IndeXY: A Framework for Constructing Indexes Larger than Memory

Chen Zhong

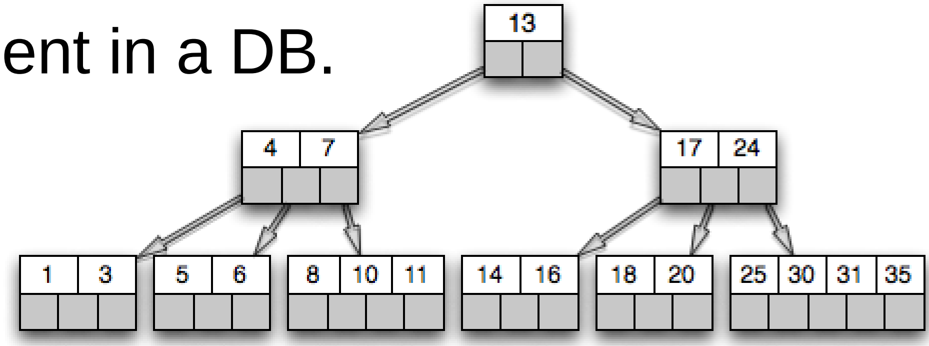
Xingsheng Zhao, Song Jiang

Qingqing Zhou, <u>Yuxing Chen</u>,

Kuang He, Anqun Pan

Insufficient Memory for Large Indexes

- Indexes are one of the most performance-critical component in a DB.



- Keep them all in the memory?
  - Usually too large:
    - For example, about 55% of the memory is occupied by the indexes in H-Store.
  - Often not necessary:
    - Not all parts of an index are accessed at a time.

- An apparent solution:
  - Use the disk as the memory extension
  - A large index spans memory and disk

State-of-the-art Practices - A Co-design approach

- In-memory databases:
  - migrate a selected subset of tuples out of the memory
    - E.g., Siberia and Anti-Caching.
  - migrate only subset of tuples, instead of index.
  - customized to database systems
    - E.g., respectively, Microsoft Hekaton and H-Store.

- On-disk databases:
  - cache the indexes in the memory
  - optimize the buffer cache management
    - For example, LeanStore uses pointer swizzling
  - use the common paged index structure for memory and disk
    - Not customized to the devices' different characterizations.

IndeXY: a Framework for Extensible Indexes

- Support integration of in-memory index (Index X) and on-disk index (Index Y).
  - Each can be independently selected
  - Each was designed for its target device.

- Provide a virtual-memory-like infrastructure at the key-value granularity.
  - The index becomes 'swappable'
  - Automate the process of maintaining a designated index memory limit by selecting keys for unloading to the disk.

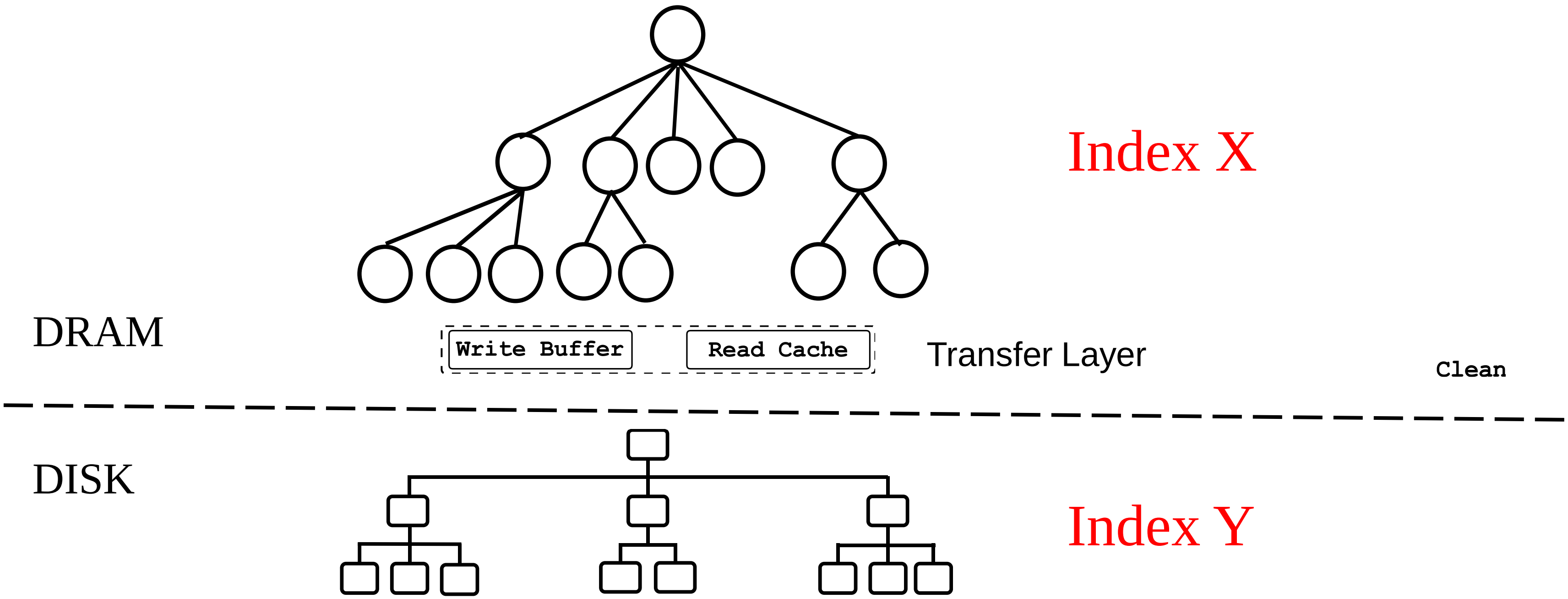
The Design Challenges

- How to minimize the changes to the existing Indexes X and Y?
  - No structural change to their data structures.
  - No logic change to the access algorithms.

- How to efficiently track access hotness?
  - Access unit is small (individual keys).
  - Both time and space overhead needs to be considered.

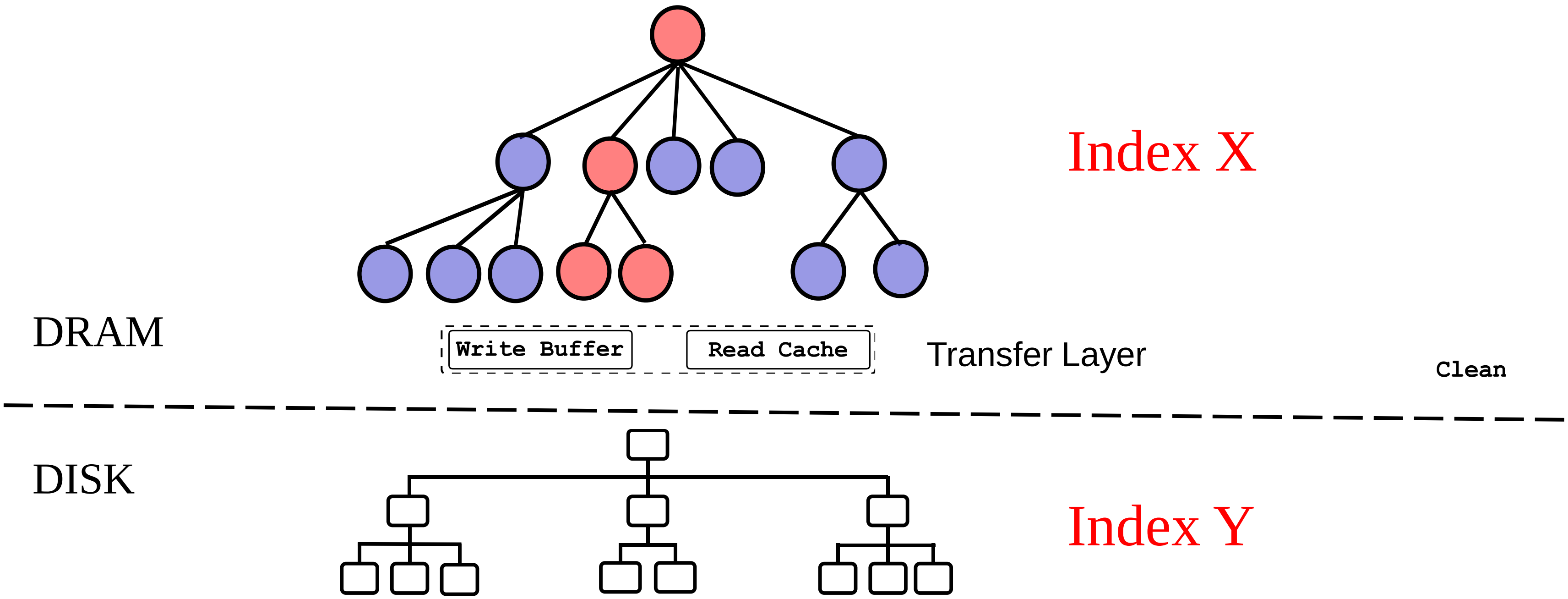
- How to generate high-performance write requests to Index Y?
  - Sequential writes are friendly to disk performance.
  - Both temporal and spatial locality needs to be considered.

The Big Picture



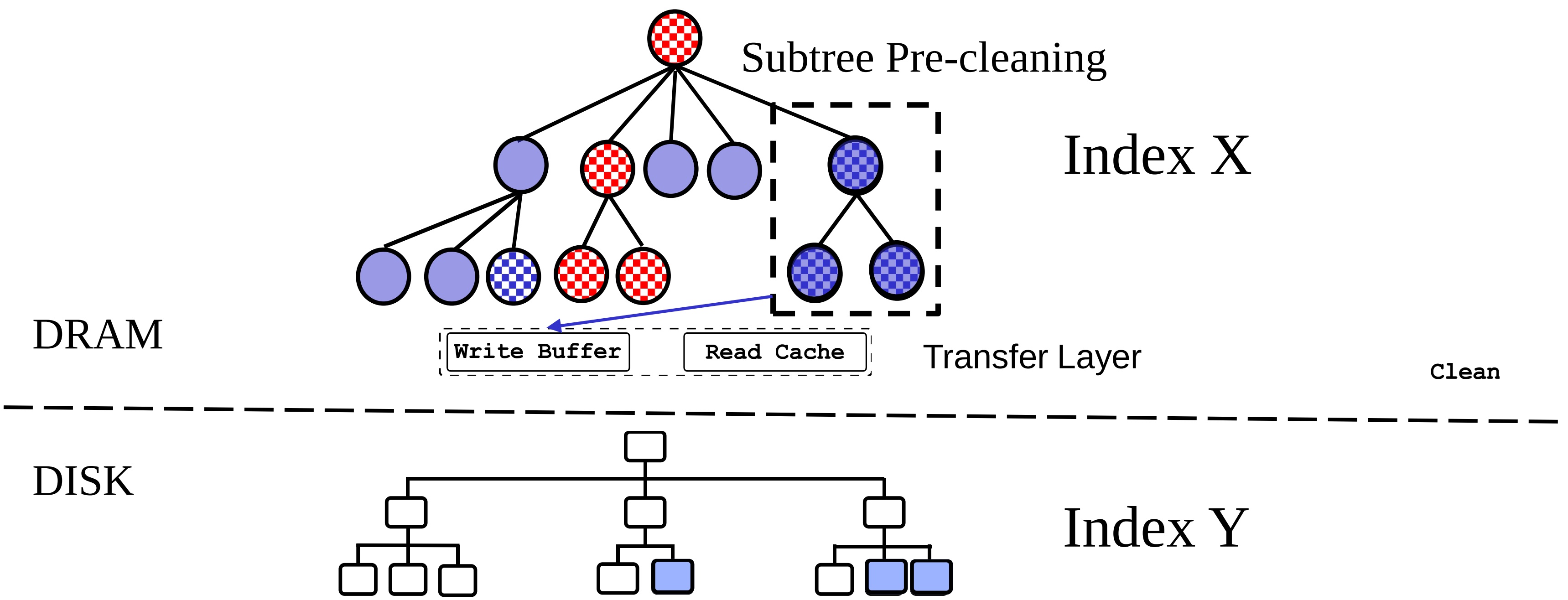
The framework’s major features are integrated into Index X.

The Big Picture



The framework's major features are integrated into Index X.

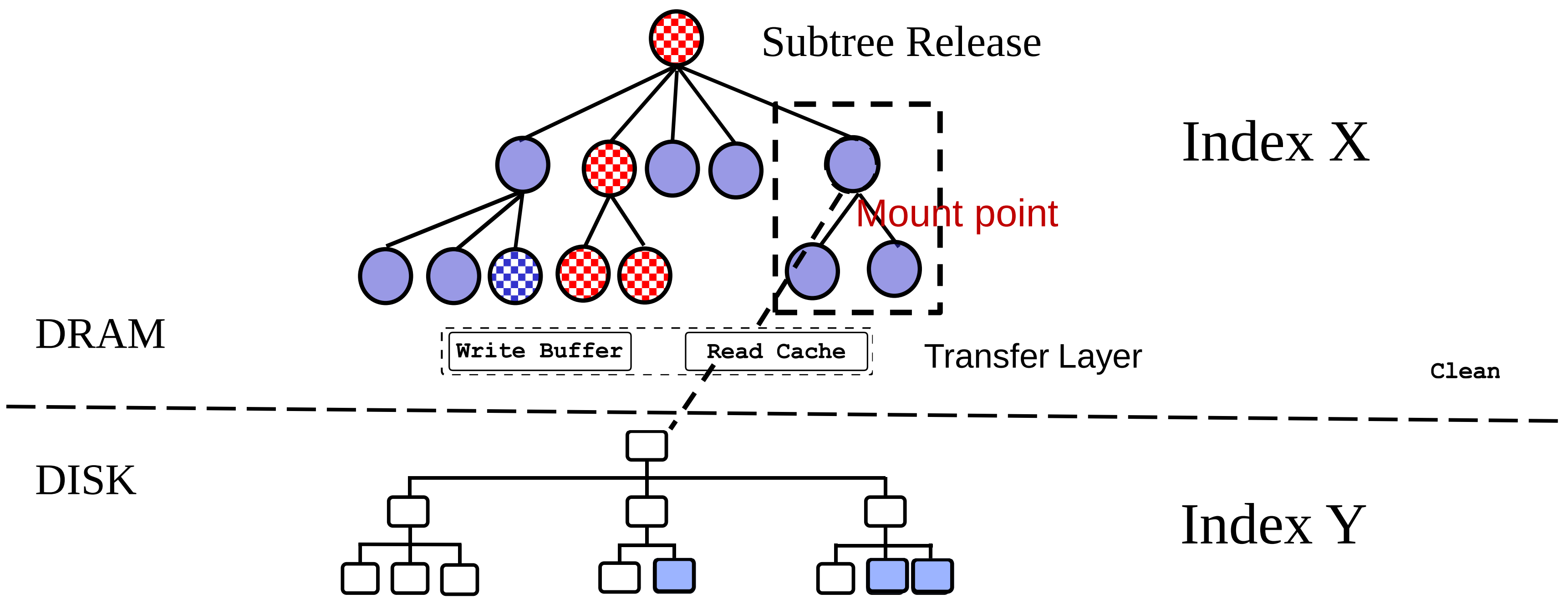
The Big Picture



Not actively dirty sub-trees will be cleaned.

- a background pre-cleaning process
- prepare for later quick space reclamation.

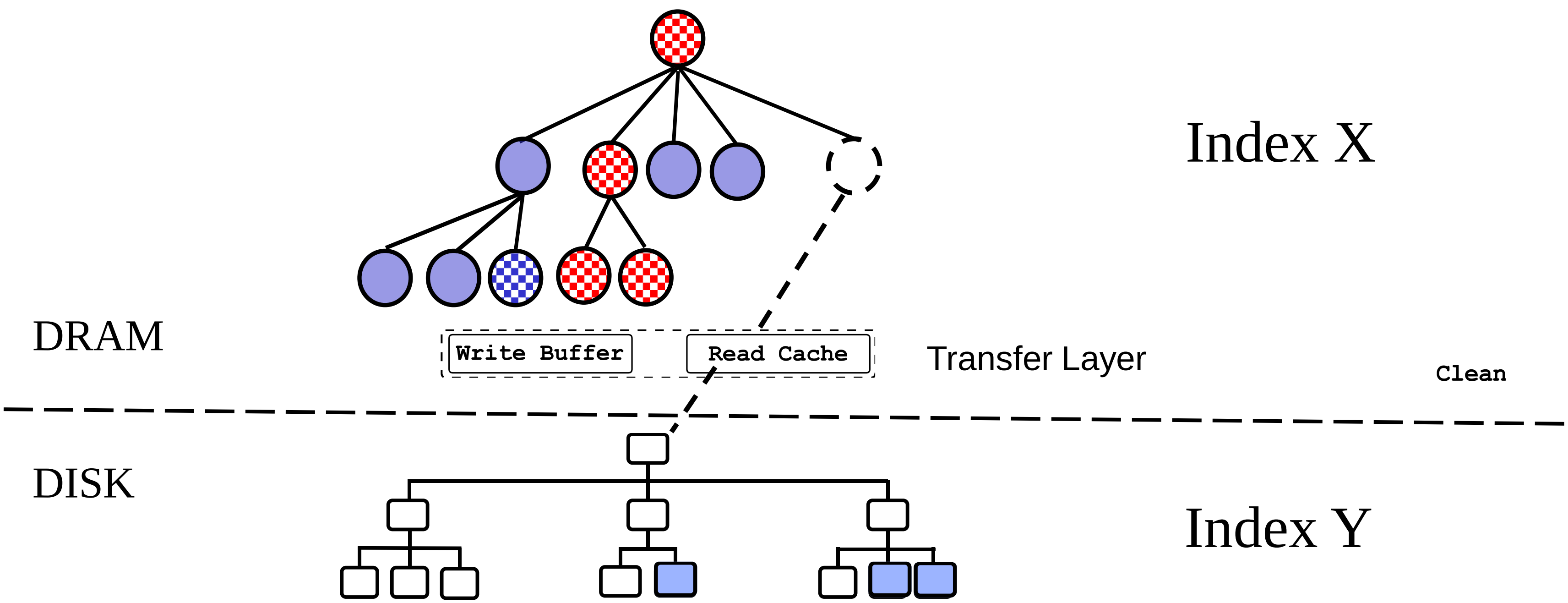
The Big Picture



Large and cold subtrees will be selected for release.

- space reclamation to keep the index size below its memory limit.

The Big Picture

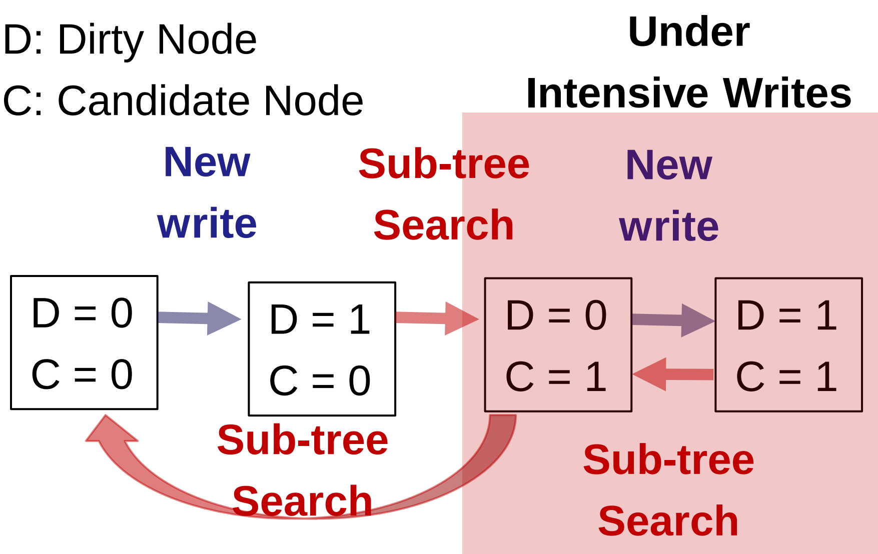


- New writes to Index X also go to a write-ahead-log for recovery.
- A thin transfer layer between Indexes X and Y is set up.
  - A write buffer to form sequences of disk writes
  - A read cache to act as a small block cache.

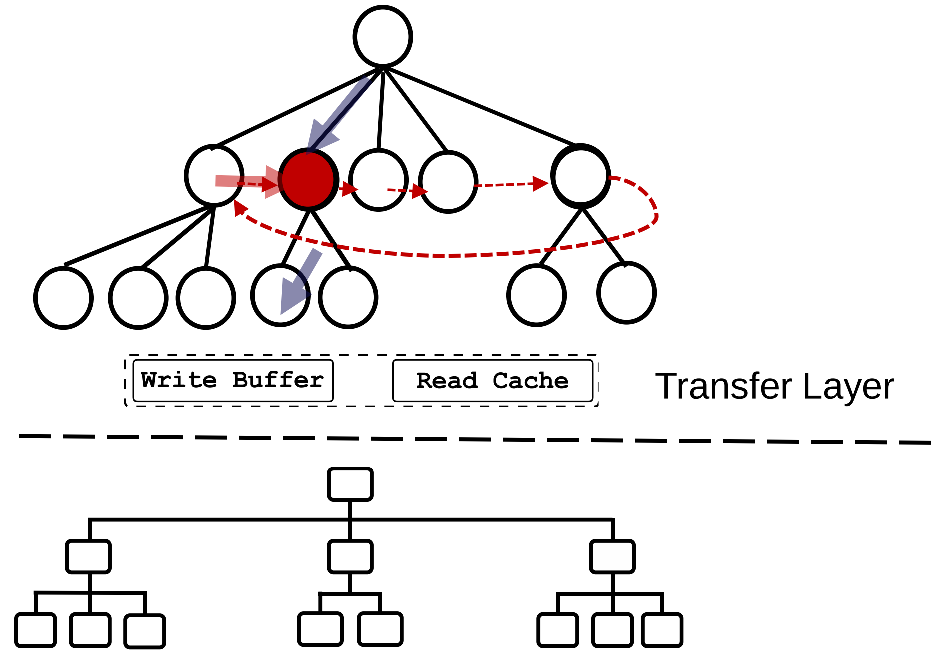
Pre-cleaning Sub-trees

D: Dirty Node

C: Candidate Node

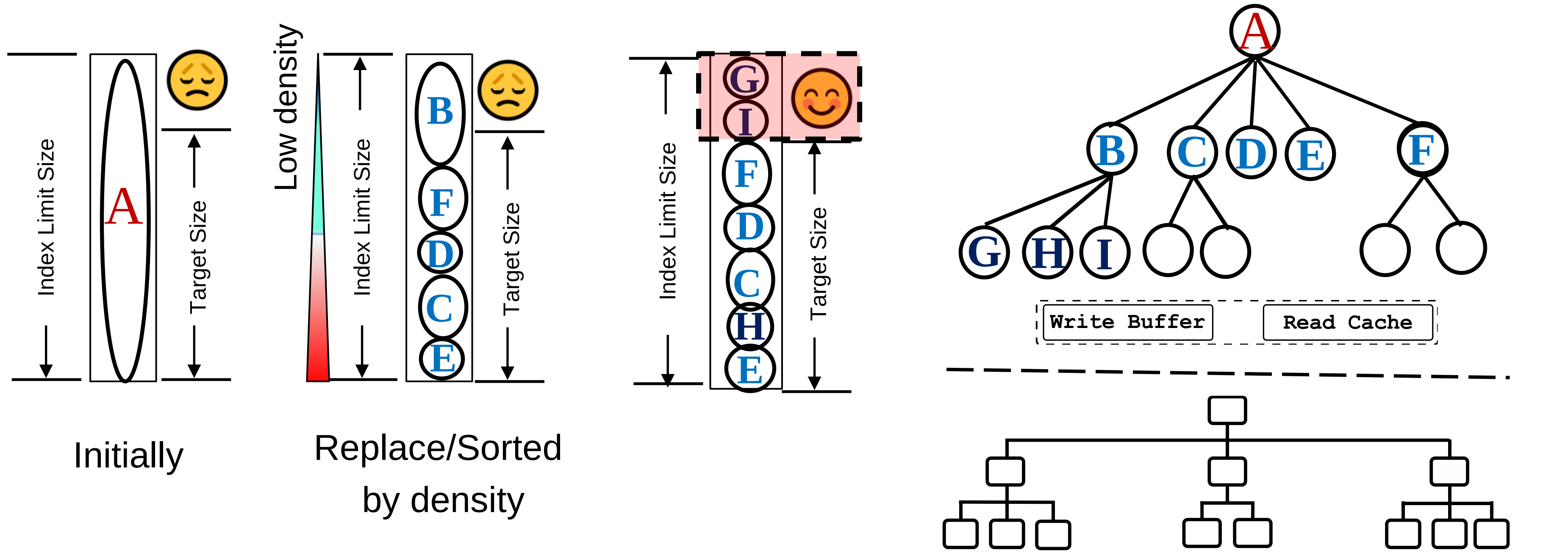


The subtree found!



- Search for a sub-tree for pre-cleaning on a linked list of inner nodes
- Use a two-bit algorithm to detect subtrees that:
  - have been written before
  - currently not under intensive writes.

Sub-tree Release



- When Index X size approaches its limit, memory release thread is triggered.
- Access density of a **subtree** = # of accesses / the subtree size
- Efficiently track the density (see the paper)
- Density-based subtree ranking algorithm to find a subtree for release:
  - The subtree is not frequently accessed; and
  - The subtree is large.

Performance Evaluation

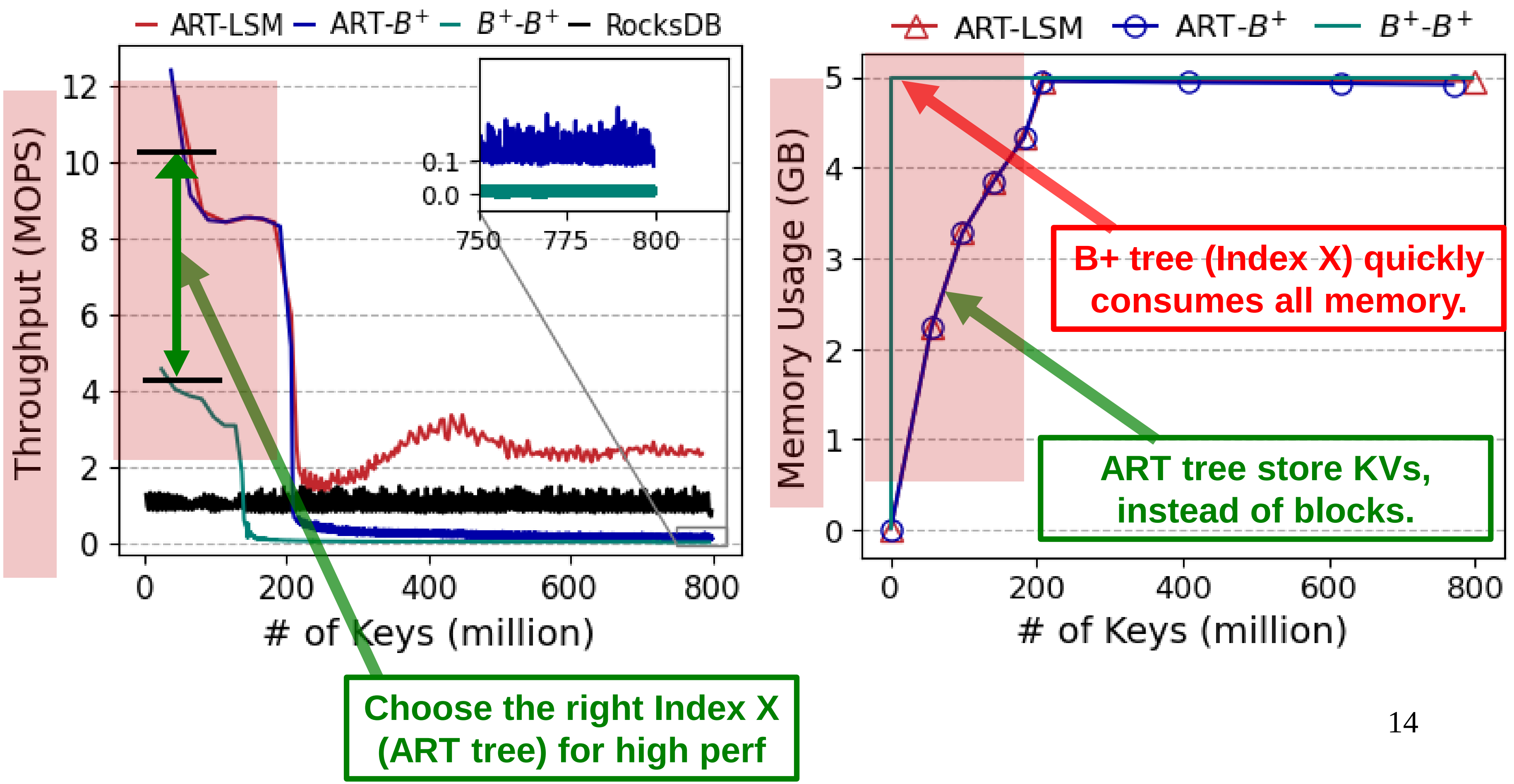
- The workloads
  - Micro-benchmarks
  - YCSB
  - TPC-C
- Experiment setup
  - Intel Xeon Platinum 8255C CPU processors
  - 128 GB DRAM, 512 GB SAMSUNG MZ7LH480 SSD
- The system

Systems	Index X	Index Y
B+ – B+	B+ Index	B+ Index
ART – B+	ART Index	B+ Index
ART – LSM	ART Index	LSM-tree Index
RocksDB	RocksDB buffer	LSM-tree Index

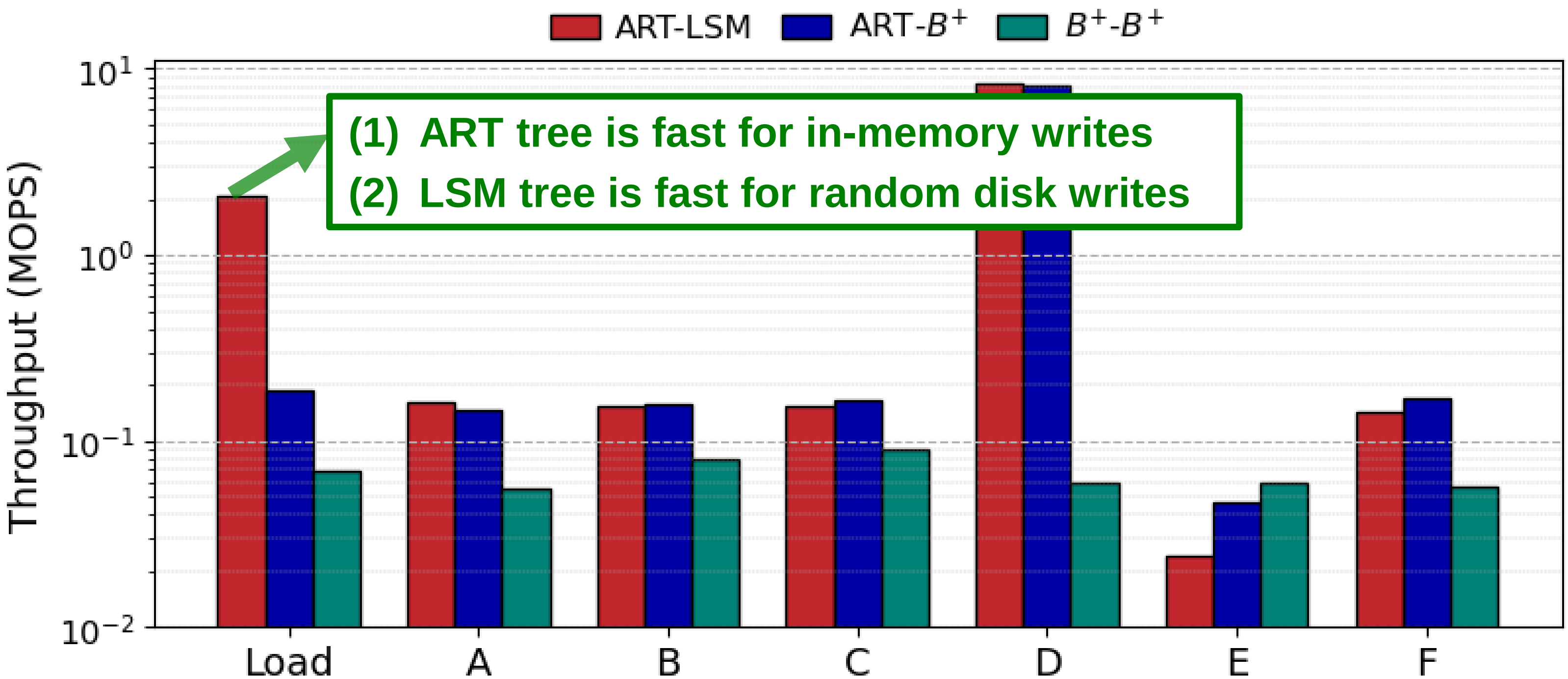
* Use LeanStore as a representative of B+ tree design

Write Performance (memory limit = 5GB)

Insert 800 million unique random key-value 16B pairs (12GB working set).



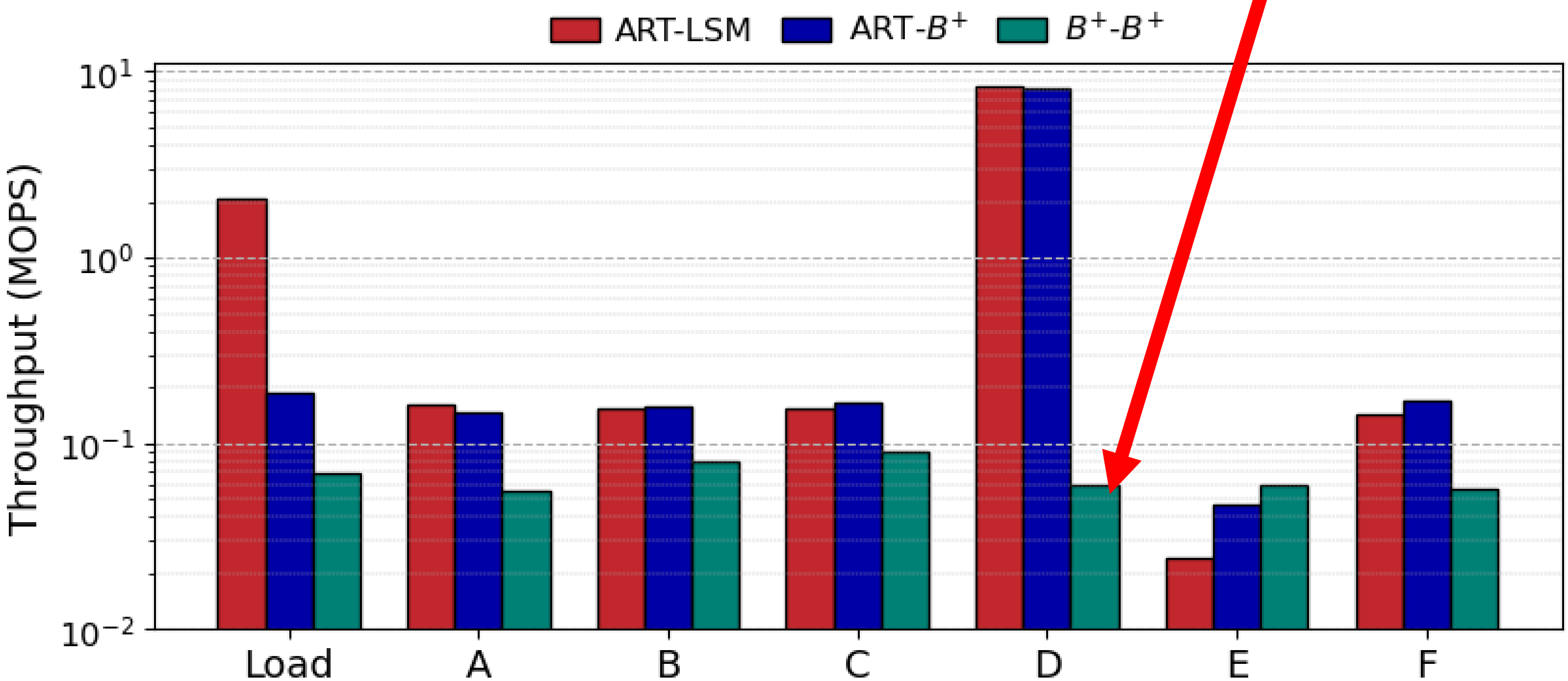
The YCSB Benchmark (memory limit = 30 GB)



Workloads	Description
Load	100% Random write
A	50% Read, 50% Update
B	95% Read, 5% Update
C	100% Read
D	95% Read Latest, 5% Update
E	95% Scan (average scan length 50, maximum 100), 5% Update
F	50% Read-Modify-Write, 50% Read

The YCSB Benchmark (memory limit = 30 GB)

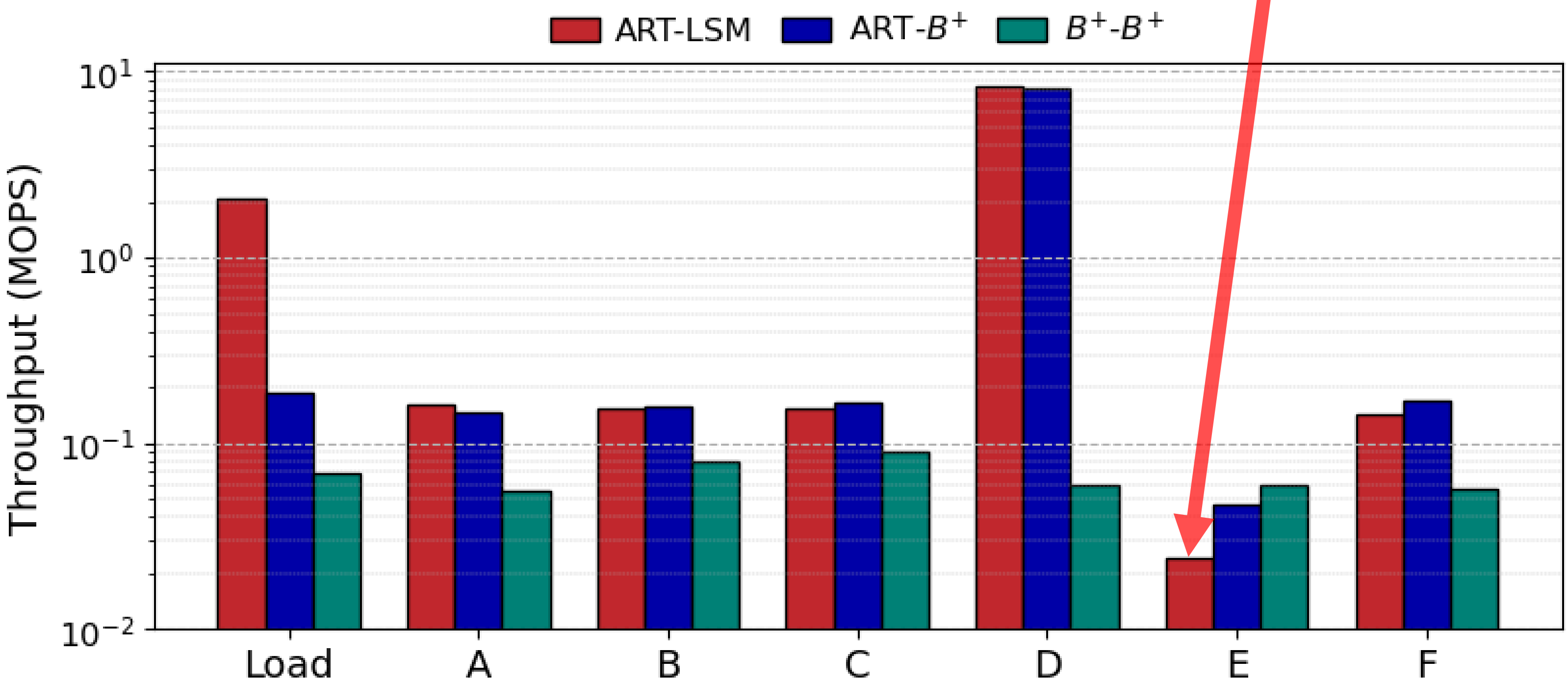
B+-B+ doesn't use right index for each index



Workloads	Description
Load	100% Random write
A	50% Read, 50% Update
B	95% Read, 5% Update
C	100% Read
D	95% Read Latest, 5% Update
E	95% Scan (average scan length 50, maximum 100), 5% Update
F	50% Read-Modify-Write, 50% Read

The YCSB Benchmark (memory limit = 30 GB)

LSM tree is not a right choice of Index Y for scans



Workloads	Description
Load	100% Random write
A	50% Read, 50% Update
B	95% Read, 5% Update
C	100% Read
D	95% Read Latest, 5% Update
E	95% Scan (average scan length 50, maximum 100), 5% Update
F	50% Read-Modify-Write, 50% Read

A Summary

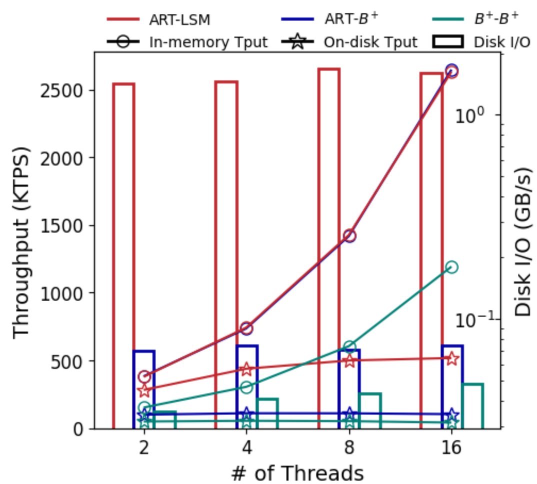
- IndeXY is a framework for a decoupled deployment of indexes in the memory and on the disk.
- IndeXY provides highly efficient supports for migrating index subsets between memory and disk.
- With the ease of selection and integration of two indexes, IndeXY makes quick development of high-performance extensible indexes possible.

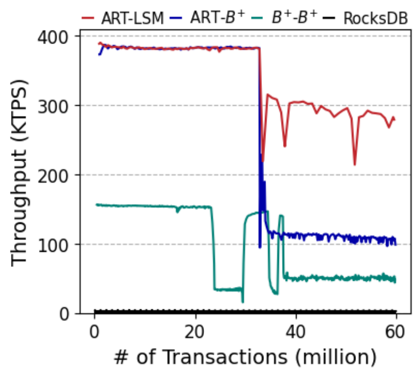
Backup Slides

TPC-C benchmark

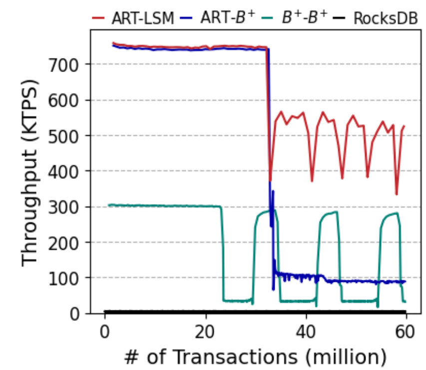
100 warehouse (~ 10GB)

New Order and Payment transactions are enabled (90%)

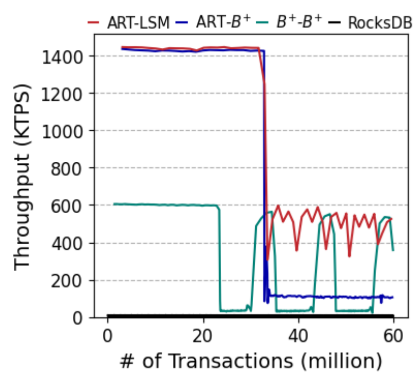




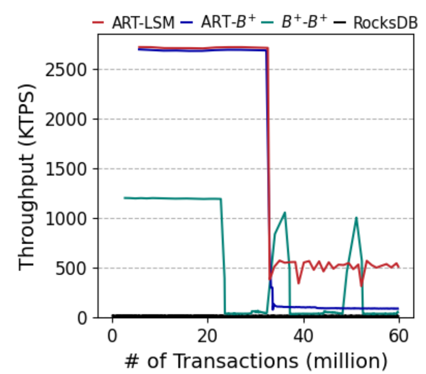
2 thread



4 thread

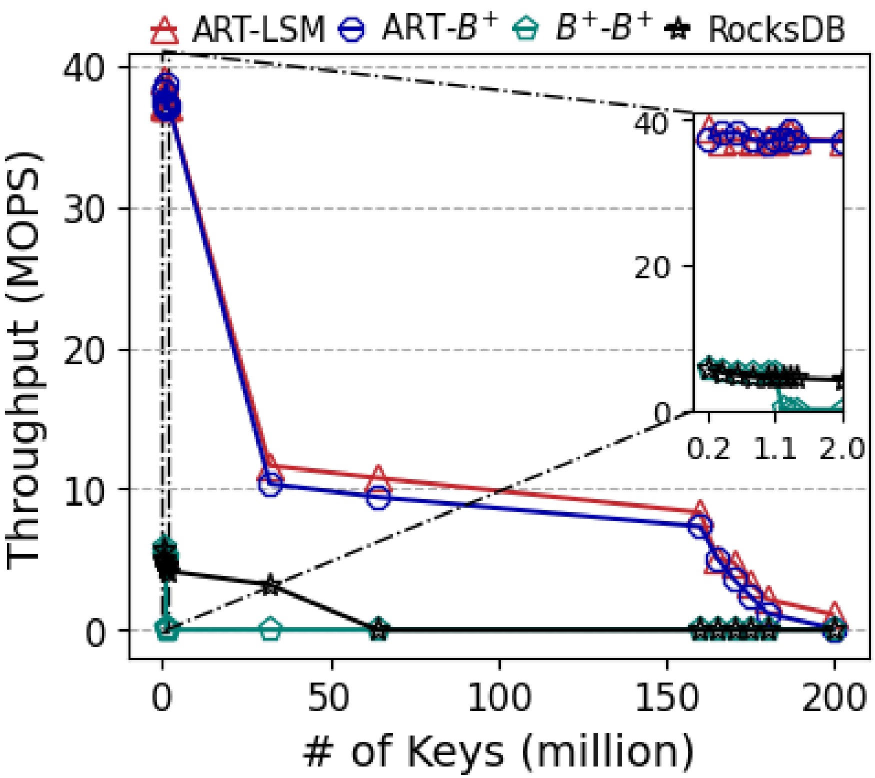


8 thread

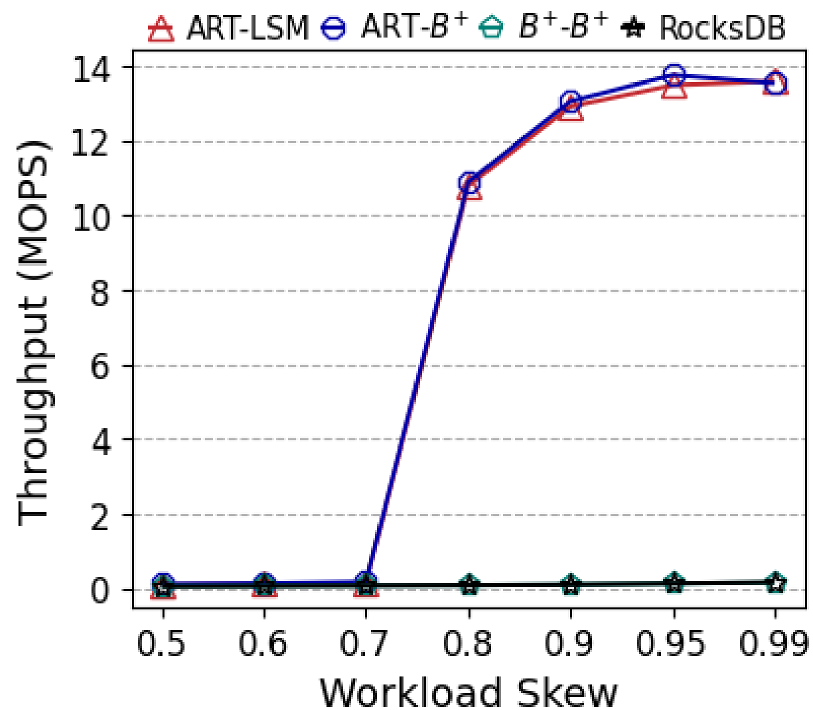


16 thread

Read Performance (Micro-benchmark)

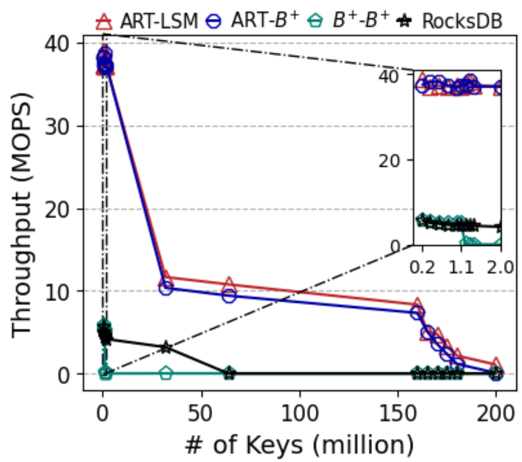


Read throughput with
different working set sizes

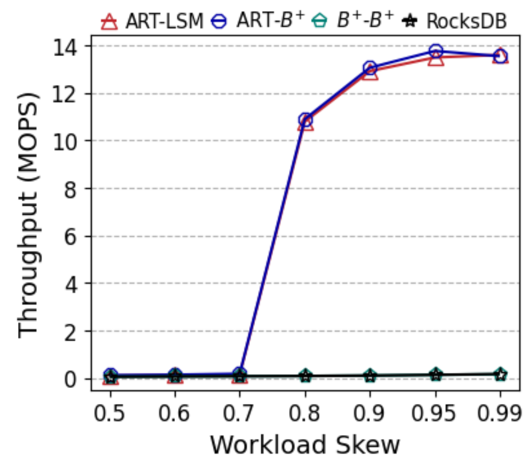


Read throughput with
Various Zipfian distributions

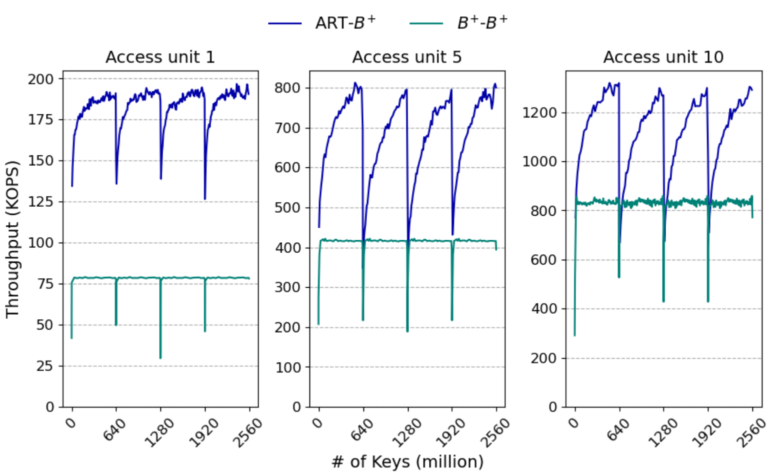
Read Performance (Micro-benchmark)



Read throughput with different working set sizes



Read throughput with Various Zipfian distributions



Lookup performance with shifting workload