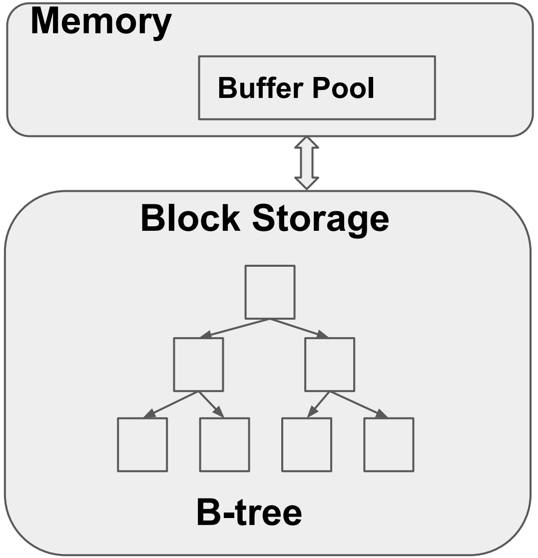
Two is Better Than One: The Case for 2-Tree for Skewed Data Set

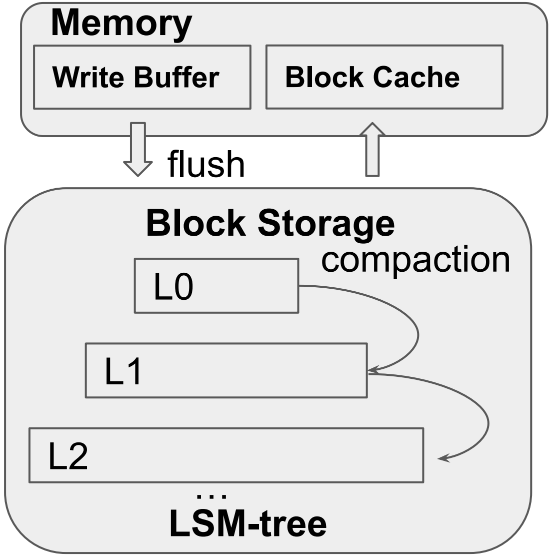
Xinjing Zhou, Xiangyao Yu, Goetz Graefe, Michael Stonebraker

MIT University of Wisconsin-Madison Google MIT

Background

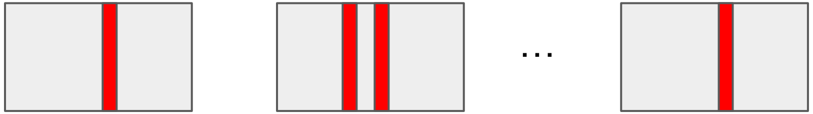
- Tree data structures in data systems
  - Access method and storage
  - Buffer-managed





Buffer Pool Utilization Issue: B-tree

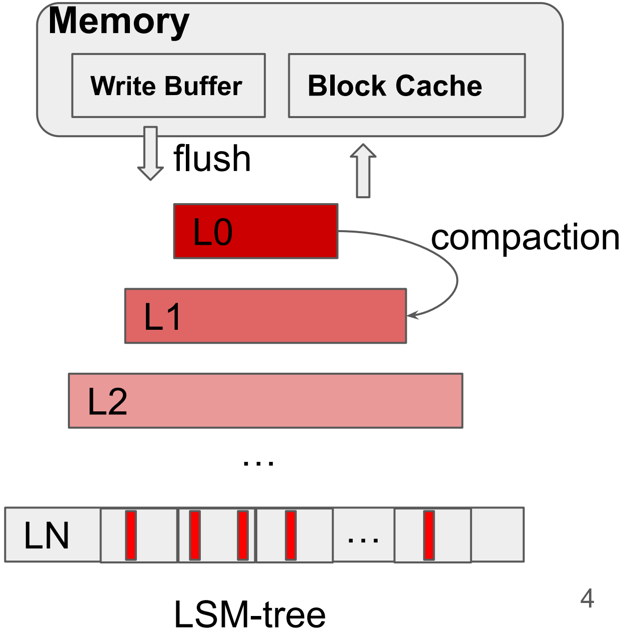
- Buffer pool memory utilization issue on **skewed workloads** for B-tree
- A few hot records in a B-tree page with many cold ones
- Records keyed on domains with no spatial locality could spread across the key space
  - e.g., user-id



B-tree node

Buffer Pool Utilization Issue: LSM-tree

- Partially mitigates the memory utilization issue
- Frequently-updated data tend to cluster around the top of tree hierarchy
  - Good buffer pool utilization on skewed reads
- Stationary data get migrated down to the bottom
  - Poor utilization on skewed reads



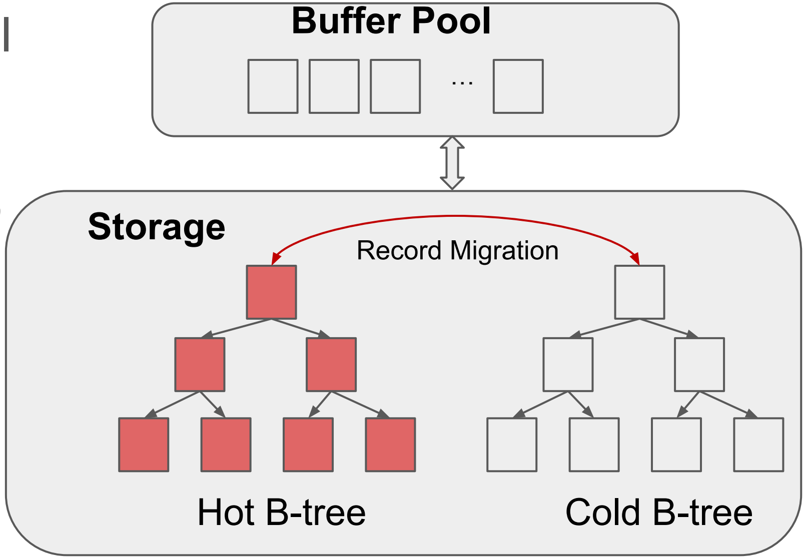
How to improve the memory utilization of buffer pool for tree structures on skewed workloads?

This Work

- **Principles**
  - Multi-structuring to physically separate hot data from cold data
  - Actively migrate data at record-level in both direction
- Applications to B-tree and LSM-tree

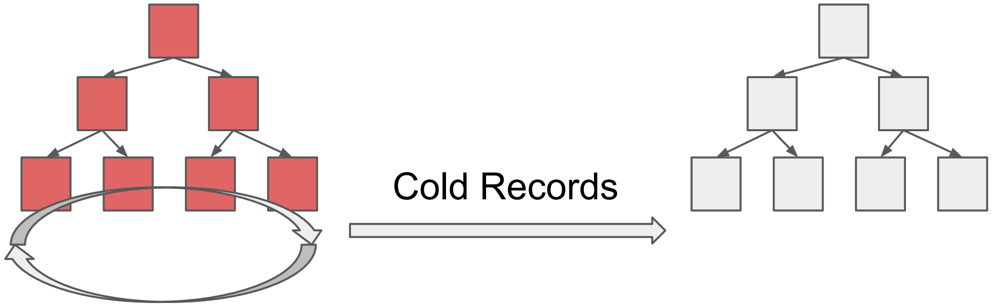
1. 2B-tree for Disk-based DBMS

- Compose two b-trees
  - Exposing single tree interface logically
- Record migration between two trees
- Size the hot tree to be close to buffer pool capacity
- Intuition: vast majority of the accesses go to the hot b-tree => increased utilization



⬇️Downward Data Migration

- Goal: evict cold records with **low overhead**
- Trigger: hot tree fills up
- Leverage efficient range scans to approximate a clock replacement



⬆️Upward Data Migration

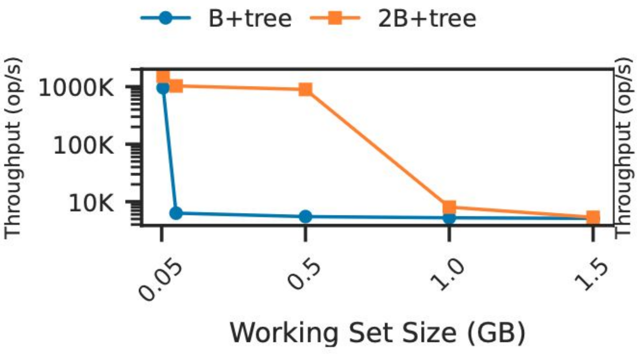
- Goal: Migrate only **warm** records upwards to the hot tree to reduce churns
- Probabilistic approach
  - Migrate a sampled set of accesses to the cold tree
  - Intuition: warm records will be more likely to be sampled

Experimental Setup

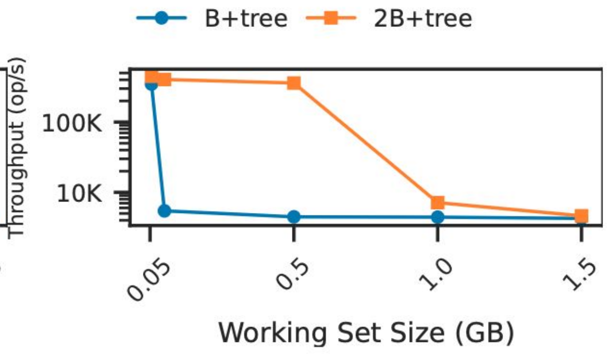
- 2B-tree implemented using LeanStore buffer pool/B-tree
- 1GB buffer pool and 5GB data set
  - 20M records, 256-byte each.
  - 16KB page size.
- Hot tree sized to be about 90% of the buffer pool capacity
- Probabilistic sampling rate: 0.5
- Workloads:
  - YCSB hotspot
    - Vary the working set size
  - YCSB zipfian
    - Working set covers all data with zipfian access distribution

Point Operations

YCSB-Hotspot

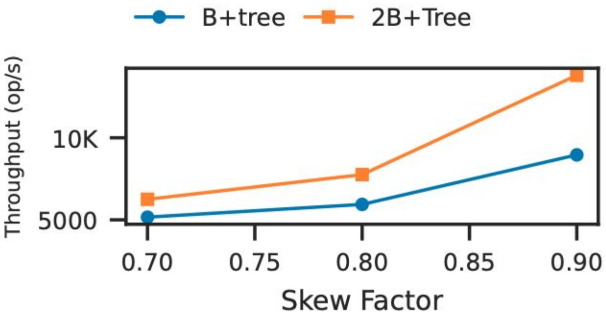


(a) Read-Only

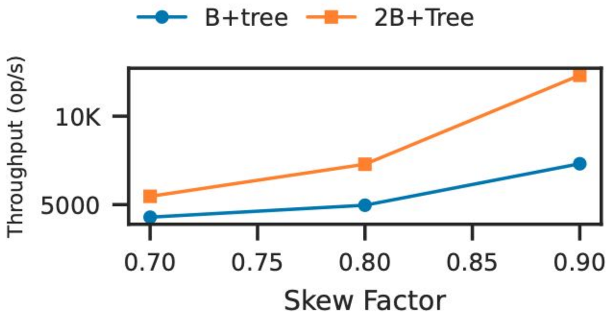


(b) Update

YCSB-Zipfian



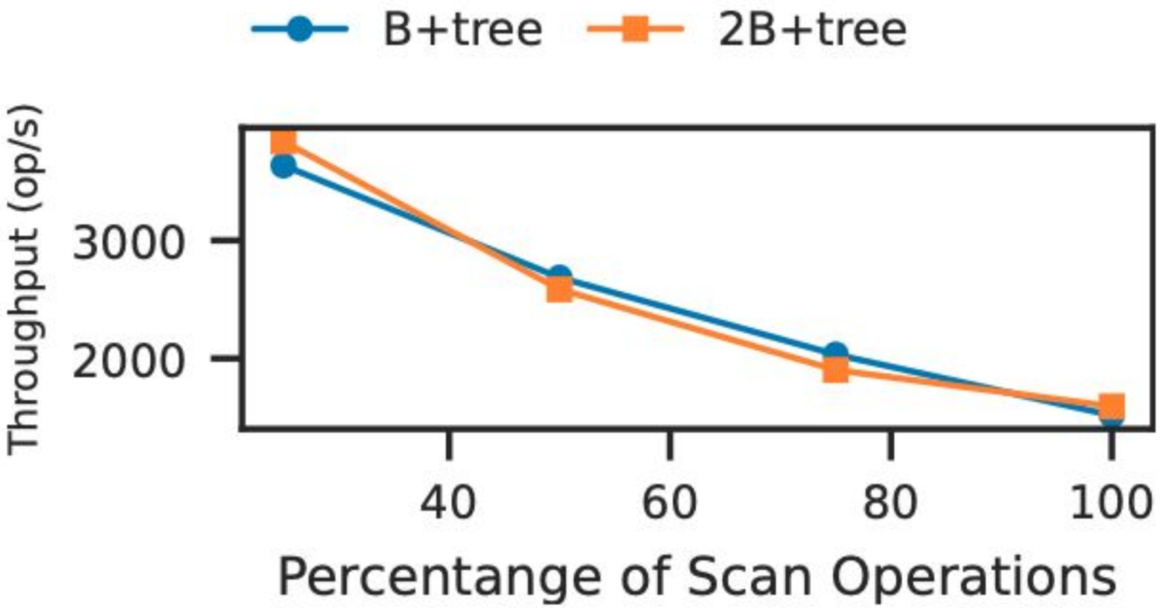
(a) Read-Only



(b) Update

Range Scan Operations

- On-par with single b+tree throughput for range scan.

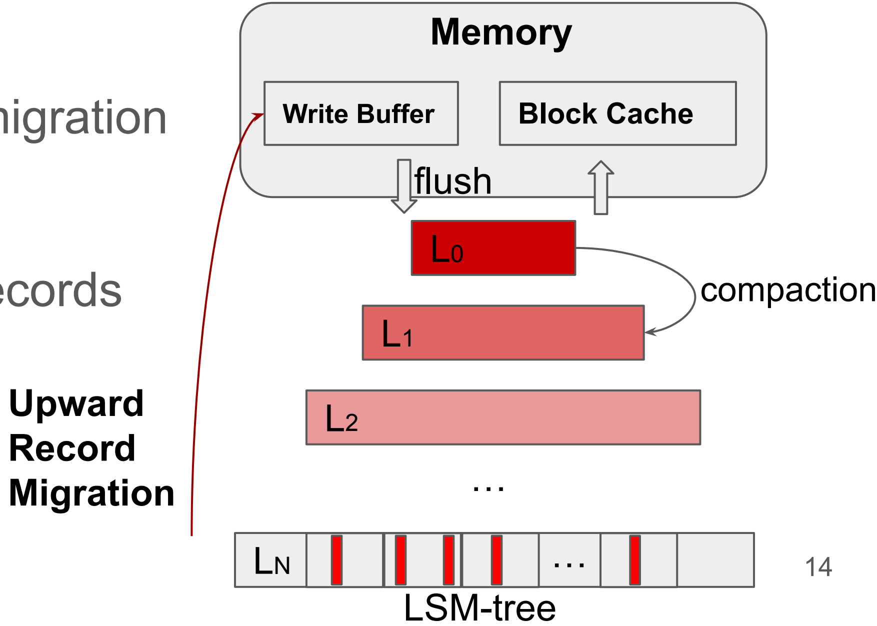


Summary: 2B+tree Improves upon Point Operations

	Point Read	Point Update	Range Scan
B+tree	★	★	★★★
2B+tree	★★★	★★★	★★★

2. Generalize to a N-Tree using LSM-tree

- A specific N-tree design using LSM-tree
  - Write-optimized
  - Better skew reads
- Augment LSM-tree with upward record migration for reads
  - Actively bring stationary but warm records closer to the top of hierarchy



Upward Migration in LSM-tree

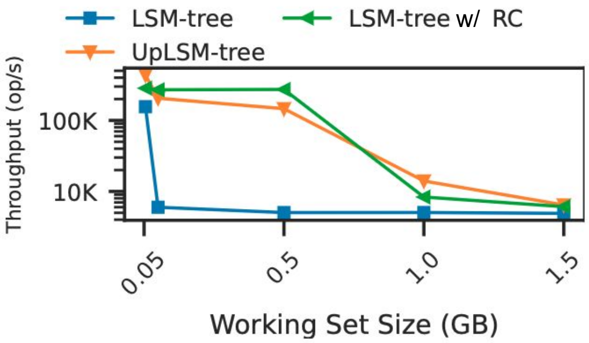
- How to identify stationary and warm records?
- Two heuristics:
  - 1. Upon block cache miss: Likely in the bottom of the hierarchy
  - 2. Apply the probabilistic upward migration to identify warm records

Experimental Setup

- We call our proposal UpLSM-tree, built on RocksDB
- Comparison using 1 GB memory budget
    - UpLSM-tree: 1GB block cache
    - Vanilla LSM-tree: 1GB block cache
    - LSM-tree with in-memory row cache for level files on disk
        - 90% memory allocated to row cache
        - 10% memory allocated to block cache

Point Operations

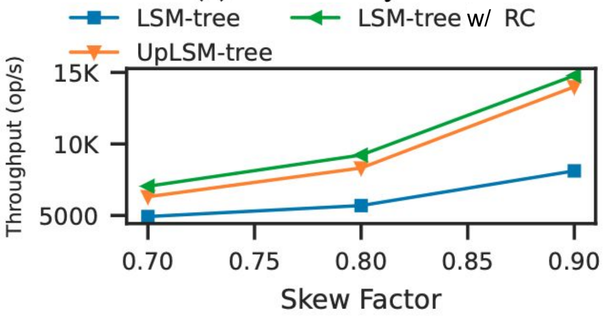
YCSB-Hotspot



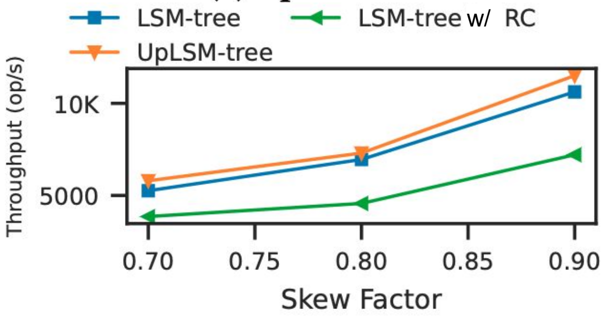
(a) Read-Only

(b) Update

YCSB-Zipfian



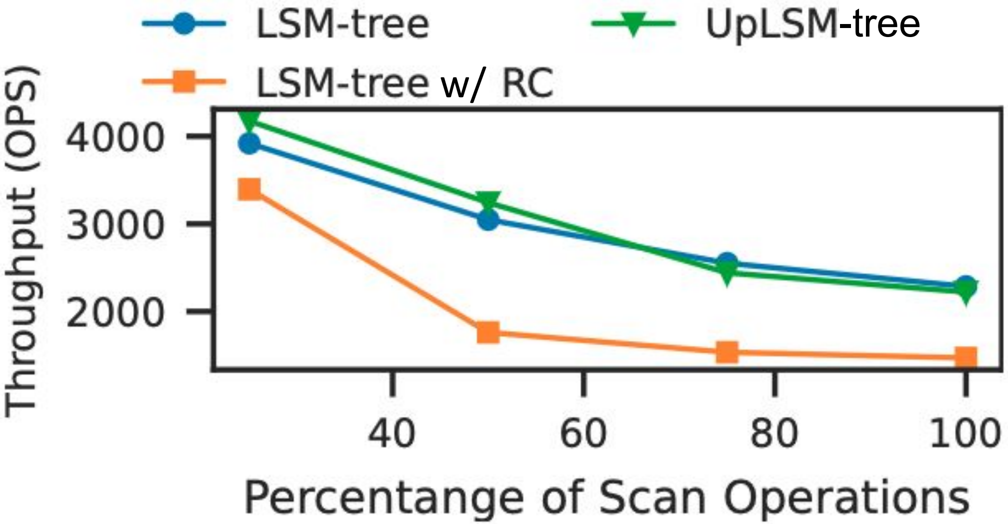
(a) Read-Only



(b) Update

Range Scan Operations

- On-par with vanilla LSM-tree
- Much better than LSM-tree with row caching
  - Row cache only helps point read operations
  - Block caching is more versatile



Summary: UpLSM-tree Dominates both Baselines

	Point Read	Point Update	Range Scan
LSM-tree	★	★★★	★★★
LSM-tree w/ RC	★★★	★	★
UpLSM-tree	★★★	★★★	★★★

Future Work

- Extend the architecture to non-tree-based structures
  - Hashing
  - Heap file with secondary indexes

Conclusion

- We studied improving memory utilization for tree data structures in data systems
  - Multi-structuring
  - Record-level migration
- Applications
  - 2B+tree for Traditional DBMS
  - A better LSM-tree
- Source Code: https://github.com/zxjcarrot/2-Tree

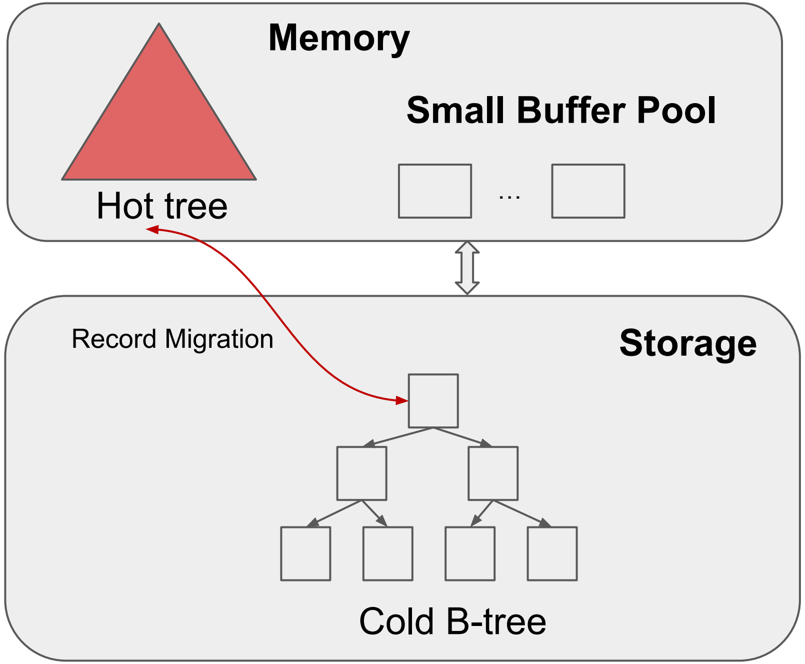
BACKUP SLIDES

Durability and Recovery of Migration

- No logical data changes
- Migration operation uses lightweight systems transaction
    - does not need to force log records to disk
    - Log persistence piggybacked on user commits
    - Analogy: btree split

2. Optimizing 2-Tree for Main-Memory DBMS

- Hot tree could stay completely in main-memory
  - Spill cold records out to disk
  - Serve larger-than-memory dataset
- A better Anti-Caching implementation
  - Scale to larger data set
  - Efficient range scan
- More details in the paper

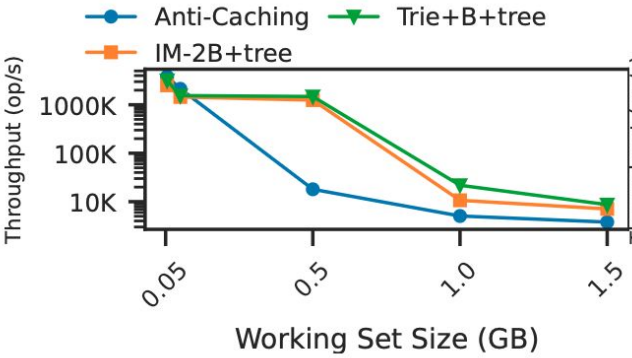


Comparison

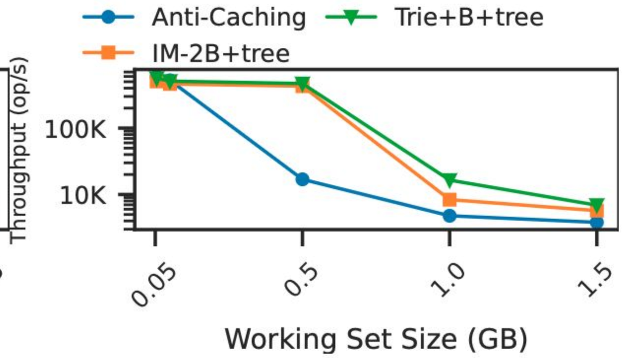
- Hot tree implemented using an in-memory btree, no buffer pool overhead
- Baseline: Original Anti-Caching implementation
  - LRU-based record eviction
  - Records stored unordered on disk

Point Operations

YCSB-Hotspot

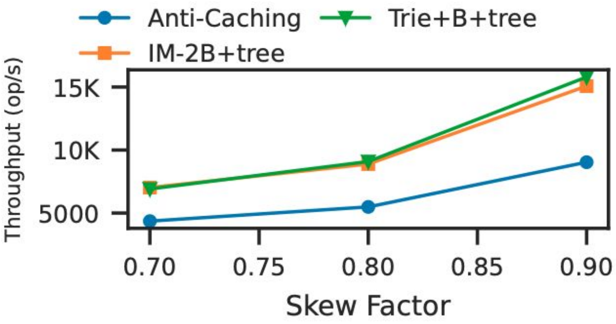


(a) Read-Only

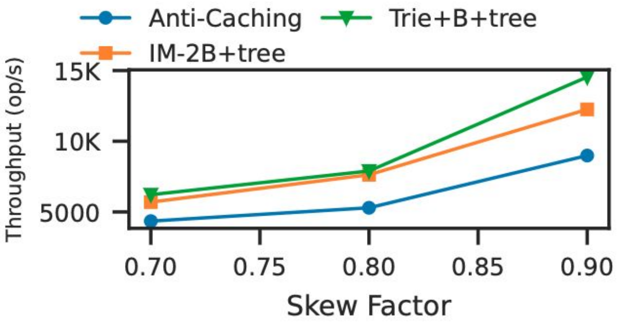


(b) Update

YCSB-Zipfian



(a) Read-Only



(b) Update

Range Scan Operations

- Significantly outperformed Anti-Caching original design