# 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 (Berkeley Lab), Amanda Li (MIT), Brian Wheatman (Johns Hopkins), Manoj Marneni (Utah)

THE UNIVERSITY OF UTAH

# Dictionary

Queries

- membership
- predecessor/successor
- range queries

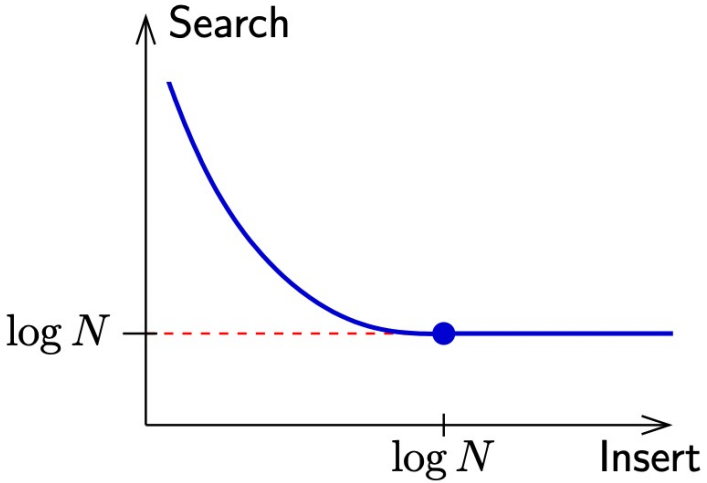
Updates

- insertions
- deletions

# Dictionary --- comparison based [Brodal and Fagerberg 2003]

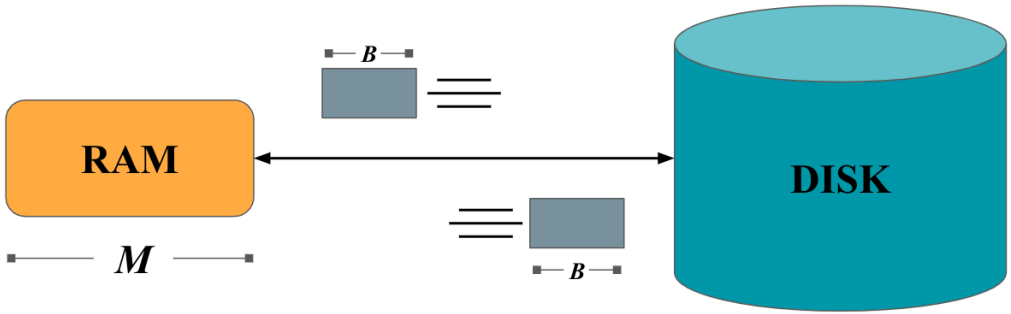
Balanced search trees

- Insert *O(*log *N)*
- Queries *O(*log *N)*



# External memory model [Aggarwal and Vitter 1988]

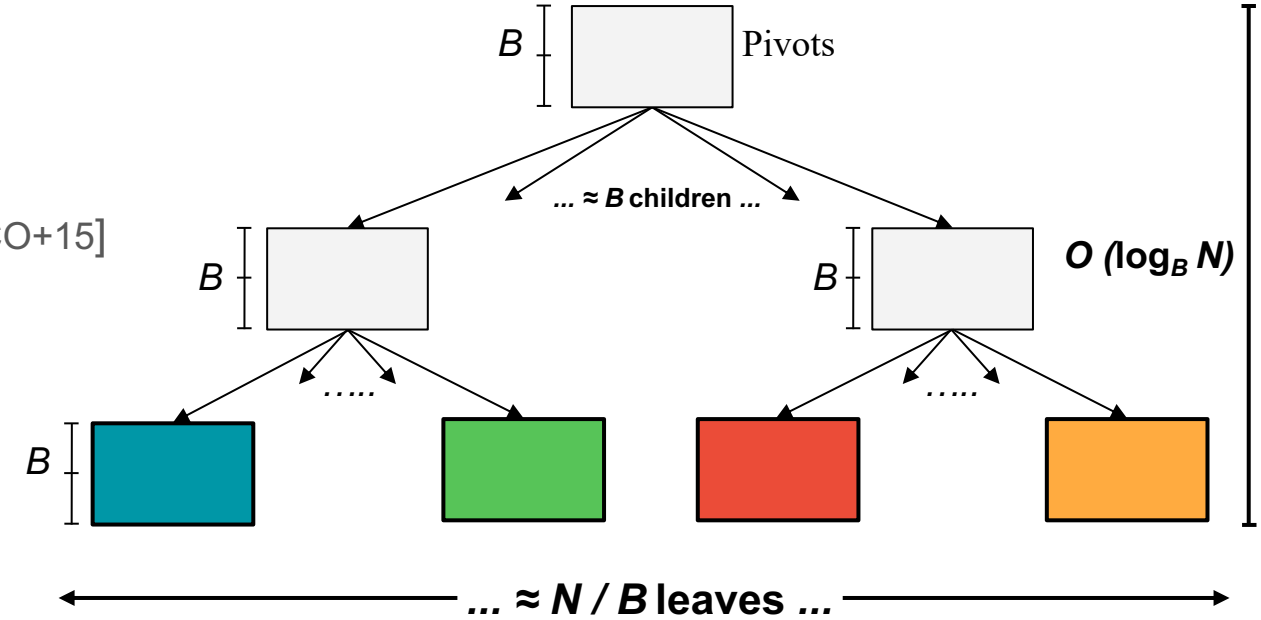
- **How computations work:**
  - Data is transferred in blocks between RAM and disk.
  - The number of block transfers dominate the running time.
- **Goal: Minimize number of block transfers**
  - Performance bounds are parameterized by block size $B$, memory size $M$, data size $N$.



# B-tree: a classic indexing data structure

B/B+ Trees [BM72] are used everywhere

- In memory indexing [ZCO+15]
- Databases [K98]
- Filesystems [RBM13]

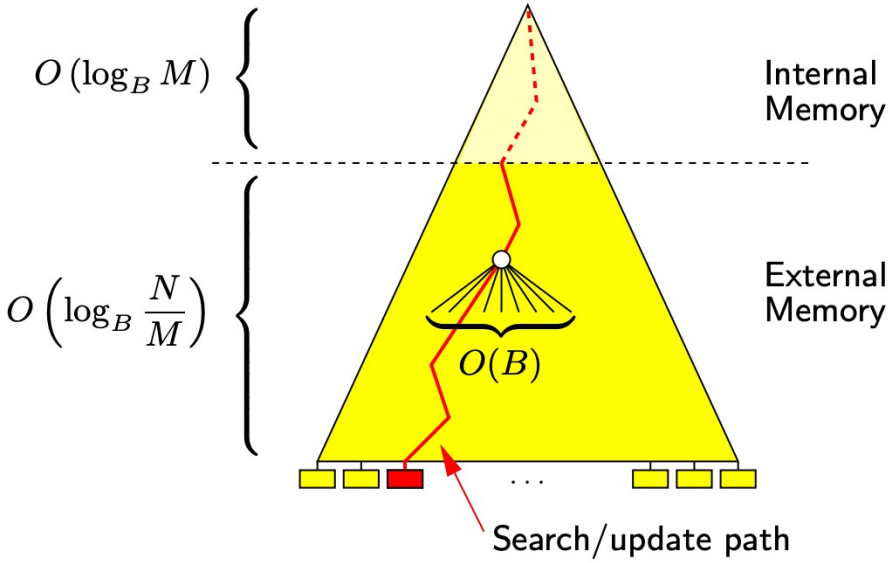



**B-Trees are asymptotically optimal for point operations**

# B-tree: in external memory model [Brodal and Fagerberg 2003]

B/B+ Trees [BM72] are used everywhere

- In memory indexing [ZCO+15]
- Databases [K98]
- Filesystems [RBM13]

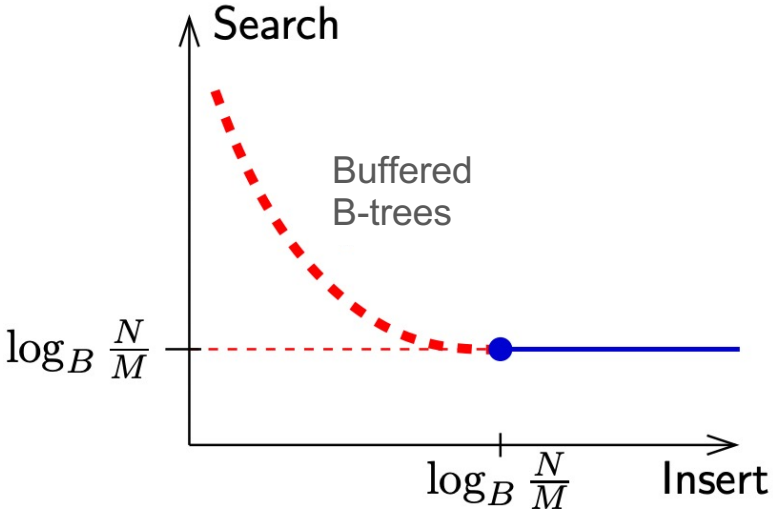



Insert
Search
} $O\left(\log_B \frac{N}{M}\right)$ I/Os

# B-tree: in external memory model [Brodal and Fagerberg 2003]

B/B$^+$ Trees [BM72] are used everywhere

- In memory indexing [ZCO+15]
- Databases [K98]
- Filesystems [RBM13]




Insert
Search
} $O\left(\log_B \frac{N}{M}\right)$ I/Os

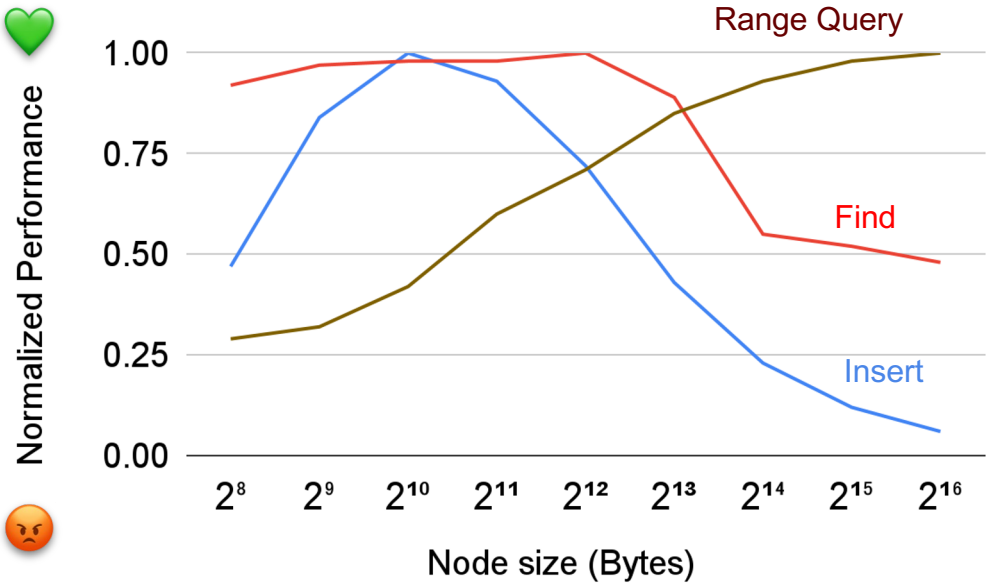
# In this talk: point-range tradeoff in in-memory B-trees

# Motivation: B-tree point-range tradeoff

They exhibit a tradeoff between point operations (updates/queries) (OLTP) and long range scans (OLAP) as a function of node size

Long range scans are critical for

- Real time analytics [PTPH12]
- Graph processing [DBGS22, PGK21, PWXB21]

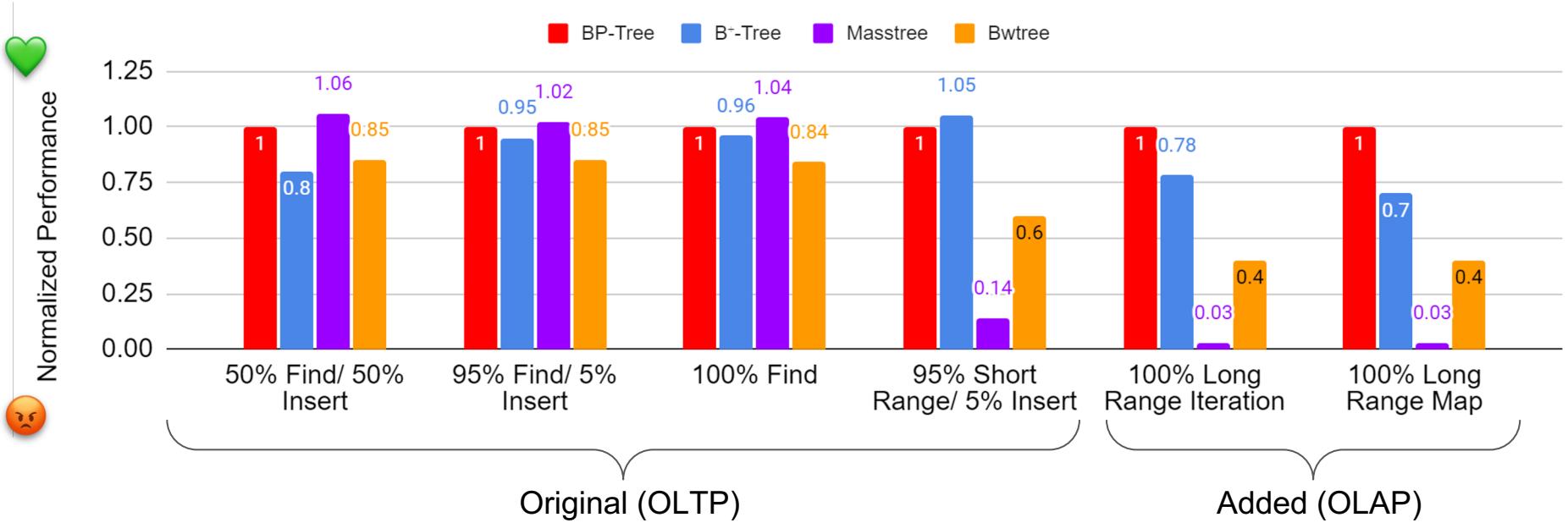



**Large nodes speed up range scans at the cost of point inserts**

We introduce BP-tree to overcome the traditional tradeoff between point operations and range scans in in-memory B-trees.

# Evaluation on YCSB benchmarks

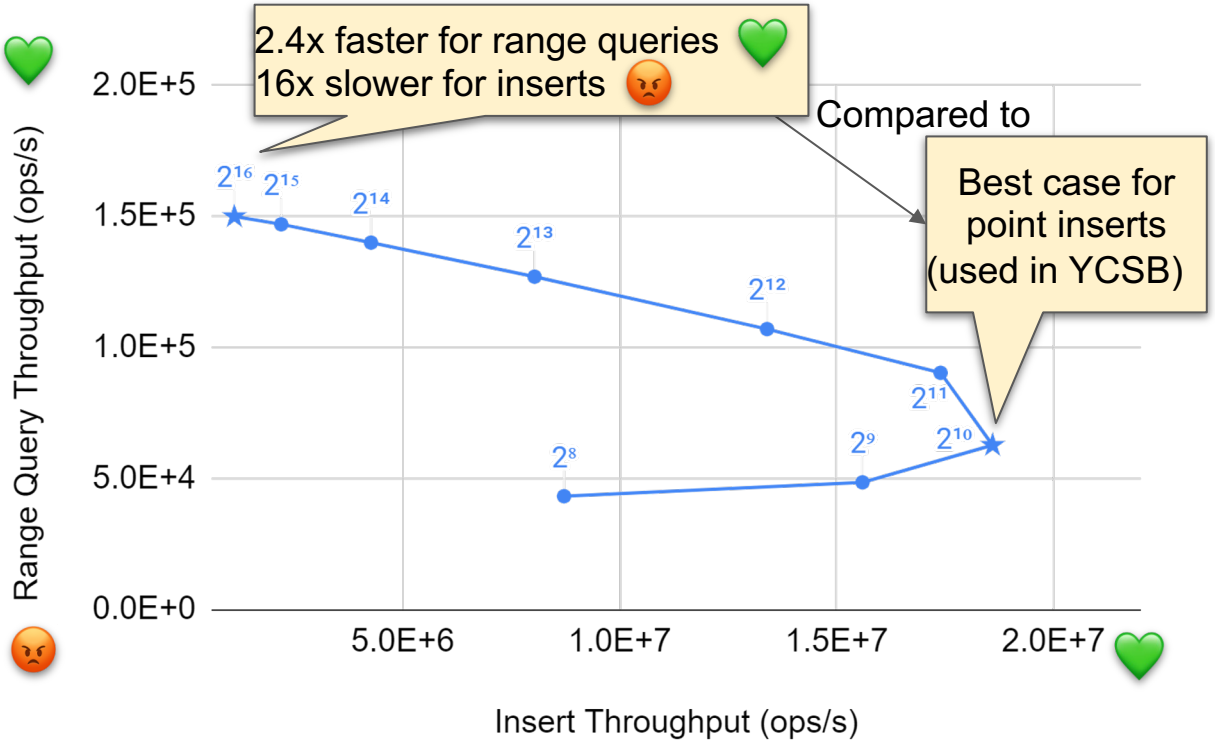
Performance of B-tree, Masstree [MKM2012], OpenBwTree [WPL+2018] and BP-tree on YCSB [CST+10]




**We match the performance of point operations and improve range queries by 2x**

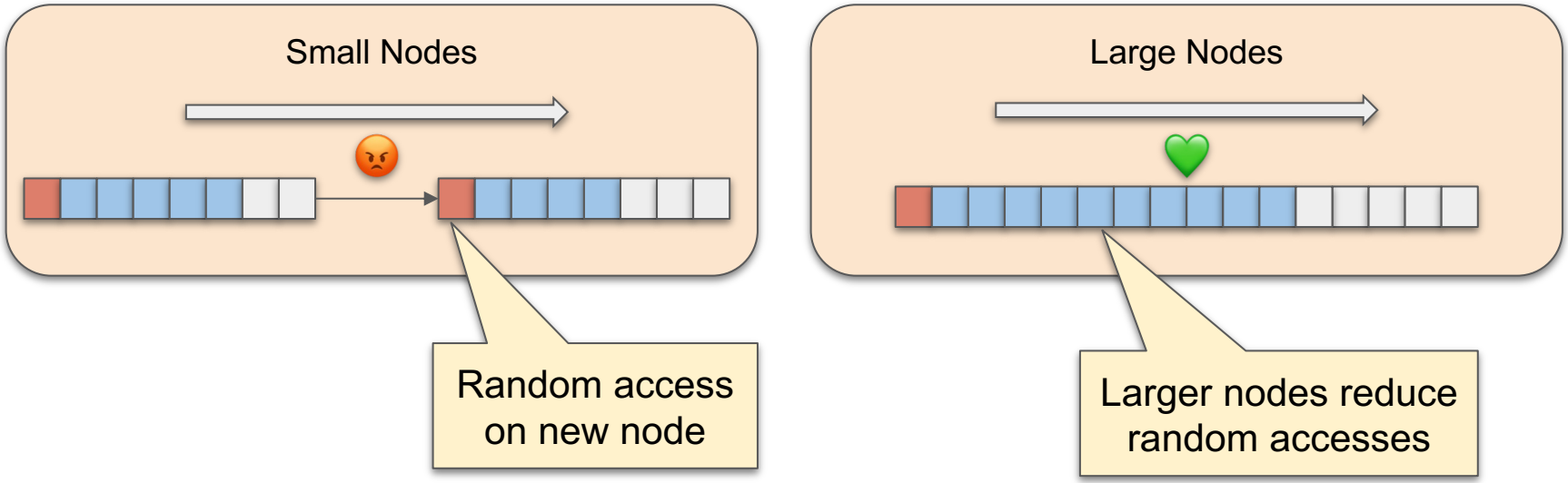
# B-tree insert/range scan trade-off

Large node sizes improve range scan throughput, but slow down inserts



# Larger nodes improve range query performance

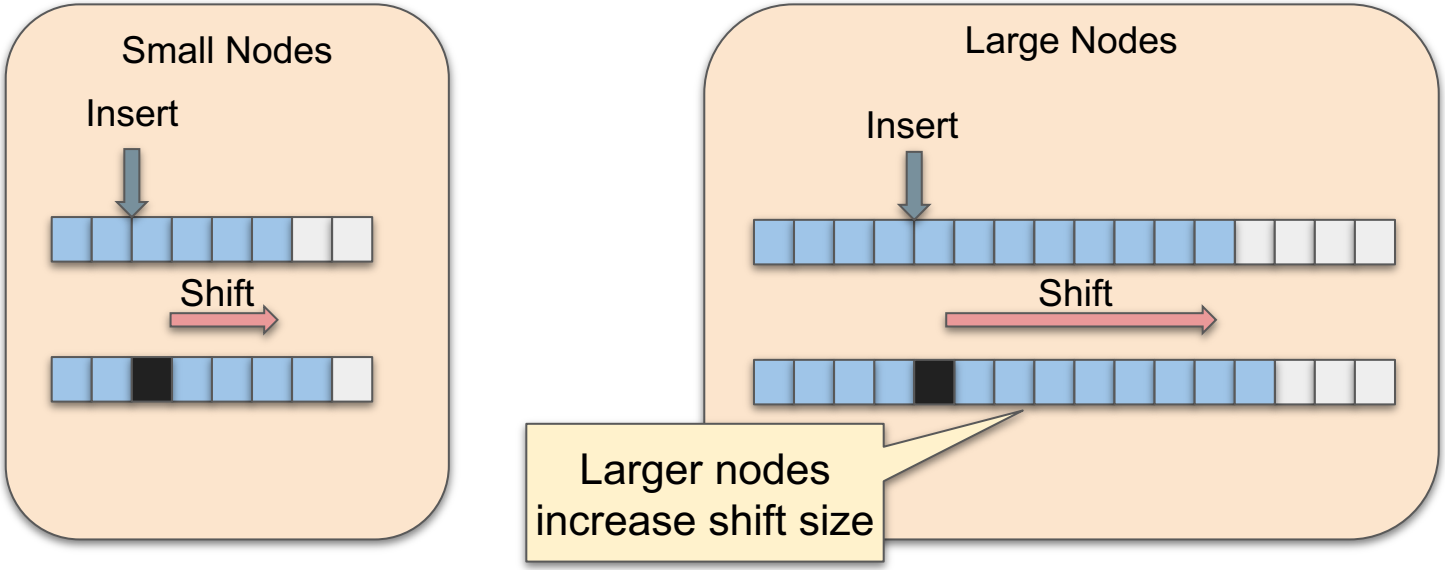
Increasing the size of nodes decreases the number of nodes accessed during long range queries and thus the number of random memory accesses



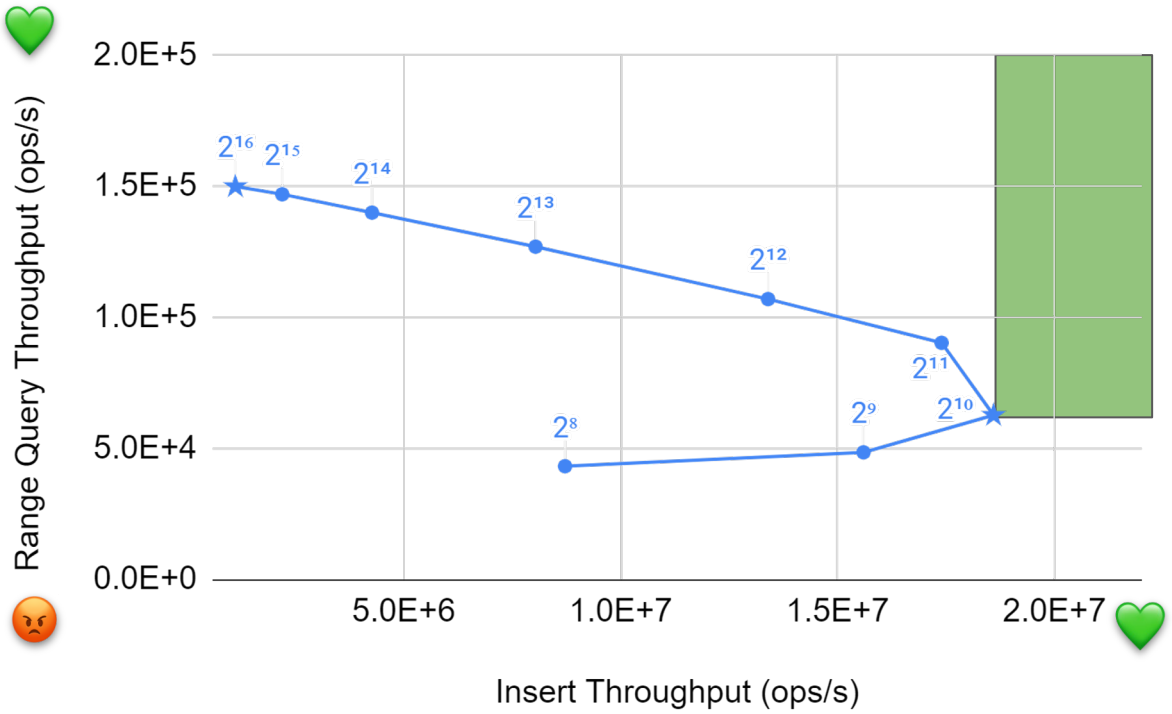
# But larger nodes require more shifting on every insert

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

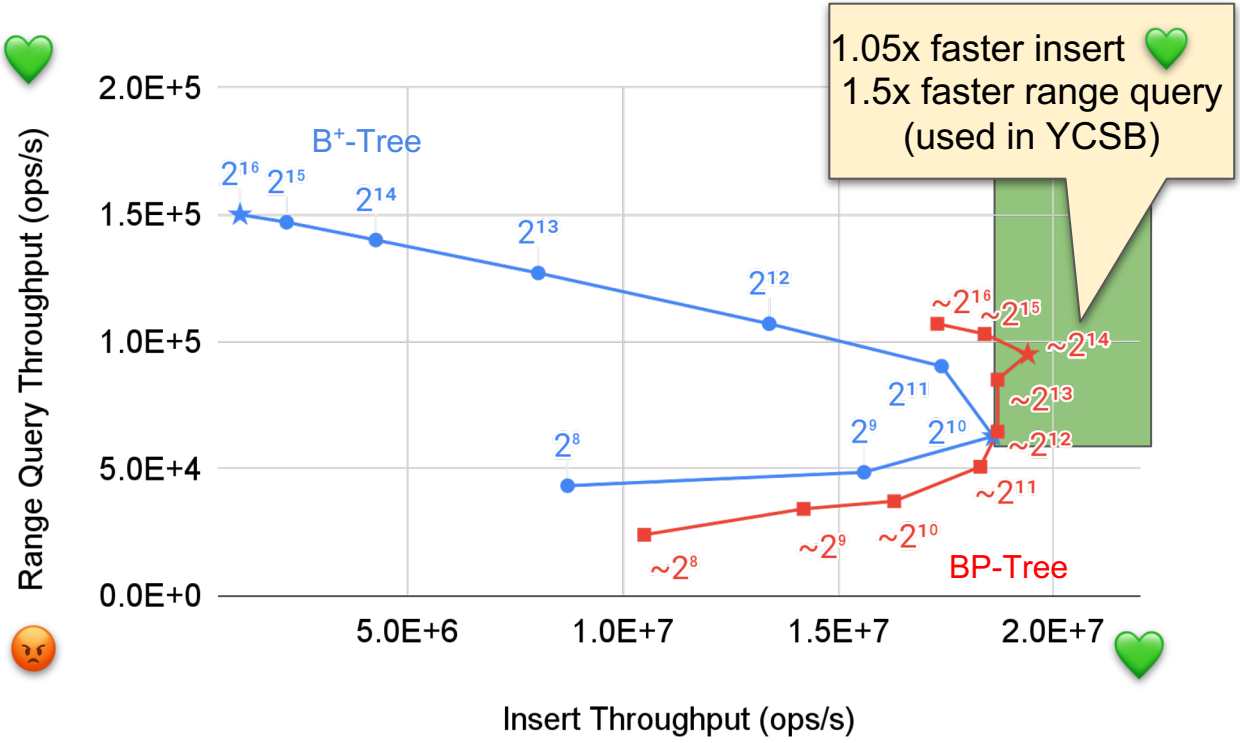


# B-tree insert/range query trade-off



# The BP-tree can improve both insert and range queries

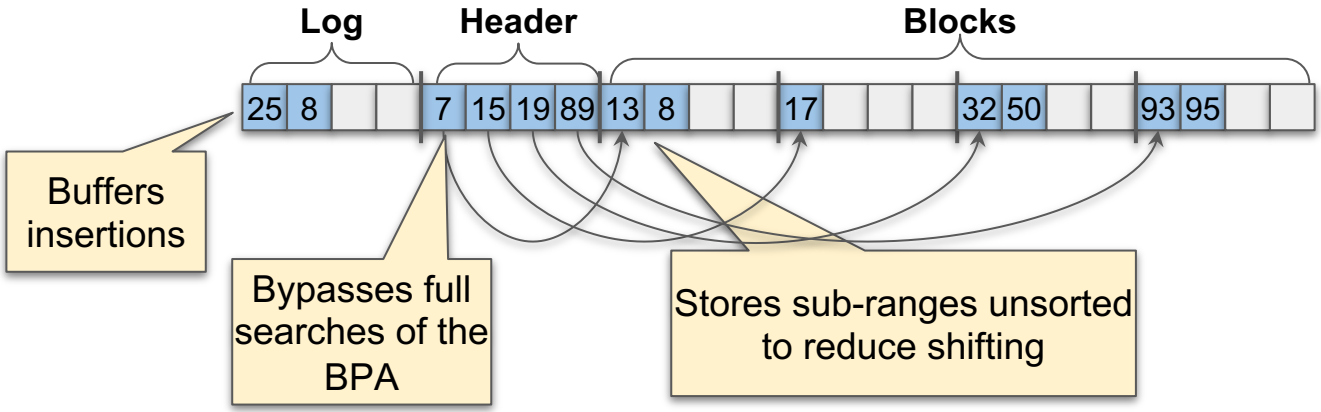
The BP-tree can improve long range operations without sacrificing point operations



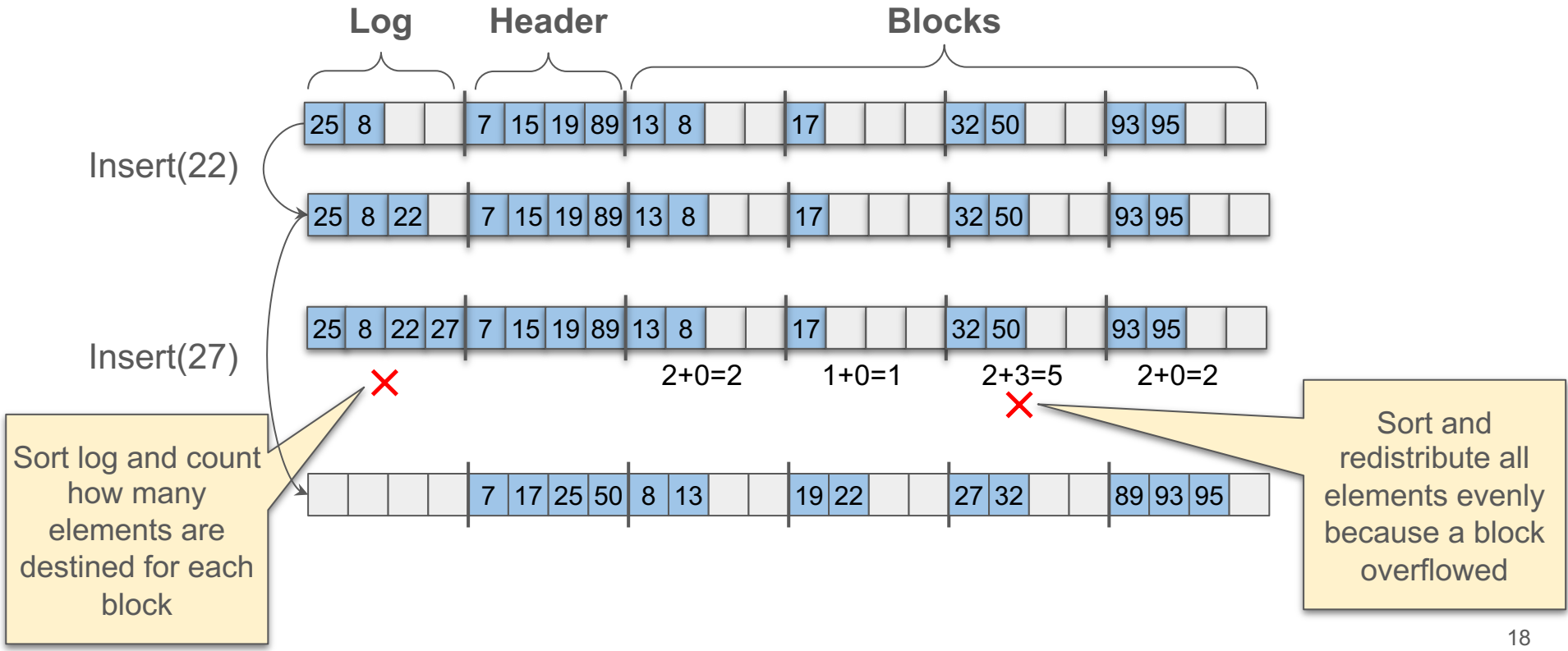
# Buffered Partitioned Array (BPA) design

The BP-tree overcomes the point-range tradeoff by using large nodes with an insert-optimized data structure in the leaves called the Buffered Partitioned Array (BPA)

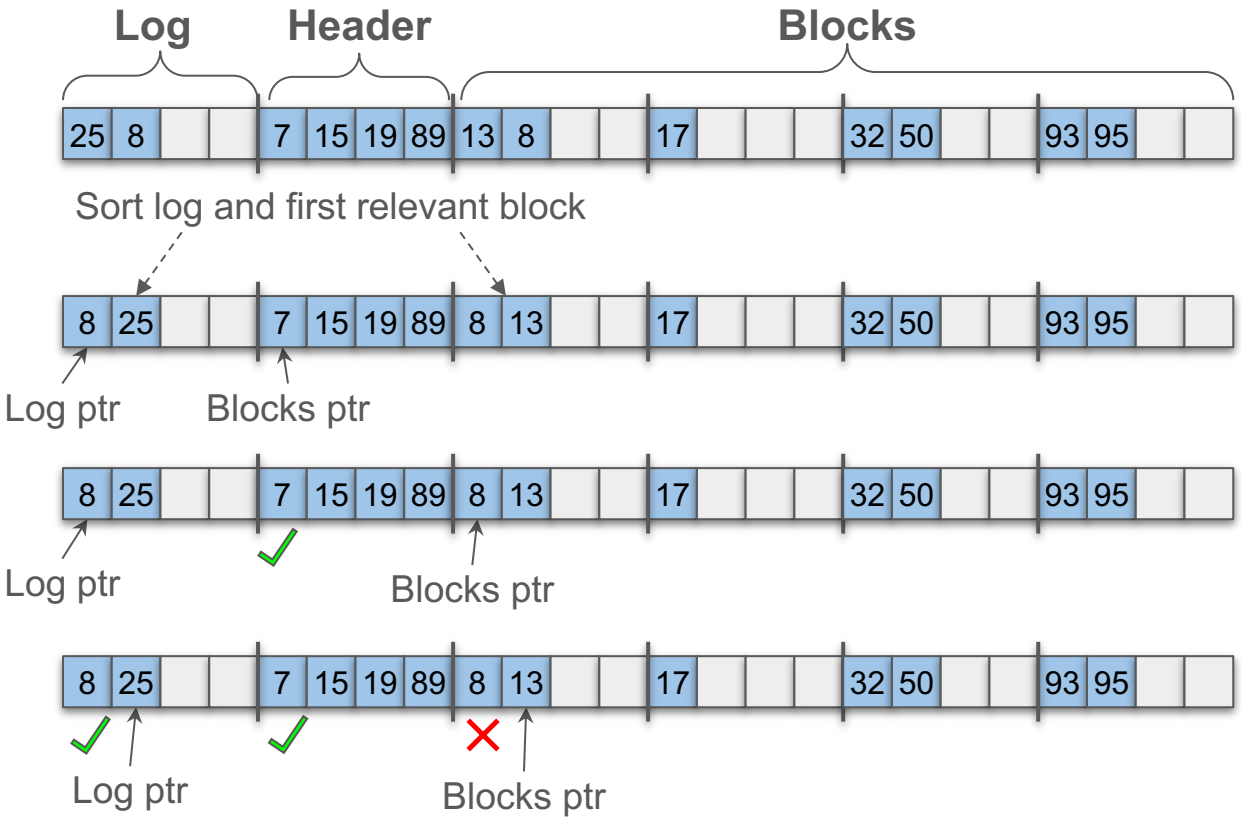
One way to think about the BPA is like collapsing the last two levels of a B-tree into one insert-optimized array-like data structure



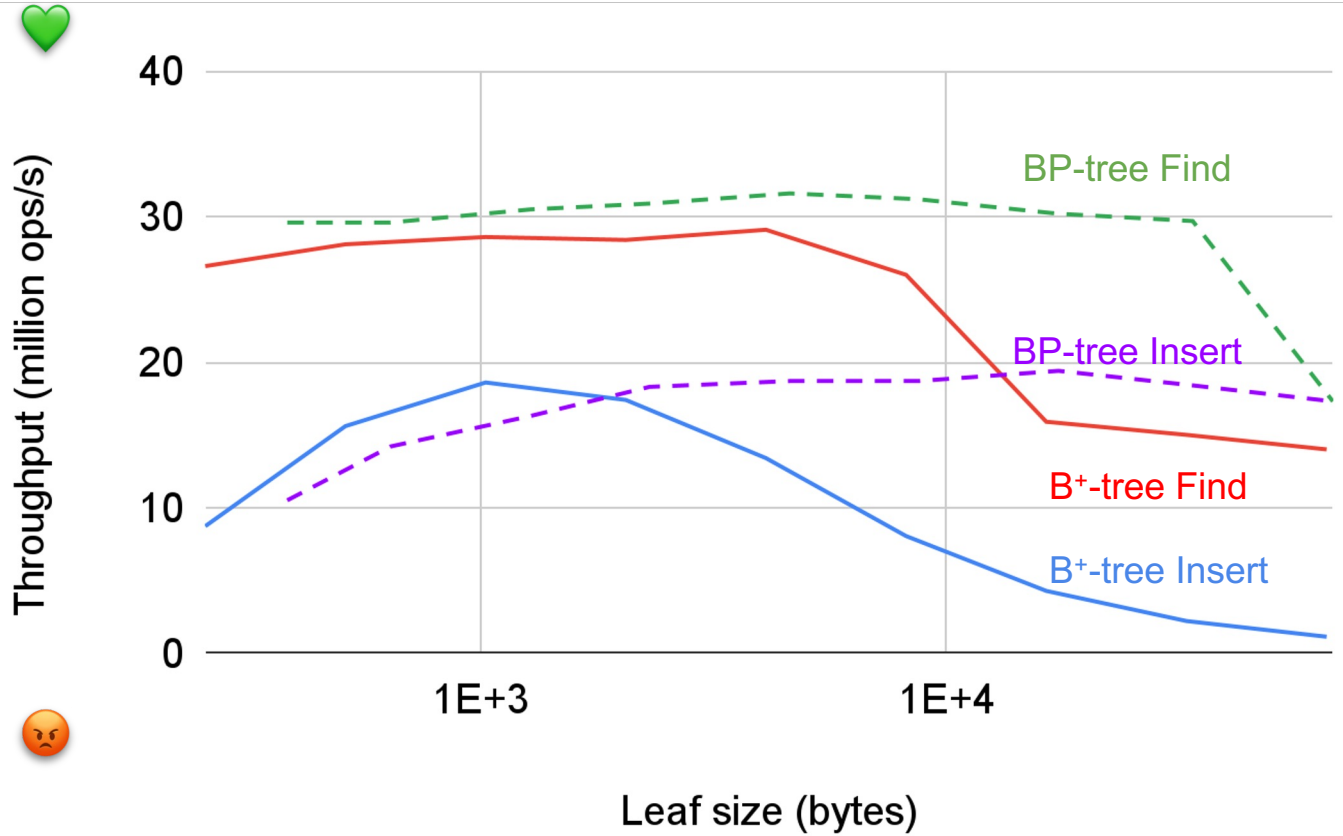
# Example insertions in a BPA



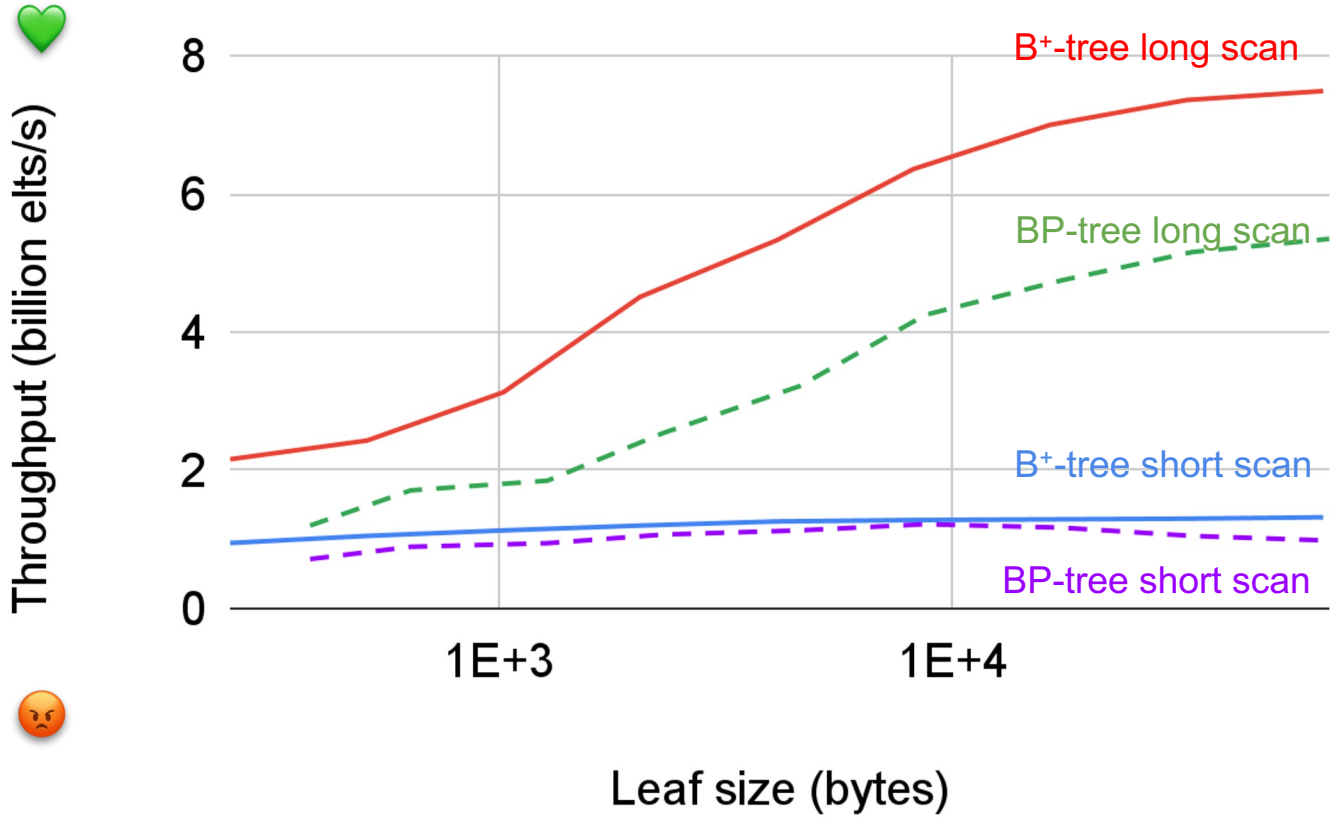
# Example range query: iterate_range(start = 7, length = 2, f)



# B-tree vs BP-tree point operations



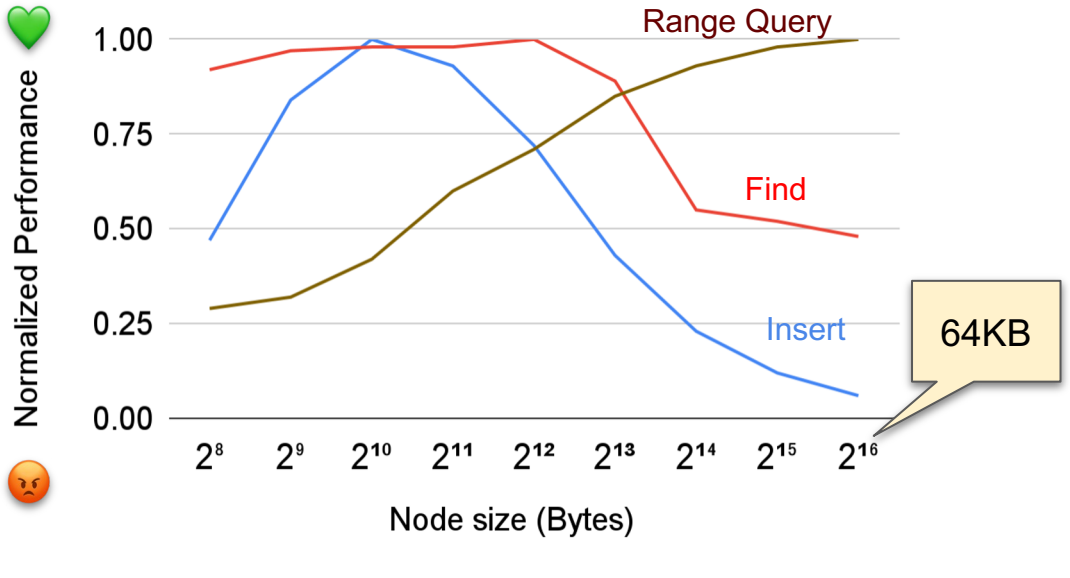
# B-tree vs BP-tree range queries



# To what extent do big nodes help range queries?

Traditionally node sizes are small (up to 256 bytes) [CGM01, HP03, B18]

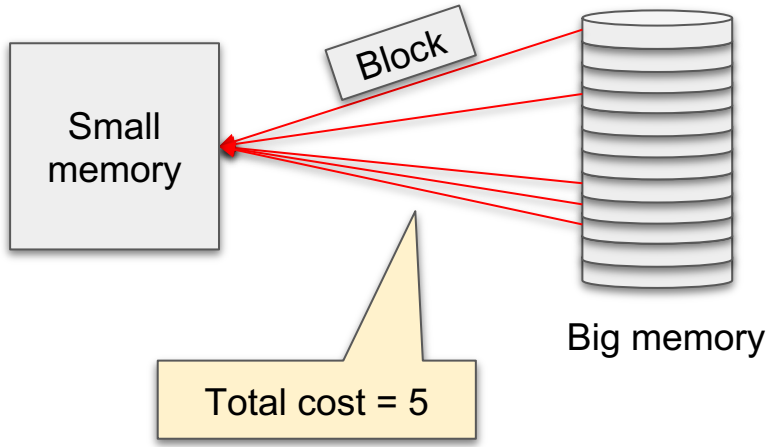
Range queries continue to improve with very large nodes



# Costs of memory access in the Disk-Access Model

DAM [AV88] is a classical model that measures disk page accesses (or cache-line accesses, in RAM)

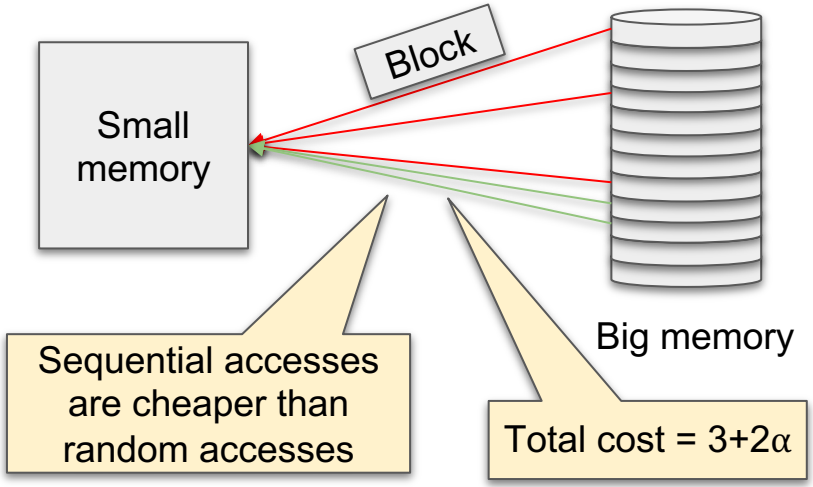
Each memory block fetch has unit cost.



# Random vs sequential access cost in the affine model

The affine model [ABZ96,BCF+19] accounts for sequential block accesses being faster than random

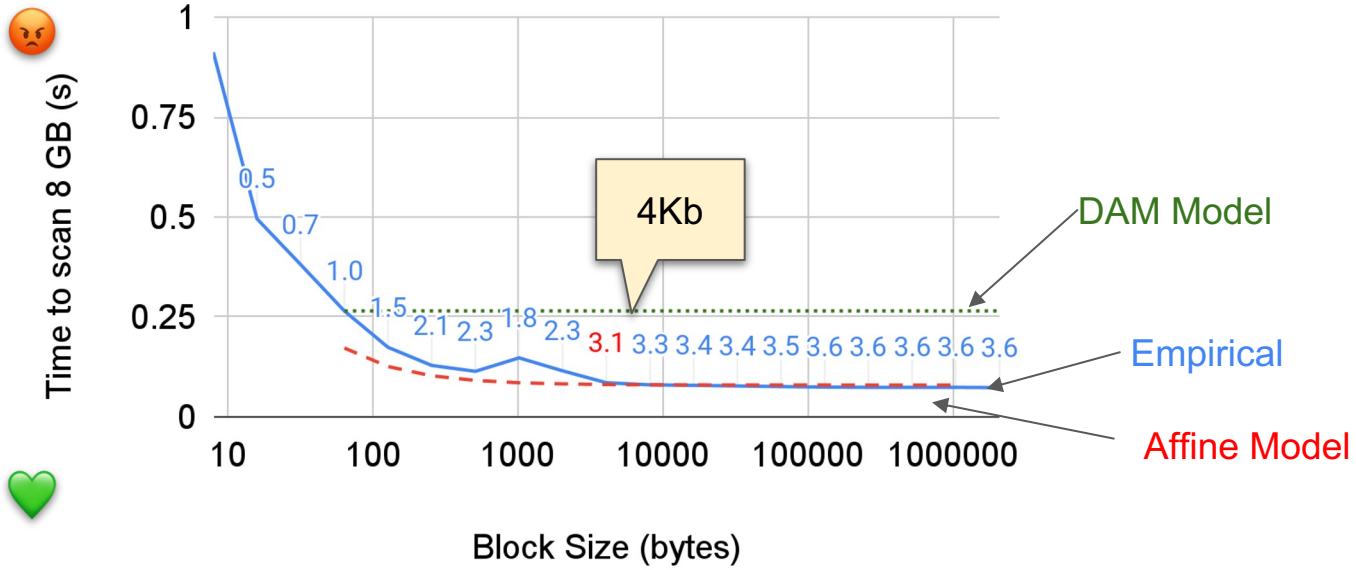
Originally designed for disks and accounted for disk seek vs read



# Empirically verifying the affine model in memory

We find the affine model also holds true for RAM

Continues to hold even when the block size goes past 1 page (4Kb)

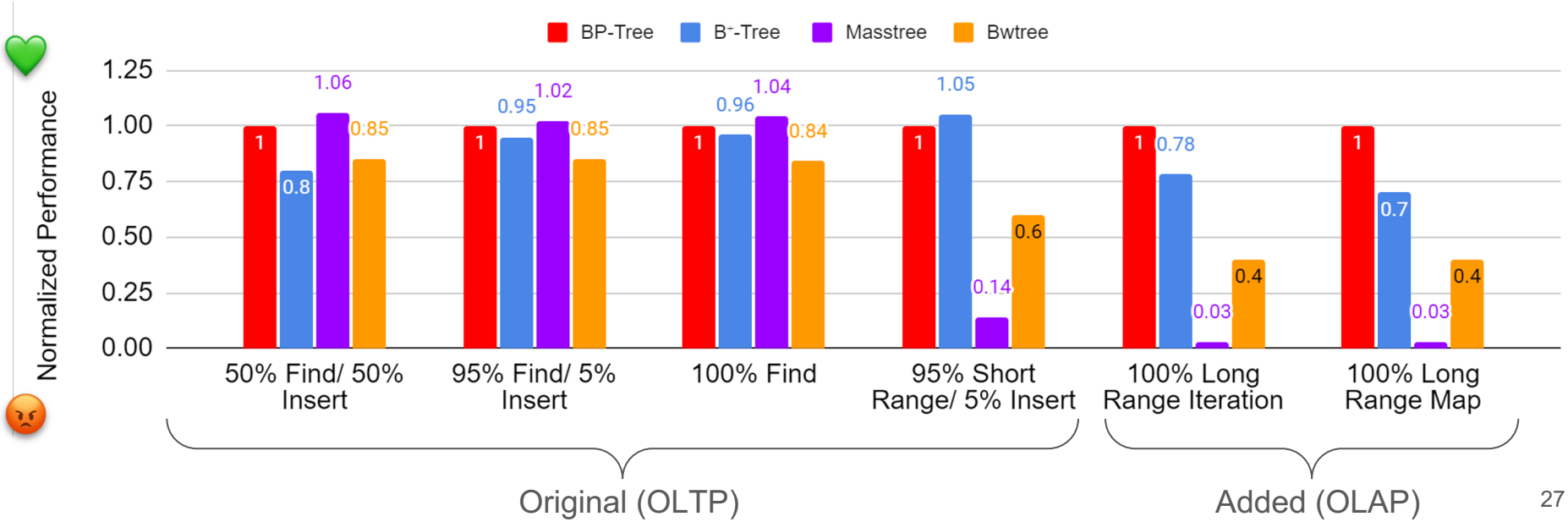


# System Setup

- 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

# Evaluation on YCSB benchmarks

Relative performance of reference B-tree, Masstree [MKM2012], OpenBwTree [WPL+2018] compared to the BP-tree on uniform random workloads generated from YCSB [CST+10]
We added workloads to test long scans (both iteration and map).



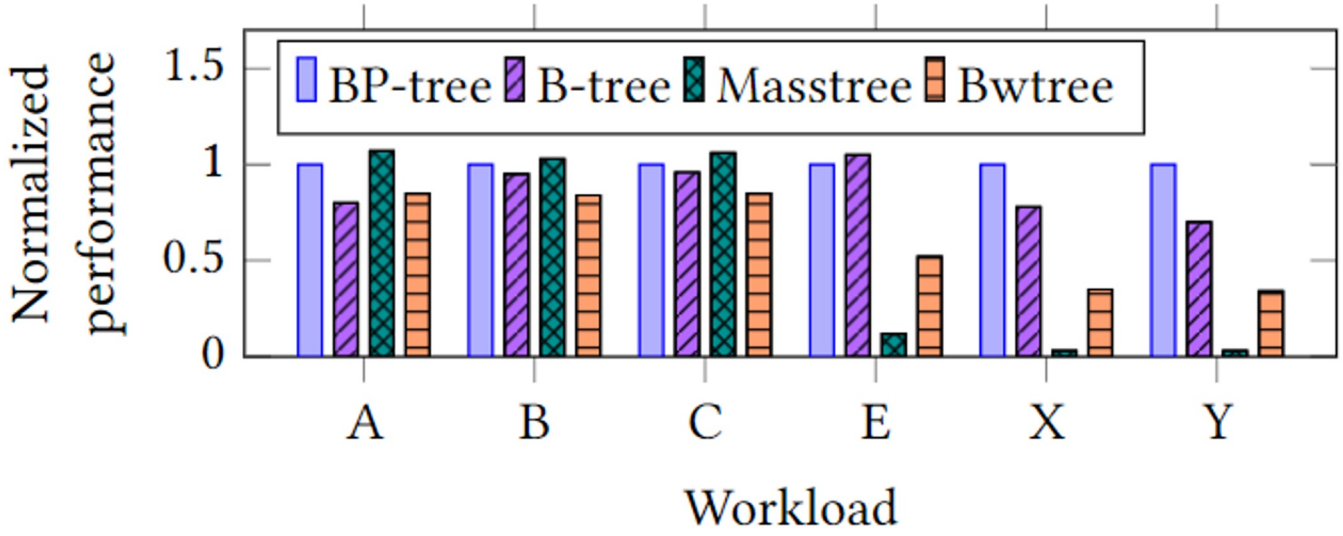



**Figure 9: Relative performance compared to the BP-tree on zipfian workloads generated from YCSB.**

# Conclusion

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.

I/O models (External memory and Affine) apply to in-memory data structures

Relaxing ordering constraint in B-tree nodes can help achieve overcome tradeoffs

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

**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 |

# Why does it work

Figure 4 from paper

Continuous reads

Even bigger than a page

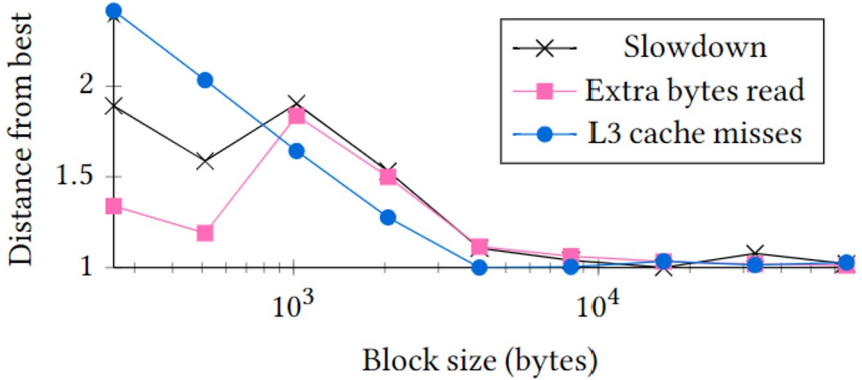



**Figure 4: Relative slowdown (time), L3 cache misses, and bytes read during the scan test.**