Out-of-Order Sliding-Window Aggregation with Efficient Bulk Evictions and Insertions

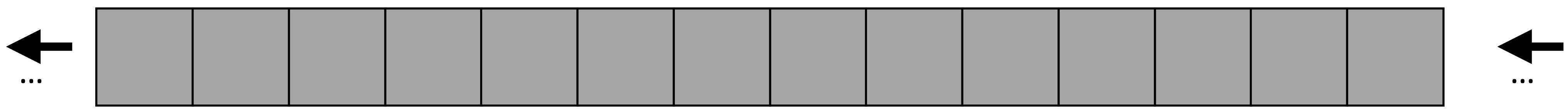
Kanat Tangwongsan
Mahidol University International College (MUIC)

Martin Hirzel
IBM Research

Scott Schneider
Meta

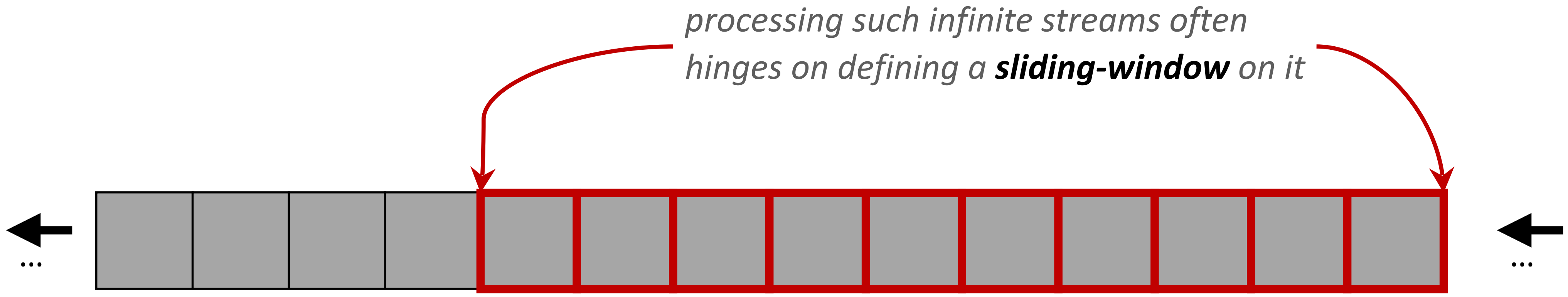
Scan me for the paper

Context & Motivation



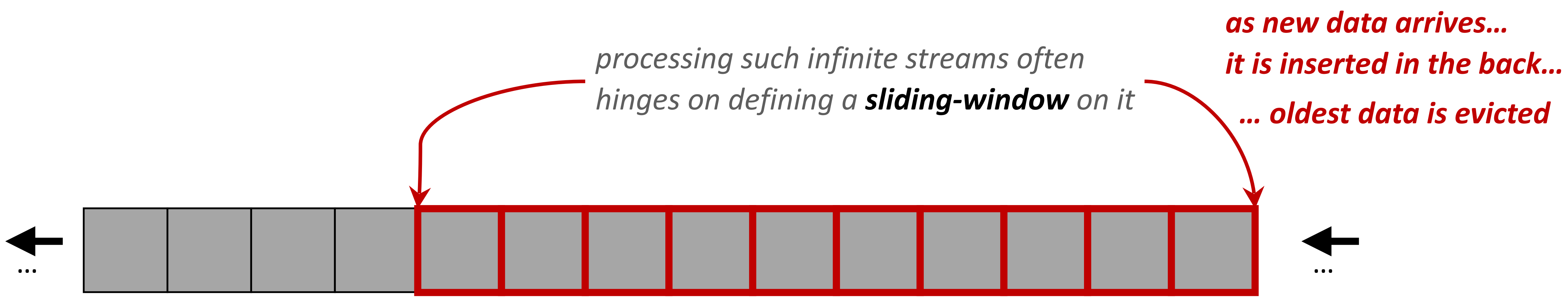
a ***stream*** = a (potentially infinite) sequence of data

Context & Motivation



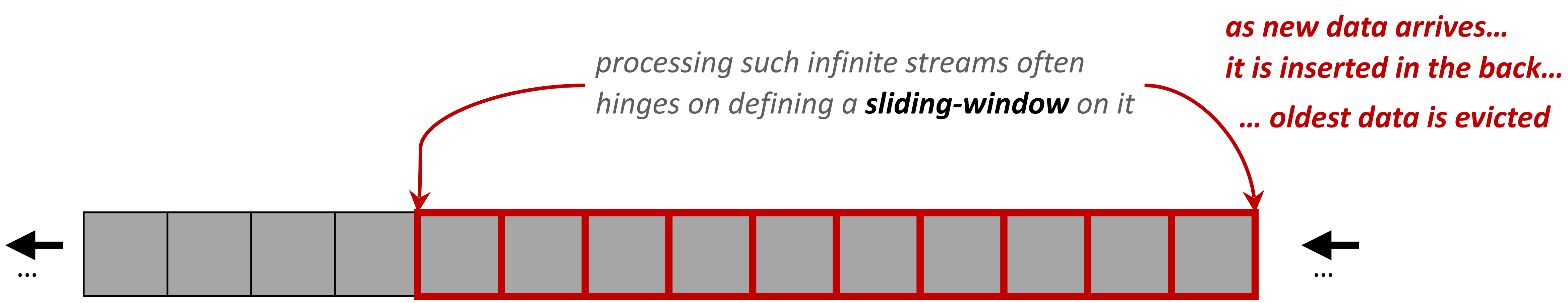
a ***stream*** = a (potentially infinite) sequence of data

Context & Motivation



a ***stream*** = a (potentially infinite) sequence of data

Context & Motivation



a ***stream*** = a (potentially infinite) sequence of data

- **Sliding-Window Aggregation**
  Combine data items in time-order using a binary operator. E.g., maxCount, min, Bloom filter, mergeable sketches. Only expect associativity, not commutativity nor inverses.

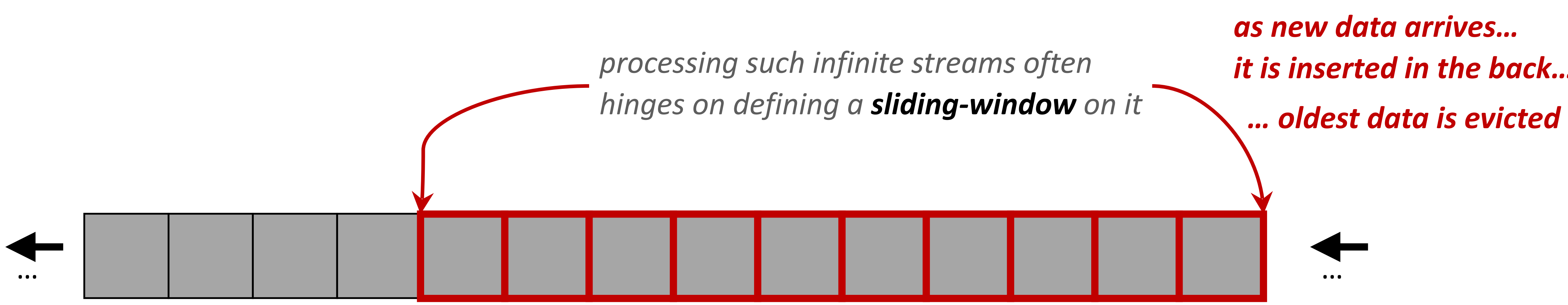
Context & Motivation

a ***stream*** = a (potentially infinite) sequence of data

- **Sliding-Window Aggregation**
  Combine data items in time-order using a binary operator. E.g., maxCount, min, Bloom filter, mergeable sketches. Only expect associativity, not commutativity nor inverses.

- **Out-of-order Streams**
  Data items are timestamped. The newest arrivals may be older than the most recent previous arrivals. E.g., clock skews across IoT devices.

Context & Motivation



a ***stream*** = a (potentially infinite) sequence of data

- **Sliding-Window Aggregation**
  Combine data items in time-order using a binary operator. E.g., maxCount, min, Bloom filter, mergeable sketches. Only expect associativity, not commutativity nor inverses.
- **Bulk Arrivals/Departures**
  Multiple data items enter/leave the window at once. E.g., catching up after an outage.
- **Out-of-order Streams**
  Data items are timestamped. The newest arrivals may be older than the most recent previous arrivals. E.g., clock skews across IoT devices.

Selected

Prior and Related Work

	Sliding-Window Aggregation	Out-of-order Support	Bulk Handling
AMTA [Villalba-Berral-Carrera, TPDS'19]	✓ Amortized $O(1)$	✗	Only bulk eviction, taking $O(\log n)$
DABA Lite [T.-Hirzel-Schneider, VLDBJ'21]	✓ Worst-case $O(1)$	✗	✗
FiBA [T.-Hirzel-Schneider, VLDB'19]	✓ Amortized $O(\log d)$	✓	✗
Data Structure Papers [Brown-Tarjan'79, Kaplan-Tarjan'95, Hinze-Paterson'06]	✗	✗	✓ Various settings

Other work and techniques: Scotty, CPiX, ChronicleDB, Hammer Slide, LightSaber, FlatFIT

This Paper: Efficient Bulk Evictions and Insertions

	Sliding-Window Aggregation	Out-of-order Support	Bulk Handling
FiBA [T.-Hirzel-Schneider, VLDB'19]	✓ Amortized $O(\log d)$	✓	✗

This Paper: Efficient Bulk Evictions and Insertions

	Sliding-Window Aggregation	Out-of-order Support	Bulk Handling
FiBA [T.-Hirzel-Schneider, VLDB'19]	✓ Amortized $O(\log d)$	✓	✗

Make FiBA natively support bulk operations → ✓

- `bulkInsert(B)` - Add a bulk of ordered data to the window
- `bulkEvict(t)` - Remove all items with timestamps ≤ `t`
- Keep query (whole window + range) the same

This Paper: Efficient Bulk Evictions and Insertions

Theorem [This Paper]:

- bulkEvict in amortized $O(\log m)$ time
- bulkInsert in amortized $O(m \log \frac{d}{m})$ time
- query in worst-case $O(1)$

where n = window size, m = the bulk size and d = out-of-order distance = # of data items in the window that overlap with the bulk

dling

FiB/

[T.-Hir

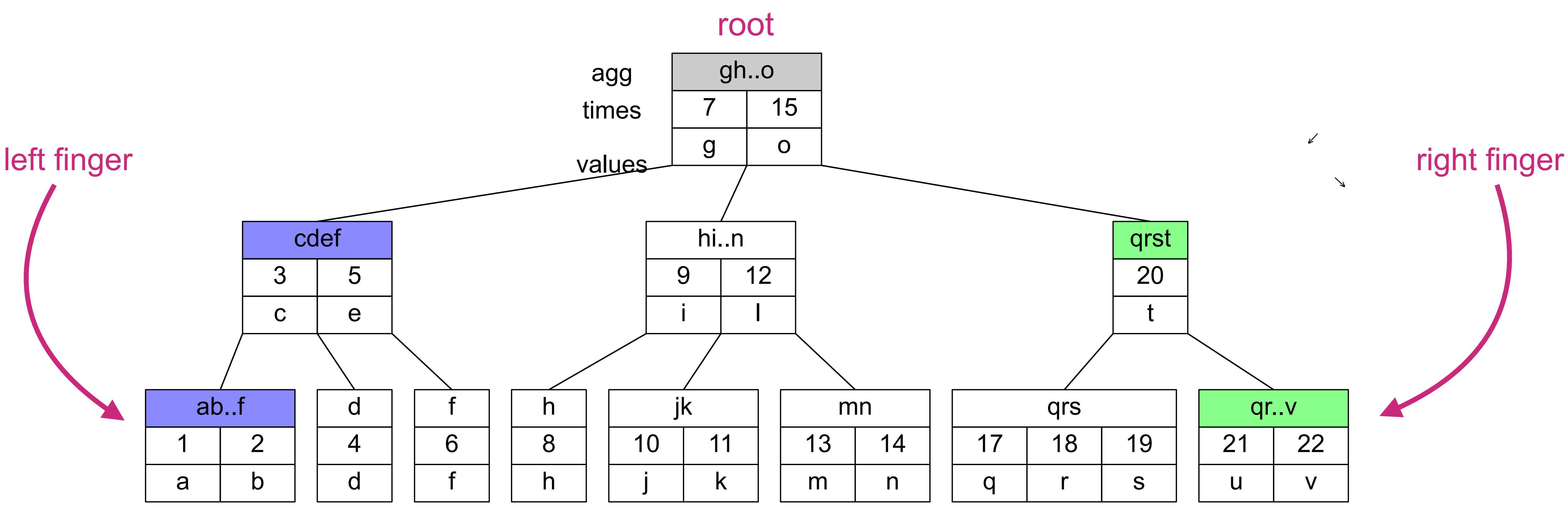
Make FiBA natively support bulk operations → ✓

- `bulkInsert(B)` - Add a bulk of ordered data to the window
- `bulkEvict(t)` - Remove all items with timestamps ≤ t
- Keep query (whole window + range) the same

This work builds on FiBA

Finger B-Tree Aggregator

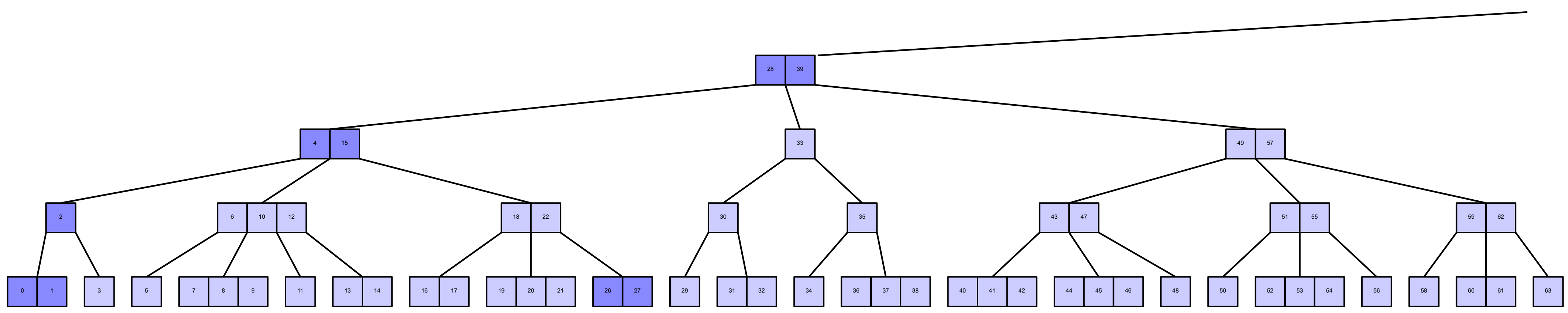
[T.-Hirzel-Schneider, VLDB'19]



- Timestamp-ordered B-Tree keeping data in internal + leaf nodes
- Left and right fingers for faster searching
- Position-aware partial aggregates

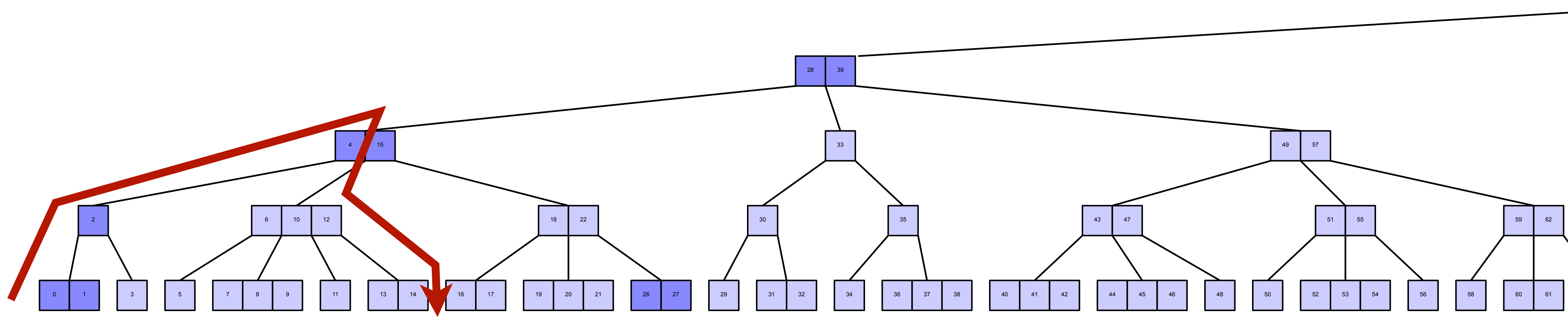
bulkEvict, intuitively…

To support `bulkEvict(t)`**…**



bulkEvict, intuitively…

To support `bulkEvict(t)`**…**

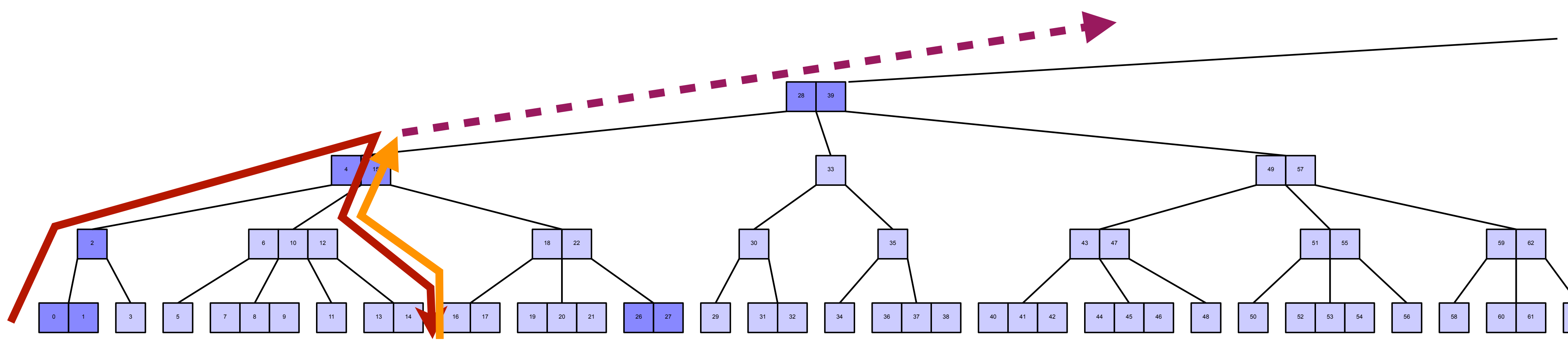


1. **Boundary search from a finger as if looking for *t***

   Goal: List every node on the discard-keep boundary and its right neighbor

bulkEvict, intuitively…

To support `bulkEvict(t)`**…**



1. **Boundary search from a finger as if looking for *t***

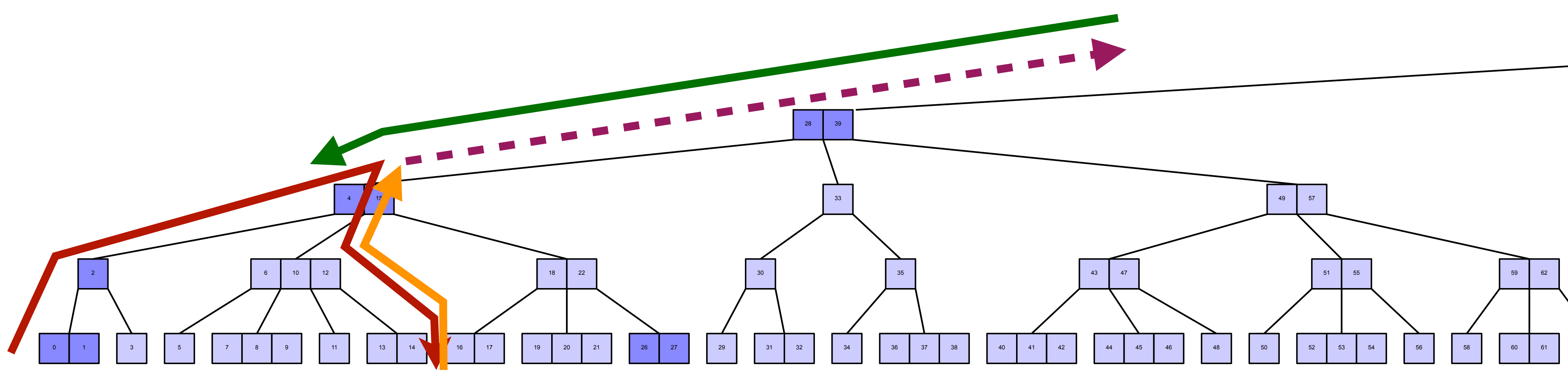
   Goal: List every node on the discard-keep boundary and its right neighbor

2. **A pass up along the boundary towards the root**

   Goal: Disconnect nodes to discard and repair the affected nodes towards the root

bulkEvict, intuitively…

To support `bulkEvict(t)`**…**



1. **Boundary search from a finger as if looking for *t***

   Goal: List every node on the discard-keep boundary and its right neighbor

2. **A pass up along the boundary towards the root**

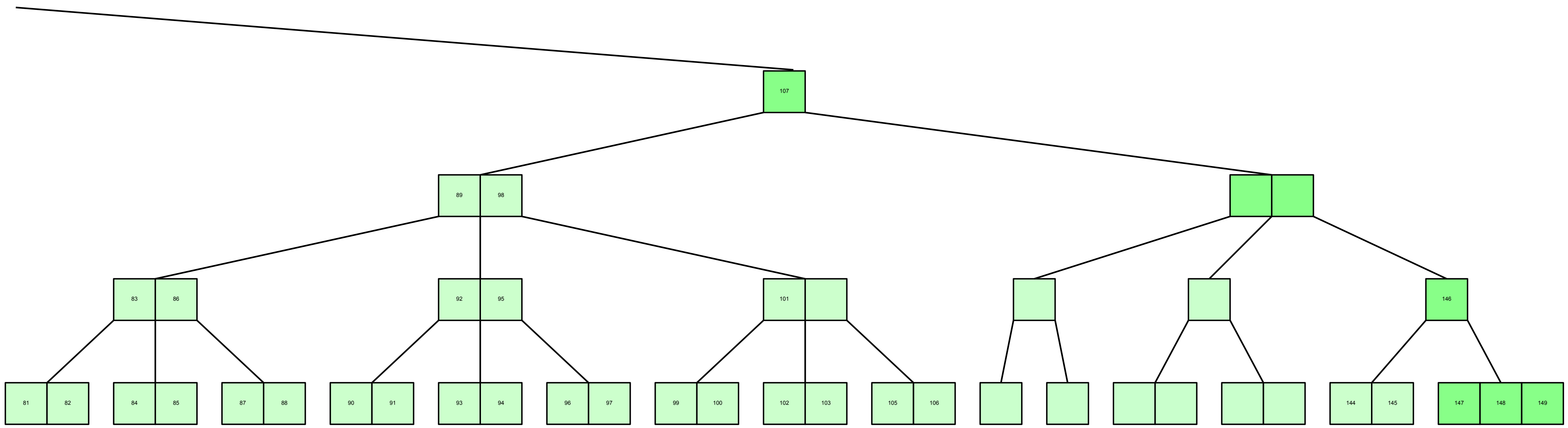
   Goal: Disconnect nodes to discard and repair the affected nodes towards the root

3. **A pass down to clean up nodes on the updated spine(s)**

   Goal: Fix the spine(s) and repair spine aggregates

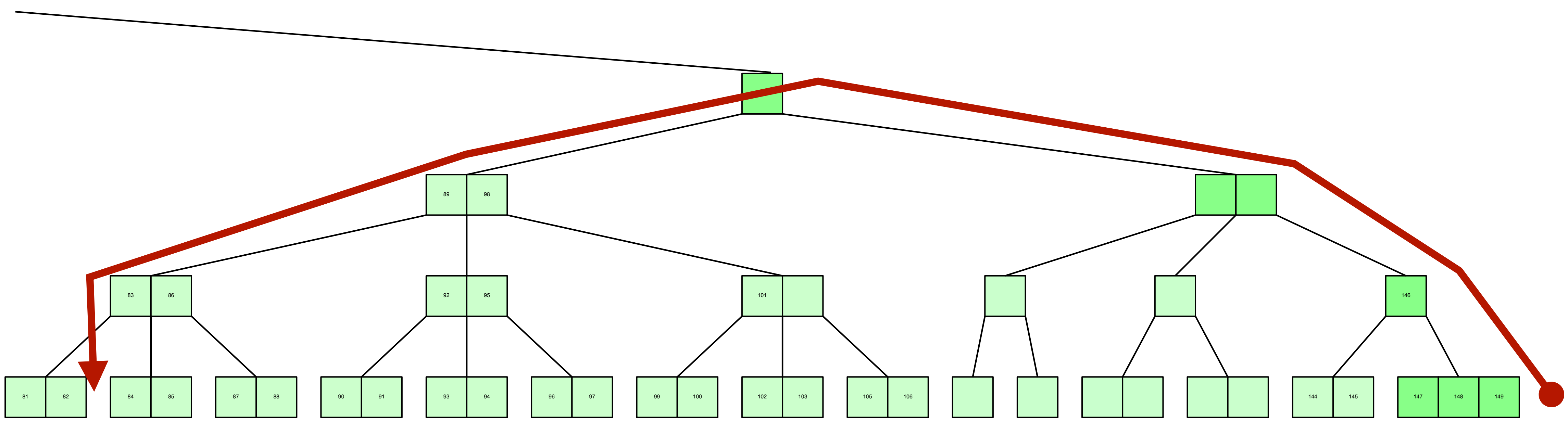
bulkInsert, intuitively…

To support `bulkInsert(B)`**…**



bulkInsert, intuitively…

To support `bulkInsert(B)`**…**

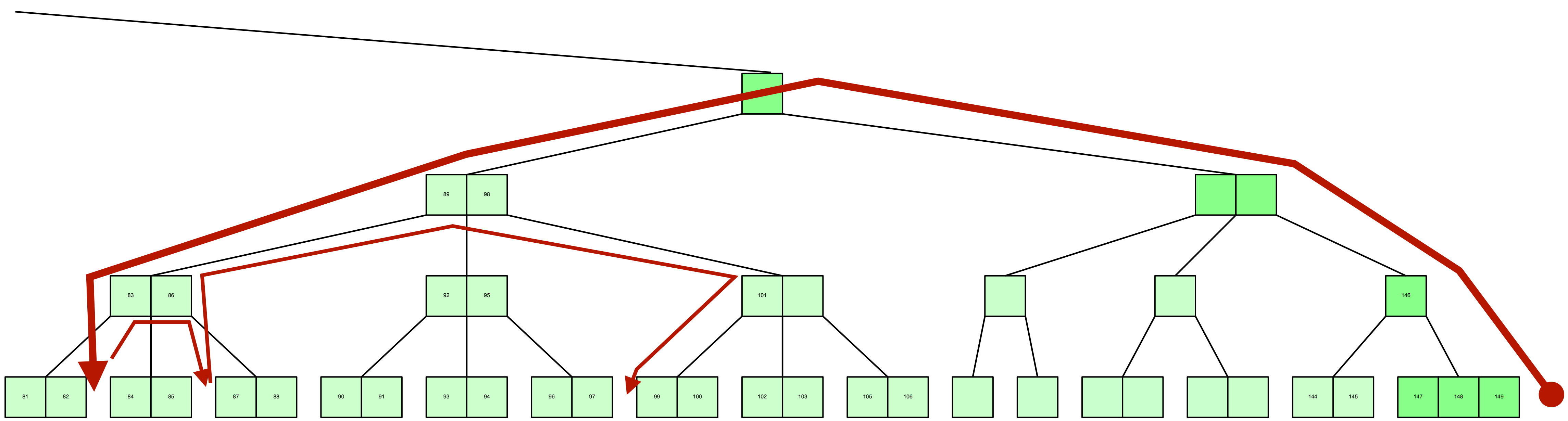


1. **Search for insertion sites, starting with the oldest entry in the batch**

   Goal: Identify the nodes where the batch entries will go to, without starting the search from scratch for every search.

bulkInsert, intuitively…

To support `bulkInsert(B)`**…**

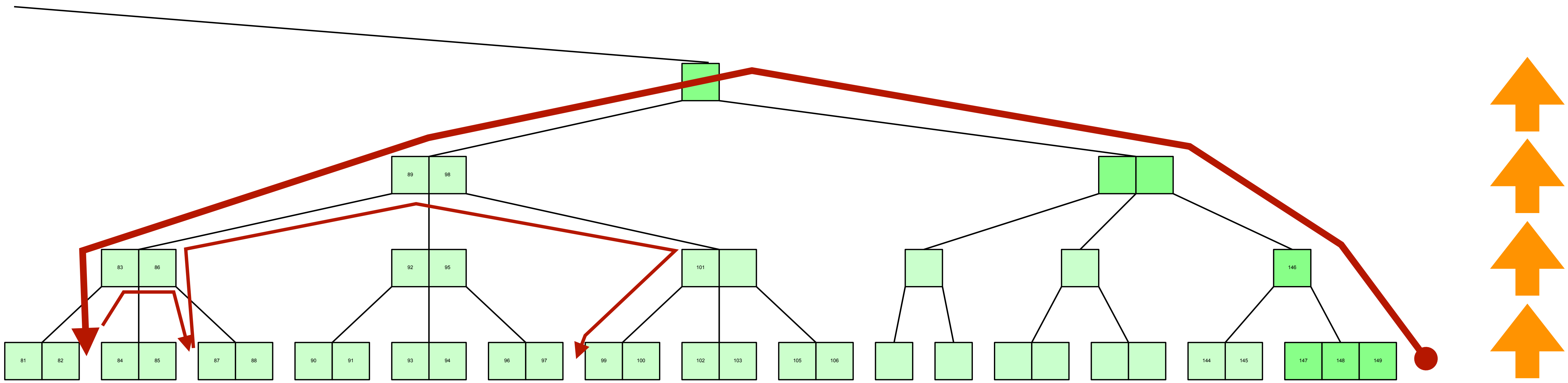


1. **Search for insertion sites, starting with the oldest entry in the batch**

   Goal: Identify the nodes where the batch entries will go to, without starting the search from scratch for every search.

bulkInsert, intuitively…

To support `bulkInsert(B)`**…**



1. **Search for insertion sites, starting with the oldest entry in the batch**

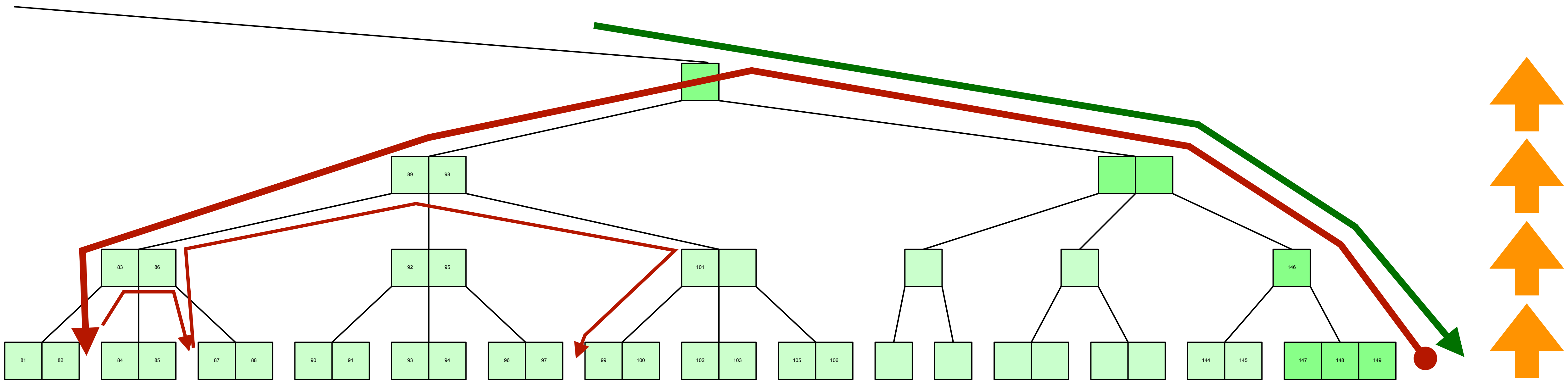
   Goal: Identify the nodes where the batch entries will go to, without starting the search from scratch for every search.

2. **Level-by-level pass up to add in new entries and split overflowing nodes**

   Goal: No more overflowing nodes and all new entries incorporated

bulkInsert, intuitively…

To support `bulkInsert(B)`**…**



1. **Search for insertion sites, starting with the oldest entry in the batch**

   Goal: Identify the nodes where the batch entries will go to, without starting the search from scratch for every search.

2. **Level-by-level pass up to add in new entries and split overflowing nodes**

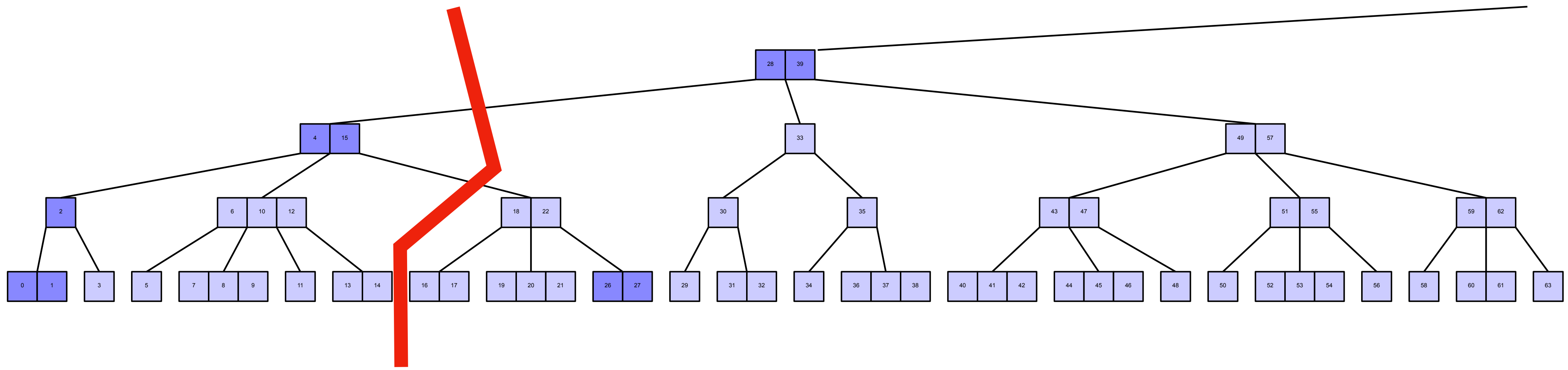
   Goal: No more overflowing nodes and all new entries incorporated

3. **A pass down to clean up nodes on the updated spine(s)**

   Goal: Fix the spine(s) and repair spine aggregates

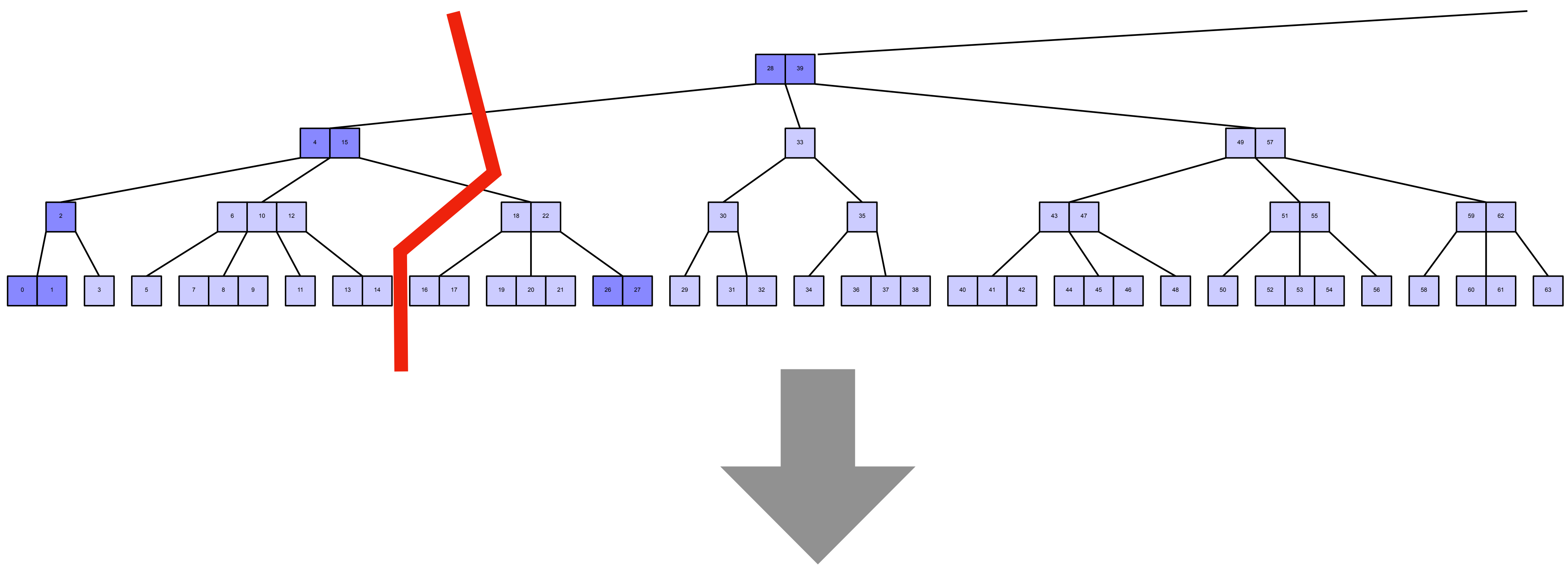
Implementation Consideration

`bulkEvict(t)` is about to remove m data items…



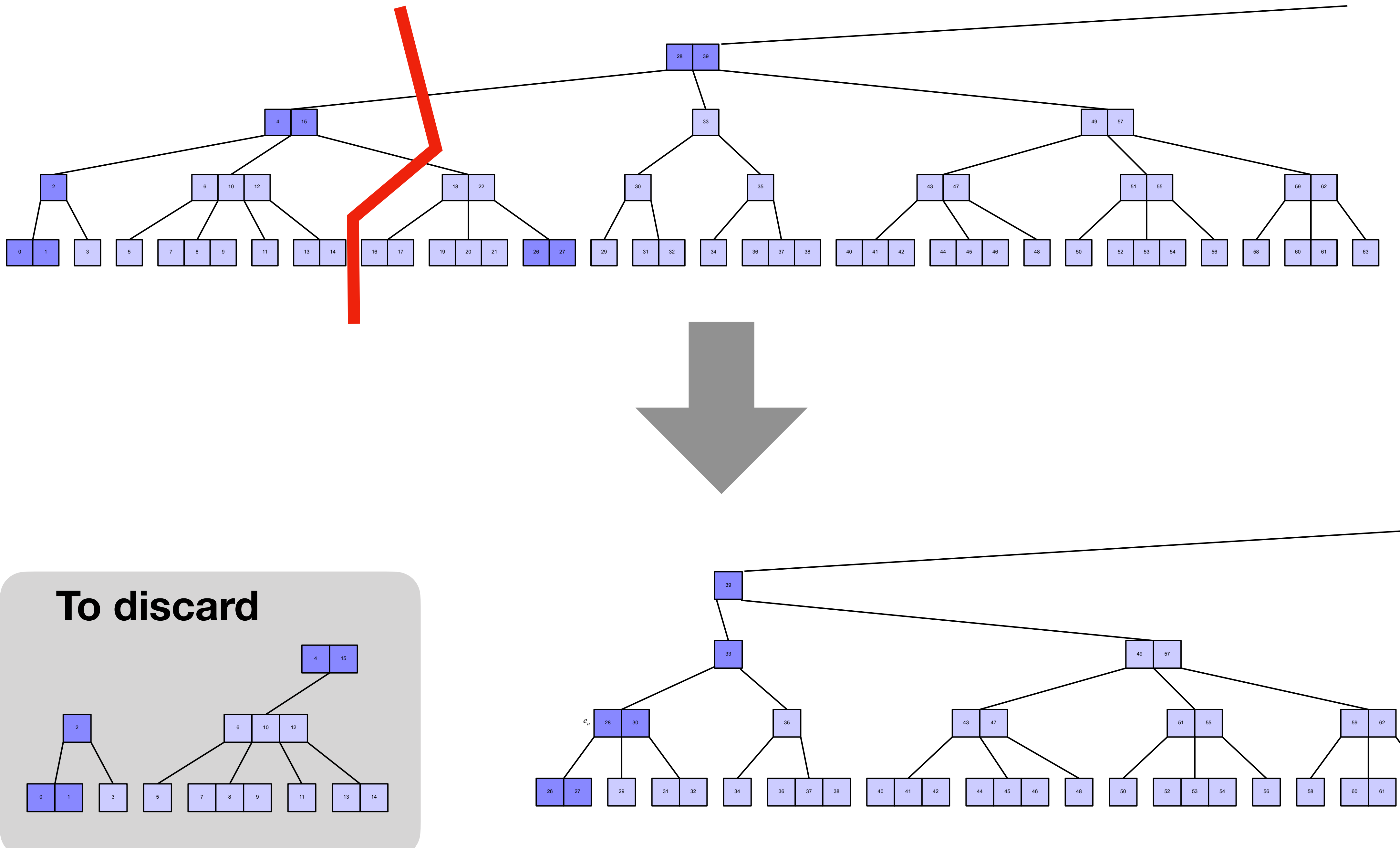
Implementation Consideration

`bulkEvict(t)` is about to remove m data items…



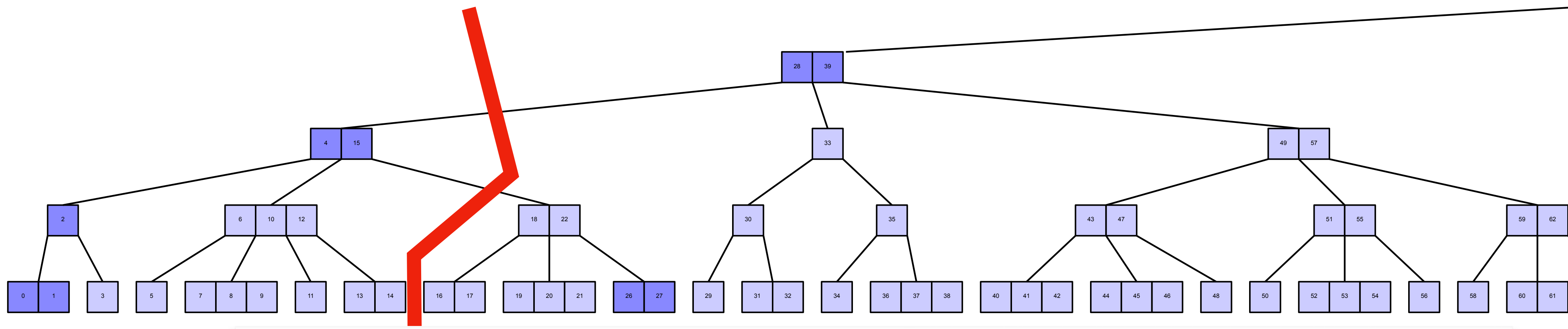
Implementation Consideration

`bulkEvict(t)` is about to remove m data items…

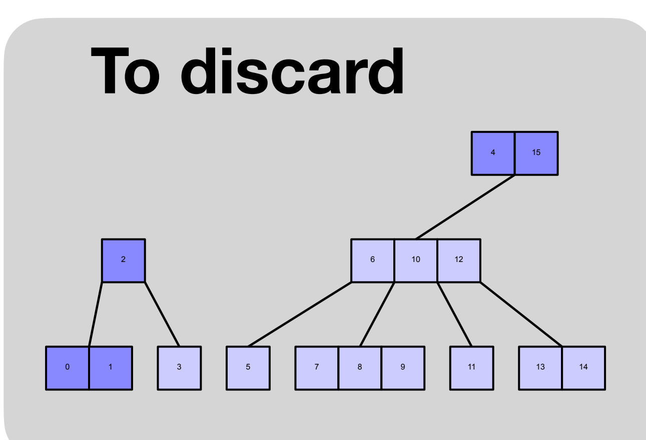


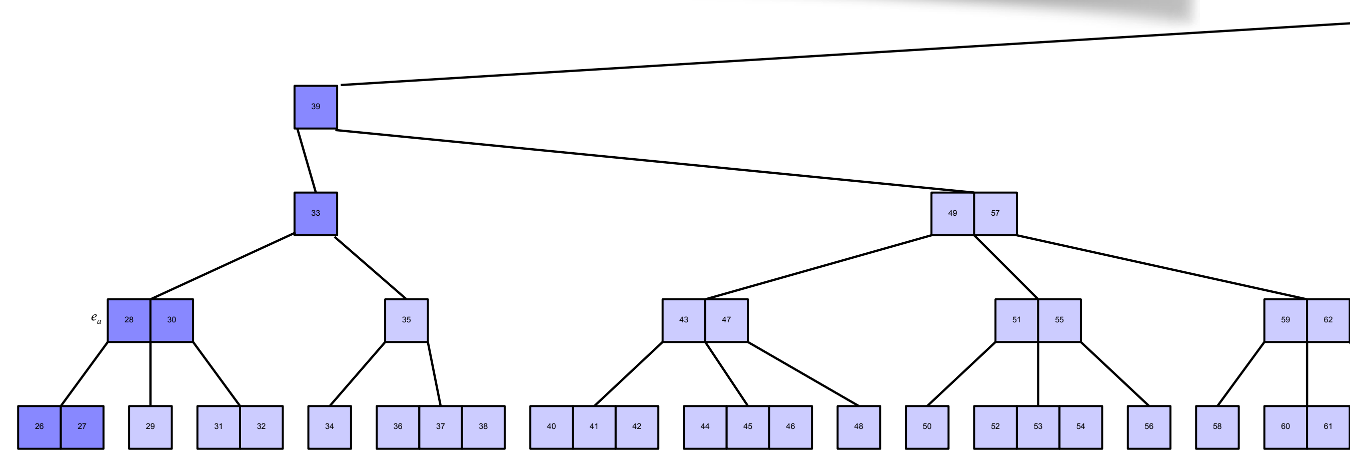
Implementation Consideration

`bulkEvict(t)` is about to remove m data items…



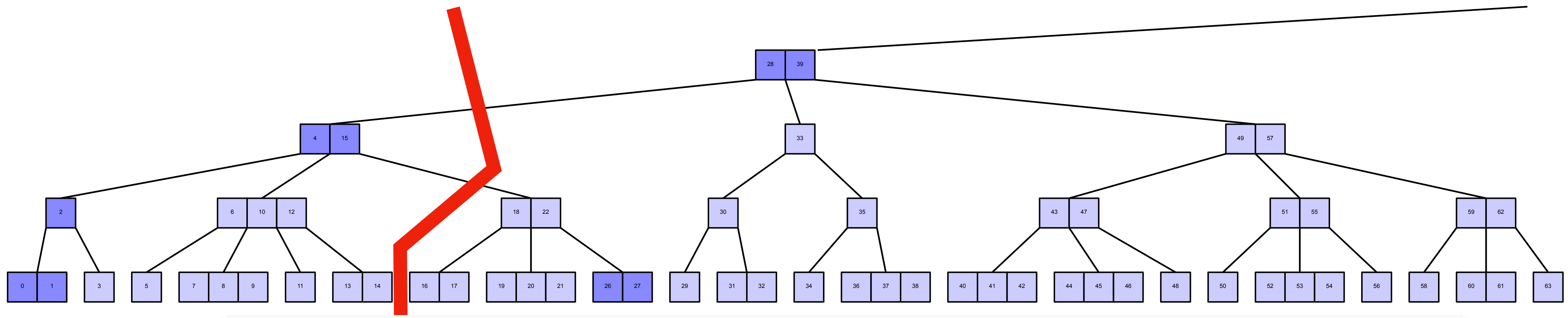
Concern: When `bulkEvict` removes m data items, it needs to discard $O(m)$ nodes but can't afford to eagerly free them





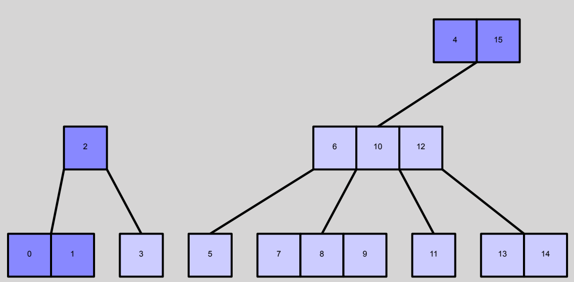
Implementation Consideration

`bulkEvict(t)` is about to remove m data items…

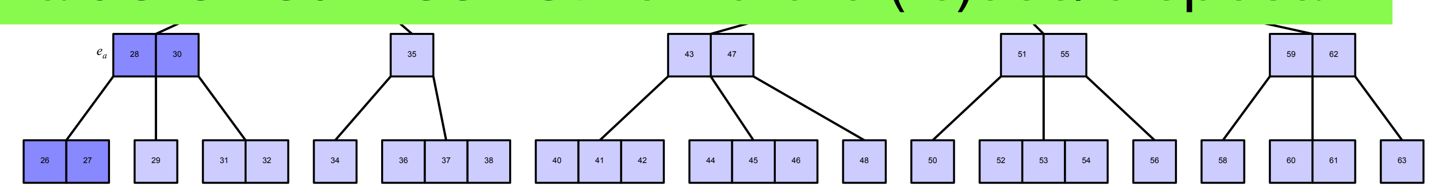


Concern: When `bulkEvict` removes m data items, it needs to discard $O(m)$ nodes but can't afford to eagerly free them

To discard



Solution: Only eagerly free those on boundary (same as the search cost) and store their children in a ***deferred free list*** for future (re)use/disposal



Experimental Analysis

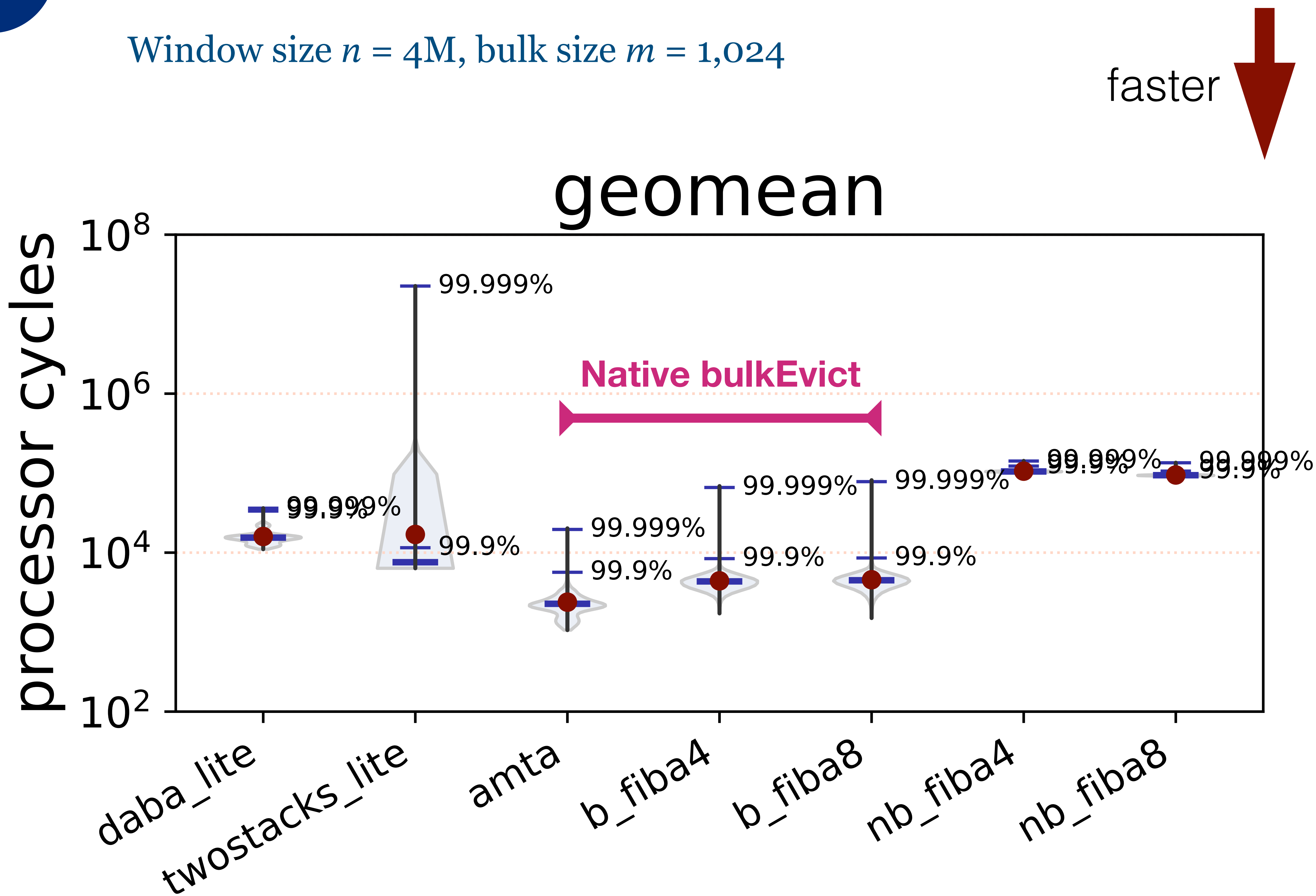
1. How does *native* `bulkEvict` alter the latency profile?
2. How does *native* `bulkInsert` alter the latency profile?
3. Does it matter on real-world data with wildly-fluctuating window sizes and out-of-order levels?

Lang/ Compiler	C++, g++ 9.4.0 using -O3
OS	Ubuntu Linux 20.04.5, Kernel 5.4.0
Machine	Intel Xeon 4310 @ 2.1Ghz (exp. run single-threaded)

1

How does *native* bulkEvict alter the latency profile?

Window size n = 4M, bulk size m = 1,024

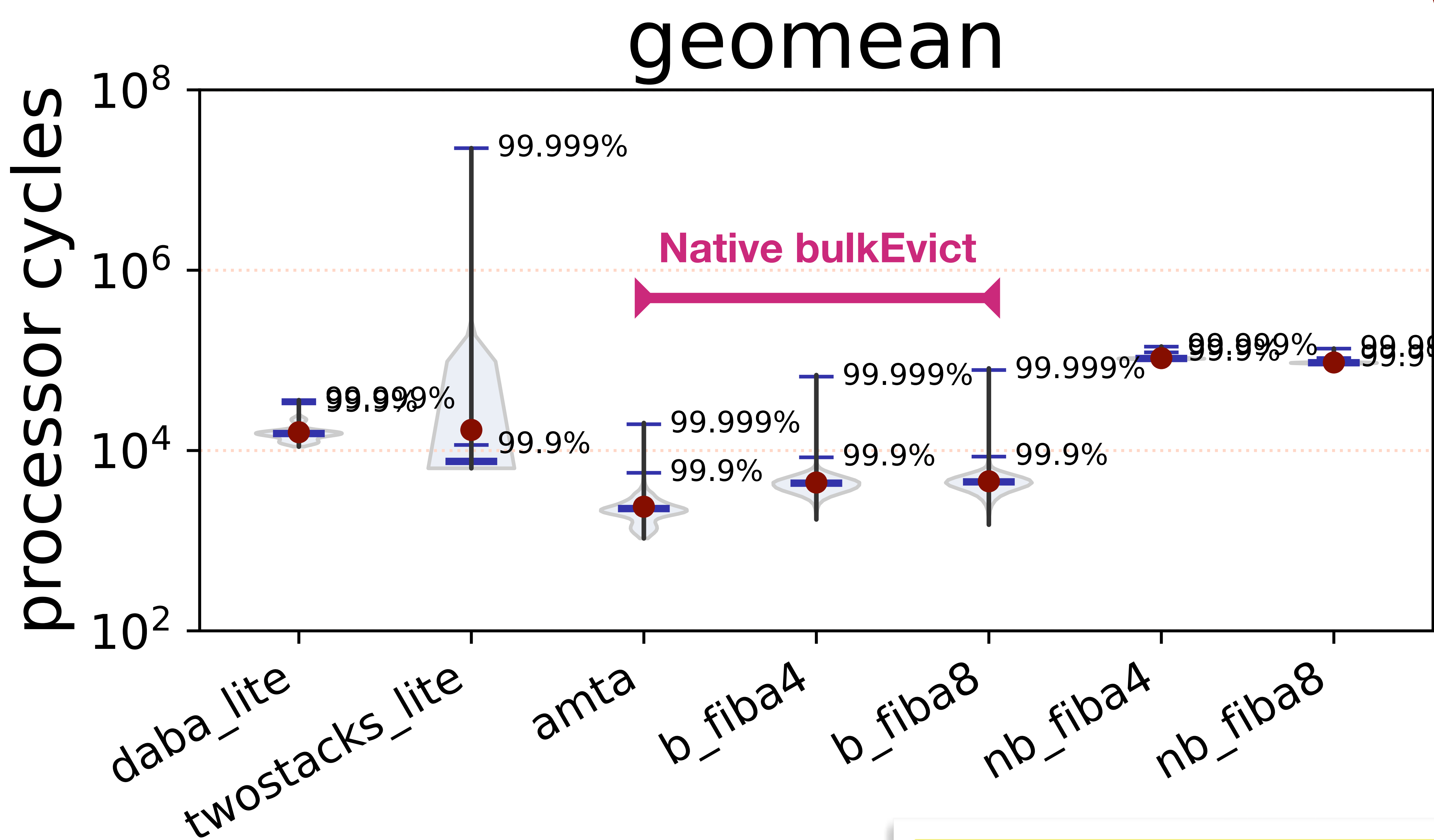


1

How does *native* bulkEvict alter the latency profile?

Window size n = 4M, bulk size m = 1,024

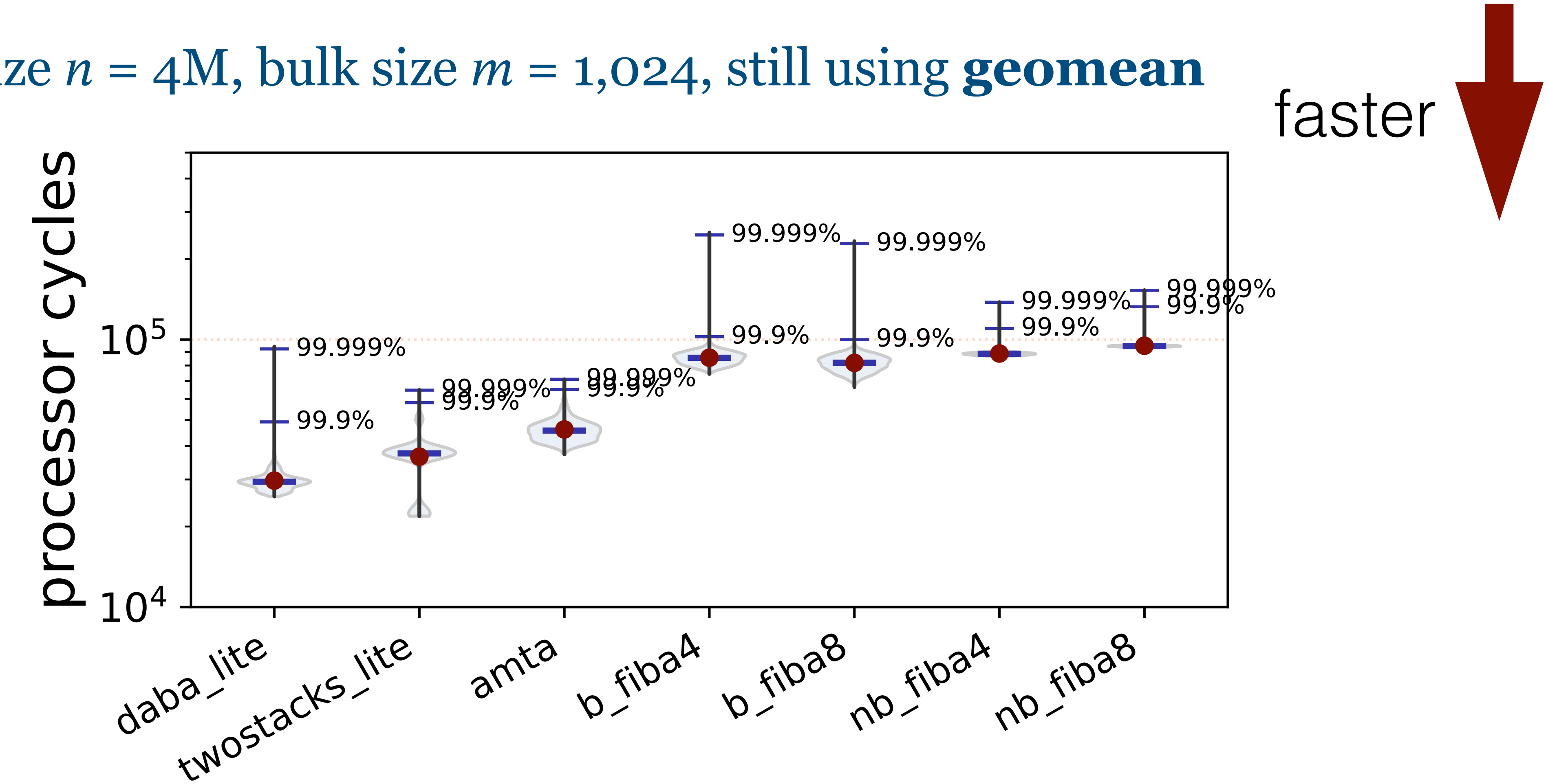
faster



..translates to improved throughput as well

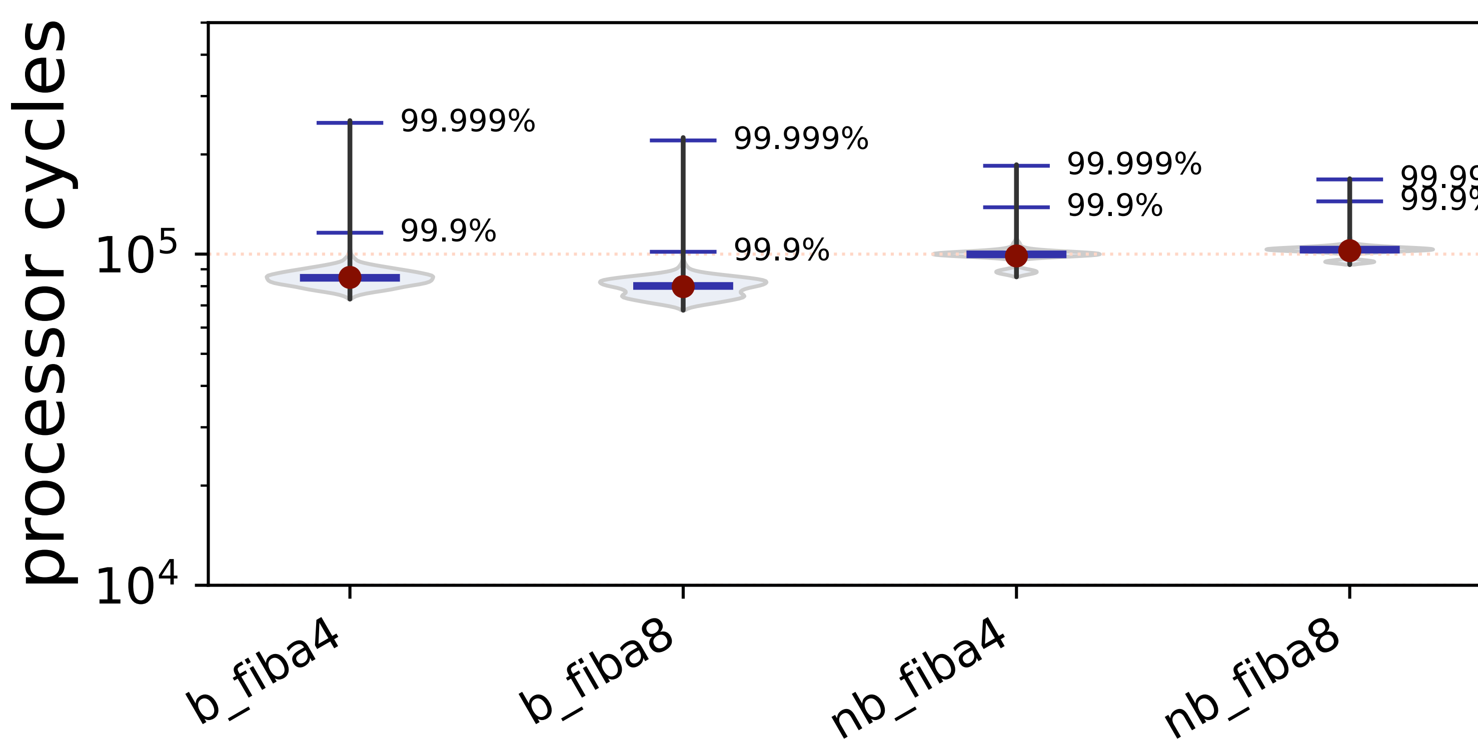
2 How does *native* bulkInsert alter the latency profile?

Window size n = 4M, bulk size m = 1,024, still using **geomean**



FIFO (in-order)

Out-of-order
d=1,024

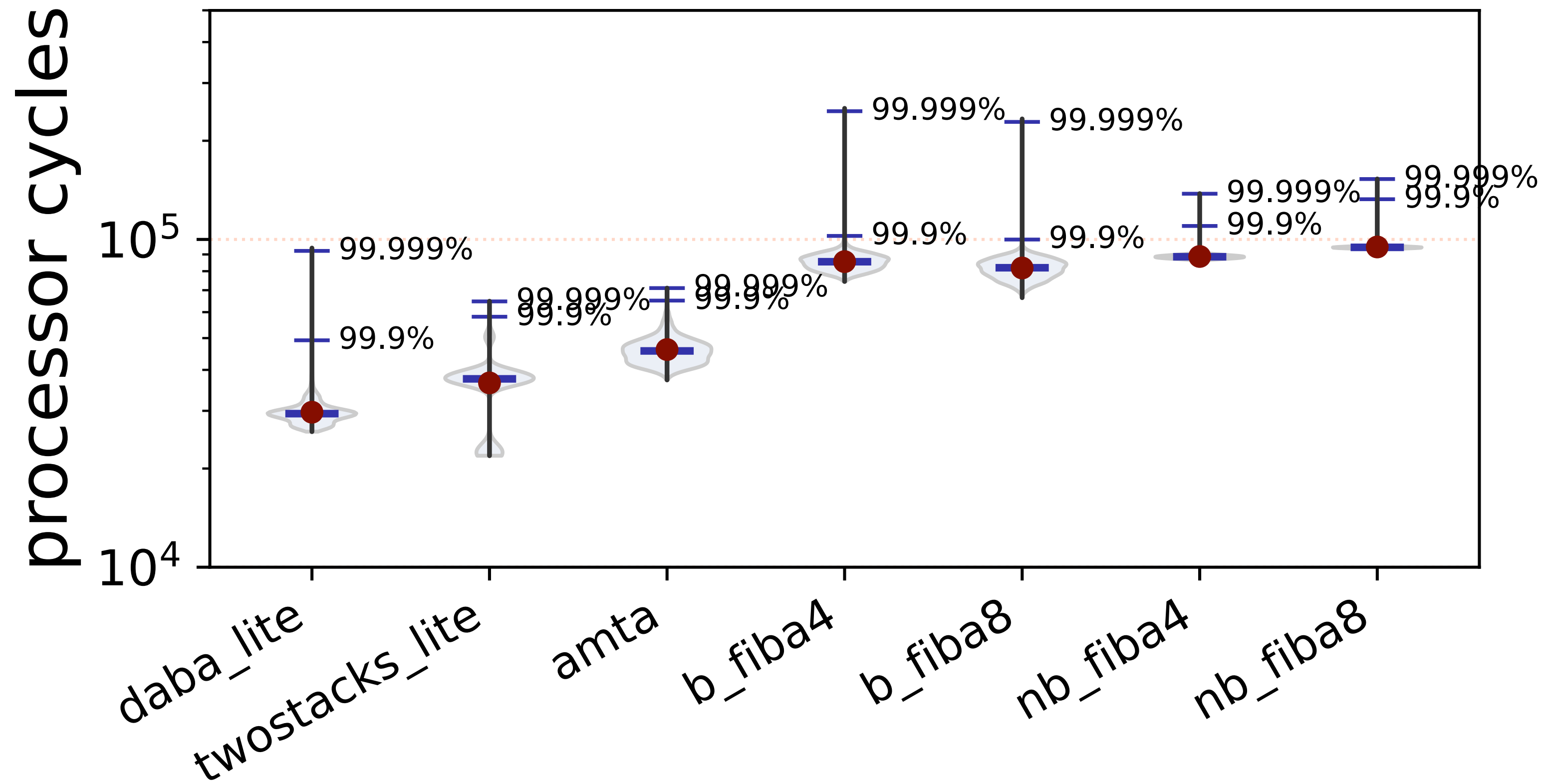


2 How does *native* bulkInsert alter the latency profile?

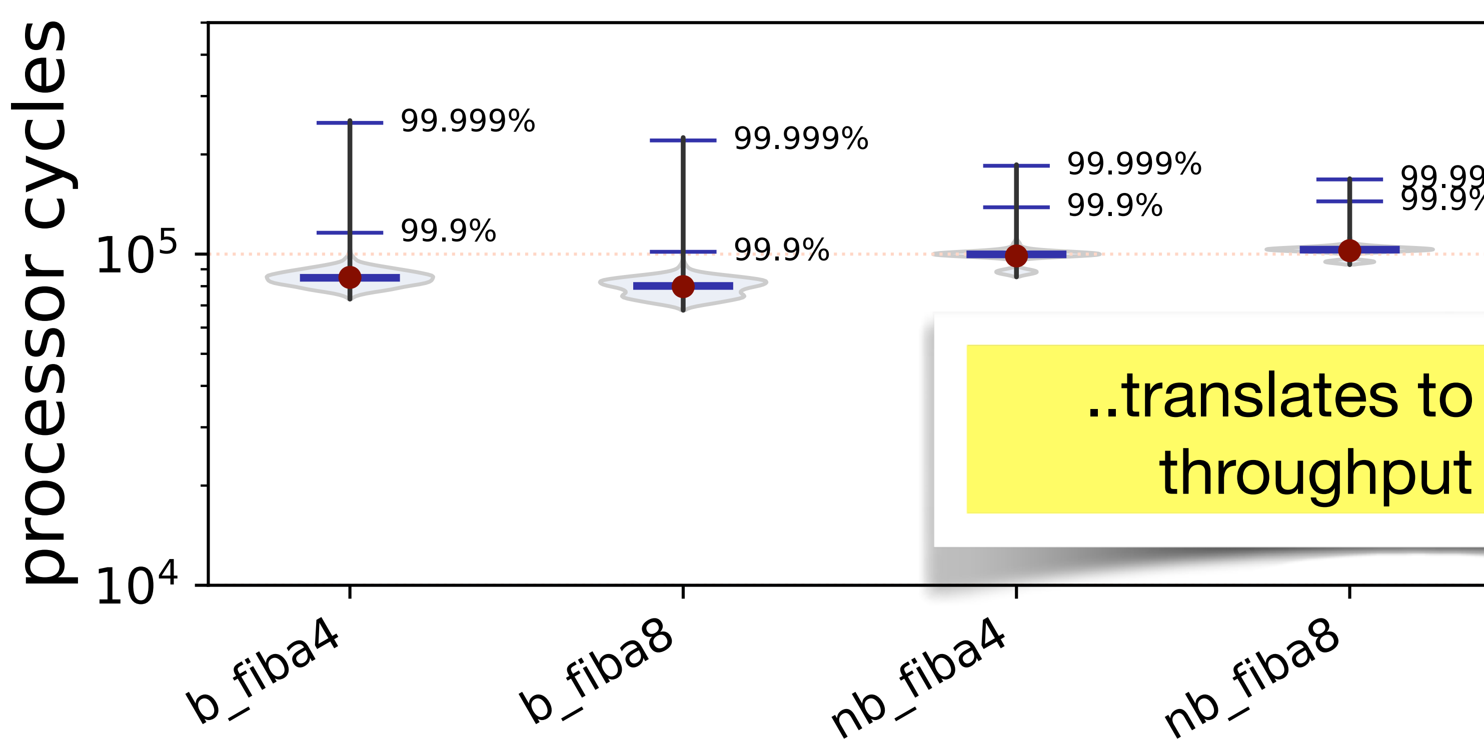
Window size n = 4M, bulk size m = 1,024, still using **geomean**

faster

FIFO (in-order)



Out-of-order
d=1,024

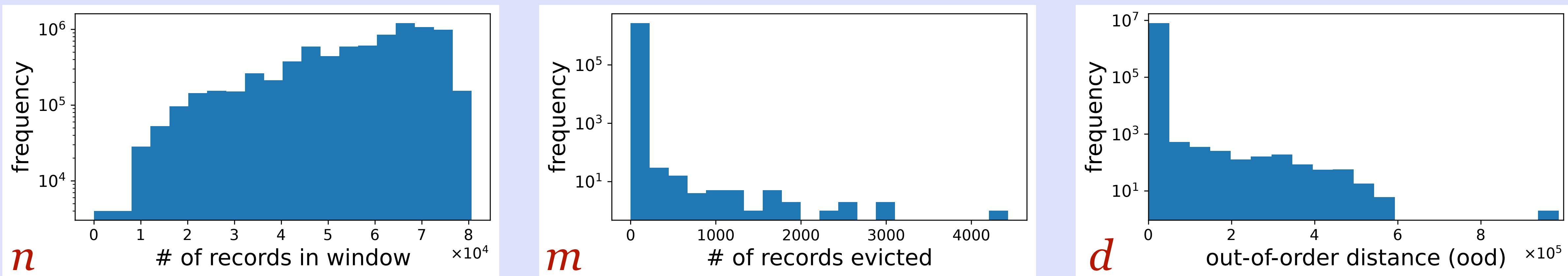


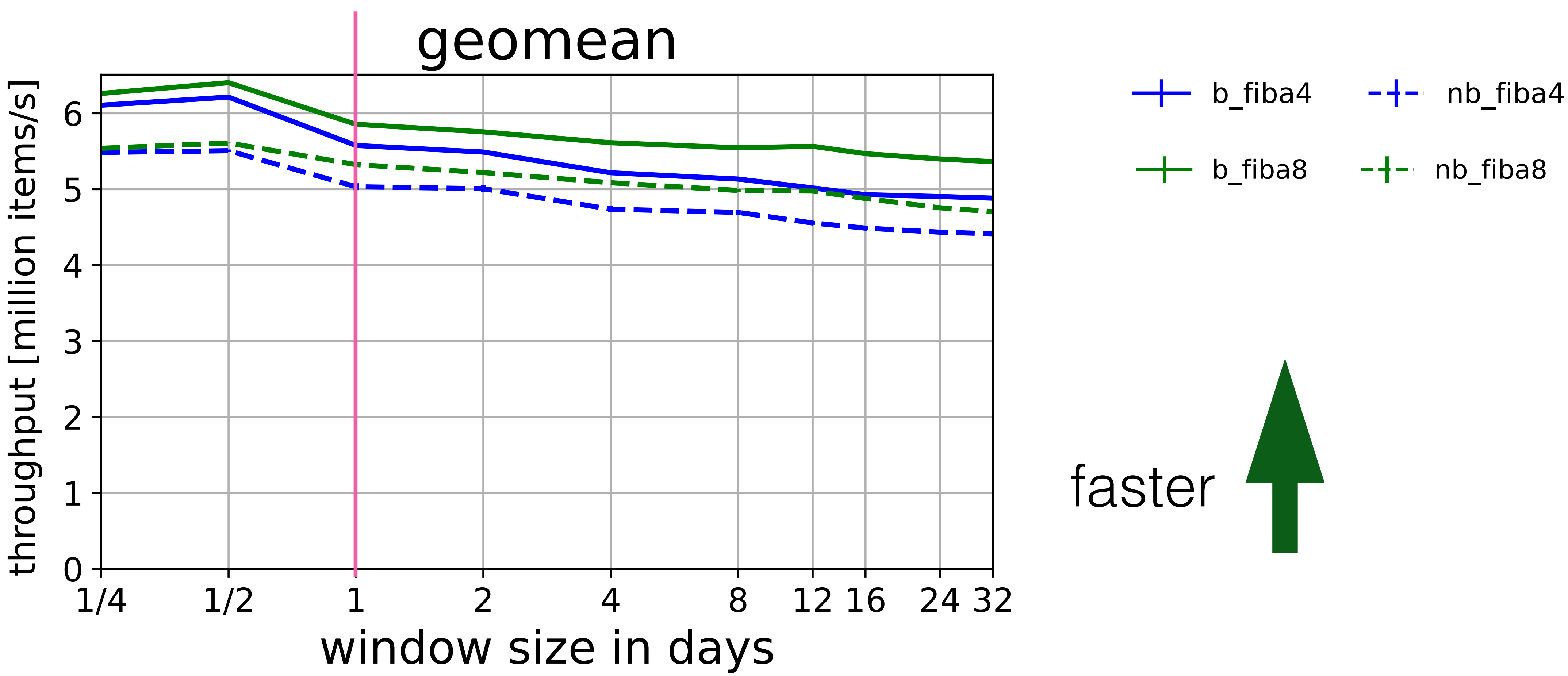
..translates to improved throughput as well

3 Does it matter on real-world data with wildly-fluctuating window sizes and out-of-order levels?

Window Size @ 1 day

NYC Citi Bike Data (Aug - Dec 2018)





Bulk FiBA: Take-Away Points

- **Efficient bulk** eviction/insertion (asymptotically better)
- Retain FiBA's efficient tuple-at-a time + queries
- Plenty more in the paper: proof(s), Flink experiments, other benchmarks, n = 1Billion, etc.
- Code is public on GitHub:
  `https://github.com/IBM/sliding-window-aggregators`

Scan me for the paper