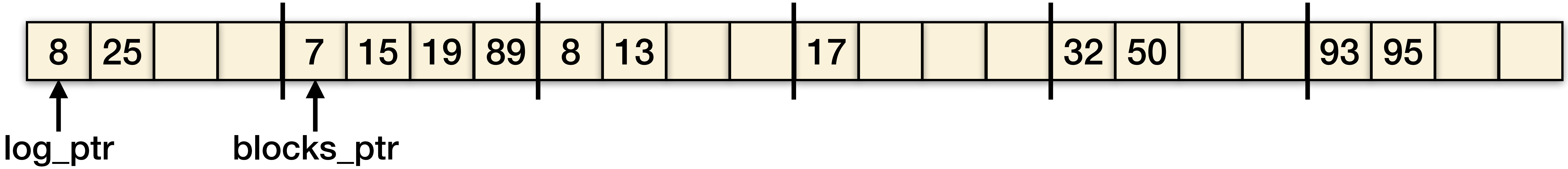
Example range query:
iterate_range(start = 7, length = 2, f)



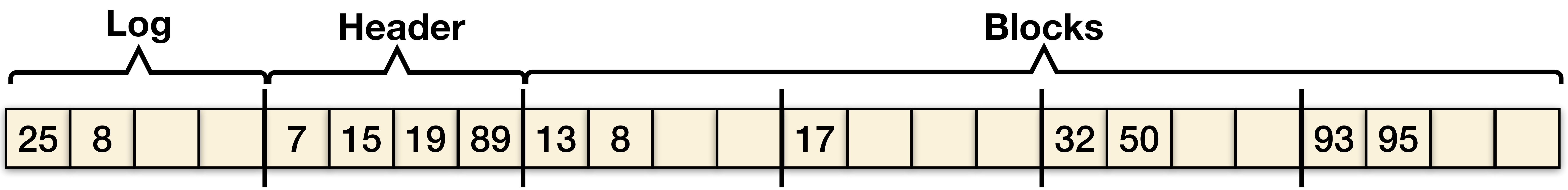
Example range query:
iterate_range(start = 7, length = 2, f)



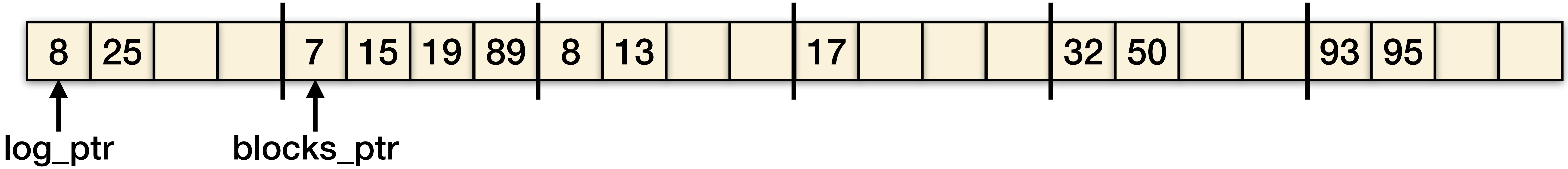
Sort the log and first relevant block, initialize the pointers:



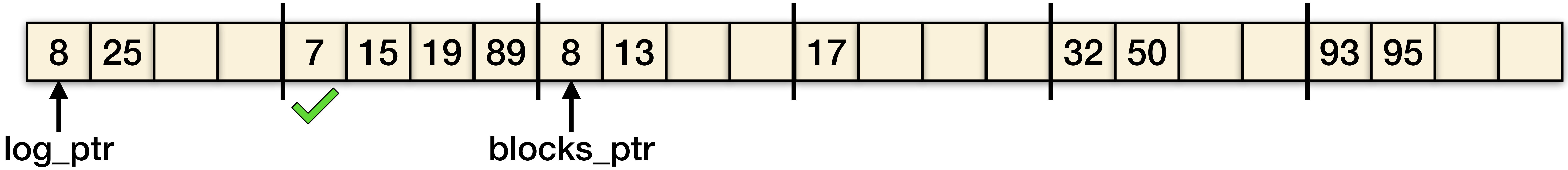
Example range query:
iterate_range(start = 7, length = 2, f)



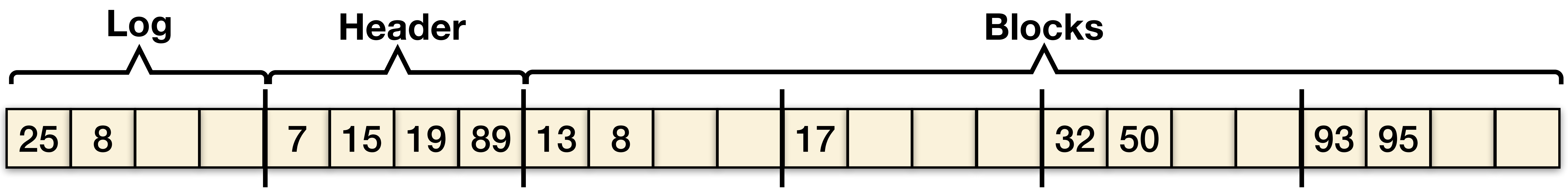
Sort the log and first relevant block, initialize the pointers:



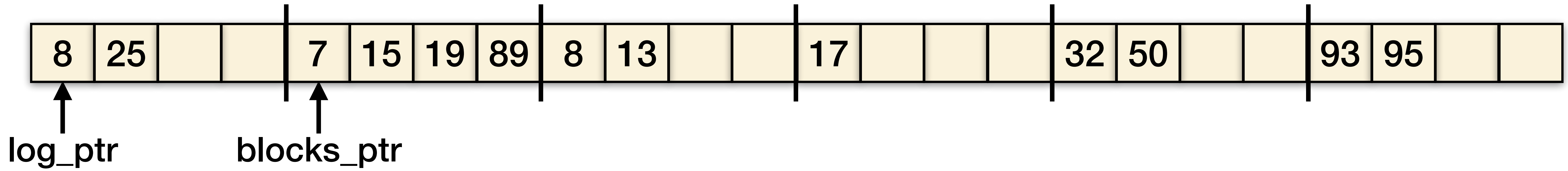
Advance the pointers to perform sorted iteration:



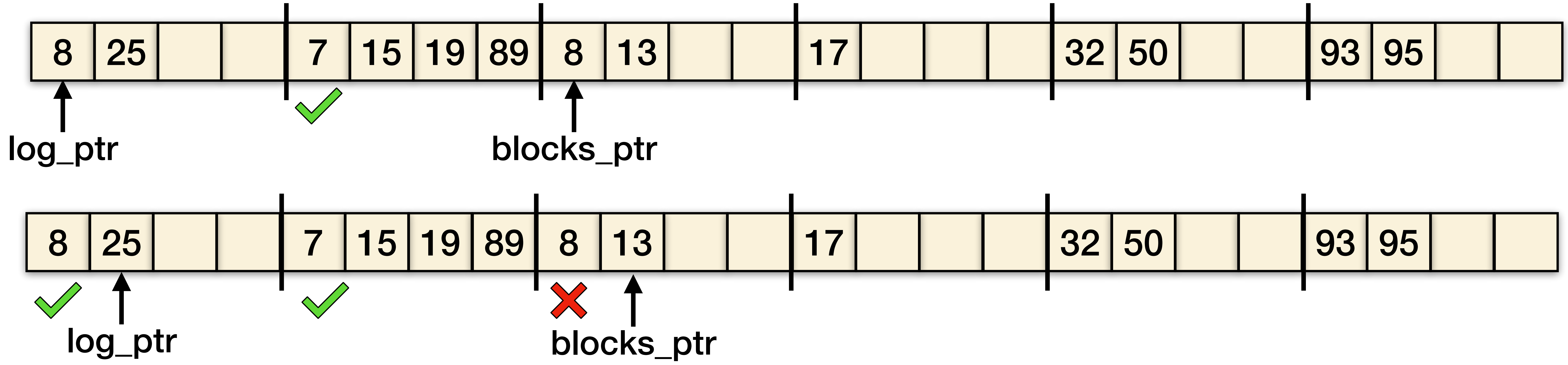
Example range query: iterate_range(start = 7, length = 2, f)



Sort the log and first relevant block, initialize the pointers:

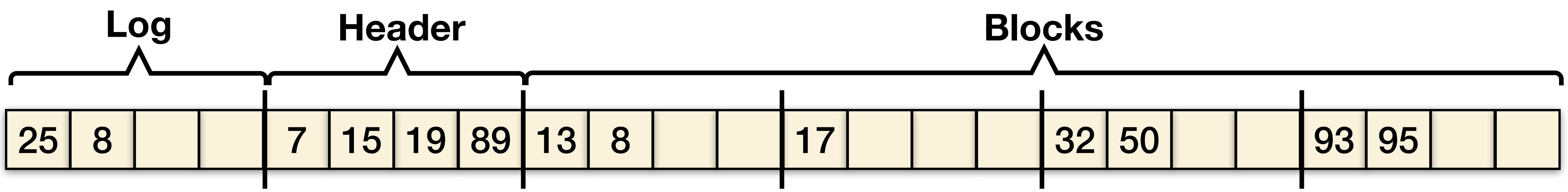


Advance the pointers to perform sorted iteration:



Bitvector Optimization

- To avoid unnecesary sorting, the BPA keeps a bit vector of length `num_blocks` that denotes **whether the elements in each block are currently sorted**.
- It sorts a block during a range query if and only if the corresponding bit in the bit vector is unset.
- The bit vector is maintained during inserts / range queries.

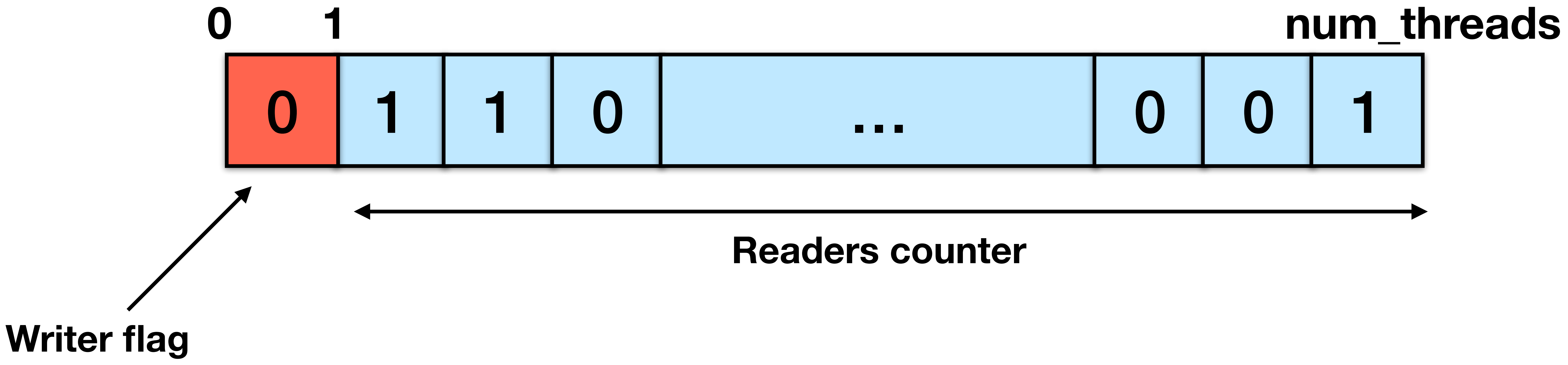


Bitvector: 0111

BP-tree concurrency control

Recall: Reader-Writer Concurrency

- A reader-writer lock allows **concurrent access for read-only operations**, whereas write operations require exclusive access.
- That is, multiple threads can read the data in parallel, but **an exclusive lock is needed for writing/modifying data**.
- All other threads (both writers and readers) are blocked when the lock is taken in write mode.



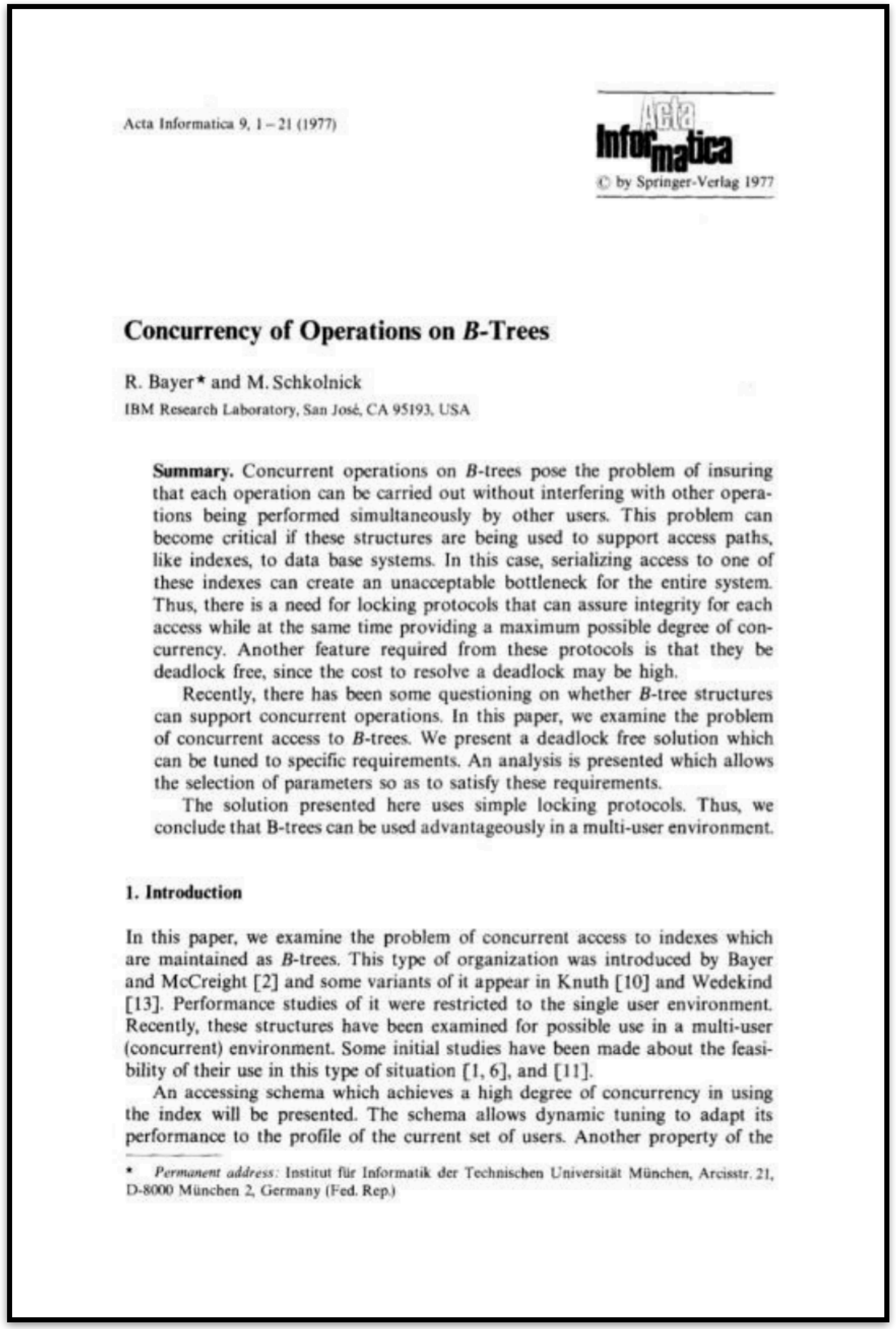
Recall: Optimistic concurrency control

Concurrency control is defined **at the node level**, so we can use the same **reader/ writer concurrency** scheme for inserts as regular B-trees.

Most modifications to a B+-tree will **not** require a split or merge.

Instead of assuming that there will be a split/merge, **optimistically traverse** the tree using read latches.

If you guess wrong, **repeat traversal** with the pessimistic algorithm.


Acta Informatica 9, 1–21 (1977)

Acta Informatica

© by Springer-Verlag 1977

Concurrency of Operations on *B*-Trees

R. Bayer* and M. Schkolnick

IBM Research Laboratory, San José, CA 95193, USA

Summary. Concurrent operations on *B*-trees pose the problem of insuring that each operation can be carried out without interfering with other operations being performed simultaneously by other users. This problem can become critical if these structures are being used to support access paths, like indexes, to data base systems. In this case, serializing access to one of these indexes can create an unacceptable bottleneck for the entire system. Thus, there is a need for locking protocols that can assure integrity for each access while at the same time providing a maximum possible degree of concurrency. Another feature required from these protocols is that they be deadlock free, since the cost to resolve a deadlock may be high.

Recently, there has been some questioning on whether *B*-tree structures can support concurrent operations. In this paper, we examine the problem of concurrent access to *B*-trees. We present a deadlock free solution which can be tuned to specific requirements. An analysis is presented which allows the selection of parameters so as to satisfy these requirements.

The solution presented here uses simple locking protocols. Thus, we conclude that B-trees can be used advantageously in a multi-user environment.

1. Introduction

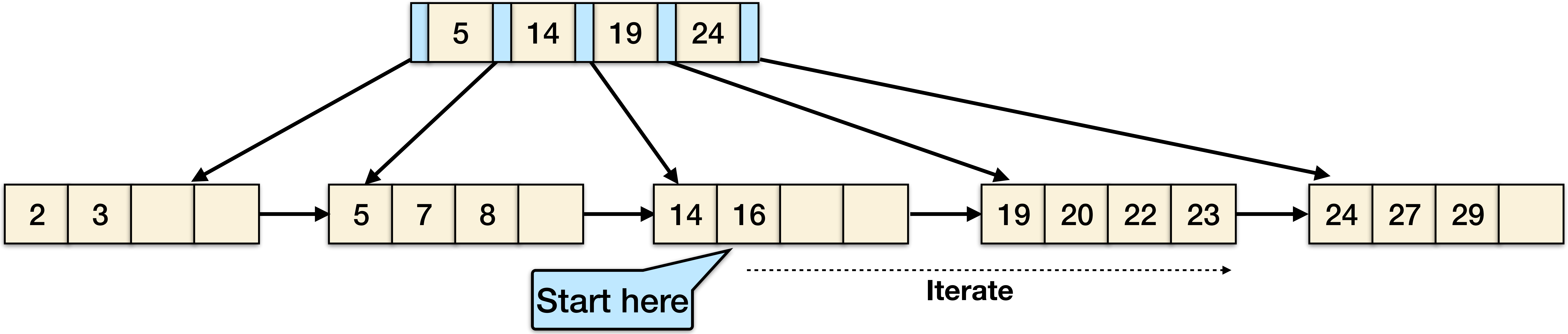
In this paper, we examine the problem of concurrent access to indexes which are maintained as *B*-trees. This type of organization was introduced by Bayer and McCreight [2] and some variants of it appear in Knuth [10] and Wedekind [13]. Performance studies of it were restricted to the single user environment. Recently, these structures have been examined for possible use in a multi-user (concurrent) environment. Some initial studies have been made about the feasibility of their use in this type of situation [1, 6], and [11].

An accessing schema which achieves a high degree of concurrency in using the index will be presented. The schema allows dynamic tuning to adapt its performance to the profile of the current set of users. Another property of the

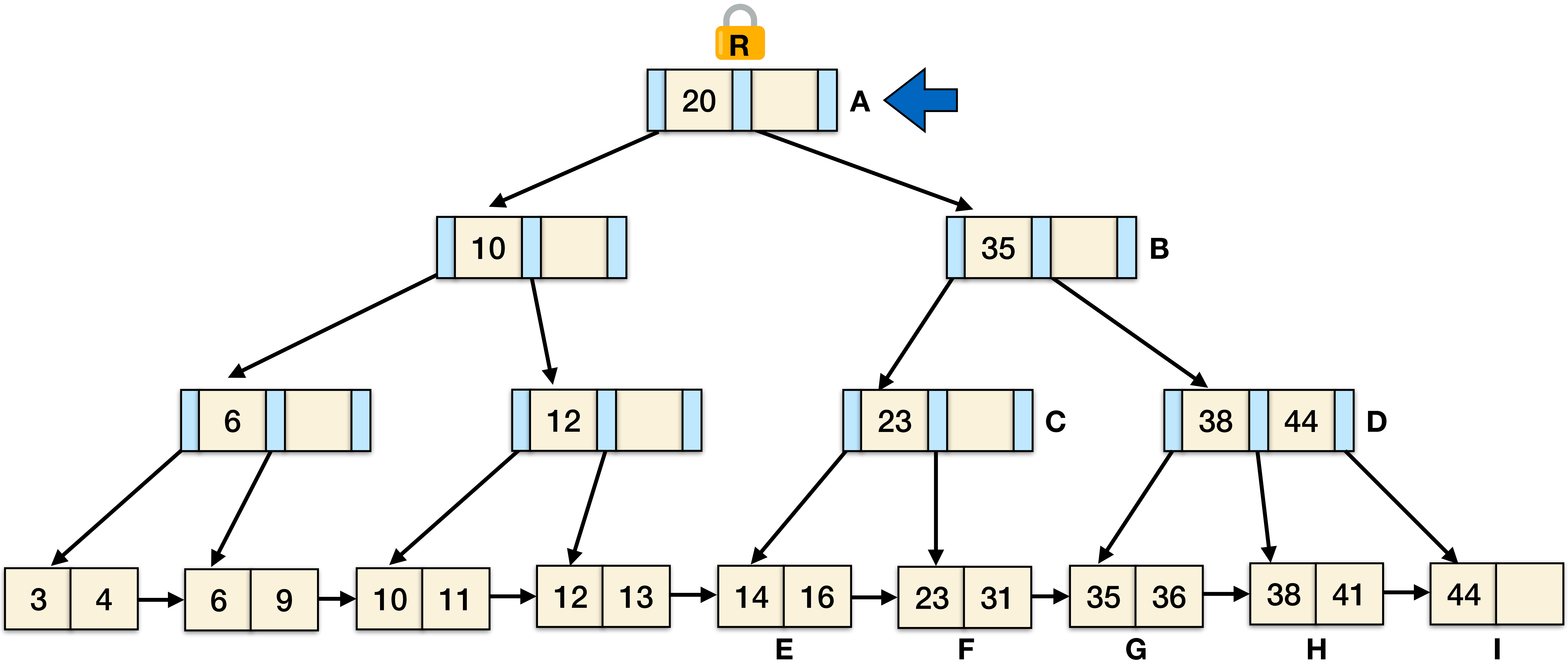
* *Permanent address:* Institut für Informatik der Technischen Universität München, Arcisstr. 21, D-8000 München 2, Germany (Fed. Rep.)

Range Query Concurrency Control

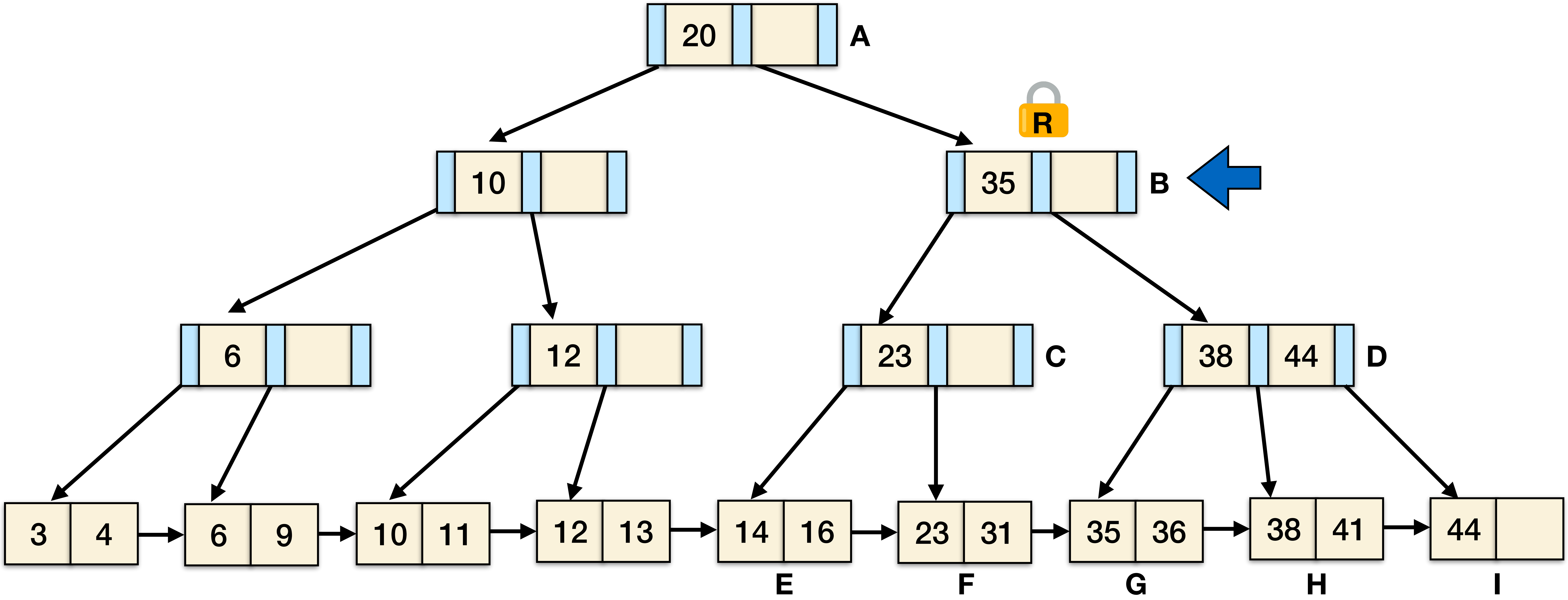
In regular B-trees, range operations are read-only, so we can just take **read locks top-down, left-right**.



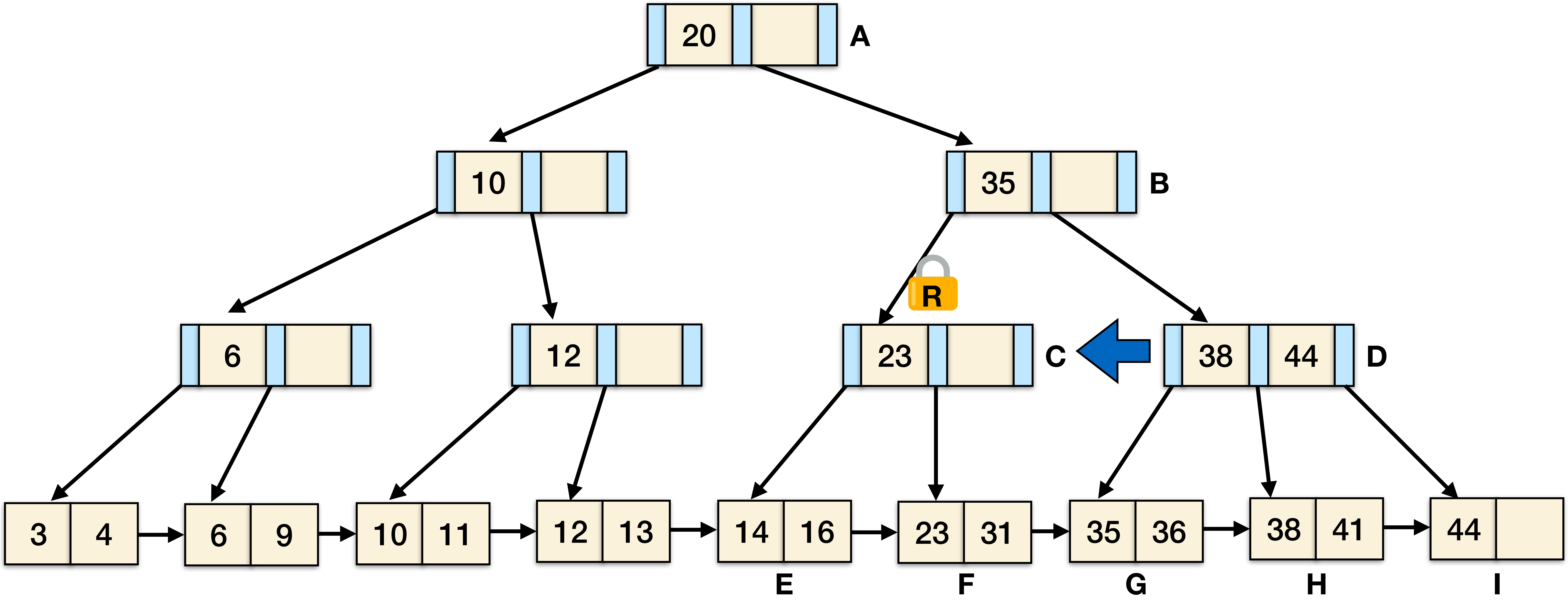
Example: iterate_range(start=25, length=2)



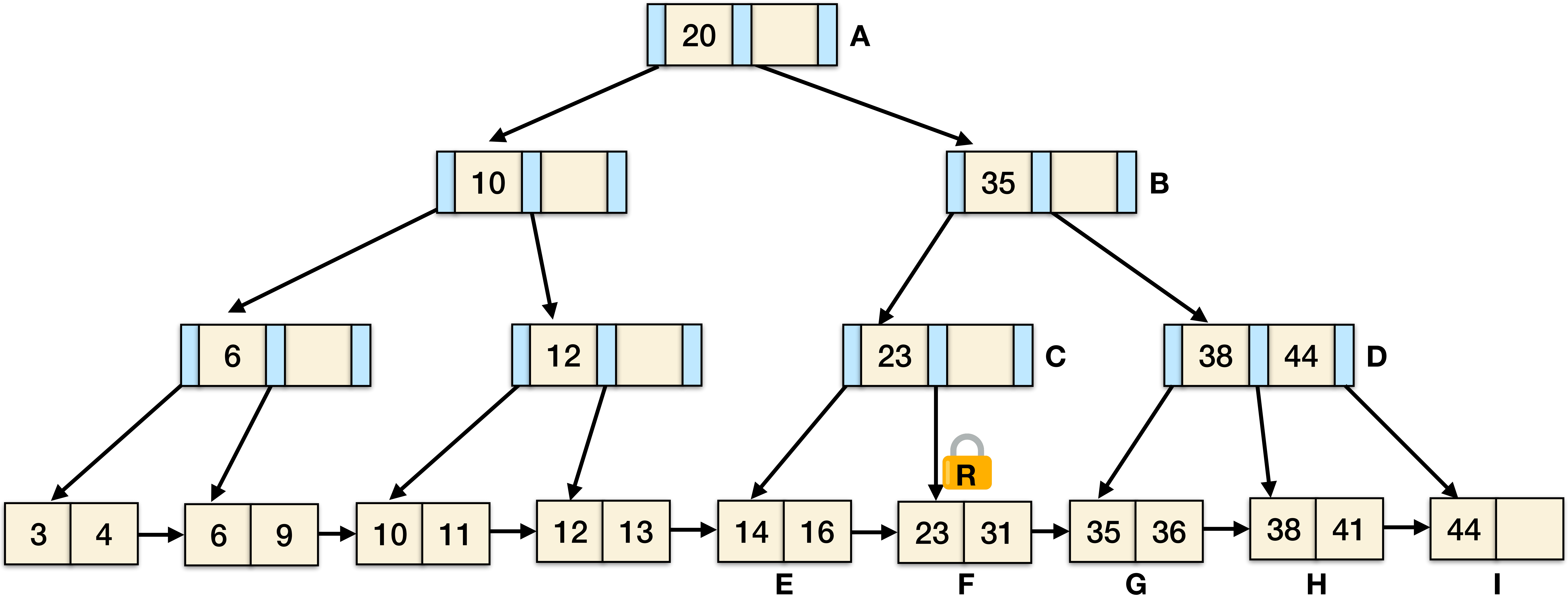
Example: iterate_range(start=25, length=2)



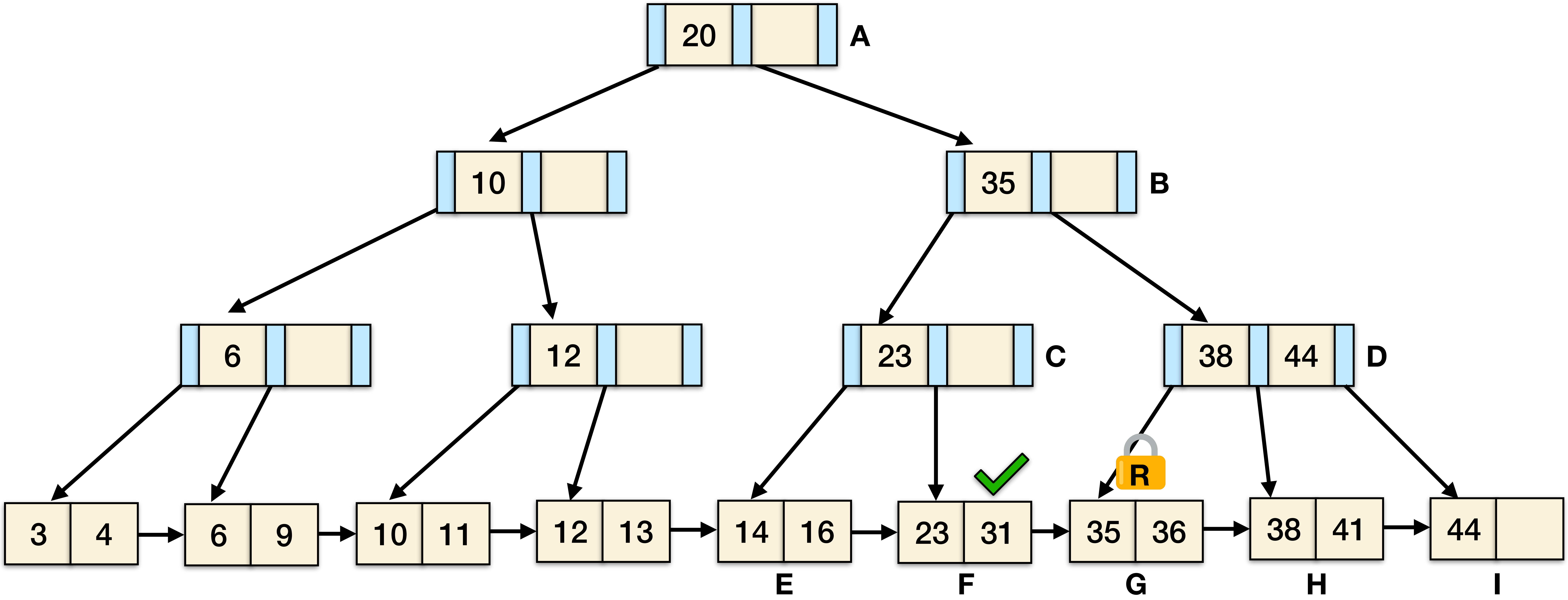
Example: iterate_range(start=25, length=2)



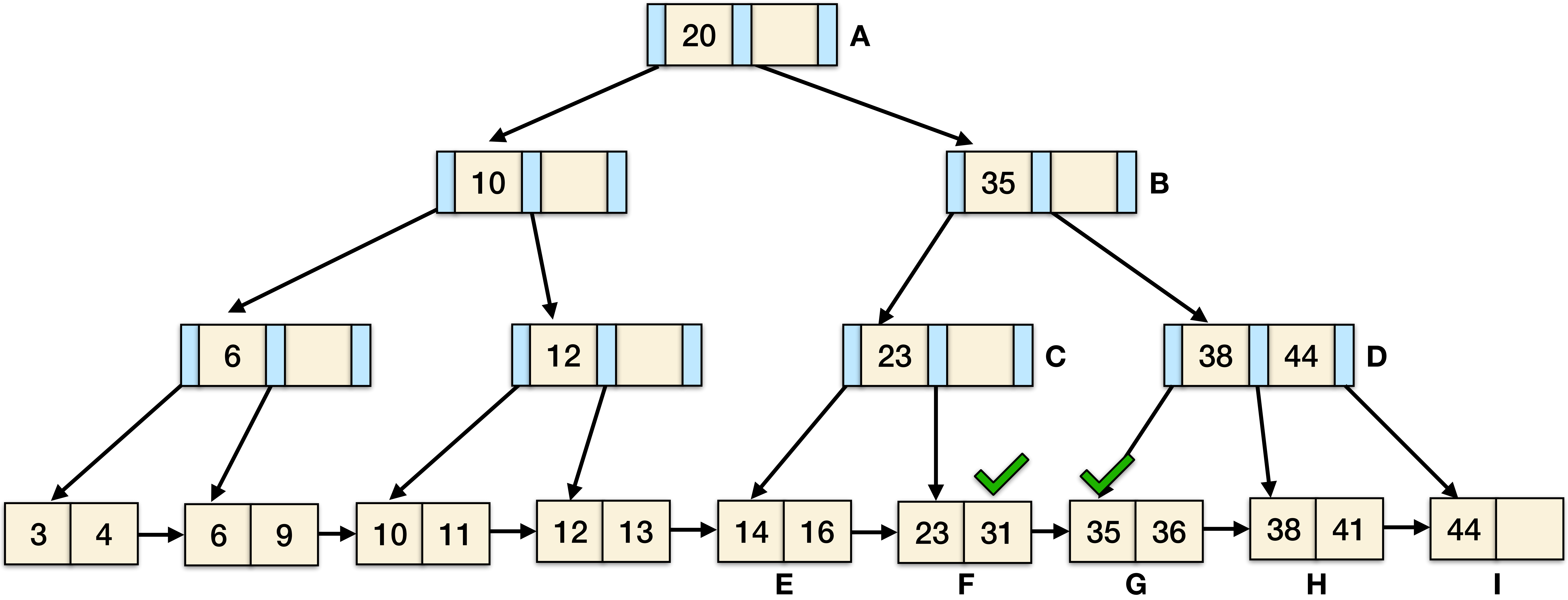
Example: iterate_range(start=25, length=2)



Example: iterate_range(start=25, length=2)



Example: iterate_range(start=25, length=2)



Question: Can we use this scheme for range query concurrency in the BP-tree?

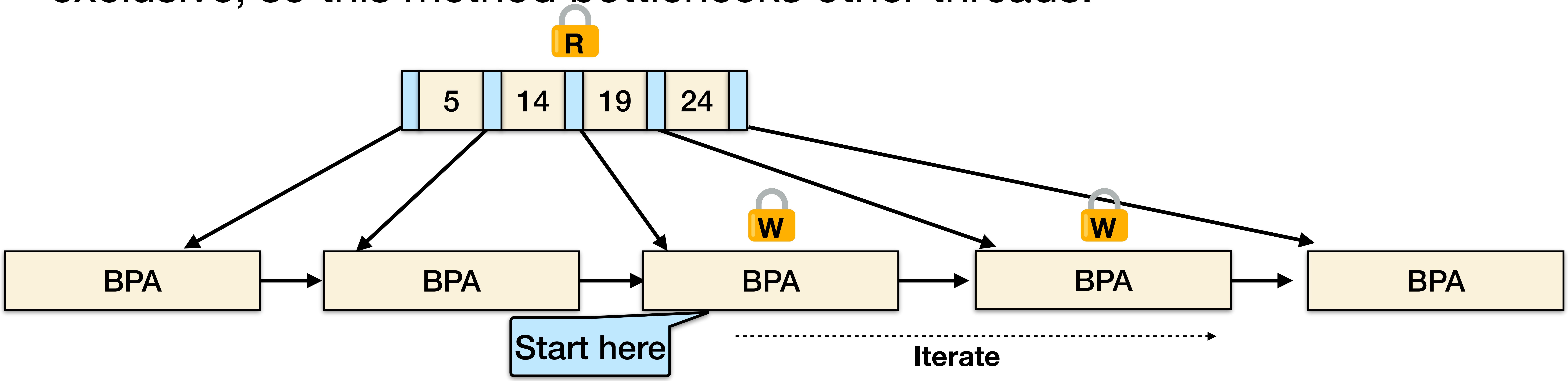
From Utah CS6530

Adapting B-tree Range Query Concurrency for the BP-tree

Problem: Range queries in the BPA **might modify the array** (to sort the log / blocks), so we can't always take a reader lock on the leaves.

Naive solution: Take read locks on the way down, then always take **writer locks on the leaves**.

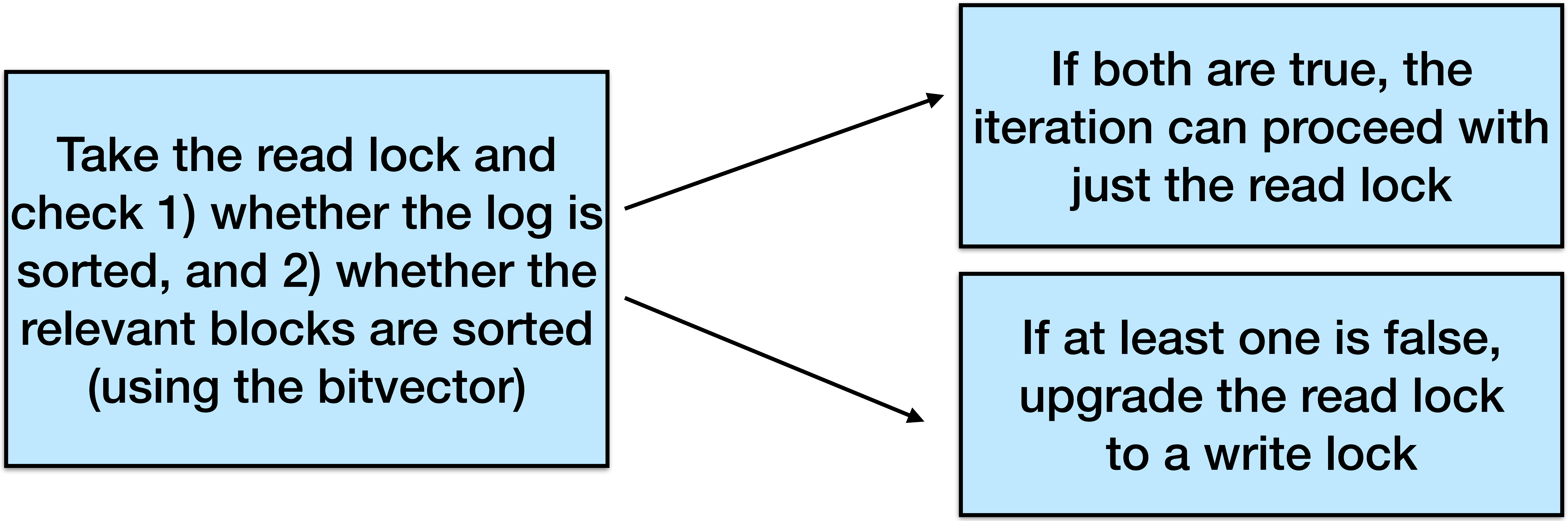
The naive solution causes **performance issues** because write locks are exclusive, so this method bottlenecks other threads.



Using the bitvector to avoid taking the write lock

We use the bitvector optimization to **avoid contention on the write lock** when the input distribution is skewed.

For each leaf touched in a range query in the BP-tree:

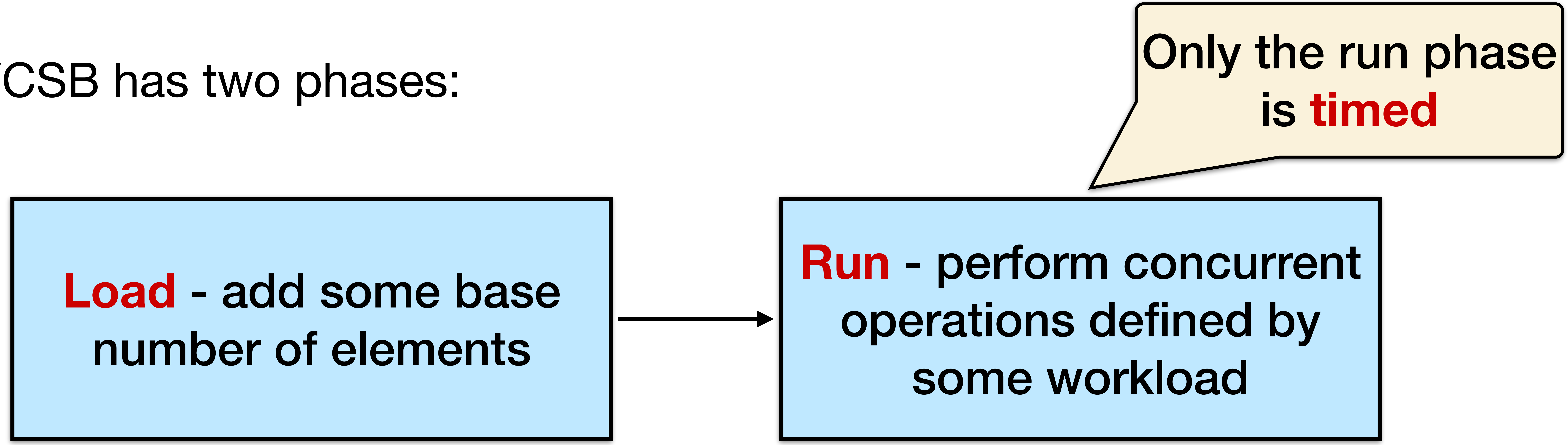


Evaluation

YCSB Evaluation Framework

We evaluate the BP-Tree on several tests using the Yahoo! cloud serving benchmark (YCSB) and compare it to a selection of different structures.

The YCSB has two phases:



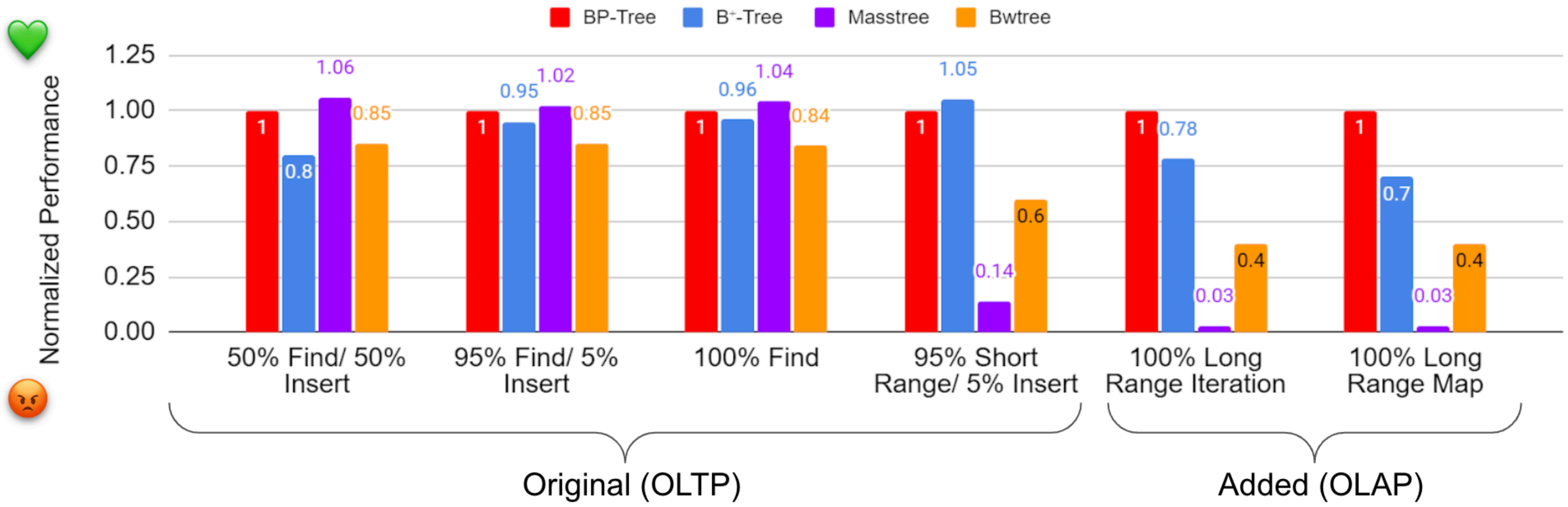
For concreteness, each phase has 100M operations. The YCSB also allow definition of input distribution (e.g., uniform random, skewed, etc.)

BP-tree system/experiment setup

- 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

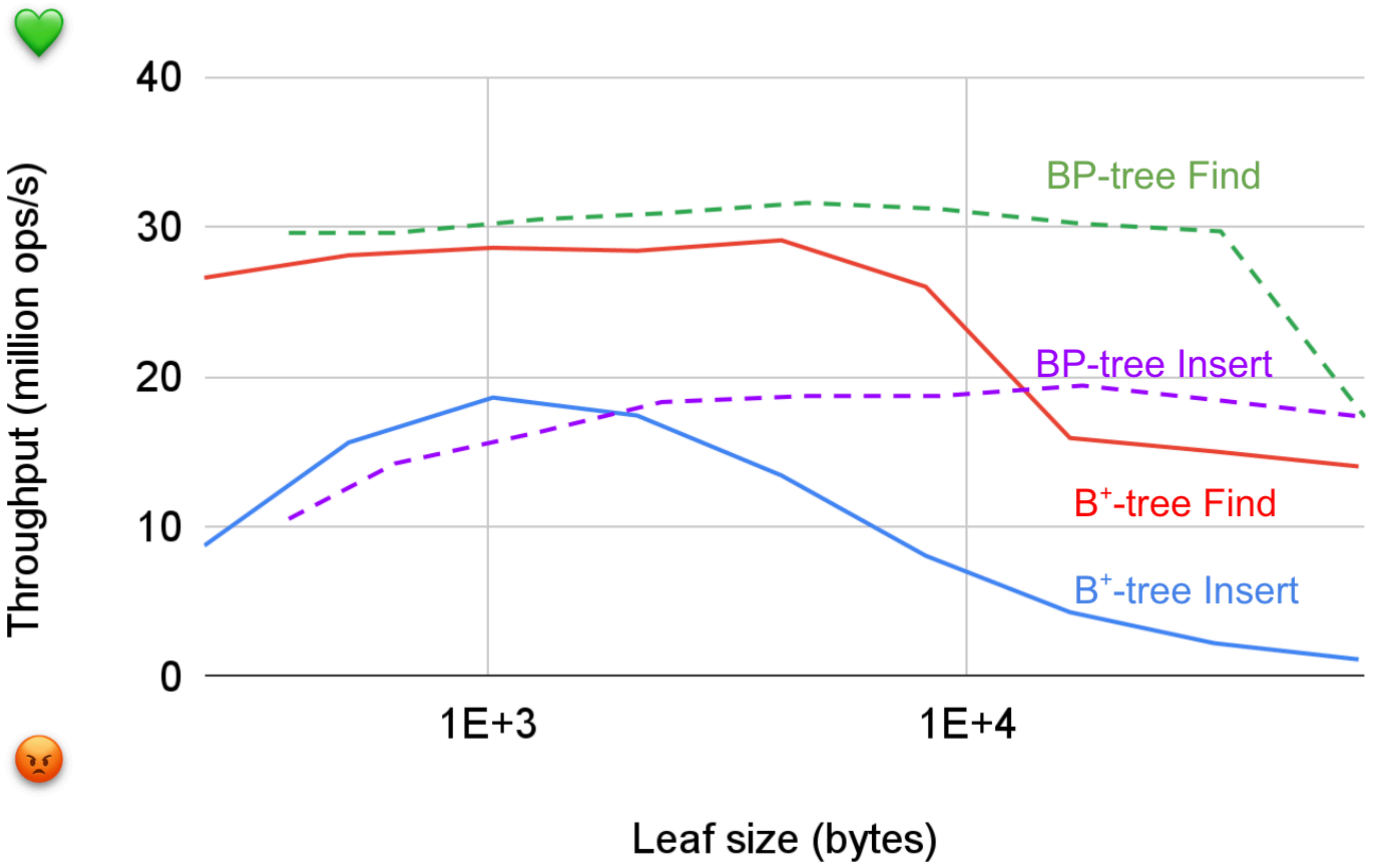
Evaluation on YCSB benchmarks

Performance of B-tree, Masstree [MKM2012], OpenBwTree [WPL+2018] and BP-tree on YCSB [CST+10] with 100M ops in both the load and run phase.

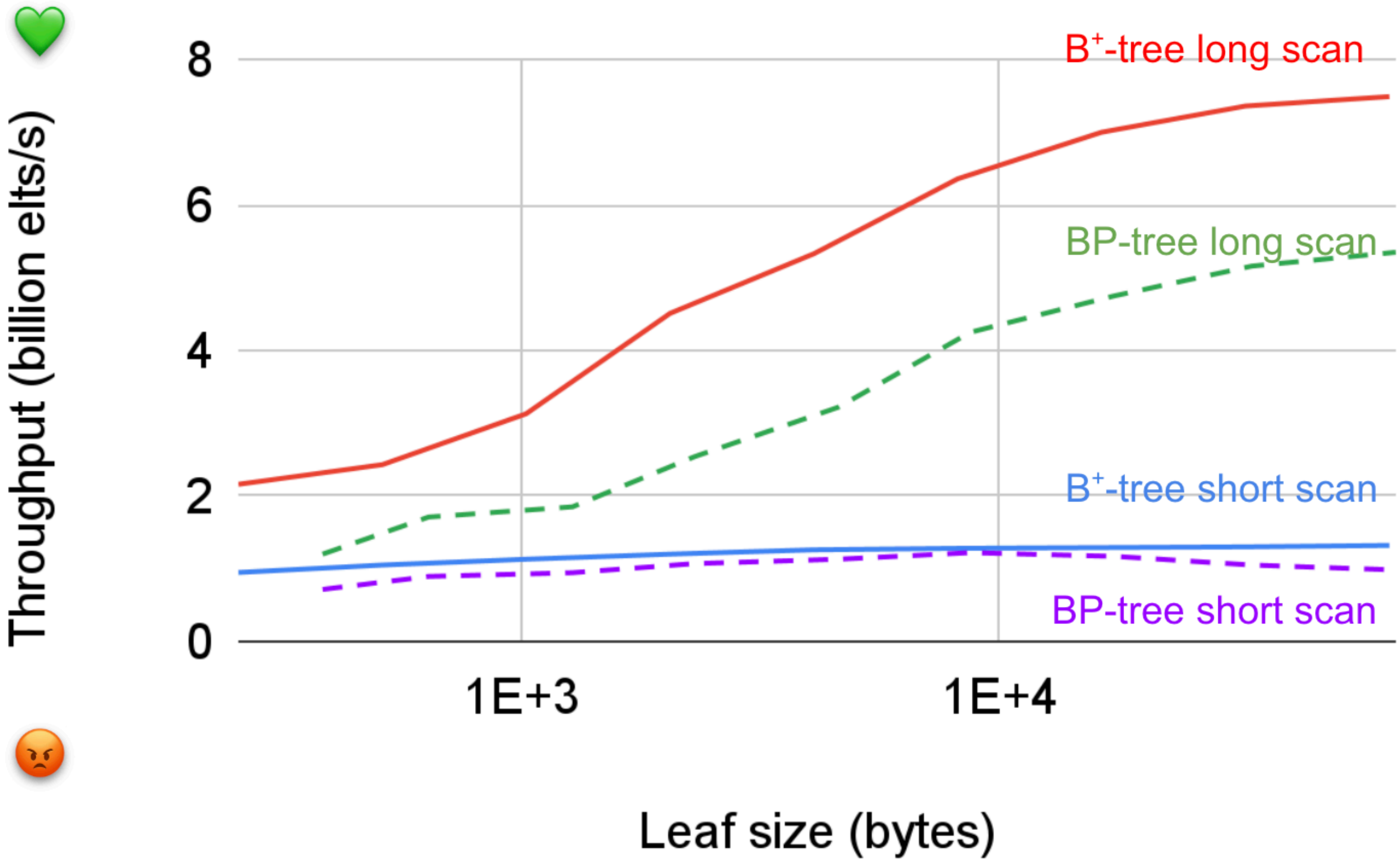


BP-tree matches the performance of point operations and improves range queries by 1.5x

B-tree vs BP-tree point operations



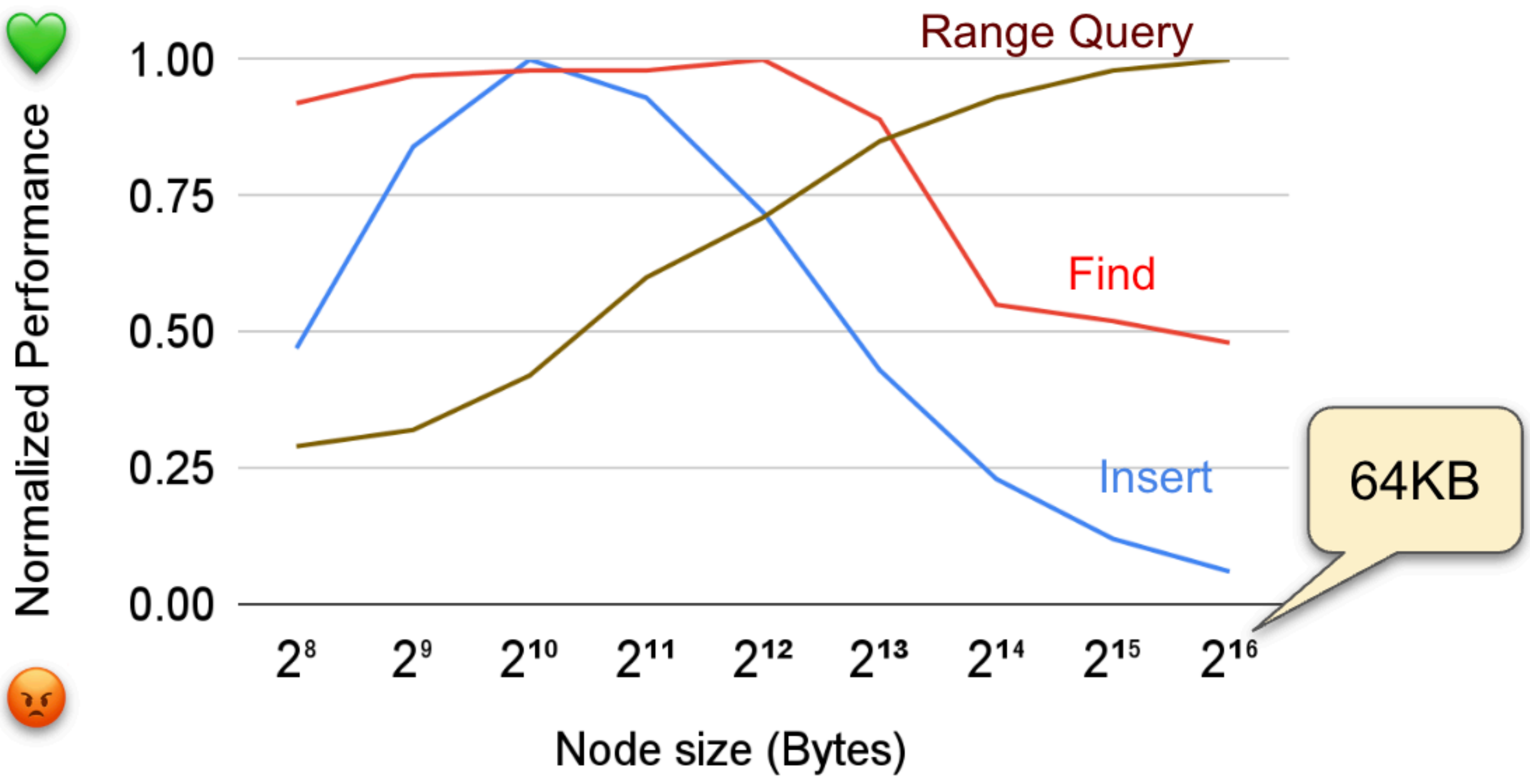
B-tree vs BP-tree range queries



Performance Modeling of Large Nodes

To what extent do big nodes help range queries?

- Traditionally node sizes are small (up to 256 bytes) [CGM01, HP03, B18]
- Range queries continue to improve with very large nodes

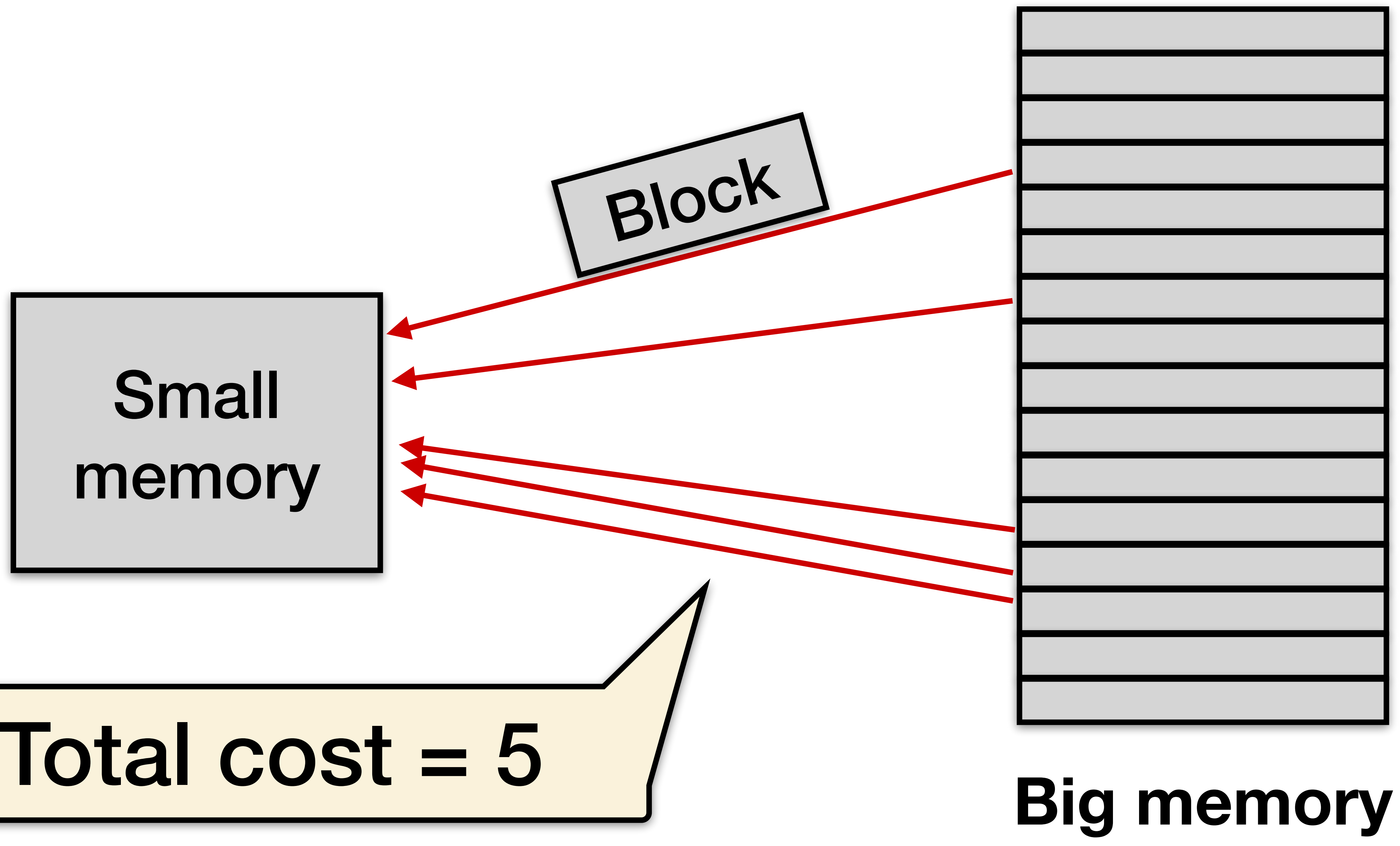


Recall: Cost of access in Disk-Access Model (DAM)

Similar to Ideal-Cache model, without tall-cache assumption

The DAM [Aggarwal and Vitter, '88] is a classical model that measures disk page access (or cache-line accesses, in RAM).

Each memory block fetch has **unit cost**.



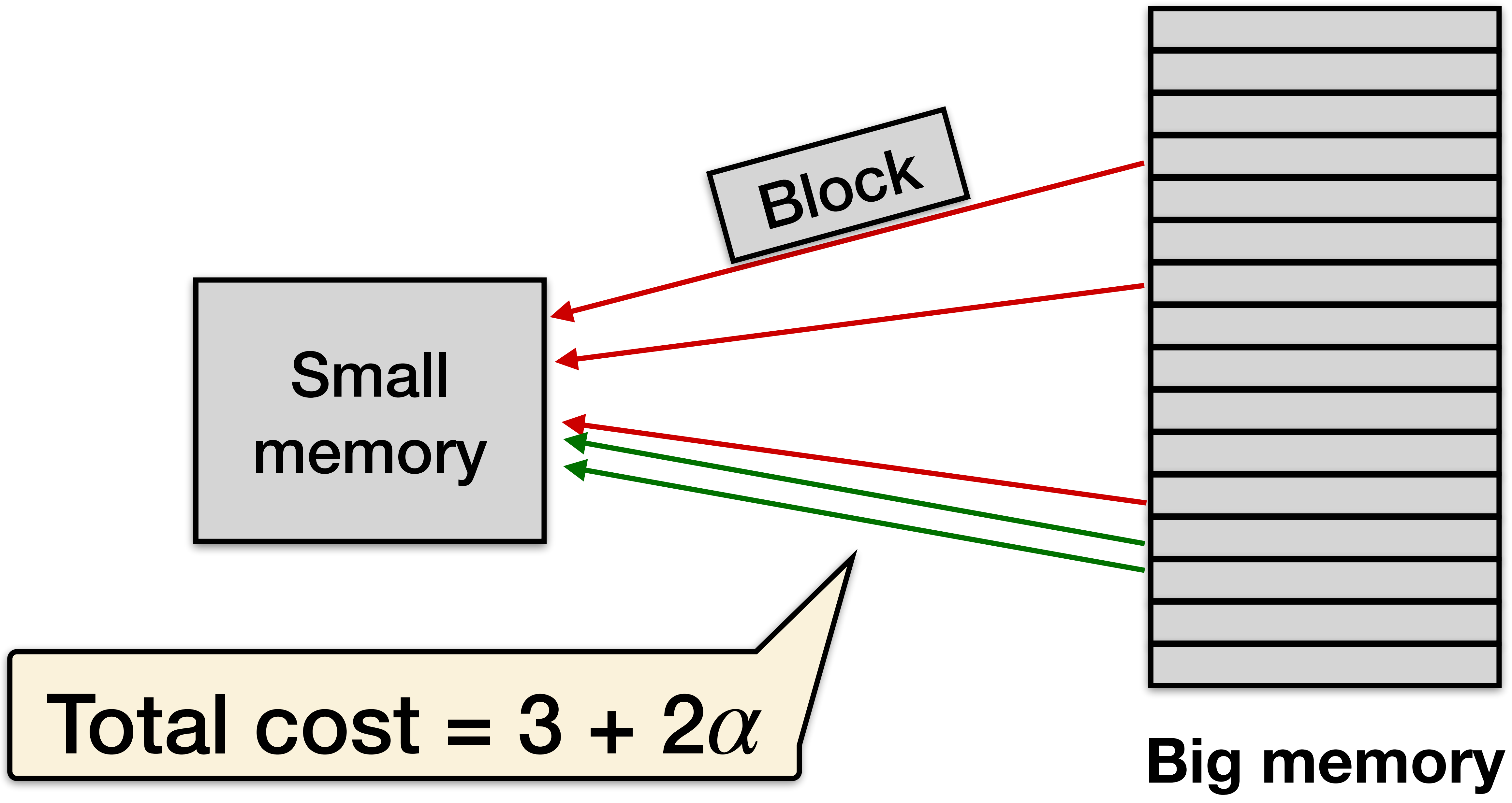
Total cost = 5

Recall: Random vs Sequential Access Cost in the Affine Model

The affine model [ABZ96, BCF+19] accounts for sequential block accesses being faster than random (due to prefetching, etc.).

Random access has unit cost, and **sequential access has cost $\alpha < 1$**.

Originally designed for disks and accounted for disk seek vs read.



Finding the empirical parameters with the scan test

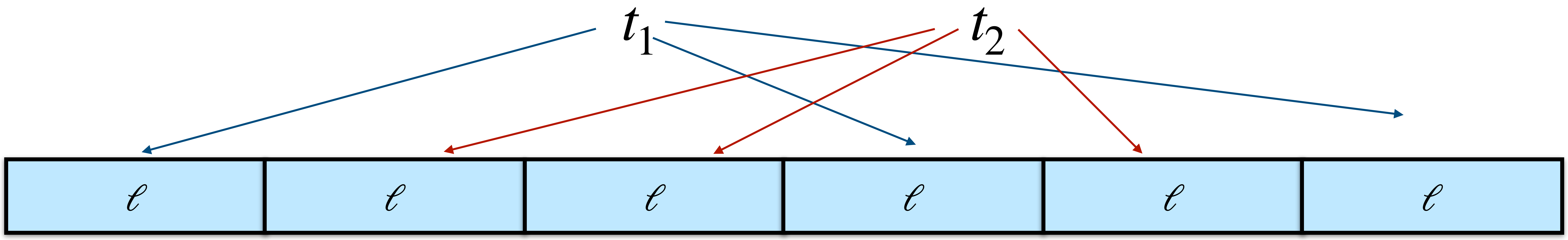
We perform the following **scan test** to empirically derive α:

Allocate a contiguous array of X bytes (X is large, in the GB range)

for block size ℓ from 1 to X in powers of 2:

Measure time as a function of varying block size

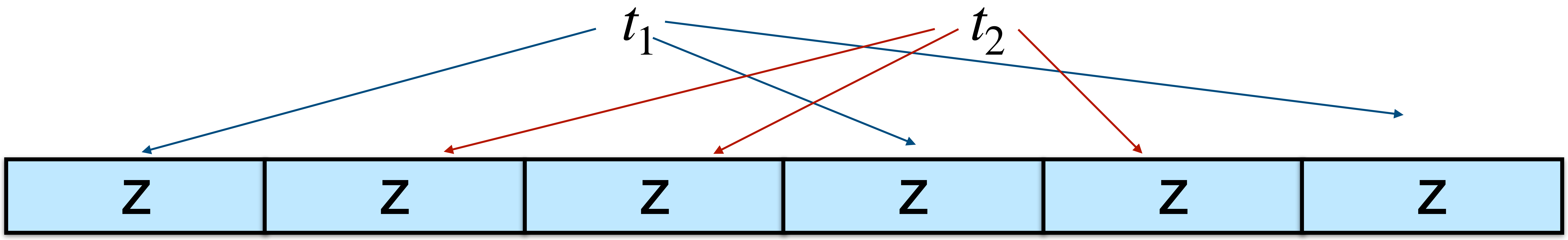
in **parallel and in random order,** scan over the entire array in separate blocks of size B



Finding the empirical parameters with the scan test

We also need to find r, the cost of reading a random location in DRAM, and w, the cost of writing to a random location in DRAM.

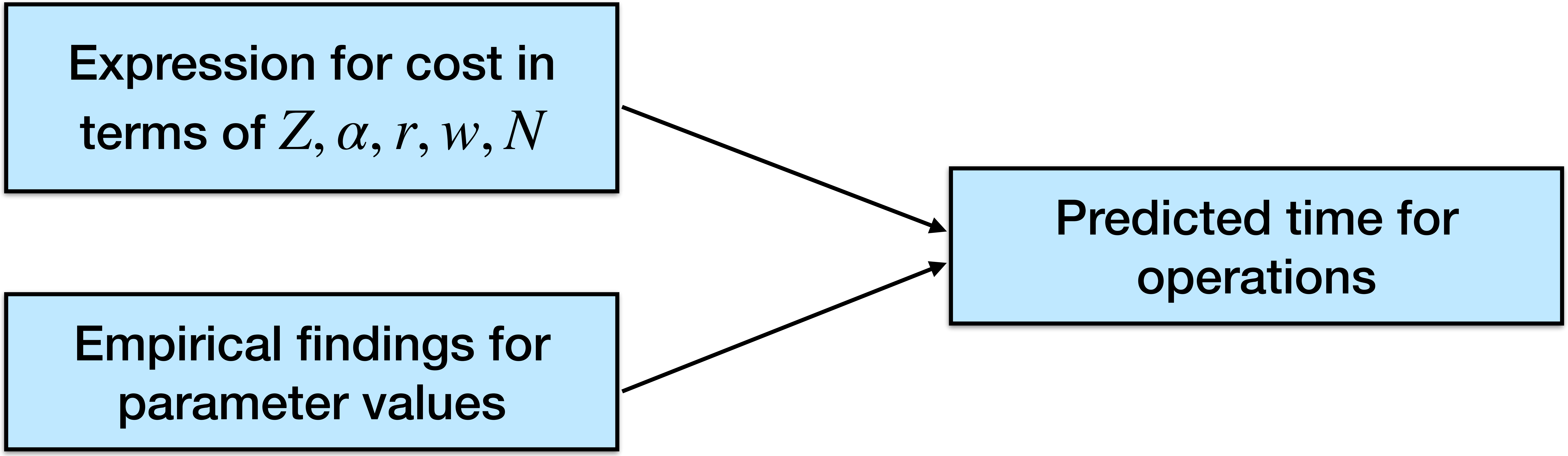
By setting $\ell = Z$ (the cache-line size), we can compute the latency of reading a random cache line in DRAM by **dividing the total time by the number of lines read**.



Results of scan test

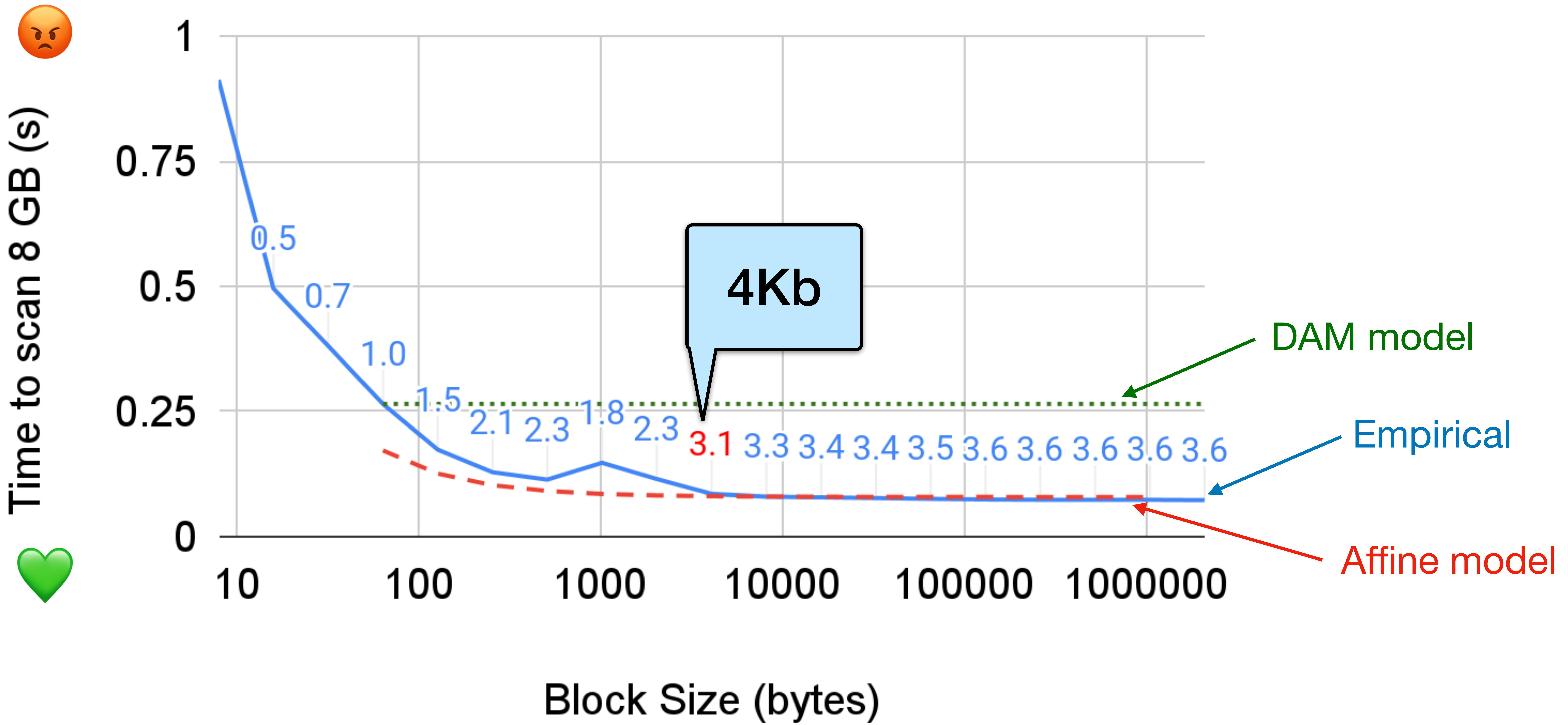
We found the following:

- $\alpha = 0.3$
- $r = 1.95$ ns
- The machine has a cache line size $Z = 64$ bytes, and we use the heuristic of $w = 2r$.



Empirically validating the affine model in memory

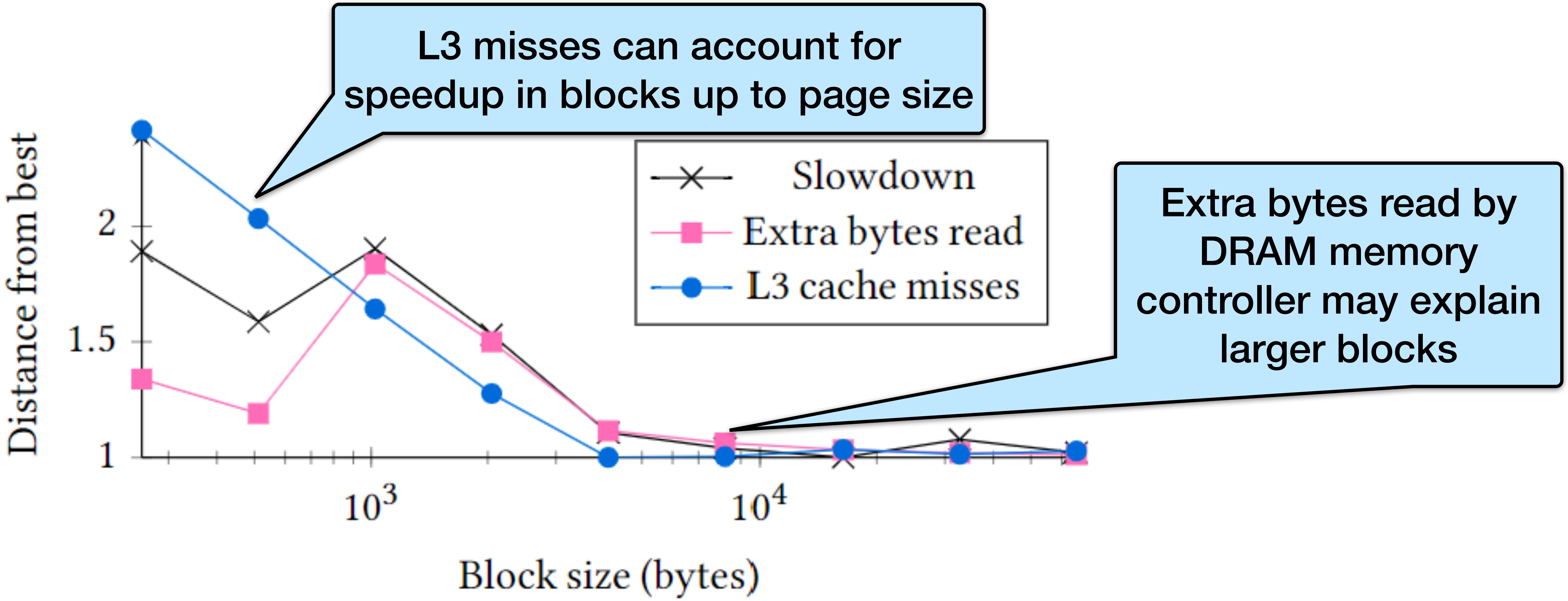
- We find the affine model also holds true for RAM using the scan test.
- Interestingly, it continues to hold even when the block goes past 1 page (4Kb) - more on that later



Why does scan performance continue to improve after page sizes?

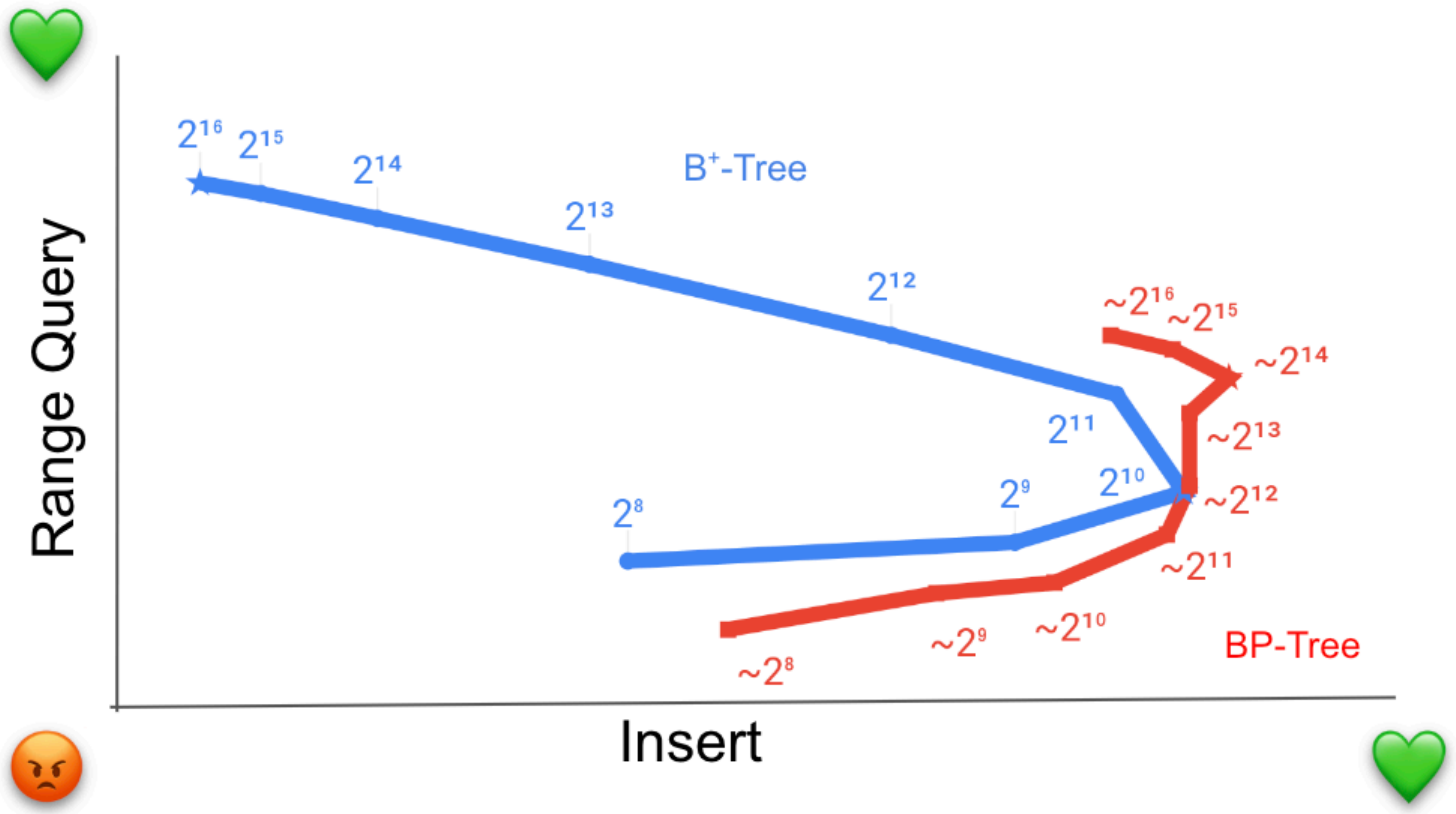
Although the cache-line prefetcher **does not cross page boundaries** [Intel manual], we continue to see performance improvements after 4Kb block reads.

To try to understand why, we used the Intel Performance Counter Monitor (PCM) to measure the extra bytes read by the memory controller and the L3 cache misses:



Summary

- B-trees (and any other blocked data structure, e.g., B-skip lists) exhibit a tradeoff between point and range operations depending on the node size.
- The affine model provides a way to analytically determine the benefits of larger node sizes during scans.
- BP-tree overcomes the decades-old point range tradeoff in B-Trees: it can increase the performance for workloads that include **both point operations and long scans**.



BACKUP PAST HERE

Optimal Search-Insert Tradeoff [Brodal, Fagerberg 03]

	insert	point query
Optimal tradeoff (function of ε=0...1)	$O\left(\frac{\log_{1+B^\varepsilon} N}{B^{1-\varepsilon}}\right)$	$O\left(\log_{1+B^\varepsilon} N\right)$
B-tree (ε=1)	$O\left(\log_B N\right)$	$O\left(\log_B N\right)$
ε=1/2	$O\left(\frac{\log_B N}{\sqrt{B}}\right)$	$O\left(\log_B N\right)$
ε=0	$O\left(\frac{\log N}{B}\right)$	$O\left(\log N\right)$

10x-100x faster inserts

Slides from: https://www3.cs.stonybrook.edu/~bender/talks/2012-Bender-Dagstuhl-write-optimized-talk.pdf

BP-tree system setup

- 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

BP-tree raw point data

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

BP-tree raw range data

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

BP-tree YCSB raw data

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

BP-tree on Zipfian

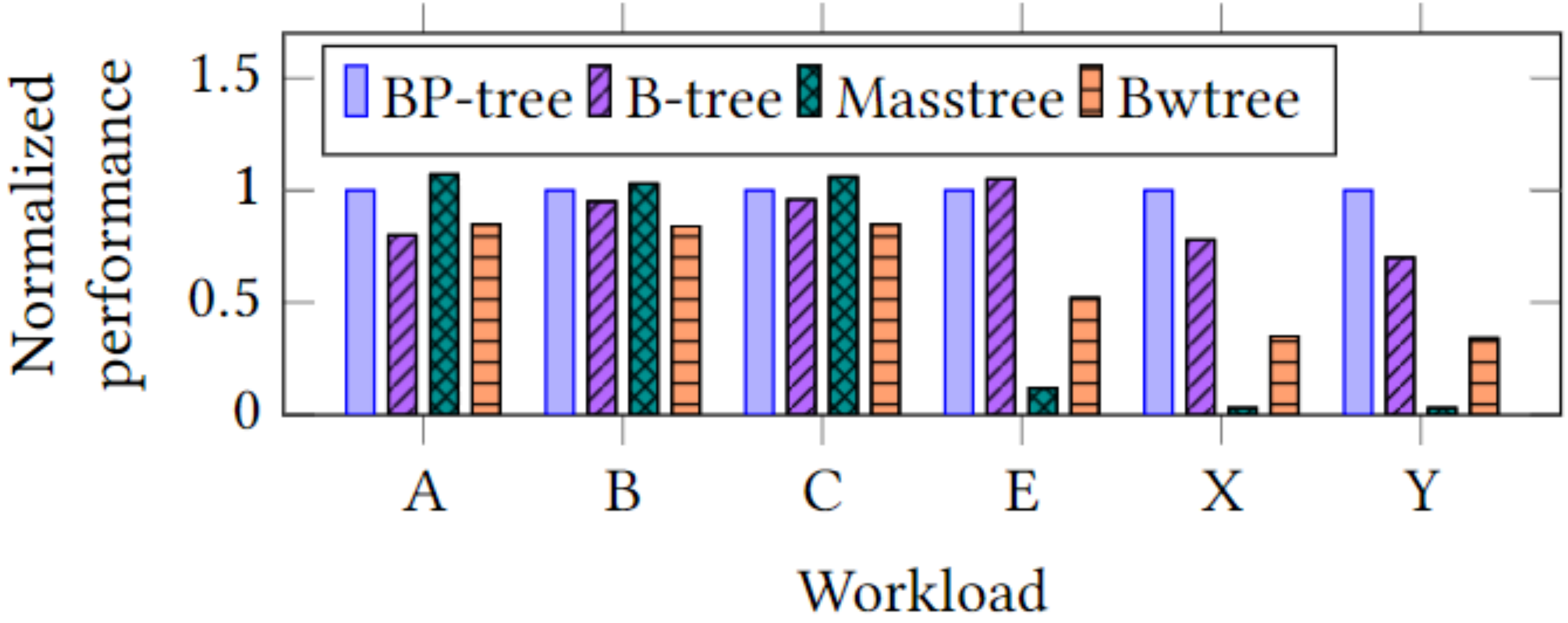


Figure 9: Relative performance compared to the BP-tree on zipfian workloads generated from YCSB.