BOSTON UNIVERSITY

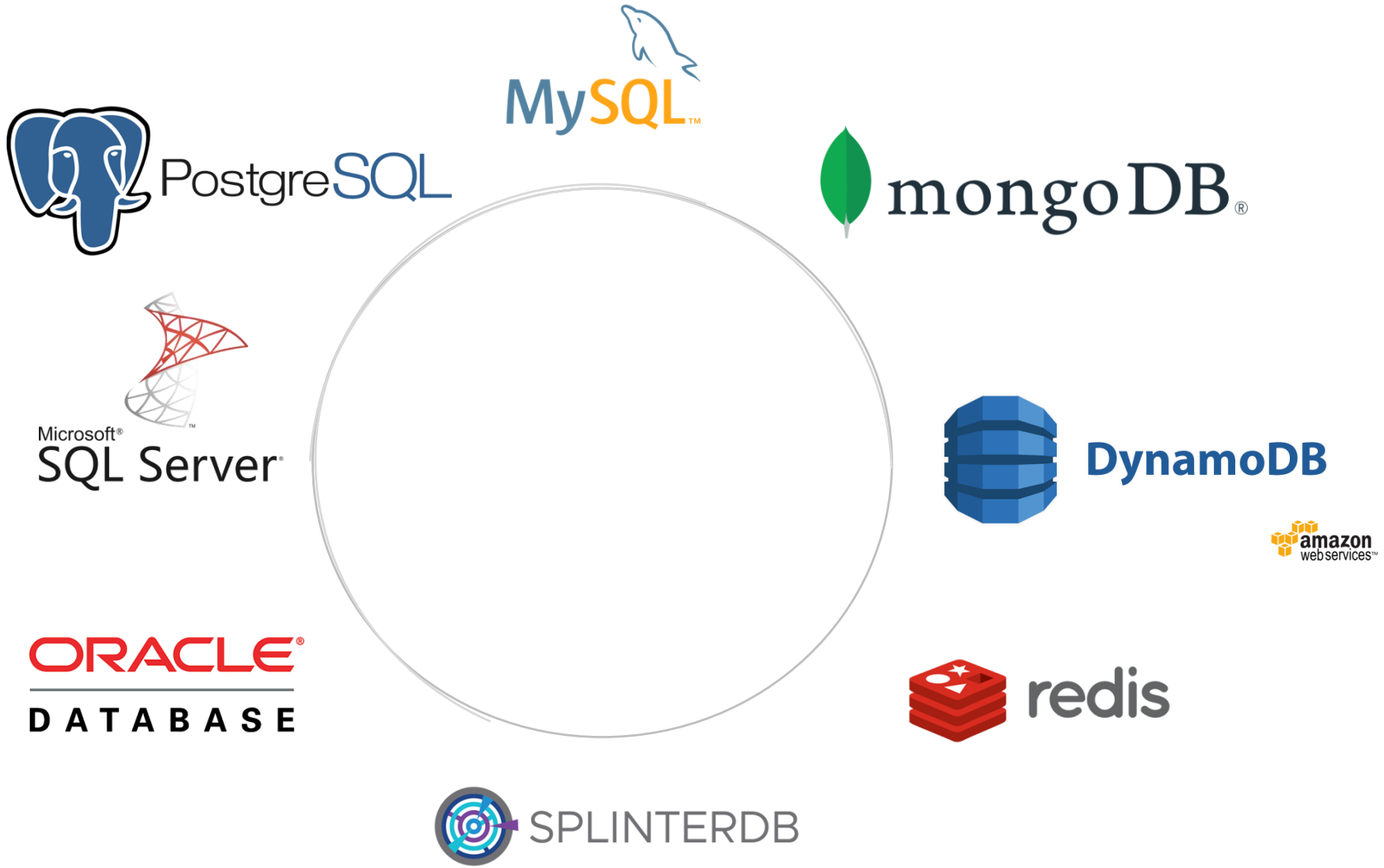
CS 561: Data Systems Architectures

Indexing for Near-Sorted Data

Prof. Manos Athanassoulis

https://bu-disc.github.io/CS561/

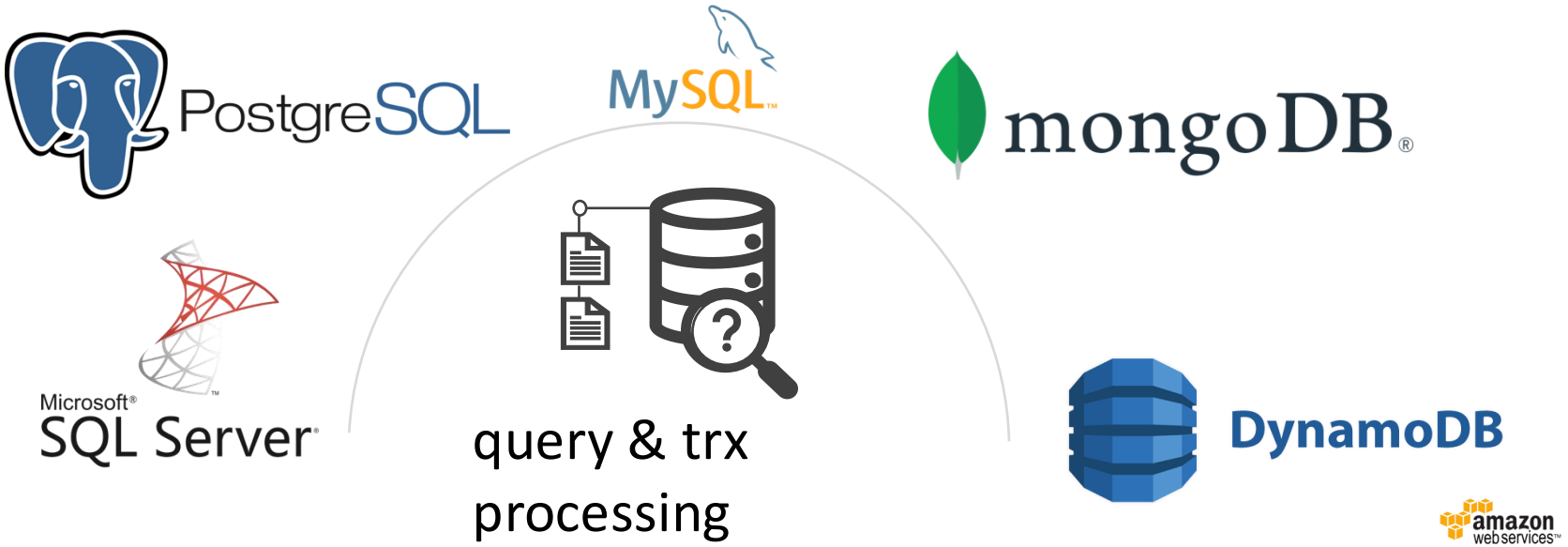
Data Systems use Indexes!

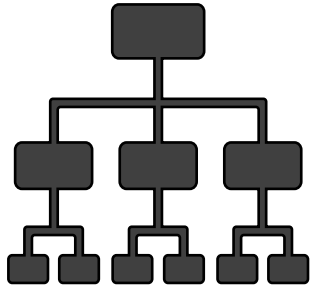


Data Systems use Indexes!



Data Systems use Indexes!





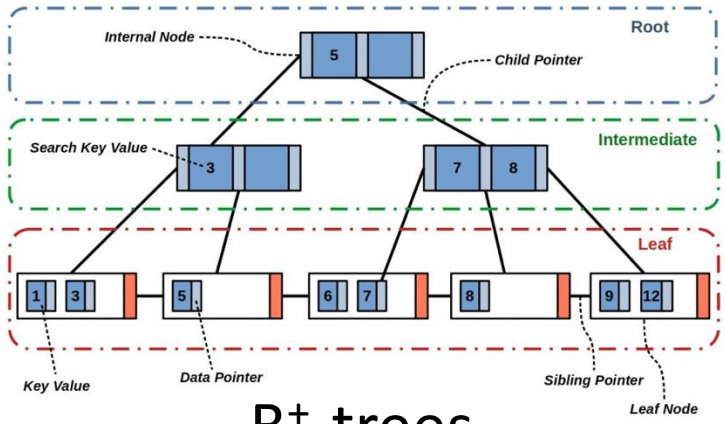
organize data



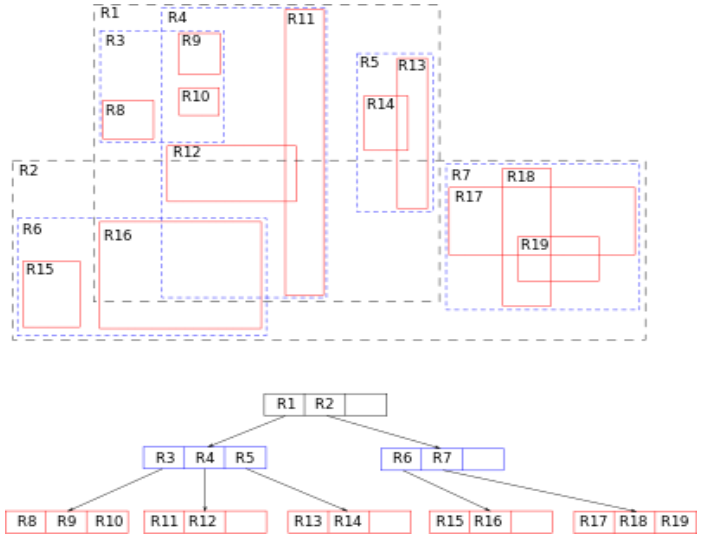


accelerate access time

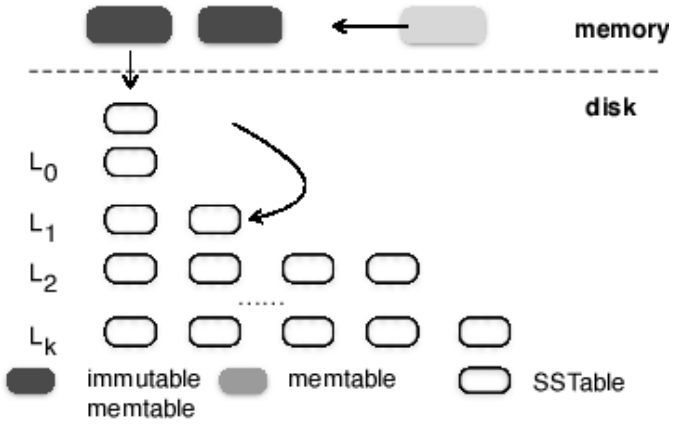
Indexes in Data Systems



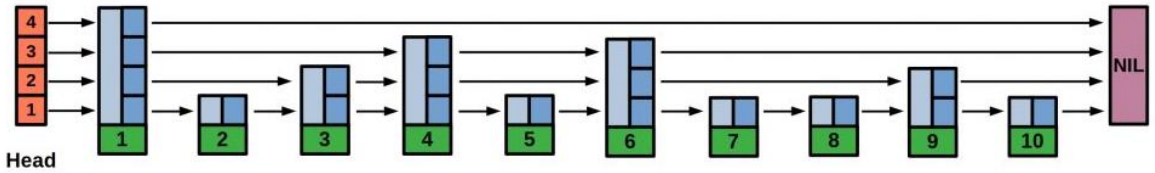
B+-trees



R-trees

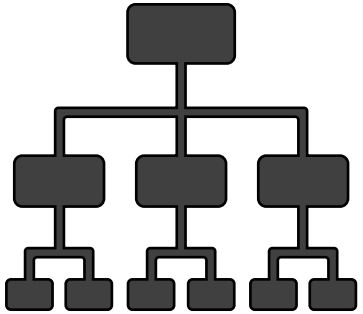


LSM-trees

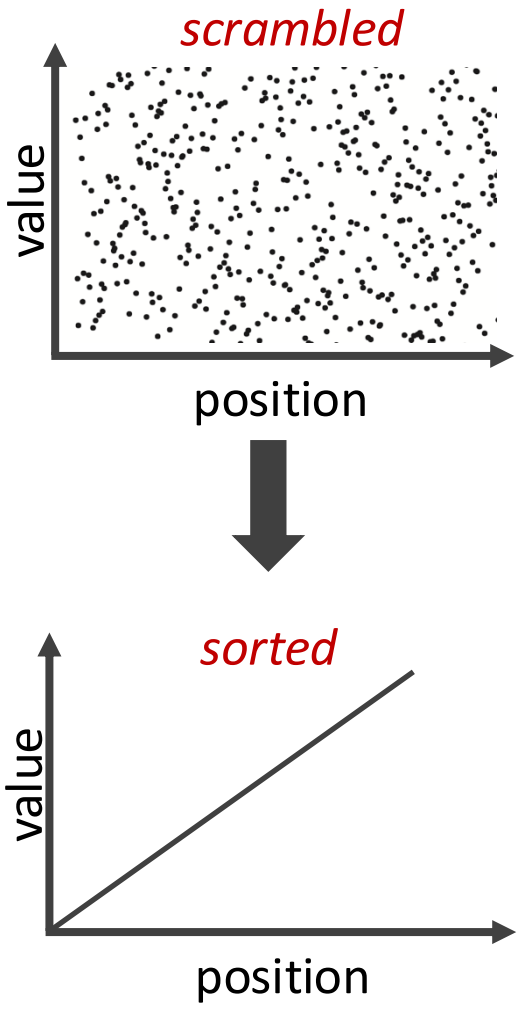


Skip lists

What is an Index? *Indexing Adds Structure*

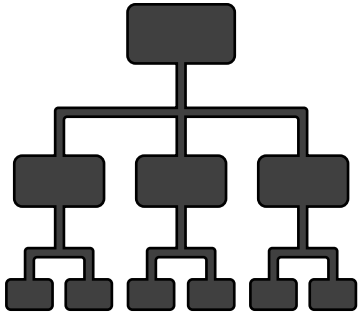


organize data

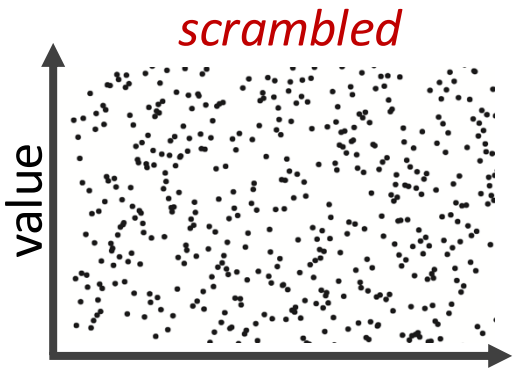


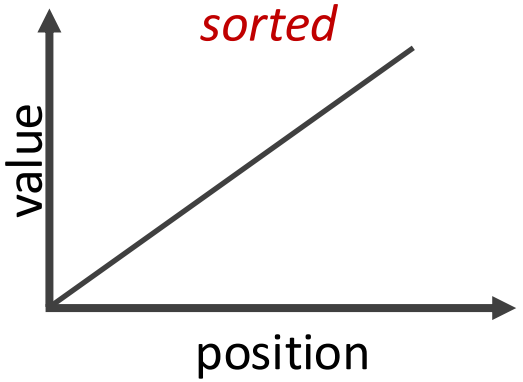
The process of ***"inducing sortedness"*** to otherwise, unsorted data

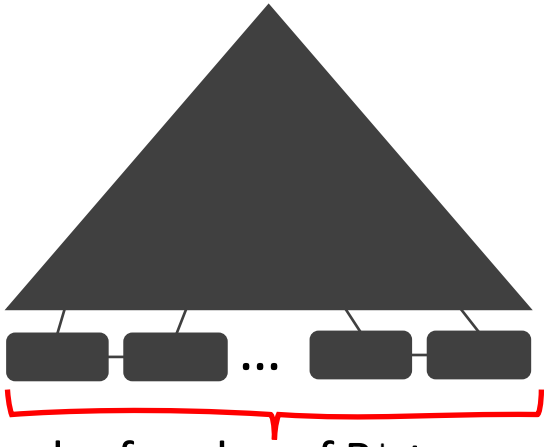
Indexing Adds Structure



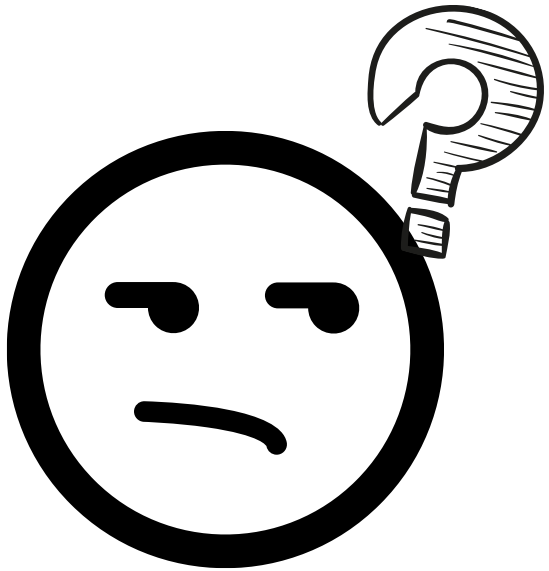
organize data





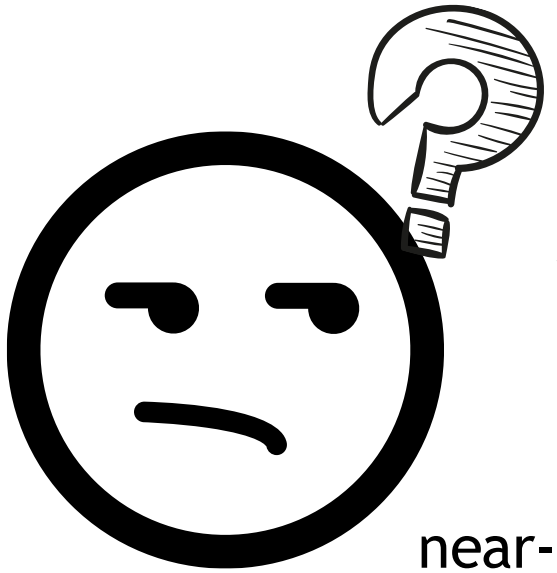






What if data already has some structure?

BOSTON
UNIVERSITY



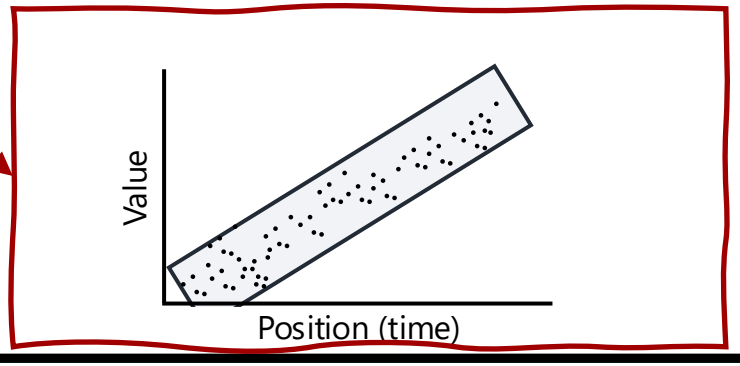
What if data already has some structure?

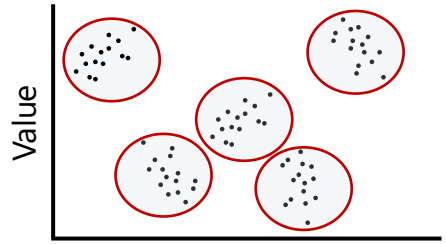
near-sortedness

locality

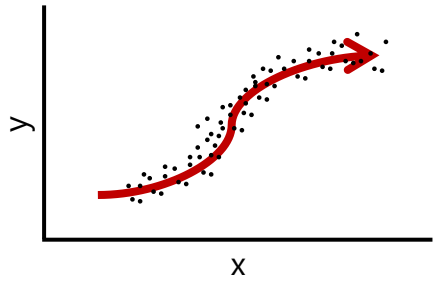
Current Focus

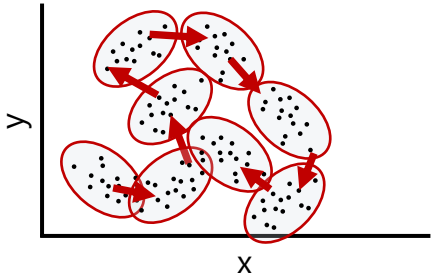
indexing one column (dimension)



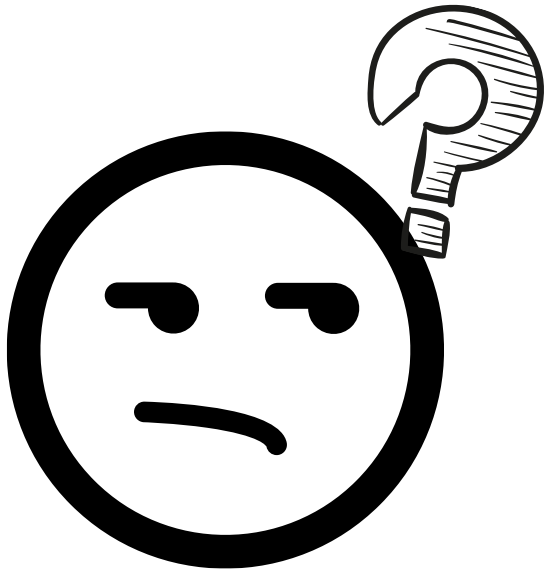


multi-dimensional indexing

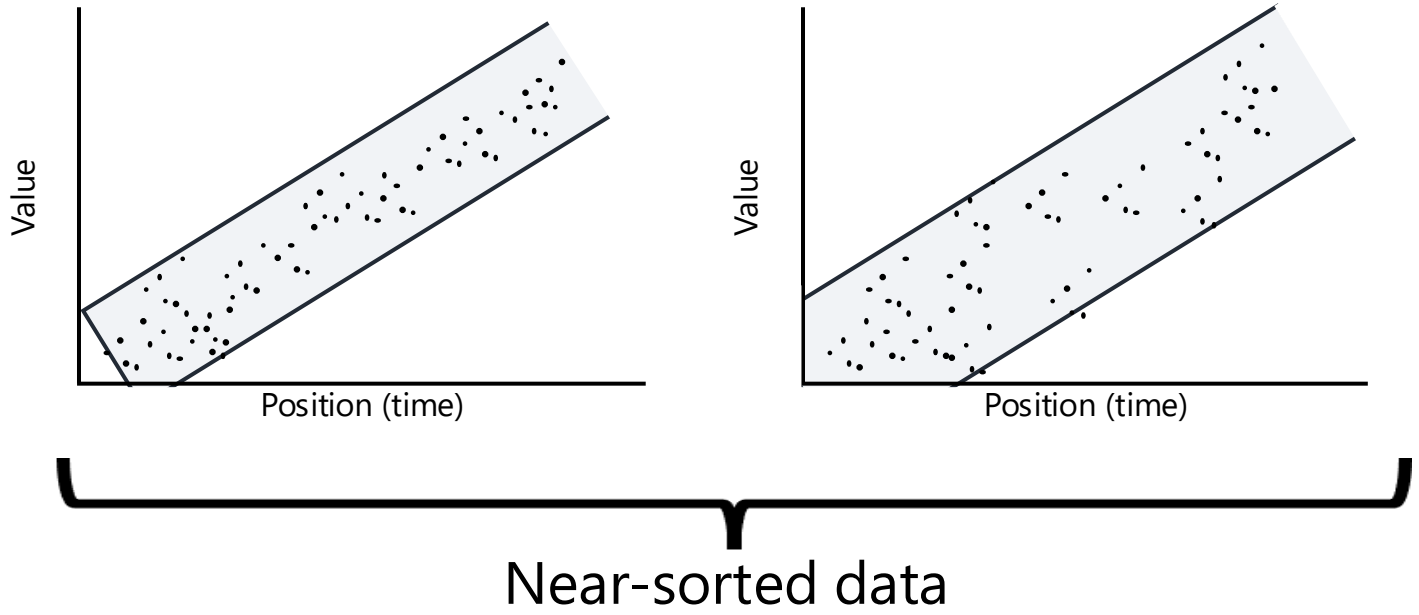




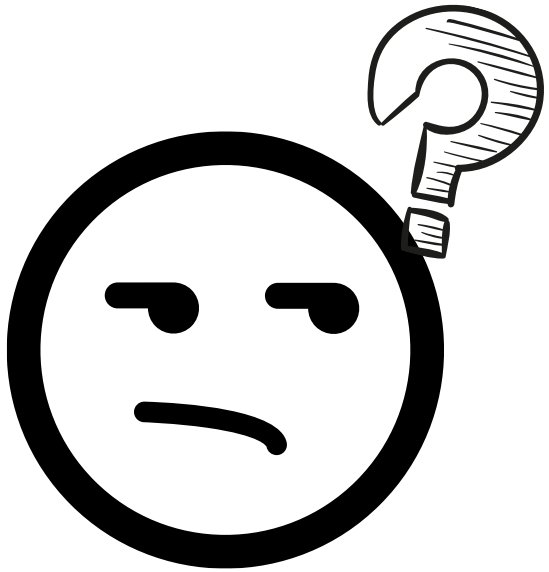
BOSTON UNIVERSITY



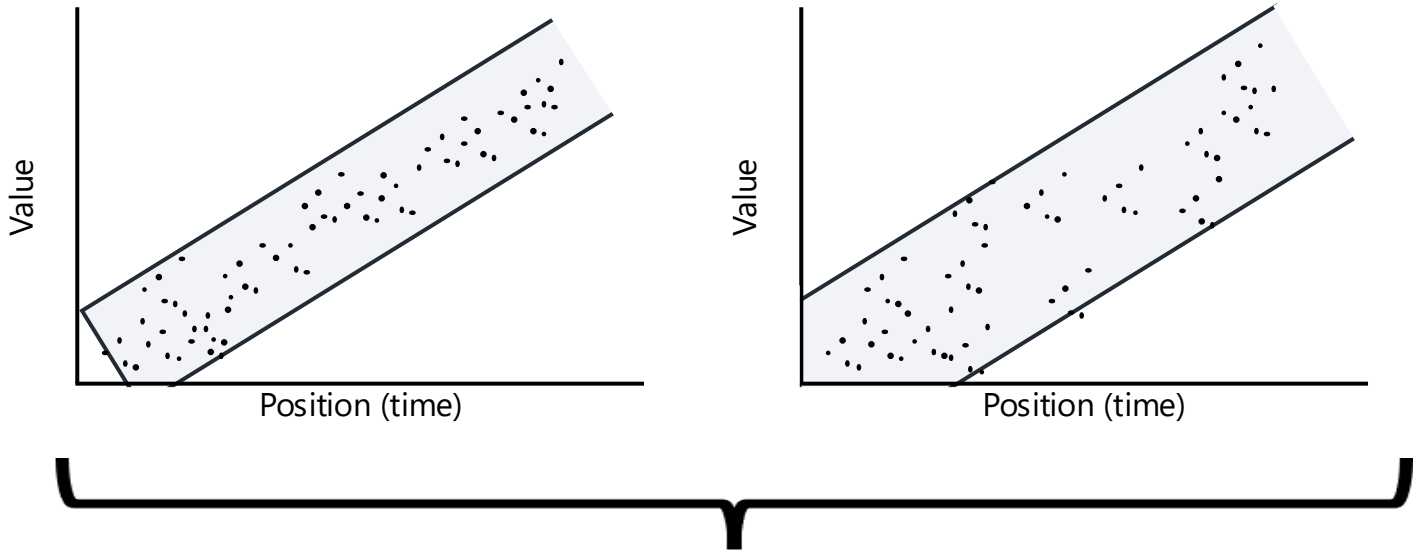
What if data already has some structure?



BOSTON UNIVERSITY



What if data already has some structure?



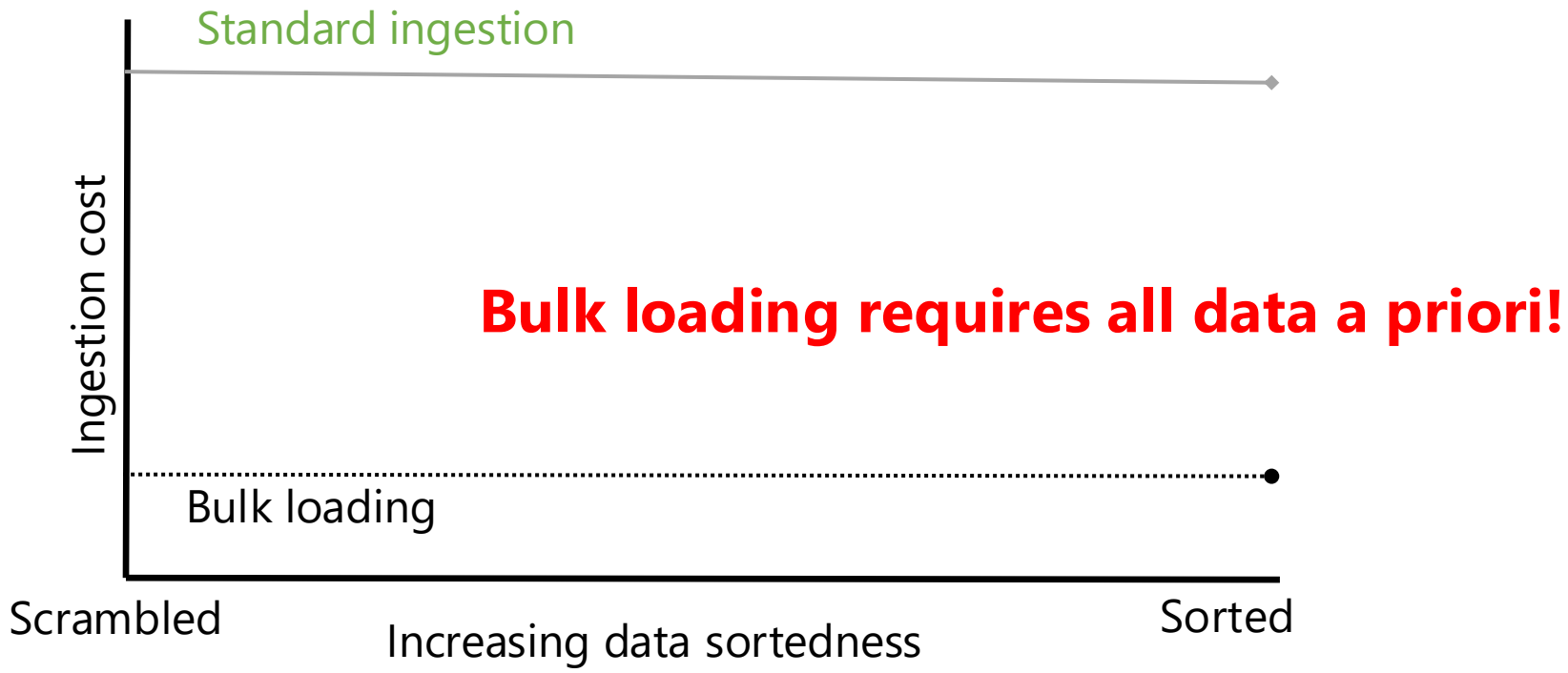
≈ treated same as unstructured data!

Irrespective of Sortedness, Same Ingestion Performance



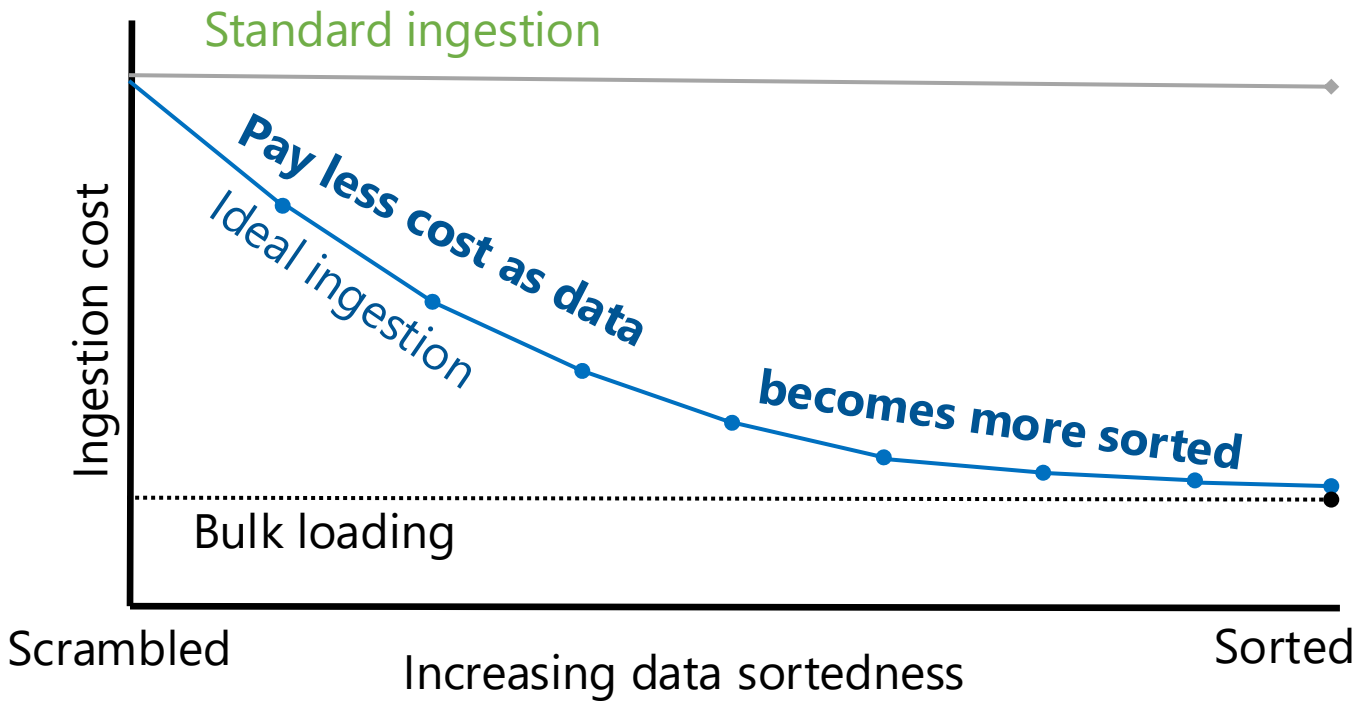
BOSTON UNIVERSITY

Are There Faster Alternatives?



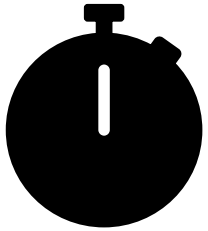
BOSTON UNIVERSITY

Ideally, Higher Sortedness Should Lead to Faster Ingestion



BOSTON UNIVERSITY

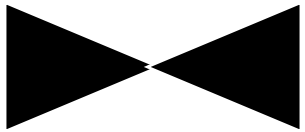
Near-Sorted Data is Frequently Found



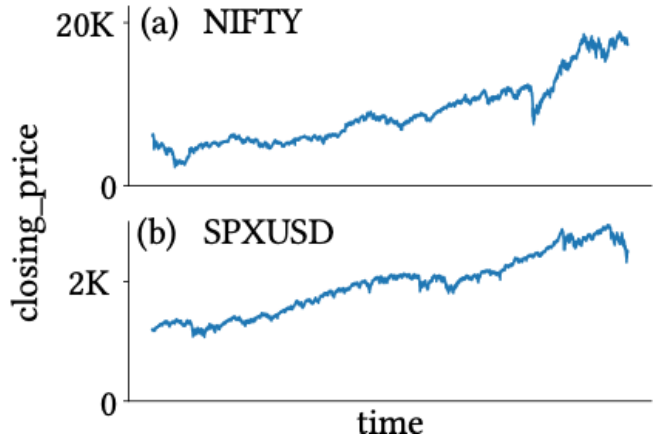
Time Series

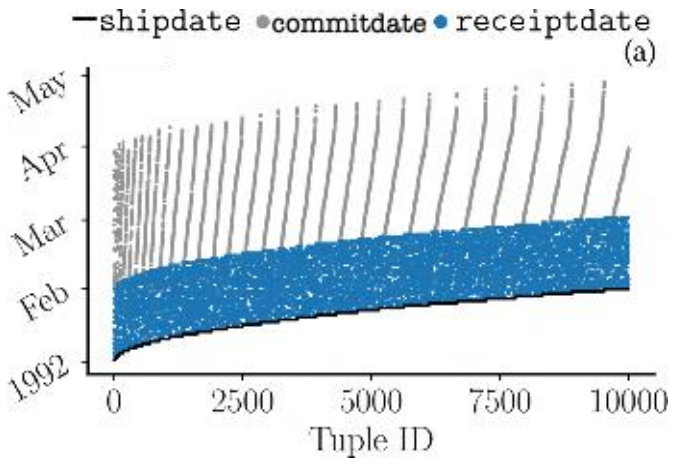


Stock market



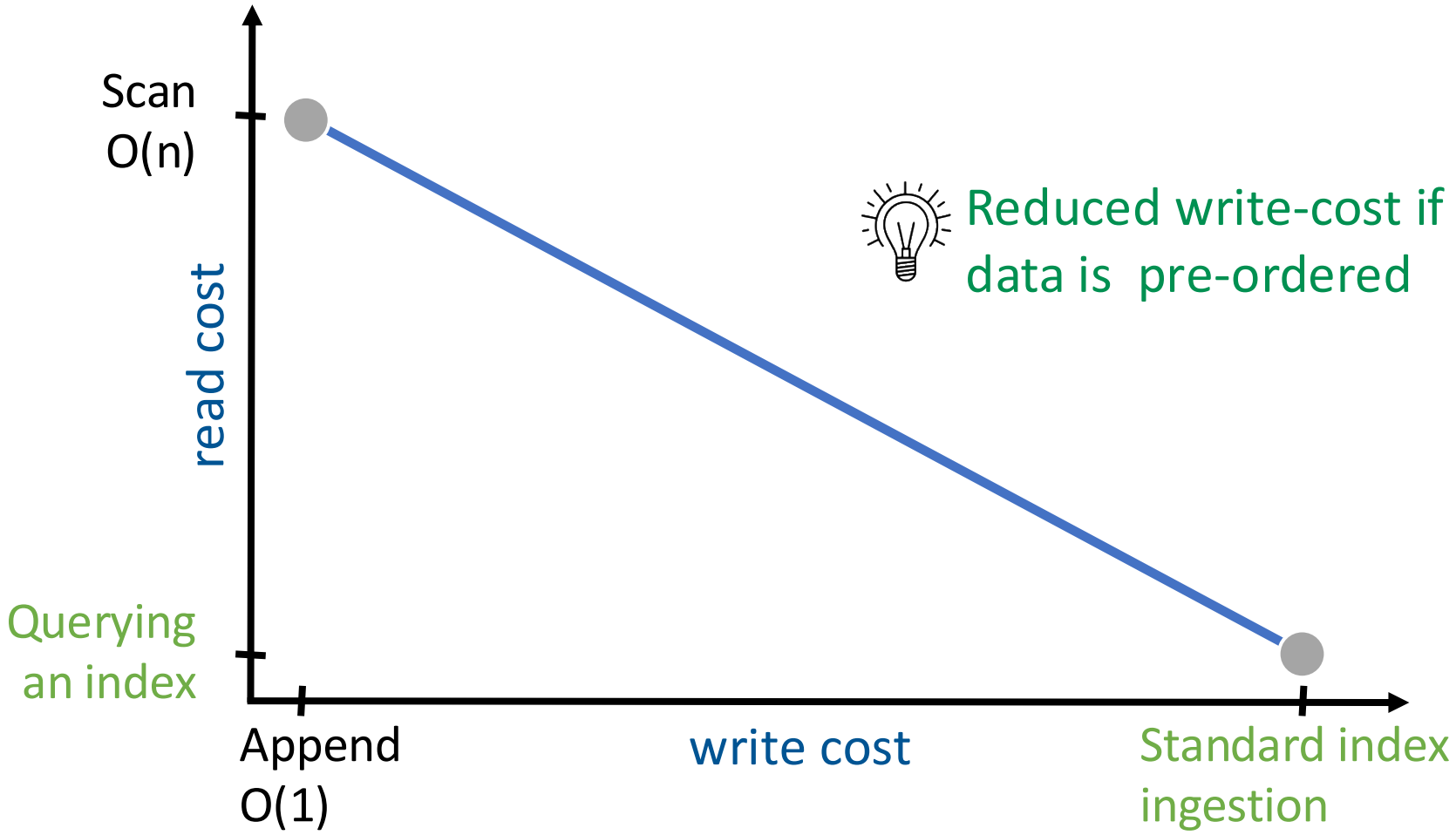
Join/query



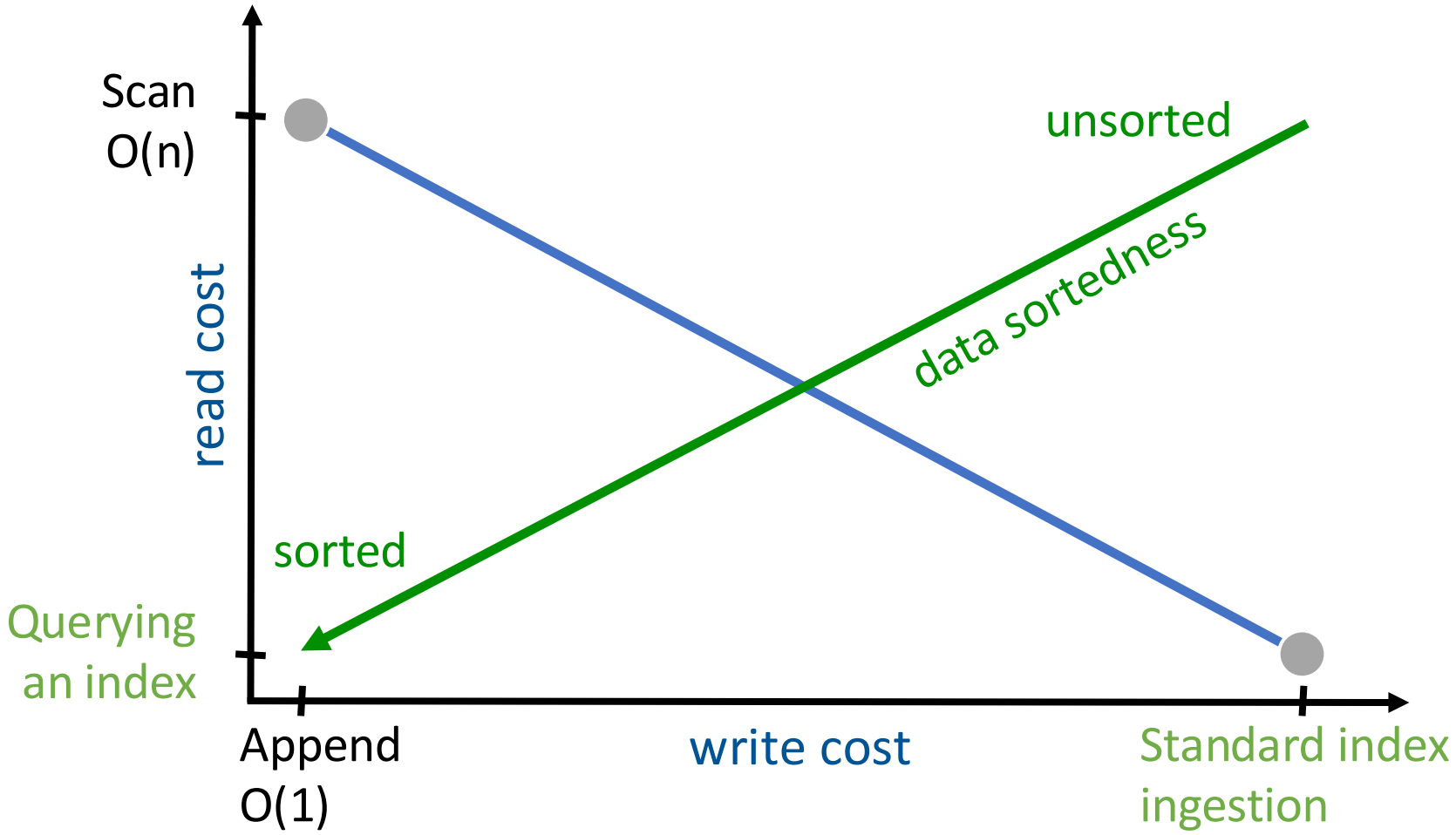


BOSTON UNIVERSITY

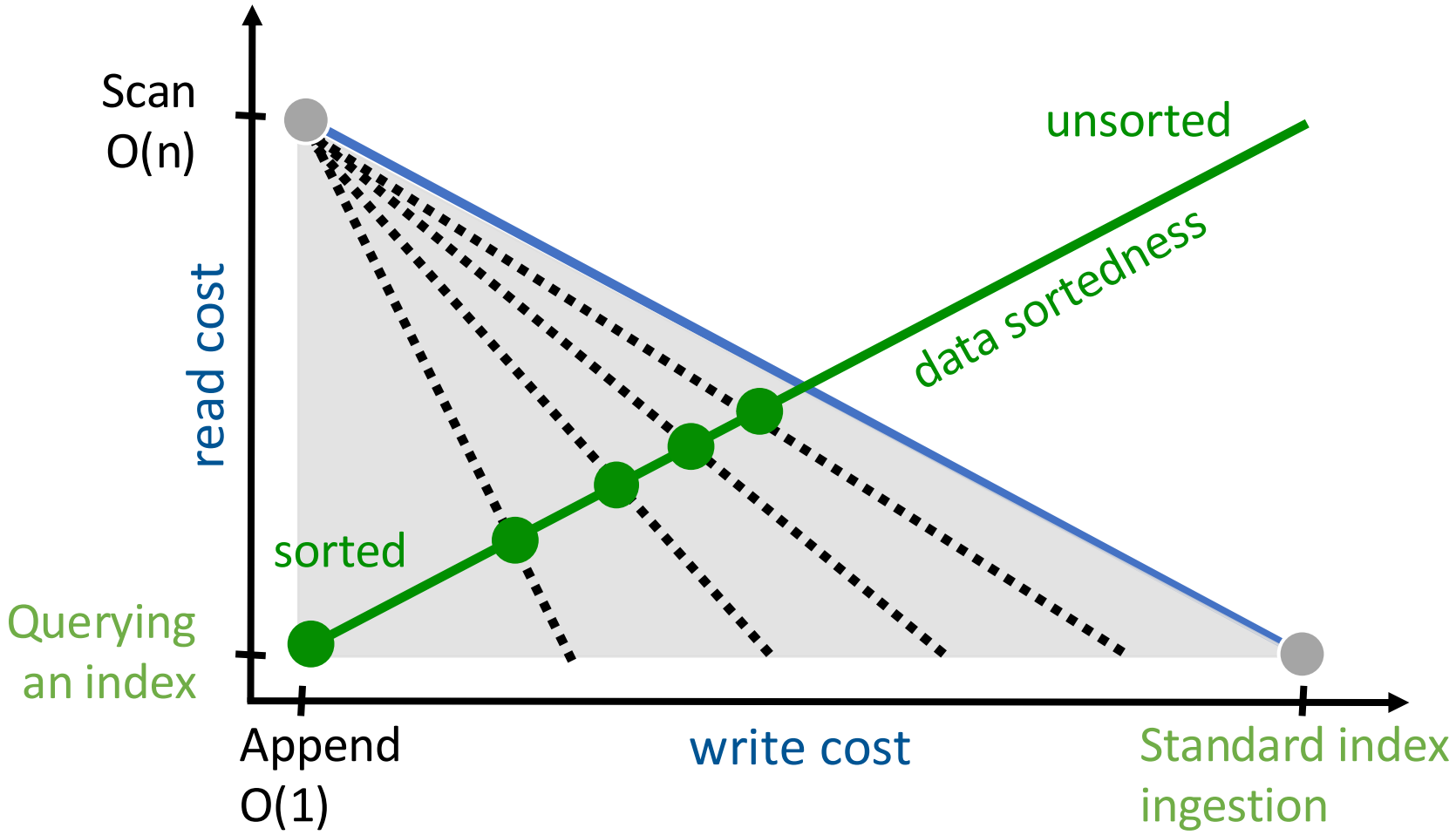
Vision for Sortedness-Aware Indexing



Vision: Sortedness-Aware Indexes



Vision: Sortedness-Aware Indexes



Agenda

Introduction

Vision

Sortedness Metrics [TPCTC'22]

Sortedness Aware (SWARE) Indexing [ICDE'23]

Quick Insertion Tree (QuIT) [EDBT'25]

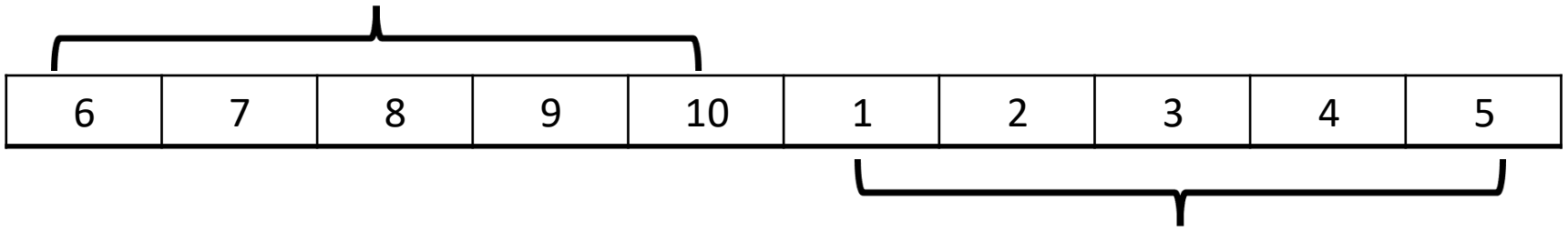
Sortedness for Learned Indexes, LSM, Joins [DBTest'24]

Quantifying Data Sortedness

Metric	
Inversions	# pairs in incorrect order
Runs	# increasing contiguous subsequences
Exchanges	least # swaps needed to establish total order

Quantifying Data Sortedness

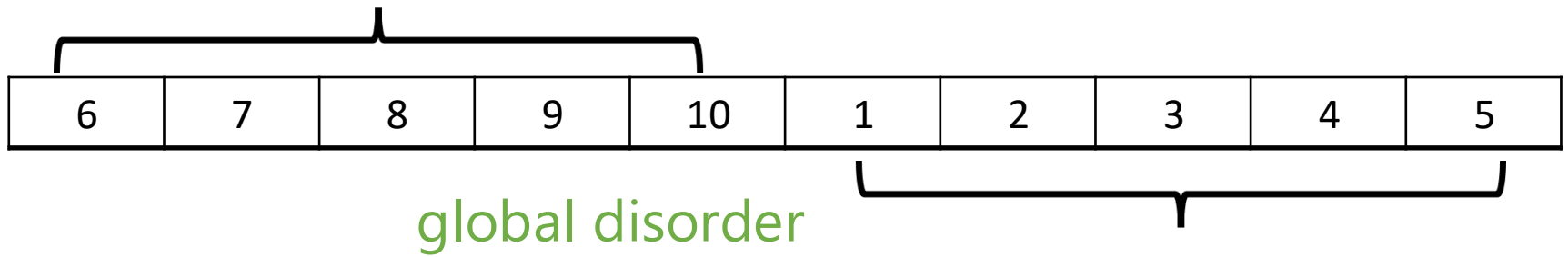
Metric	
Inversions	# pairs in incorrect order
Runs	# increasing contiguous subsequences
Exchanges	least # swaps needed to establish total order



BOSTON UNIVERSITY

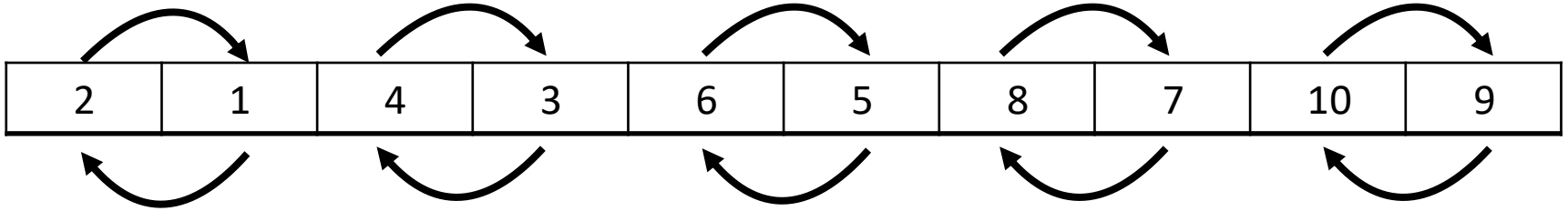
Quantifying Data Sortedness

Metric	
Inversions	# pairs in incorrect order
Runs	# increasing contiguous subsequences
Exchanges	least # swaps needed to establish total order



Quantifying Data Sortedness

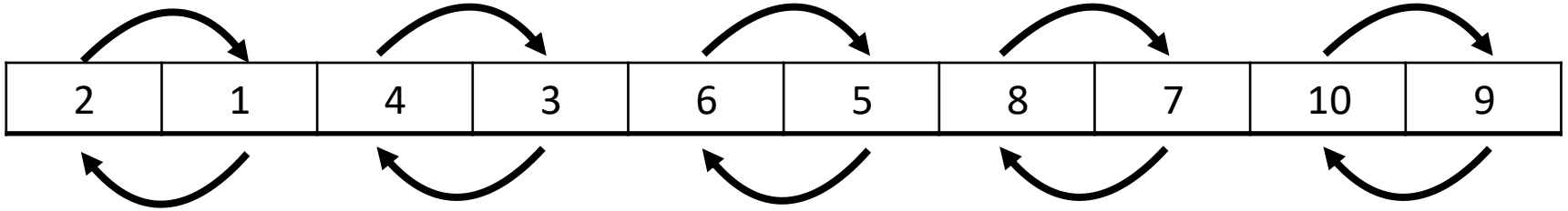
Metric	
Inversions	# pairs in incorrect order
Runs	# increasing contiguous subsequences
Exchanges	least # swaps needed to establish total order



BOSTON UNIVERSITY

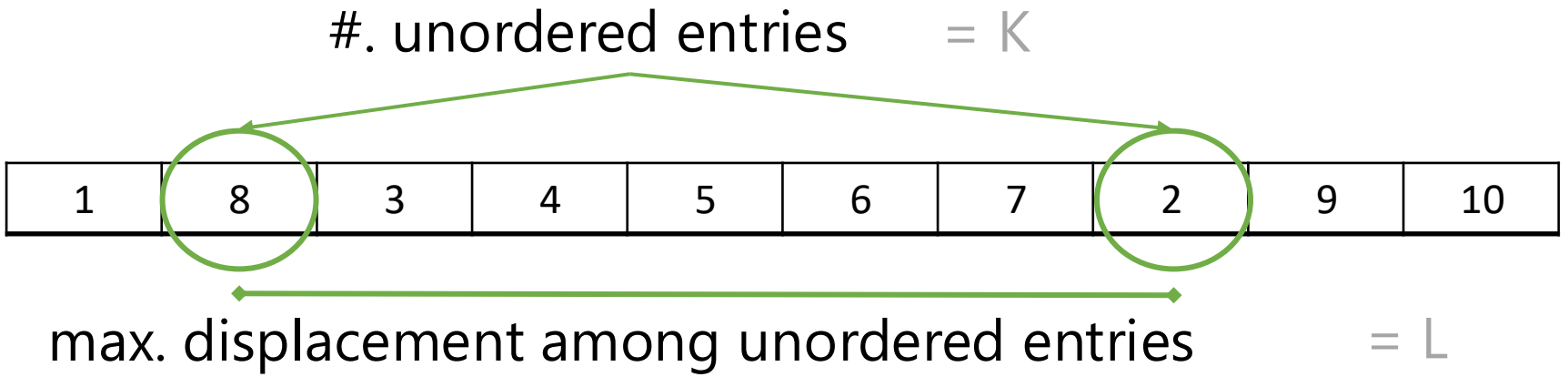
Quantifying Data Sortedness

Metric	
Inversions ✅	# pairs in incorrect order
Runs ⛔	# increasing contiguous subsequences
Exchanges ✅	least # swaps needed to establish total order



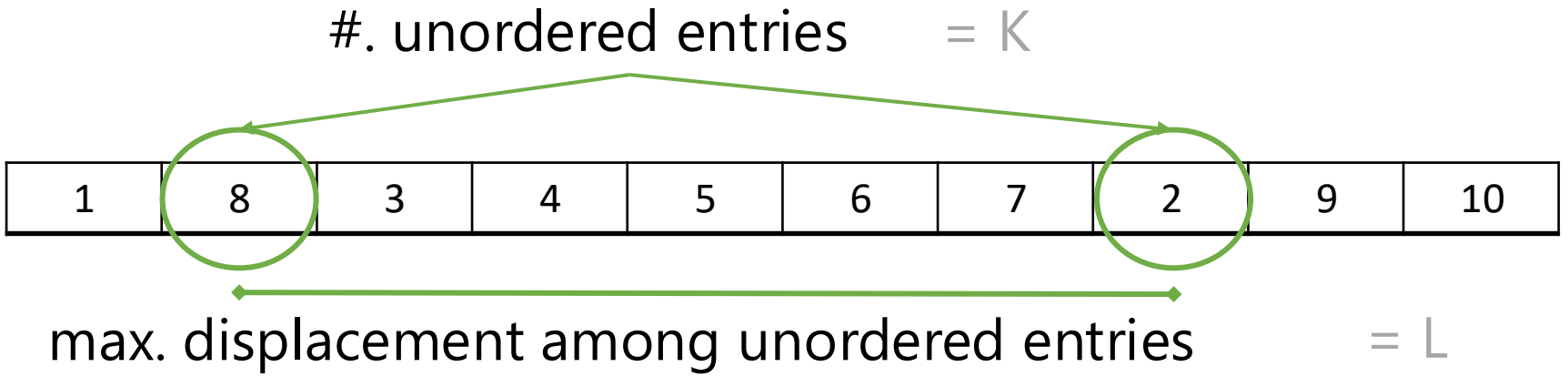
local disorder

(K, L)-Sortedness Metric



[inspired by BenMoshe, ICDT 2011]

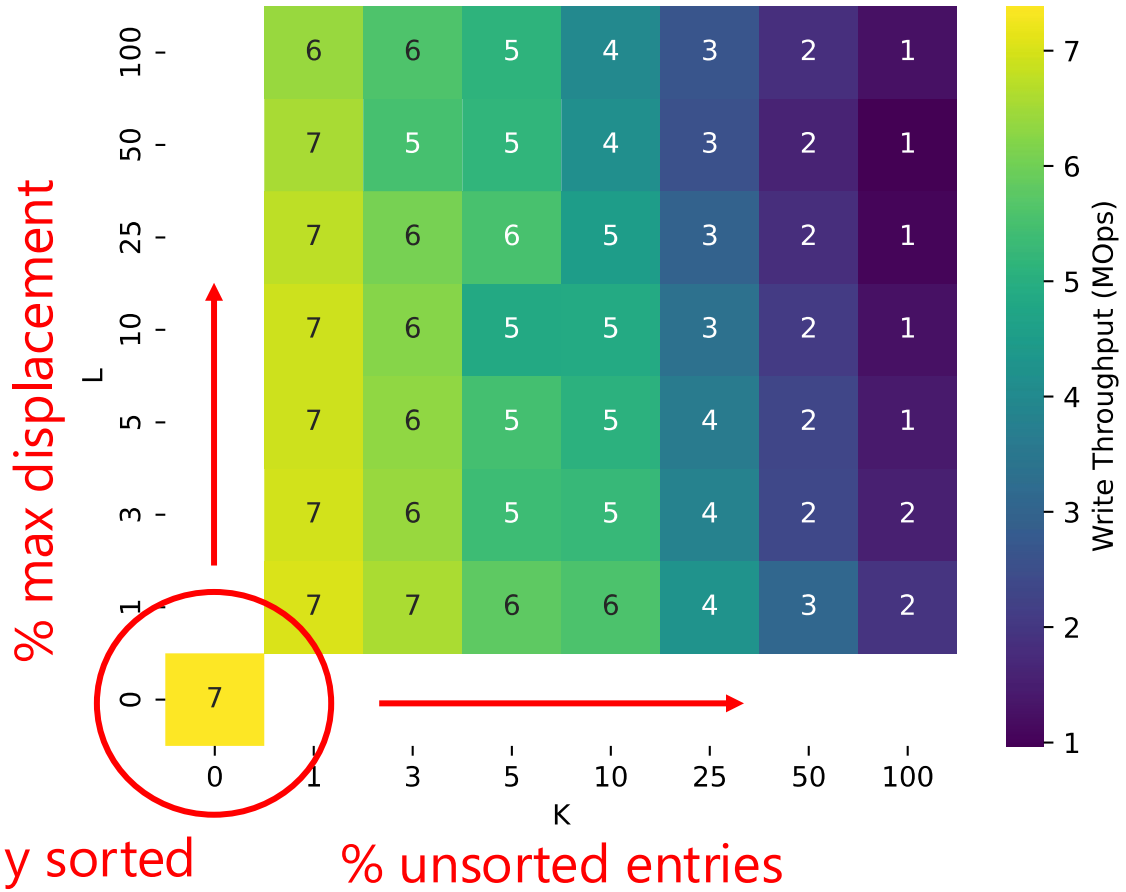
(K, L)-Sortedness Metric



Generate data collections varying K and L

[inspired by BenMoshe, ICDT 2011]

Sortedness Benchmarking on B⁺-Tree



☛ B⁺-tree always performs index traversals

Why higher throughput with high sortedness?

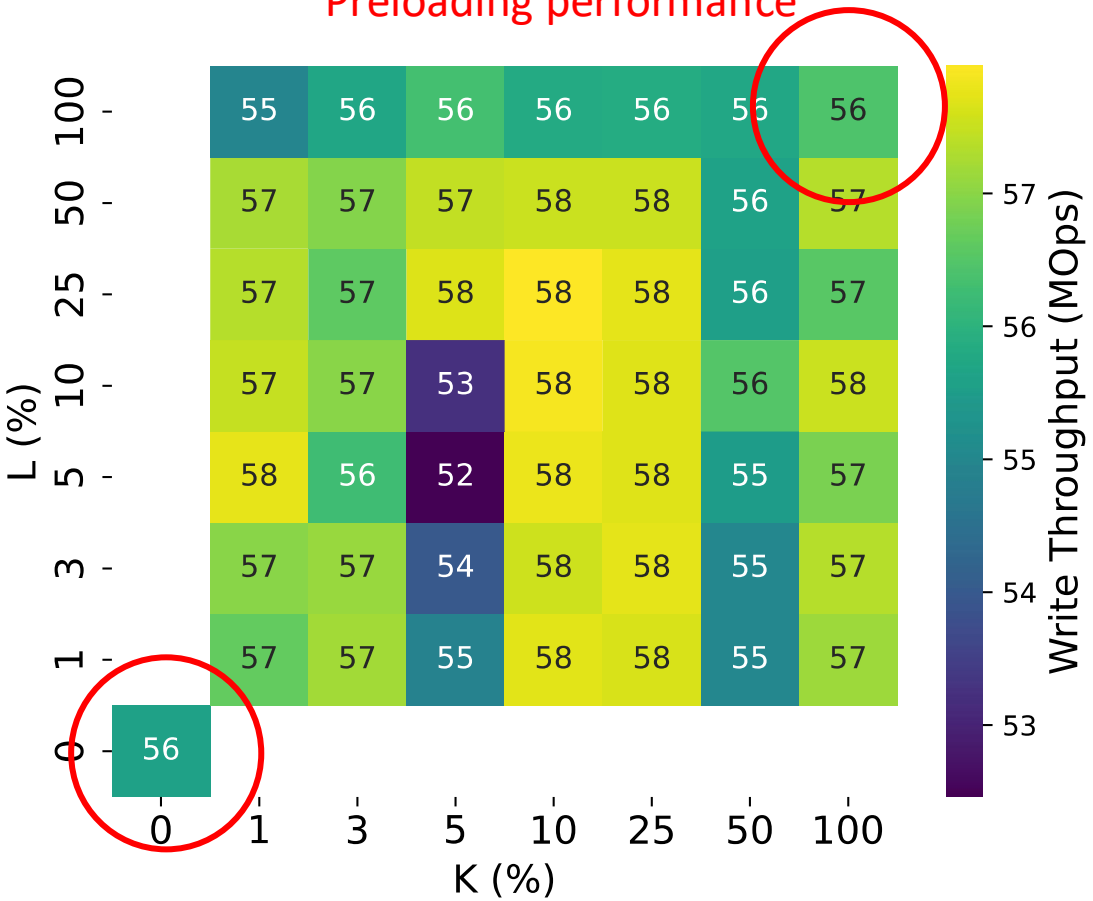
✓ improved performance due to caching effects

However, this is not the ideal performance…

Is this ideal?



Preloading performance



Bulk loading is 8x-

How can we bridge this gap?

BOSTON UNIVERSITY

How do we make B^+-trees Sortedness-Aware?

BOSTON
UNIVERSITY

Agenda

Introduction

Vision

Sortedness Metrics [TPCTC'22]

Sortedness Aware (SWARE) Indexing [ICDE'23]

Quick Insertion Tree (QuIT) [EDBT'25]

Sortedness for Learned Indexes, LSM, Joins [DBTest'24]

BOSTON UNIVERSITY

Sortedness-Aware Paradigm (SWARE)



intelligent buffering

opportunistic bulk loading

increased fill and split factor

SWARE framework can be **applied to any tree-index!**

SWARE Ingestion

append & sort if needed

SWARE Buffer



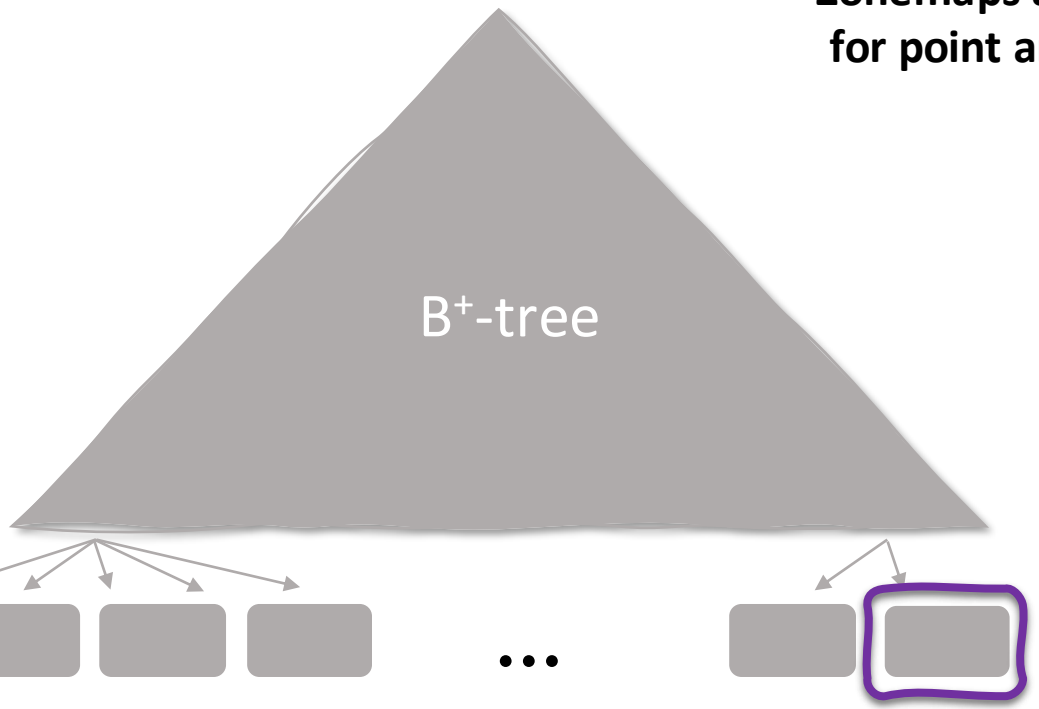


intelligent buffering

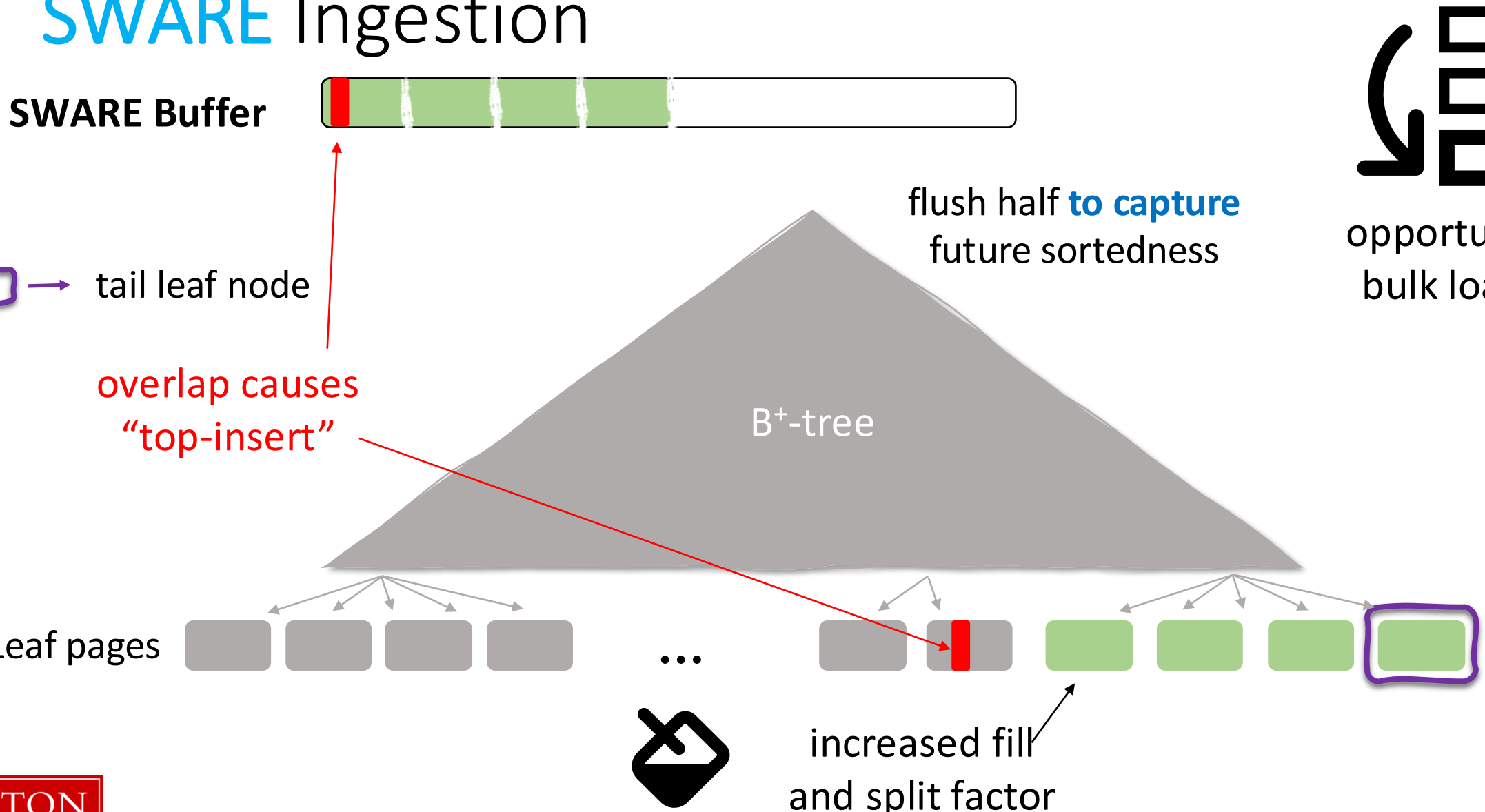
Zonemaps and Bloom filters for point and range queries



tail leaf node



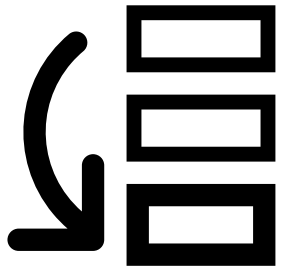
SWARE Ingestion



SWARE Ingestion

SWARE Buffer



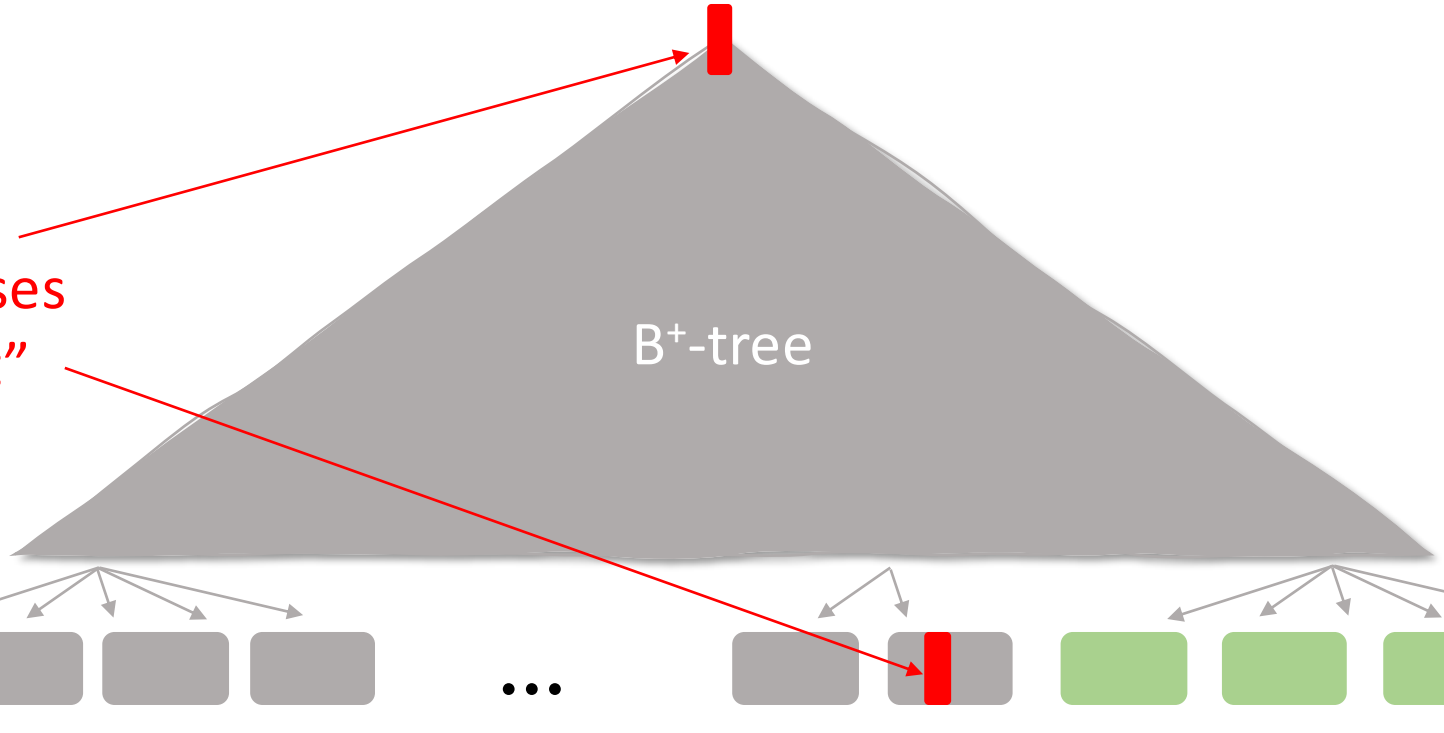


opportunistic bulk loading



tail leaf node

overlap causes "top-insert"

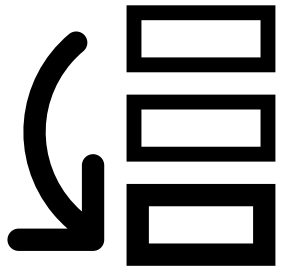


SWARE Ingestion

SWARE Buffer



opportunistic bulk loading resumes!

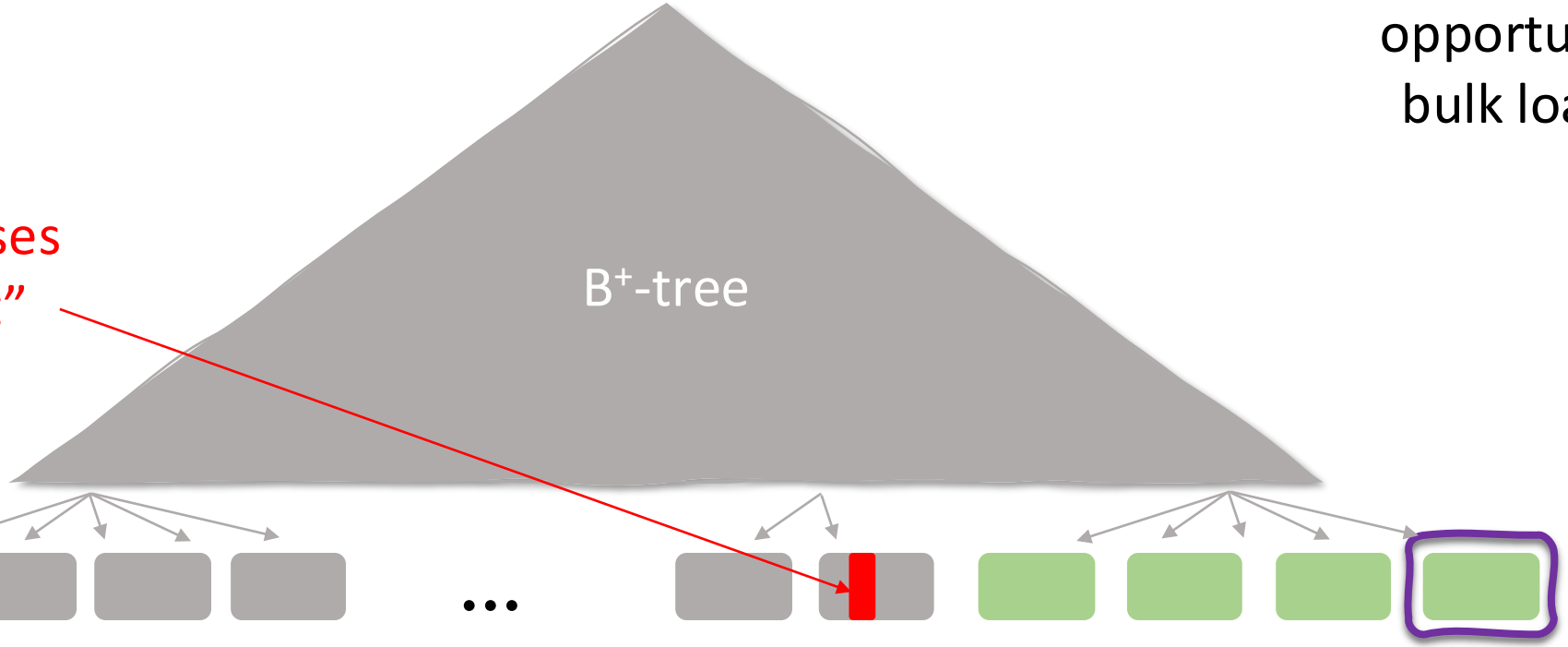


opportunistic bulk loading

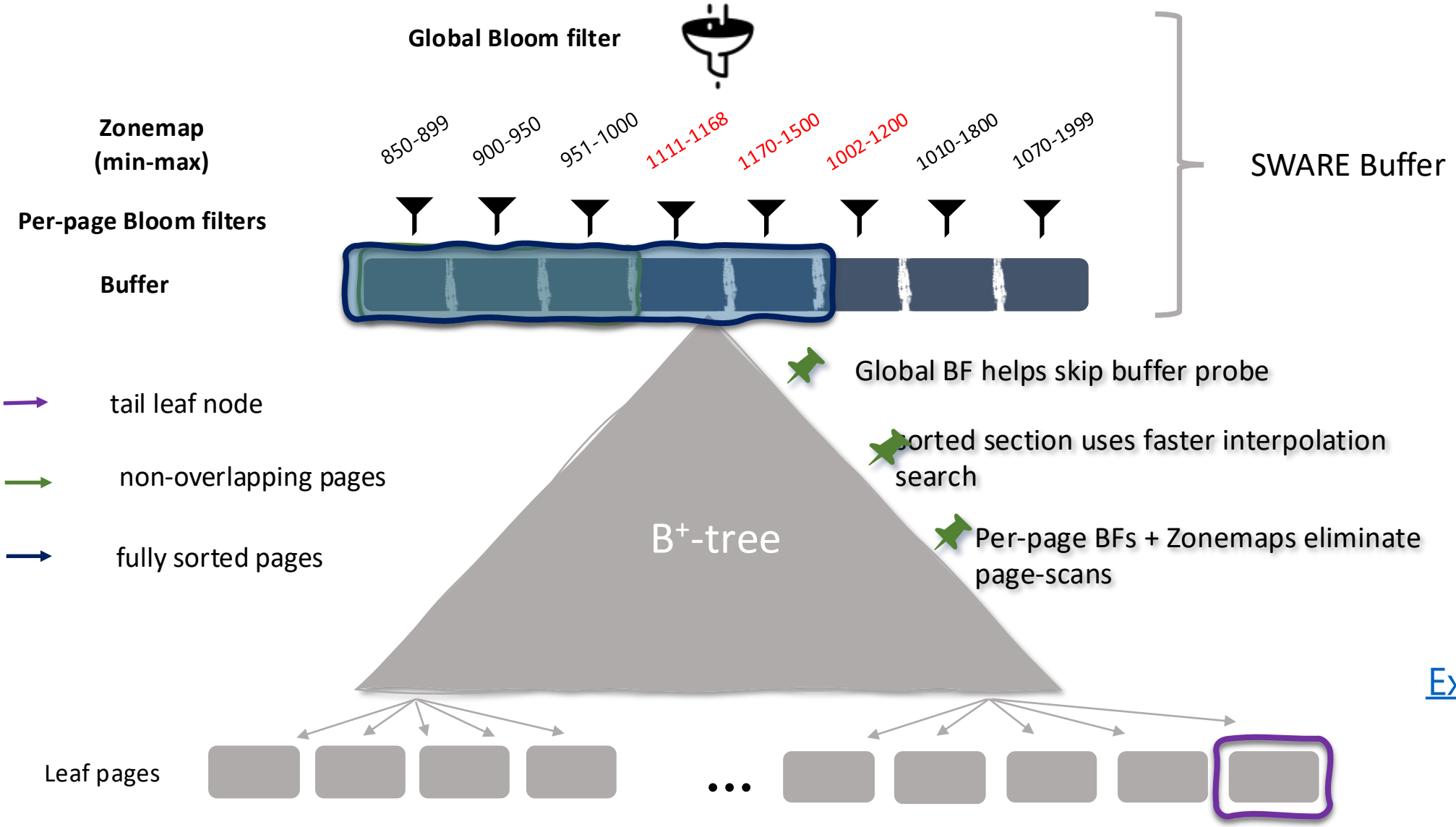


tail leaf node

overlap causes "top-insert"



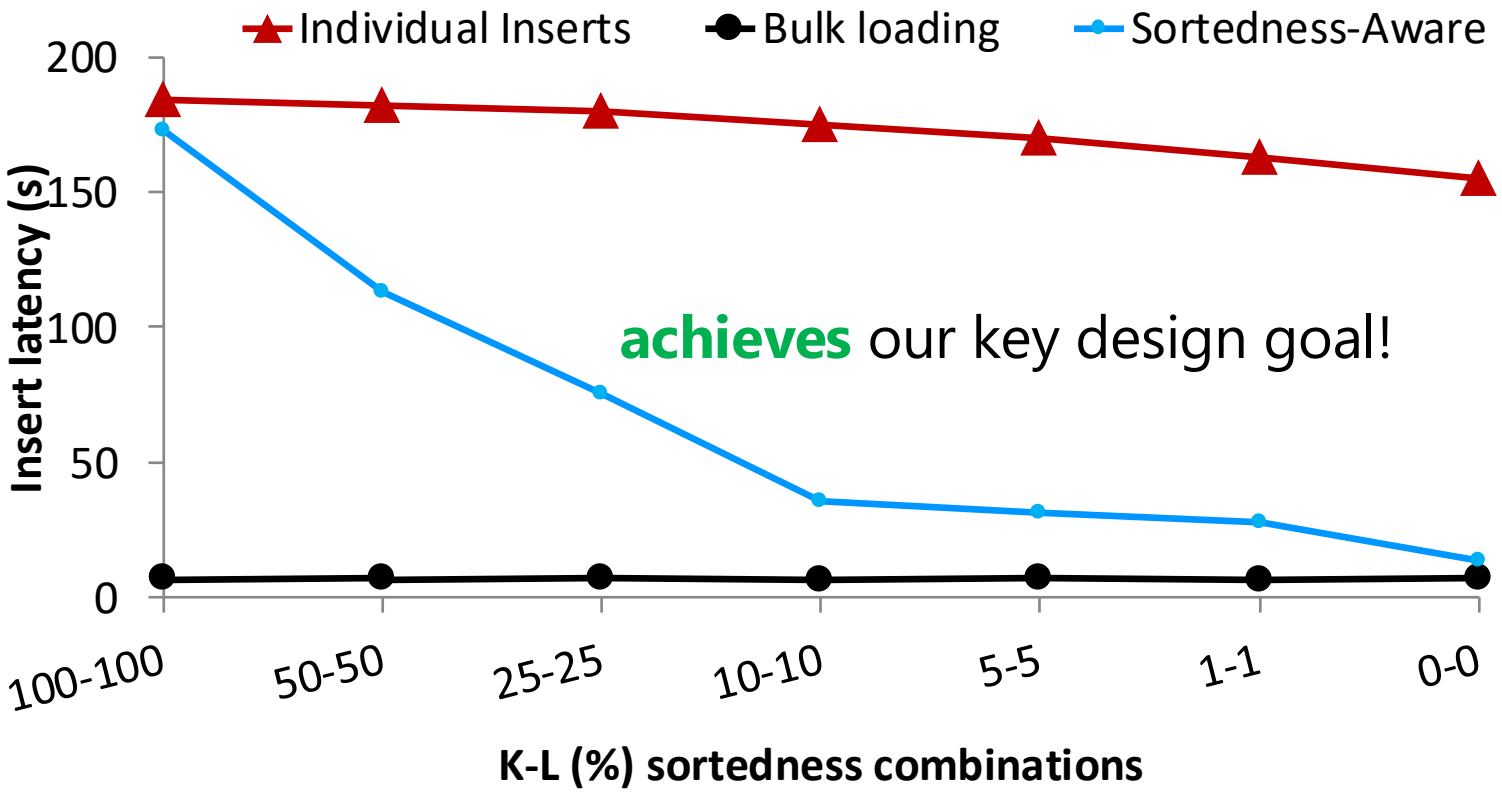
SWARE Lookups



Example

BOSTON UNIVERSITY

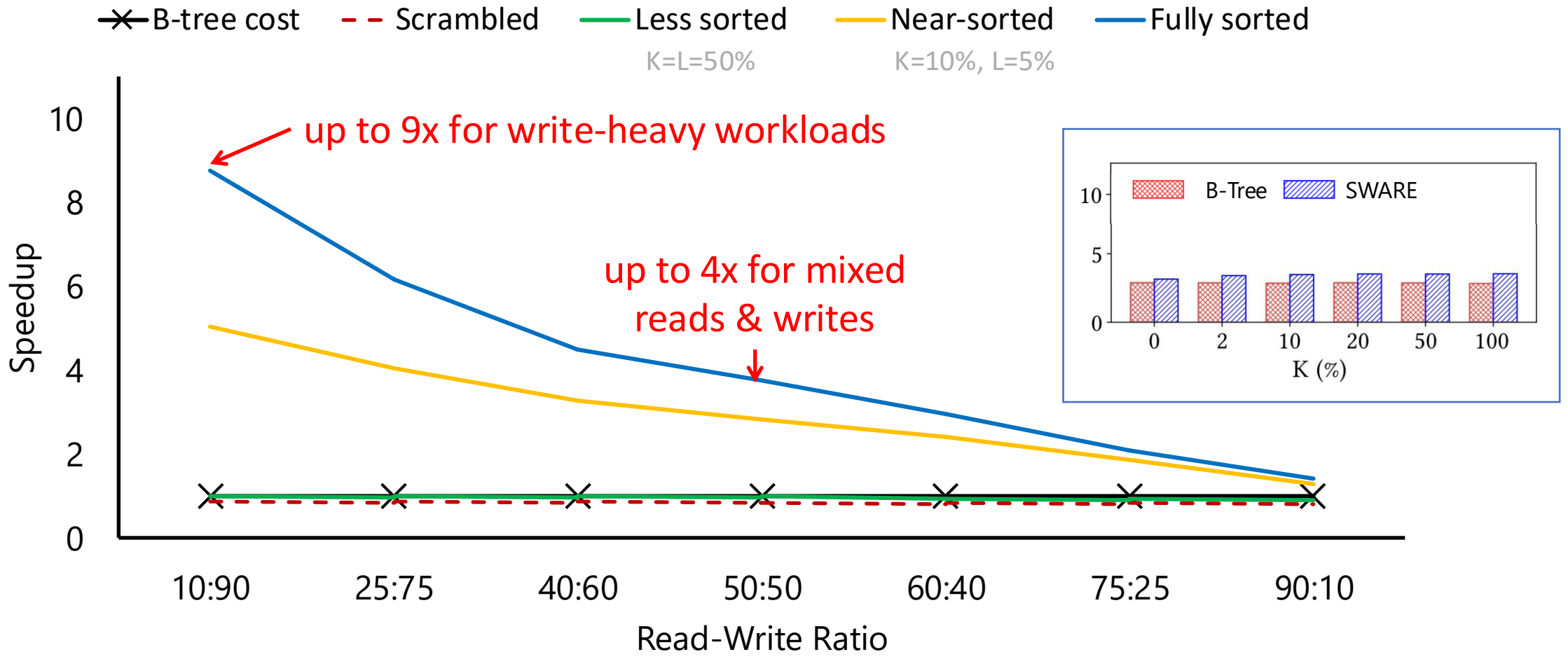
SWARE Bridges Bulk-loading & Top-Inserts





high sortedness → more bulk loading

SWARE Benefits for ↑ Writes

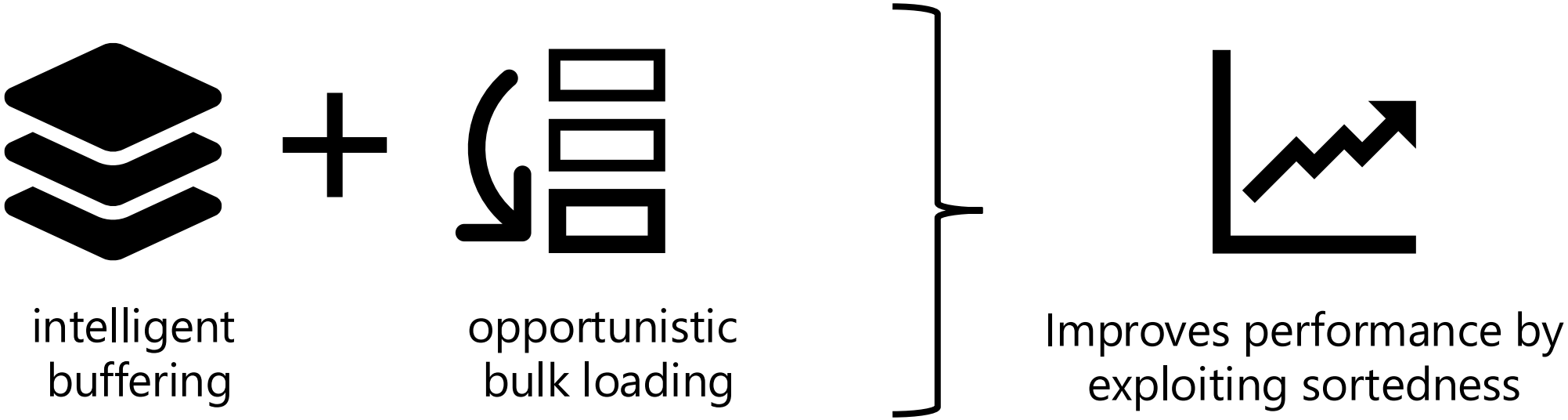


Use *sortedness* as a *resource*!

BUT, reads are ~10% slower!

BOSTON UNIVERSITY

Summarizing SWARE [ICDE 2023]



intelligent buffering

opportunistic bulk loading

Improves performance by exploiting sortedness

BUT, also increases Design Complexity!

Buffering hurts read-intensive workloads!

Can we achieve fast ingestions without buffering?

BOSTON UNIVERSITY

Agenda

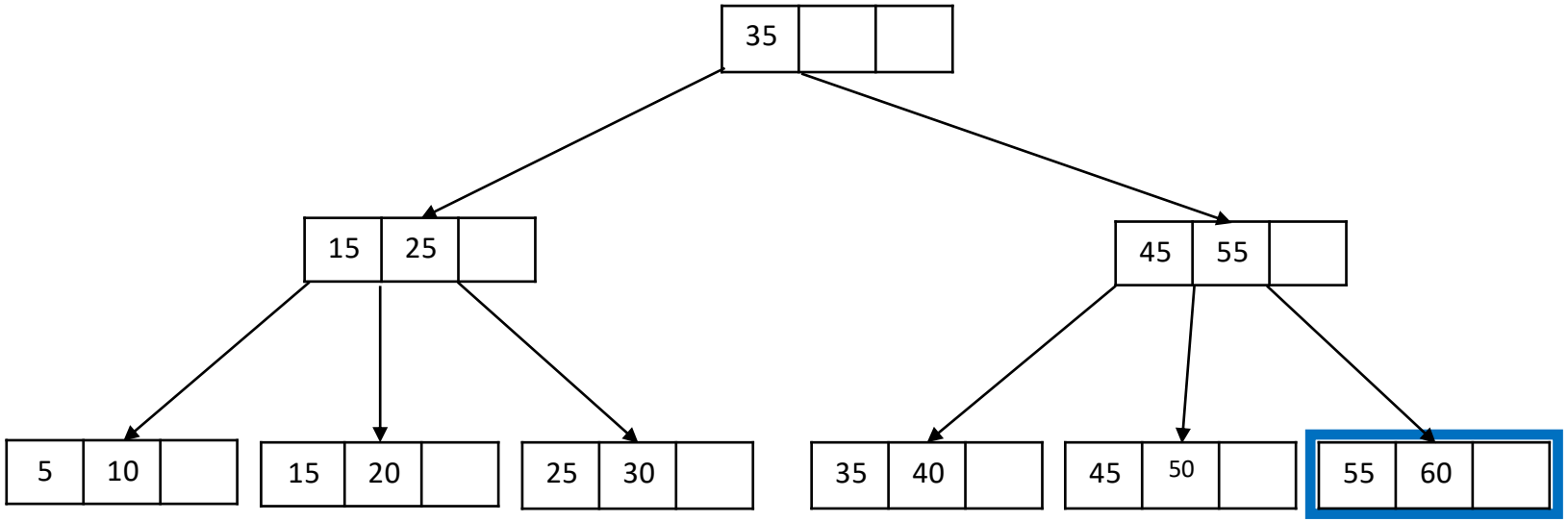
- Introduction
- Vision
- Sortedness Metrics [TPCTC'22]
- Sortedness Aware (SWARE) Indexing [ICDE'23]
- Quick Insertion Tree (QuIT) [EDBT'25]
- Sortedness for Learned Indexes, LSM, Joins [DBTest'24]

BOSTON UNIVERSITY

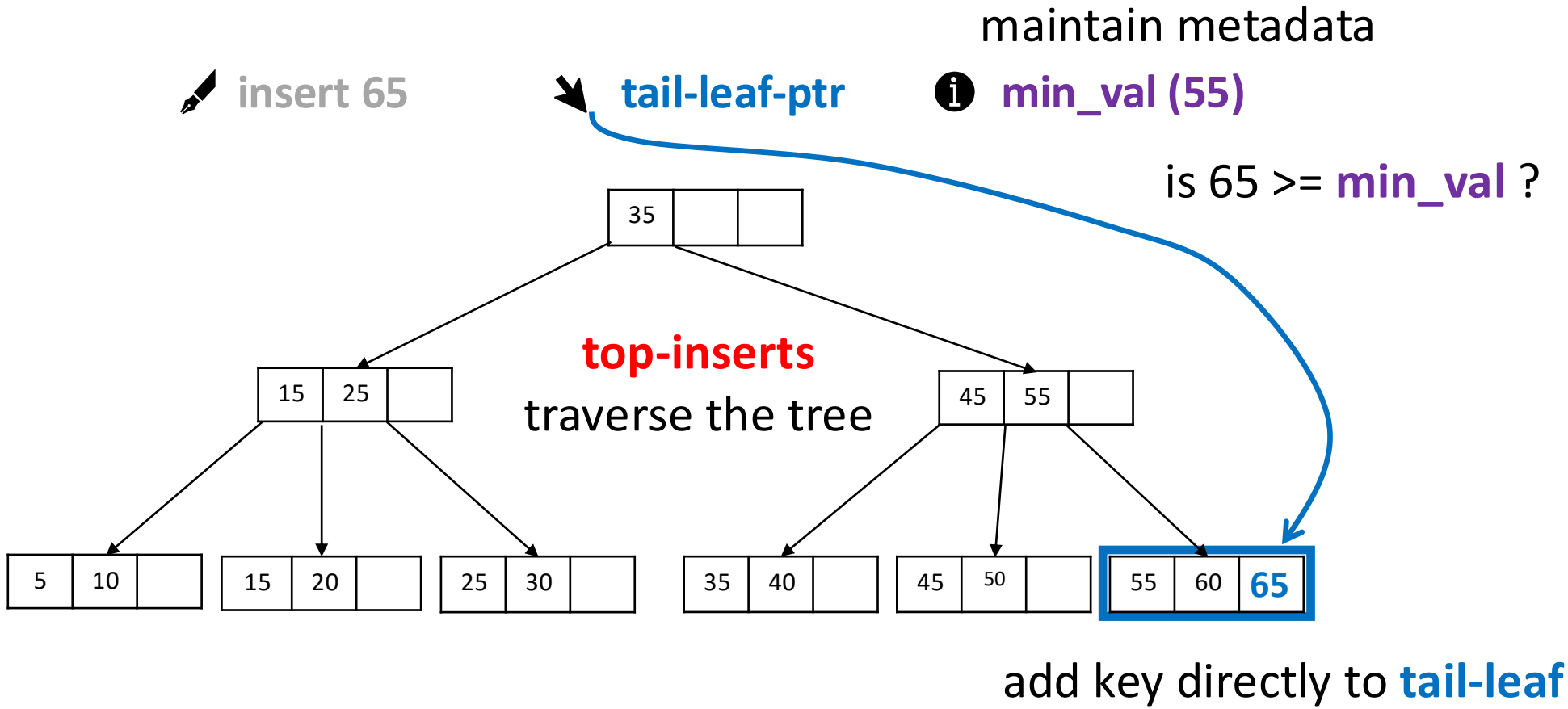
Tail-Leaf Inserts to B$^+$-tree (e.g., PostgreSQL)

maintain metadata

➘ **tail-leaf-ptr** ❶ **min_val (55)**



Tail-Leaf Inserts to B$^+$-tree



Tail-Leaf Inserts to B$^+$-tree

maintain metadata

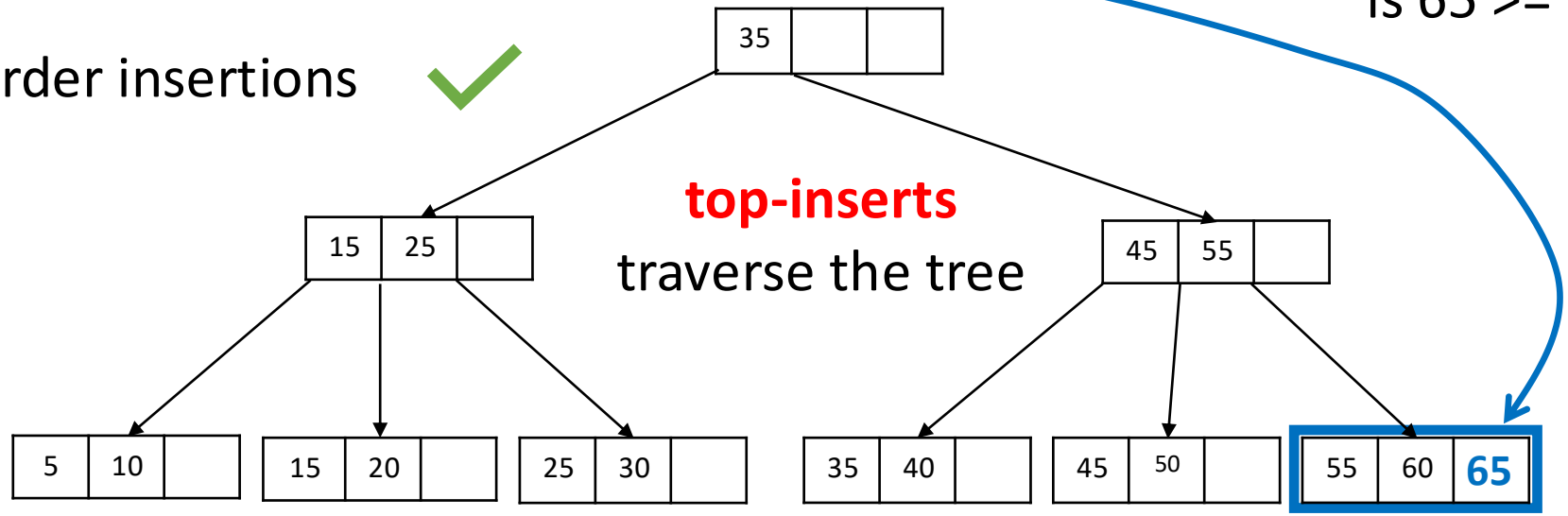
insert 65

tail-leaf-ptr

min_val (55)

is 65 >= min_val ?

helps in-order insertions



top-inserts

traverse the tree

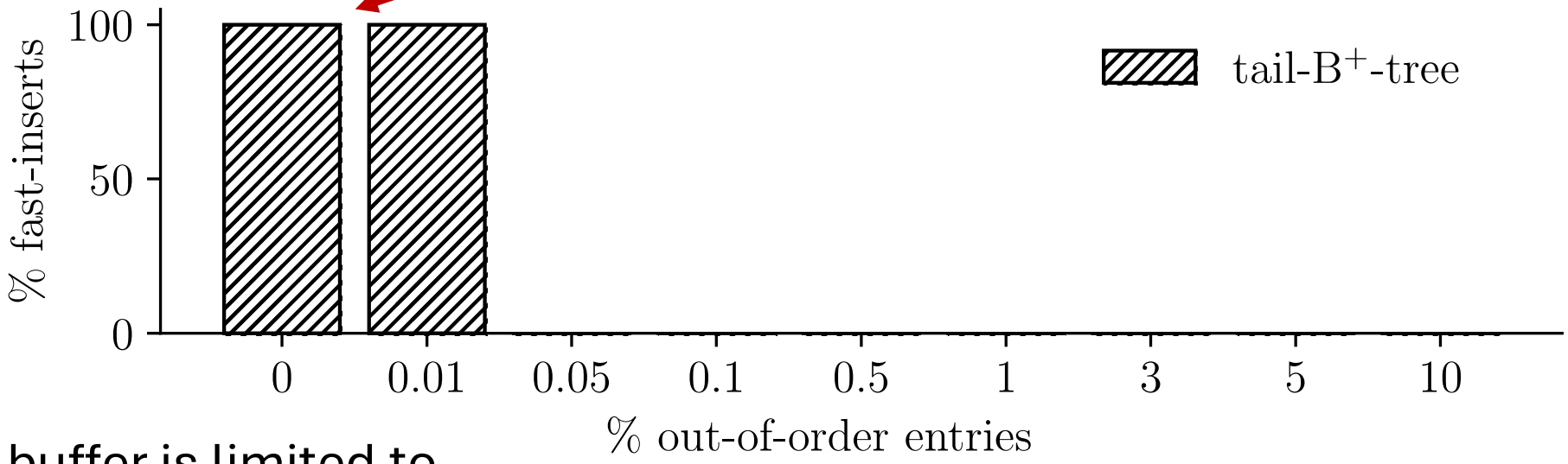
add key directly to tail-leaf



does this also work with near-sorted data?

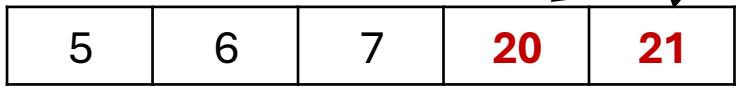
Does This Always Work?

Works for fully and very highly sorted data

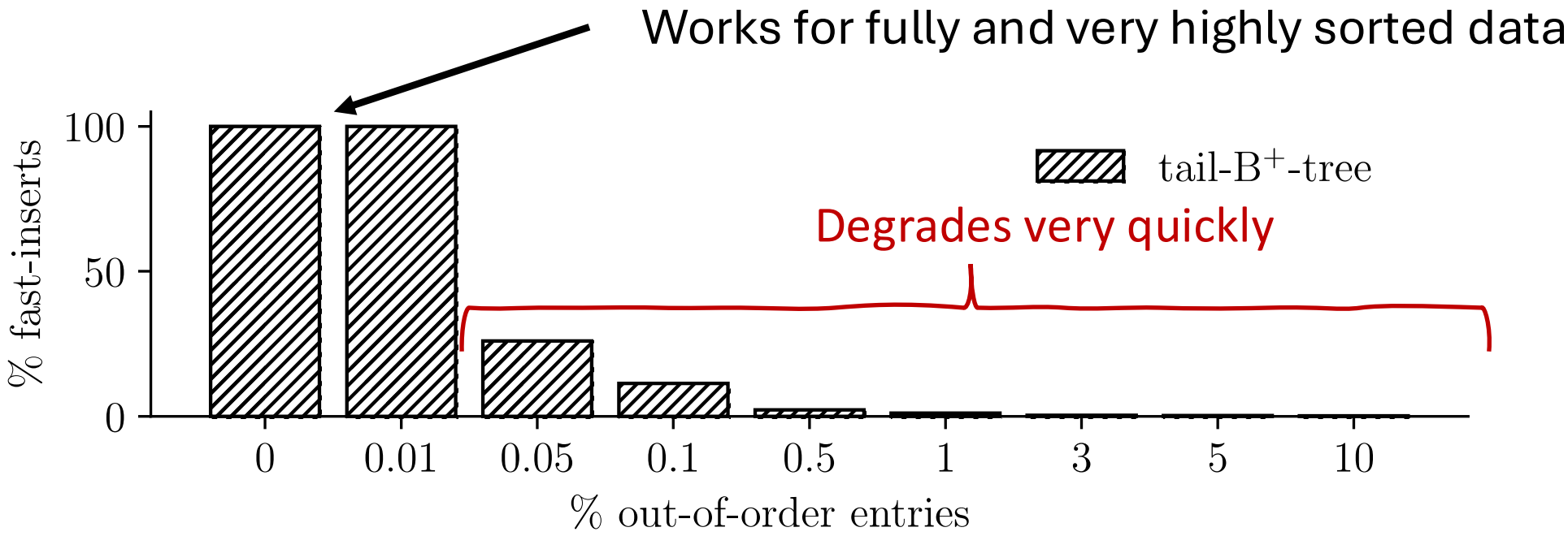


Tail-leaf's buffer is limited to leaf node!

outliers



Does This Always Work?



Successive Insertions are **likely** to occur at the **same leaf** when inserting **near-sorted** data

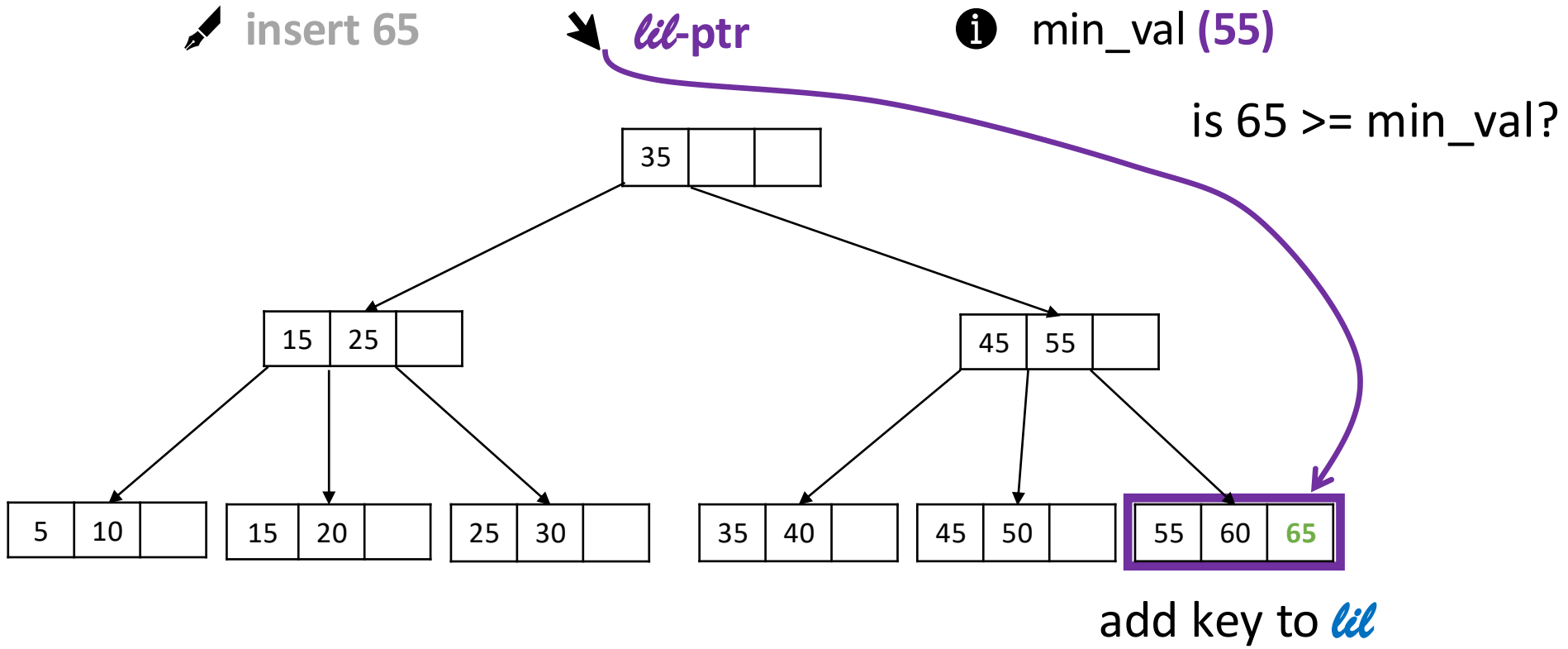
Successive Insertions are **likely** to occur at the **same leaf** when inserting **near-sorted** data



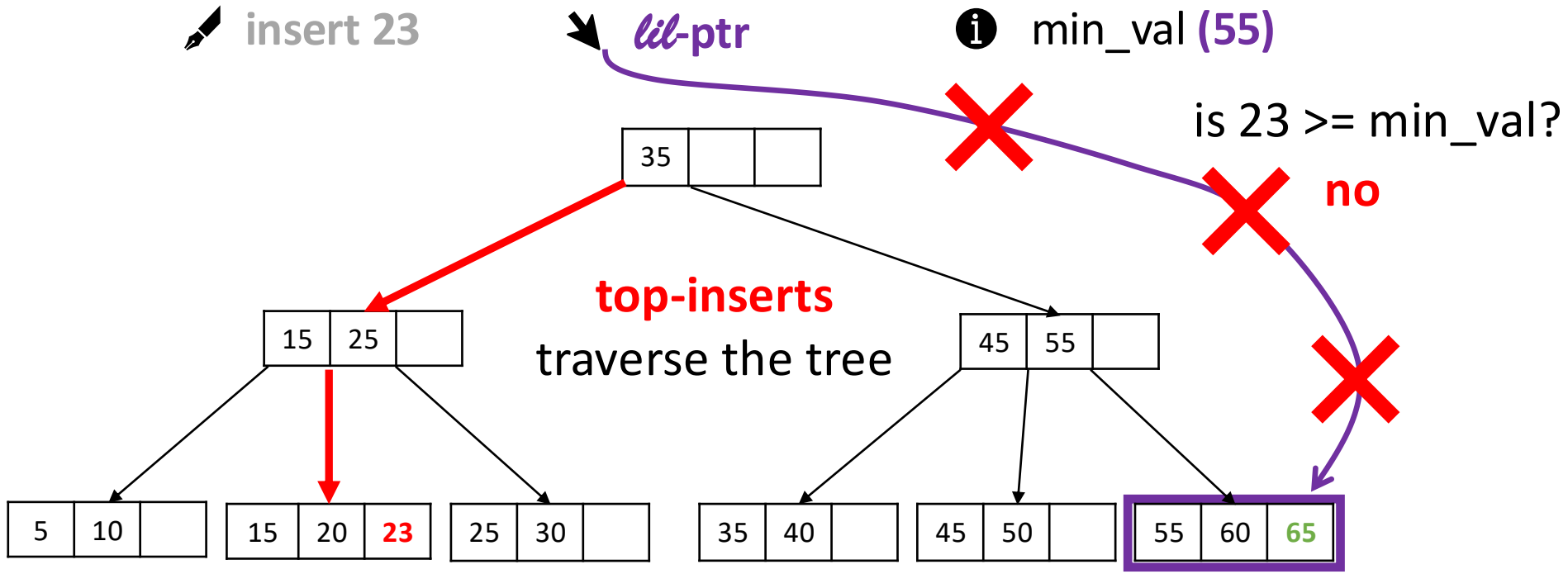
Idea: Keep track of the last inserted leaf

BOSTON UNIVERSITY

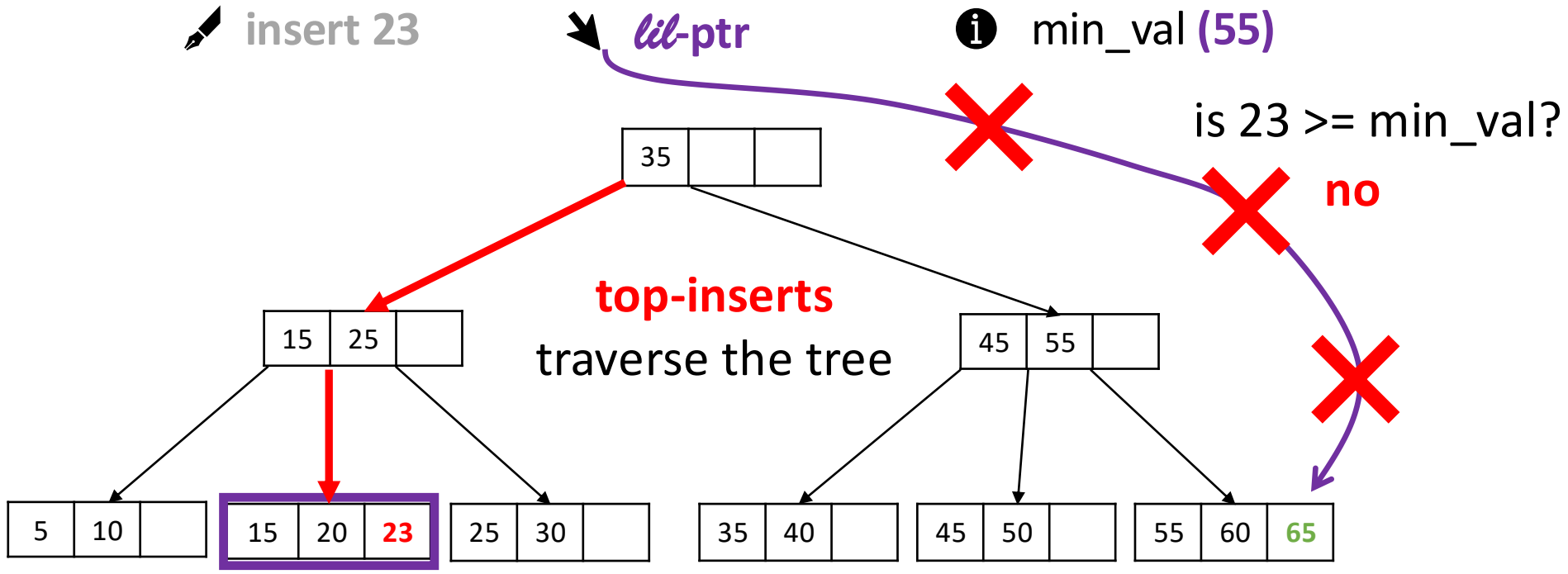
Following the Last Insertion Leaf (*lil*)…



Following the Last Insertion Leaf (*lil*)…



Following the Last Insertion Leaf (*lil*)…



however, *lil* is also updated!

lil in Action



lil achieves **higher** fraction of **fast-inserts**

BOSTON UNIVERSITY

Is *lil* Ideal?



out-of-order insert in *lil* causes 2 top-inserts:

one moves *lil* to a different node

one moves *lil* back to the in-order node

lil pays a **penalty** for with every missed fast-insert!

Ideally, we should incur ***at most*** one **top-insert** for every **out-of-order** entry

BOSTON UNIVERSITY

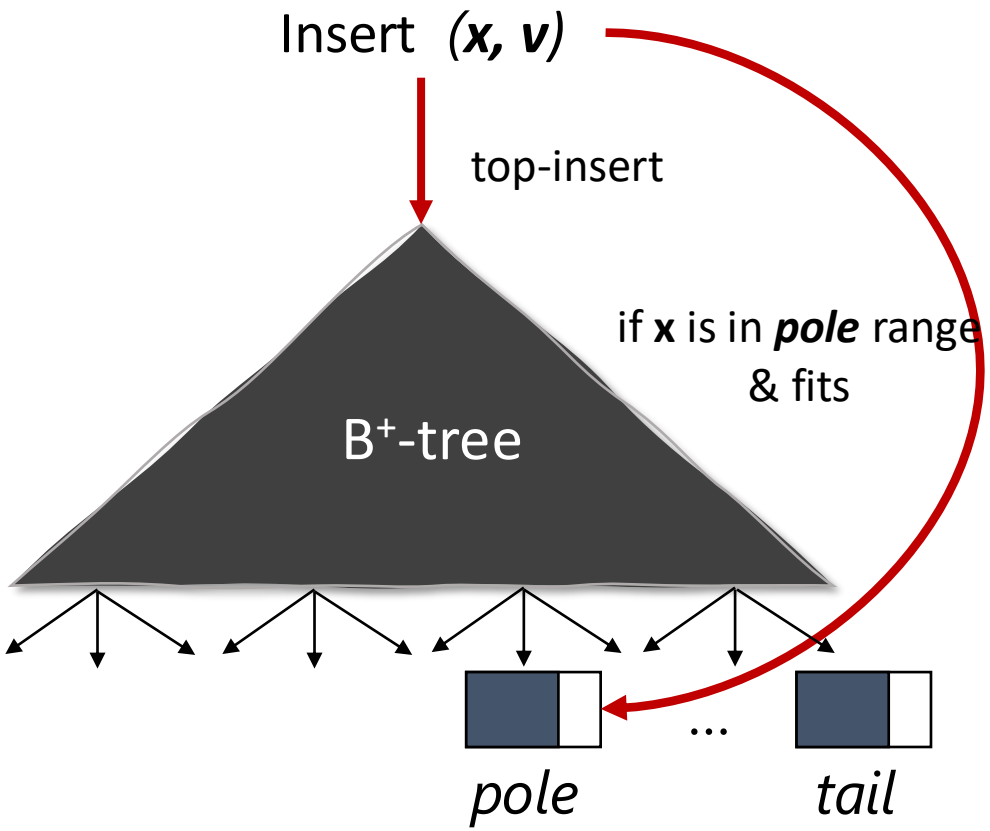
Ideally, we should incur ***at most*** one **top-insert** for every **out-of-order** entry



Idea: Predict the next ordered leaf

it could be *any* node!

Predicted Ordered Leaf (*pole*)

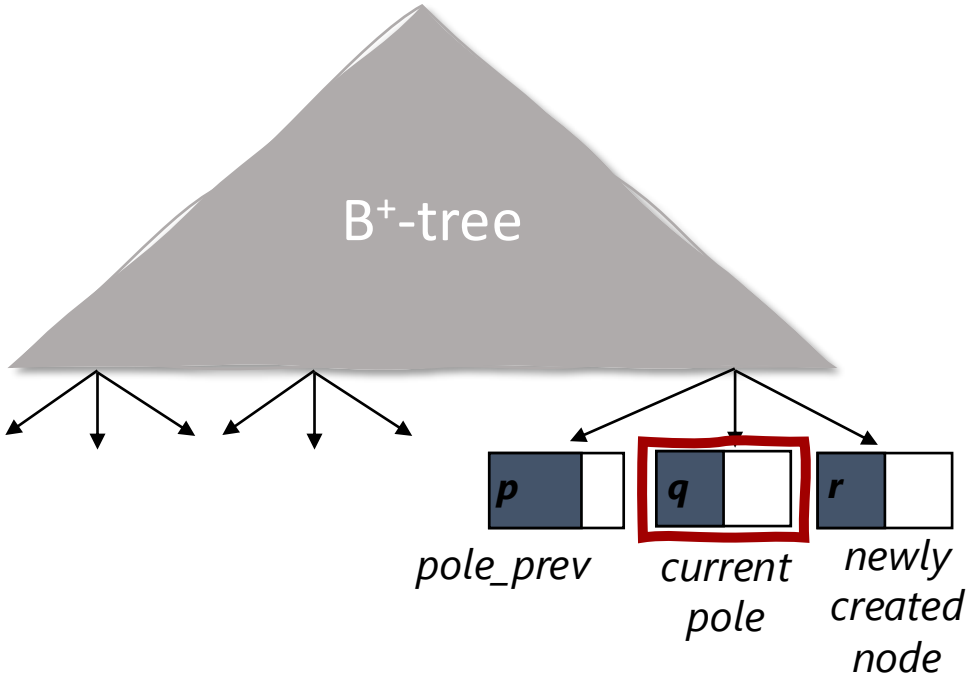


tail-leaf quickly fills up with outliers

lil naively switches fast-path

pole needs to be updated – when?

When Pole Splits



Legend

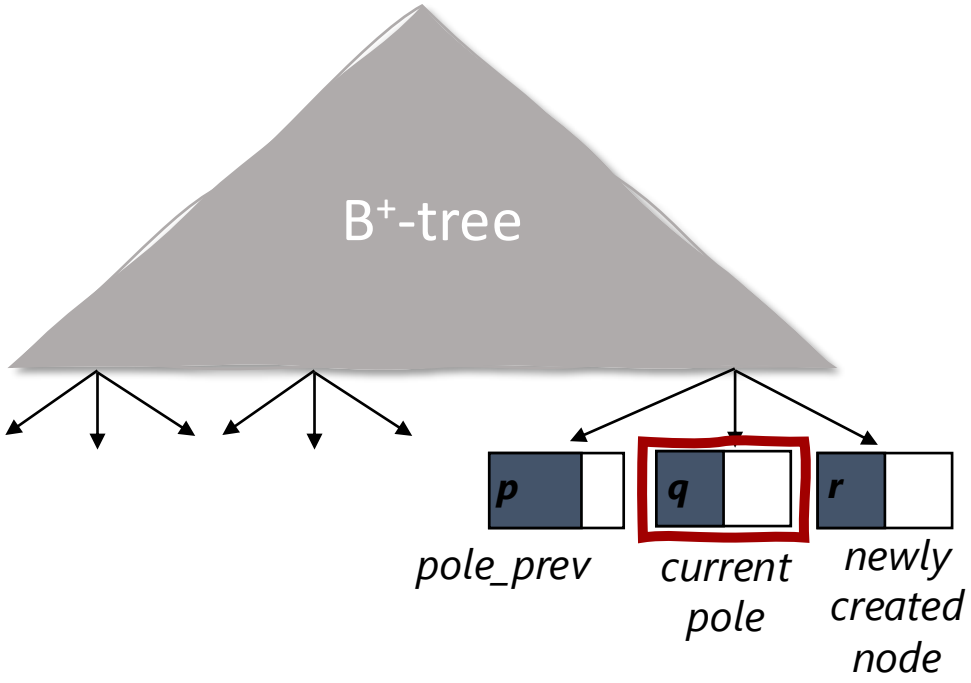
p = smallest entry in node previous to *pole;*
q = smallest entry in *pole*
r = smallest entry in newly created node
= pointer to *pole* node

Predict using *IKR (In-order Key estimatoR)*

$$x = q + \left(\frac{q - p}{pole_prev_{size}} \right) \cdot pole_{size} \cdot (scale)$$

density between two non-outliers

When Pole Splits



Legend

p = smallest entry in node previous to *pole;*
q = smallest entry in *pole*
r = smallest entry in newly created node
□ = pointer to *pole* node

Predict using *IKR (In-order Key estimatoR)*

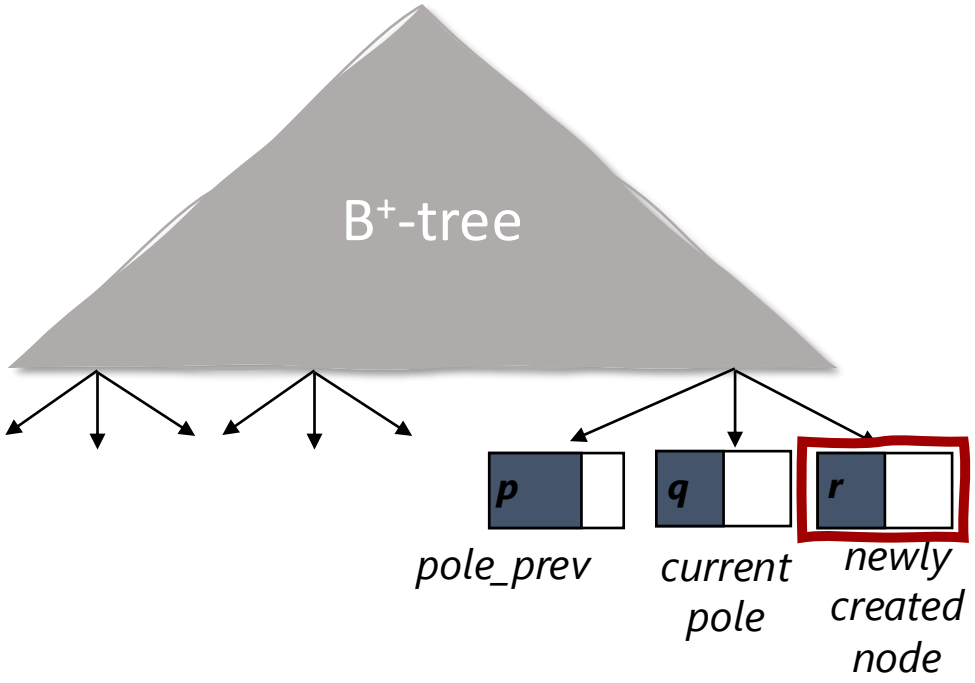
if $r >$

$$x = q + \left(\frac{q - p}{pole_prev_{size}}\right) \cdot pole_{size} \cdot (scale)$$

density between two non-outliers

*new node has **only outliers** → **pole** stays as is*

When Pole Splits



Legend

p = smallest entry in node previous to *pole;*
q = smallest entry in *pole*
r = smallest entry in newly created node
= pointer to *pole* node

Predict using *IKR (In-order Key estimatoR)*

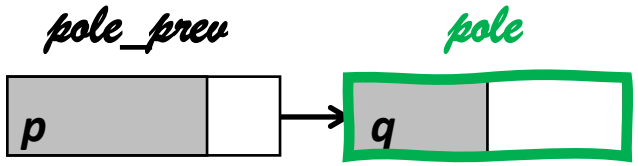
if $r \leq$ $x = q + \left(\frac{q - p}{pole_prev_{size}}\right) \cdot pole_{size} \cdot (scale)$

density between two non-outliers

***r* is *not* an outlier → *update pole* to newly created node**

Full Insertion Algorithm

```
if(key is within pole range){
     if(pole.size < capacity){
          pole.insert(entry);
     }
     else{
          pole_next = pole.split();
          r = pole_next.min;
          x = IKR(q, p, pole_prev.size, pole.size);
          if(r <= x){
               pole_prev = pole;
               pole = pole.next;
          }
     }
}
else{
     l = insert(entry);
}
```

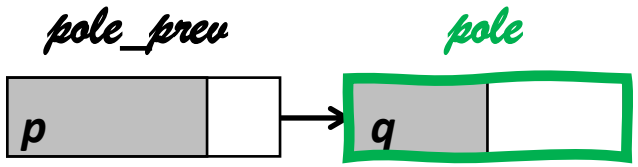


Full Insertion Algorithm

check if ***key* qualifies** for a fast-insert

```
if(key is within pole range){
    if(pole.size < capacity){
        pole.insert(entry);
    }
    else{
        pole_next = pole.split();
        r = pole_next.min;
        x = IKR(q, p, pole_prev.size, pole.size)
        if(r <= x){
            pole_prev = pole;
            pole = pole.next;
        }
    }
}
else{
    i = insert(entry);
}
```

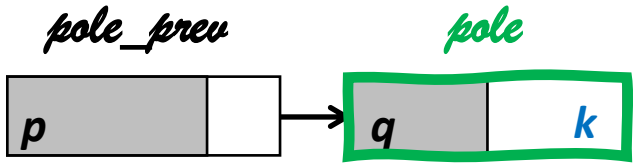
otherwise, **top-insert** through root



Predicting the Ordered Leaf (*pole*)

```
if(key is within pole range){
    if(pole.size < capacity){
        pole.insert(entry);
    }
    else{
        pole_next = pole.split();
        r = pole_next.min;
        x = IKR(q, p, pole_prev.size, pole.size);
        if(r <= x){
            pole_prev = pole;
            pole = pole.next;
        }
    }
}
else{
    l = insert(entry);
}
```

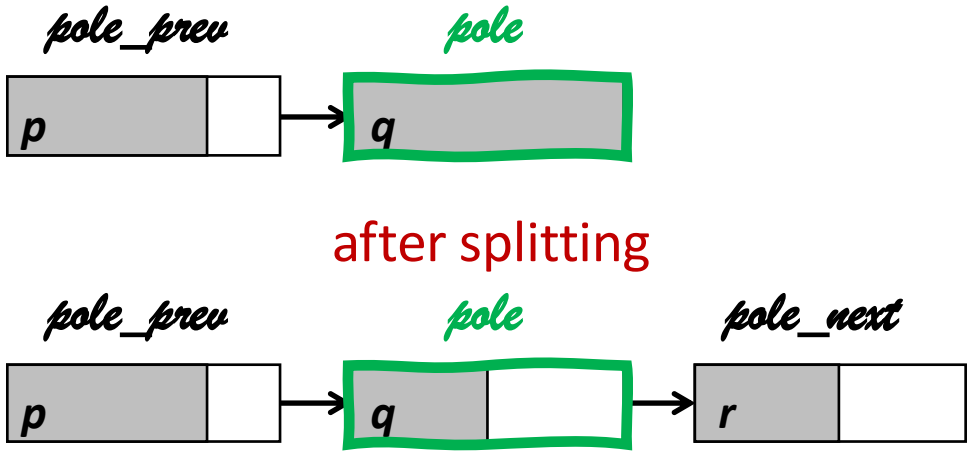
if *pole* is not full, insert *k* into *pole*



Predicting the Ordered Leaf (*pole*)

```
if(key is within pole range){
     if(pole.size < capacity){
          pole.insert(entry);
     }
     else{
          pole_next = pole.split();
          r = pole_next.min;
          x = IKR(q, p, pole_prev.size, pole.size);
          if(r <= x){
               pole_prev = pole;
               pole = pole.next;
          }
     }
}
else{
     l = insert(entry);
}
```

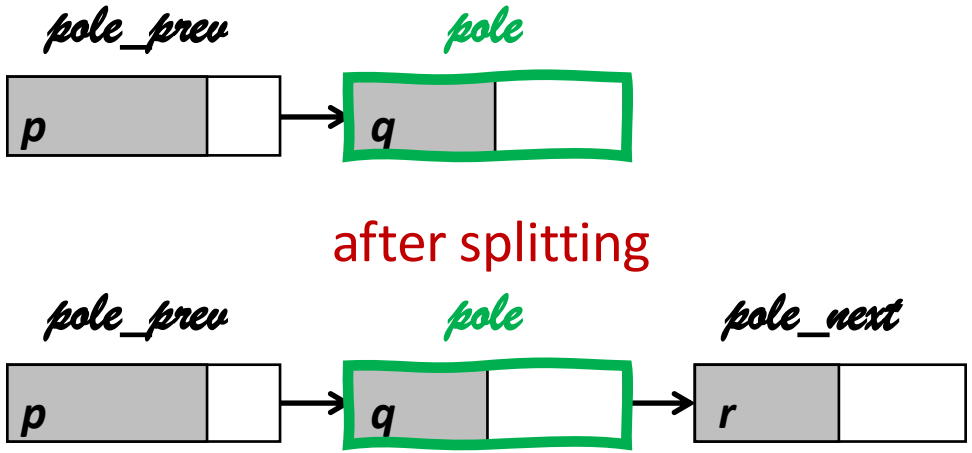
otherwise, split *pole*



Predicting the Ordered Leaf (*pole*)

```
if(key is within pole range){
     if(pole.size < capacity){
          pole.insert(entry);
     }
     else{
          pole_next = pole.split();
          r = pole_next.min;
          x = IKR(q, p, pole_prev.size, pole.size);
          if(r <= x){
               pole_prev = pole;
               pole = pole.next;
          }
     }
}
else{
     l = insert(entry);
}
```

use *IKR*



$$x = q + \left(\frac{q-p}{pole_prev_{size}}\right) \cdot pole_{size} \cdot I(scale)$$

density between two non-outliers

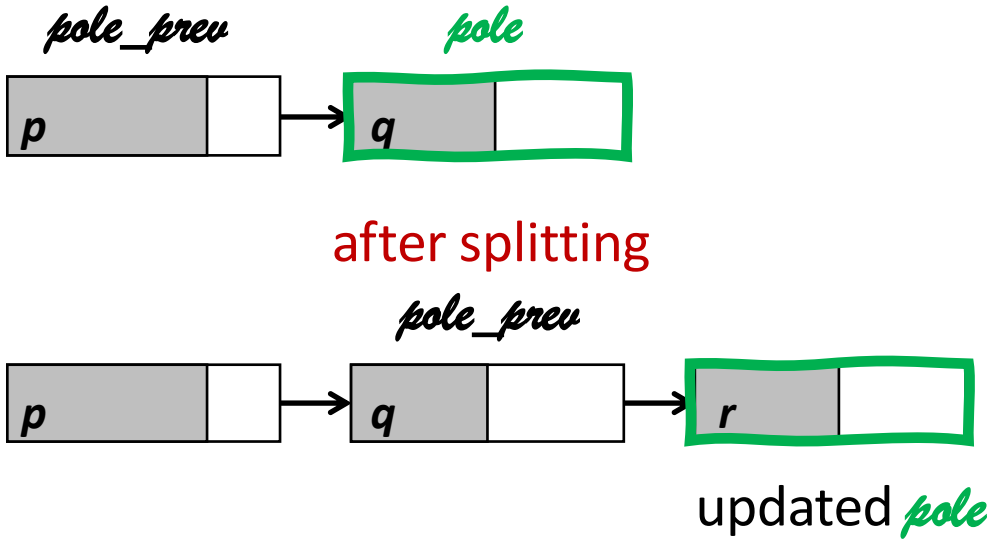
capture small deviations

default to ***1.5***

Predicting the Ordered Leaf (*pole*)

```
if(key is within pole range){
     if(pole.size < capacity){
          pole.insert(entry);
     }
     else{
          pole_next = pole.split();
          r = pole_next.min;
          x = IKR(q, p, pole_prev.size, pole.size);
          if(r <= x){
               pole_prev = pole;
               pole = pole.next;
          }
     }
}
else{
     l = insert(entry);
}
```

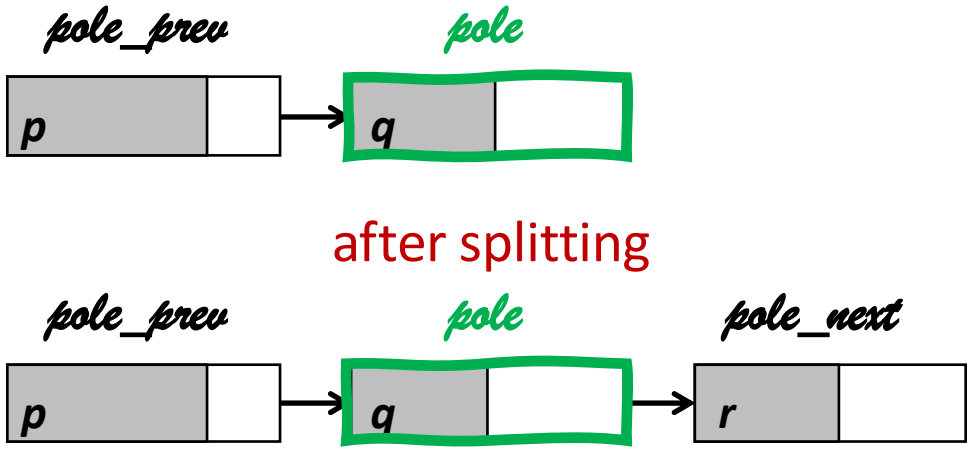
update *pole* if *pole_next* has at least one non-outlier



Predicting the Ordered Leaf (*pole*)

```
if(key is within pole range){
     if(pole.size < capacity){
          pole.insert(entry);
     }
     else{
          pole_next = pole.split();
          r = pole_next.min;
          x = IKR(q, p, pole_prev.size, pole.size);
          if(r <= x){
               pole_prev = pole;
               pole = pole.next;
          }
     }
}
else{
     l = insert(entry);
}
```

otherwise, *pole* remains as is



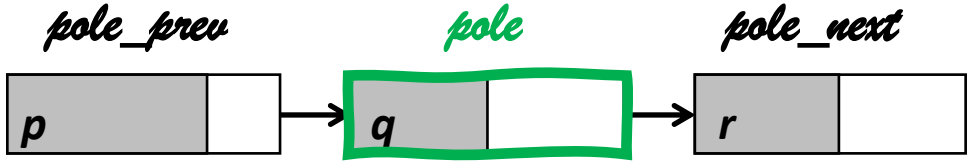
Can We Better Utilize Space?

high sortedness => poor space utilization

can we find better split points?

IKR can also return the split point

return last key <= x in *pole*

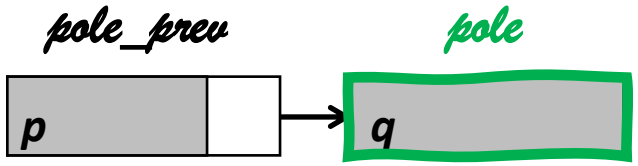


Can We Better Utilize Space?

high sortedness => poor space utilization

```
l_pos = IKR(q, p, pole_prev.size);

if(l_pos <= 50%){
     pole_next = pole.split(l_pos);
}
else{
     pole.next = pole.split(l_pos - 1);
     pole_prev = pole;
     pole = pole.next;
}
```



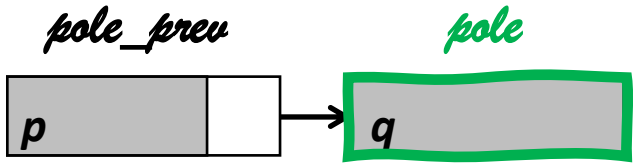
let's call this key ℓ and its position ℓ_pos

Can We Better Utilize Space?

high sortedness => poor space utilization

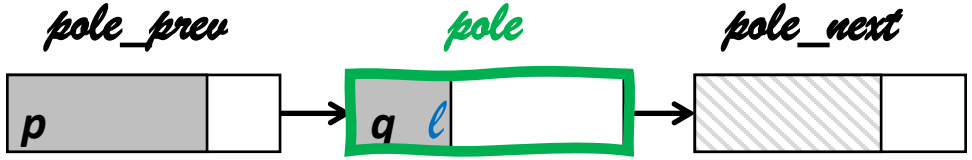
```
l_pos = IKR(q, p, pole_prev.size);

if(l_pos <= 50%){
     pole_next = pole.split(l_pos);
}
else{
     pole.next = pole.split(l_pos - 1);
     pole_prev = pole;
     pole = pole.next;
}
```



if $l_pos \leq 50\%$

outliers dominate if *pole*



split at ℓ

moves all outliers to ***pole_next***

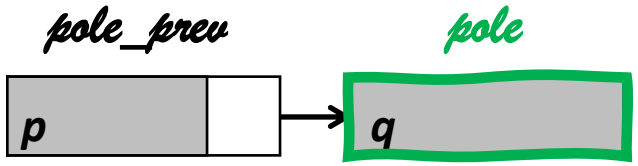
the idea is that we make room for future in-order insertions in ***pole***

Can We Better Utilize Space?

high sortedness => poor space utilization

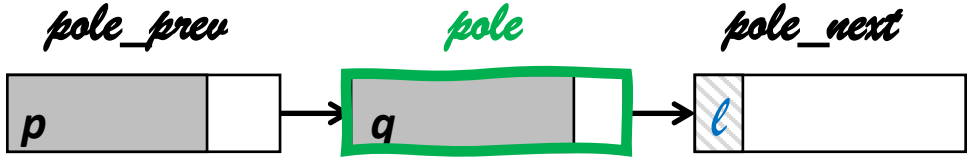
```
l_pos = IKR(q, p, pole_prev.size);

if(l_pos <= 50%){
     pole_next = pole.split(l_pos);
}
else{
     pole.next = pole.split(l_pos - 1);
     pole_prev = pole;
     pole = pole.next;
}
```



if *l_pos* > 50%

pole has few outliers



split at *l_pos* - 1

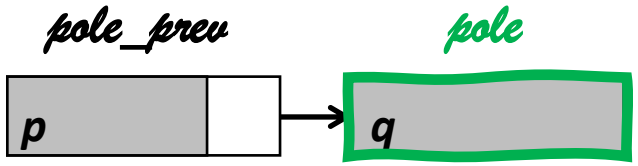
moves at least one non-outlier to *pole_next*

Can We Better Utilize Space?

high sortedness => poor space utilization

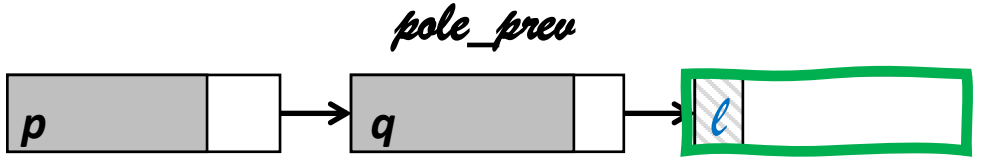
```
l_pos = IKR(q, p, pole_prev.size);

if(l_pos <= 50%){
     pole_next = pole.split(l_pos);
}
else{
     pole.next = pole.split(l_pos - 1);
     pole_prev = pole;
     pole = pole.next;
}
```



if *l_pos* > 50%

pole has few outliers



split at *l_pos* - 1 updated *pole*

leaves more space in *pole* to receive in-order insertions

pole_prev is more compact (more than 50% full)

BOSTON UNIVERSITY

Quick Insertion Tree

similar to **B⁺-tree** in design

+

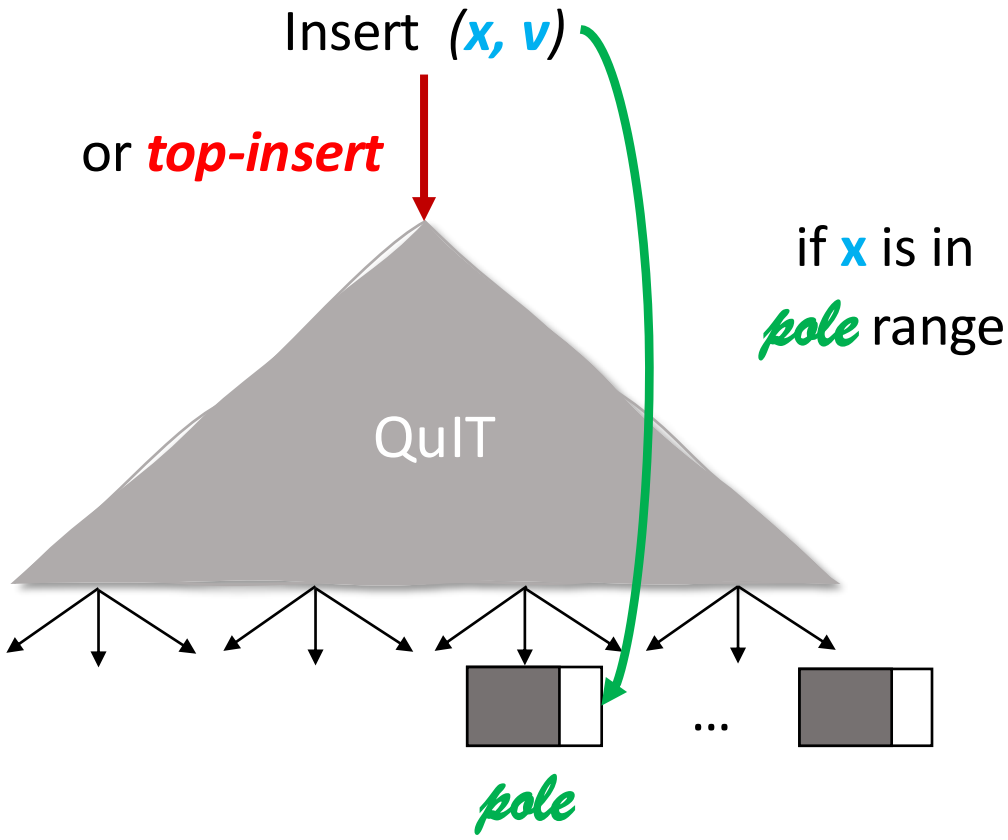
offers **fast-path** ingestion

+

sortedness-aware

+

minimal metadata + ***tuning***



Evaluating QuIT

System:

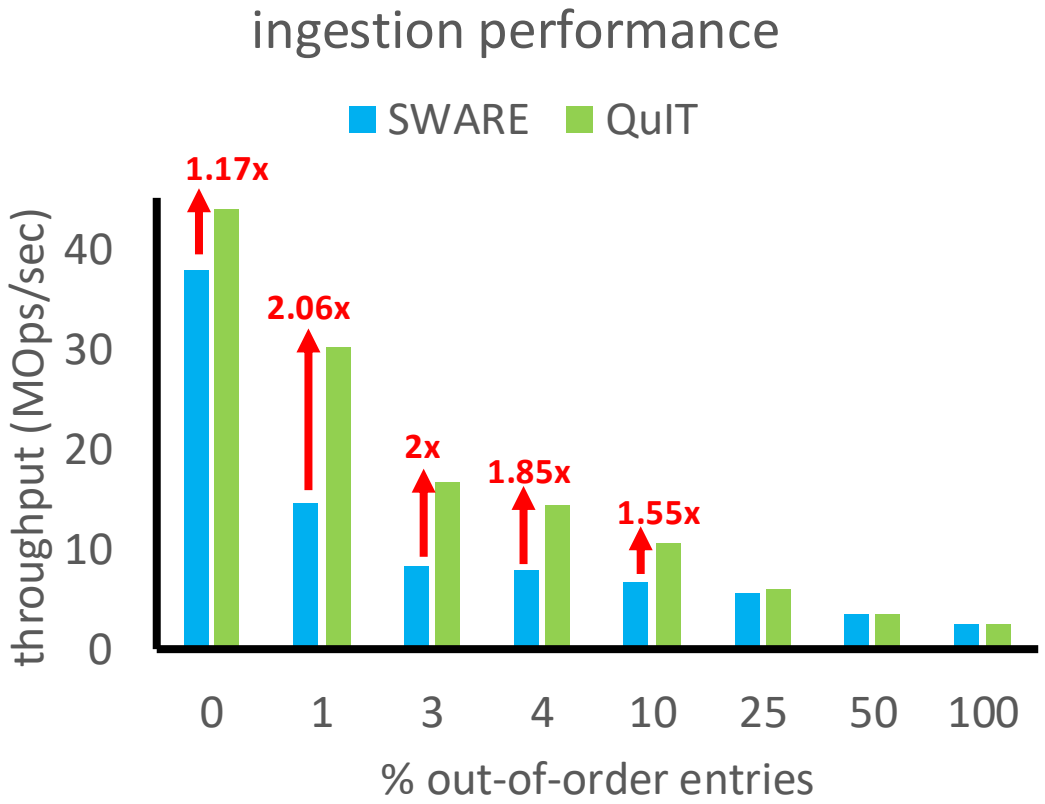
- Intel Xeon Gold 5230
- 2.1GHZ processor w. 20 cores
- 384GB RAM, 28MB L3 cache

Index Setup:

- Node size = 4KB
- Entire index in memory
- fuzzy scale in ***IKR*** = 1.5
- 500M entries (4B + 4B)

BOSTON UNIVERSITY

QuIT v/s SWARE



integrate SWARE with same B⁺-tree as QuIT

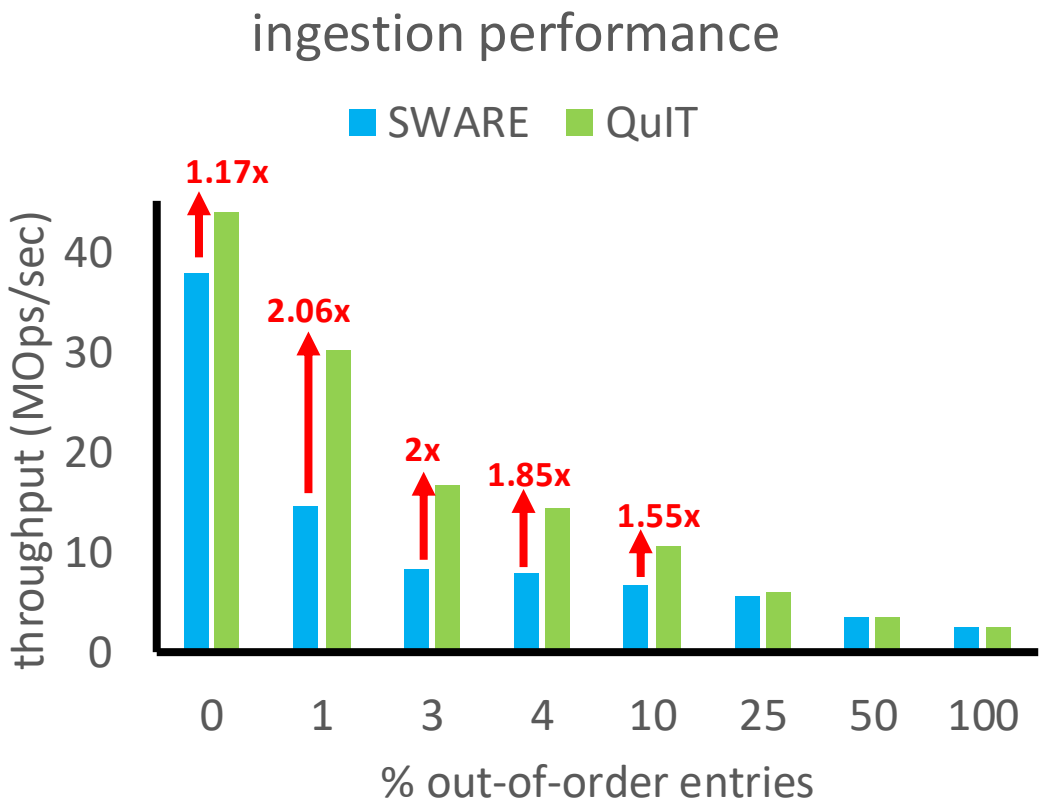
up to 2.06x faster

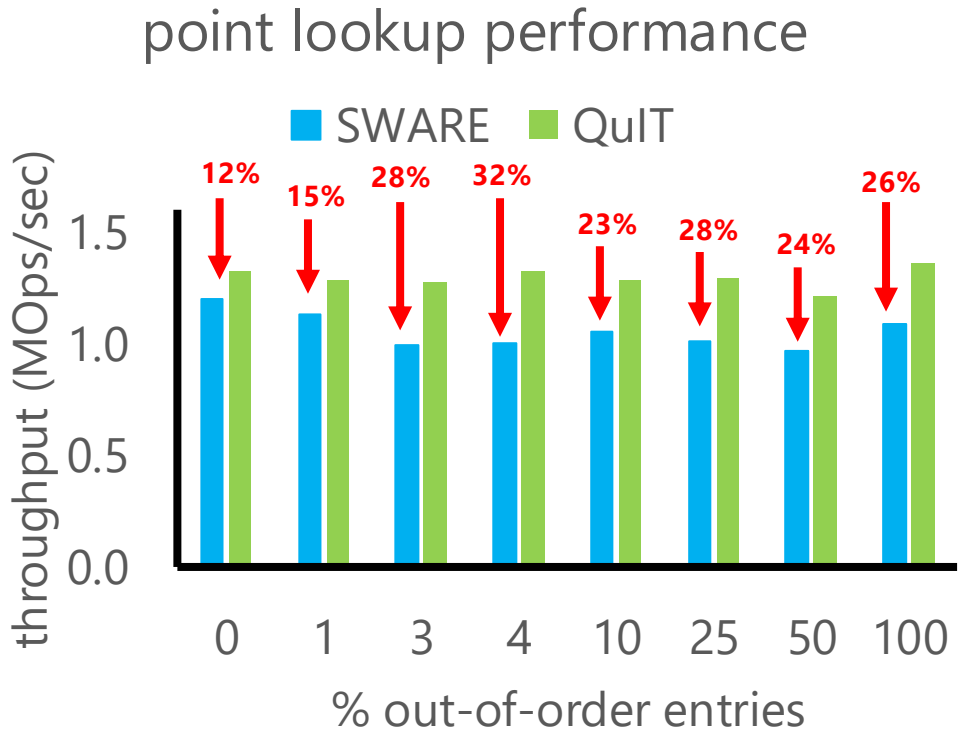
avoids buffer management ✓

minimal metadata ✓

BOSTON UNIVERSITY

QuIT v/s SWARE

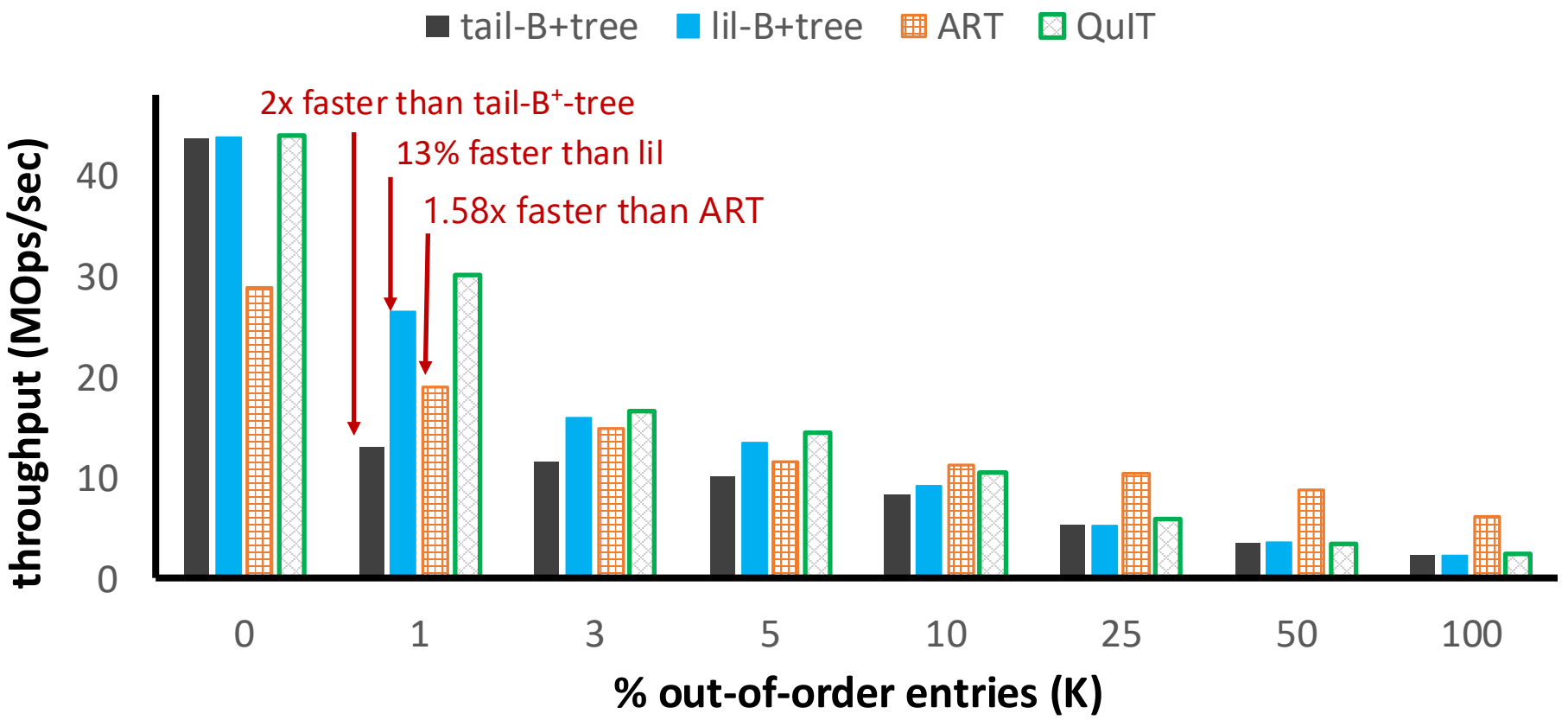




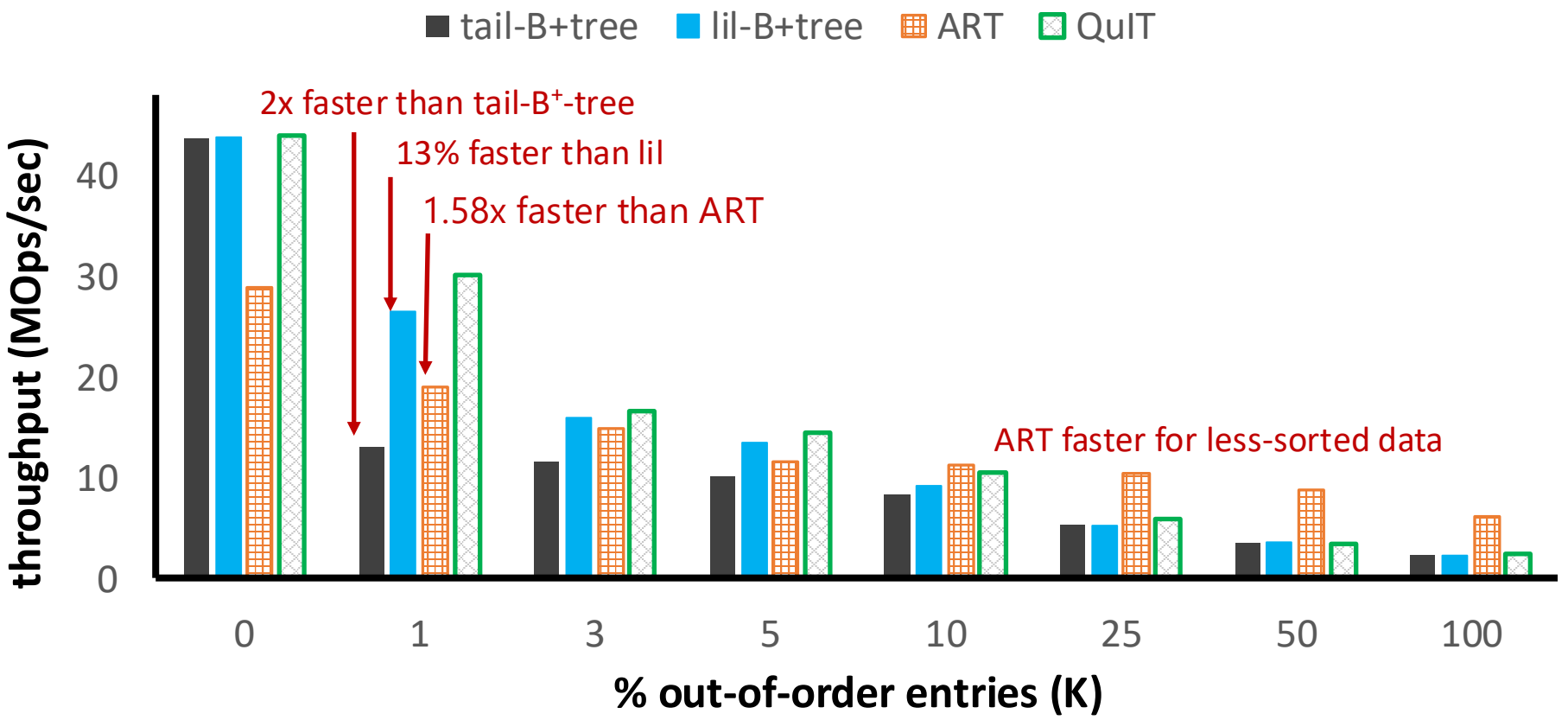
no buffering ⇒ no read overhead!

BOSTON UNIVERSITY

QuIT Performs Best With Near-Sorted Data

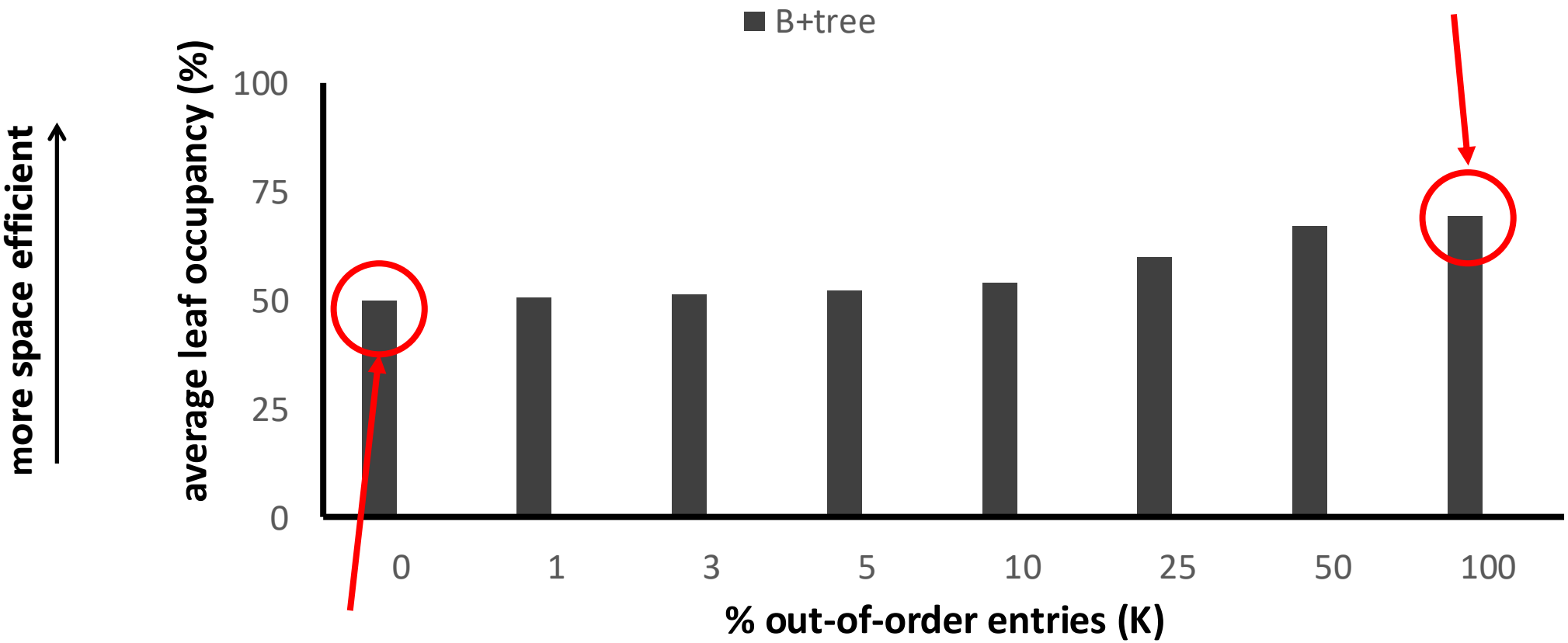


QuIT Performs Best With Near-Sorted Data



QuIT is Space Efficient

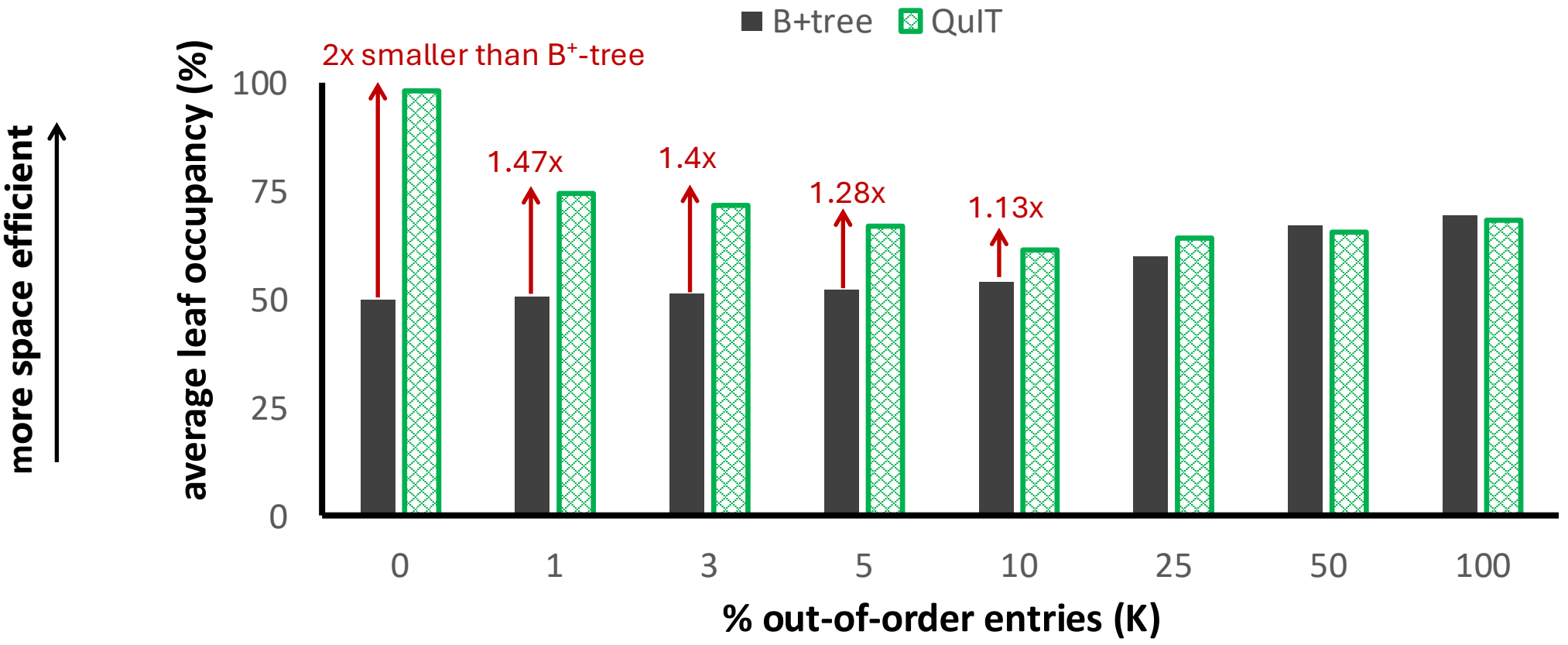
fully scrambled data → textbook 67%



full sortedness → worst case (unless bulk loading)

BOSTON UNIVERSITY

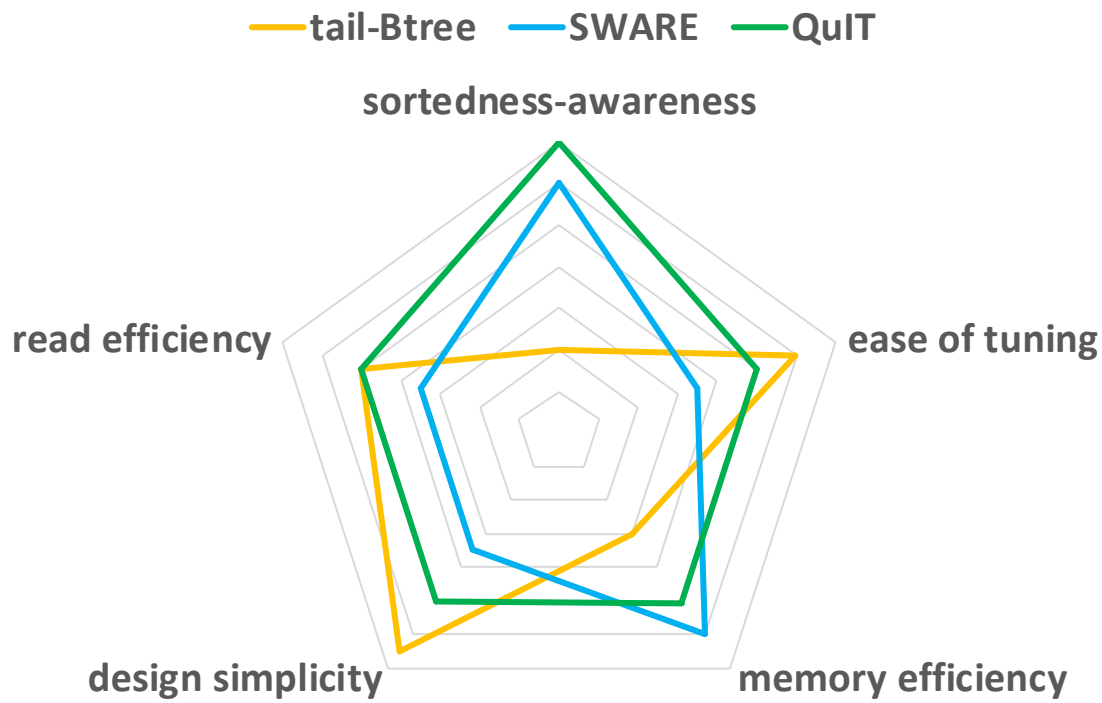
QuIT is Space Efficient



higher leaf occupancy reduces memory footprint✓

BOSTON UNIVERSITY

Qualitatively Comparing SWARE & QuIT



QuIT offers: **higher** sortedness-awareness + **no** read penalty + **minimal** design & tuning complexity

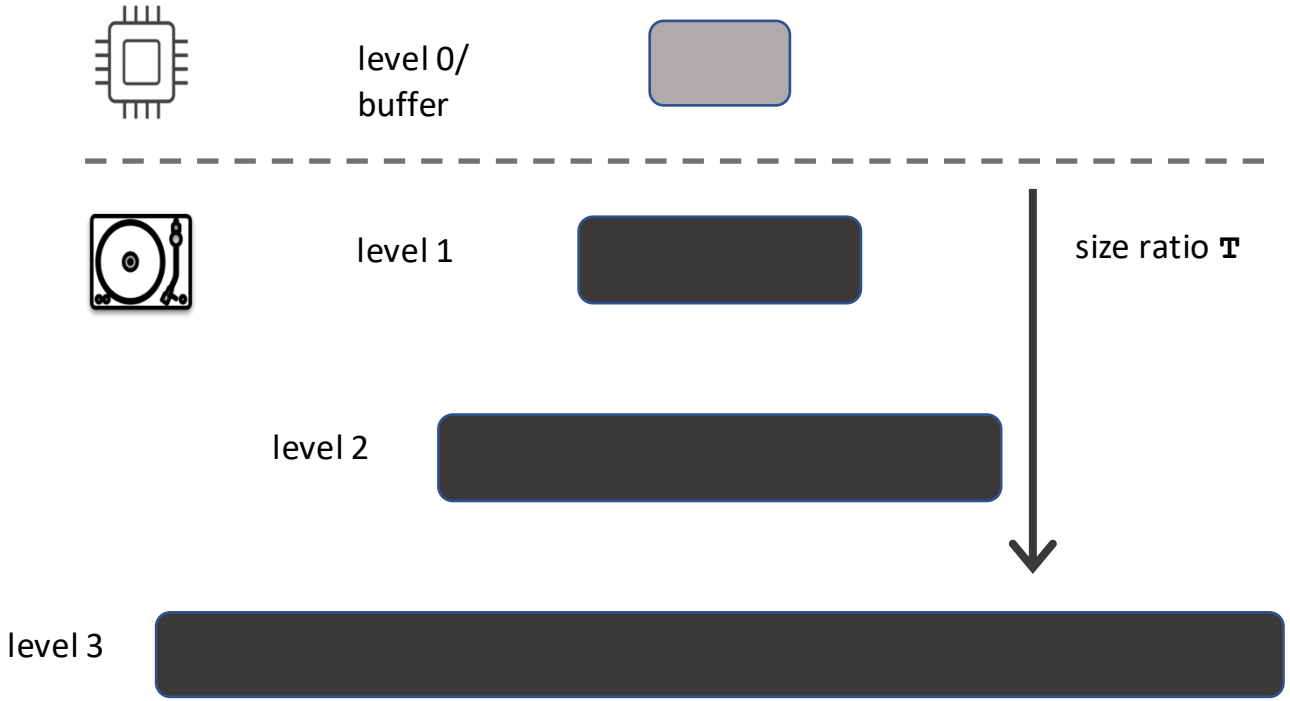
BOSTON UNIVERSITY

LSM trees are meant to be ingestion optimized!

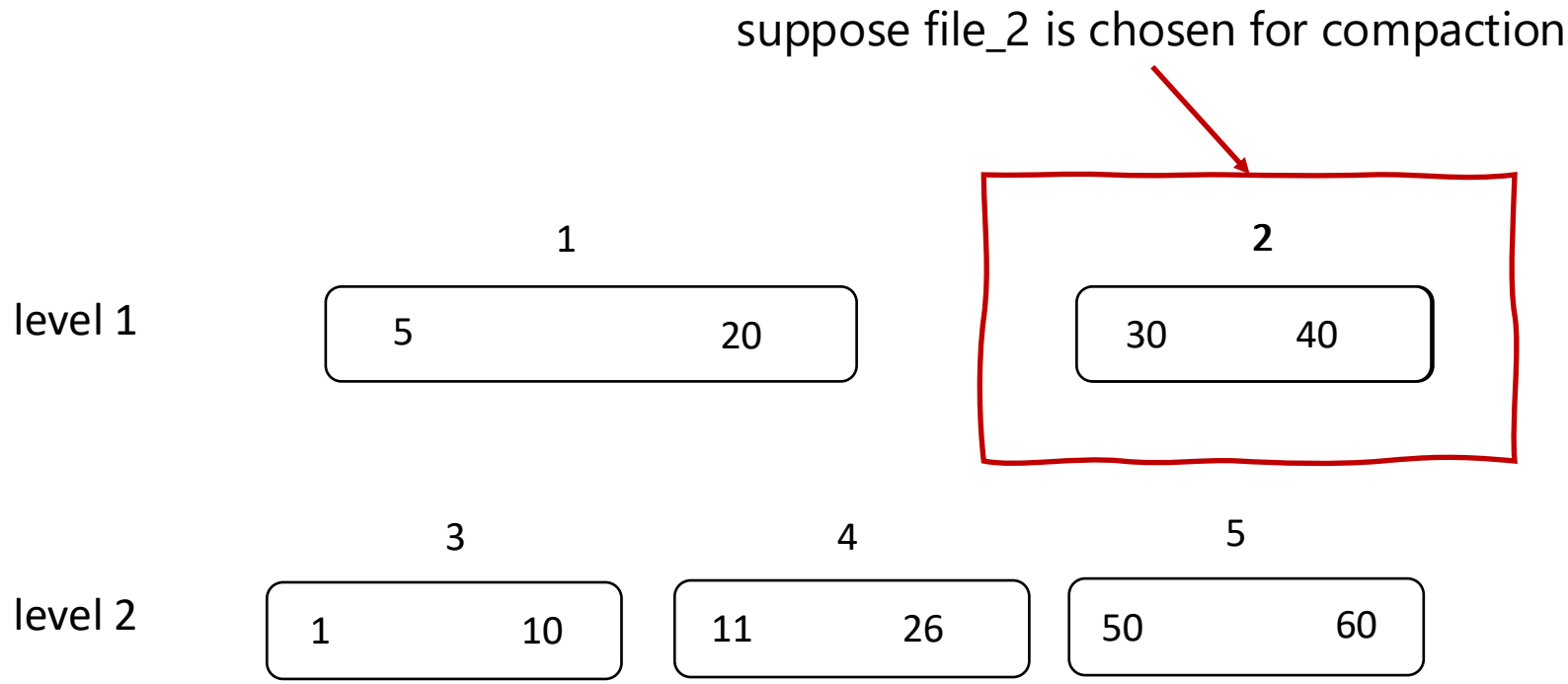
How about data sortedness?

BOSTON UNIVERSITY

Can LSM-trees Capture Data Sortedness?



Trivial Moves

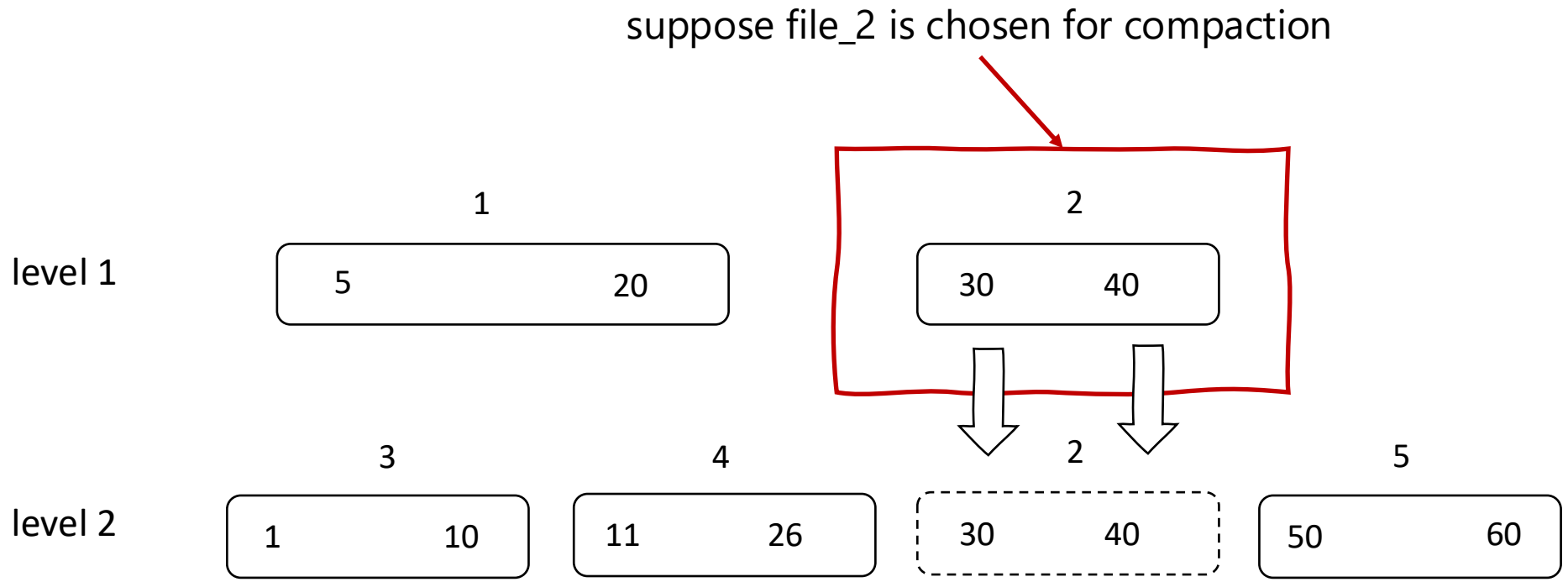


compaction would re-write file_2 in level_2

BUT, file_2 has no overlapping keys in level_2

Trivial Moves saves wasteful effort!

Trivial Moves

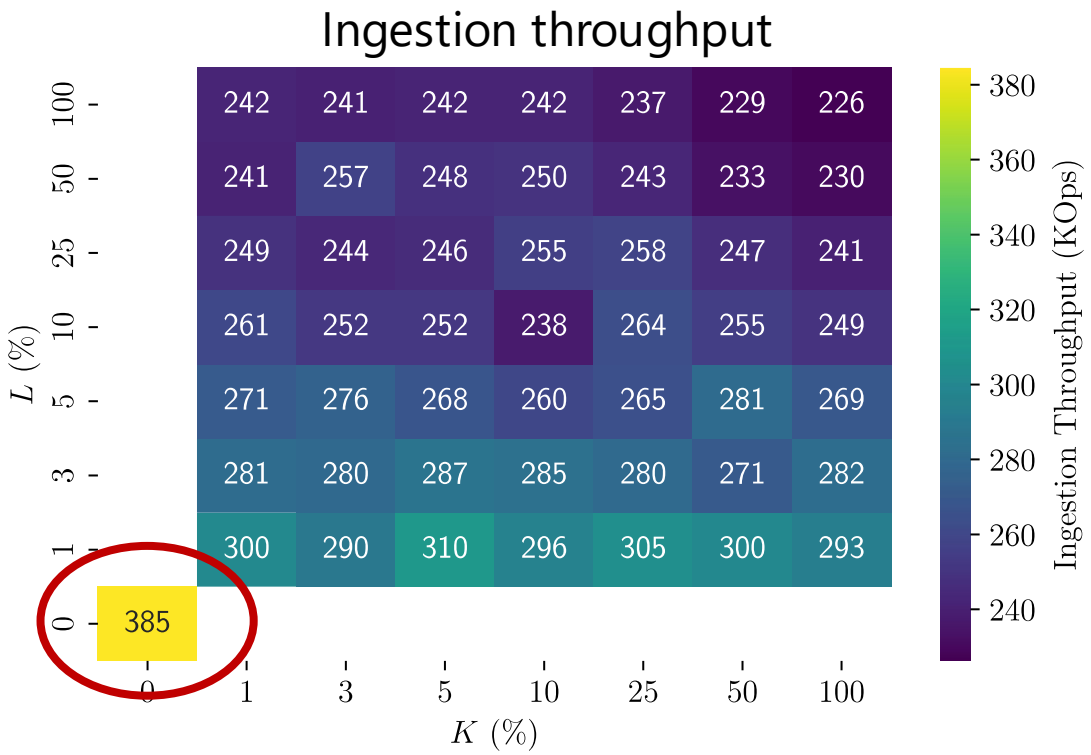


compaction would re-write file_2 in level_2

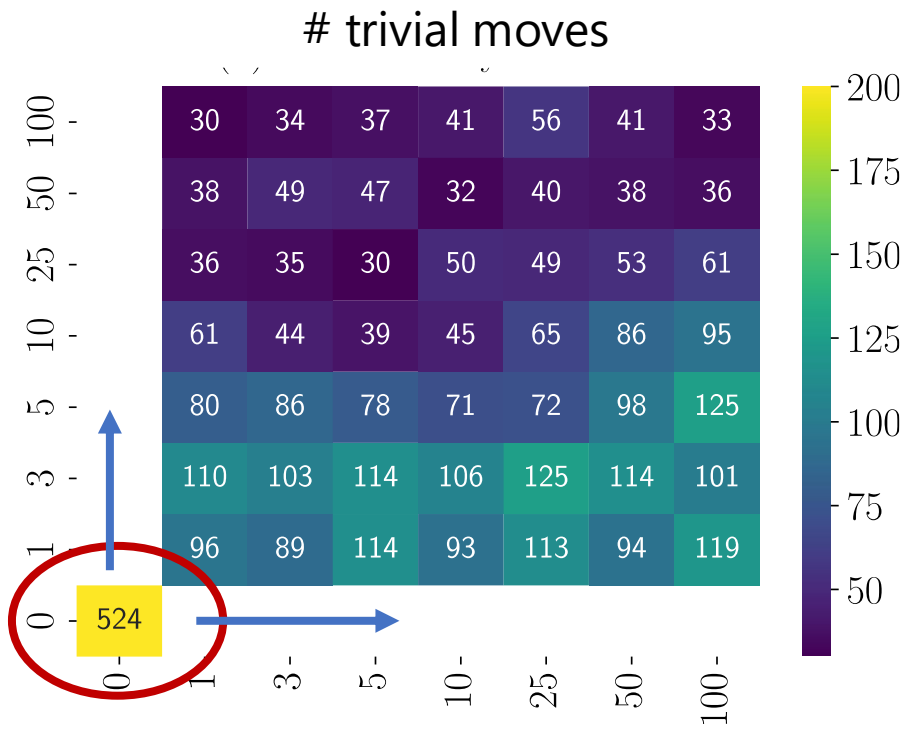
BUT, file_2 has no overlapping keys in level_2

Trivial Moves saves wasteful effort!

LSM Benefits from Trivial Moves

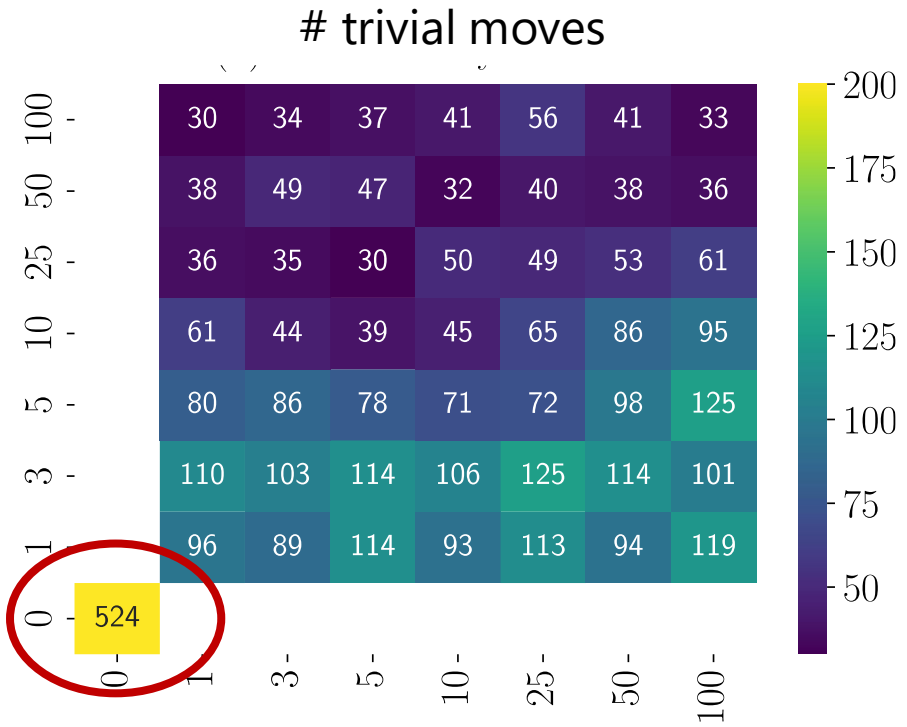


Trivial moves improve ingestion throughput

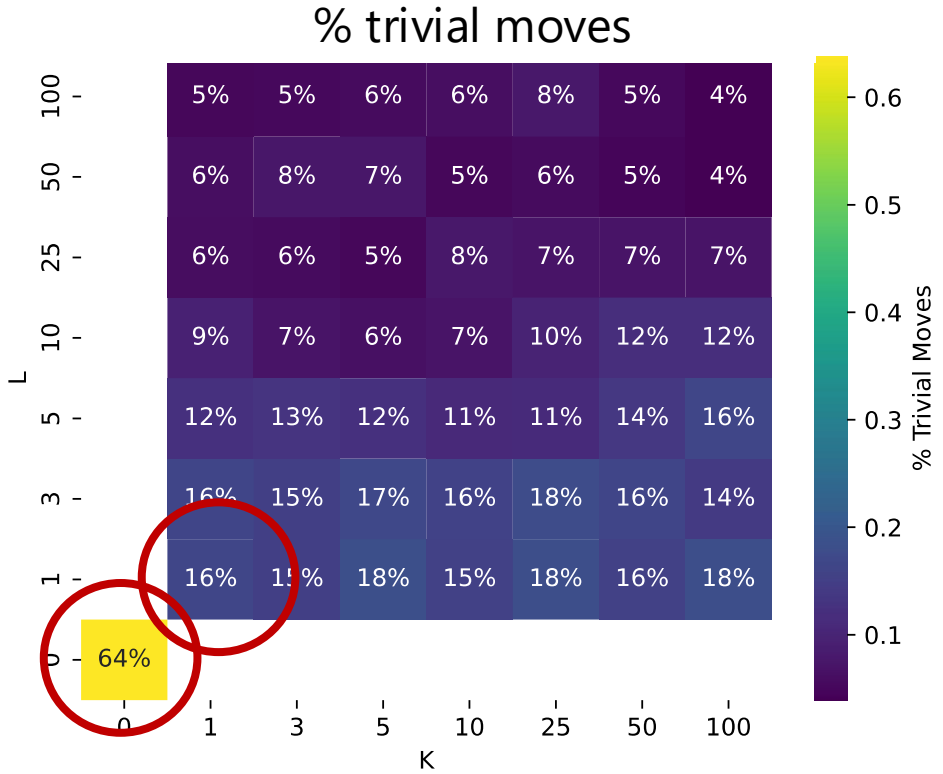


Yet, there is a significant drop in trivial moves even with minor unorderness

LSM Benefits from Trivial Moves



Yet, there is a significant drop in trivial moves even with minor unorderness



A large headroom for improvement!

RocksDB always compacts in level 0

SWARE buffering would further reduce this!

BOSTON UNIVERSITY

CS 561: Data Systems Architectures

Indexing for Near-Sorted Data

Prof. Manos Athanassoulis

https://bu-disc.github.io/CS561/

Additional Details on SWARE and QuIT

Sortedness-Aware (SWARE) Paradigm



intelligent buffering

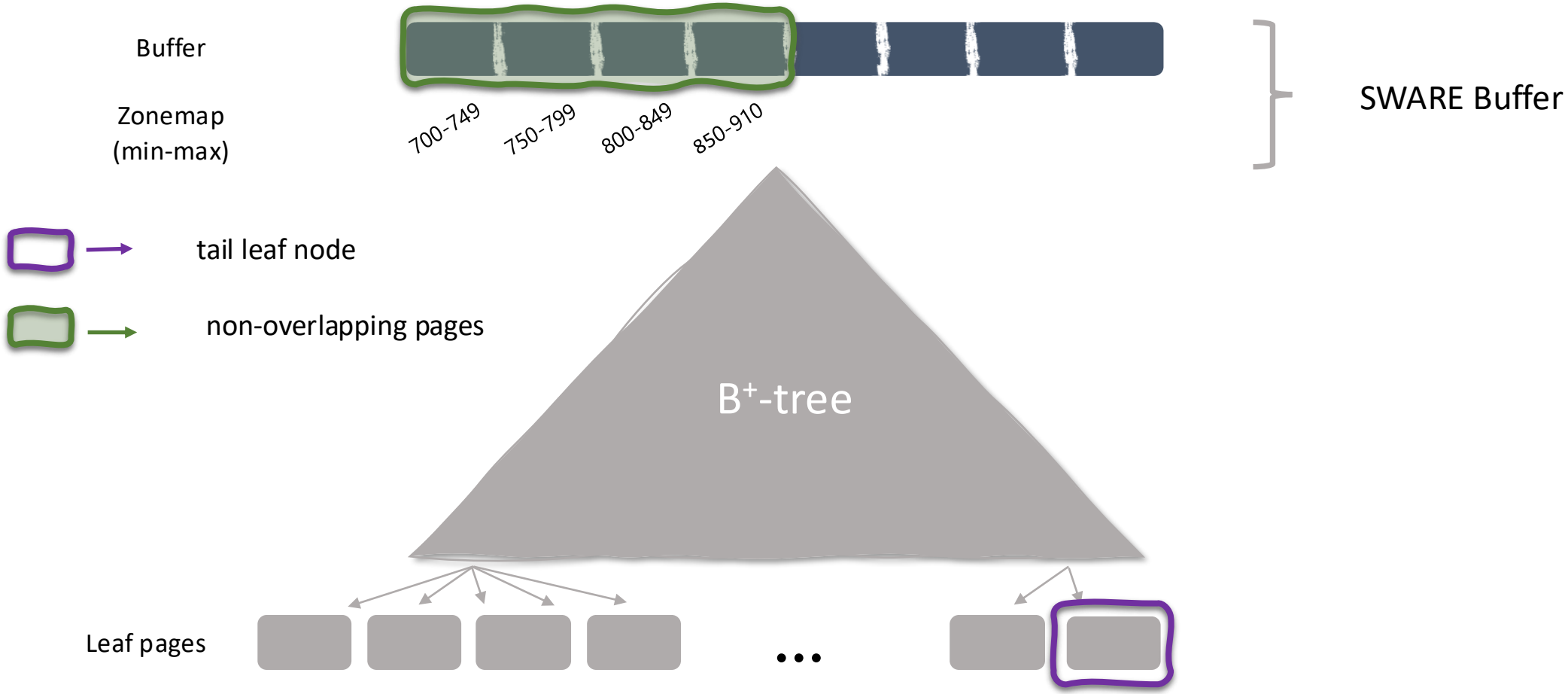
opportunistic bulk loading

increased fill and split factor

SWARE framework can be applied to any tree-index!

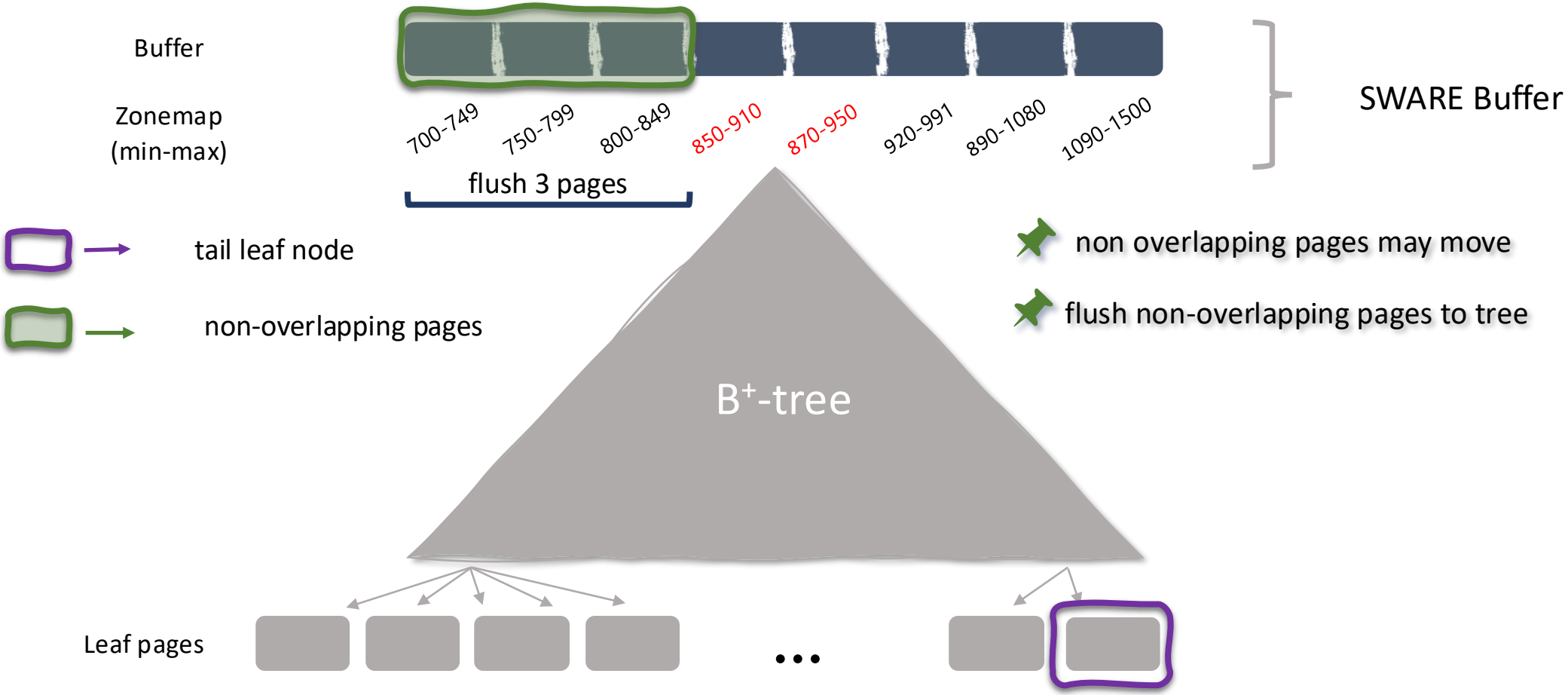
BOSTON UNIVERSITY

SWARE Ingestions



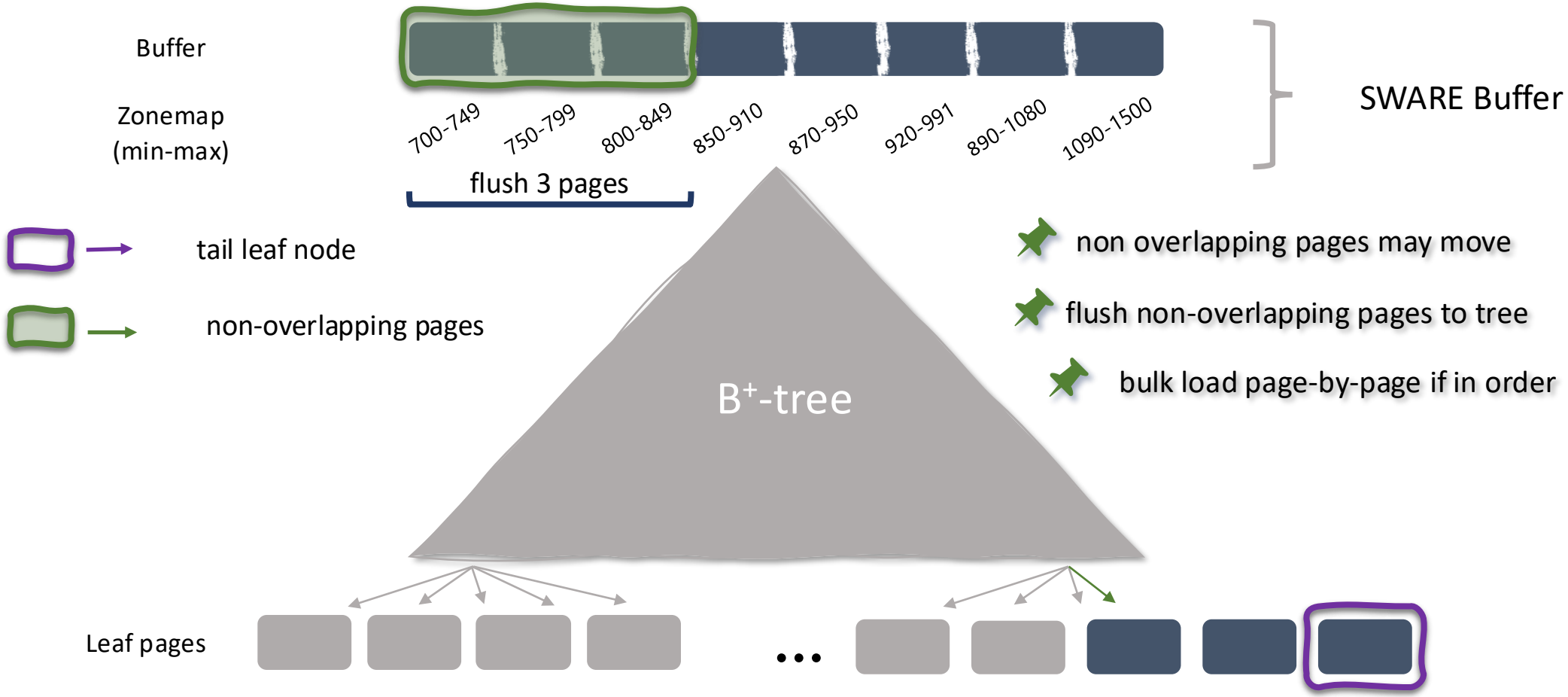
BOSTON
UNIVERSITY

SWARE Ingestions



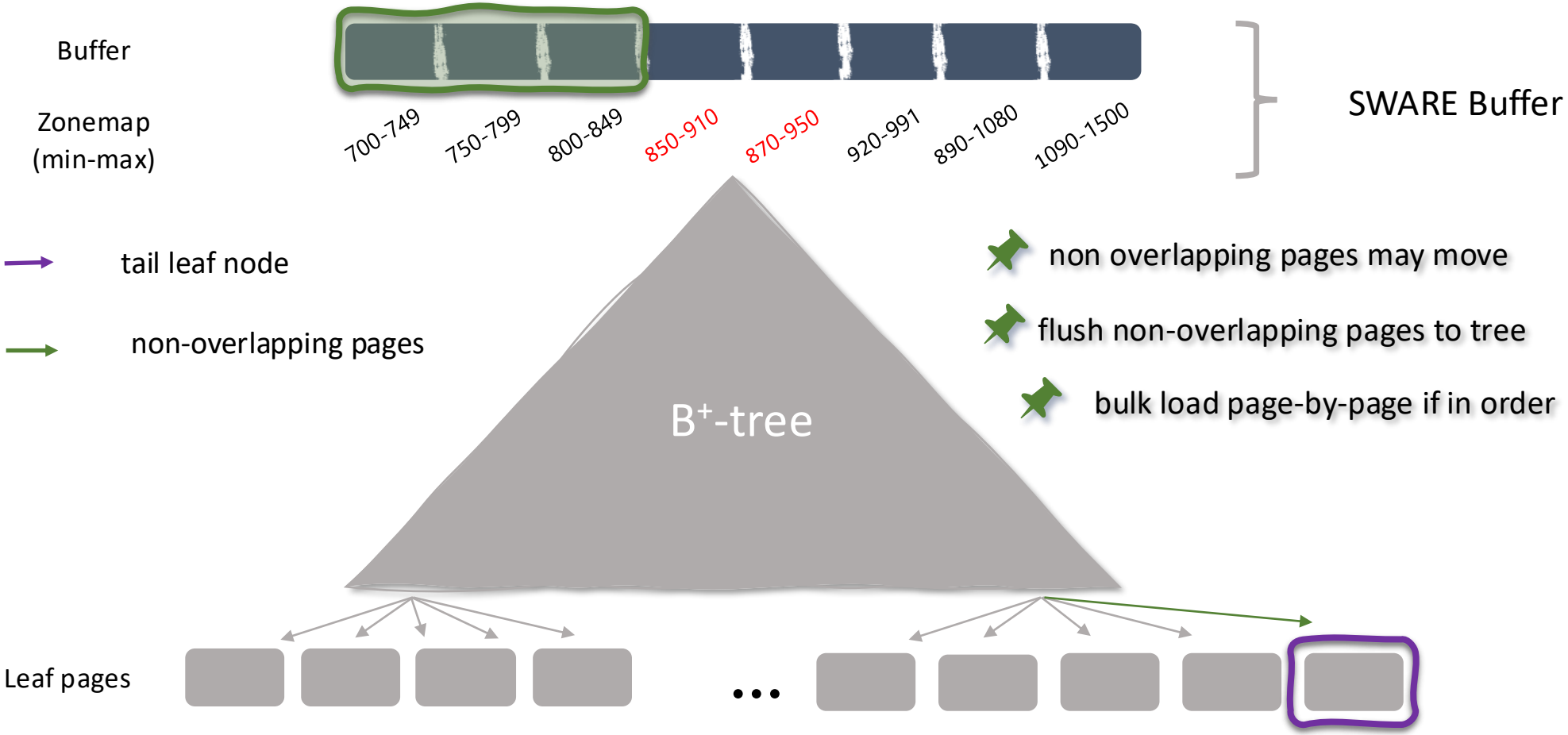
BOSTON UNIVERSITY

SWARE Ingestions



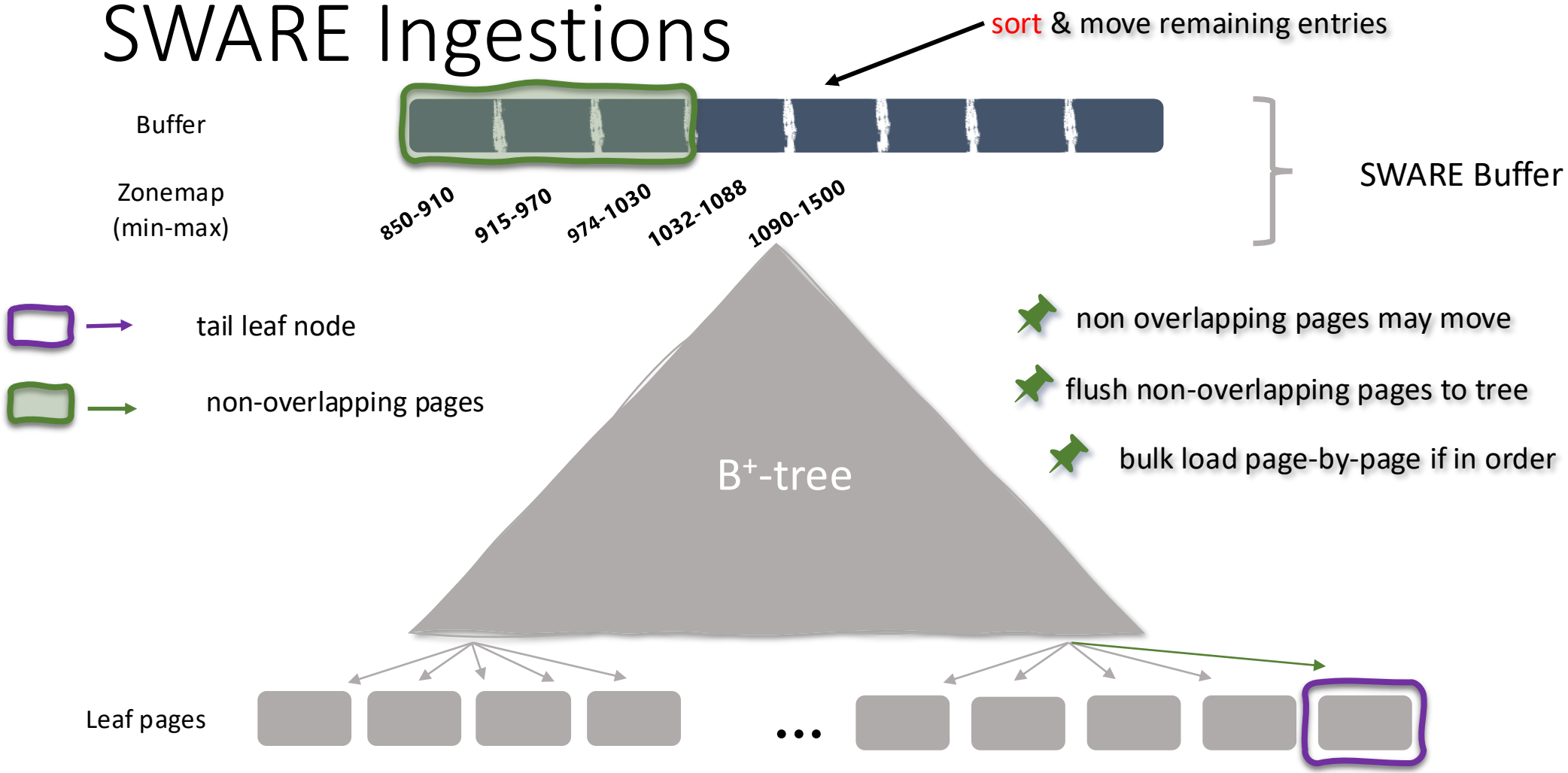
BOSTON UNIVERSITY

SWARE Ingestions



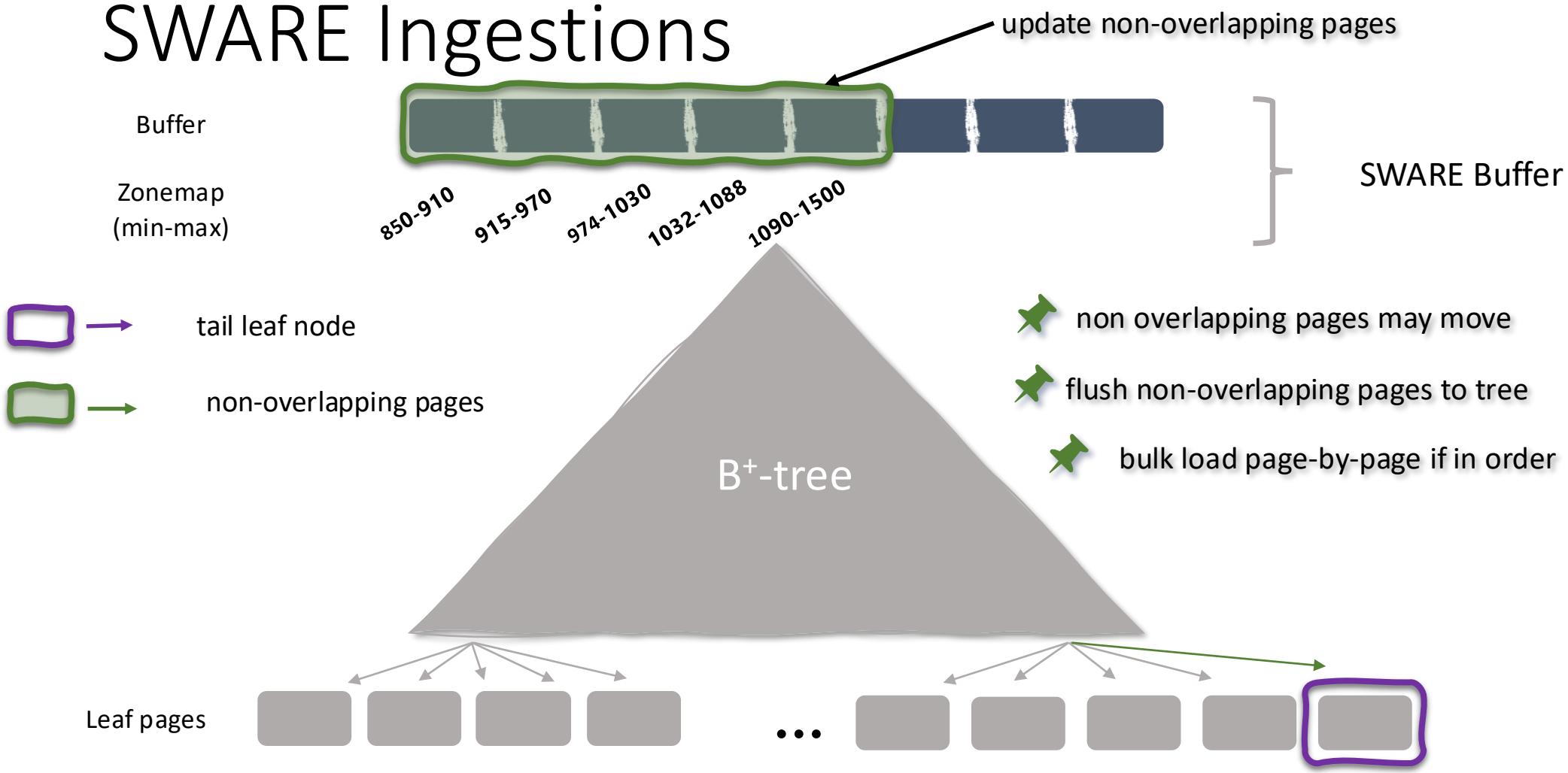
BOSTON UNIVERSITY

SWARE Ingestions



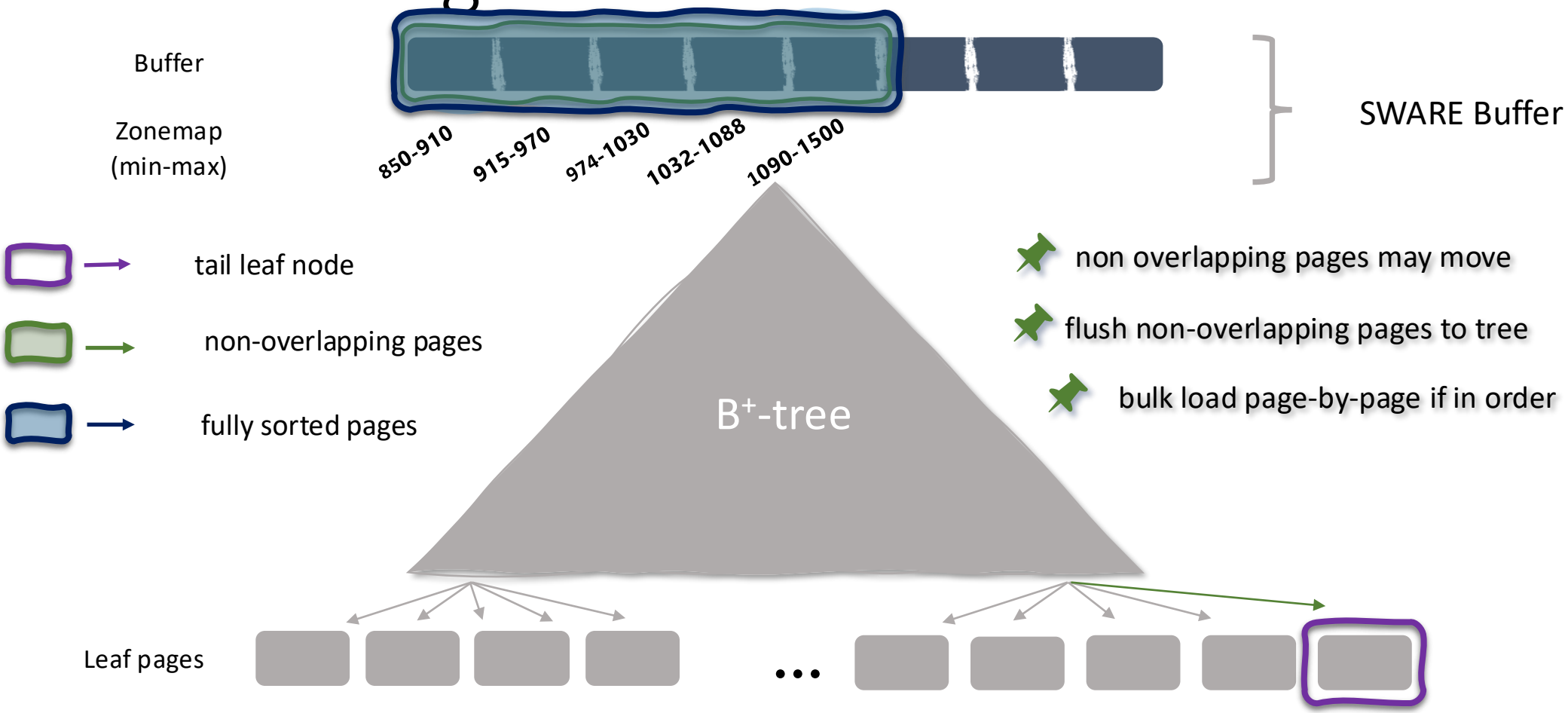
BOSTON UNIVERSITY

SWARE Ingestions



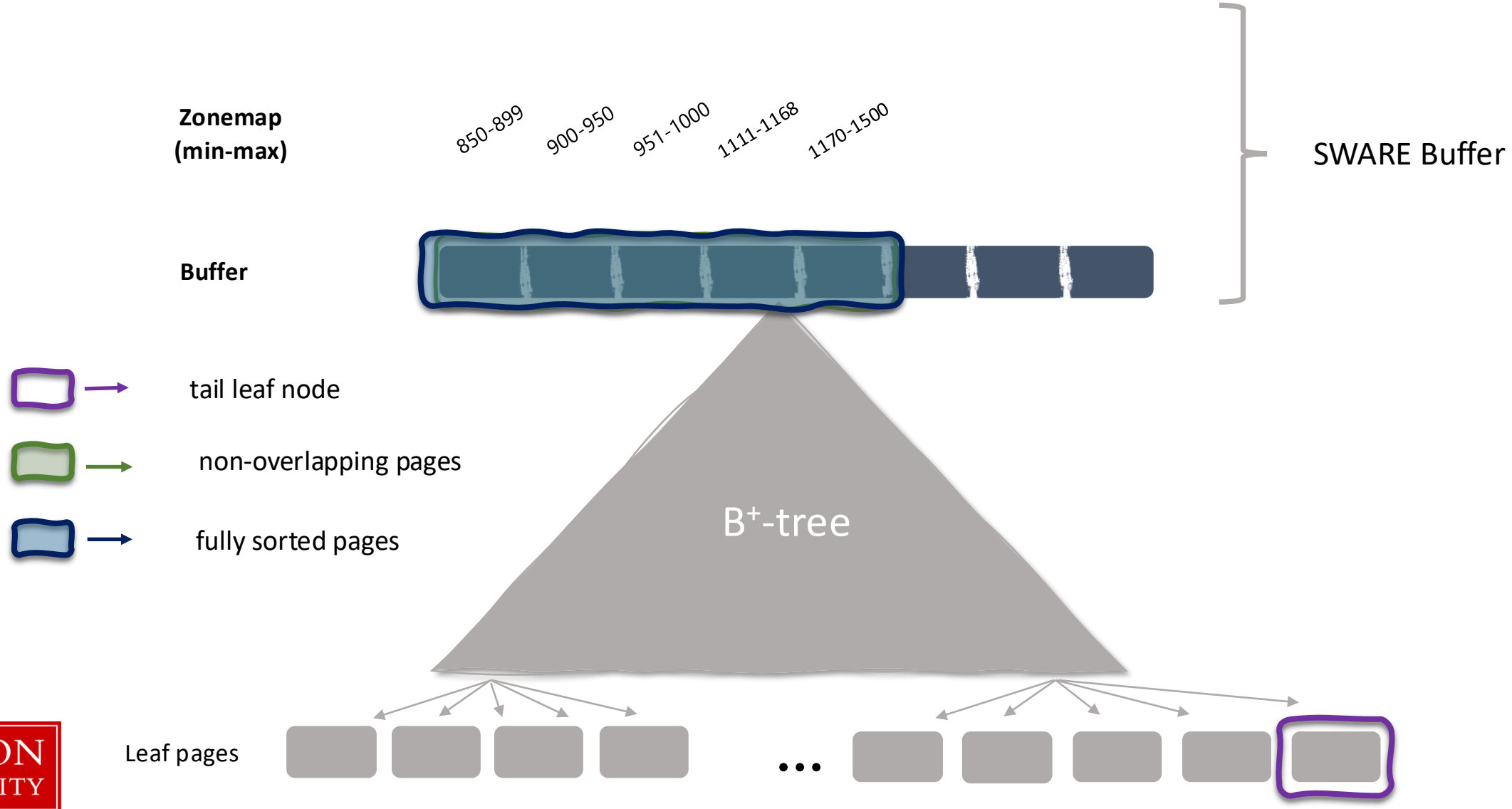
BOSTON UNIVERSITY

SWARE Ingestions



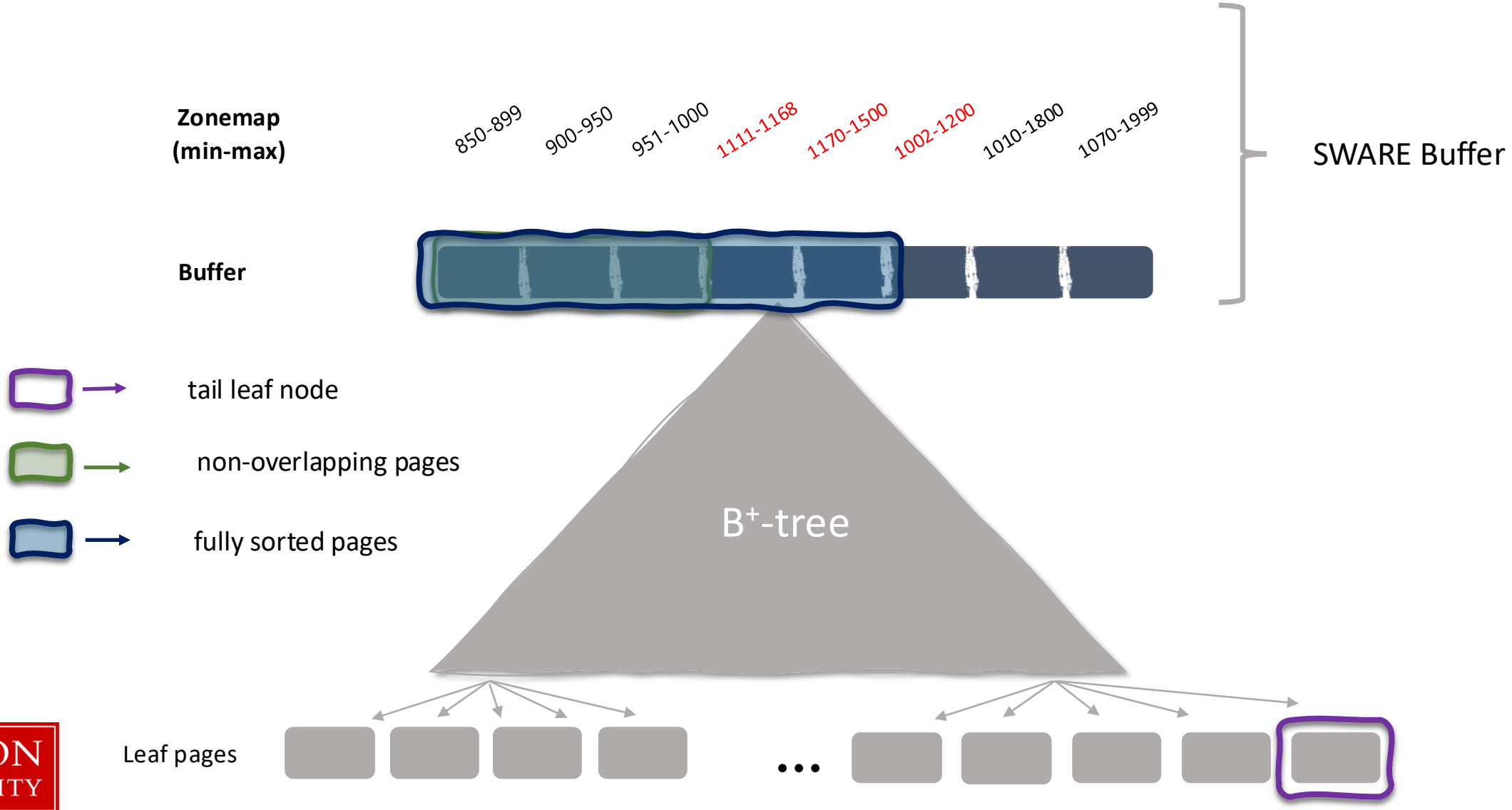
BOSTON UNIVERSITY

SWARE Lookups



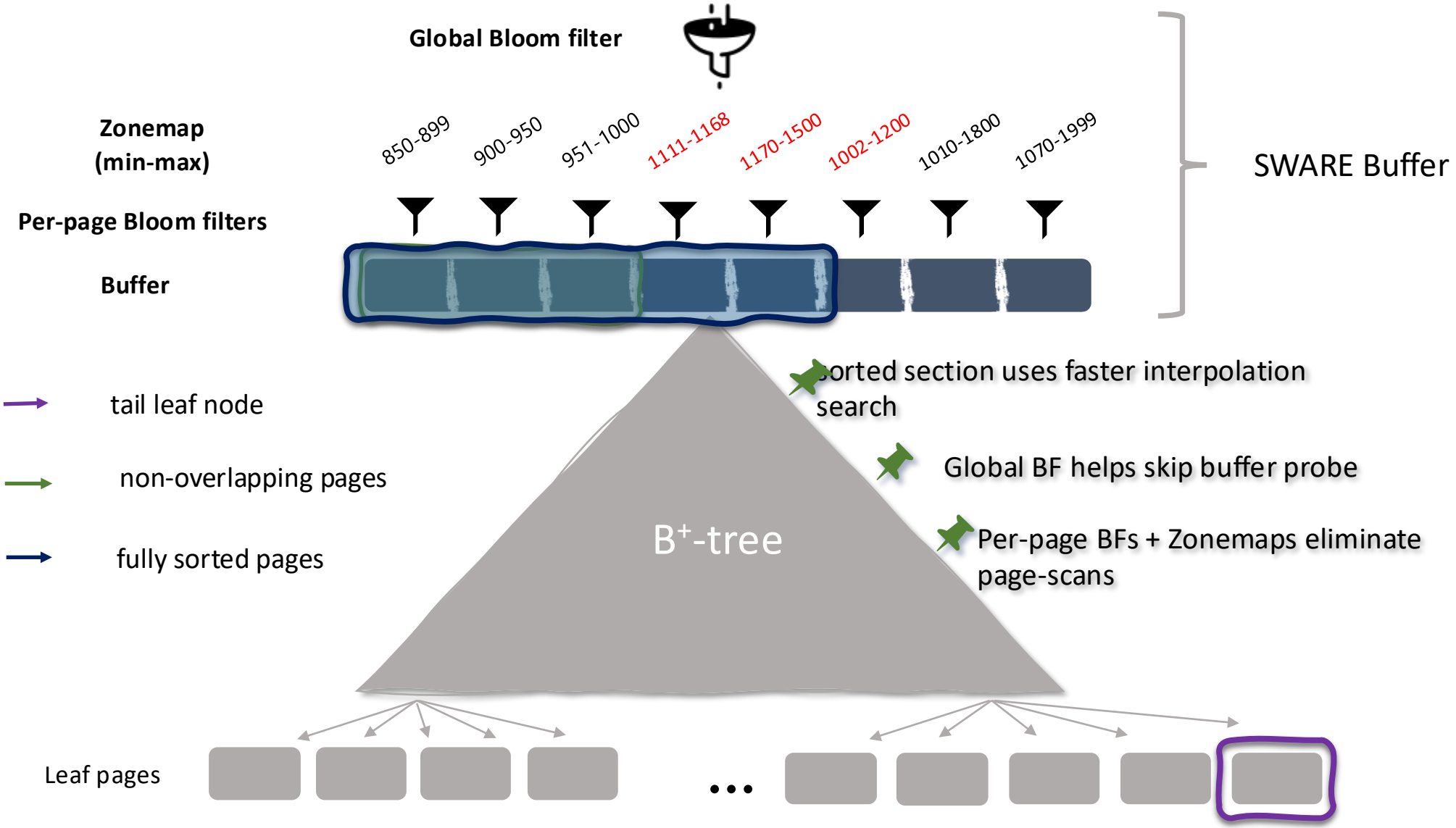
BOSTON UNIVERSITY

SWARE Lookups



BOSTON UNIVERSITY

SWARE Lookups



BOSTON UNIVERSITY

Experimental Evaluation

System Setup:

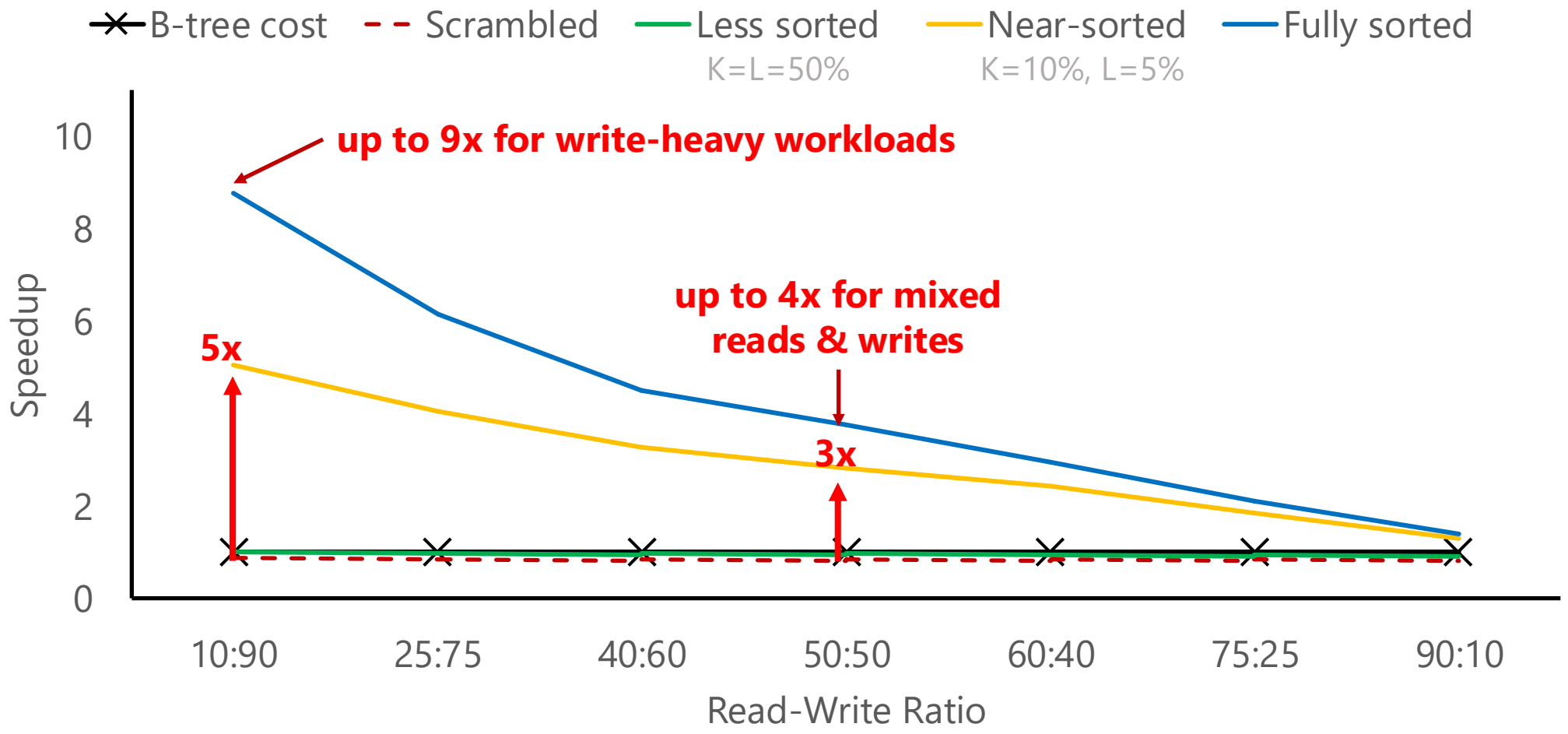
- Intel Xeon Gold 5230
- 2.1GHZ processor w. 20 cores
- 384GB RAM, 28MB L3 cache

Index Setup:

- Buffer = 40MB; flush <= 50%
- BFs = 10 BPK; Murmur Hash
- Split at 80%

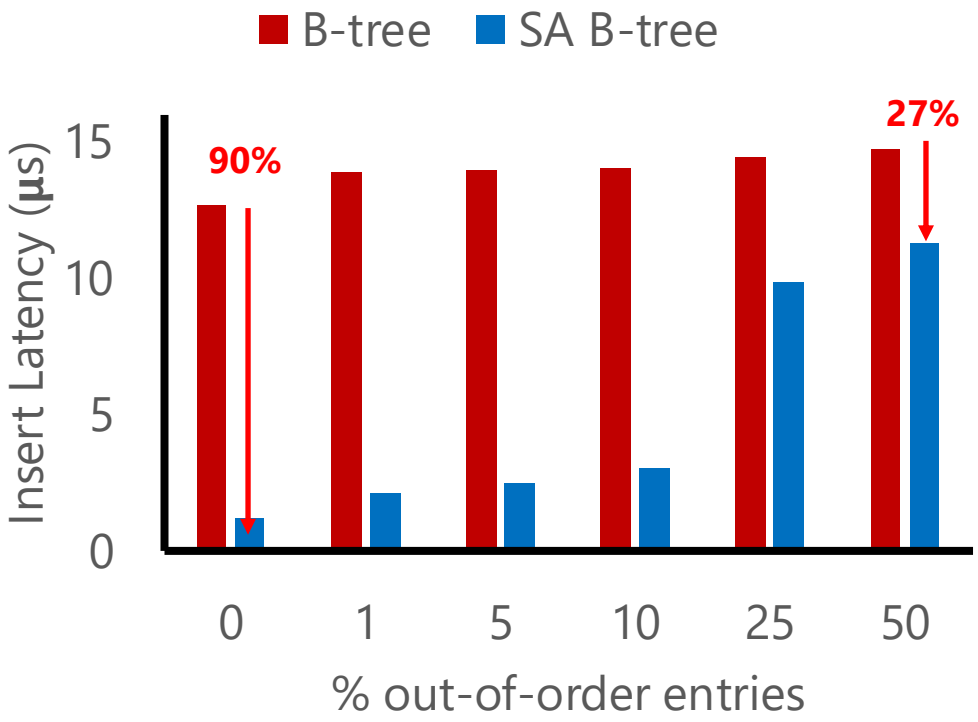
B^+-tree design inspired by STX::B-tree can also work as B^ε-tree

Evaluating SWARE Under Varying Sortedness

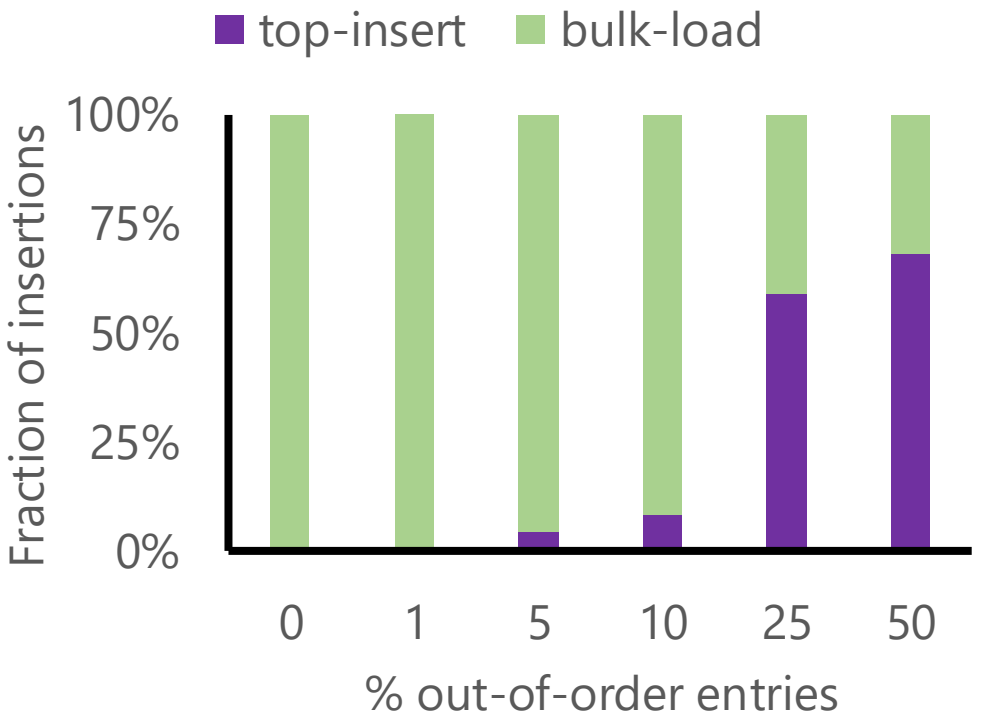


BOSTON UNIVERSITY

Raw Ingestion Performance



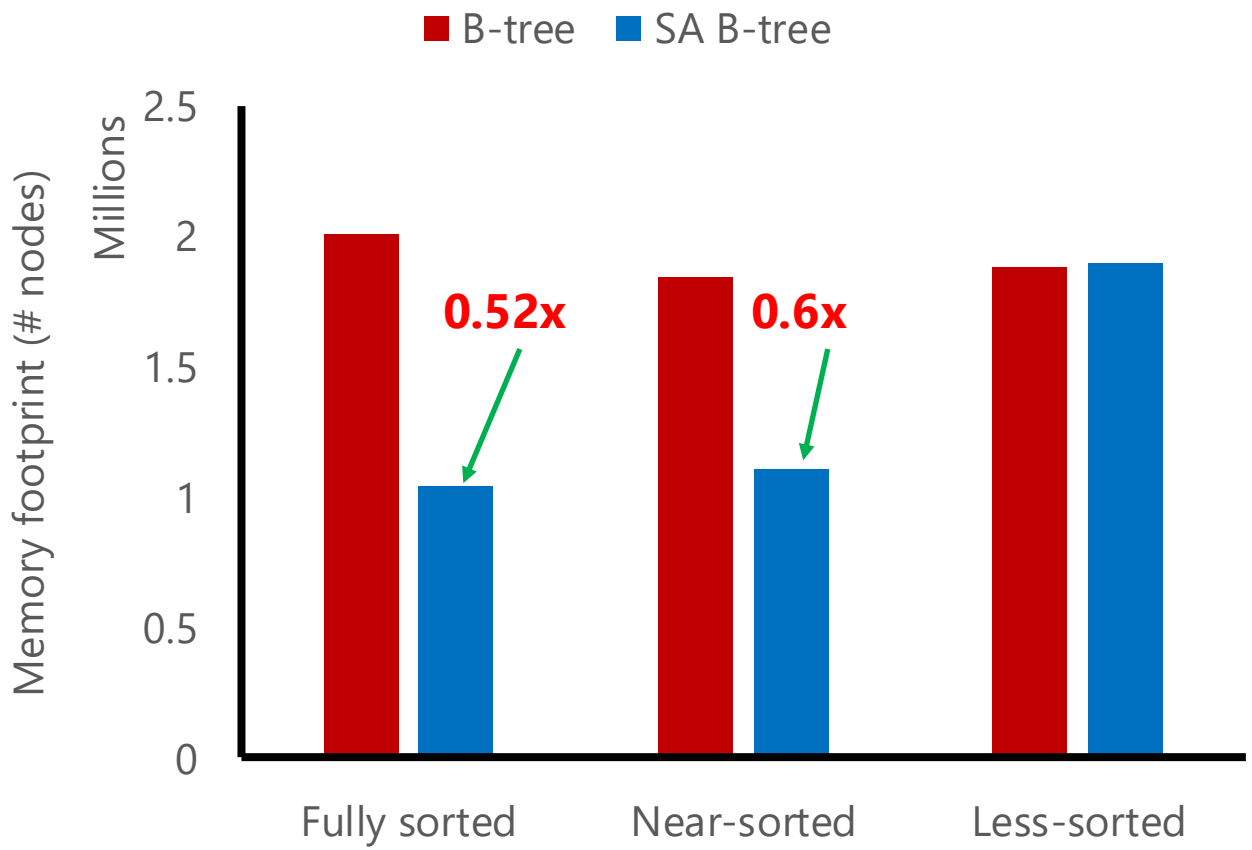
ingestion latency reduced between 27-90%



bulk loading maximized with high data sortedness

BOSTON UNIVERSITY

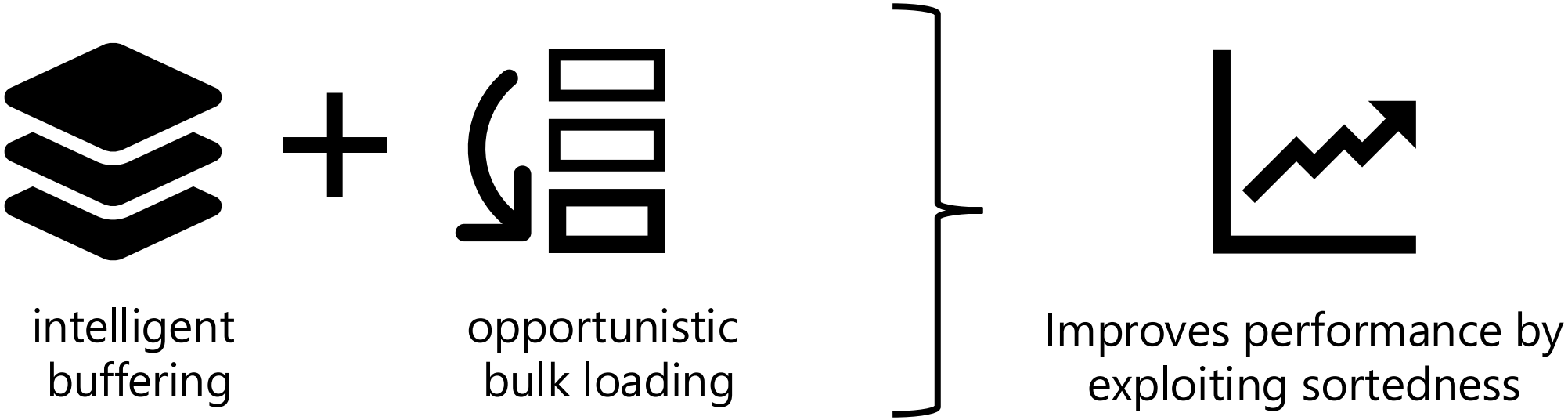
SWARE Improves Space Utilization



increased fill/split factor helps reduce memory footprint

BOSTON UNIVERSITY

Summarizing SWARE [ICDE 2023]



intelligent buffering

opportunistic bulk loading

Improves performance by exploiting sortedness

BUT, also increases Complexity in Design!

Concurrency Control in QuIT

```
metadata.writeLock();

if(key within pole range){
      pole.writeLock();

      if(pole is at CAPACITY){
            top_insert(key, value);
            metadata.unlock();
      }
      else{
            metadata.update(fp_size);
            metadata.unlock();
            pole.insert(key, value);
      }
}
else{
      metadata.unlock();
      top_insert(key, value);
}
```

lock-crabbing for top inserts
+
careful lock handling for fast-path

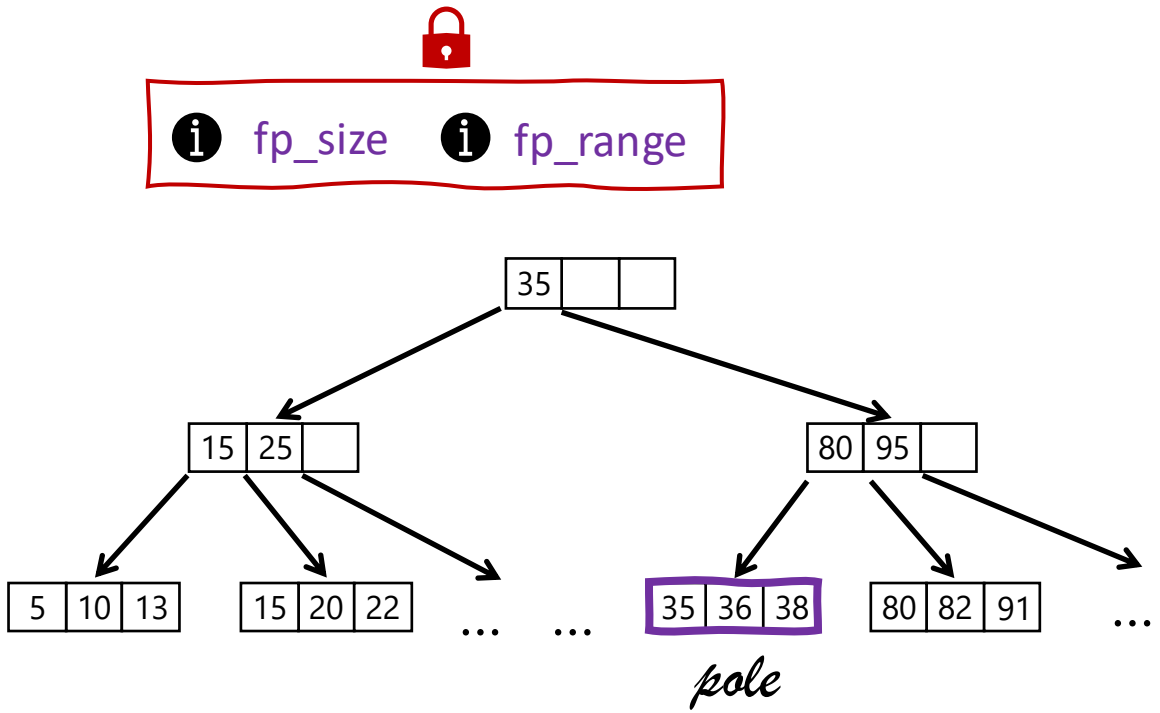
Concurrency Control in QuIT

```
metadata.writeLock();

if(key within pole range){
     pole.writeLock();

     if(pole is at CAPACITY){
          top_insert(key, value);
          metadata.unlock();
     }
     else{
          metadata.update(fp_size);
          metadata.unlock();
          pole.insert(key, value);
     }
}
else{
     metadata.unlock();
     top_insert(key, value);
}
```

lock meta-data to check for fast-insert



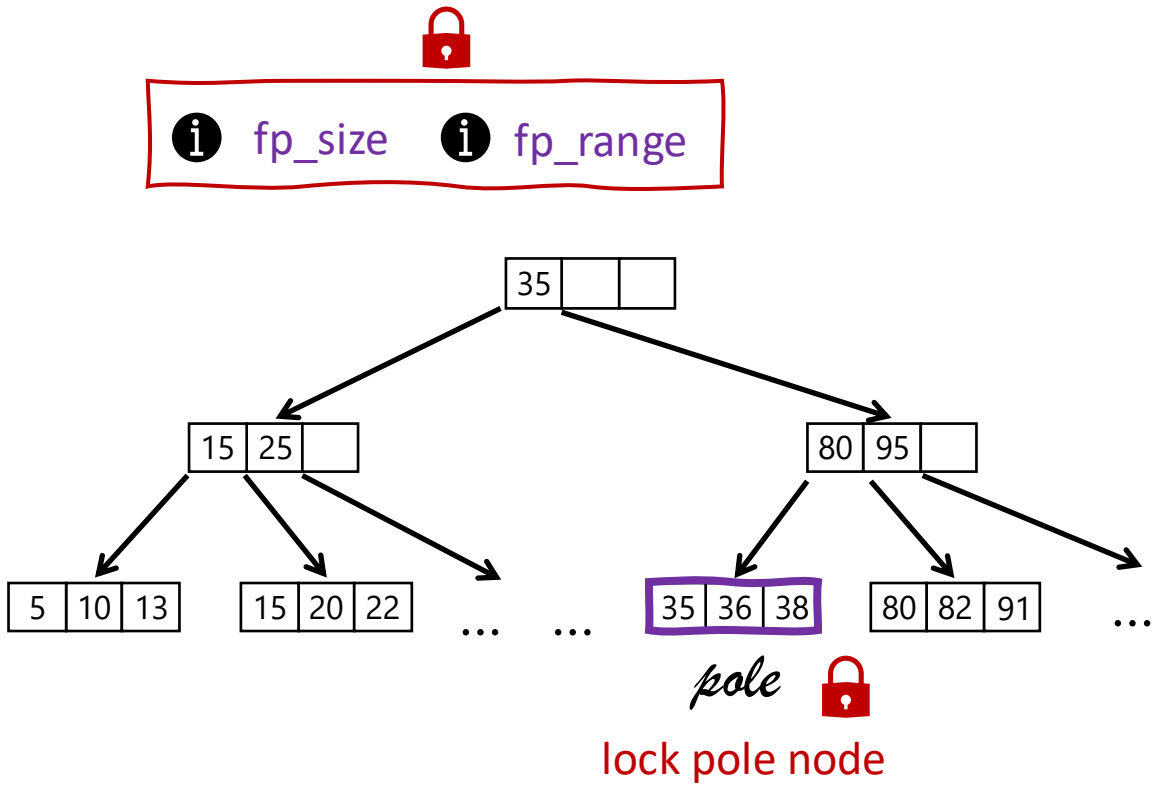
Concurrency Control in QuIT

```
metadata.writeLock();

if(key within pole range){
      pole.writeLock();

      if(pole is at CAPACITY){
            top_insert(key, value);
            metadata.unlock();
      }
      else{
            metadata.update(fp_size);
            metadata.unlock();
            pole.insert(key, value);
      }
}
else{
      metadata.unlock();
      top_insert(key, value);
}
```

lock meta-data to check for fast-insert



Concurrency Control in QuIT

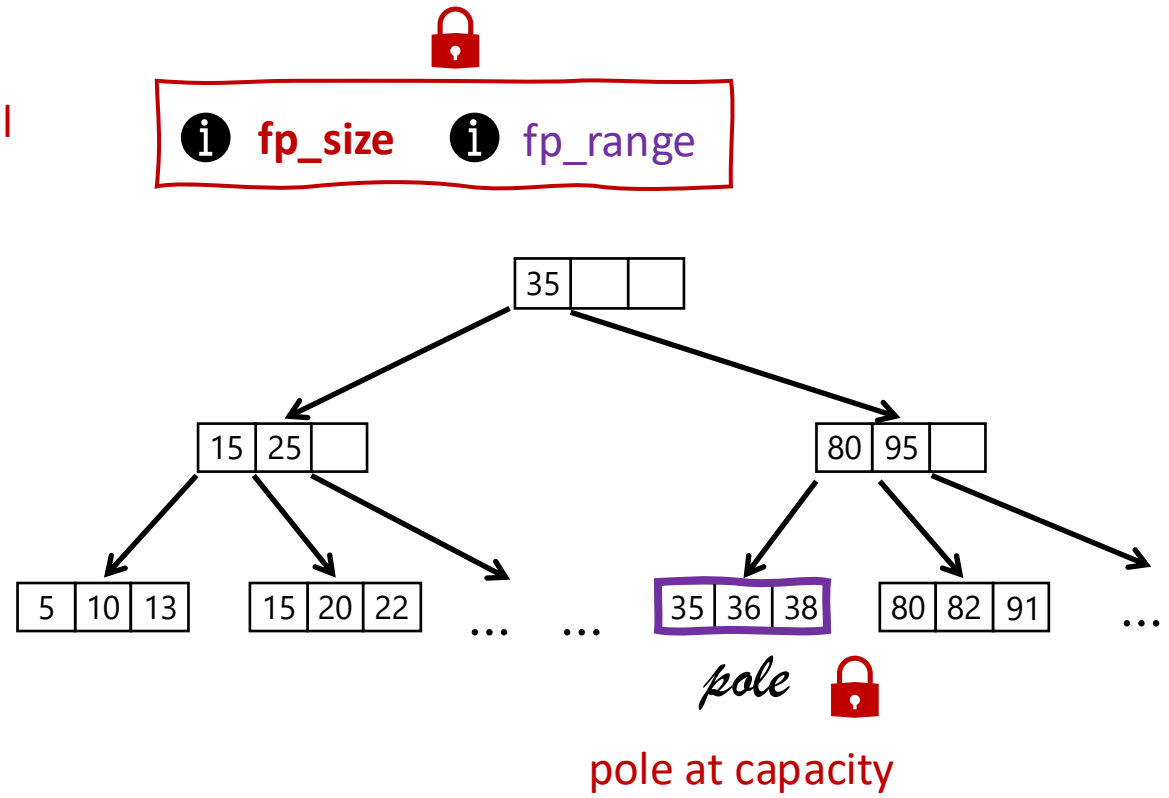
```
metadata.writeLock();

if(key within pole range){
      pole.writeLock();

      if(pole is at CAPACITY){
            top_insert(key, value);
            metadata.unlock();
      }
      else{
            metadata.update(fp_size);
            metadata.unlock();
            pole.insert(key, value);
      }
}
else{
      metadata.unlock();
      top_insert(key, value);
}
```

lock meta-data to check for fast-insert

check if pole is full



pole at capacity

Concurrency Control in QuIT

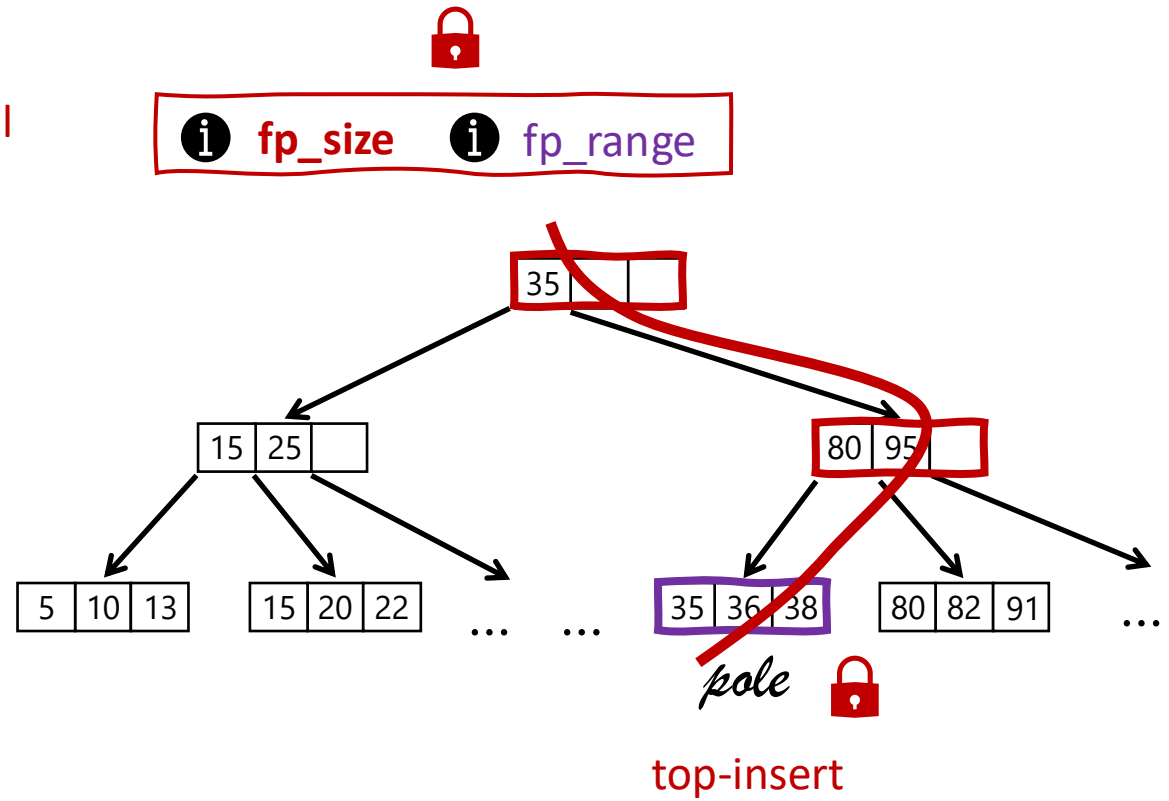
```
metadata.writeLock();

if(key within pole range){
      pole.writeLock();

      if(pole is at CAPACITY){
            top_insert(key, value);
            metadata.unlock();
      }
      else{
            metadata.update(fp_size);
            metadata.unlock();
            pole.insert(key, value);
      }
}
else{
      metadata.unlock();
      top_insert(key, value);
}
```

lock meta-data to check for fast-insert

check if pole is full



Concurrency Control in QuIT

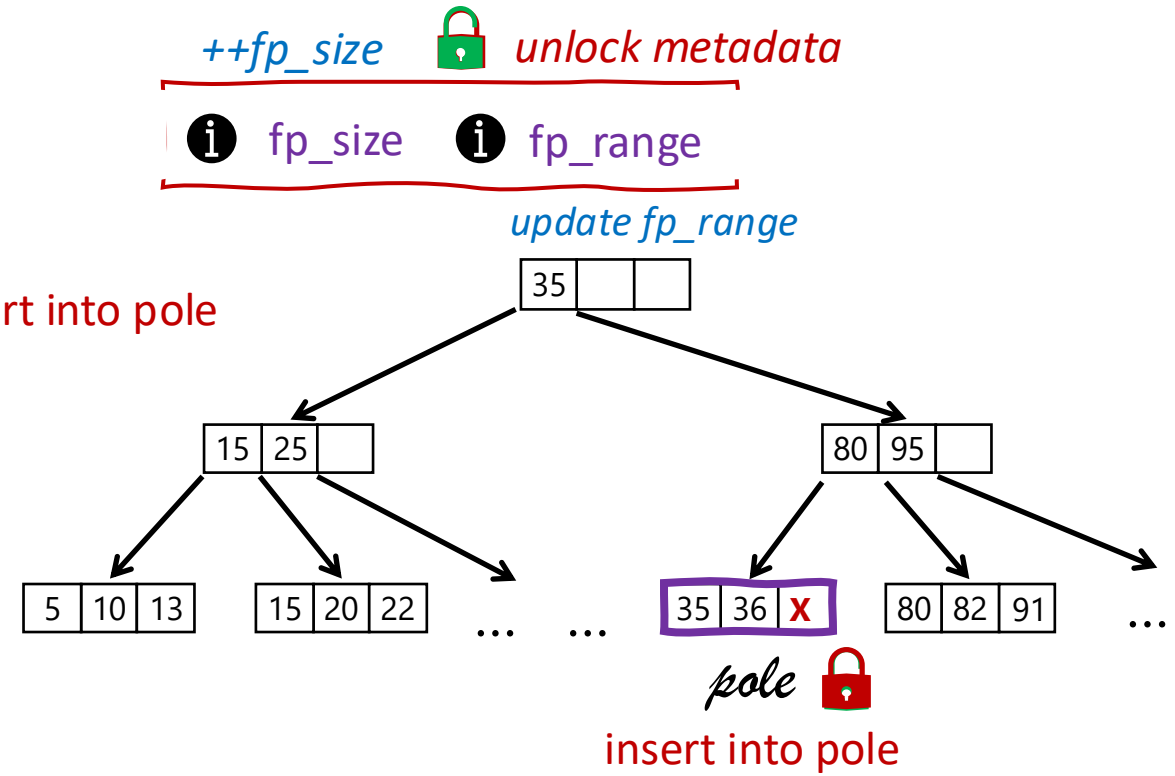
```
metadata.writeLock();

if(key within pole range){
      pole.writeLock();

      if(pole is at CAPACITY){
            top_insert(key, value);
            metadata.unlock();
      }
      else{
            metadata.update(fp_size);
            metadata.unlock();
            pole.insert(key, value);
      }
}
else{
      metadata.unlock();
      top_insert(key, value);
}
```

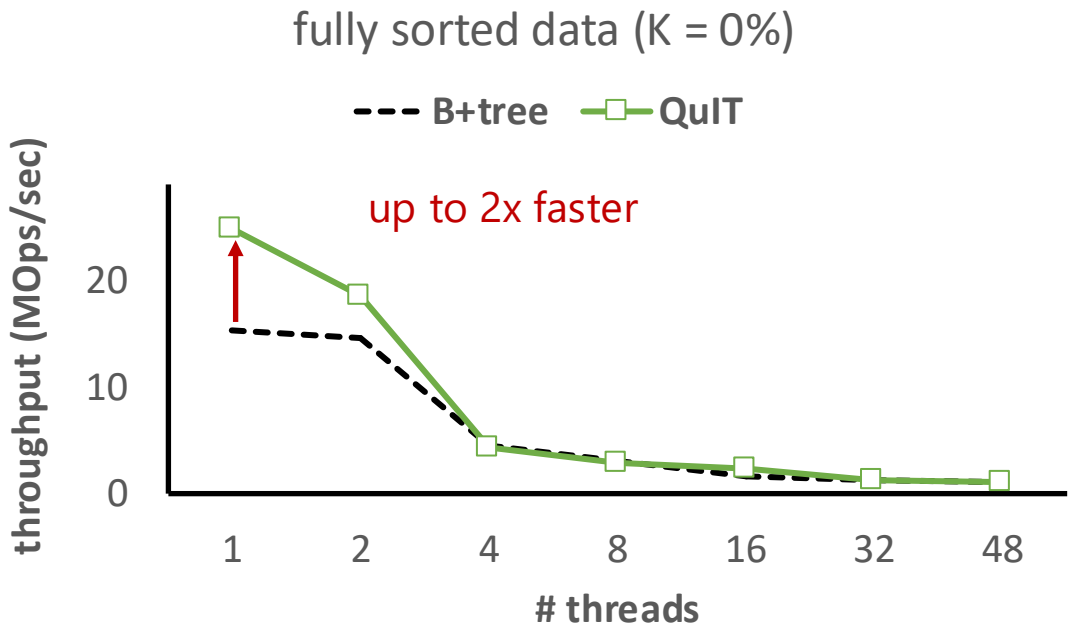
lock meta-data to check for fast-insert

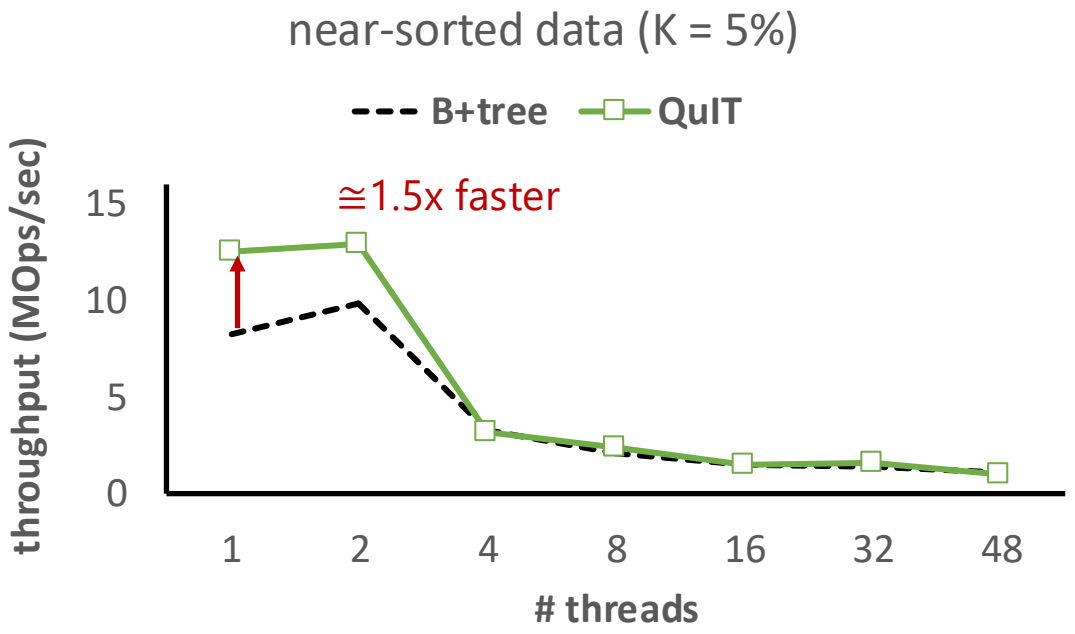
otherwise, insert into pole



QuIT under High Contention

Setup:
500M k-v pairs (8B+8B)
entire index in-memory



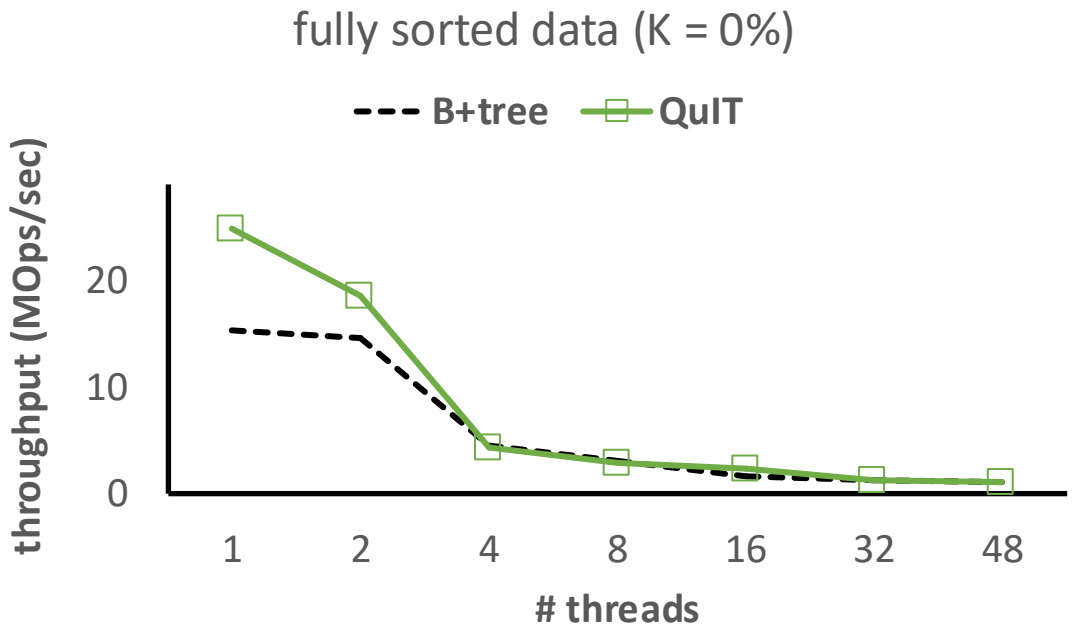


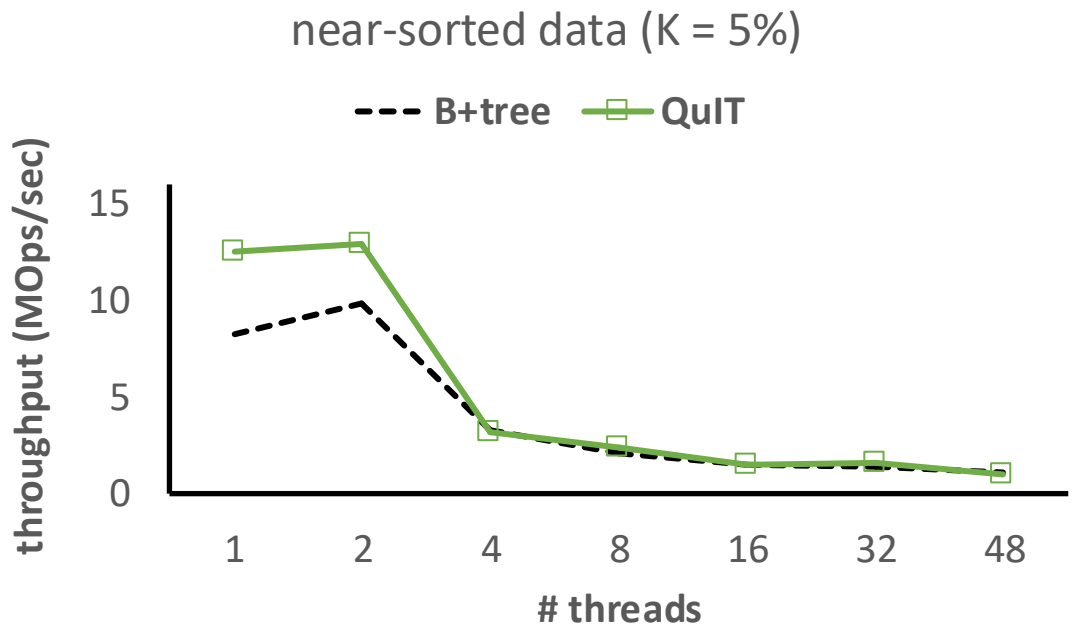
expected as ↑ sortedness ⇒ ↑ contention

BOSTON UNIVERSITY

QuIT under High Contention

Setup:
500M k-v pairs (8B+8B)
entire index in-memory





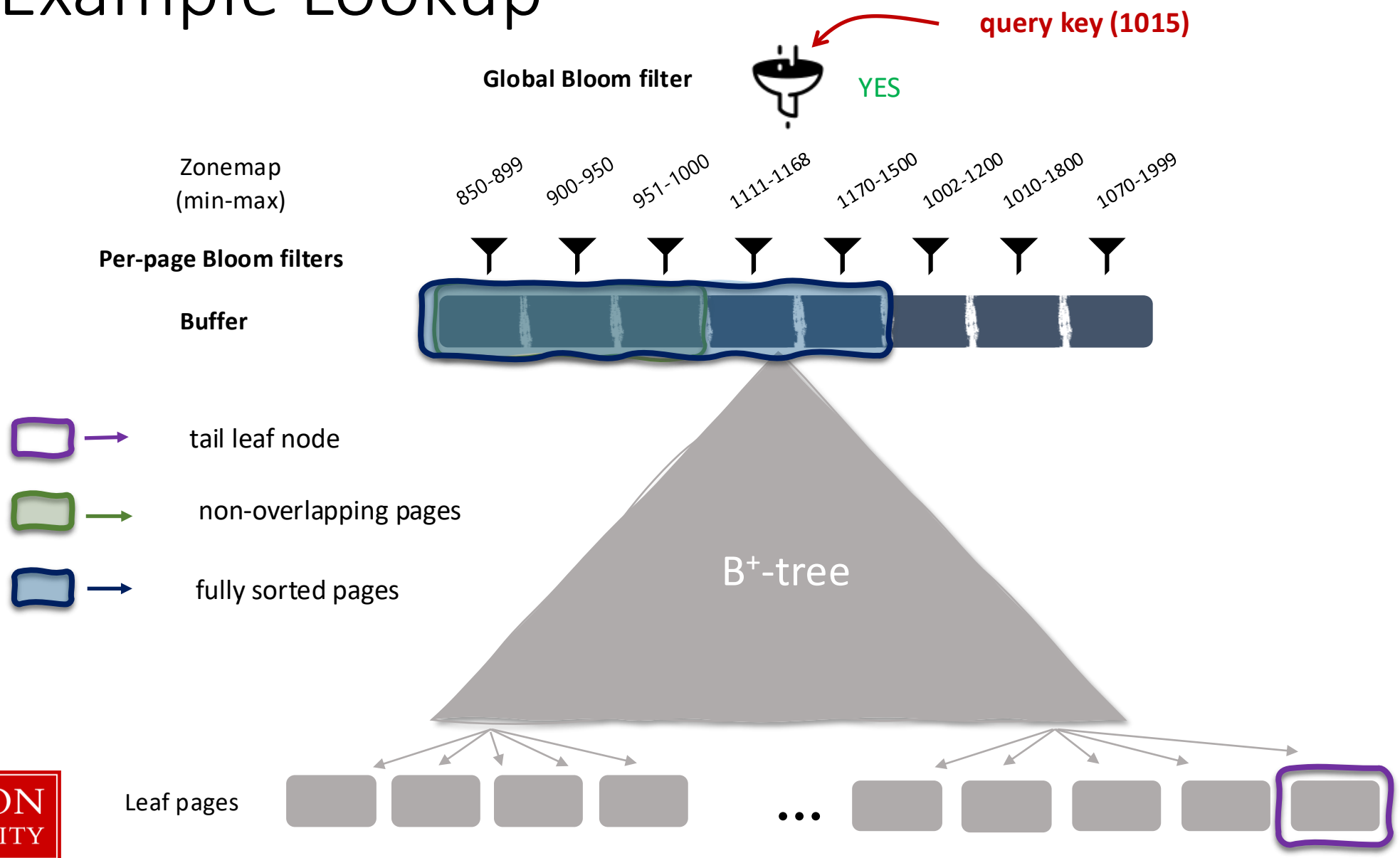
expected as ↑ sortedness ⇒ ↑ contention

insertions alter tree-structure & x-lock

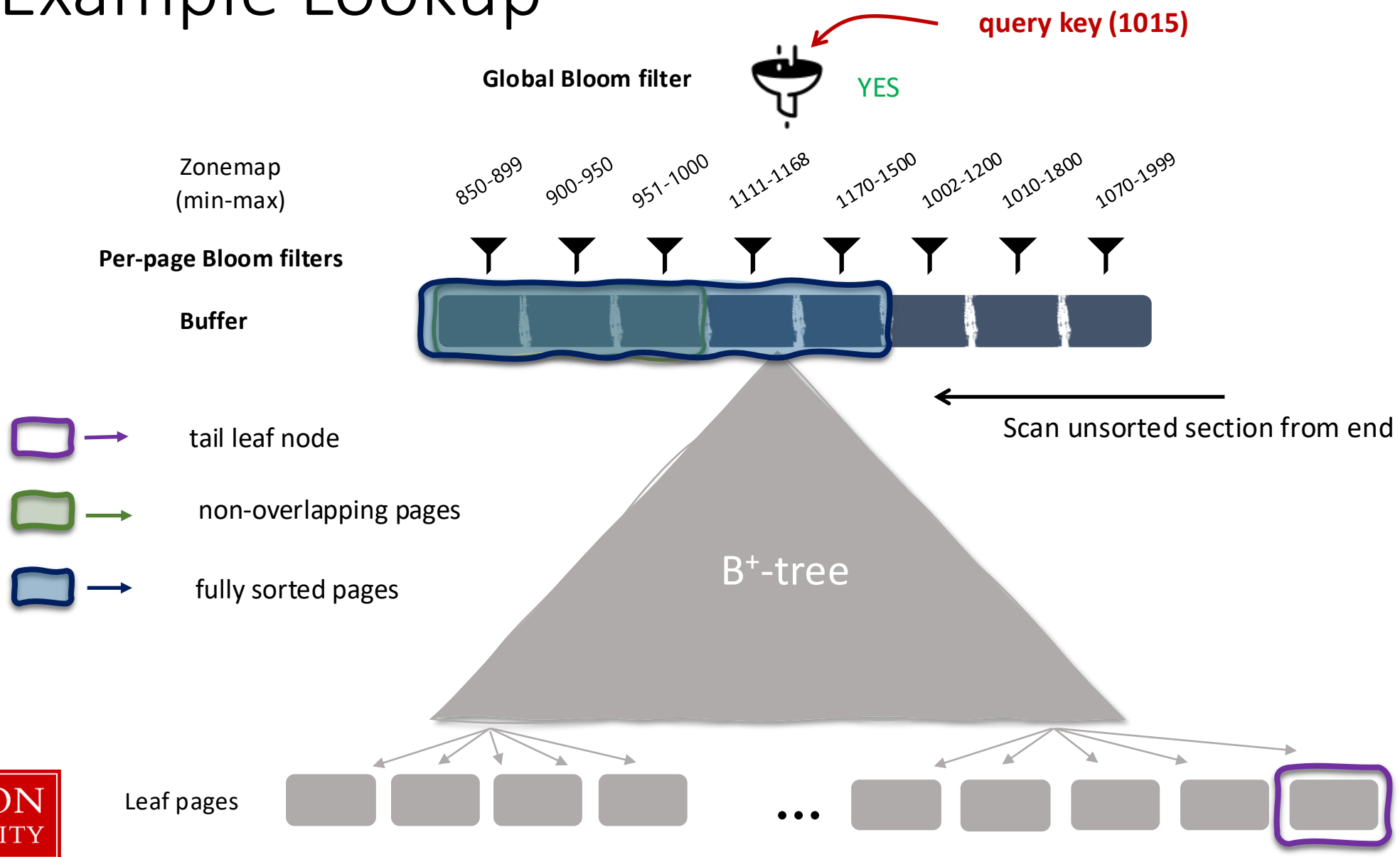
} longer critical sections hurt scalability

BOSTON UNIVERSITY

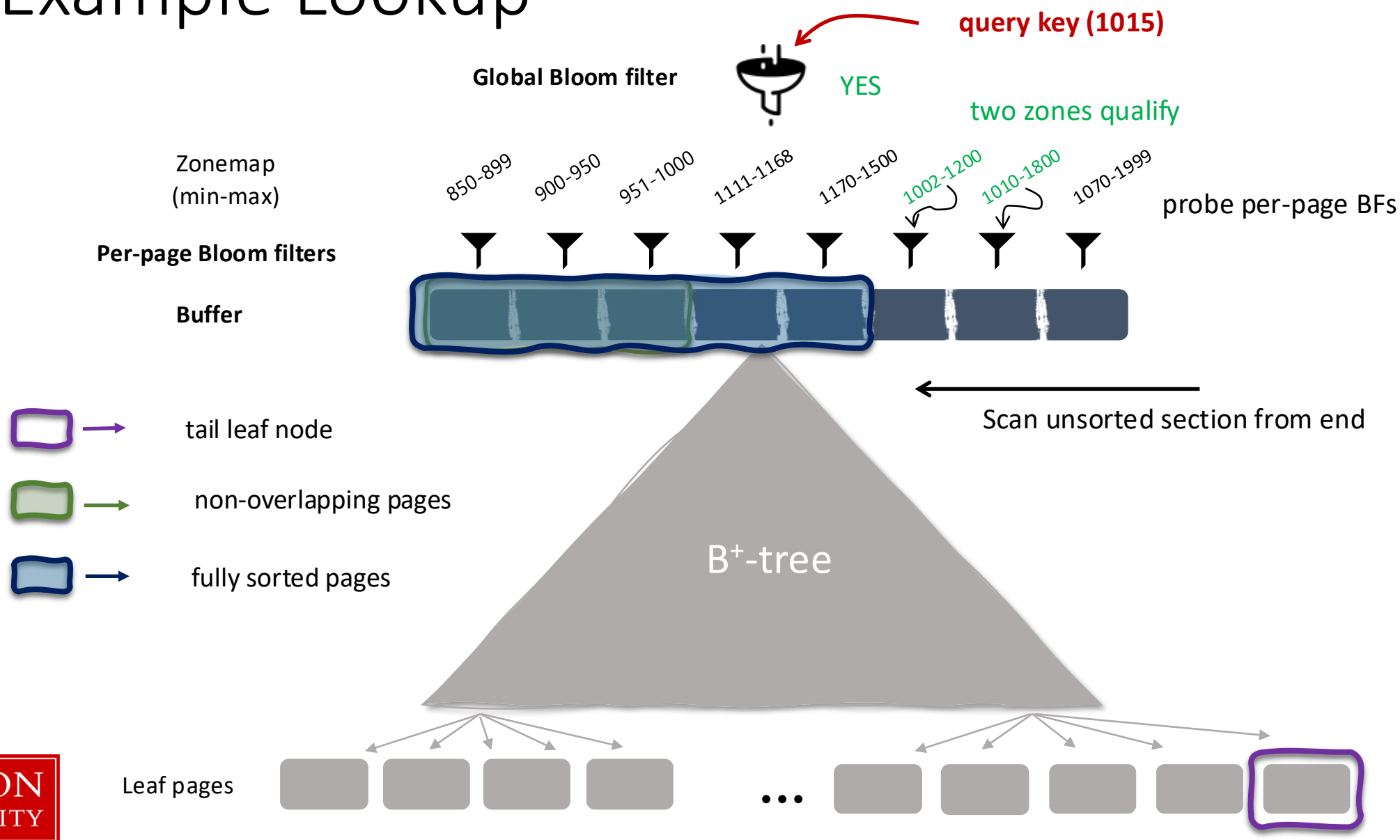
Example Lookup



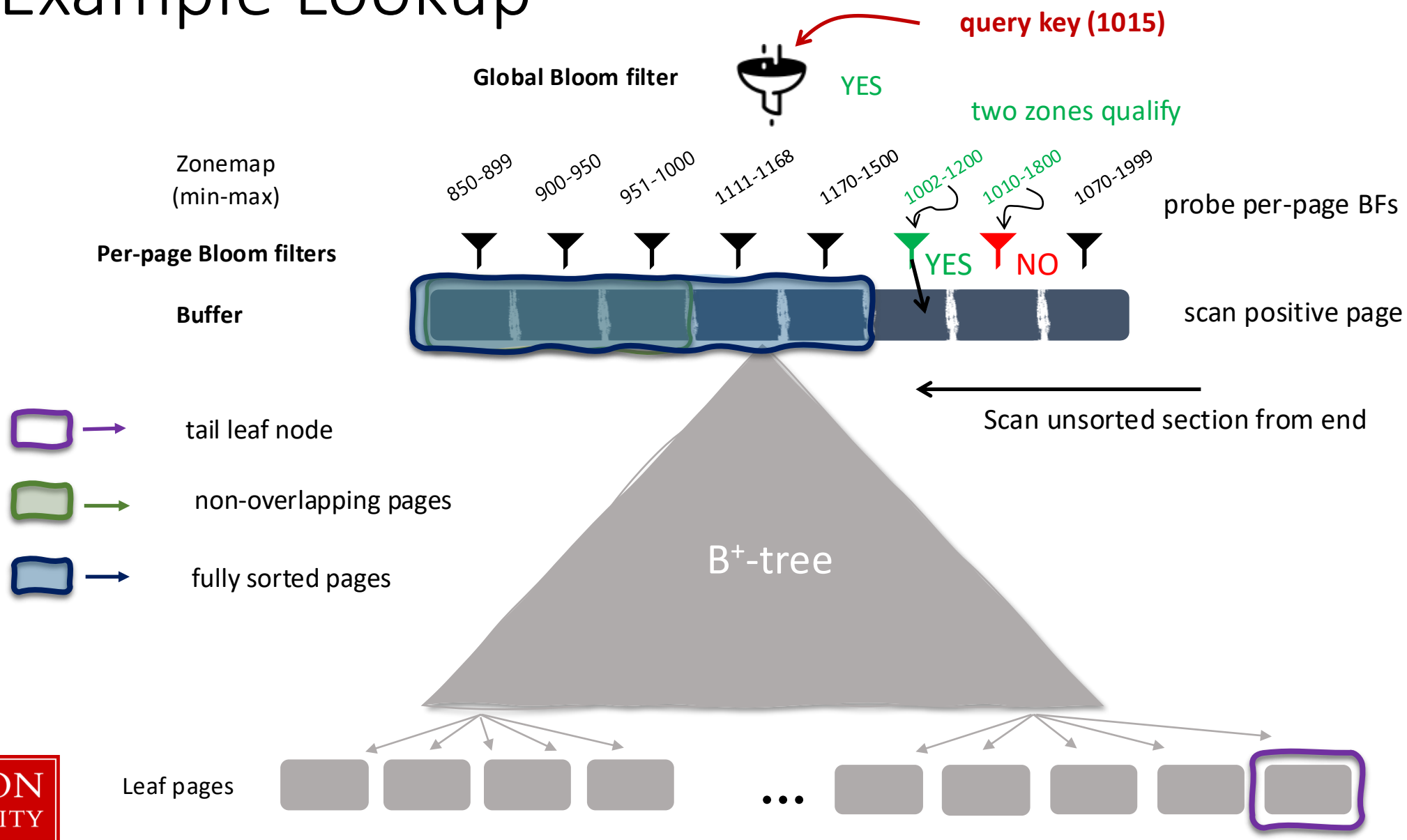
Example Lookup



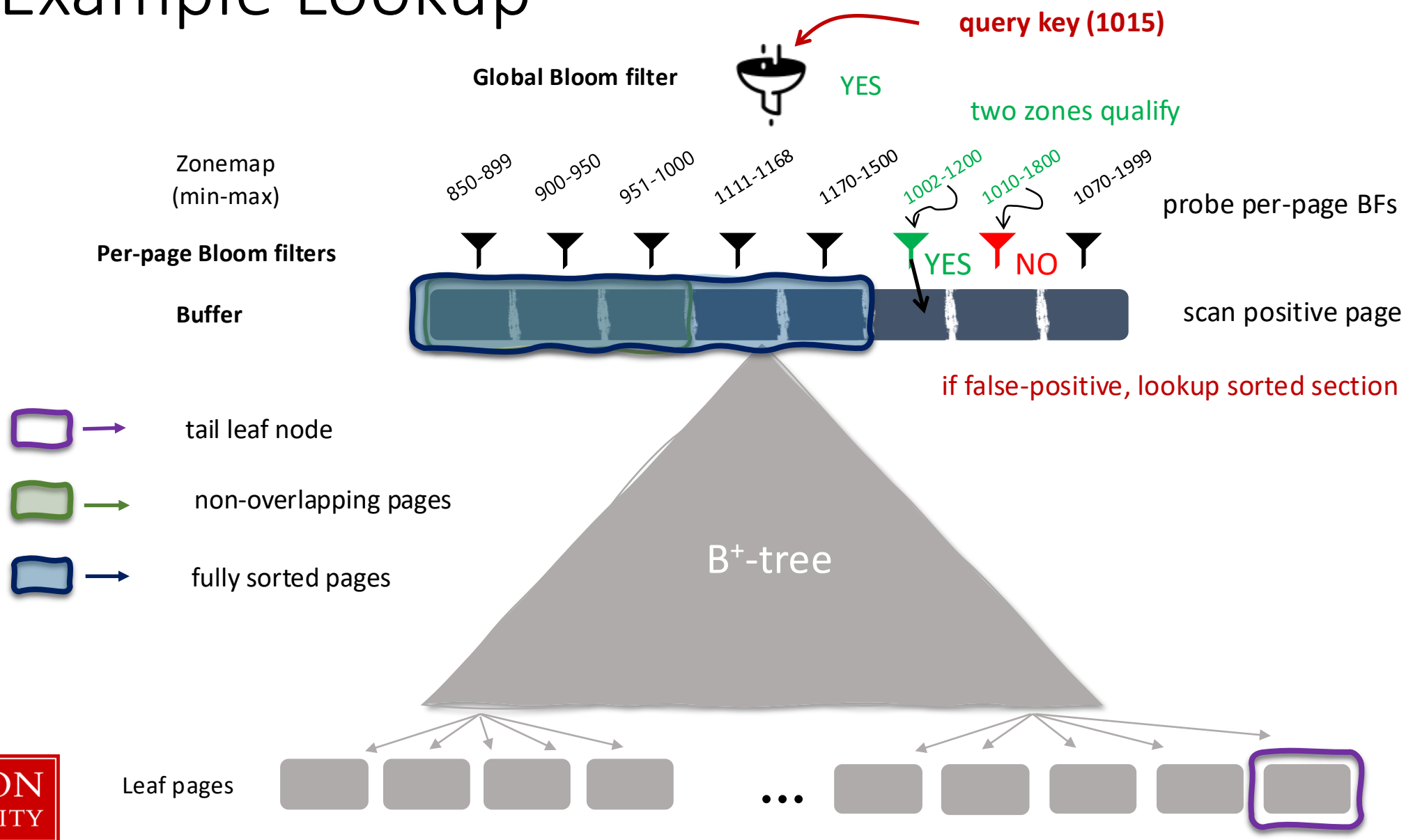
Example Lookup



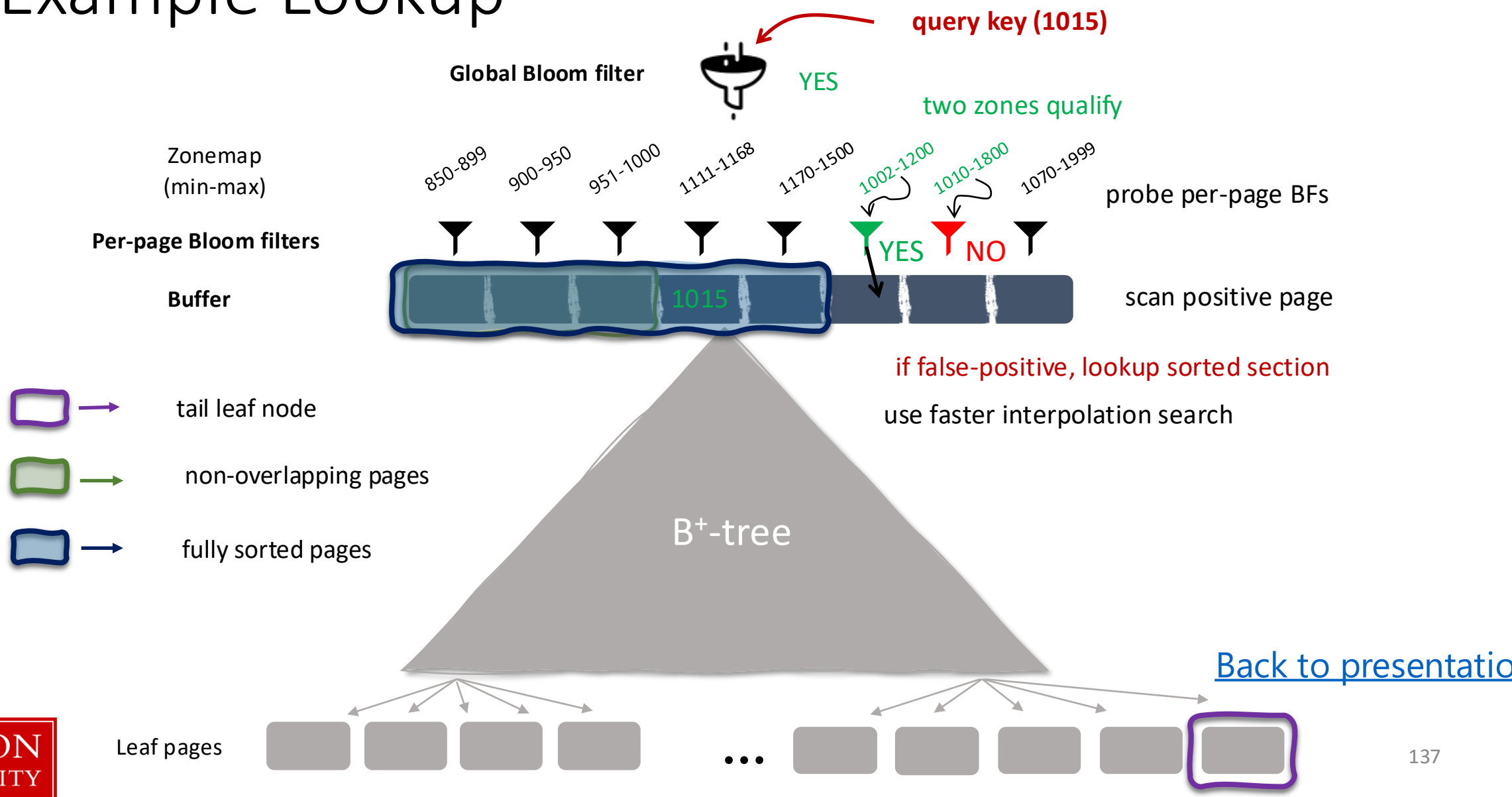
Example Lookup



Example Lookup



Example Lookup



Back to presentation