BOSTON UNIVERSITY

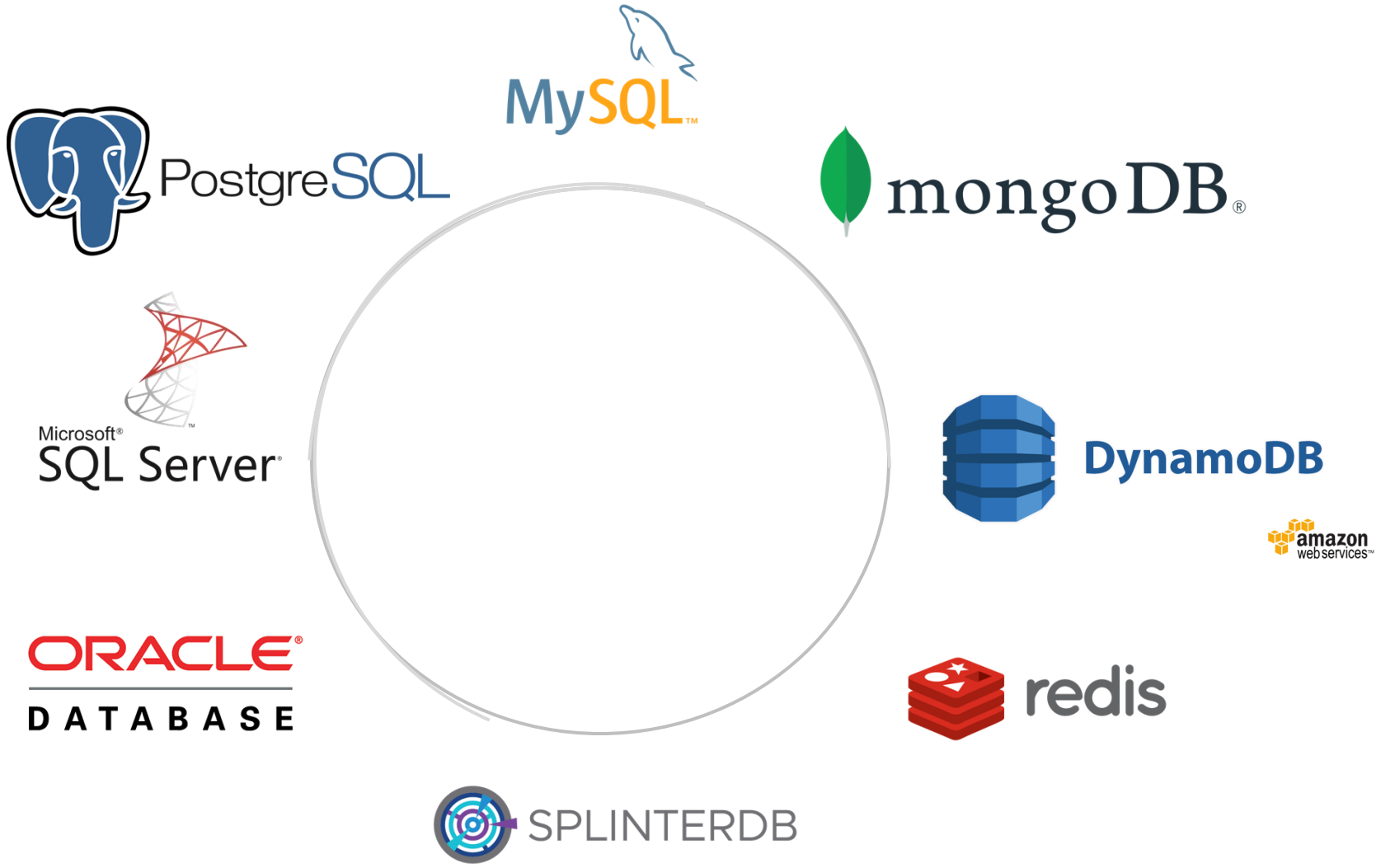
Sortedness-Aware Indexing

CS561: Data Systems Architecture

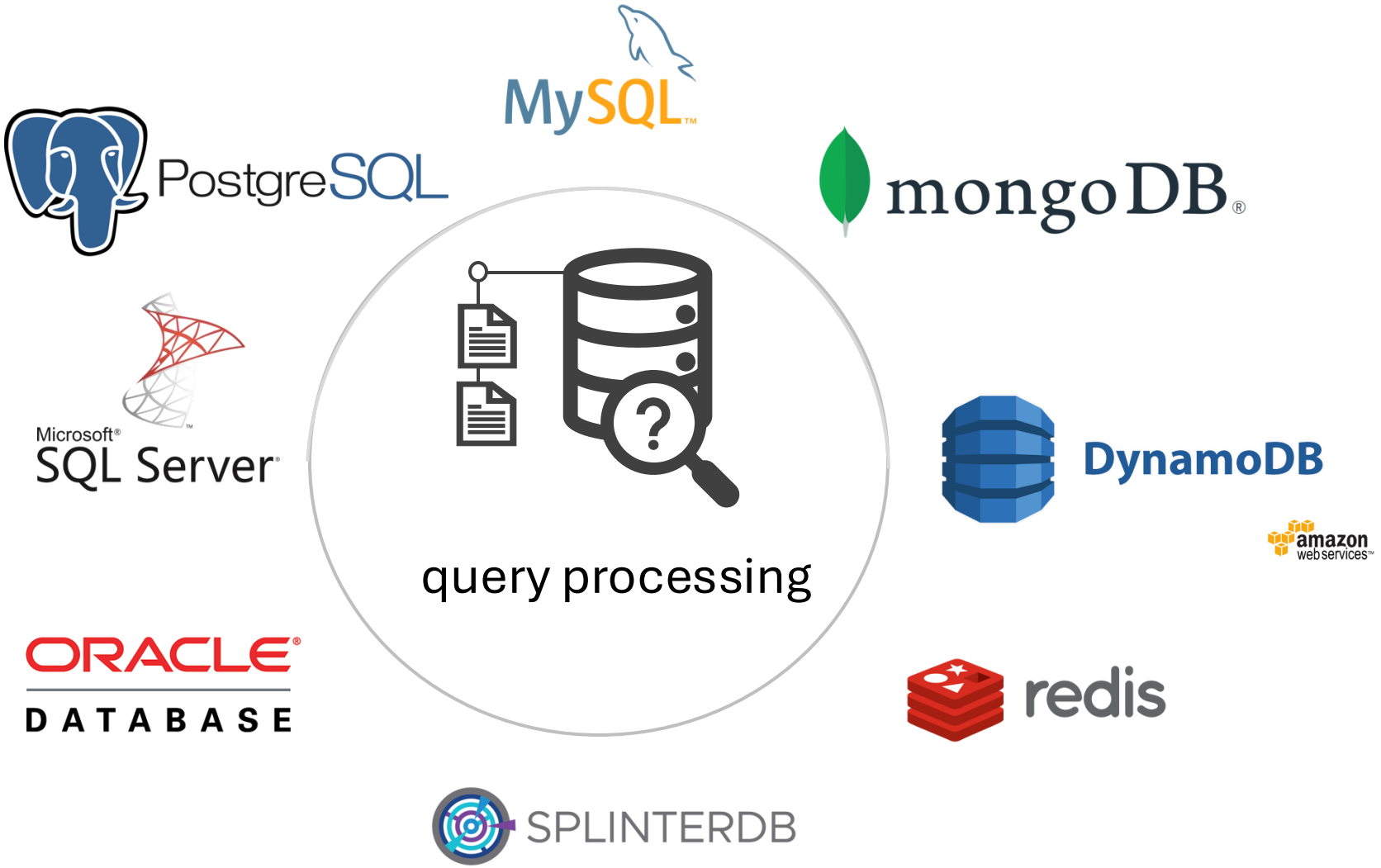
Aneesh Raman

DiSC lab

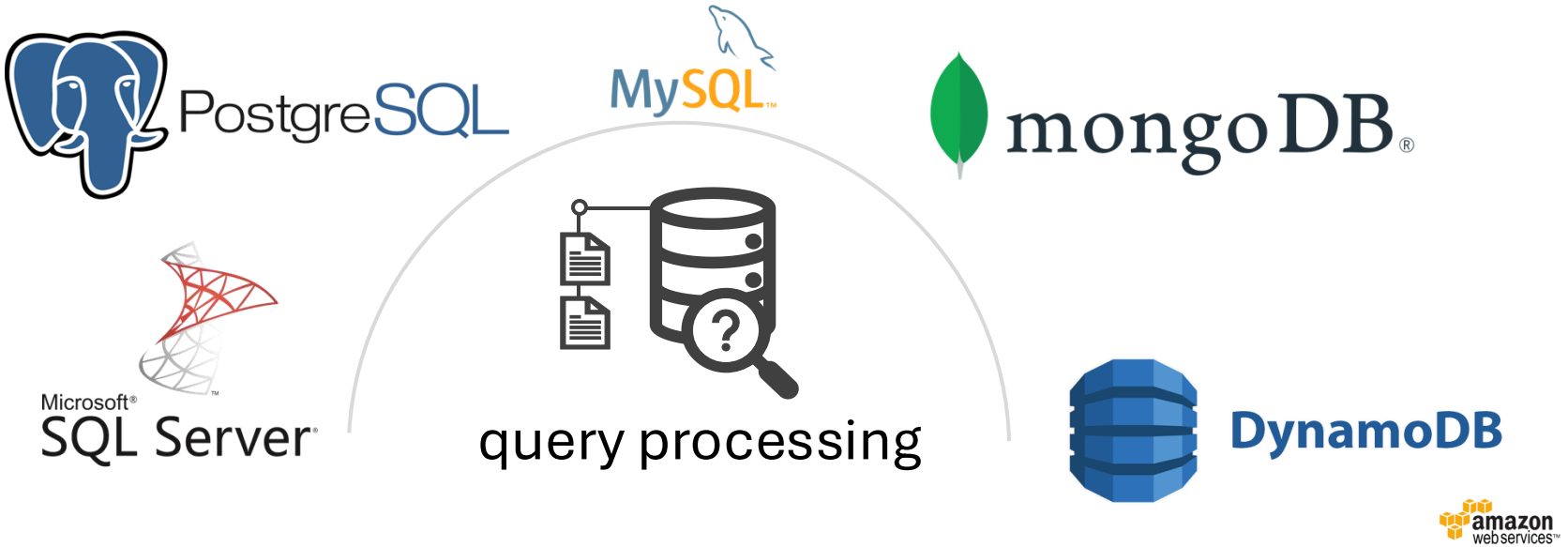
Indexes Are Everywhere!

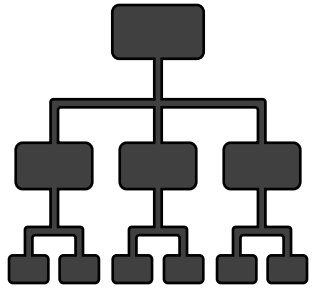


Indexes Are Everywhere!



Indexes Are Everywhere!

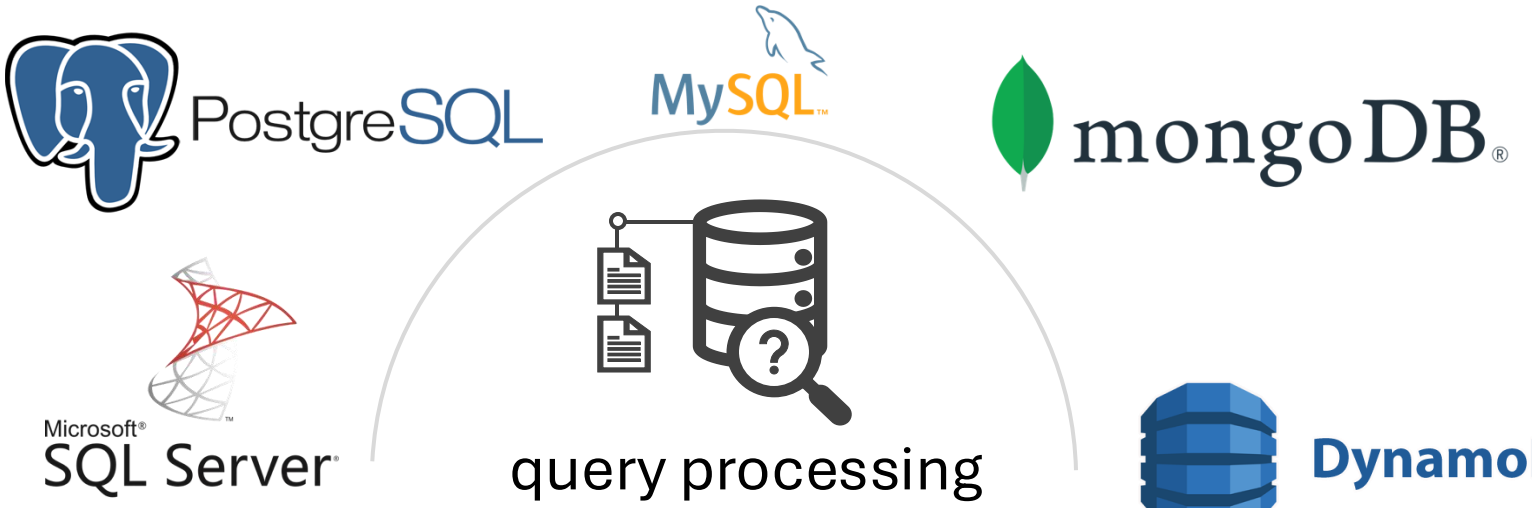


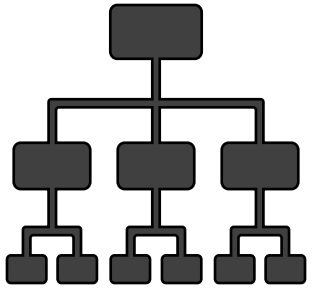


organize data

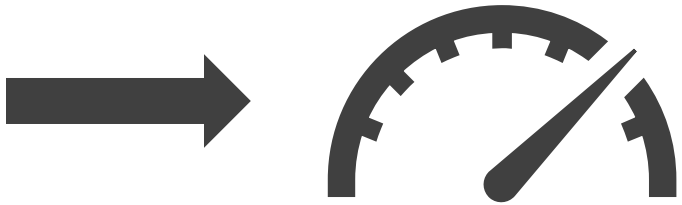
accelerates queries

Indexes Are Everywhere!





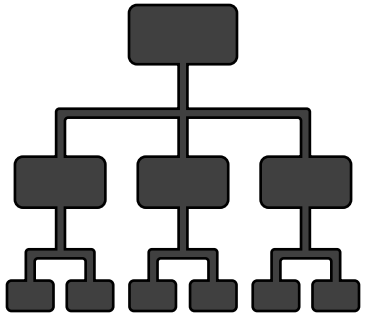
organize data



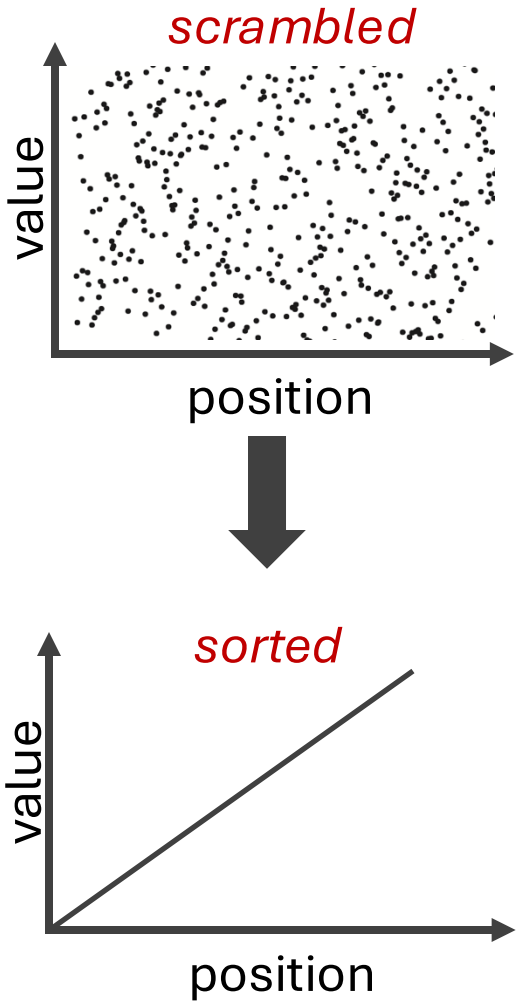
accelerates queries

build + maintain + store

Indexing Adds Structure

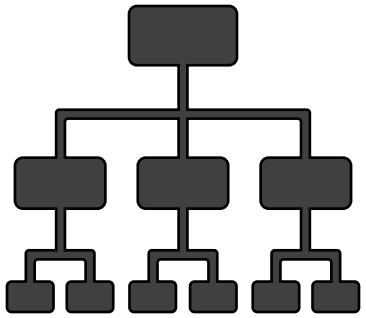


organize data

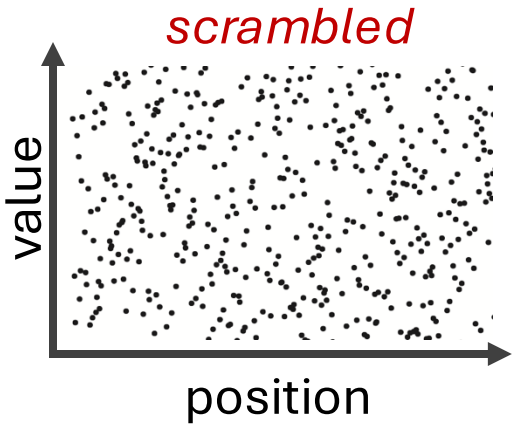


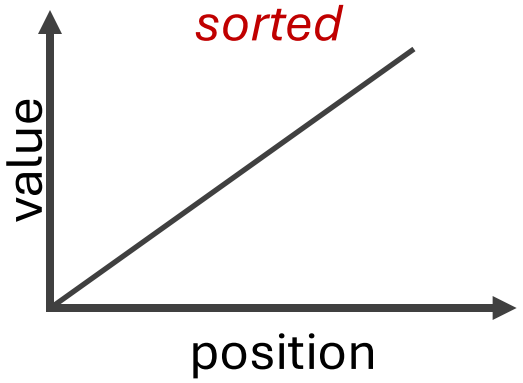
The process of ***"inducing sortedness"*** to otherwise, unsorted data

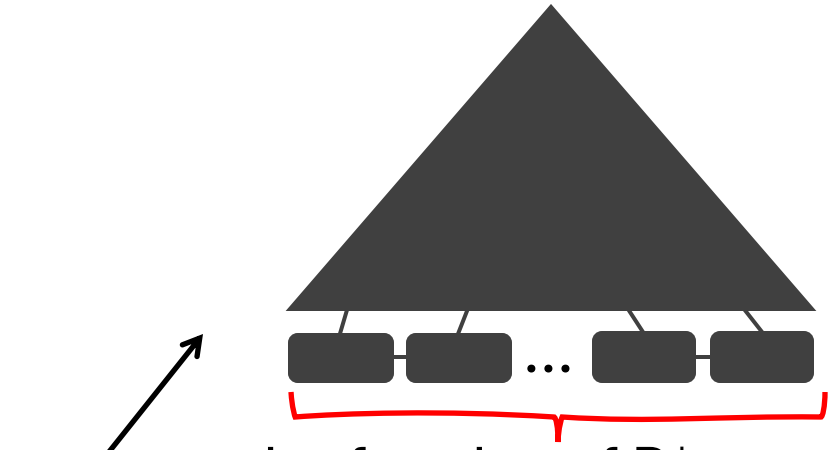
Indexing Adds Structure

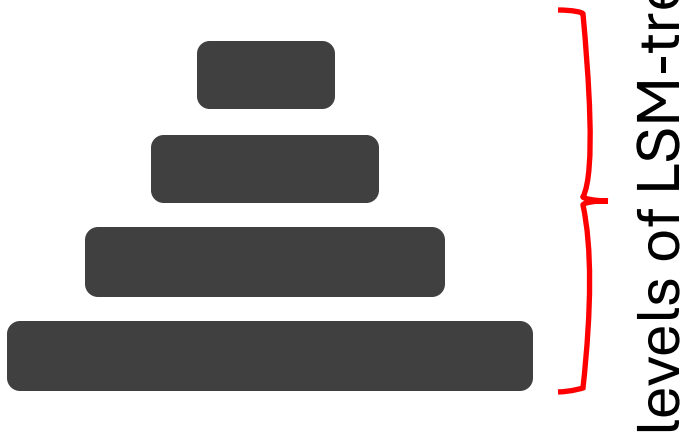


organize data

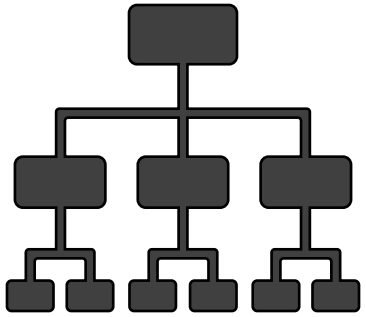




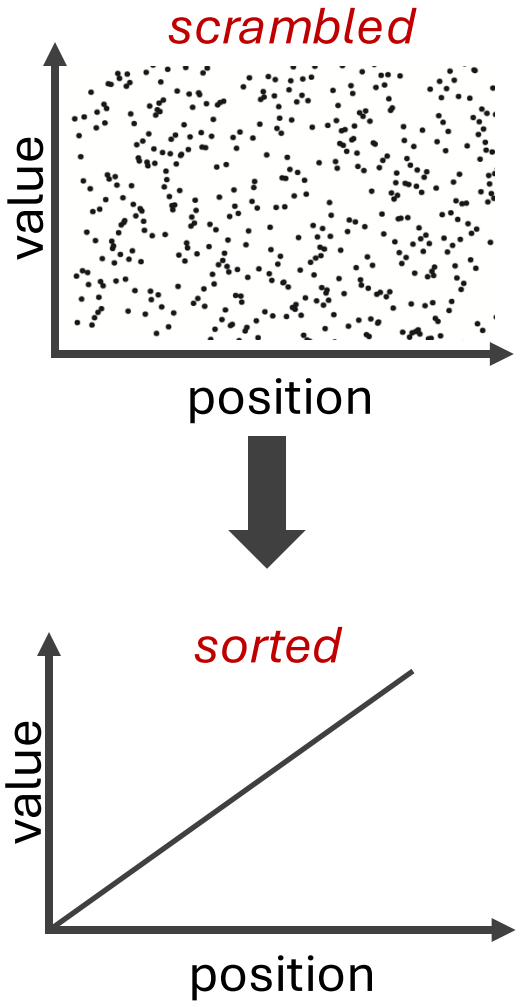




Indexing Adds Structure

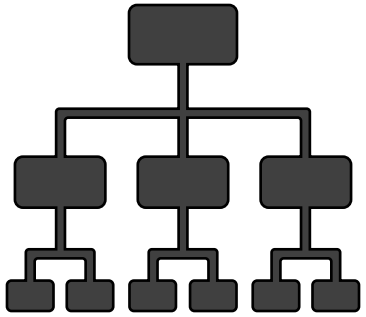


organize data

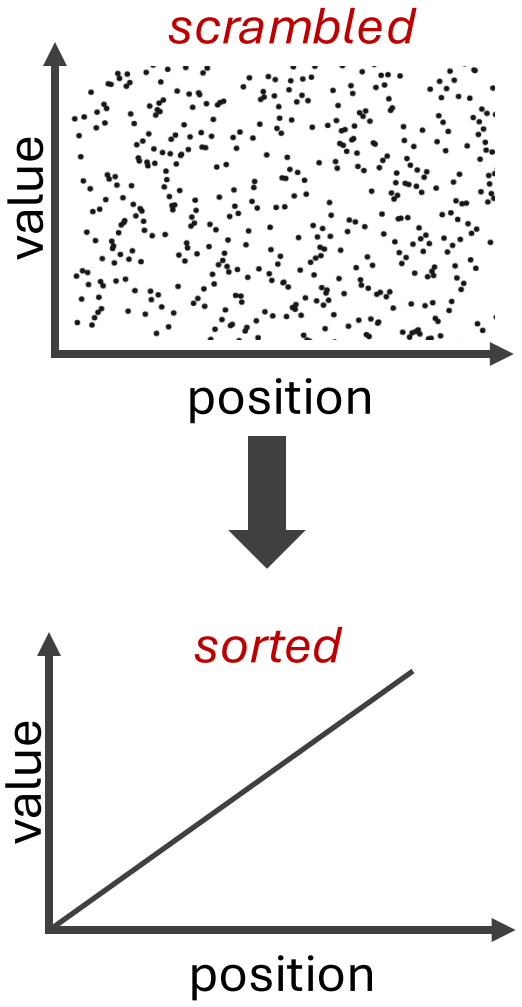


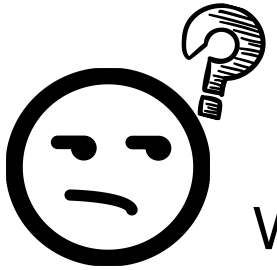
What if the incoming data is **already sorted?**

Indexing Adds Structure

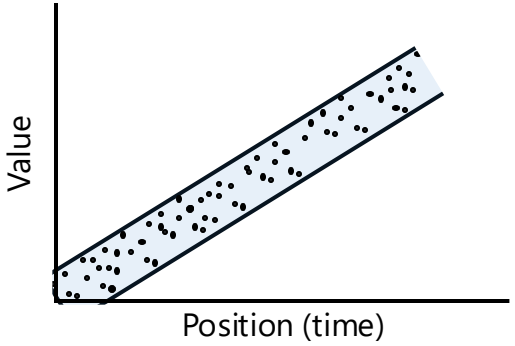


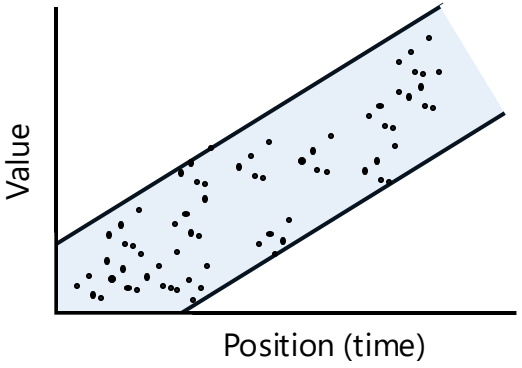
organize data



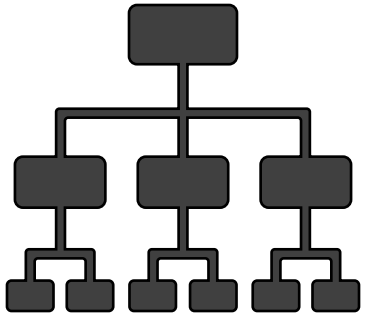


What if the incoming data is **already sorted?**

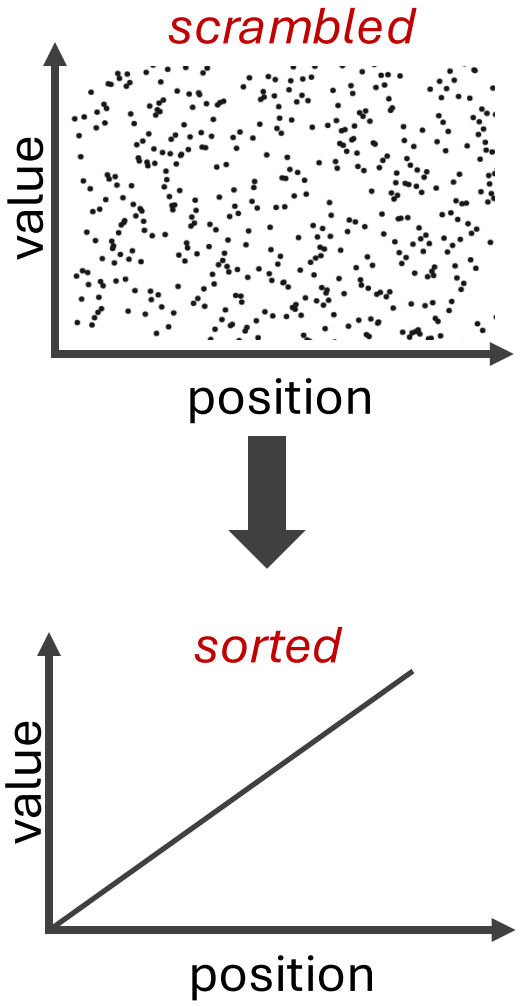


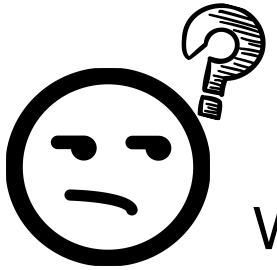


Indexing Adds Structure

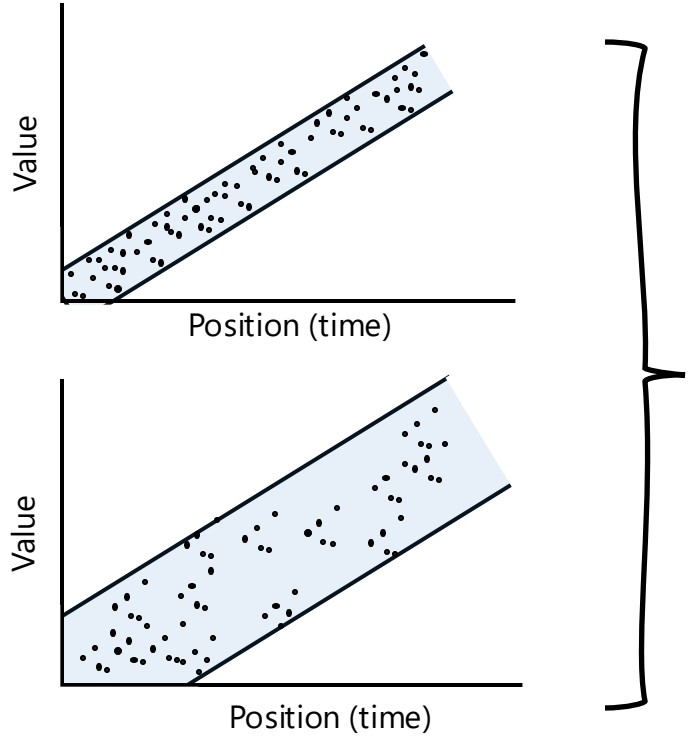


organize data

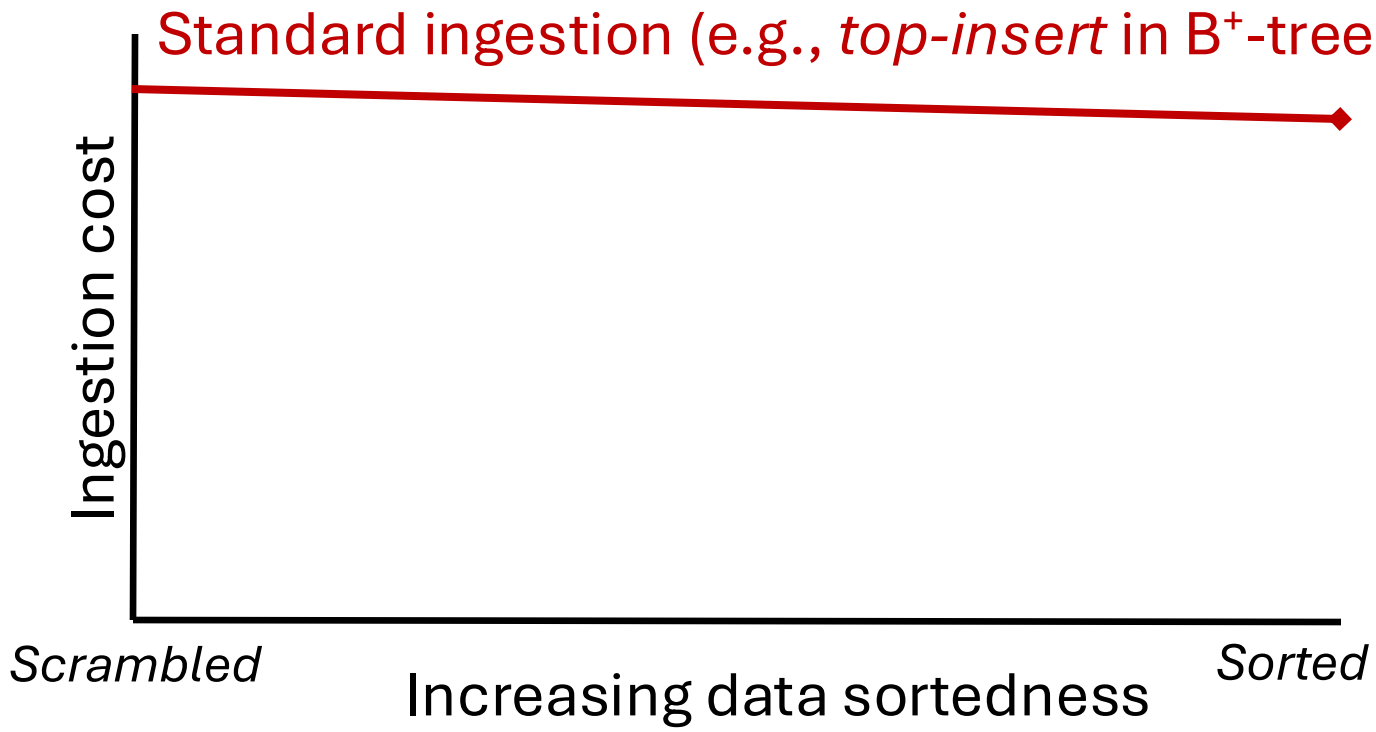




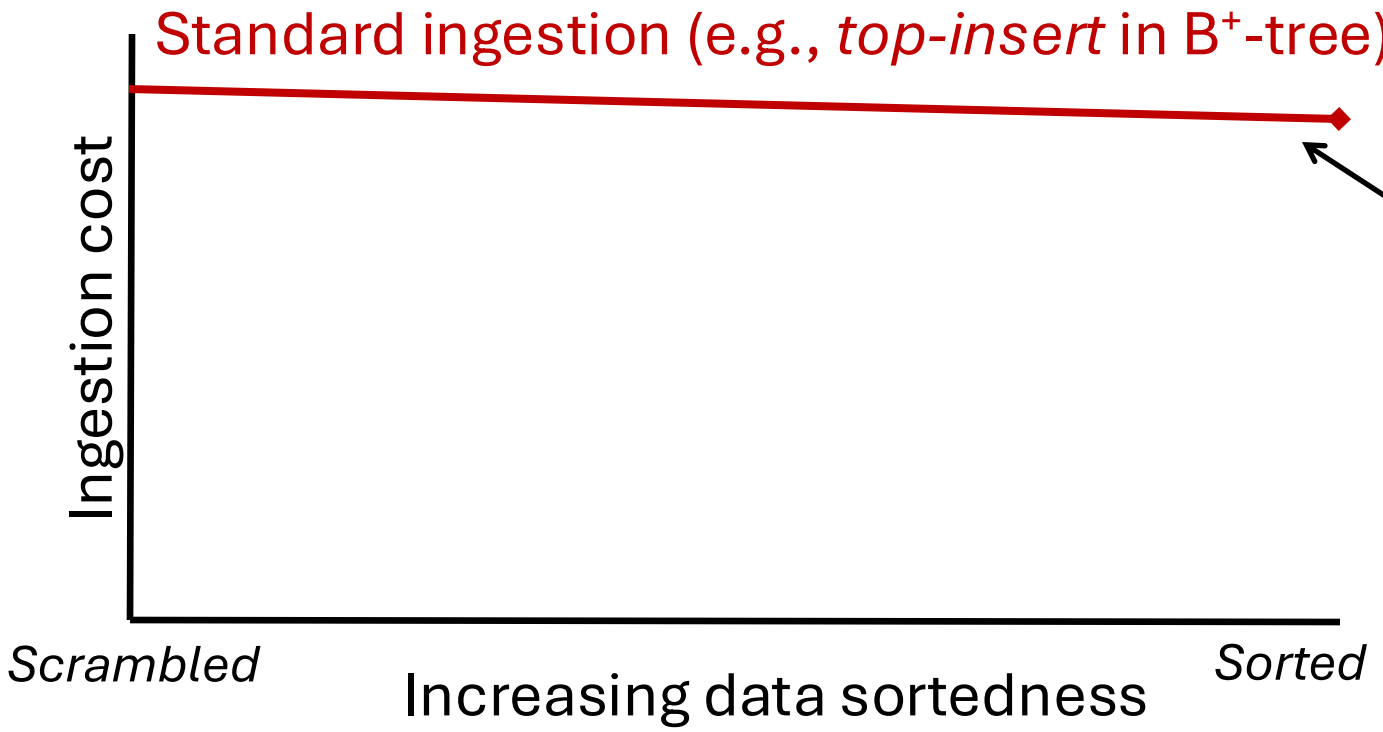
What if the incoming data is **already sorted?**



Irrespective of Sortedness, Same Performance



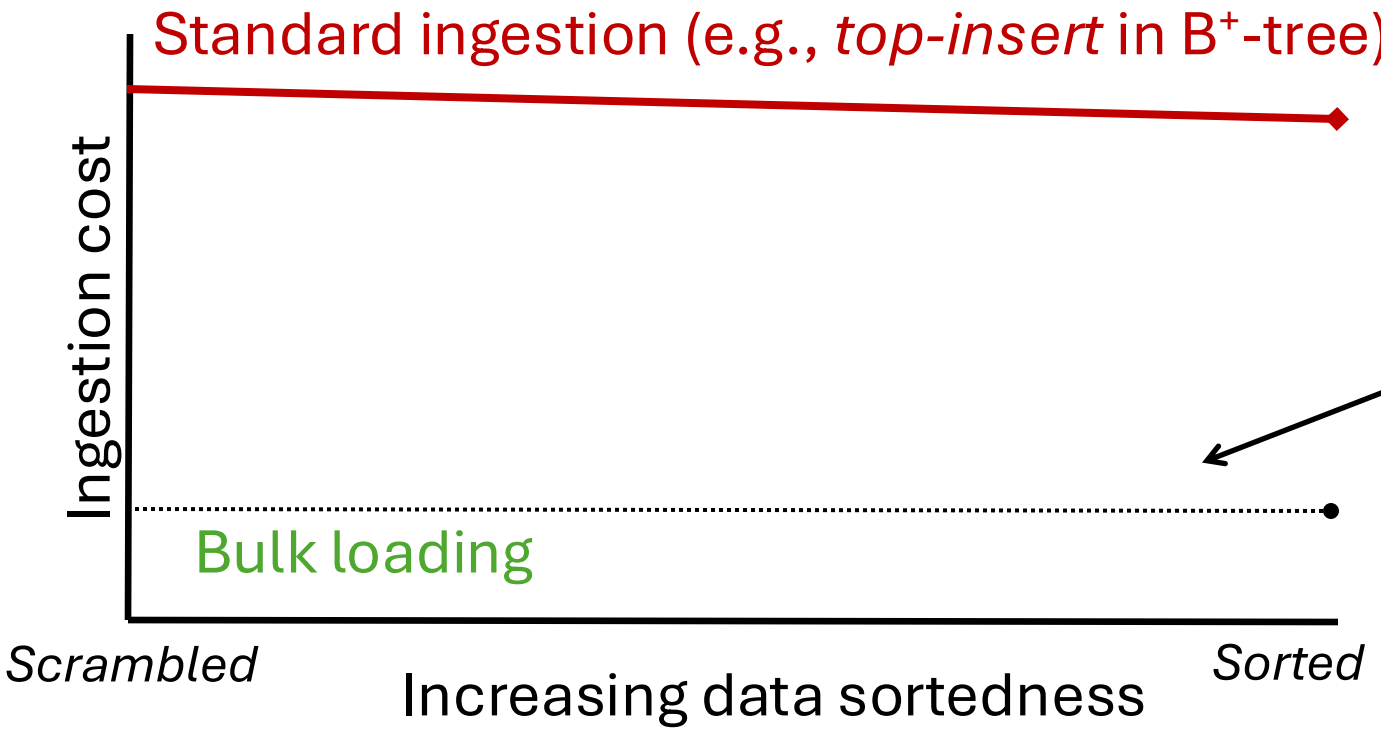
Irrespective of Sortedness, Same Performance



better performance, but due to increased caching

by design, **cannot exploit** inherent order

Are There Faster Alternatives?

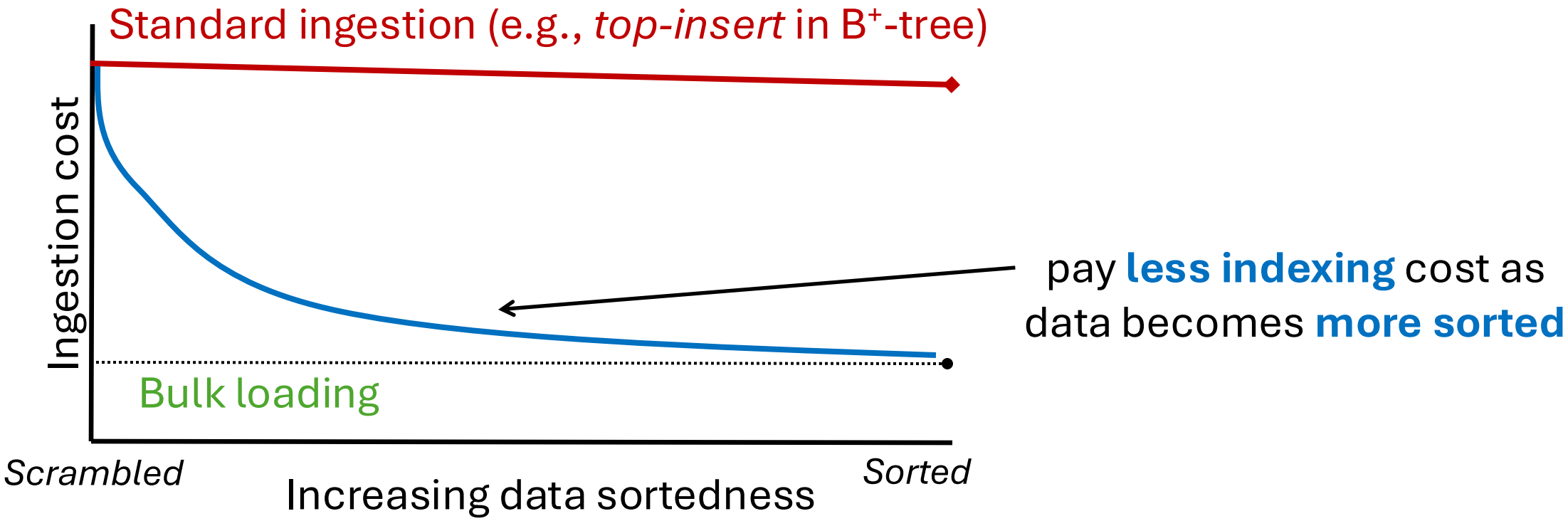


by design, **cannot exploit** inherent order

builds the index bottom-up

requires all data ***a priori***

Ideally, Higher Sortedness => Faster Ingestion



Agenda


Quantifying Data Sortedness
Quick Insertion Tree
Introduction and Vision
Sortedness-Aware (SWARE) Paradigm

Agenda

Introduction and Vision

Quantifying Data

Sortedness

Sortedness-Aware (SWARE) Paradigm

Quick Insertion Tree

Agenda


Introduction and Vision
Quantifying Data Sortedness
Quick Insertion Tree
Sortedness-Aware (SWARE) Paradigm

Agenda


Introduction and Vision
Quantifying Data Sortedness
Sortedness-Aware (SWARE) Paradigm
Quick Insertion Tree

How Do We Quantify ***Sortedness***?

inversions	# pairs in incorrect order
runs	# increasing cont. subsequences
exchanges	least # swaps to establish total order

How Do We Quantify ***Sortedness***?

legacy metrics	**inversions**	# pairs in incorrect order
	runs	# increasing cont. subsequences
	exchanges	least # swaps to establish total order

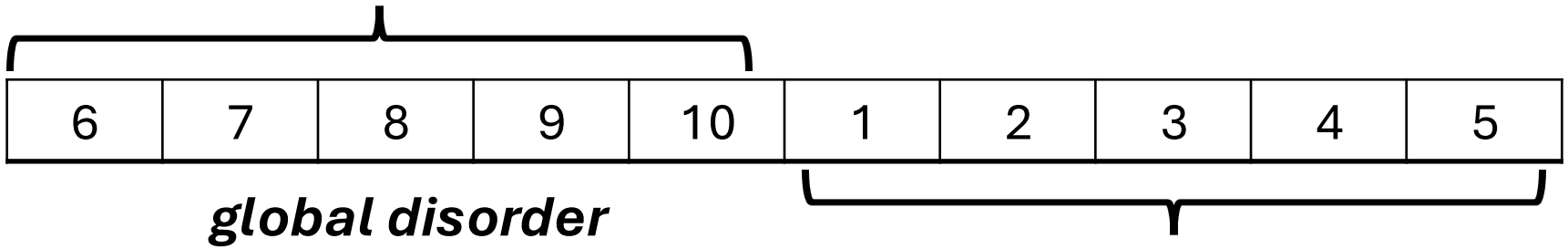
simple, yet have some obvious pitfalls

How Do We Quantify ***Sortedness***?

legacy metrics

inversions ✗	# pairs in incorrect order
runs ✓	# increasing cont. subsequences
exchanges ✗	least # swaps to establish total order

simple, yet have some obvious pitfalls

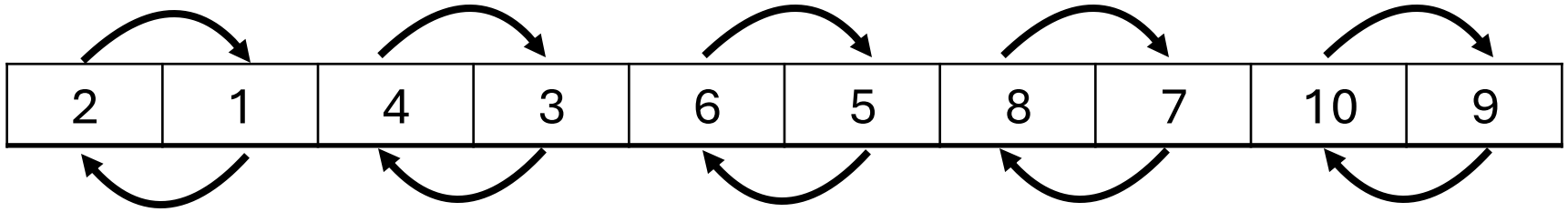


How Do We Quantify ***Sortedness***?

legacy metrics

- **inversions** ✓ — # pairs in incorrect order
- **runs** ✗ — # increasing cont. subsequences
- **exchanges** ✓ — least # swaps to establish total order

simple, yet have some obvious pitfalls



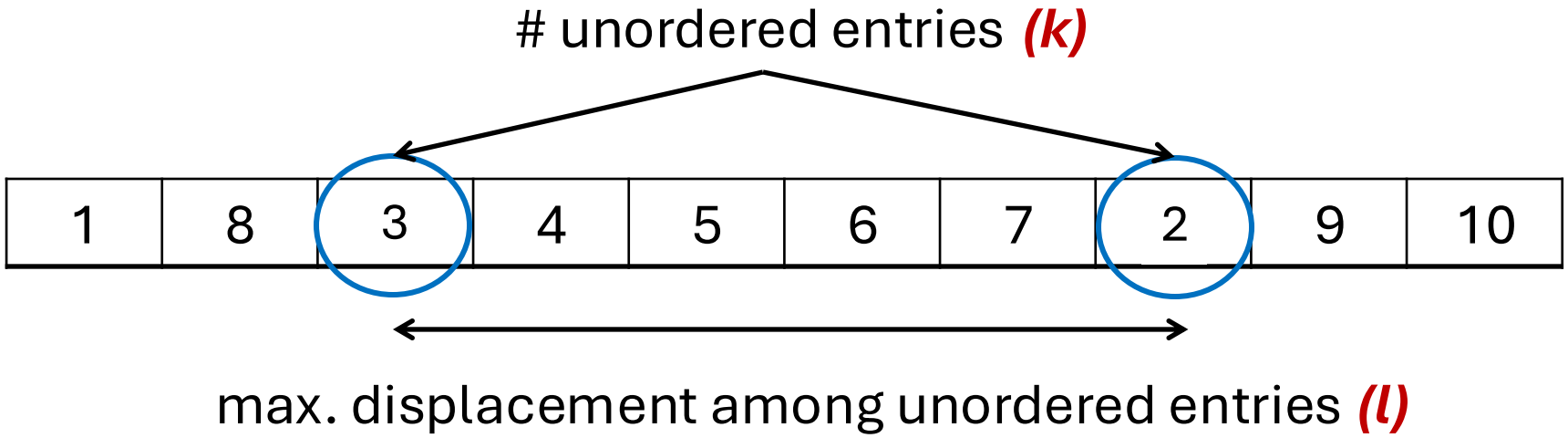
local disorder

A More Intuitive Metric?

(k-l) Sortedness Metric

[Raman2022]

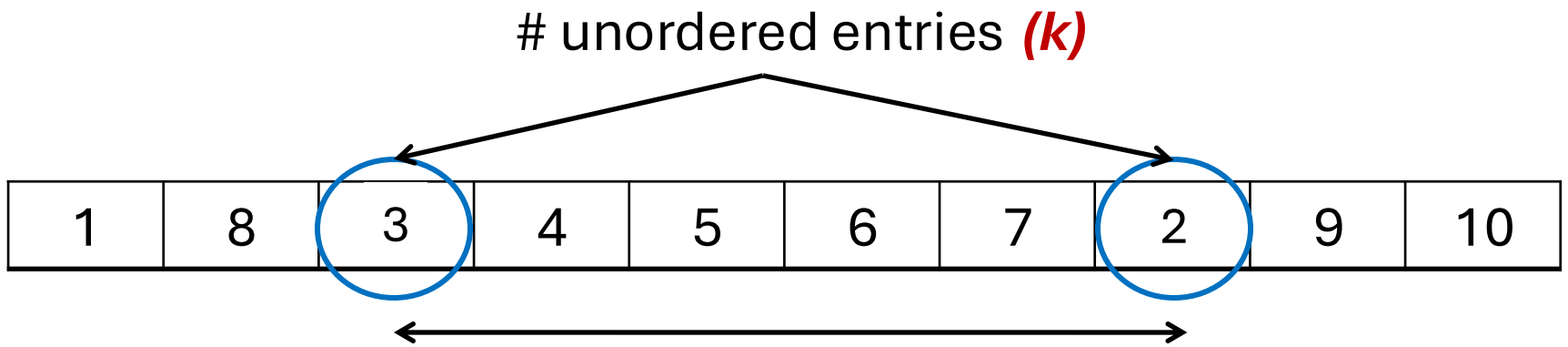
(k-l) Sortedness Metric



[inspired by BenMoshe, ICDT 2011]

[Raman2022]

(k-l) Sortedness Metric



fully sorted: K or L = 0

near-sorted: low K and low L

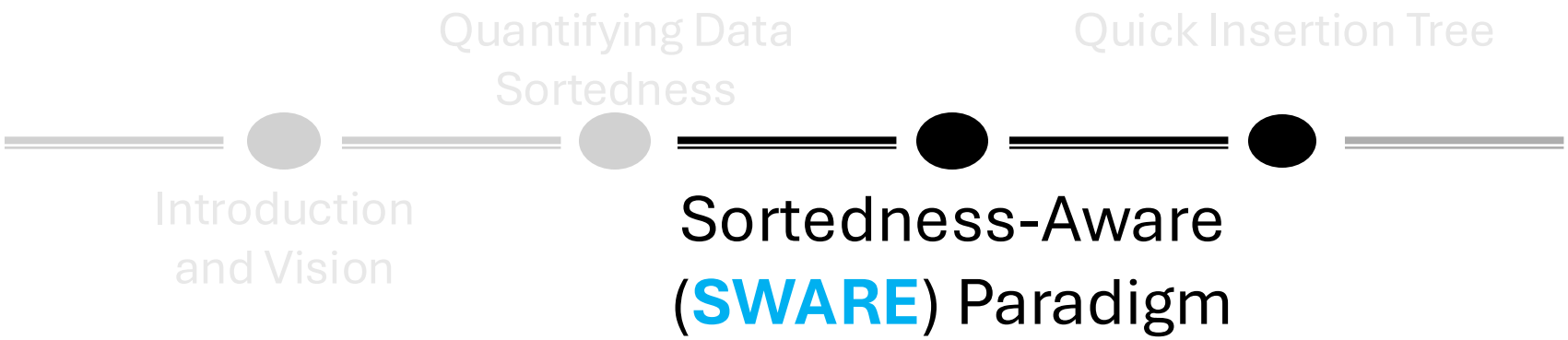
less-sorted: high K and high L

scrambled: K = L = 100%

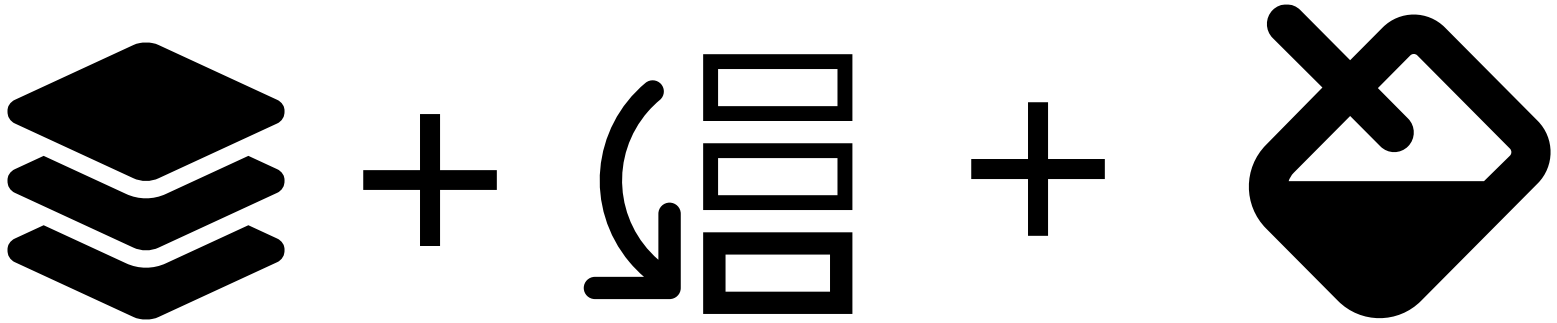
low K and high L; or
high K and low L
} addresses global & local disorder

[inspired by BenMoshe, ICDT 2011]

Coming Up…


Quantifying Data Sortedness
Quick Insertion Tree
Introduction and Vision
Sortedness-Aware (SWARE) Paradigm

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

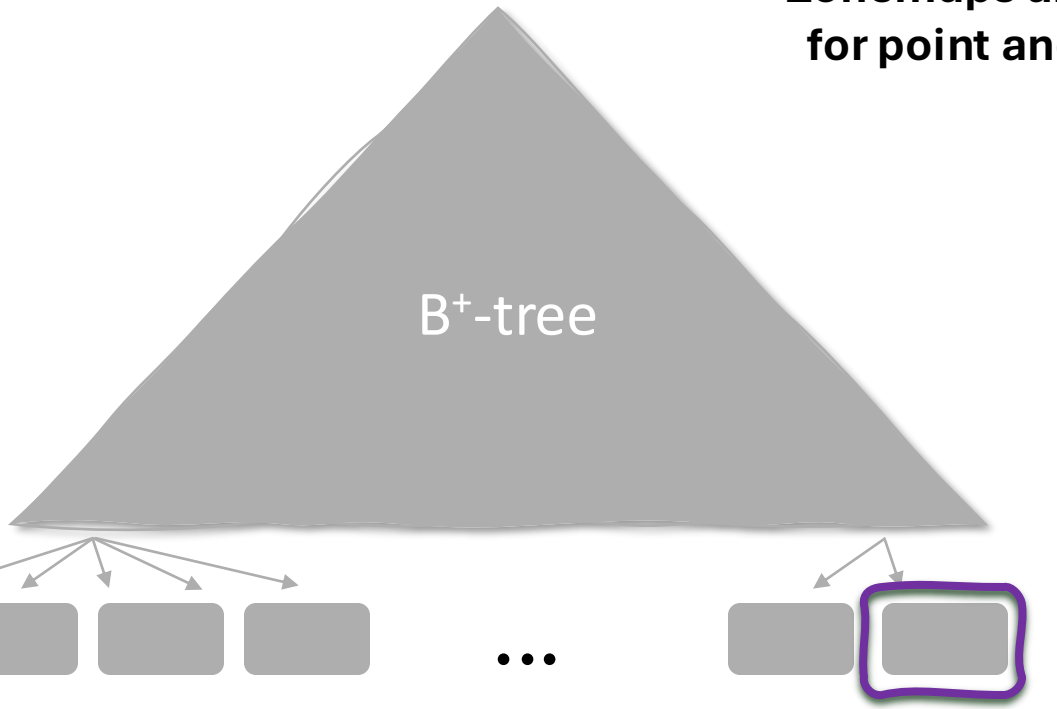
append & sort if needed

SWARE Buffer

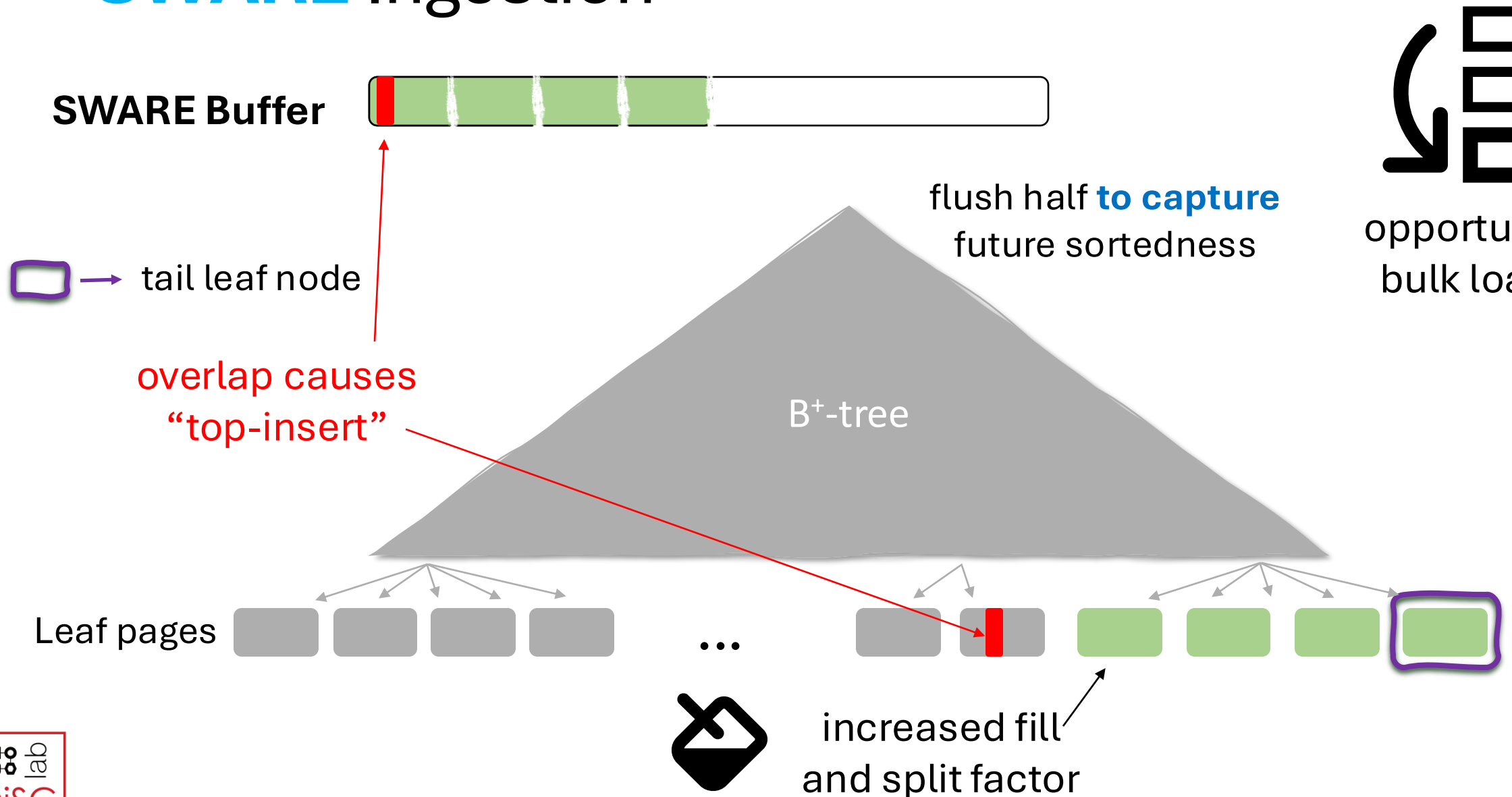
intelligent buffering

Zonemaps and Bloom filters for point and range queries

tail leaf node

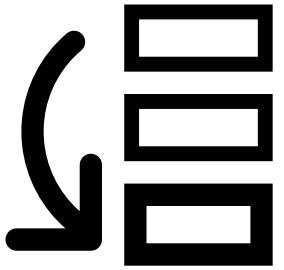


SWARE Ingestion

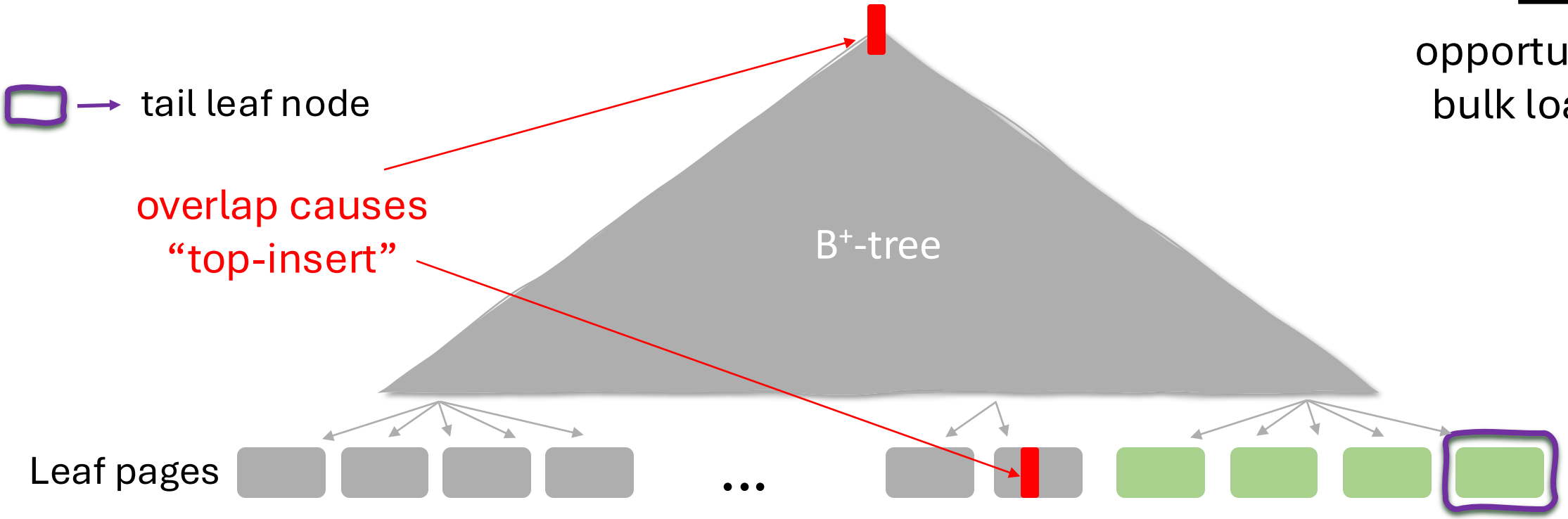


SWARE Ingestion

SWARE Buffer

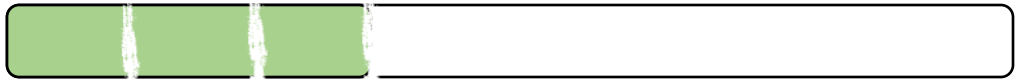


opportunistic bulk loading

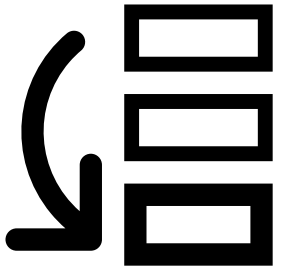


SWARE Ingestion

SWARE Buffer



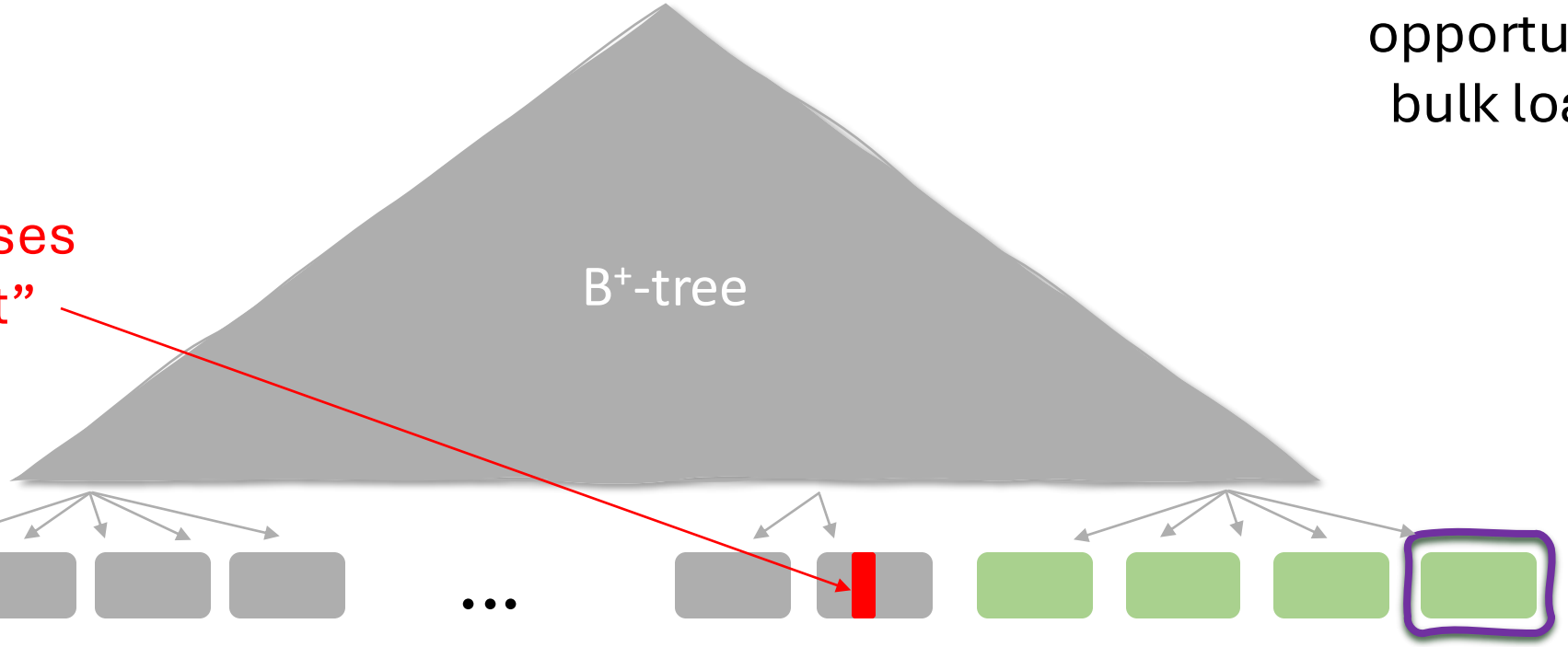
opportunistic bulk loading resumes!



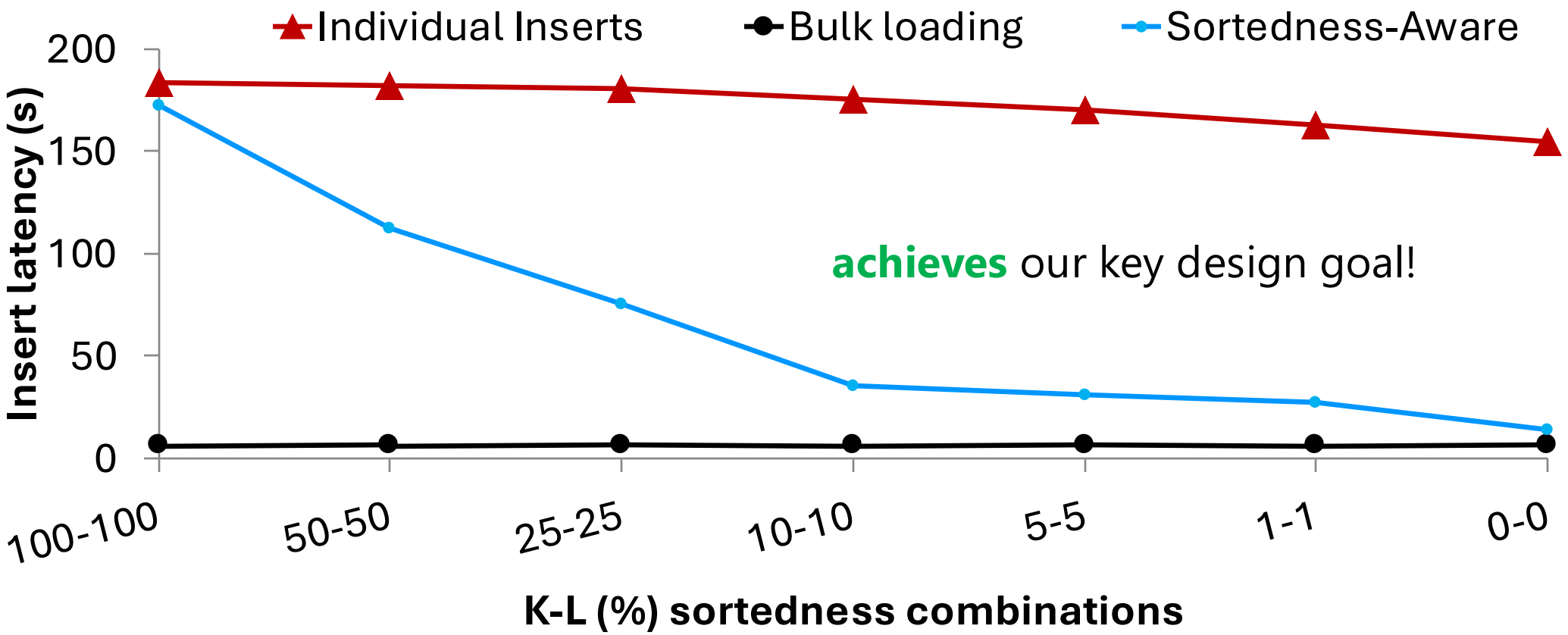
opportunistic bulk loading

tail leaf node

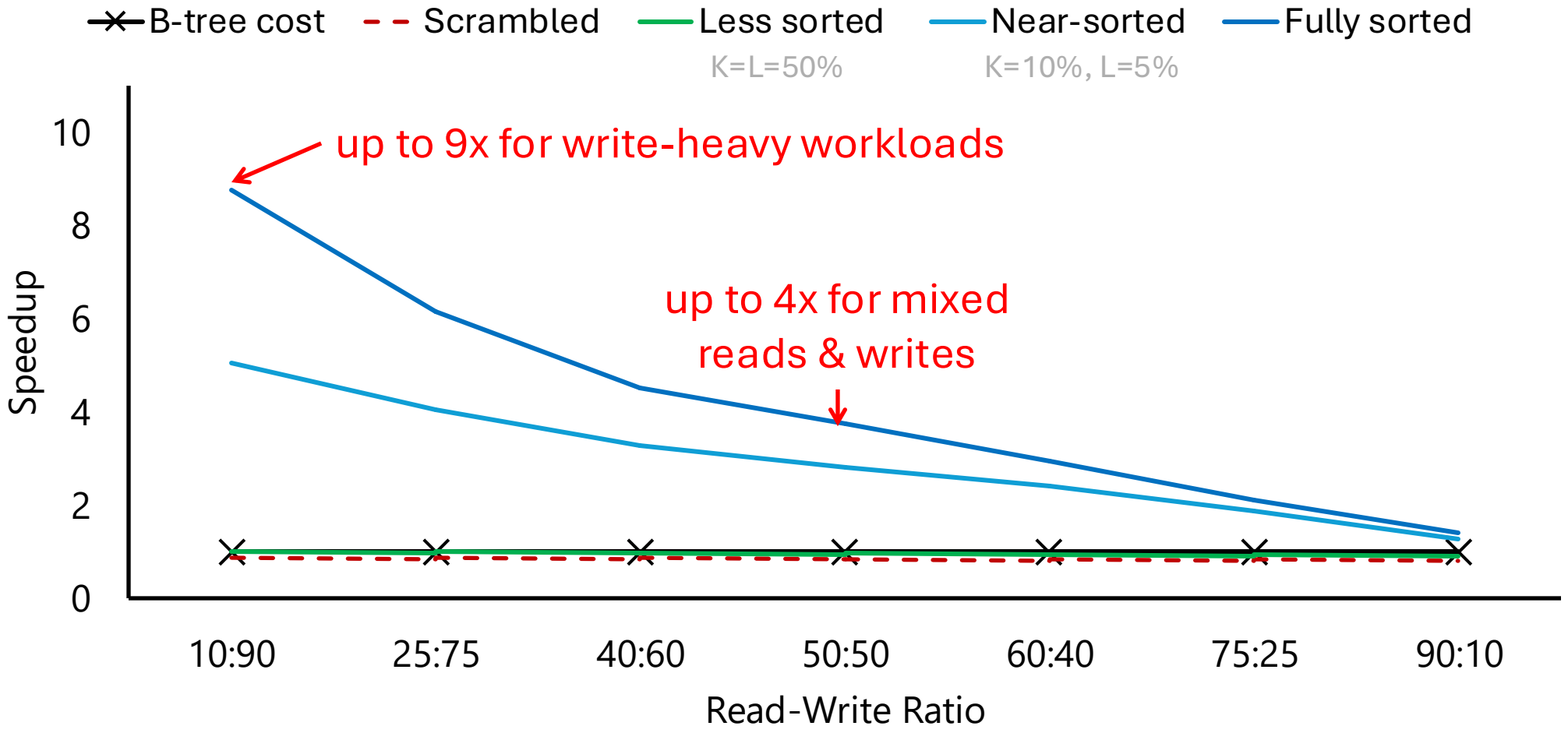
overlap causes "top-insert"



SWARE Bridges Bulk-loading & Top-Inserts



SWARE Benefits for ↑ Writes

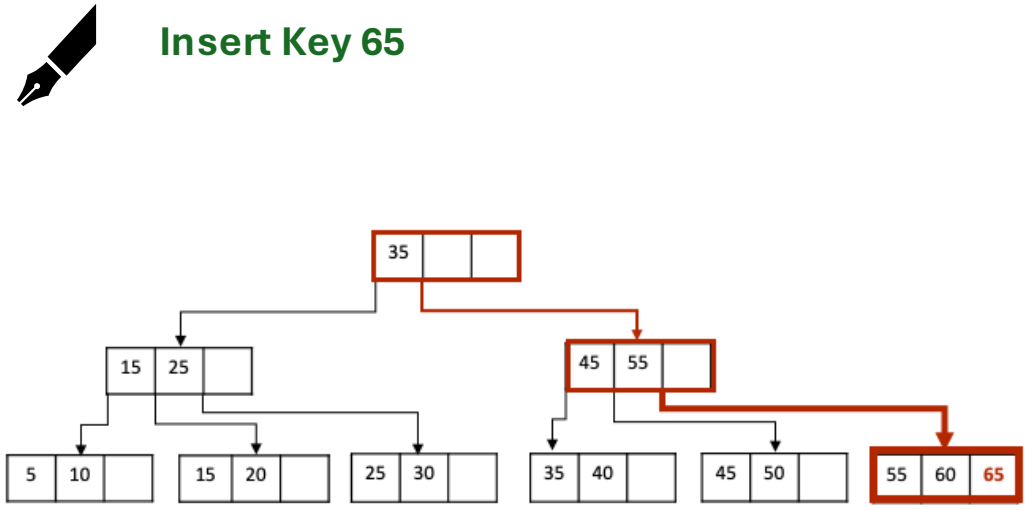


Use *sortedness* as a *resource*!

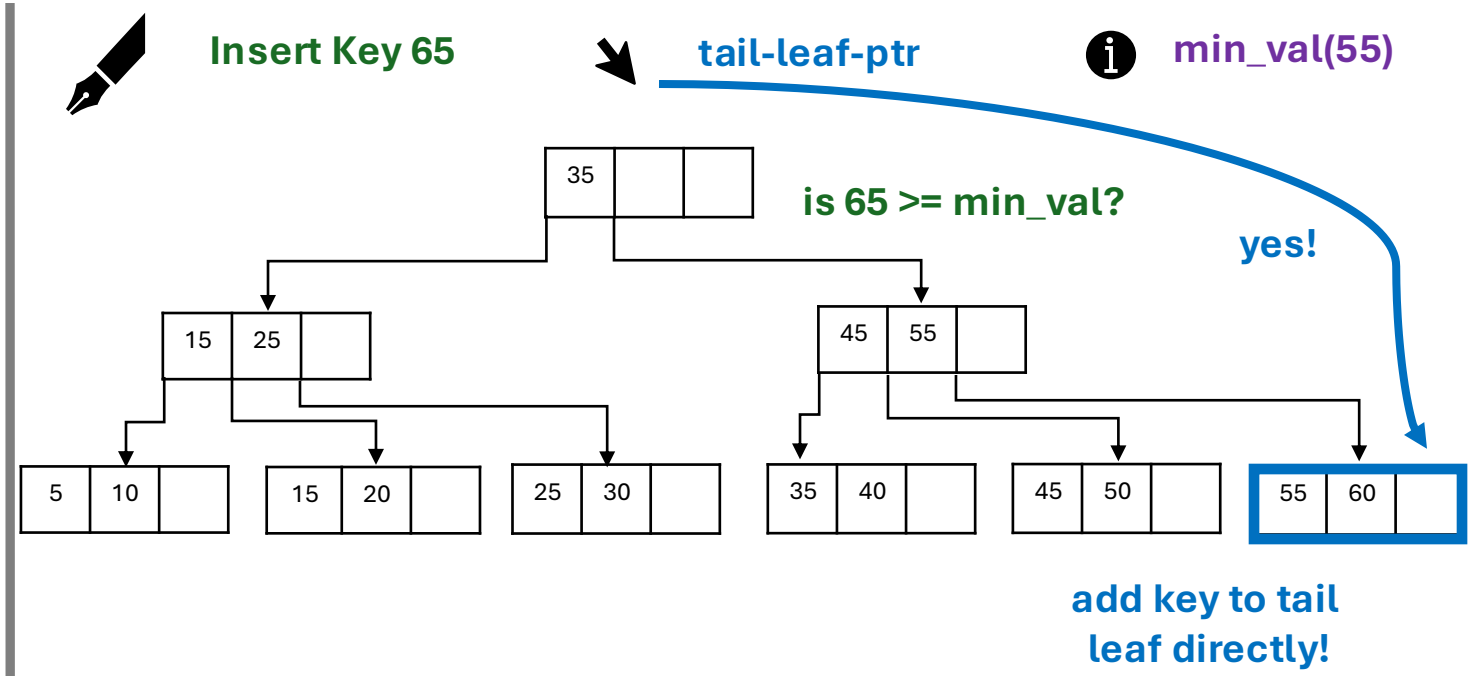
Can We Exploit Sortedness w/o Buffering?

Tail-Leaf Inserts in PostgreSQL

Normal Insertion (top-insert)

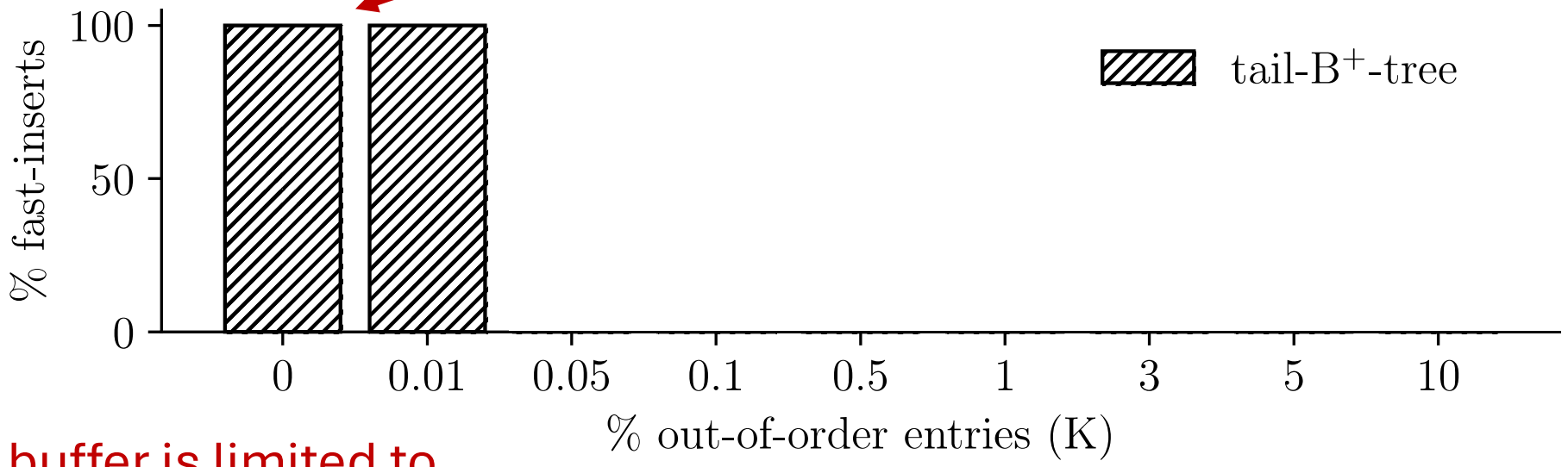


Tail-leaf Insertion



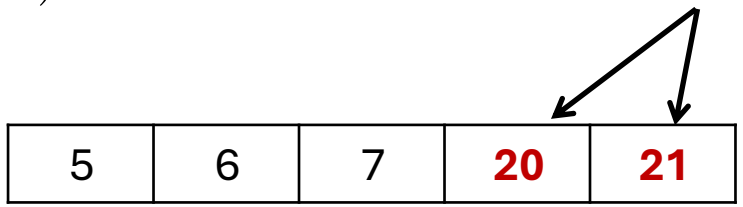
Does This Always Work?

Works for fully and very highly sorted data

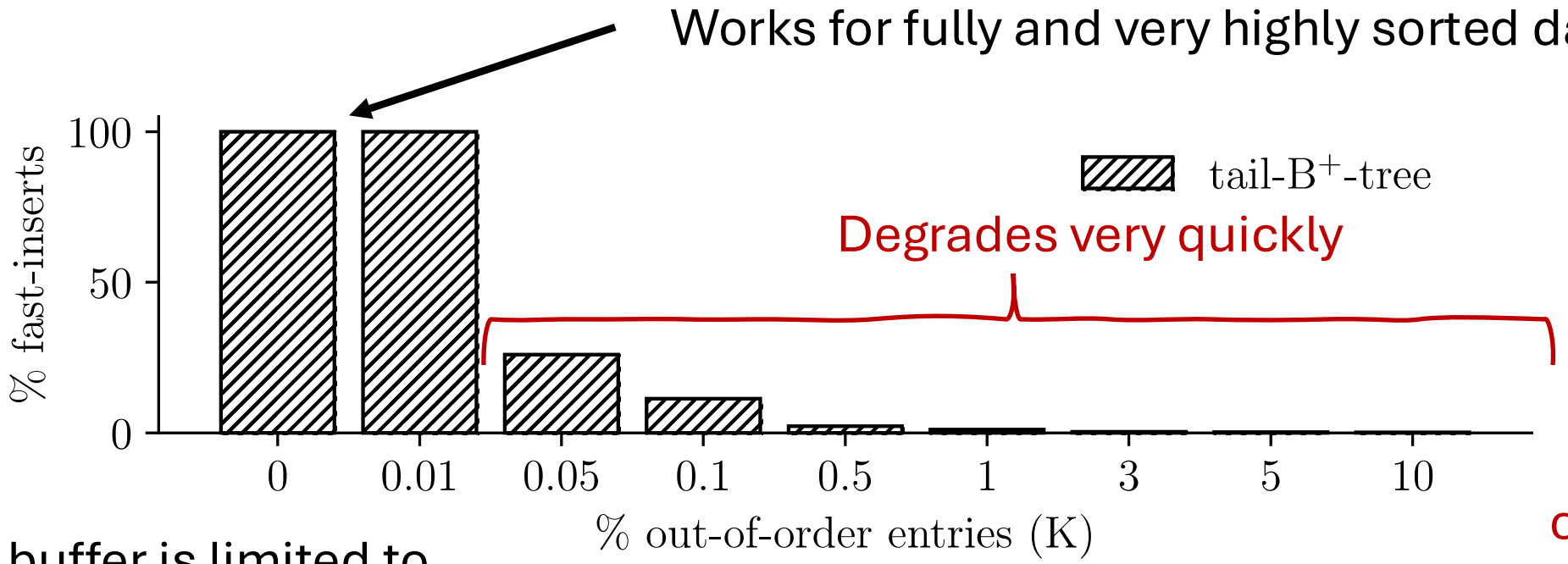


Tail-leaf's buffer is limited to leaf node!

outliers



Does This Always Work?

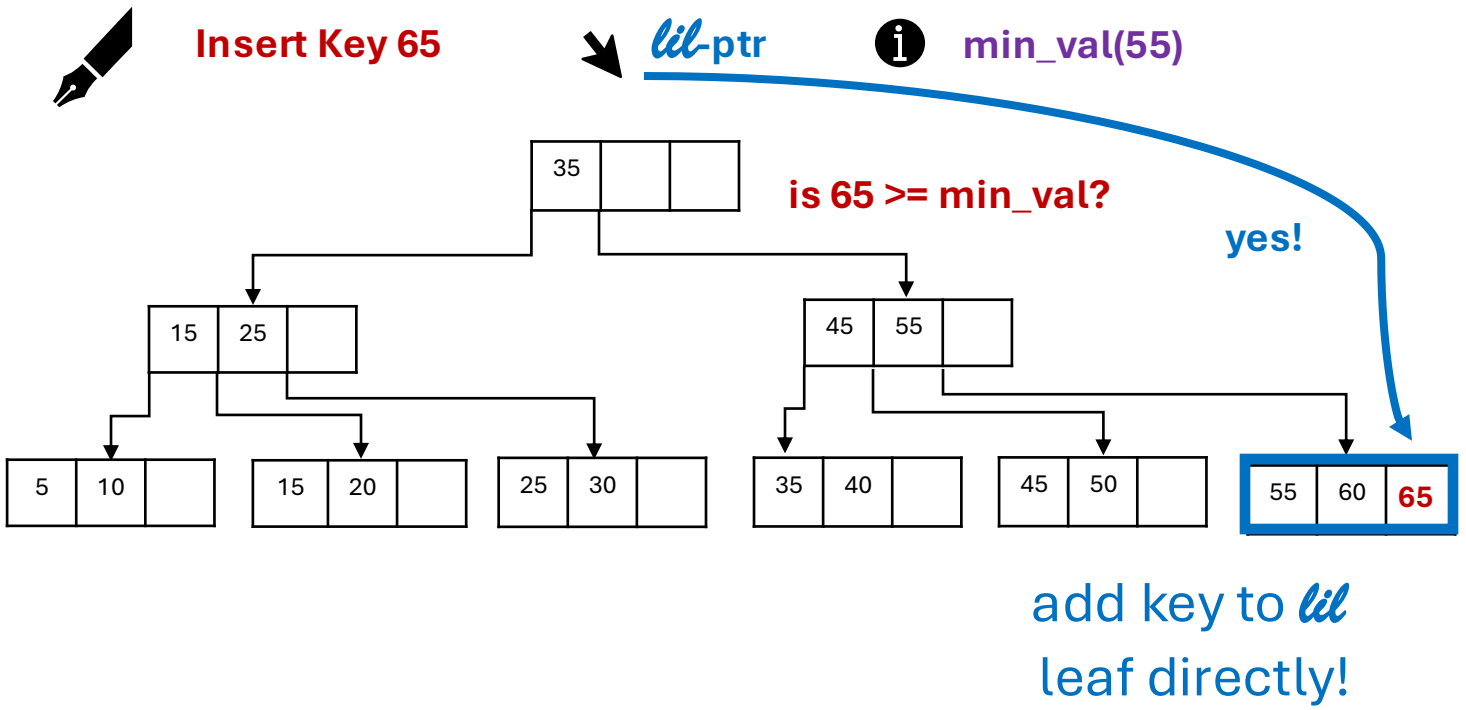


Tail-leaf's buffer is limited to leaf node!

outliers

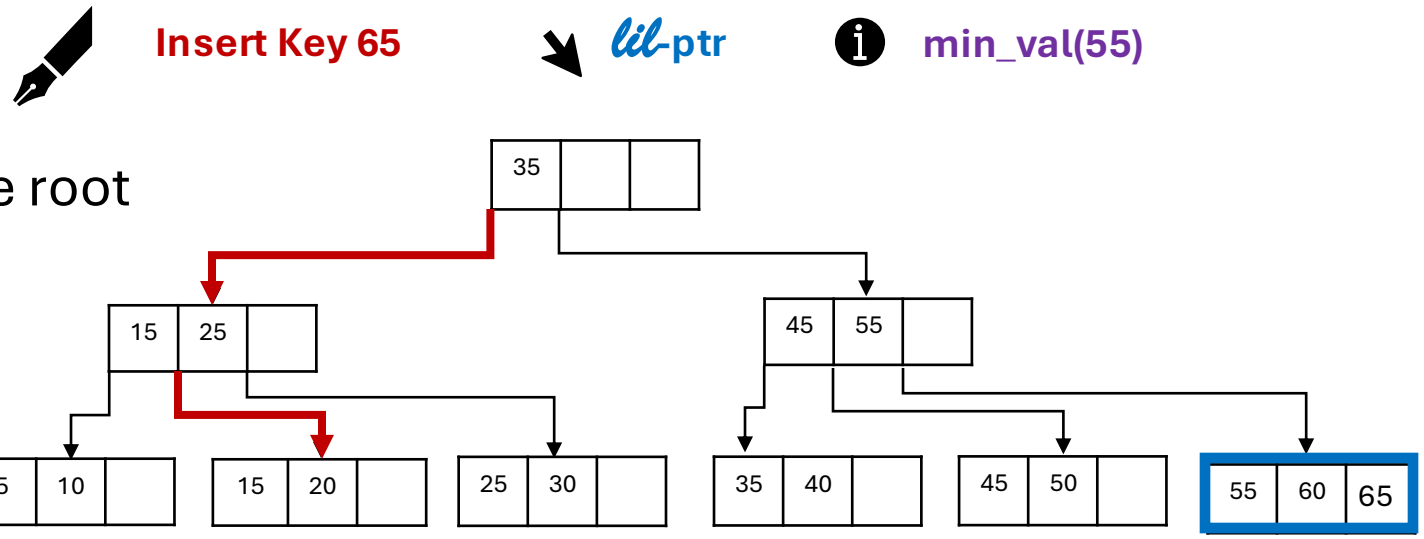
5	6	7	20	21

Last Insertion Leaf (*lil*)



What if we insert an out-of-order key instead?

Last Insertion Leaf *(lil)*



top-insert through the root

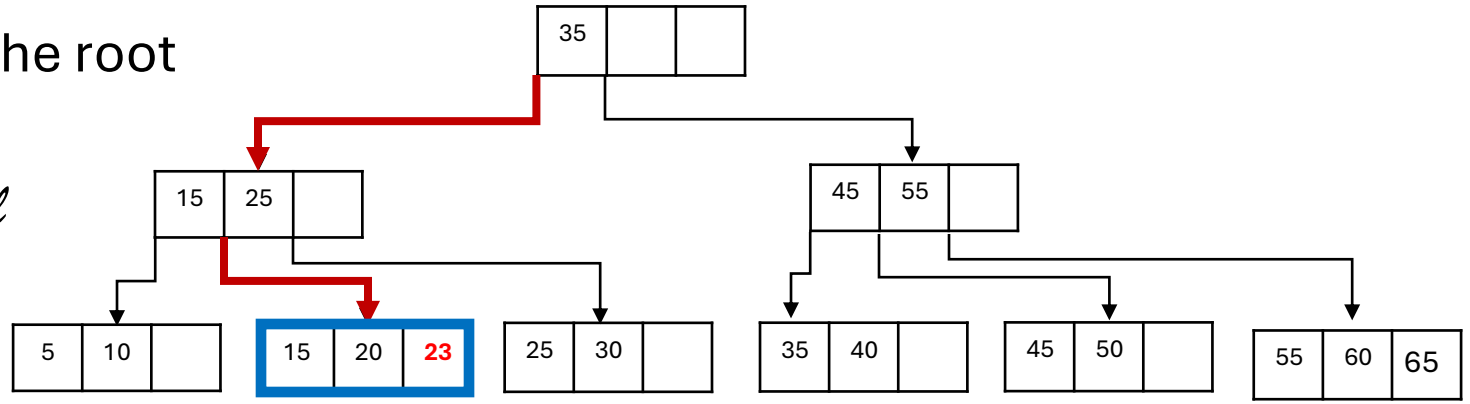
Last Insertion Leaf (*lil*)

also update min_val!

Insert Key 65 | *lil*-ptr | min_val(55)

top-insert through the root

update pointer to *lil*



Every top-insert will update *lil*

lil in Action

lil achieves higher fraction of fast-inserts

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 every out-of-order insert!**

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

Quick Insertion Tree

similar to **B⁺-tree** in design

+

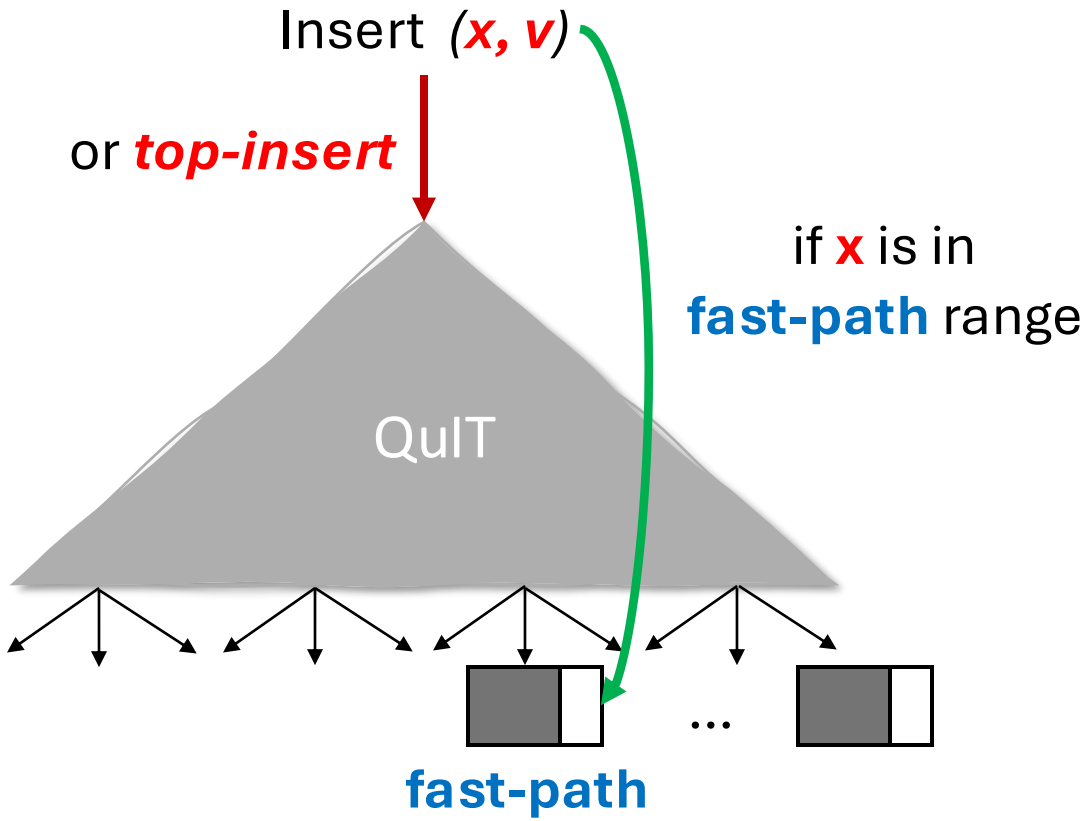
offers **fast-path** ingestion

+

sortedness-aware

+

minimal metadata + ***tuning***

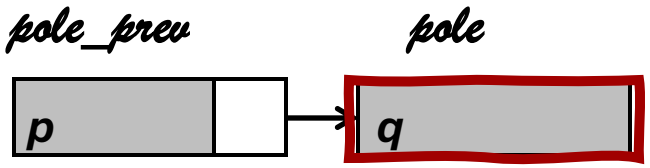


Predicting the Ordered Leaf (*pole*)

tail-leaf quickly fills up with outliers

lil naively switches fast-path

decision: when do we **update** the **fast-path** ?

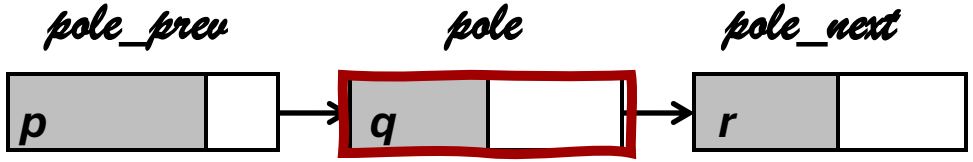


Predicting the Ordered Leaf (*pole*)

tail-leaf quickly fills up with outliers

lil naively switches fast-path

decision: when do we **update** the **fast-path** ?

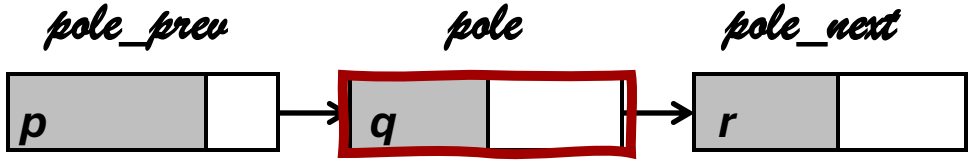


Predicting the Ordered Leaf (*pole*)

tail-leaf quickly fills up with outliers

lil naively switches fast-path

decision: when do we **update** the **fast-path** ?



predict using *I*n-order *K*ey estimato*R* (*IKR*)

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

capture (scale) small deviations

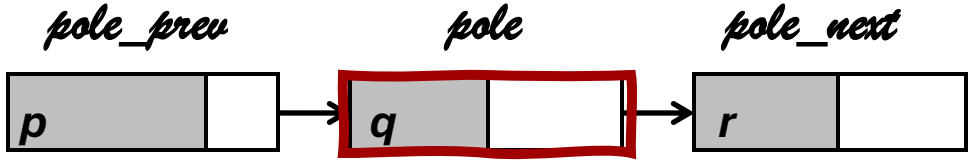
density between two non-outliers

Predicting the Ordered Leaf (*pole*)

tail-leaf quickly fills up with outliers

lil naively switches fast-path

decision: when do we **update** the **fast-path** ?



predict using *I*n-order *K*ey estimato*R* (*IKR*)

$$x = q + \left(\frac{q - p}{\lfloor pole\_prev_{size} \rfloor}\right) \cdot pole_{size}$$

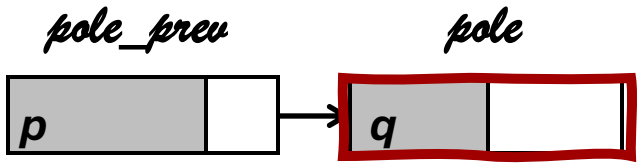
capture (scale) small deviations

density between two non-outliers

any **key** > **x** is an **outlier**

Predicting the Ordered Leaf (*pole*)

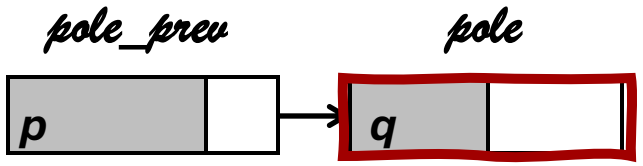
```
if(key is within pole range){
     if(pole.size < capacity){
          pole.insert(entry);
     }
     else{
          pole_next = pole.split();
          r = pole_next.min;
          x = IKR(q, p, pole_prev.size, pole.size);
          if(r <= x){
               pole_prev = pole;
               pole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



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);
}
```

check if *key* qualifies for a fast-insert

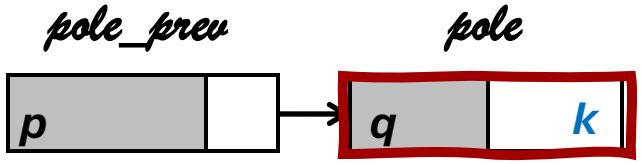


otherwise, *top-insert* through root

Predicting the Ordered Leaf (*pole*)

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

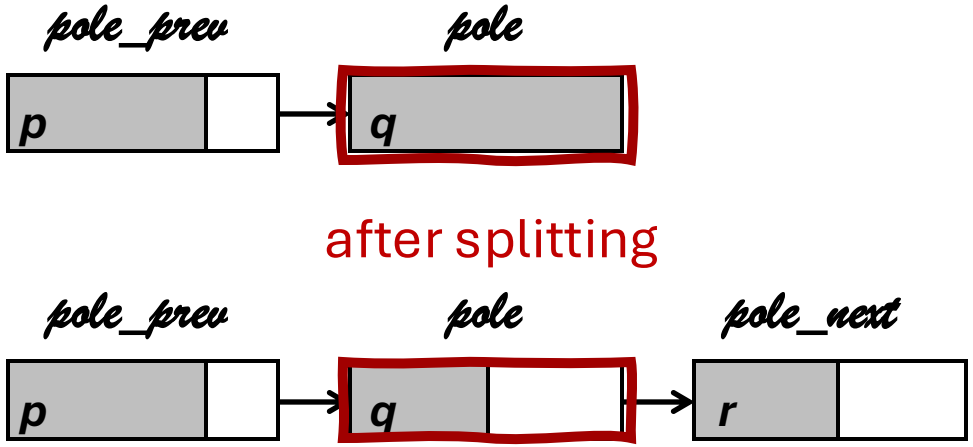
if *pole* is not full, simply insert



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);
}
```

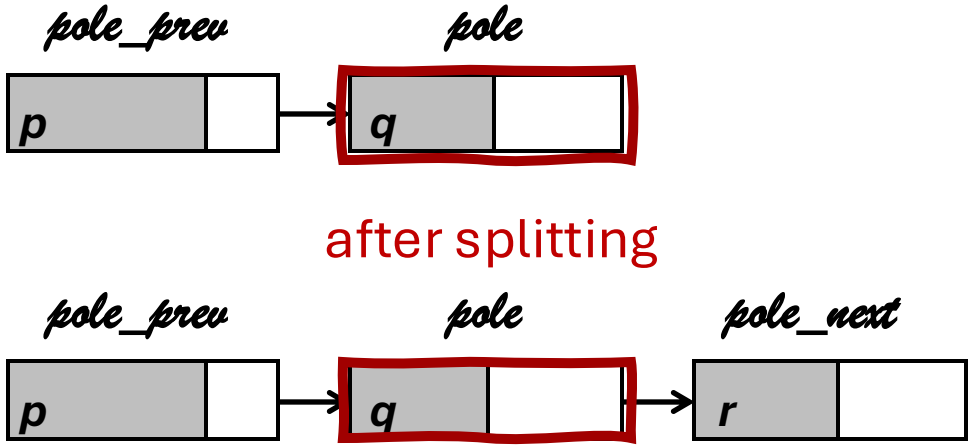
otherwise, split *pole*



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);
}
```

calculate *x*



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

density between two non-outliers

capture small deviations

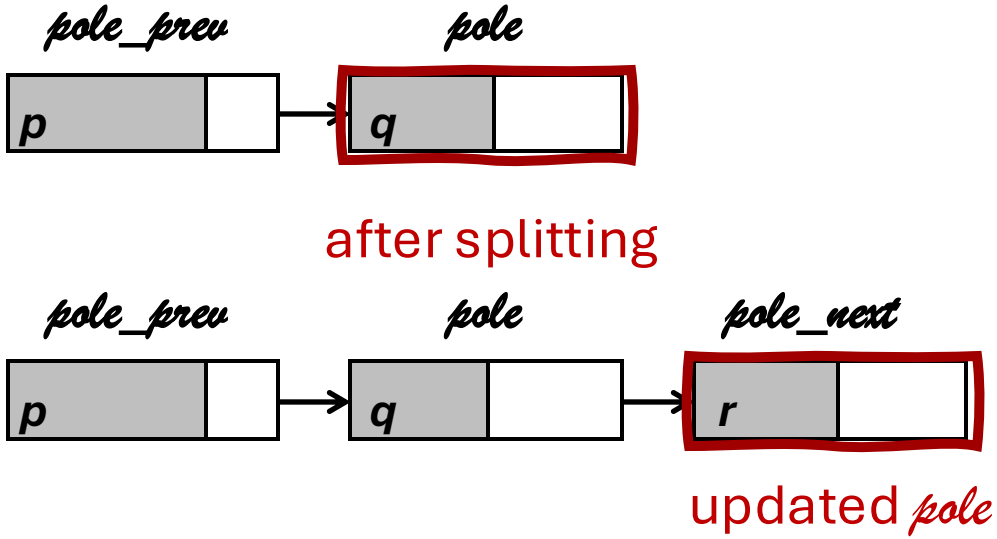
default to ***1.5***

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);
}
```

if *pole_next* has at least one non-outlier

otherwise, *pole* remains as is



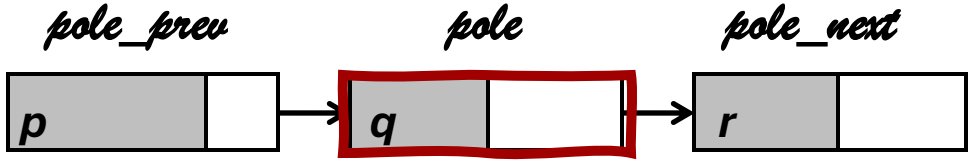
Can We Better Utilize Space?

high sortedness => poor space utilization

can we find better split points?

IKR can also return the split point

return last key <= x in *pole*

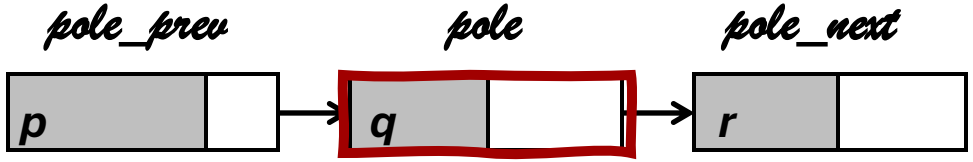


Can We Better Utilize Space?

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;
}
```



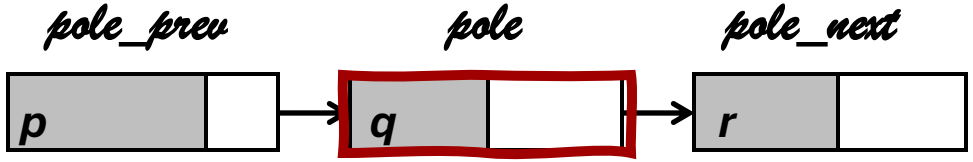
let's call this key ℓ

Can We Better Utilize Space?

high sortedness => poor space utilization

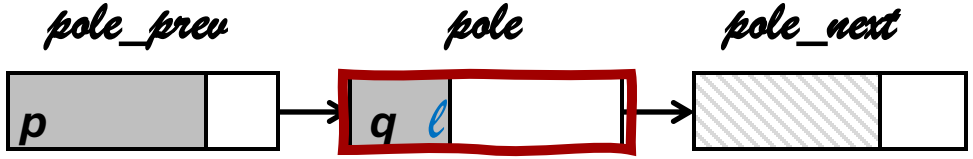
```
l, l_pos = IKR(q, p, pole_prev.size);

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

after splitting, if $\ell\_pos \leq 50\%$



split at ℓ

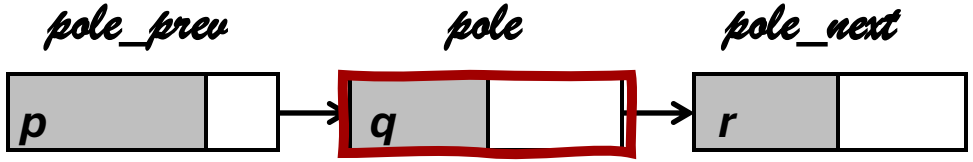
this moves all outliers to *pole_next*

Can We Better Utilize Space?

high sortedness => poor space utilization

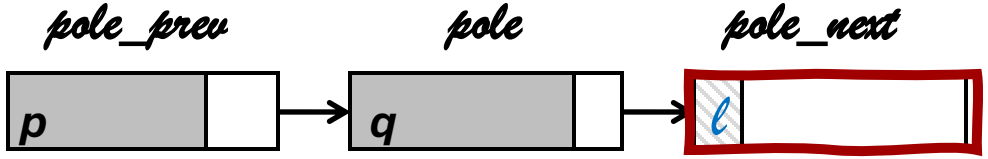
```
l_pos = IKR(q, p, pole_prev.size);

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



let's call this key ℓ

after splitting, if ℓ_pos > 50%



split at ℓ_pos - 1 updated *pole*

this moves at least one non-outlier to ***pole_next***

Evaluating QuIT

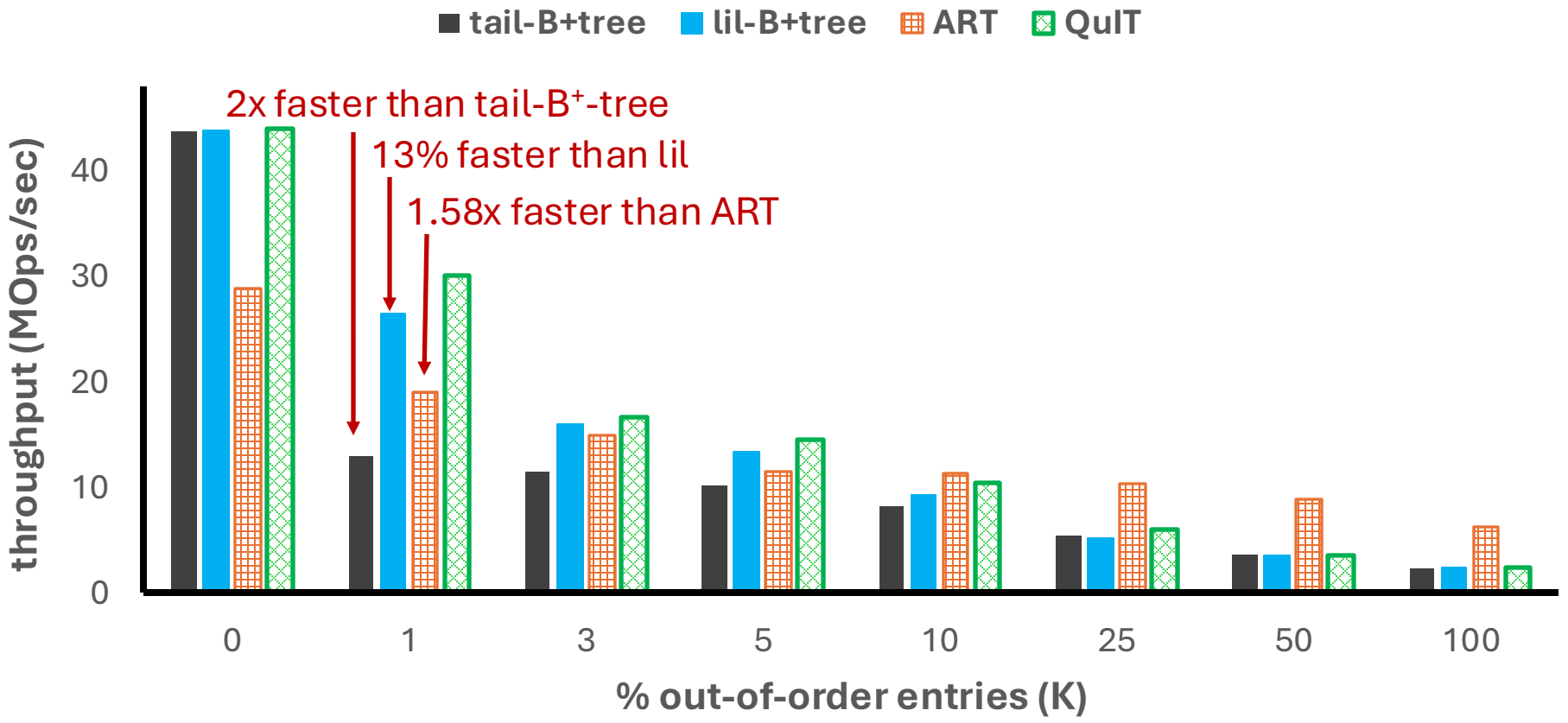
System:

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

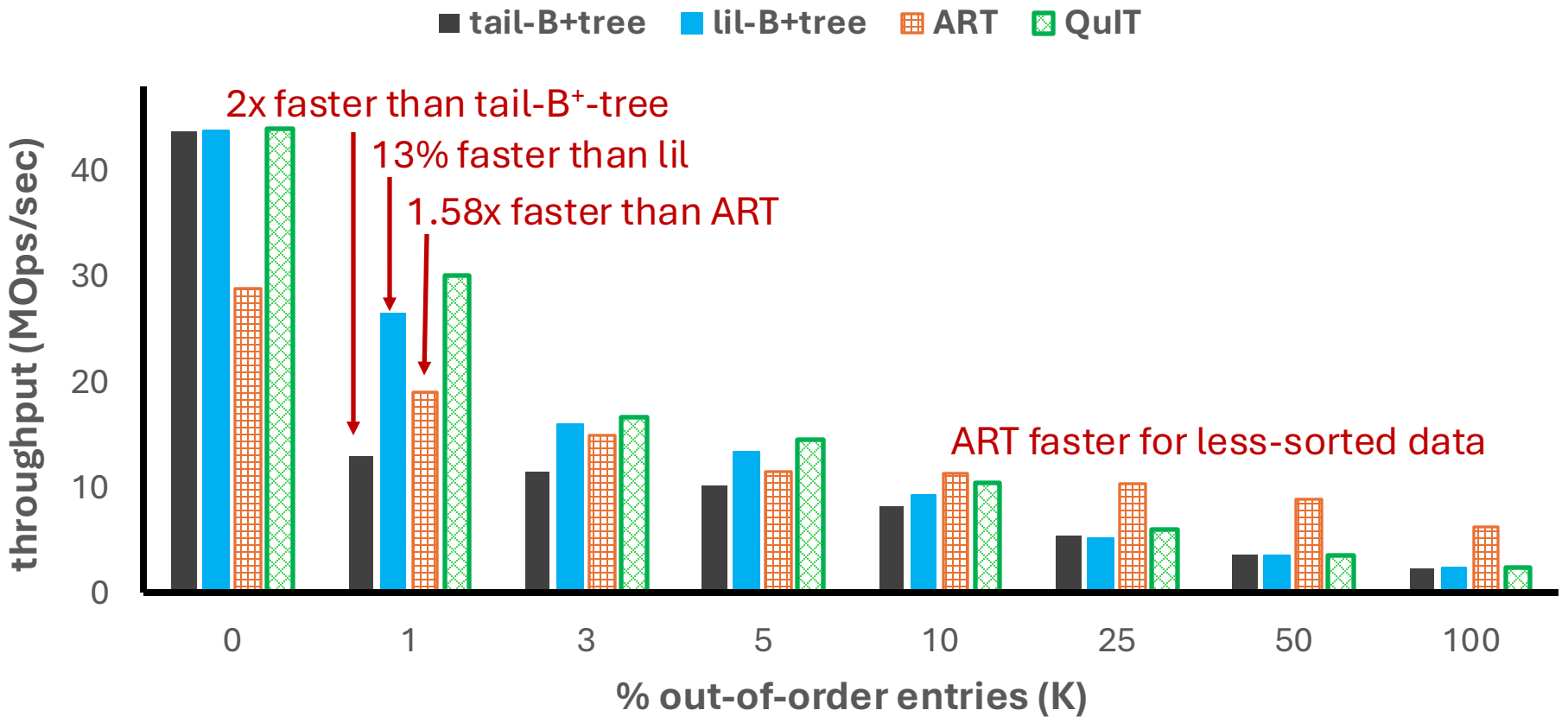
Index Setup:

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

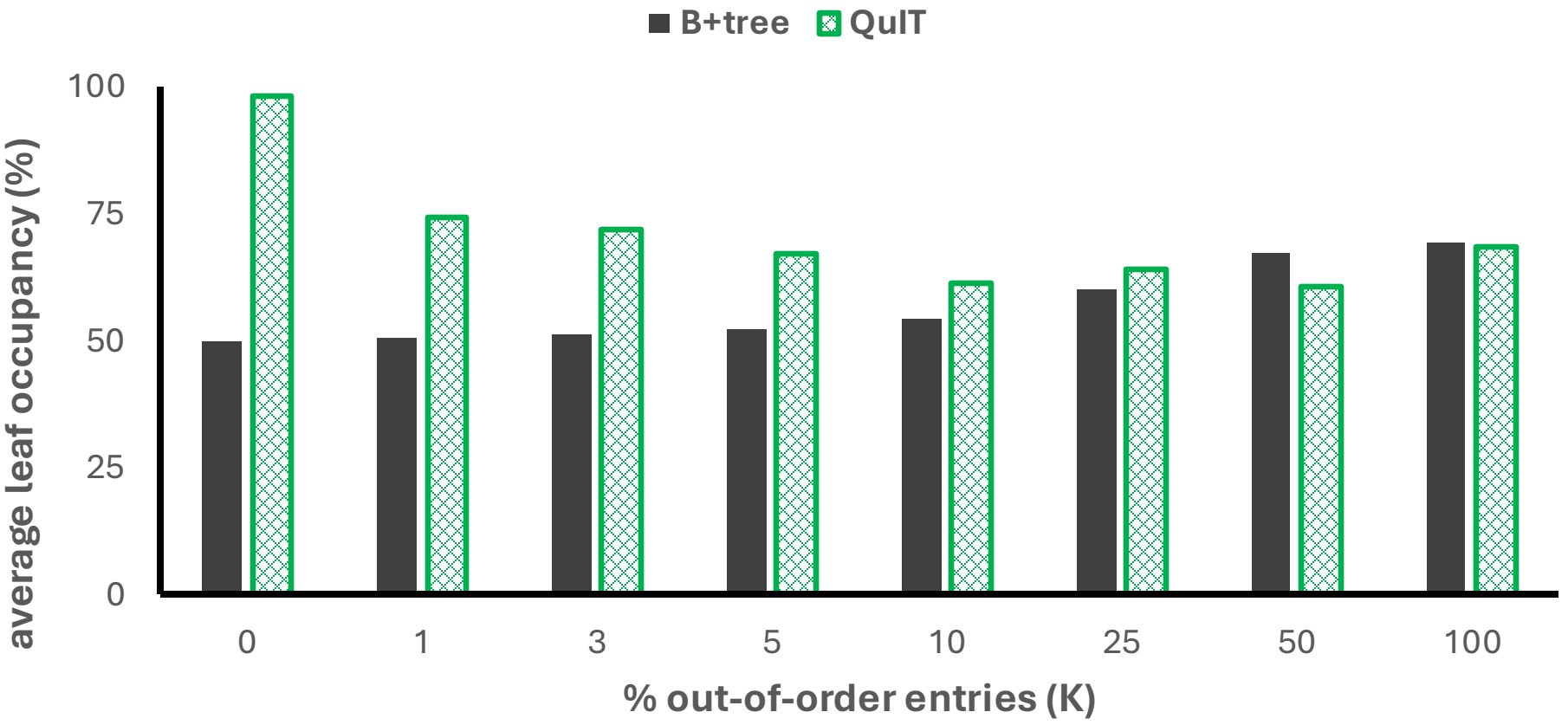
QuIT Outperforms All Baselines



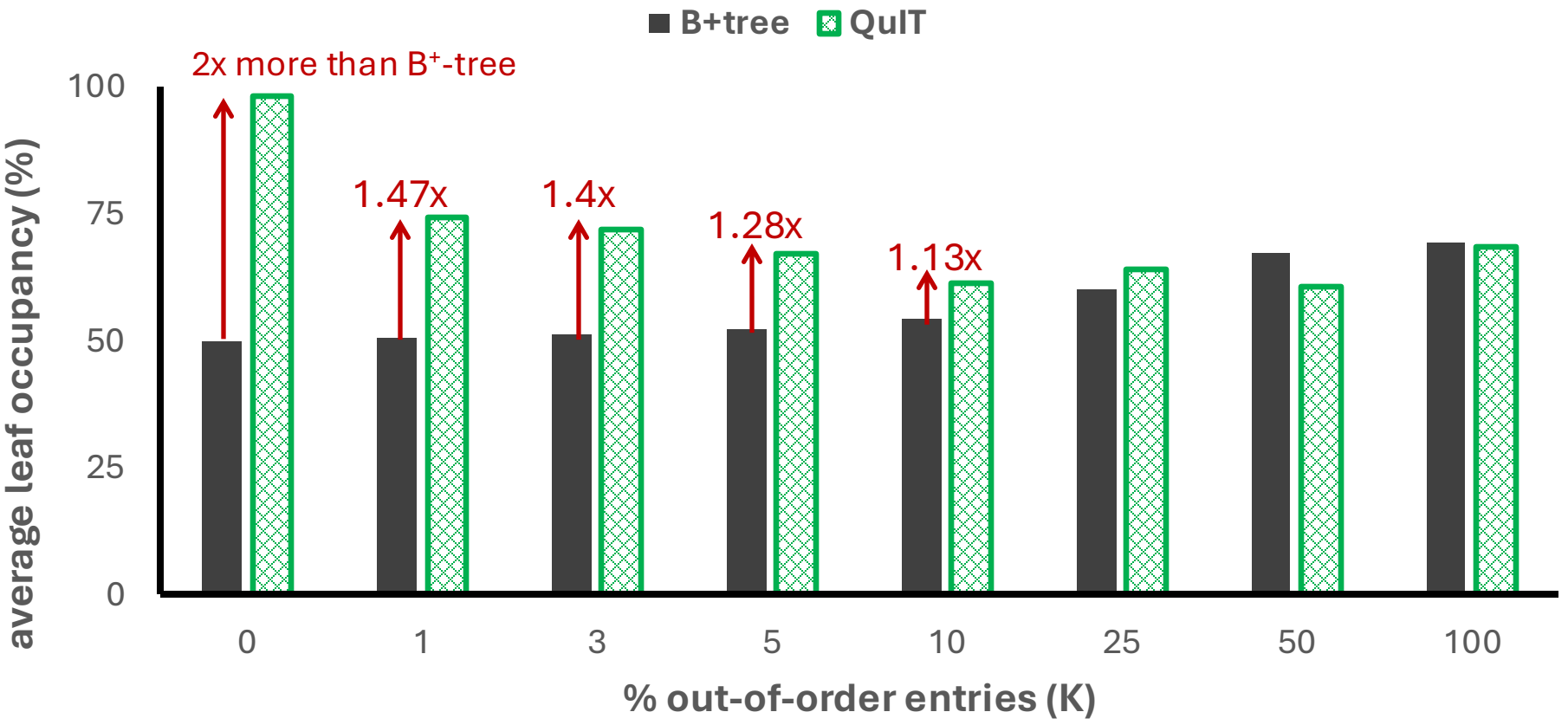
QuIT Outperforms All Baselines



QuIT Increases Occupancy in L

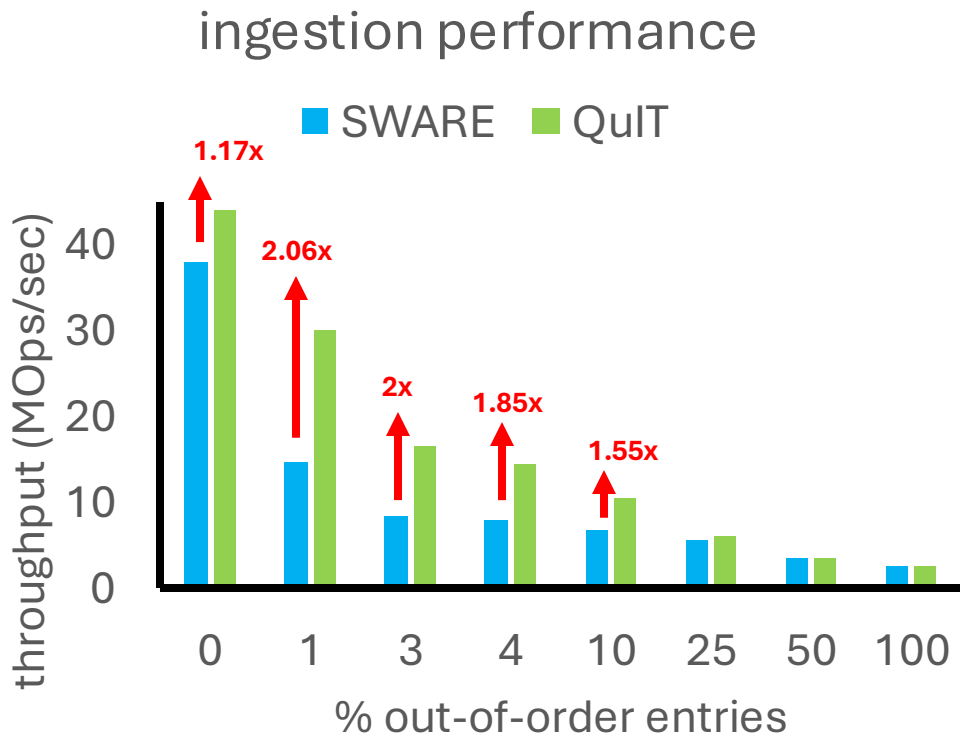


QuIT Increases Occupancy in L



higher leaf occupancy reduces memory footprint✓

QuIT v/s SWARE



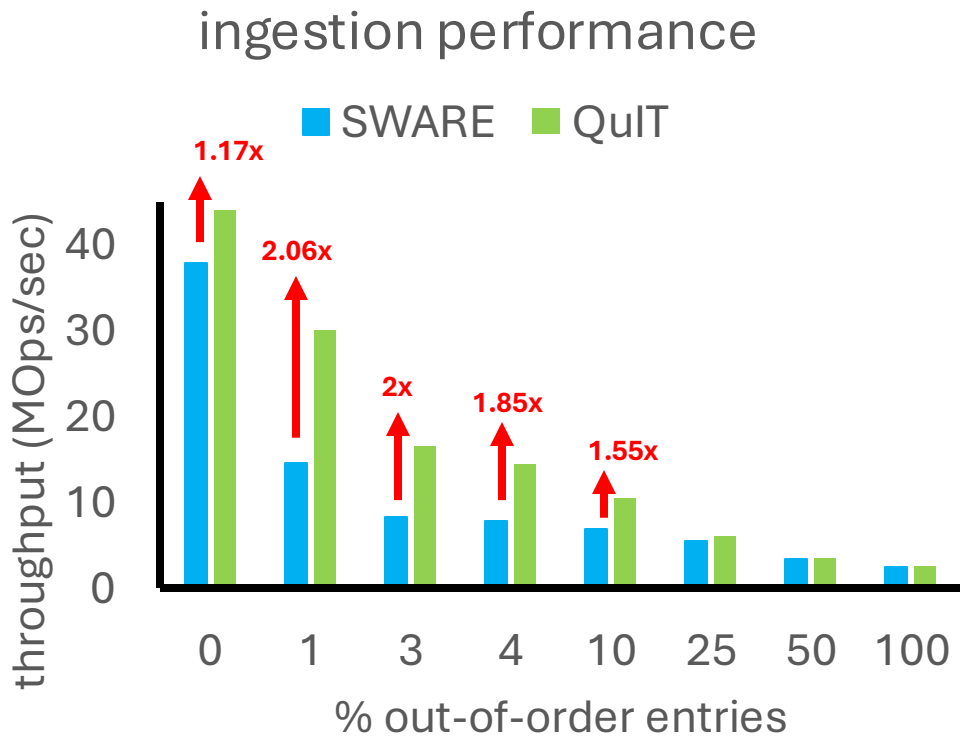
integrate SWARE with same B⁺-tree as QuIT

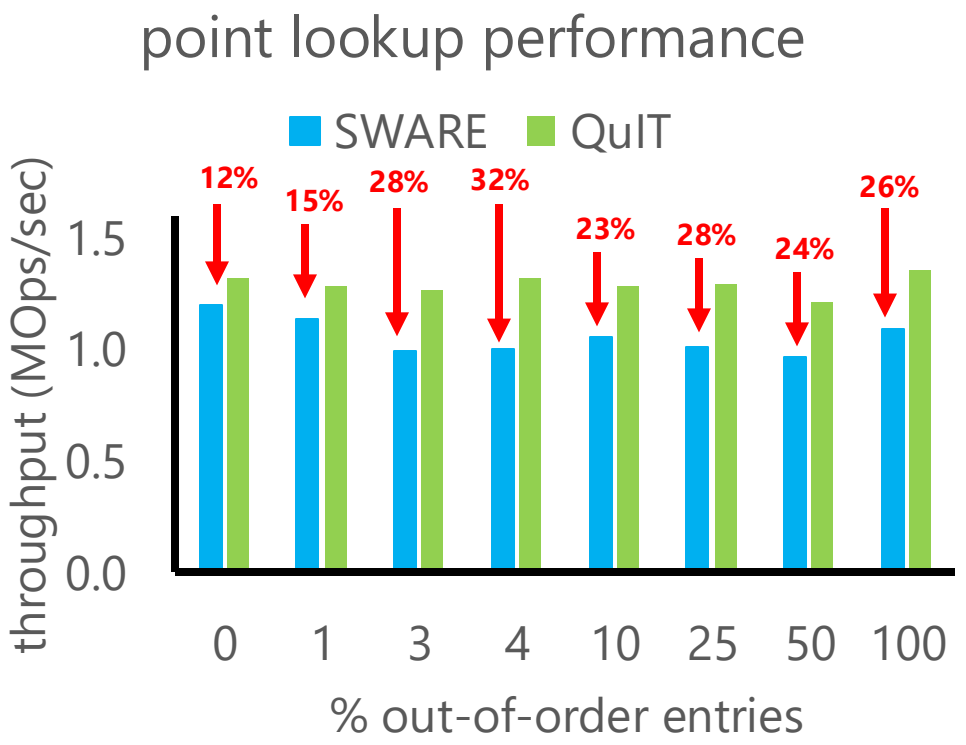
up to 2.06x faster

avoids SWARE buffer management ✓

minimal metadata ✓

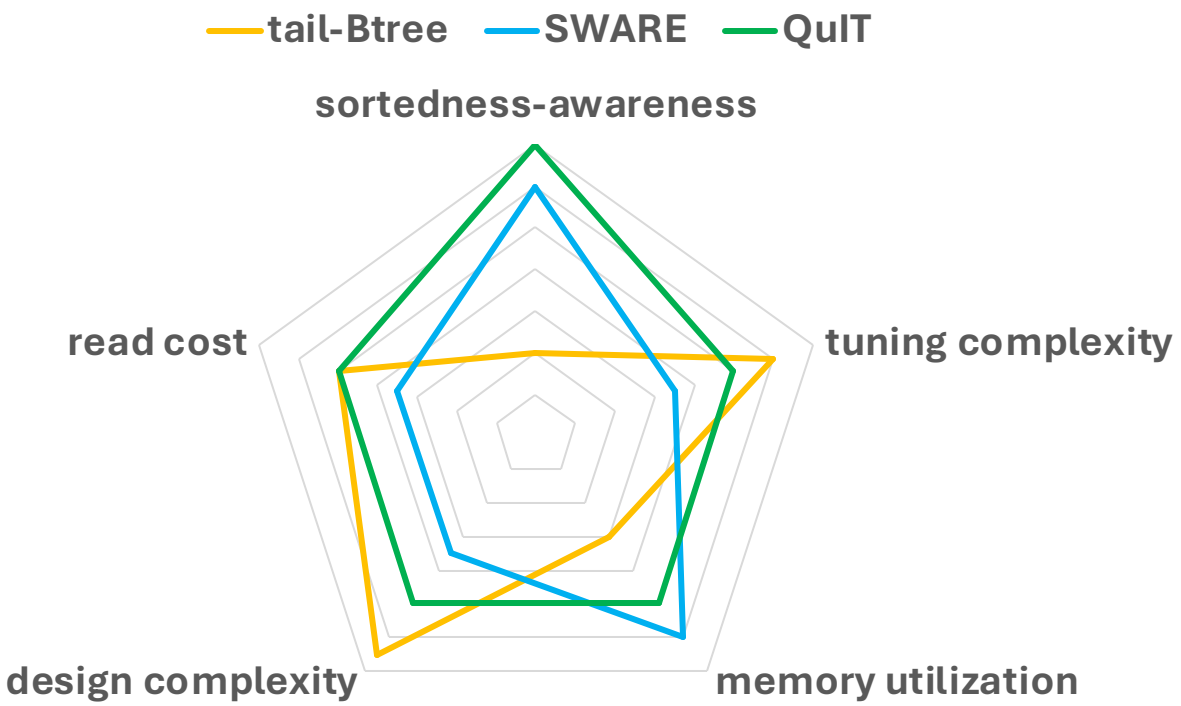
QuIT v/s SWARE





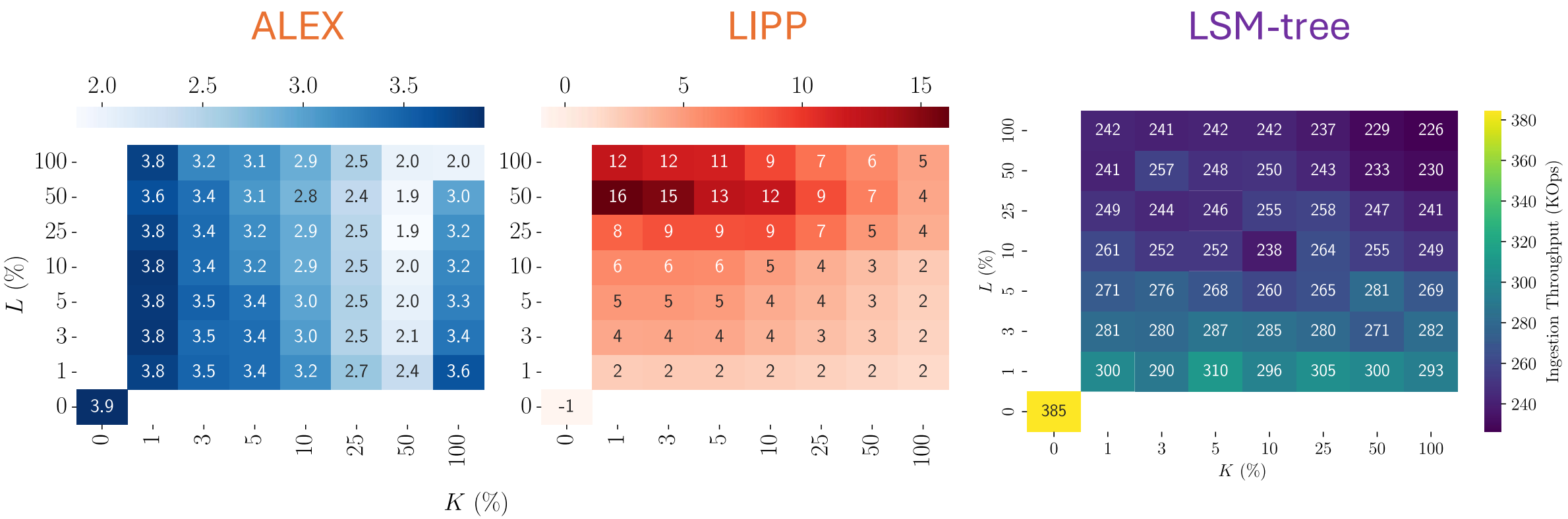
No buffering ⇒ no read overhead!

Qualitatively Comparing SWARE & QuIT



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

Benchmarking Learned and LSM Indexes



Summary

Identify “sortedness” as a resource to reduce indexing cost

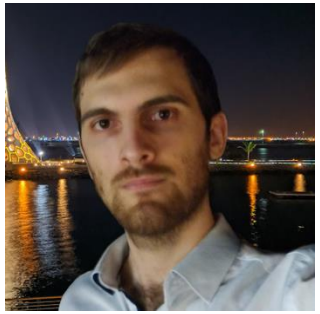
SWARE framework offers sortedness-awareness to any index, but incurs read penalty

QuIT offers fast ingestion with no read penalty + lightweight design

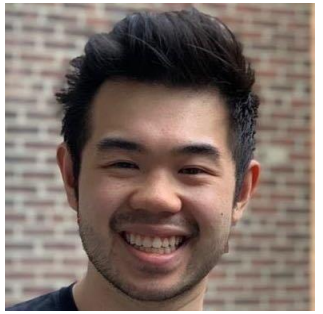
Currently exploring concurrency control for QuIT

Our Team

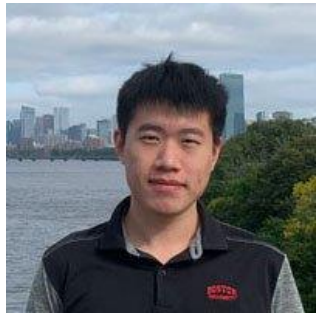
Aneesh Raman



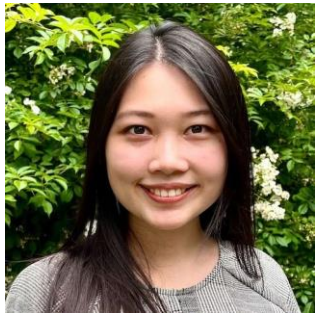
Kostas Karatsenidis



Andy Huynh



Jinqi Lu



Shaolin Xie

Subhadeep Sarkar

Matthaios Olma