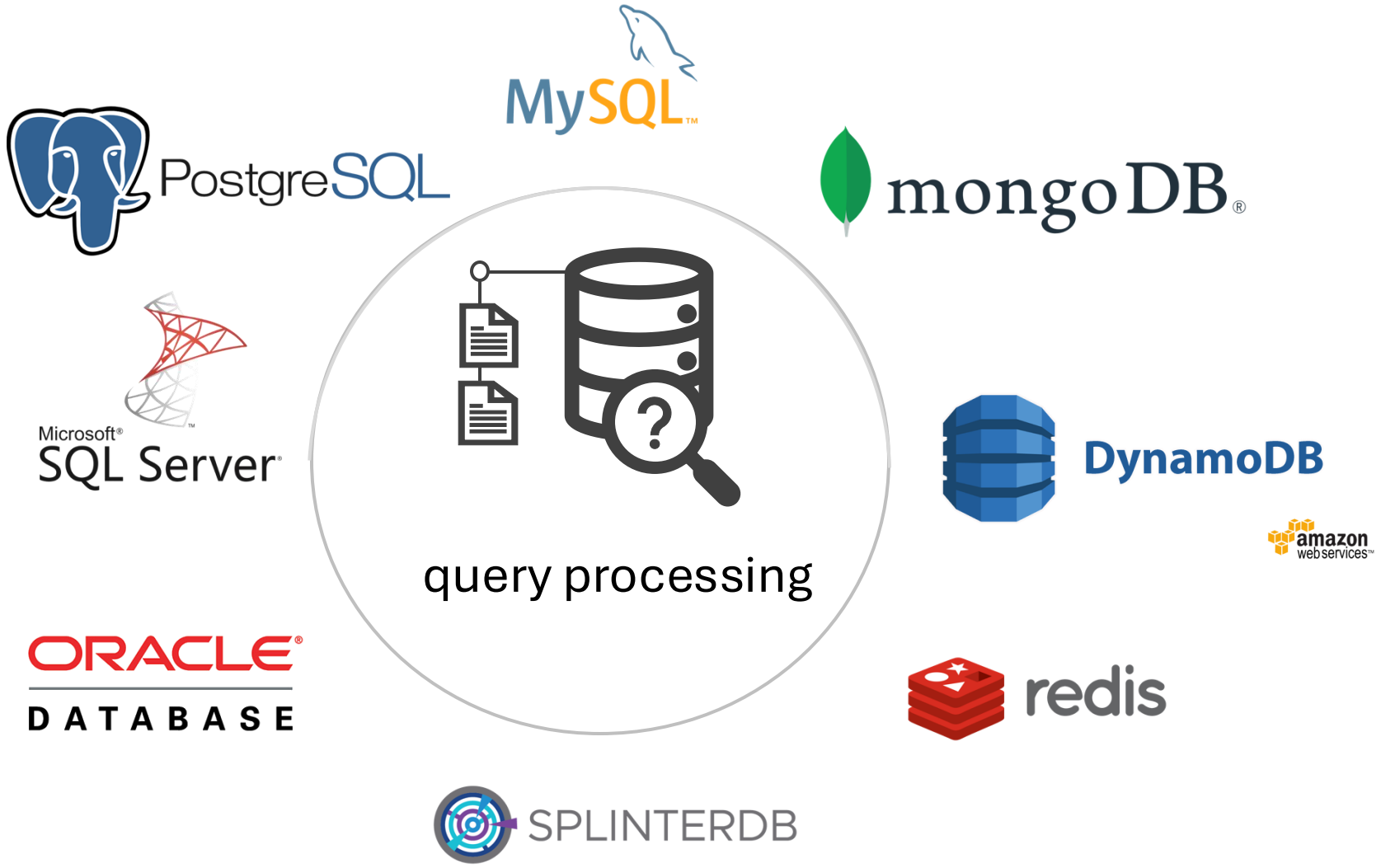
QuIT your B$^+$-tree for the *Quick Insertion Tree*

EDBT 2025

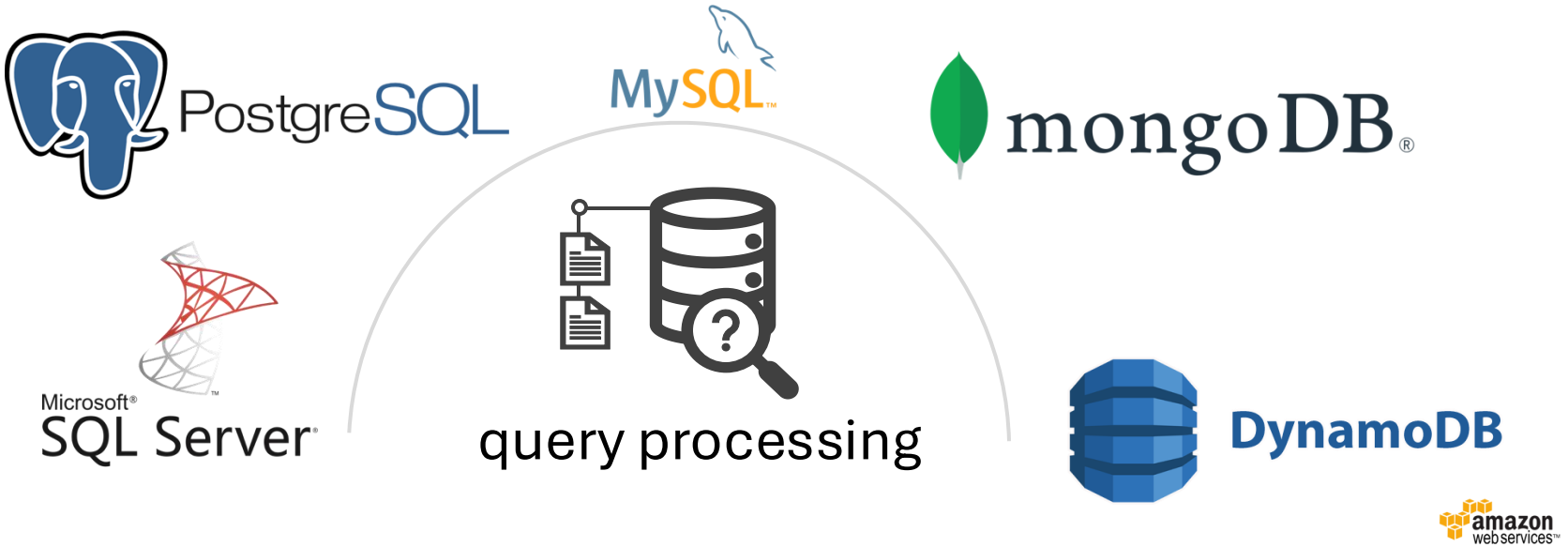
Aneesh Raman*, Konstantinos Karatsenidis*, Shaolin Xie,

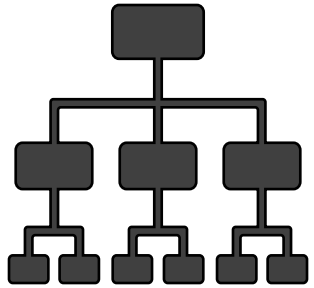
Matthaios Olma, Subhadeep Sarkar, Manos Athanassoulis

Indexes Are Everywhere!



Indexes Are Everywhere!

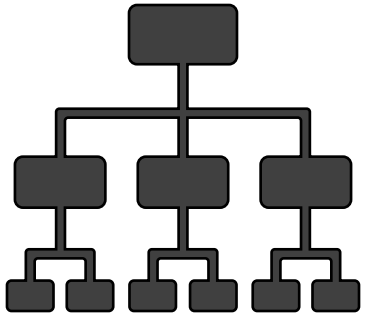




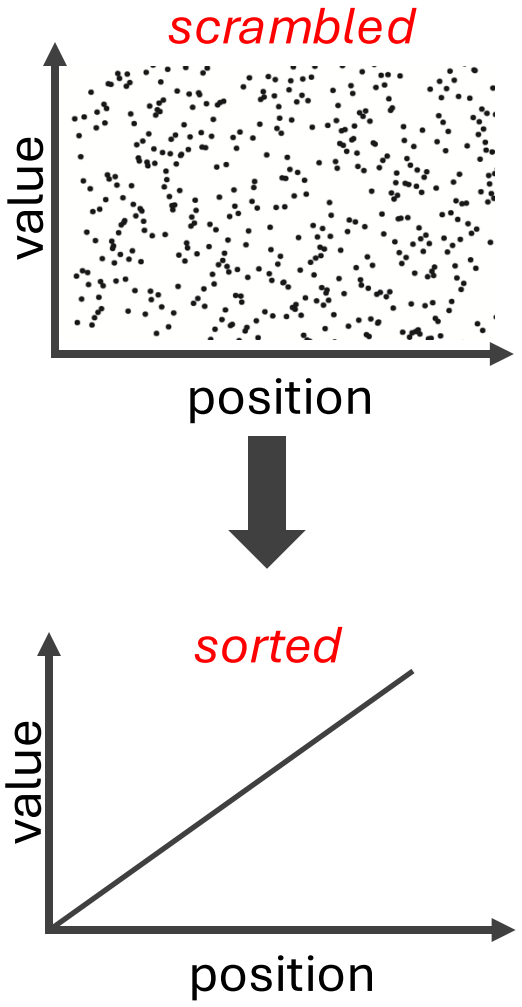
organize data

accelerates lookups

Indexing Adds Structure

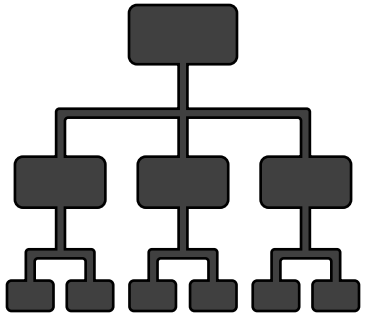


organize data

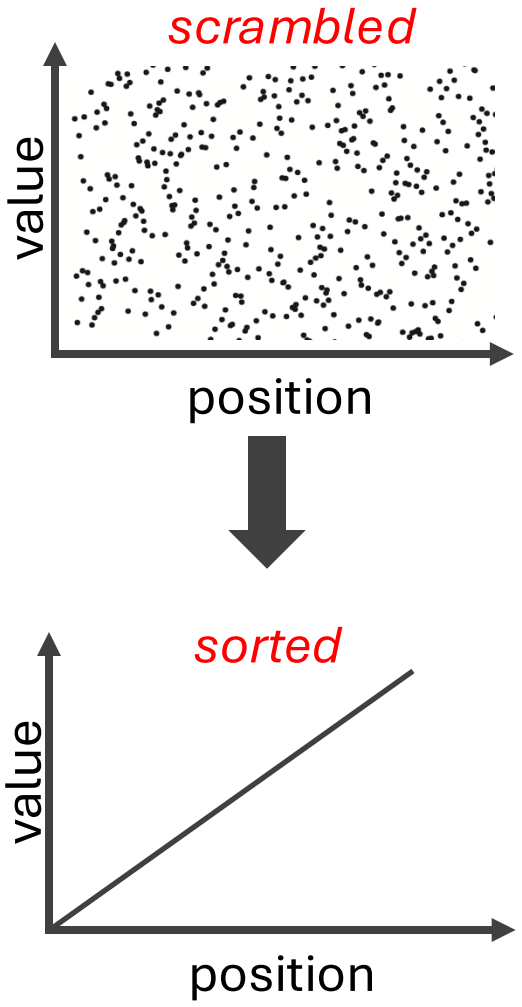


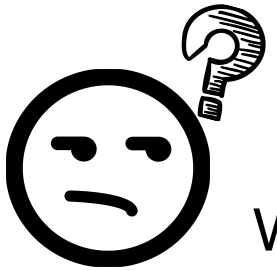
The process of ***"inducing sortedness"*** to otherwise, unsorted data

Indexing Adds Structure



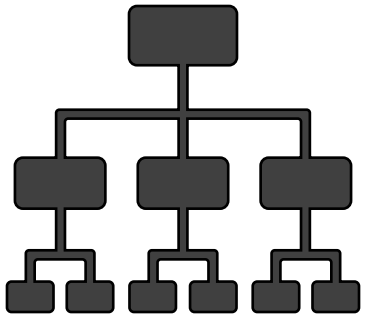
organize data



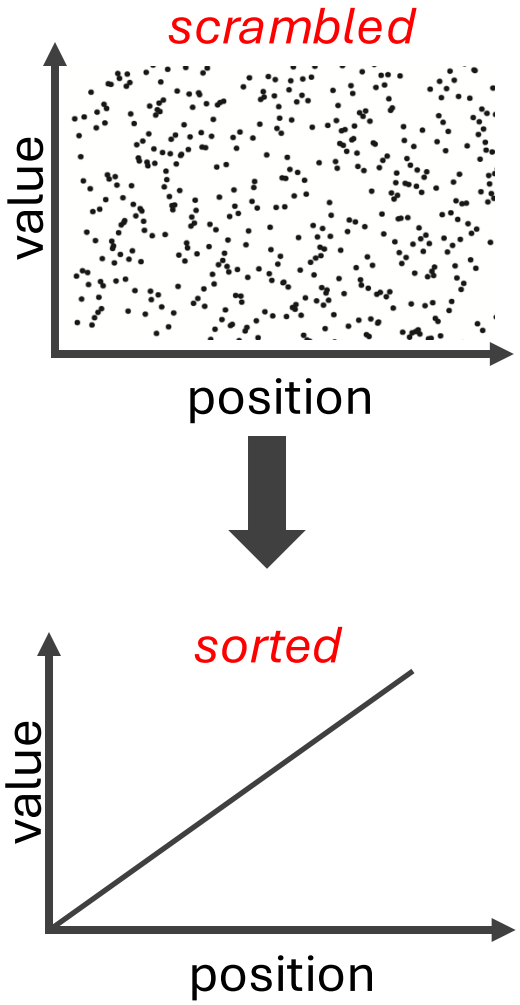


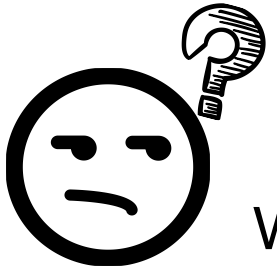
What if the incoming data is **already sorted?**

Indexing Adds Structure



organize data





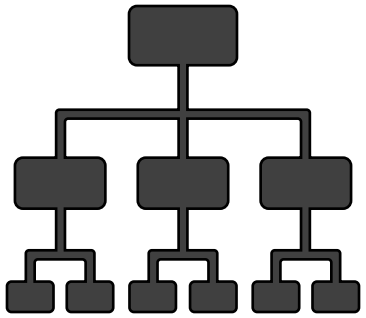
What if the incoming data is **already sorted?**

fully sorted?

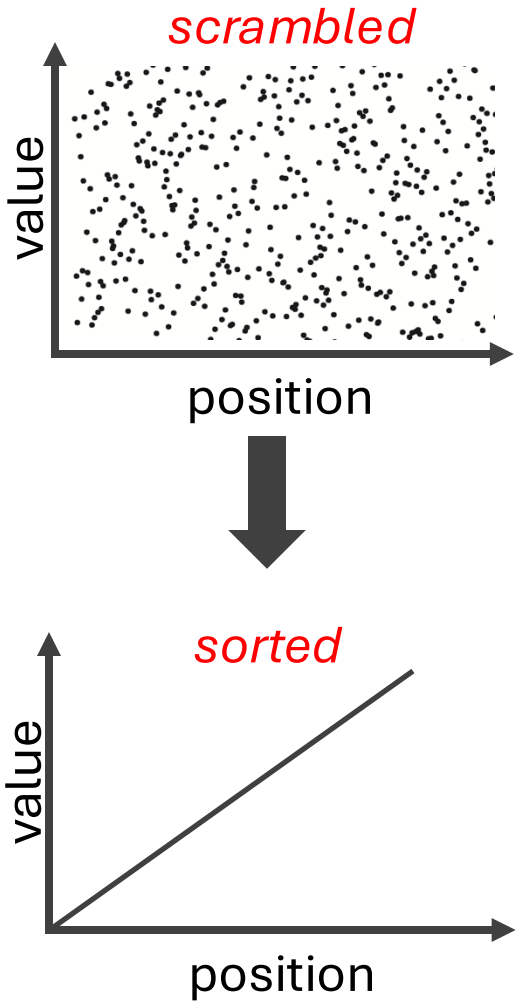
easy! => **bulk load** the data

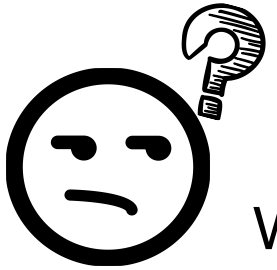
how about **near-sorted data?**

Indexing Adds Structure

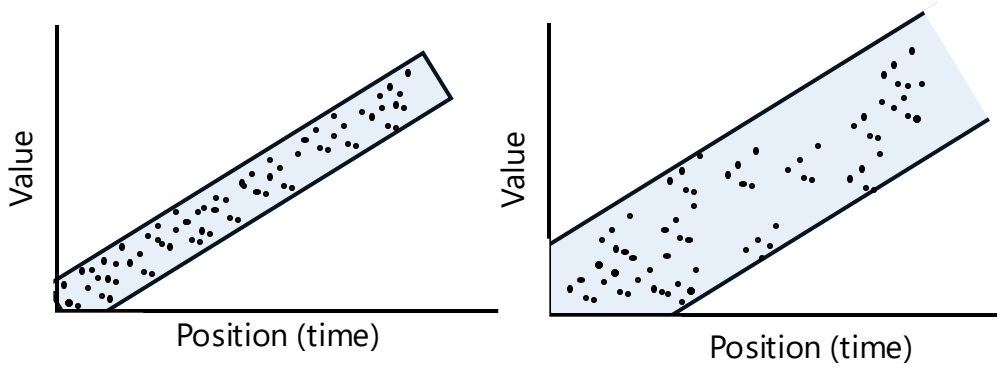


organize data



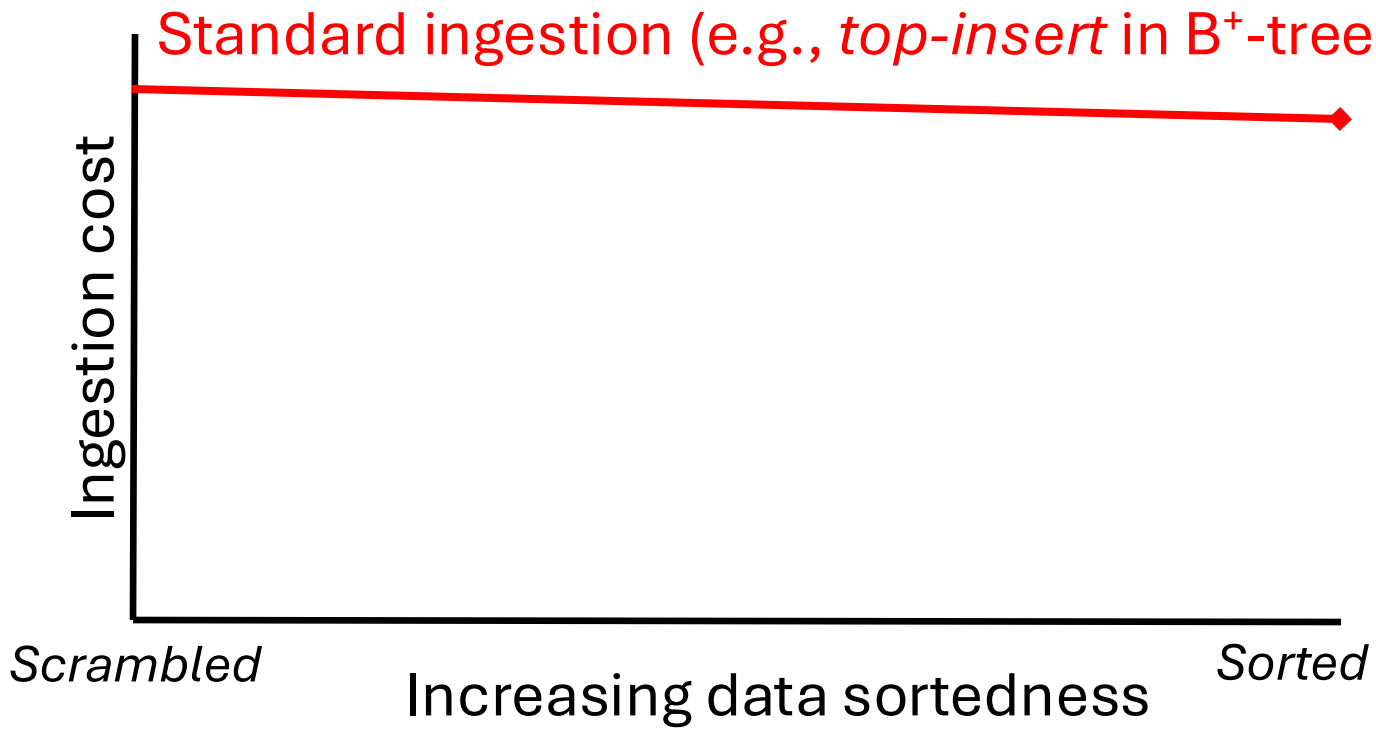


What if the incoming data is **already sorted?**

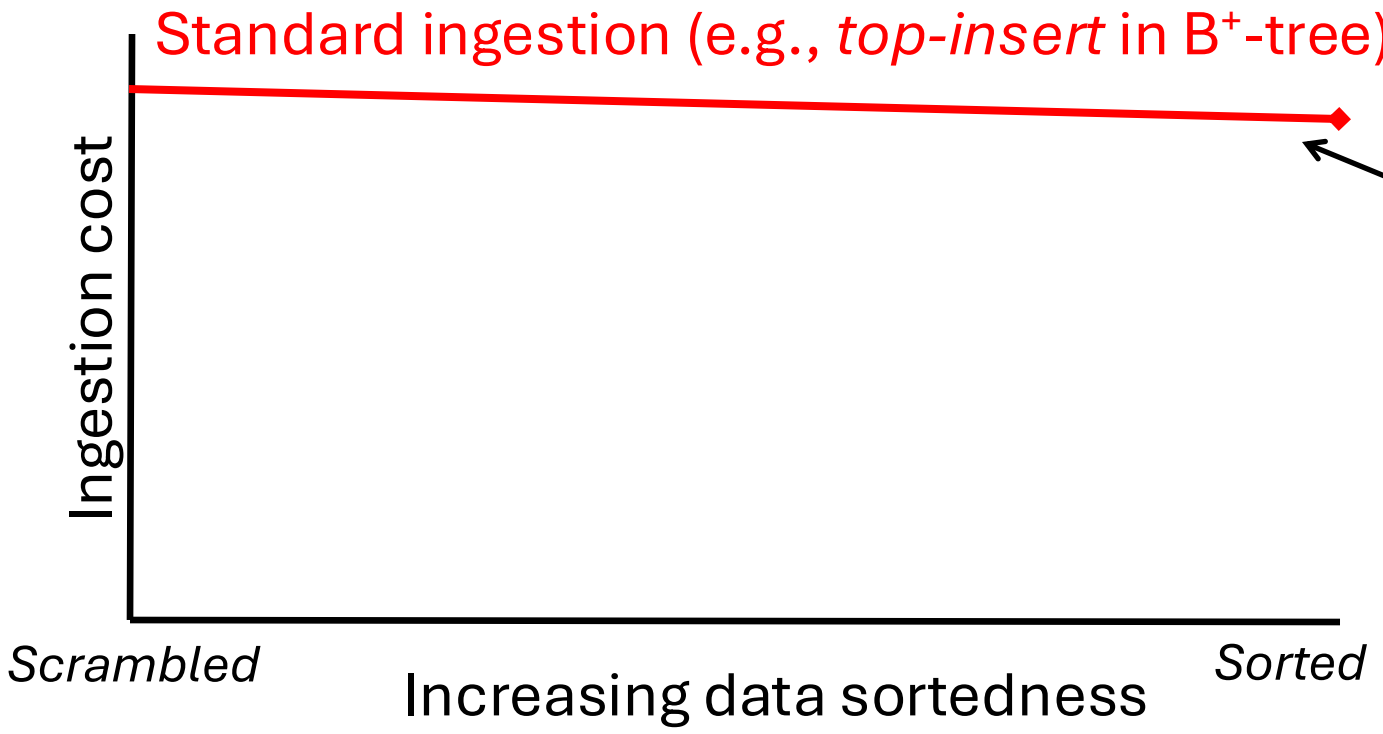


near-sorted data

Irrespective of Sortedness, Same Performance



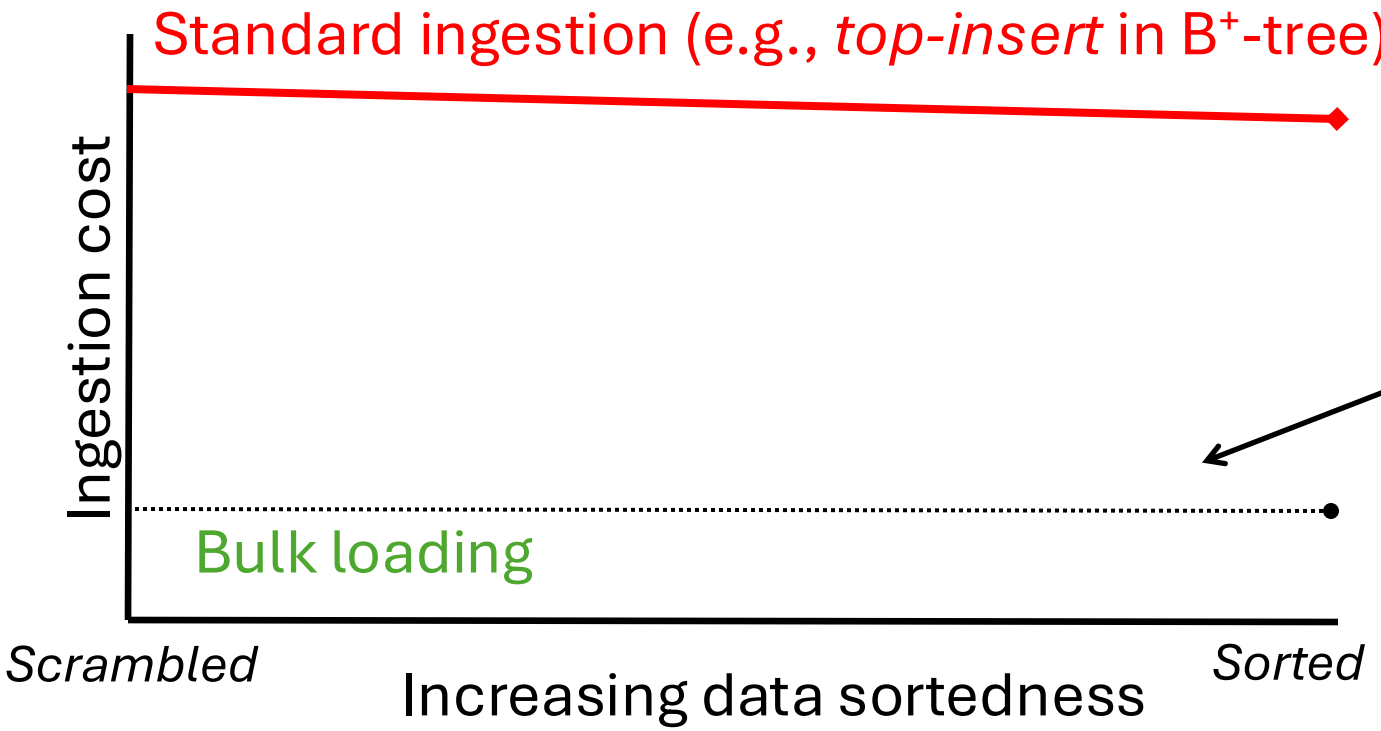
Irrespective of Sortedness, Same Performance



locality gives some benefits

by design, **cannot exploit** inherent order

Are There Faster Alternatives?

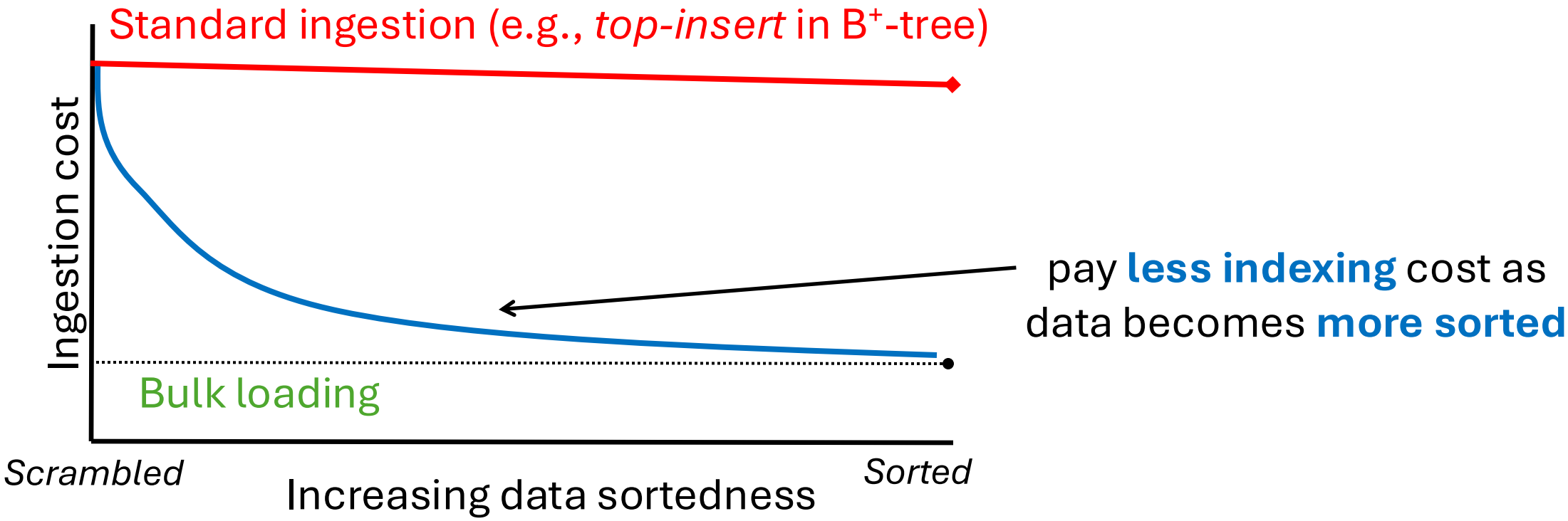


by design, **cannot exploit** inherent order

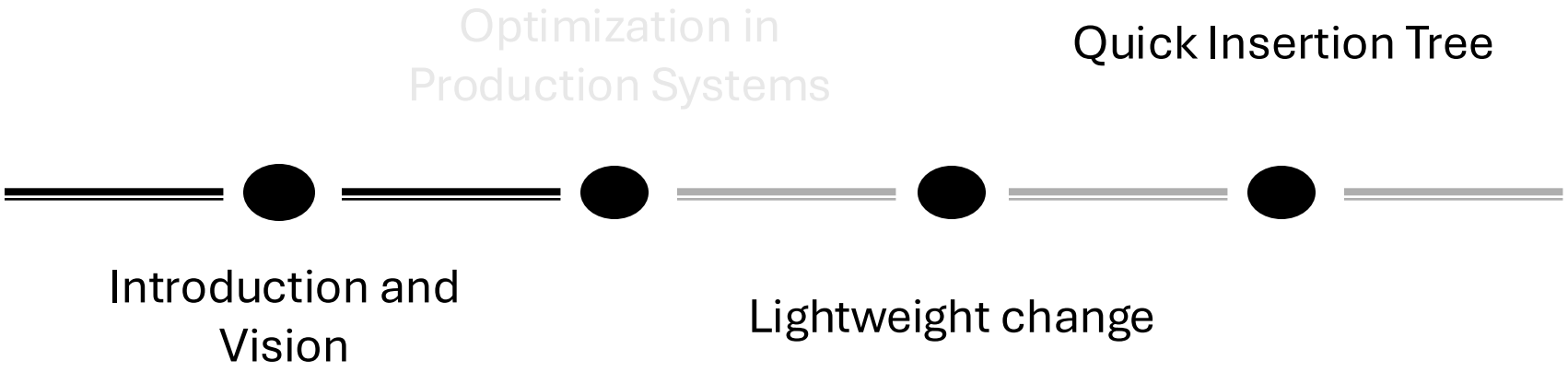
builds the index bottom-up

requires all data ***a priori***

Ideally, Higher Sortedness => Faster Ingestion



Agenda

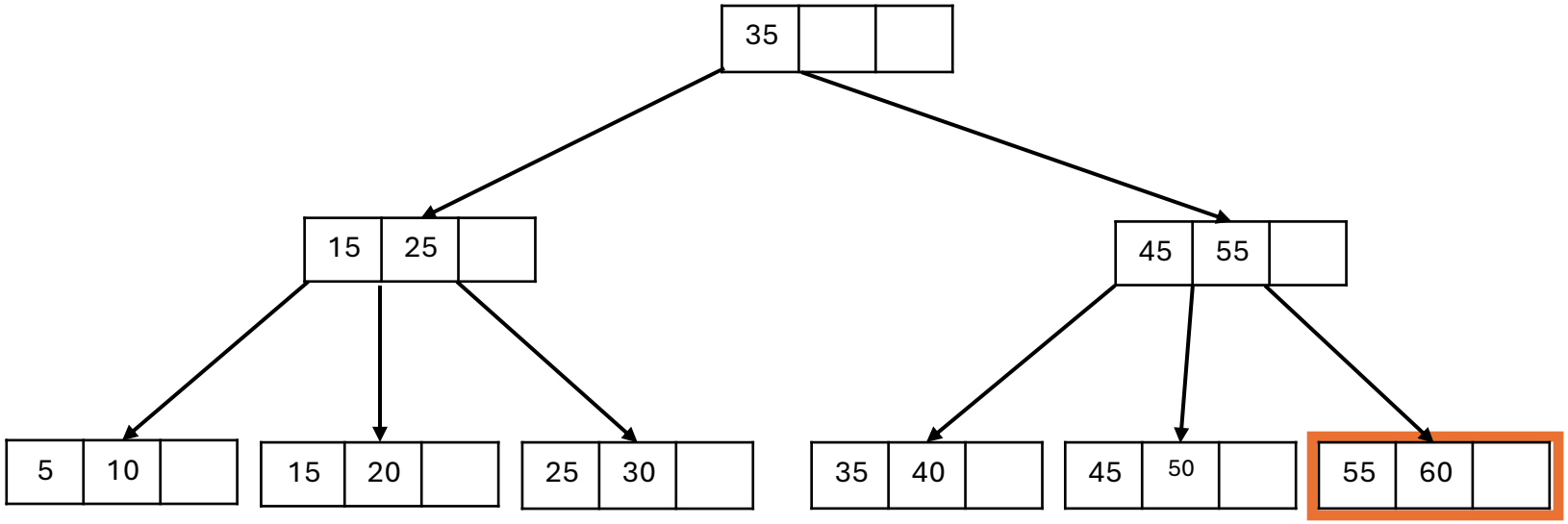

Optimization in Production Systems
Quick Insertion Tree
Introduction and Vision
Lightweight change

Inserting To the Tail Leaf

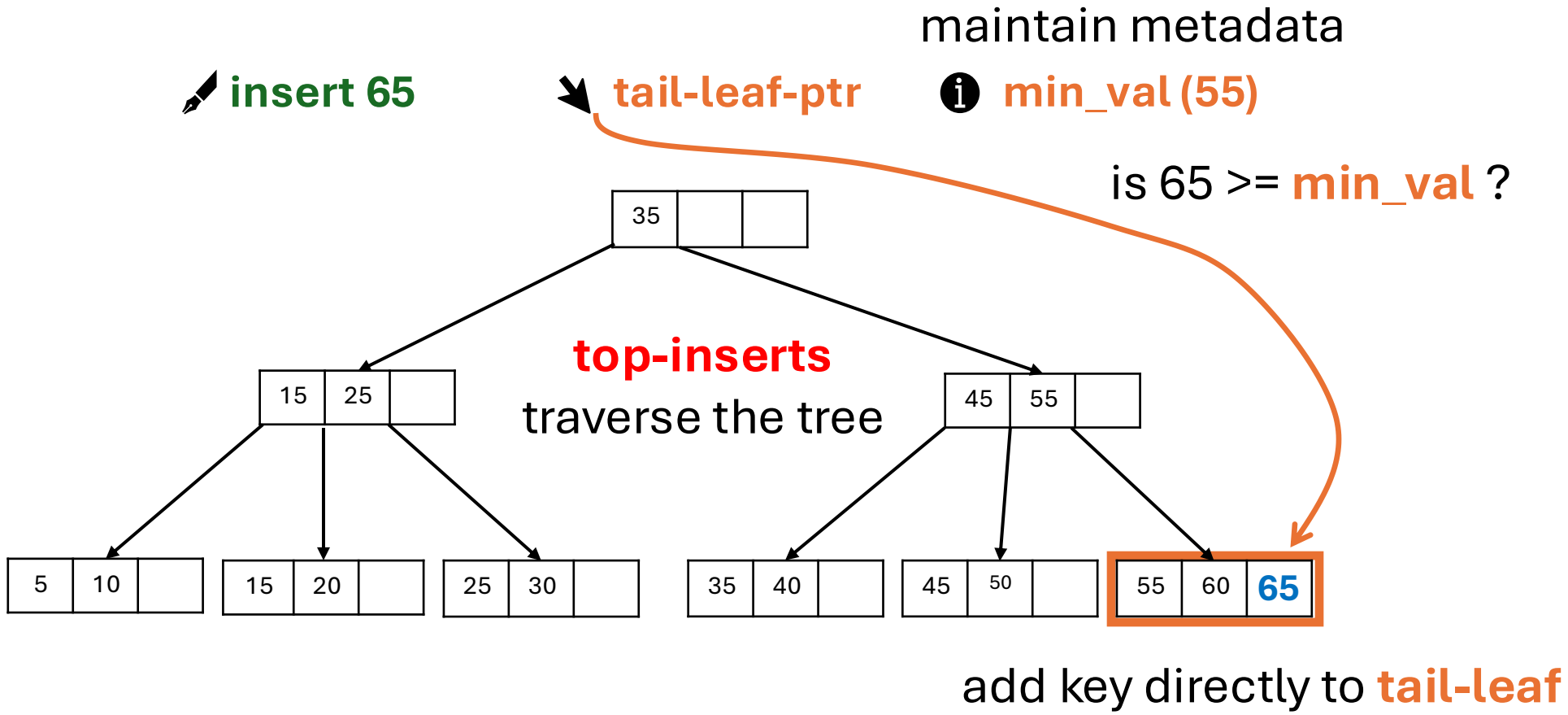
maintain metadata

tail-leaf-ptr

min_val (55)



Inserting To the Tail Leaf



Inserting To the Tail Leaf

maintain metadata

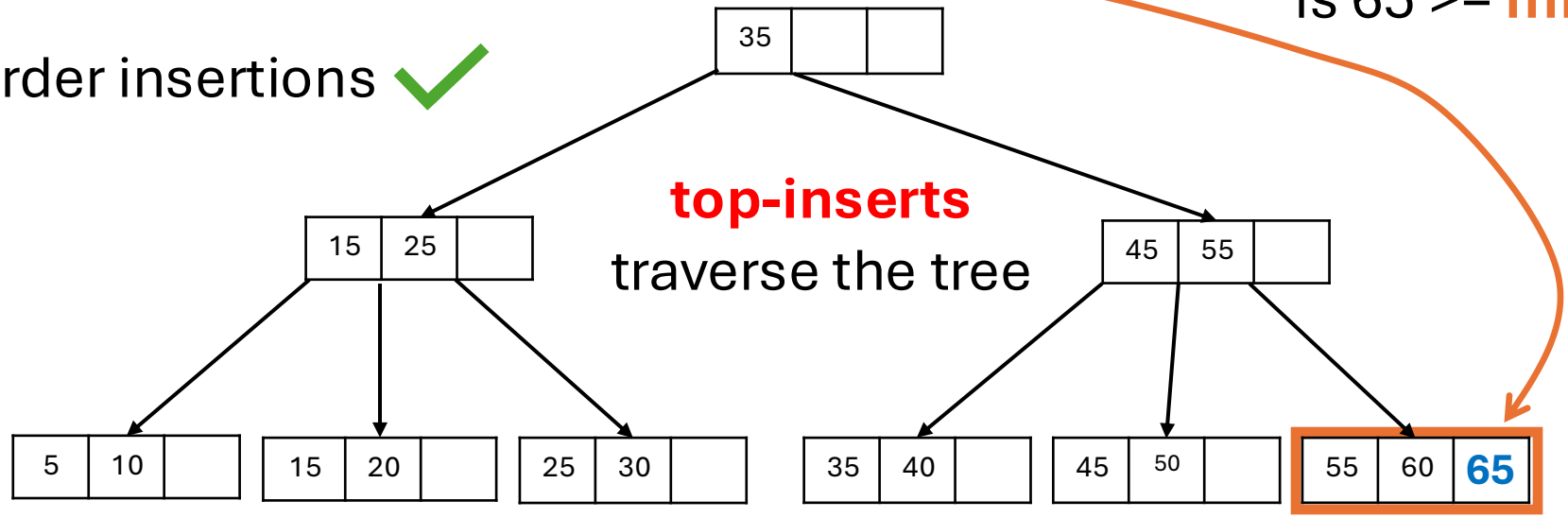
insert 65

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 work for near-sorted data?

Does This Always Work?

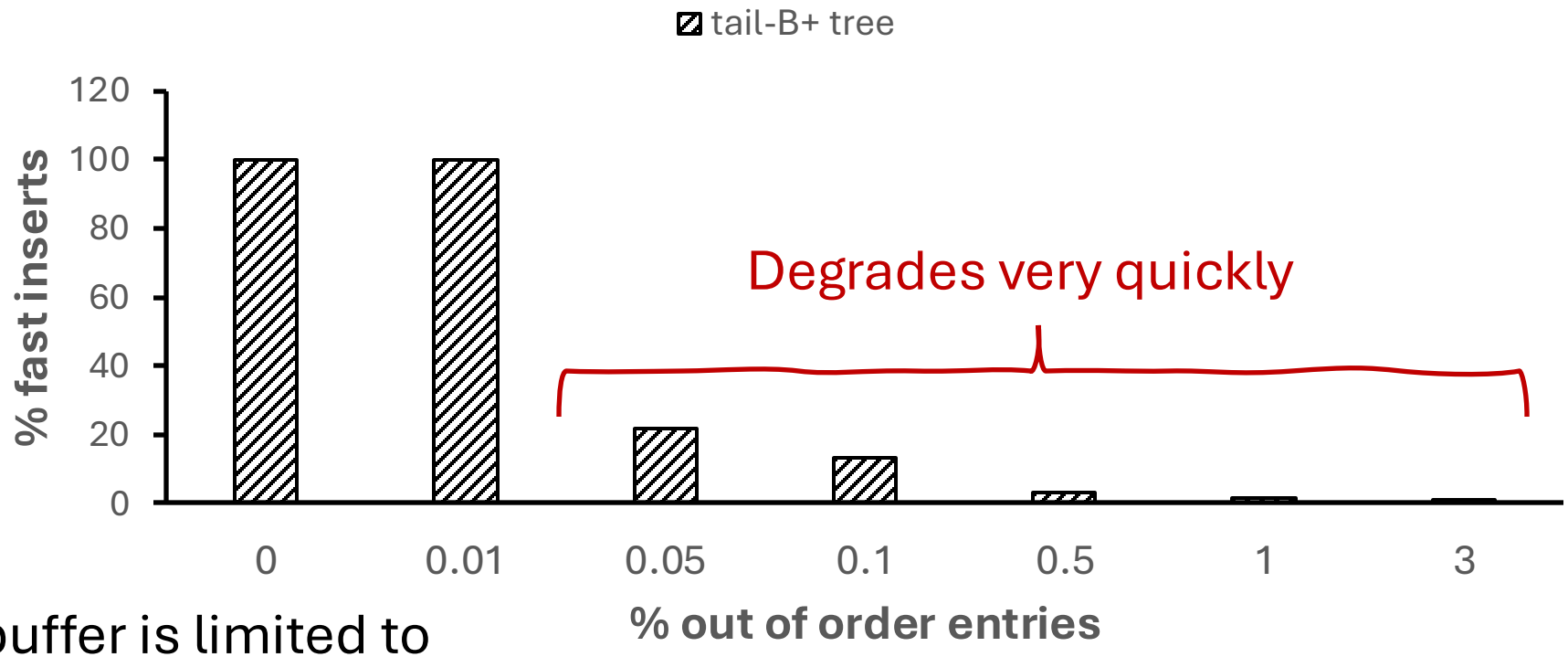
Works for fully and very highly sorted data

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

outliers

5	6	7	**20**	**21**

Does This Always Work?



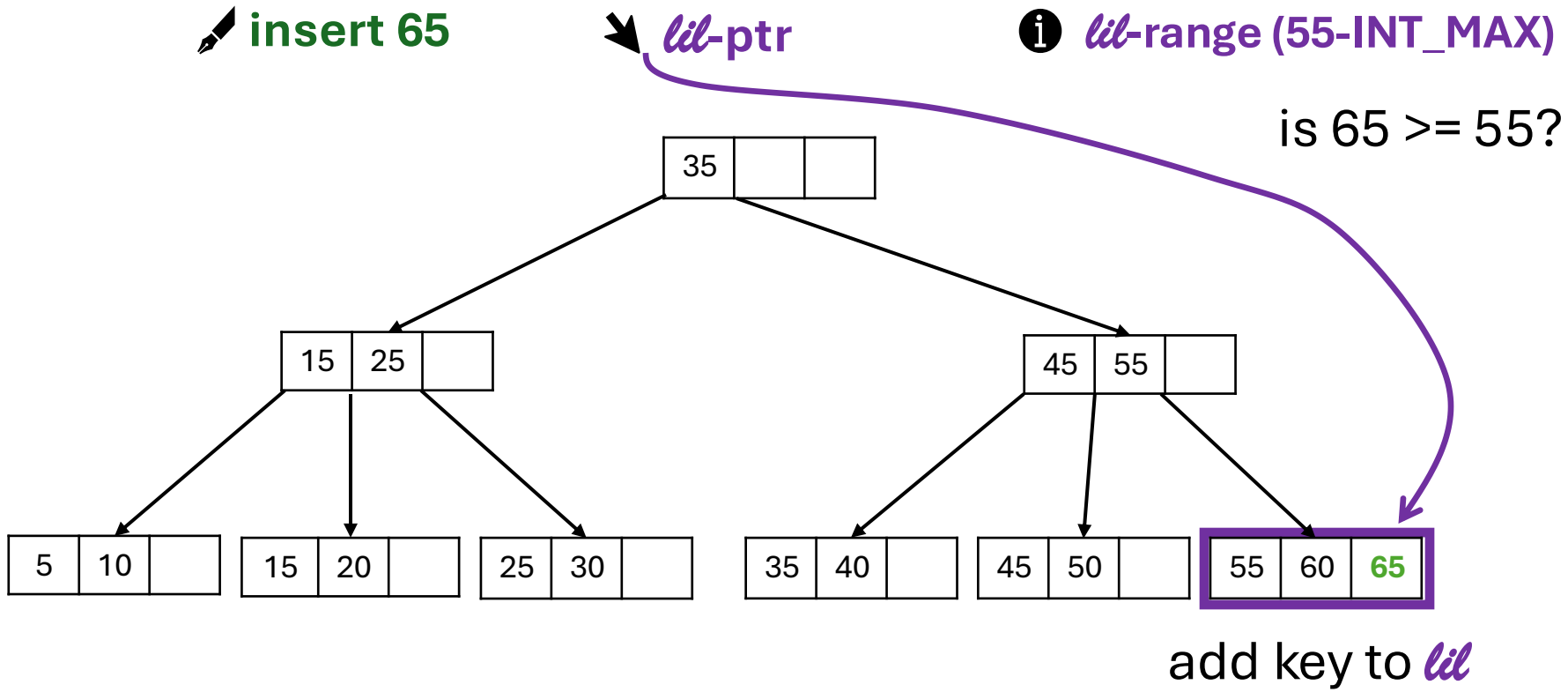
Tail-leaf's buffer is limited to leaf node!

outliers

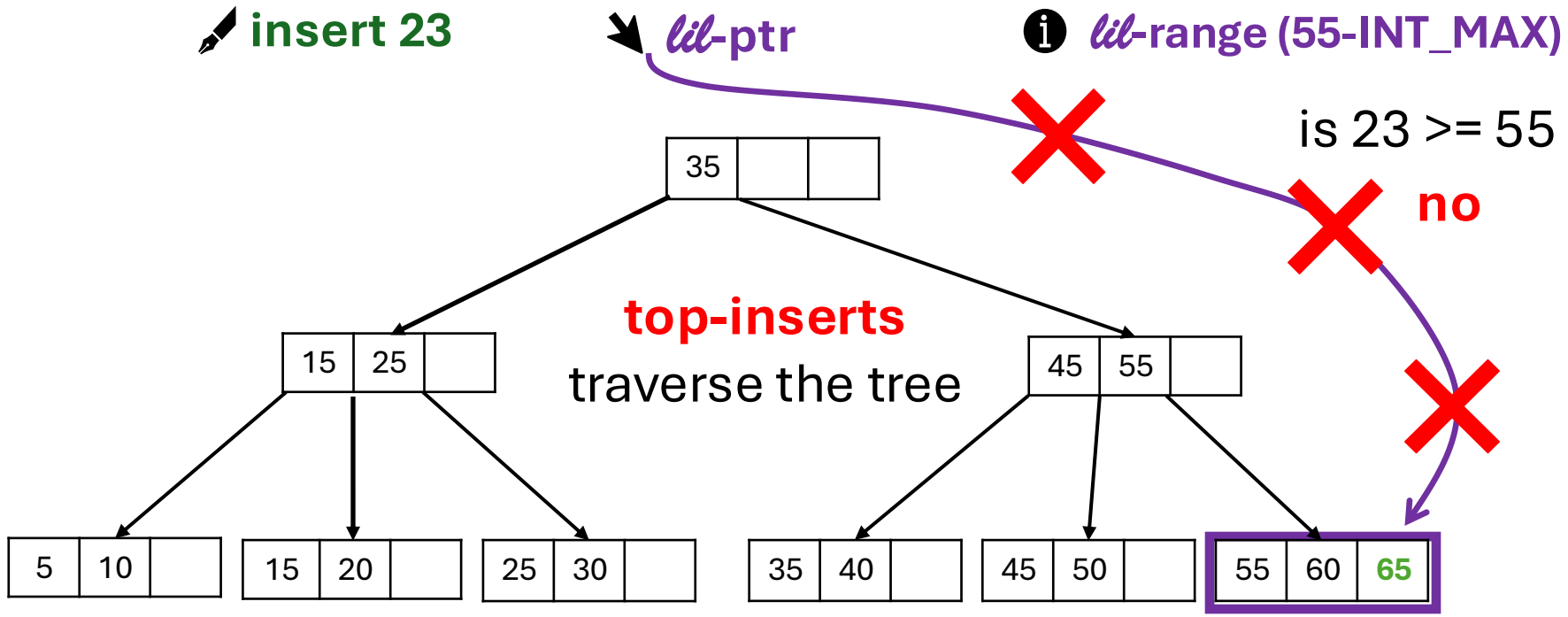
5	6	7	20	21

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

Following the Last Insertion Leaf (*lil*)…

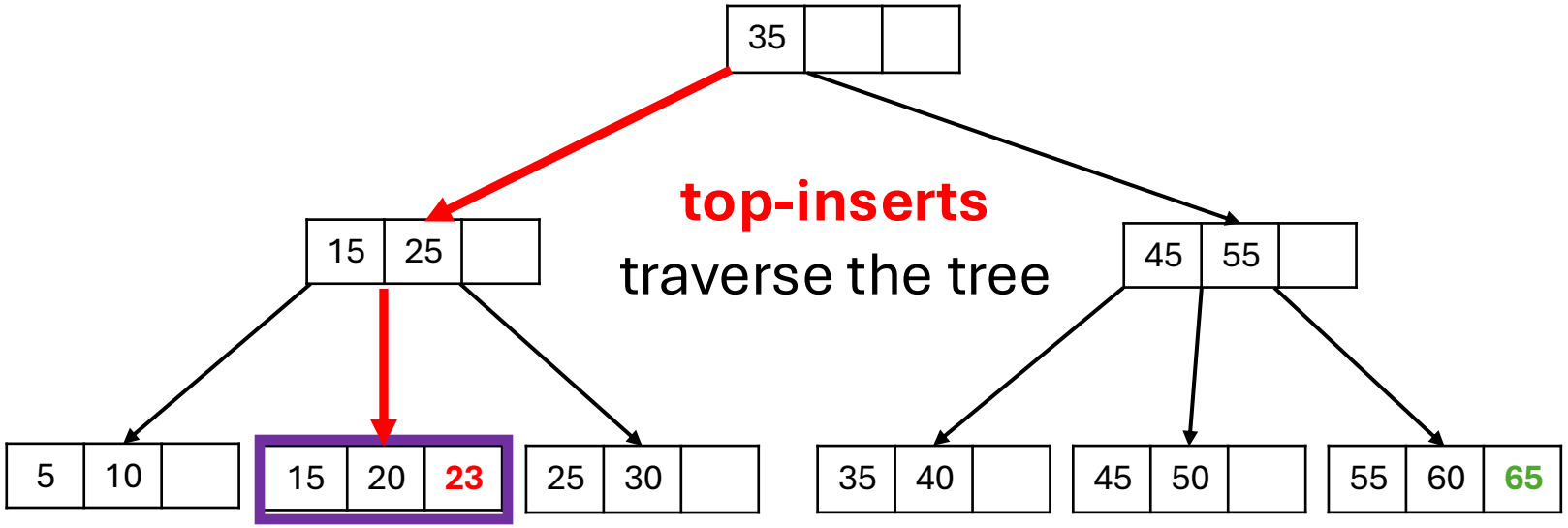


Following the Last Insertion Leaf (*lil*)…



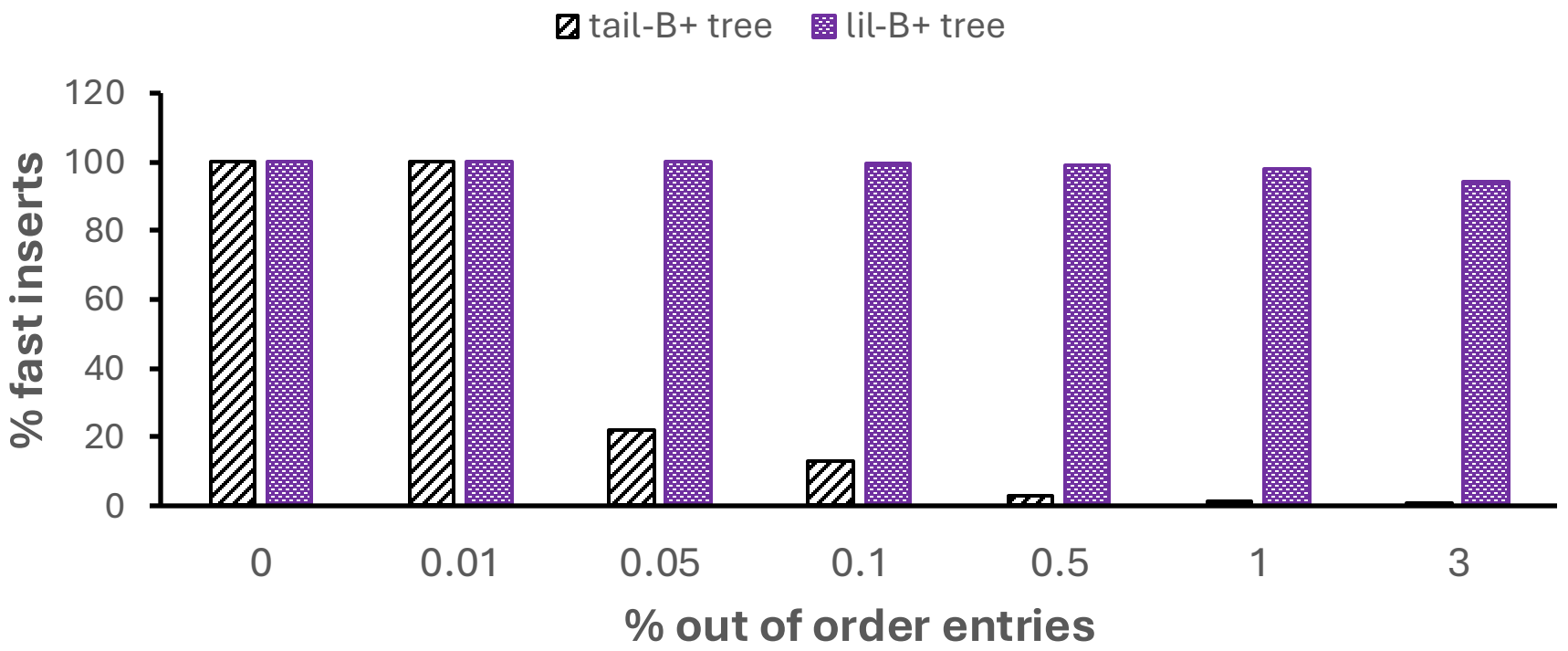
Following the Last Insertion Leaf (*lil*)…

insert 23 | *lil*-ptr | *lil*-range (15-23)



however, *lil* is also updated!

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 with every missed fast-insert!

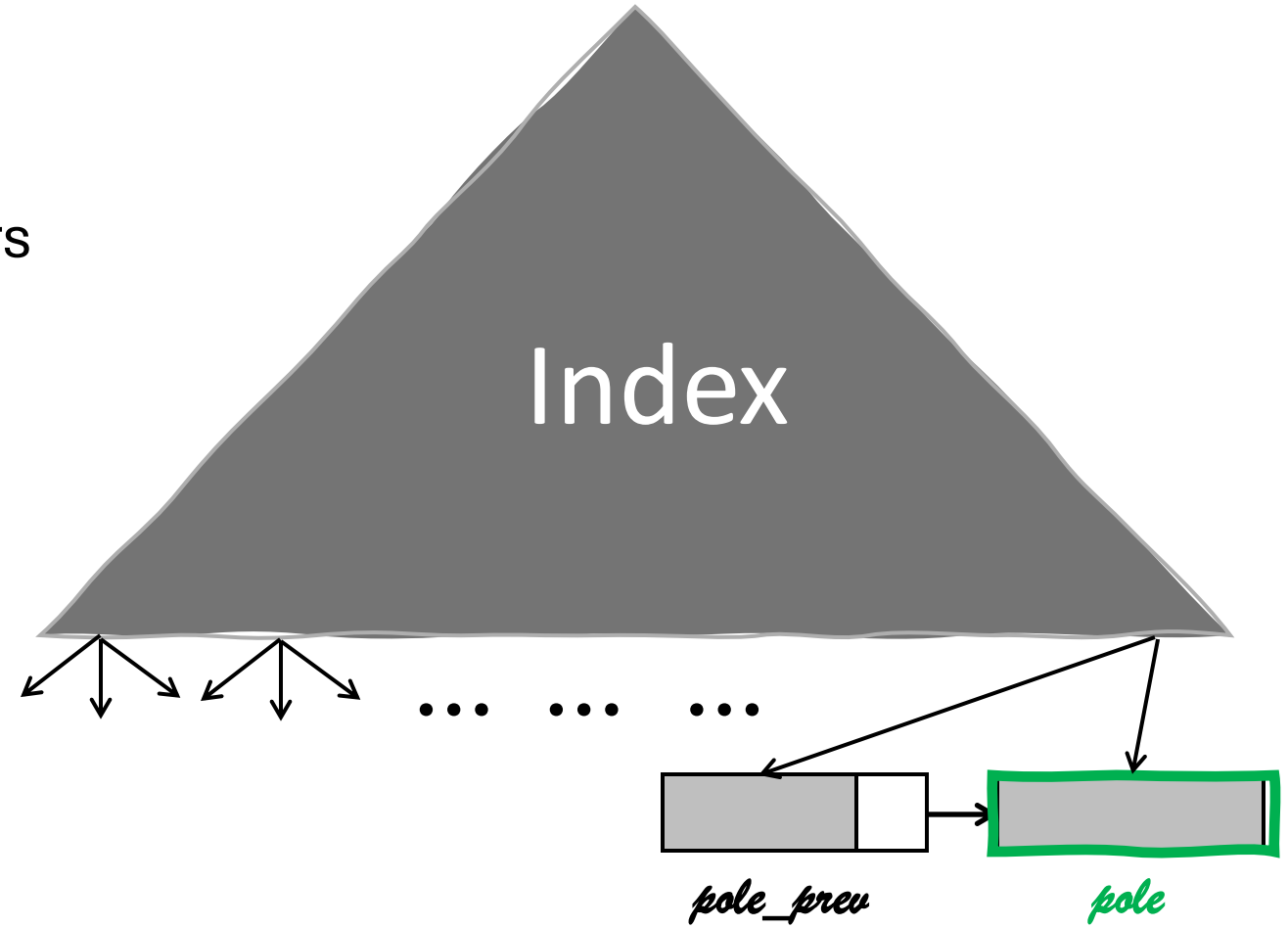
Ideally, we should incur ***at most*** one **top-insert** for every **out-of-order** entry

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** ?

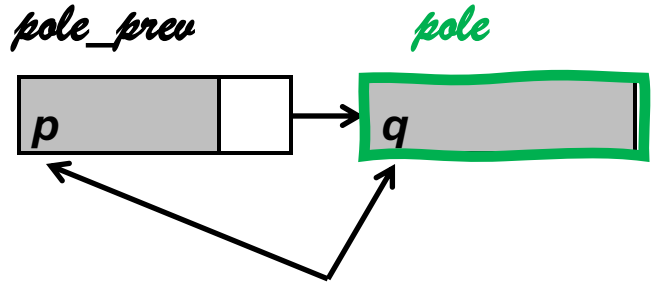


Predicting the Ordered Leaf (*pole*)

tail-leaf quickly fills up with outliers

lil naively switches fast-path

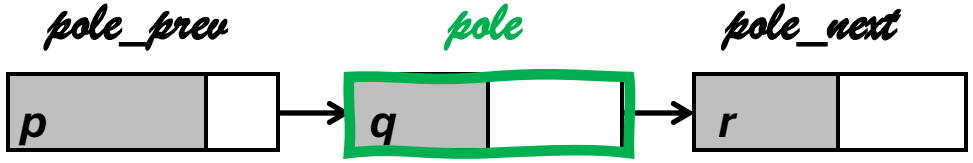
decision: when do we update the **fast-path** ?



necessarily in-order keys

Predicting the Ordered Leaf (*pole*)

Should we move *pole*?



tail-leaf quickly fills up with outliers

lil naively switches fast-path

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

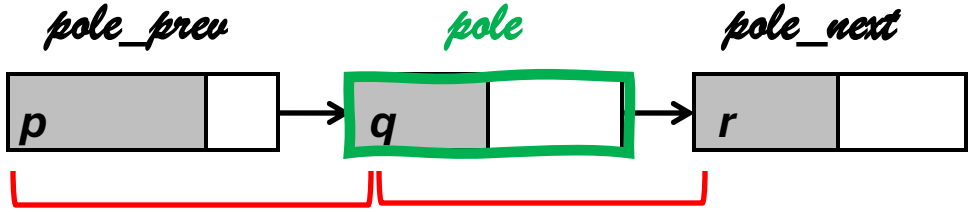
Predicting the Ordered Leaf (*pole*)

tail-leaf quickly fills up with outliers

lil naively switches fast-path

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

Compare the density



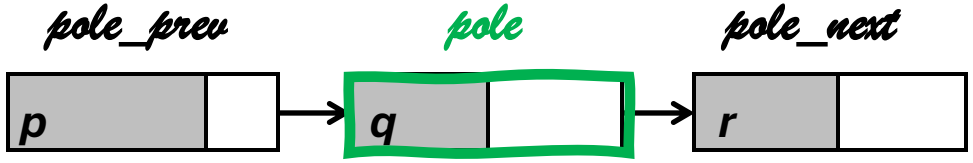
predict using *In-order Key estimatoR* (*IKR*)

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

density between two non-outliers

Predicting the Ordered Leaf (*pole*)

tail-leaf quickly fills up with outliers



lil naively switches fast-path

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

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

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

capture small deviations

r is an **outlier**

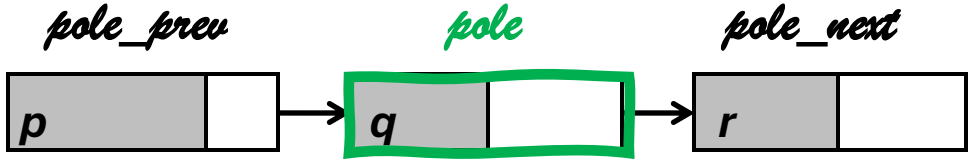
density between two non-outliers

Can We Better Utilize Space?

high sortedness => poor space utilization

can we find better split points?

IKR can also return the split point

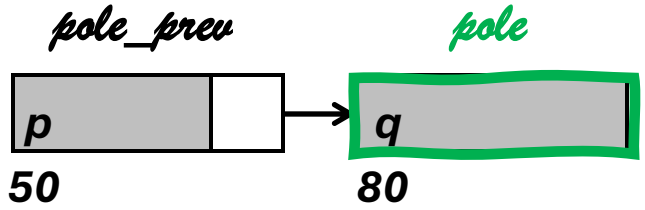


Can We Better Utilize Space?

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;
}
```



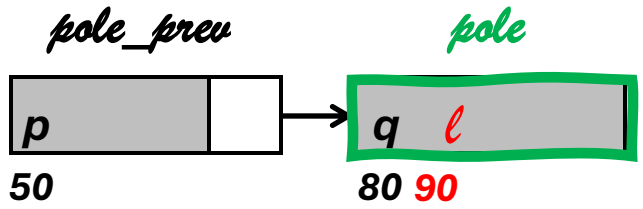
let's call this key l and its position l_pos

Can We Better Utilize Space?

high sortedness => poor space utilization

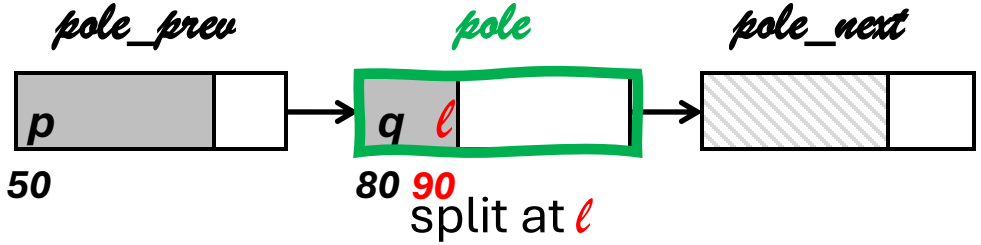
```
l, l_pos = IKR(q, p, pole_prev.size);

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 $\ell\_pos \leq 50\%$

outliers dominate *pole*



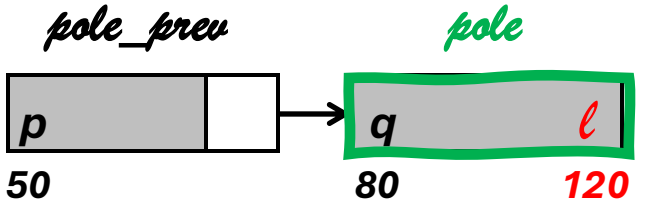
moves all outliers to *pole_next*

Can We Better Utilize Space?

high sortedness => poor space utilization

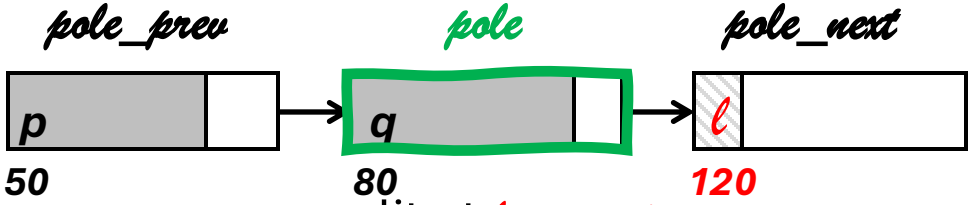
```
l_pos = IKR(q, p, pole_prev.size);

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



if $l\_pos > 50\%$

pole has few outliers



split at $l\_pos - 1$

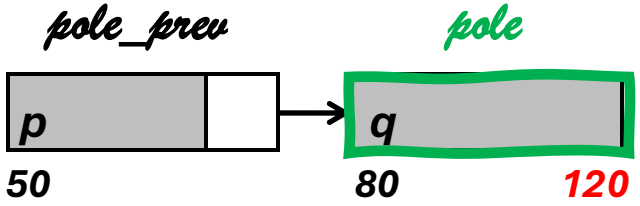
moves at least one non-outlier to *pole_next*

Can We Better Utilize Space?

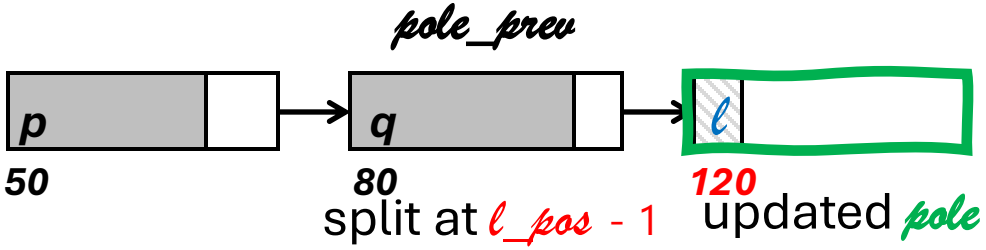
high sortedness => poor space utilization

```
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;
}
```



pole has few outliers



leaves more space in *pole*

Quick Insertion Tree

similar to **B⁺-tree** in design

+

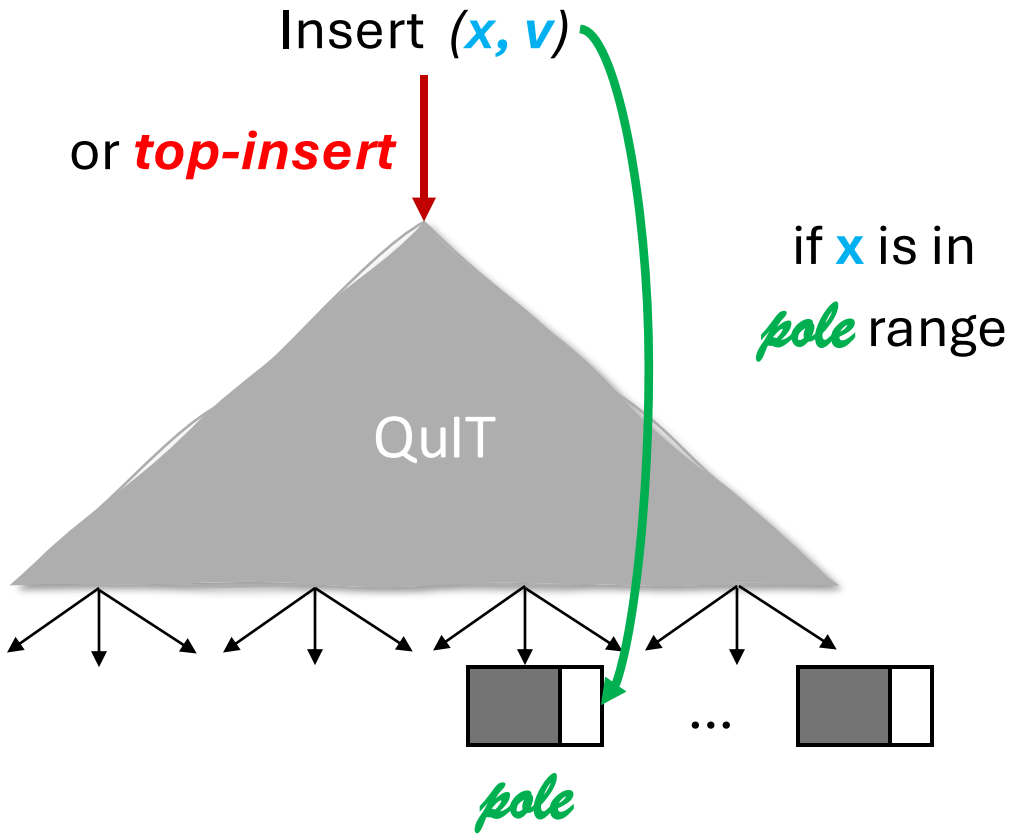
offers **fast-path** ingestion

+

sortedness-aware

+

minimal metadata + ***tuning***



Evaluating QuIT

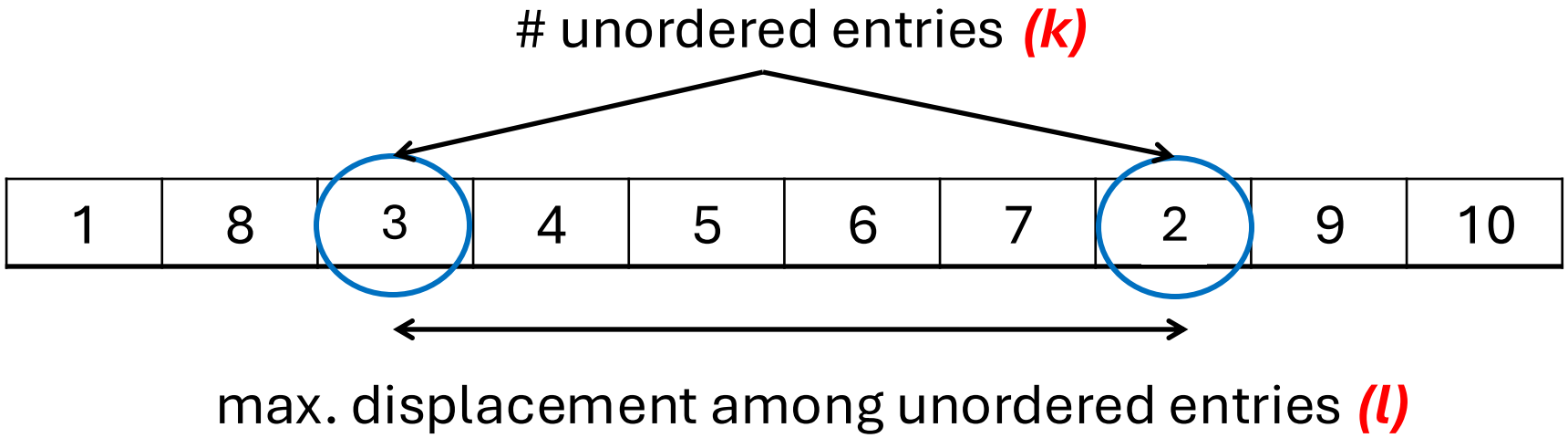
System:

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

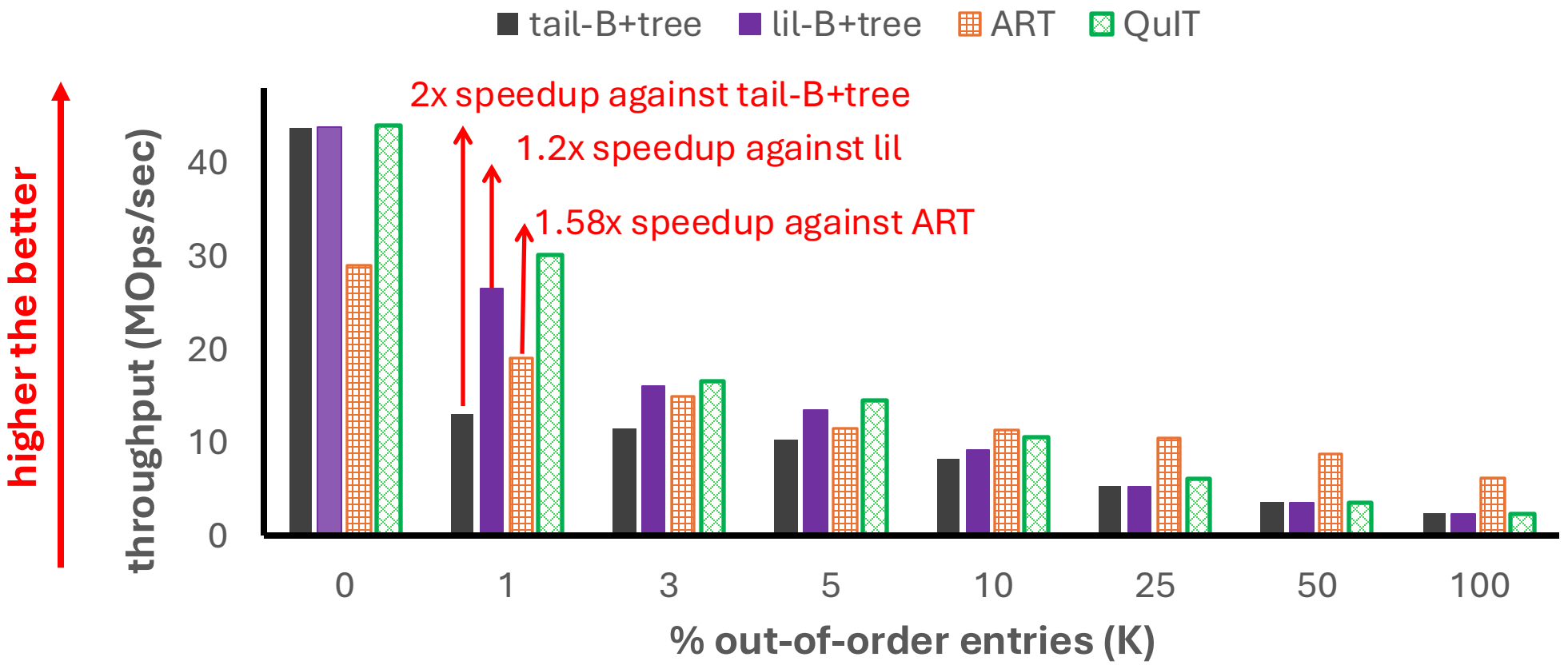
Index Setup:

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

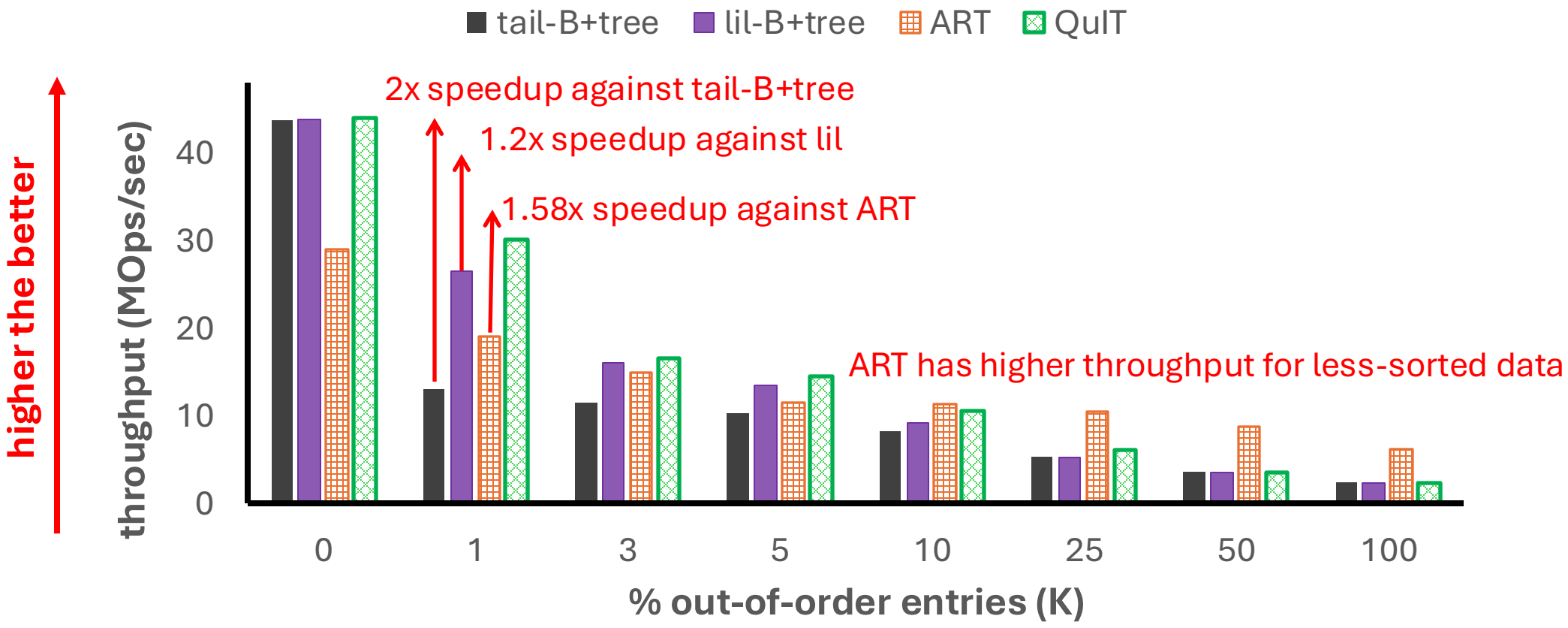
(k-l) Sortedness Metric



QuIT Performs Best With Near-Sorted Data

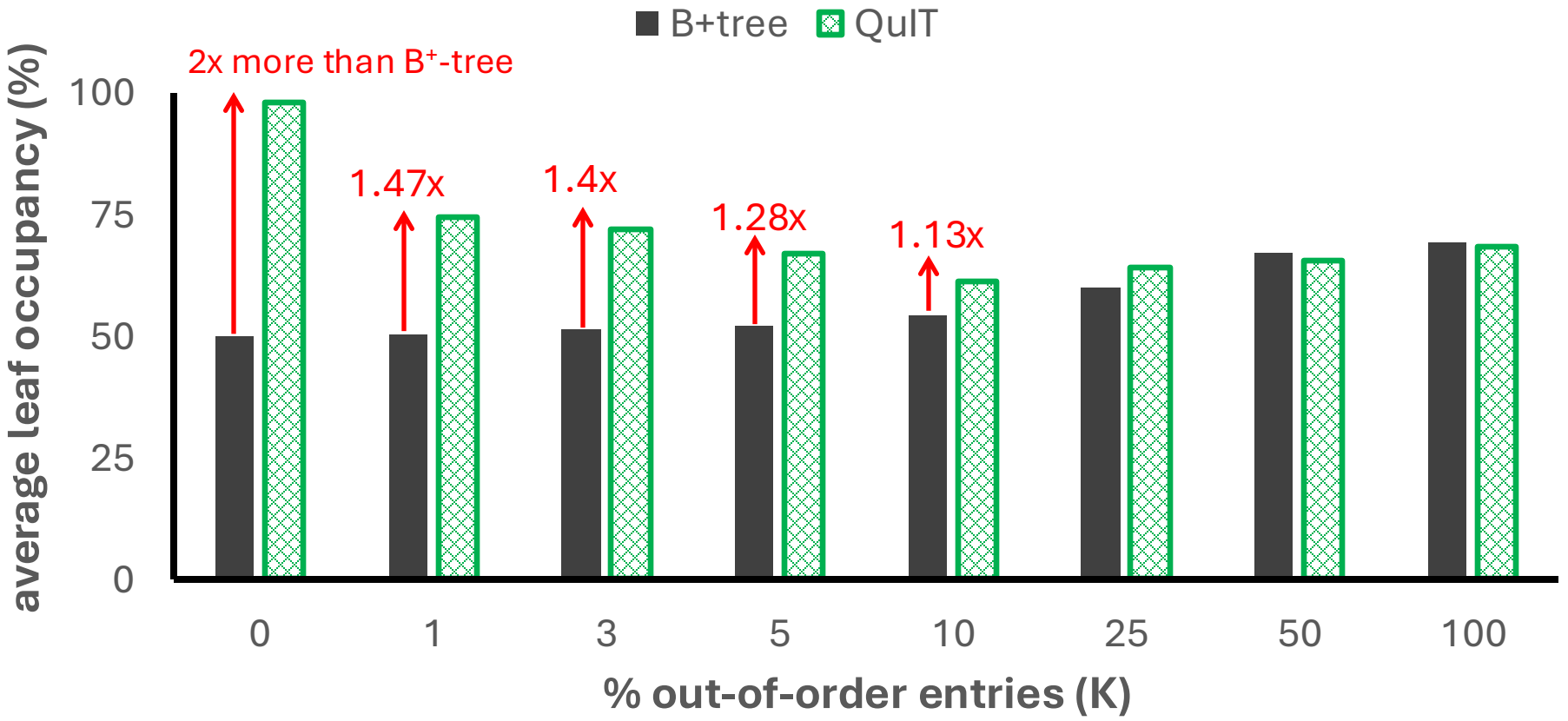


QuIT Performs Best With Near-Sorted Data



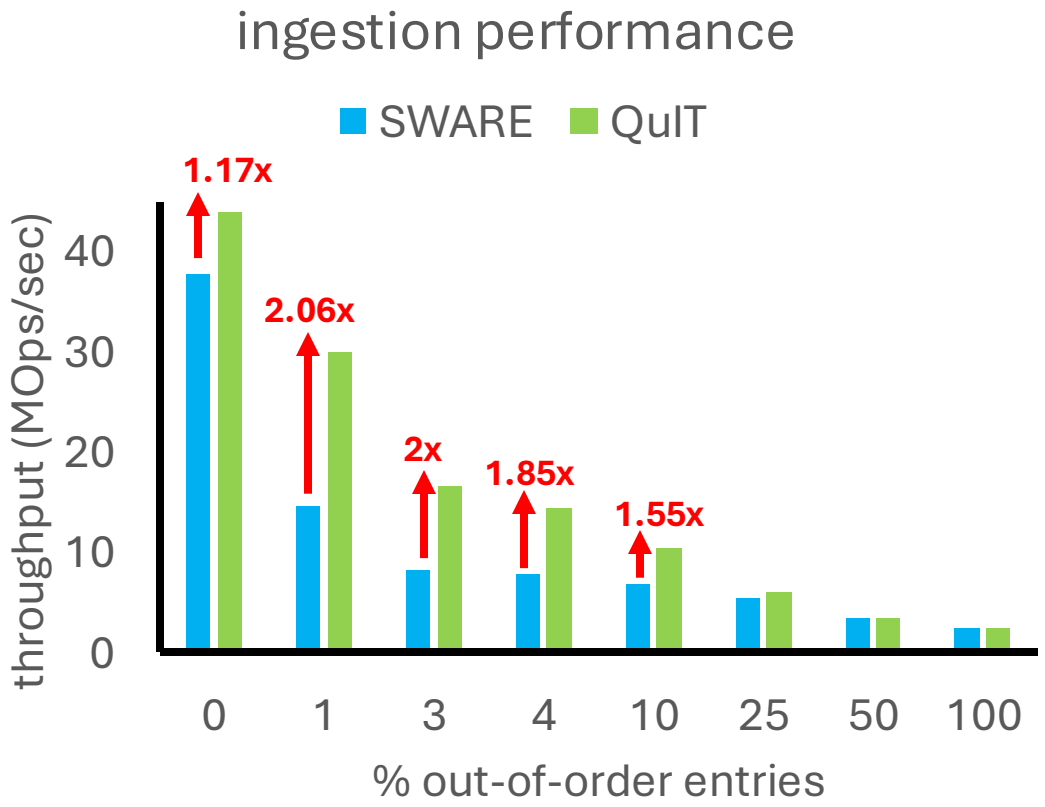
QuIT is Space Efficient

QuIT is Space Efficient



higher leaf occupancy reduces memory footprint✓

QuIT v/s SWARE



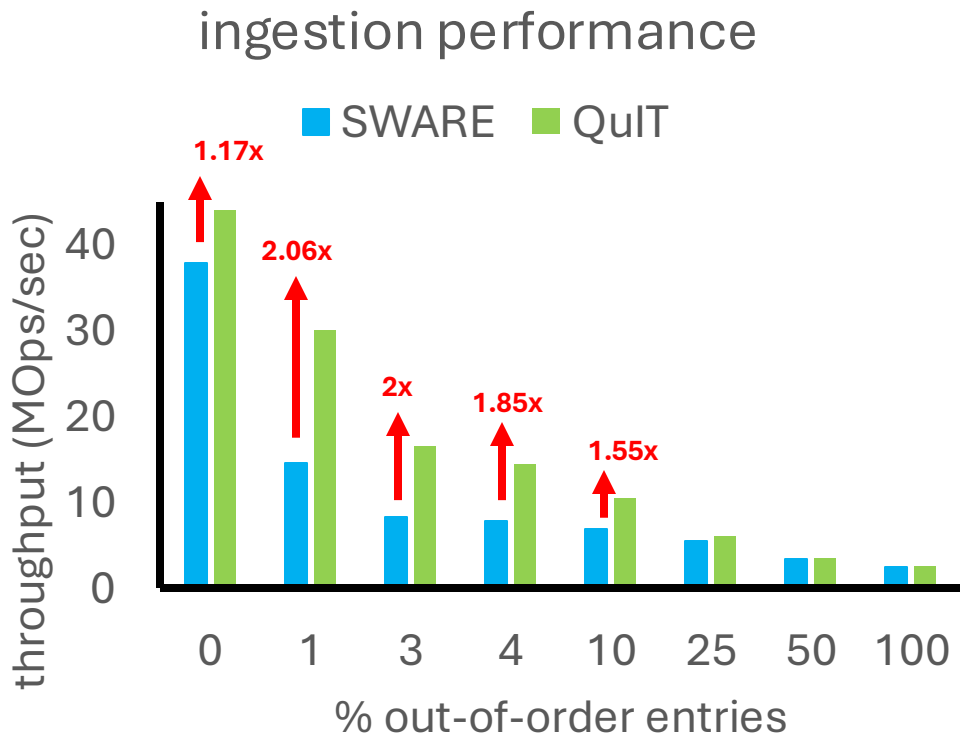
integrate SWARE with same B⁺-tree as QuIT

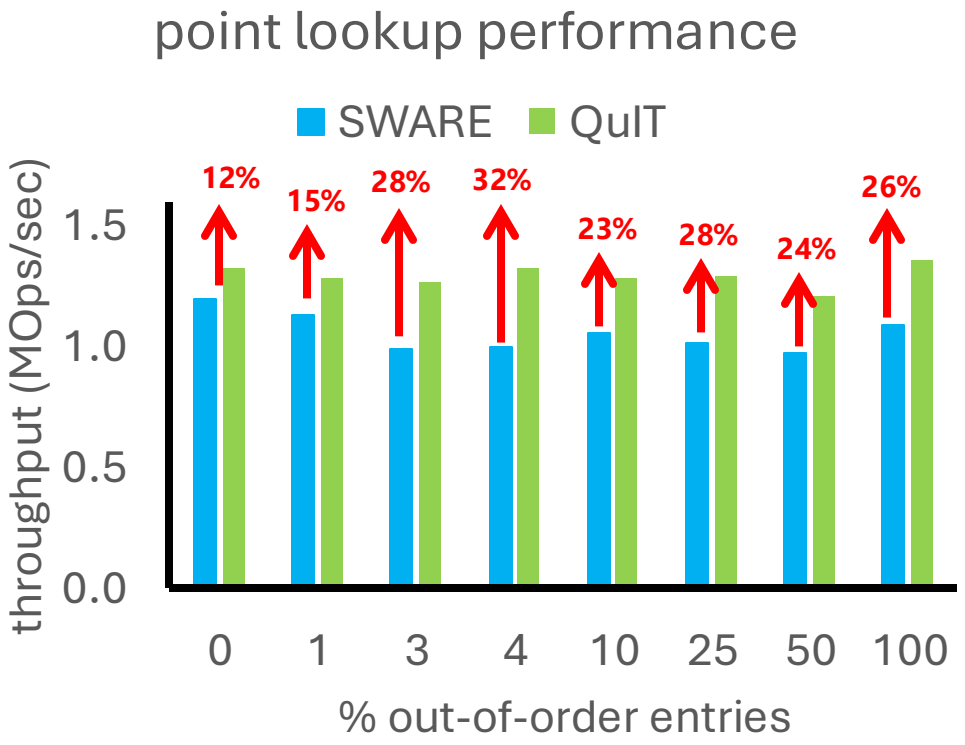
up to 2.06x faster

avoids buffer management ✓

minimal metadata ✓

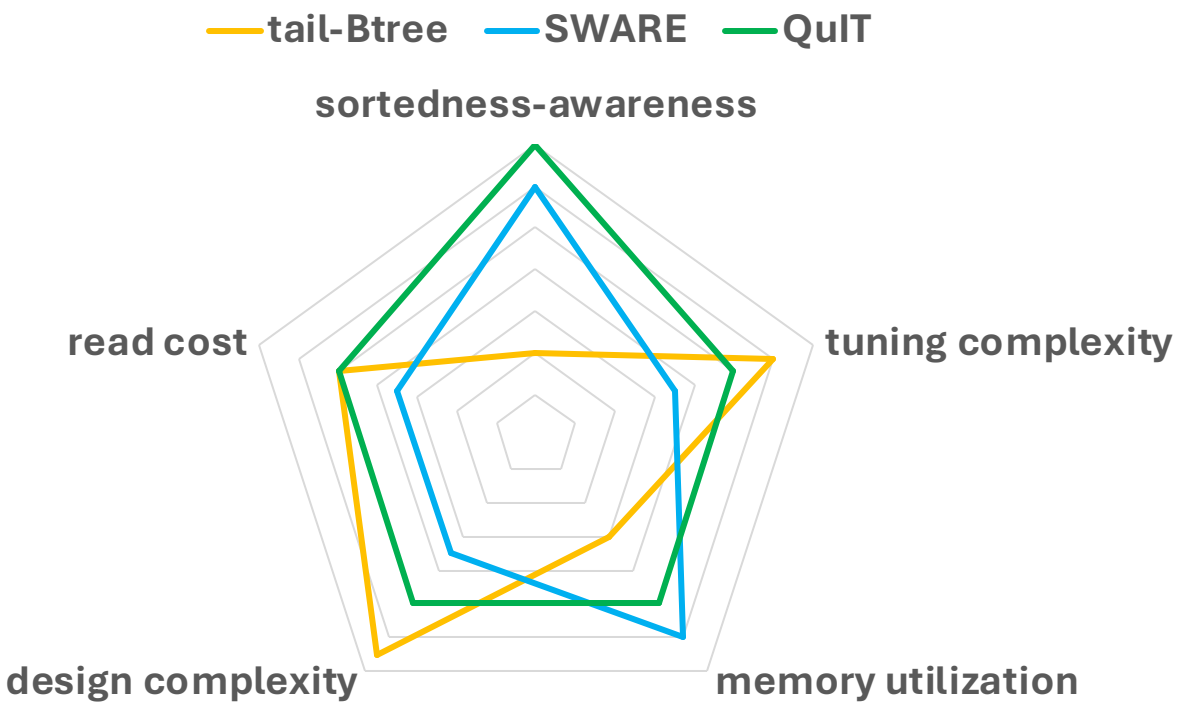
QuIT v/s SWARE





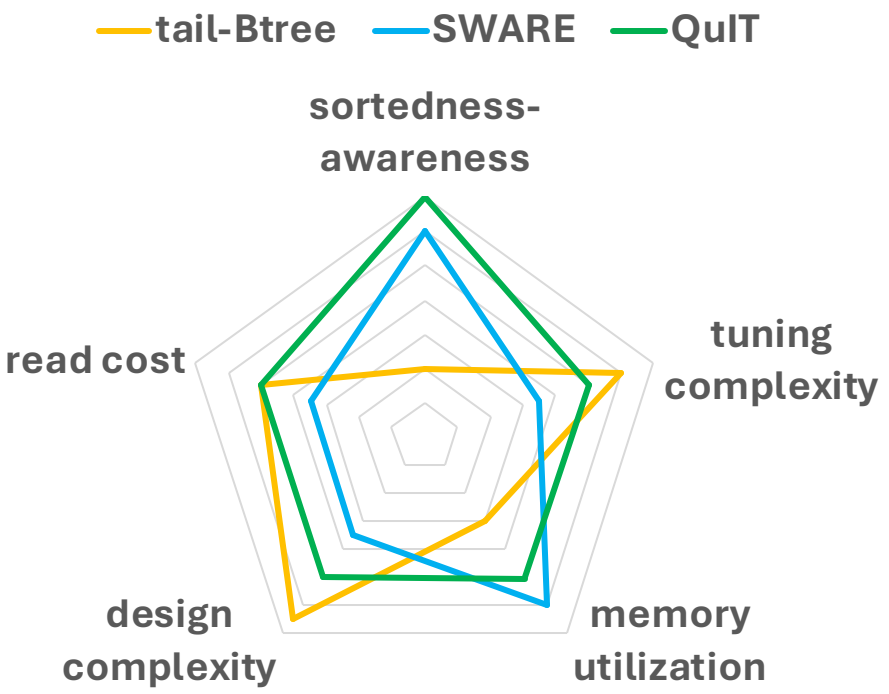
No buffering ⇒ no read overhead!

Summary



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

Summary

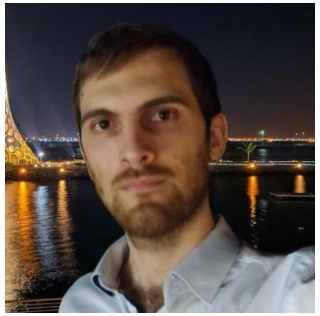


QuIT offers: **higher** sortedness-awareness + **no** read penalty + **minimal** design & tuning complexity

Our Team

find us if you have questions!

Aneesh Raman



Kostas Karatsenidis

Shaolin Xie

Subhadeep Sarkar

Matthaios Olma

Manos Athanassoulis