Indexing for Near-Sorted Data

Aneesh Raman

Subhadeep Sarkar

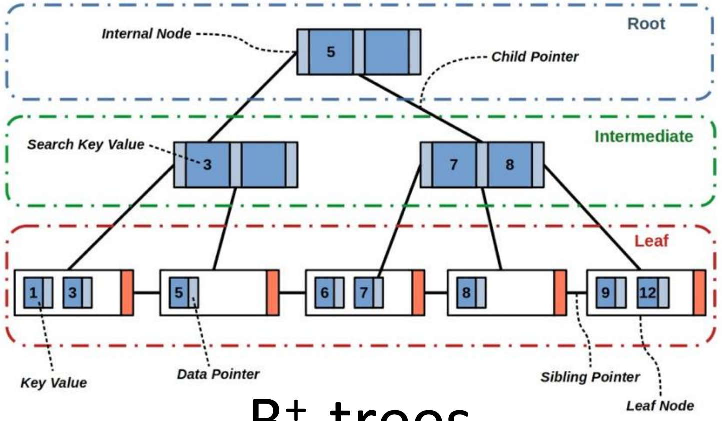
Matthaios Olma

Manos Athanassoulis

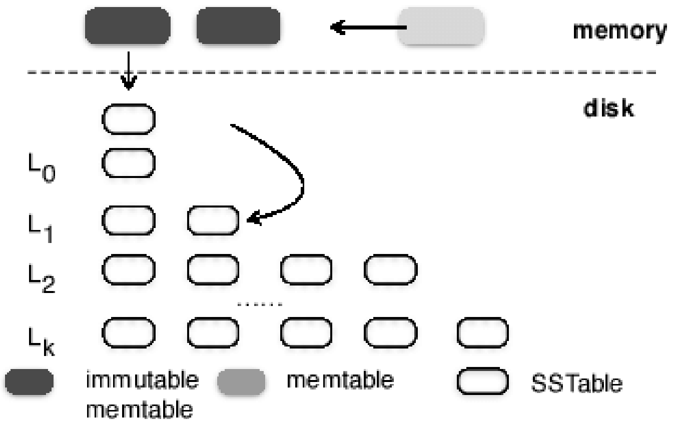
BOSTON UNIVERSITY

DiSC lab

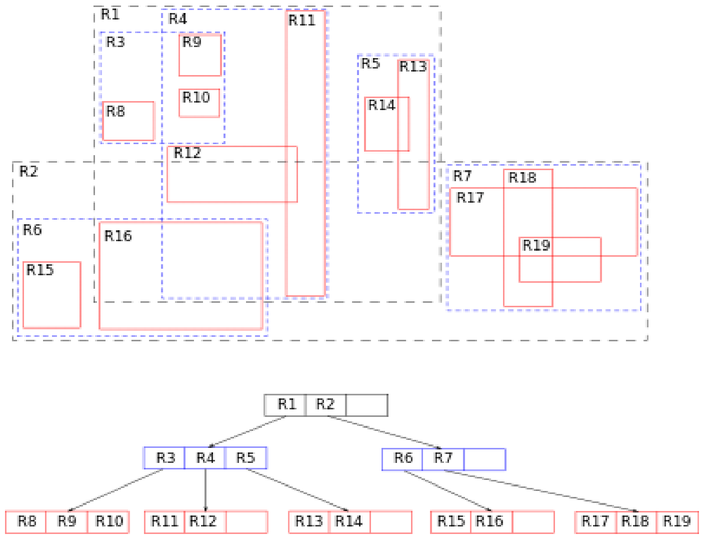
Indexes in Databases



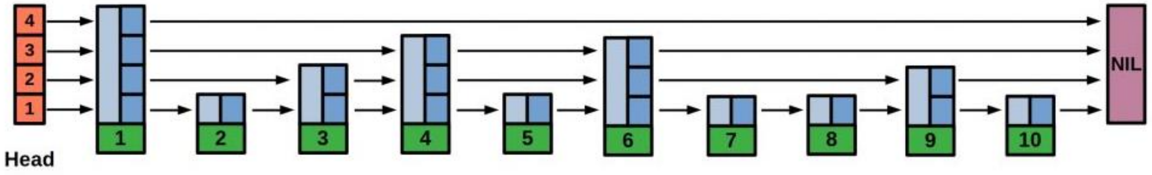
B+-trees



LSM-trees



R-trees



Skip lists

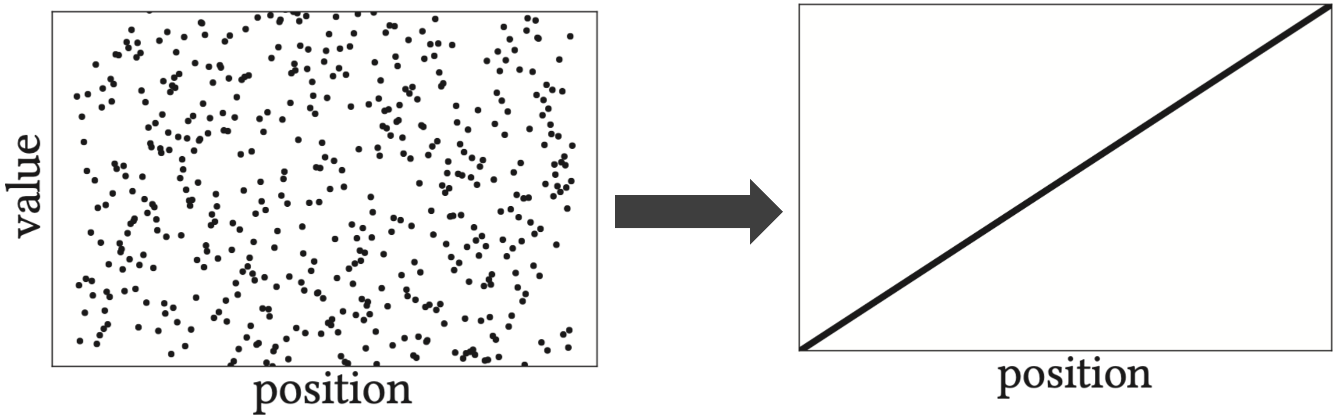
BOSTON UNIVERSITY

DiSC lab

Indexes in Databases

organize
data

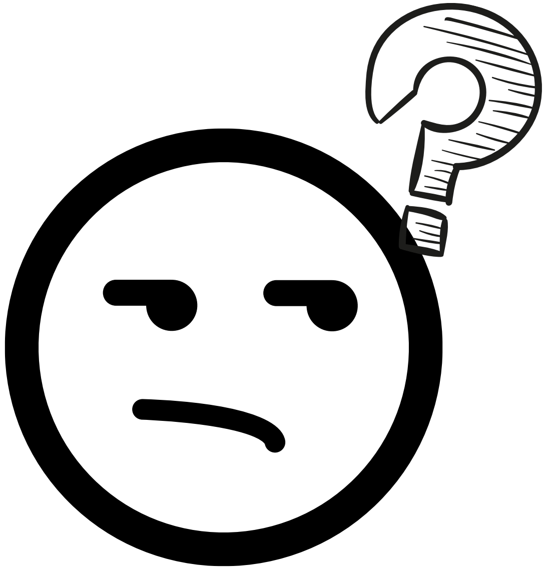
efficient
queries



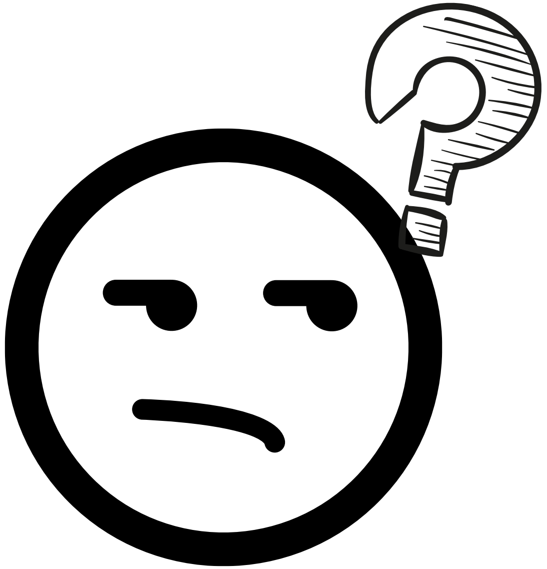
unstructured
data

structured
data

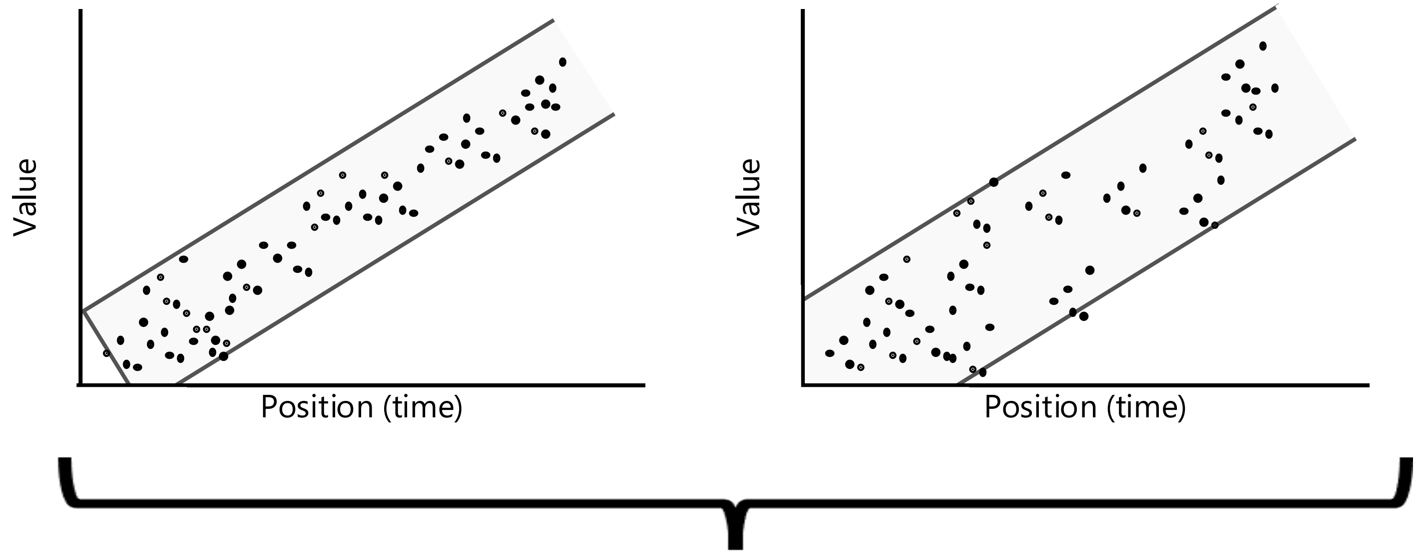
The process of inducing “*sortedness*” to an otherwise unsorted data collection



What if data already has some structure?



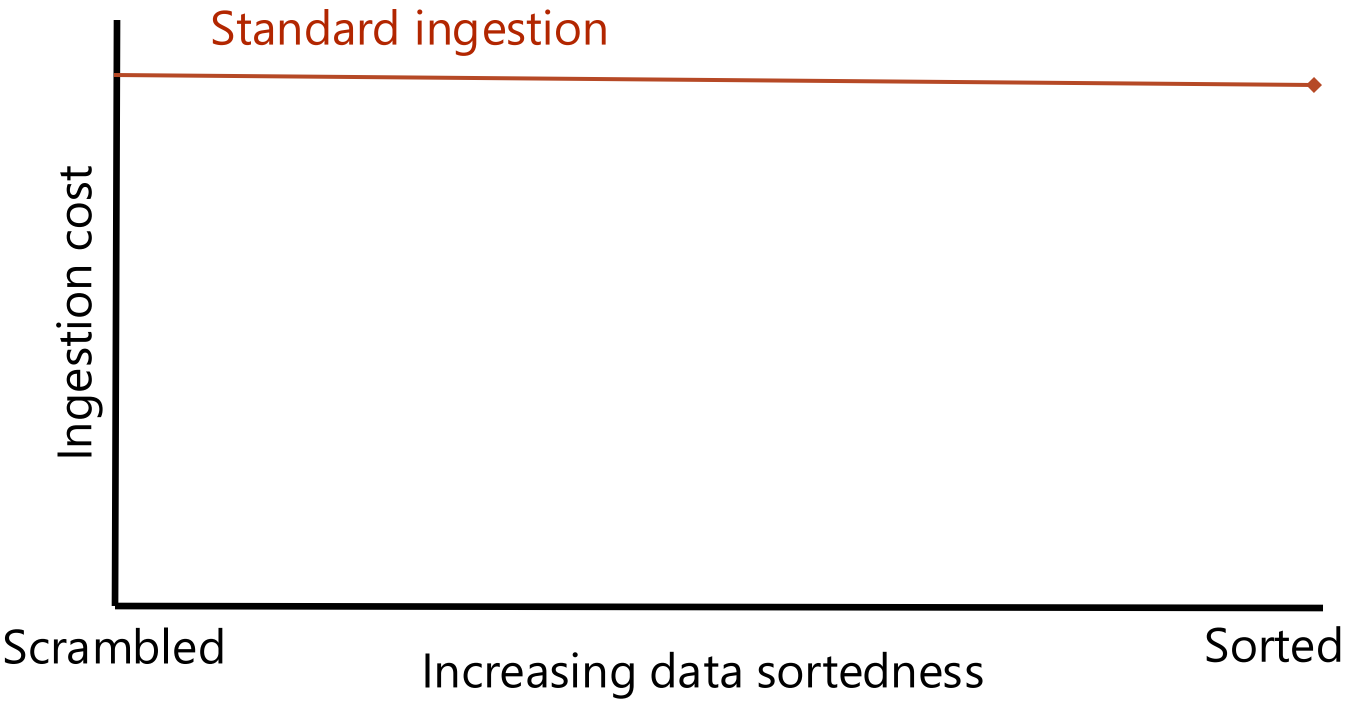
What if data already has some structure?



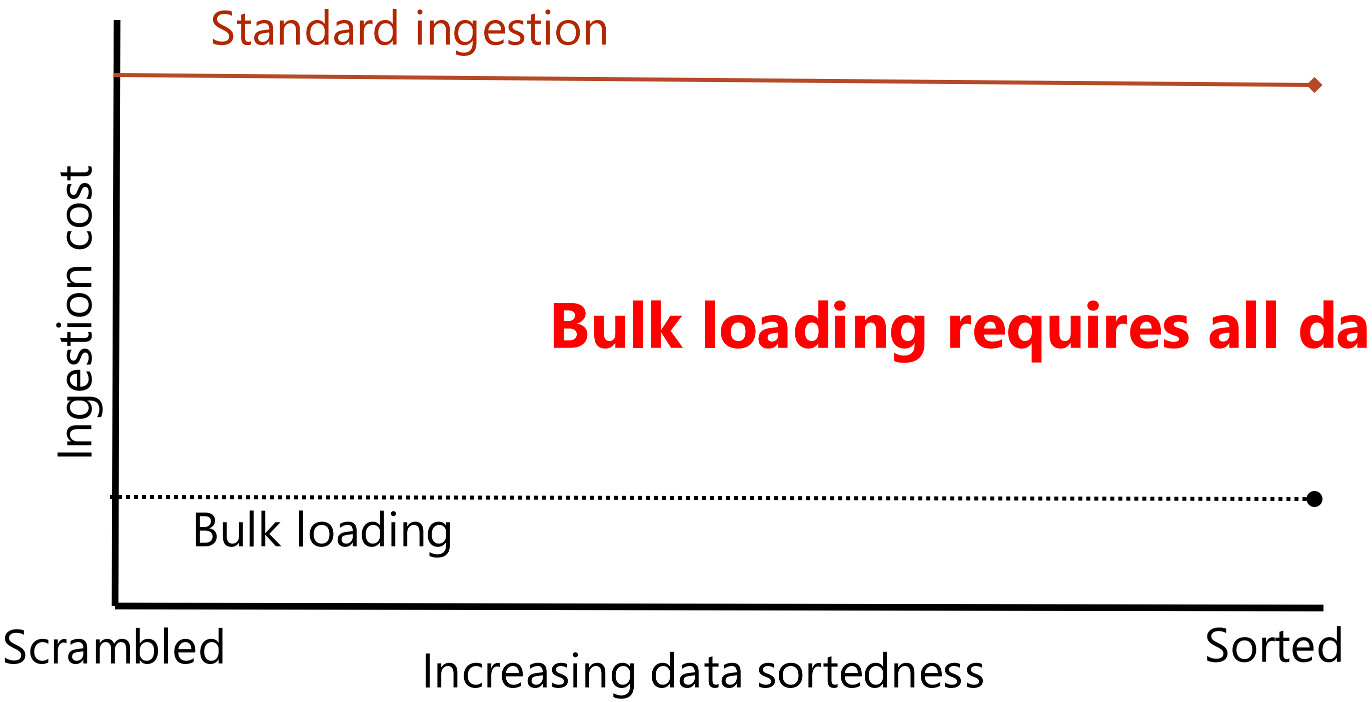
Near-sorted data

≊ treated same as unstructured data!

Irrespective of Sortedness, Same Ingestion Performance

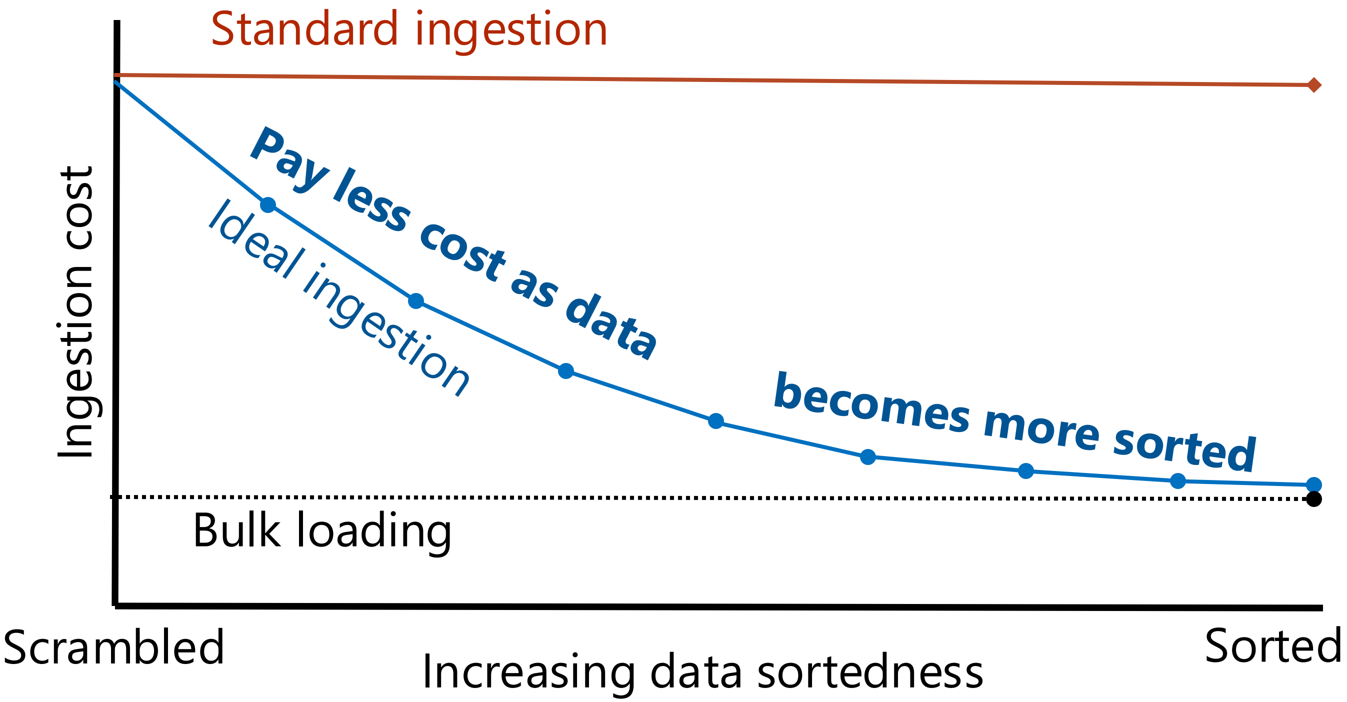


Are There Faster Alternatives?



Bulk loading requires all data a priori!

Ideally, Higher Sortedness Should Lead to Faster Ingestion



Near-Sorted Data is Frequently Found

Time Series

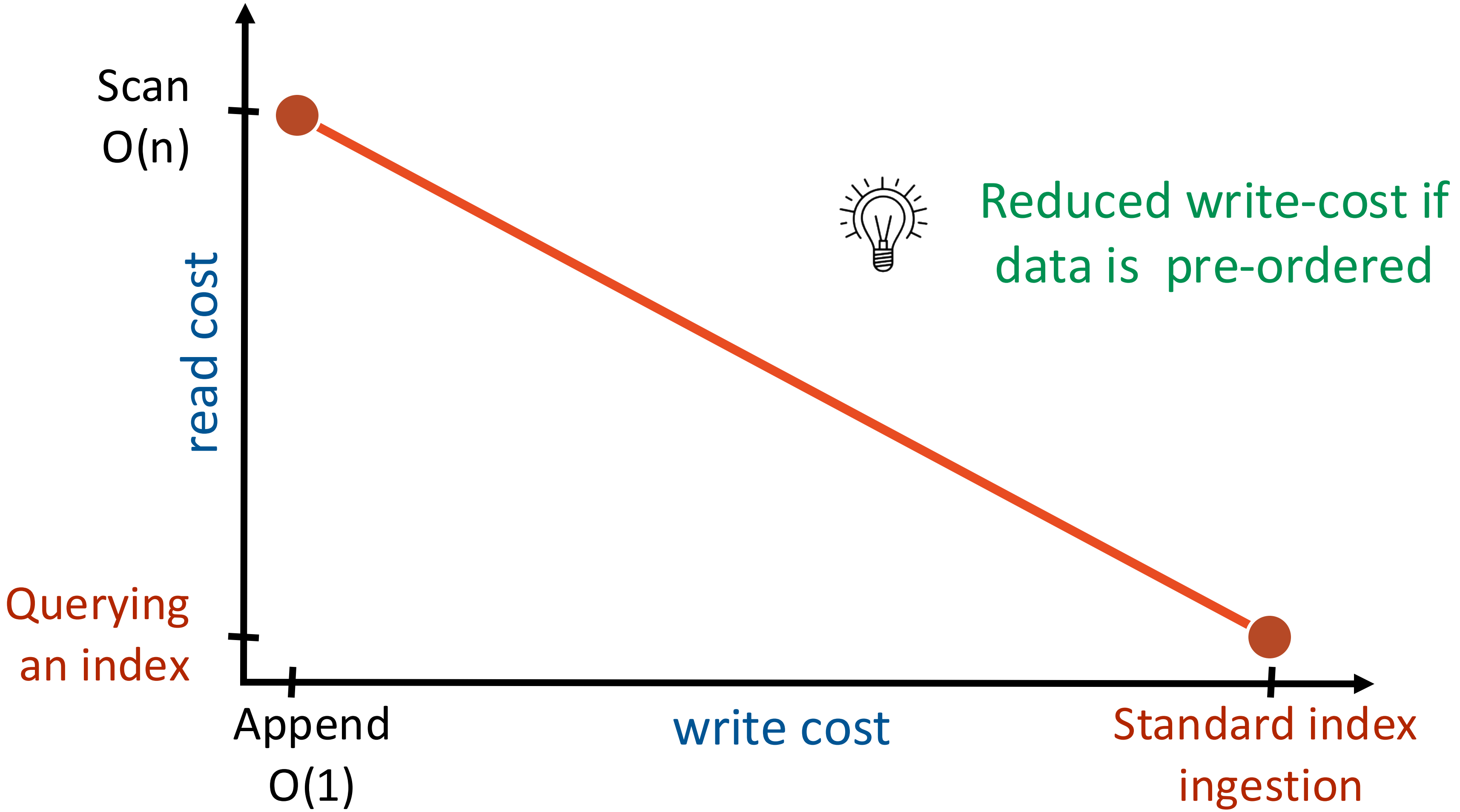
Stock market

Join/query

efficient reads + fast writes

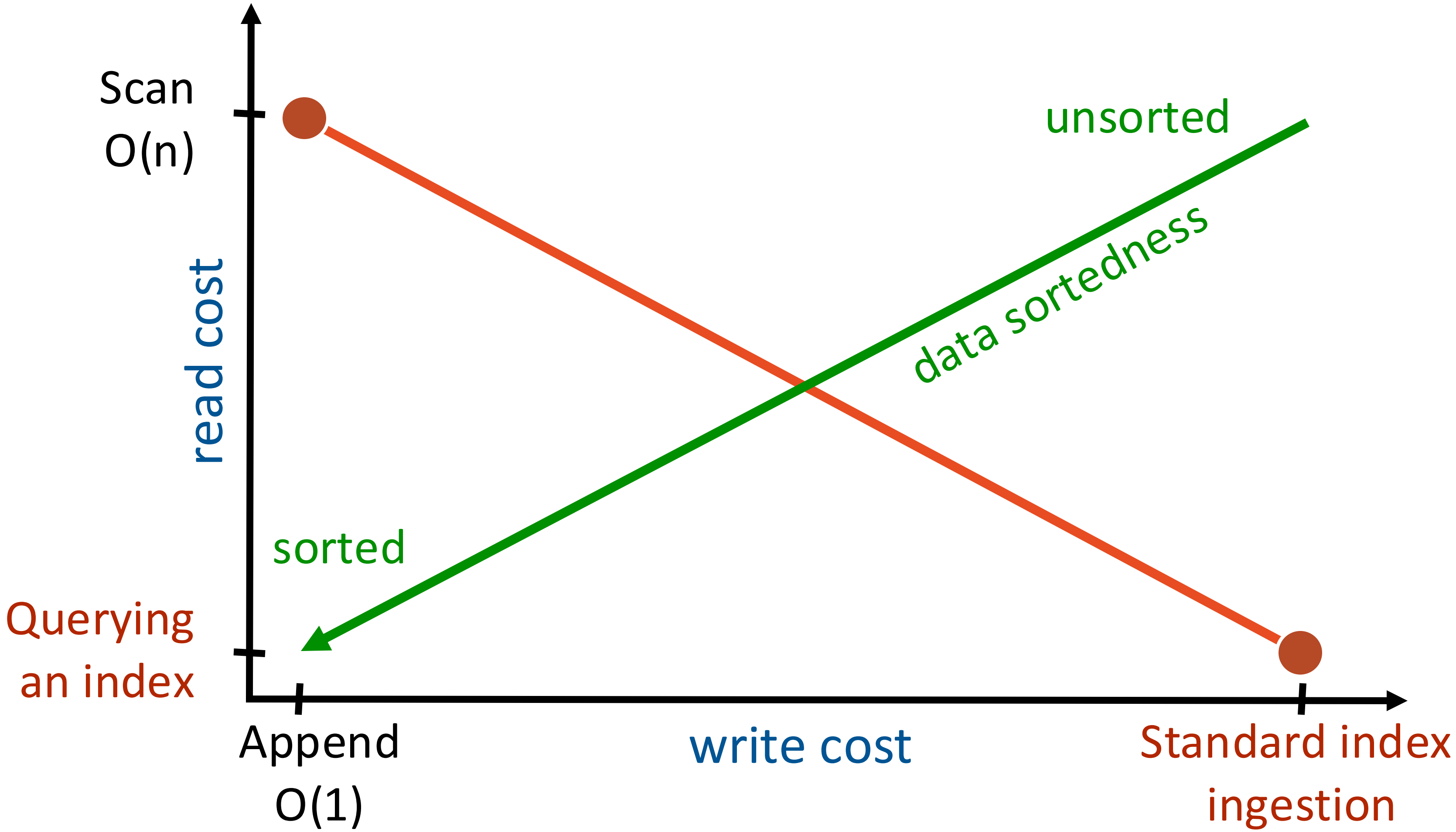
classical indexes carry *redundant* effort!

Vision for Sortedness-Aware Indexing

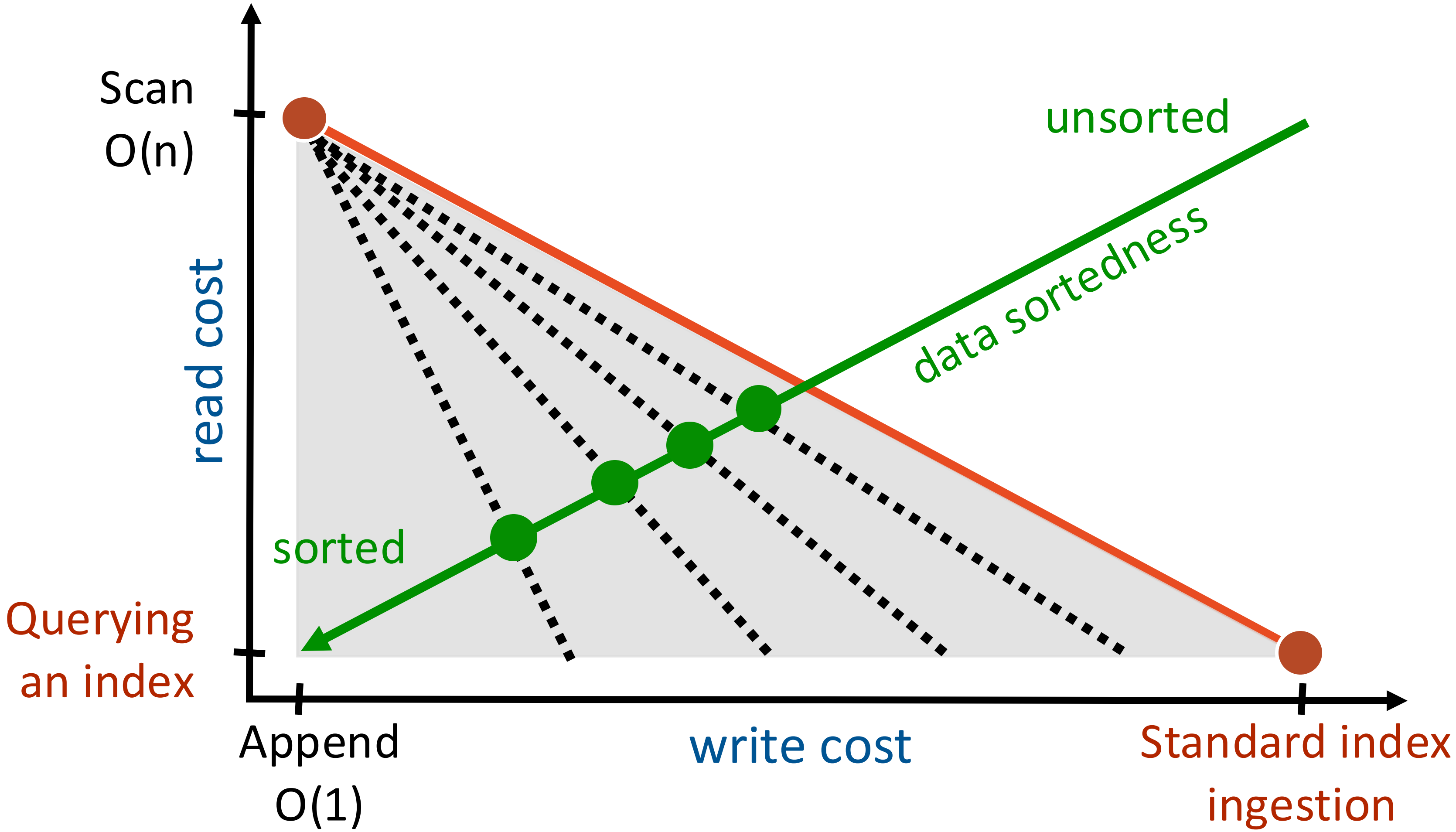


💡 Reduced write-cost if data is pre-ordered

Vision: Sortedness-Aware Indexes



Vision: Sortedness-Aware Indexes



Agenda

Introduction

Vision

Sortedness Metrics

Sortedness Aware (SWARE) Indexing

A Simpler Design

Open Questions

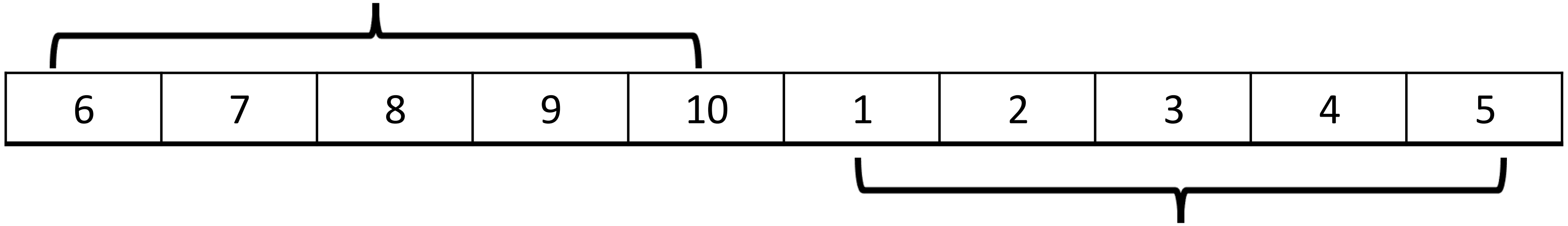
Quantifying Data Sortedness

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

Any downsides of the ”simple” metrics?

Quantifying Data Sortedness

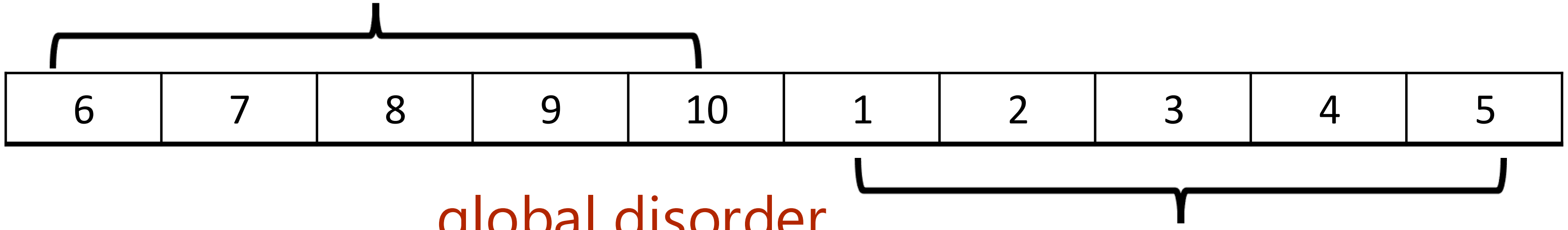
Metric	Description
Inversions	# pairs in incorrect order
Runs	# increasing contiguous subsequences
Exchanges	least # swaps needed to establish total order



6	7	8	9	10	1	2	3	4	5

Quantifying Data Sortedness

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

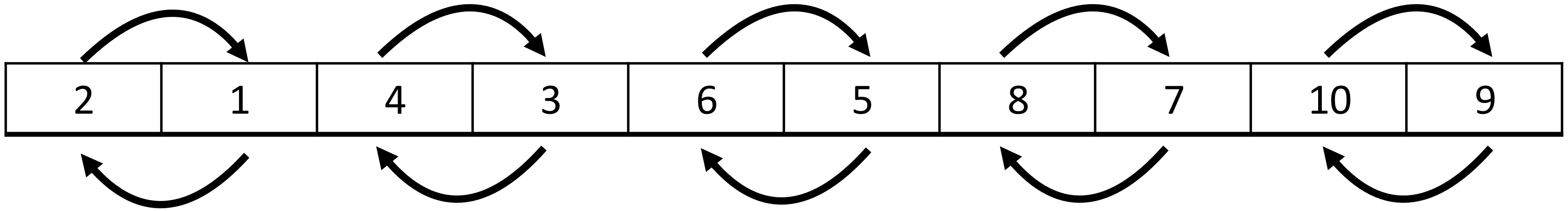


6	7	8	9	10	1	2	3	4	5

global disorder

Quantifying Data Sortedness

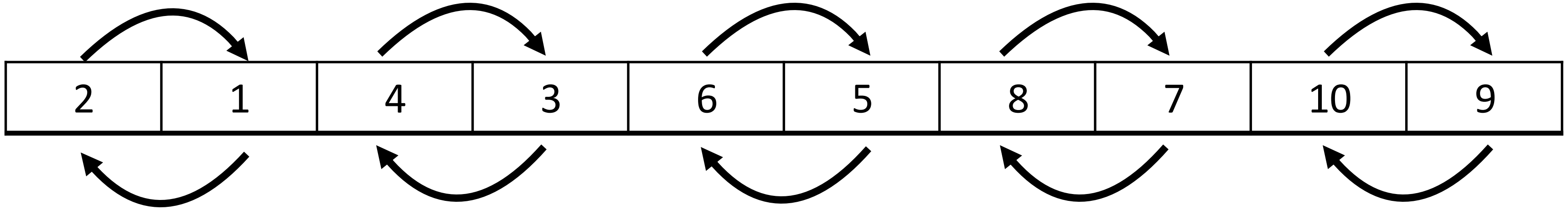
Metric	Description
Inversions	# pairs in incorrect order
Runs	# increasing contiguous subsequences
Exchanges	least # swaps needed to establish total order



2	1	4	3	6	5	8	7	10	9

Quantifying Data Sortedness

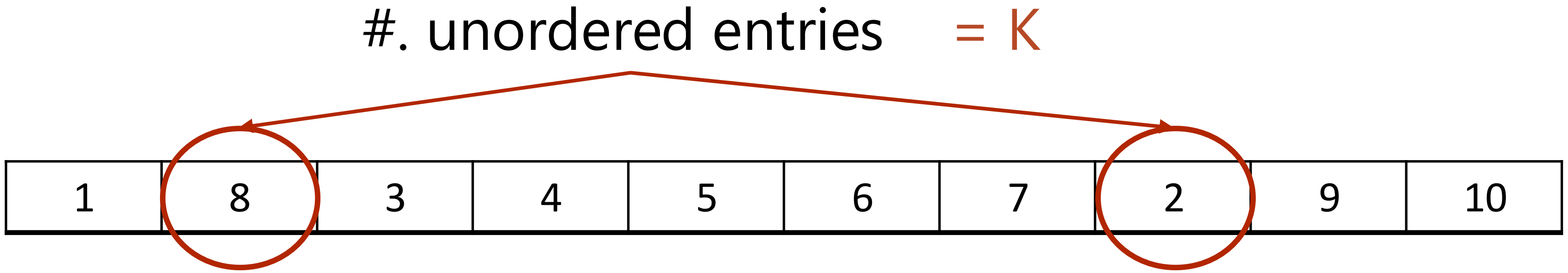
Metric	Description
Inversions ✅	# pairs in incorrect order
Runs ⛔	# increasing contiguous subsequences
Exchanges ✅	least # swaps needed to establish total order



2	1	4	3	6	5	8	7	10	9

local disorder

(K, L)-Sortedness Metric

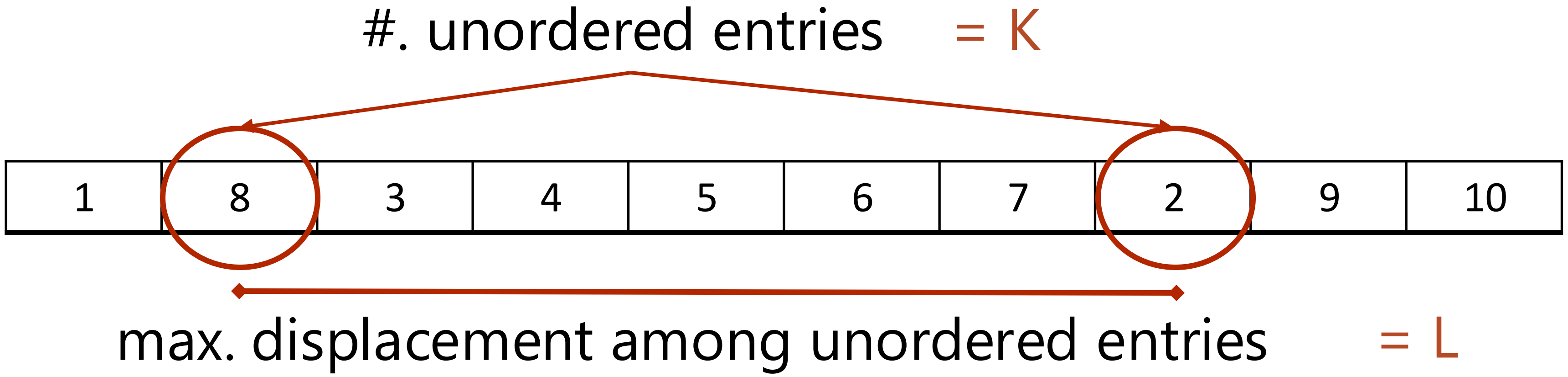


#. unordered entries = K

1	8	3	4	5	6	7	2	9	10

[inspired by BenMoshe, ICDT 2011]

(K, L)-Sortedness Metric



#. unordered entries = K

1	8	3	4	5	6	7	2	9	10

max. displacement among unordered entries = L

[inspired by BenMoshe, ICDT 2011]

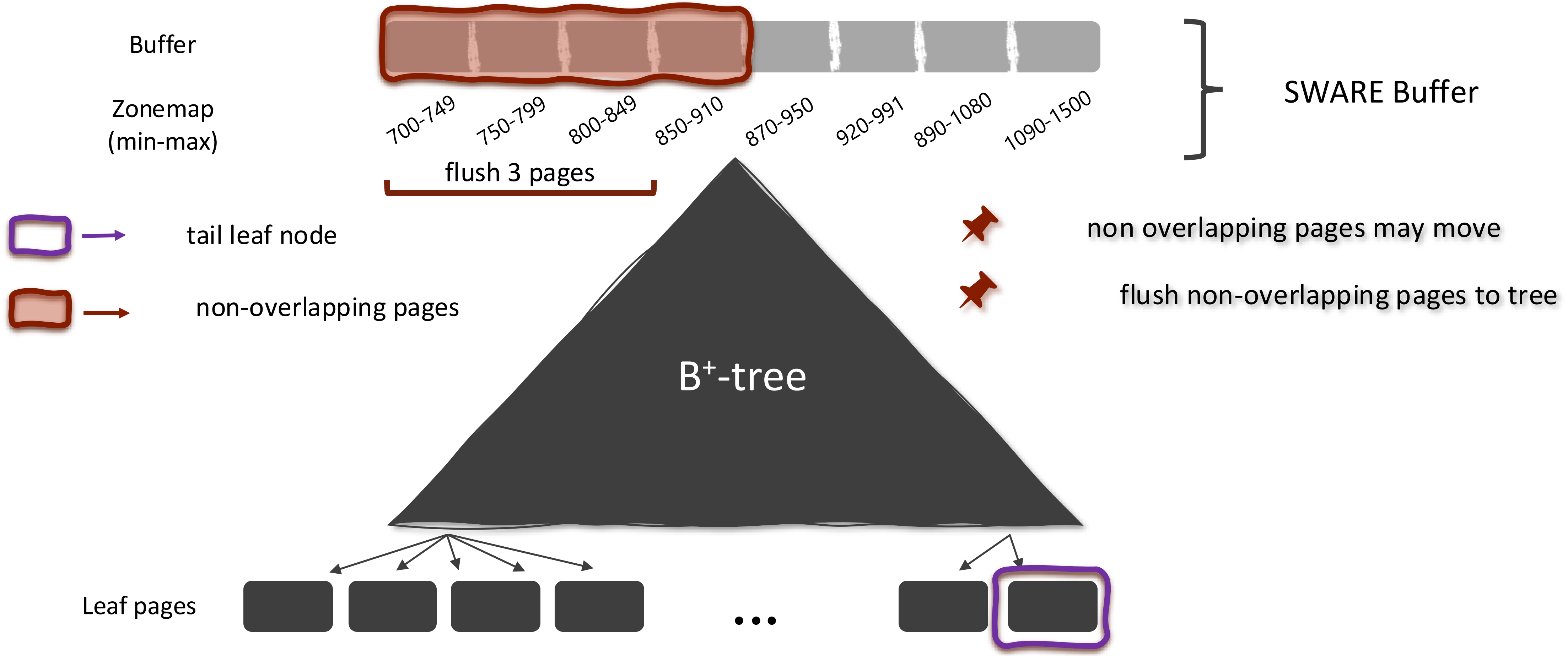
The Sortedness-Aware (SWARE) Paradigm

Sortedness-Aware (SWARE) Paradigm

intelligent buffering + opportunistic bulk loading + increased fill and split factor

SWARE framework can be applied to any tree-index!

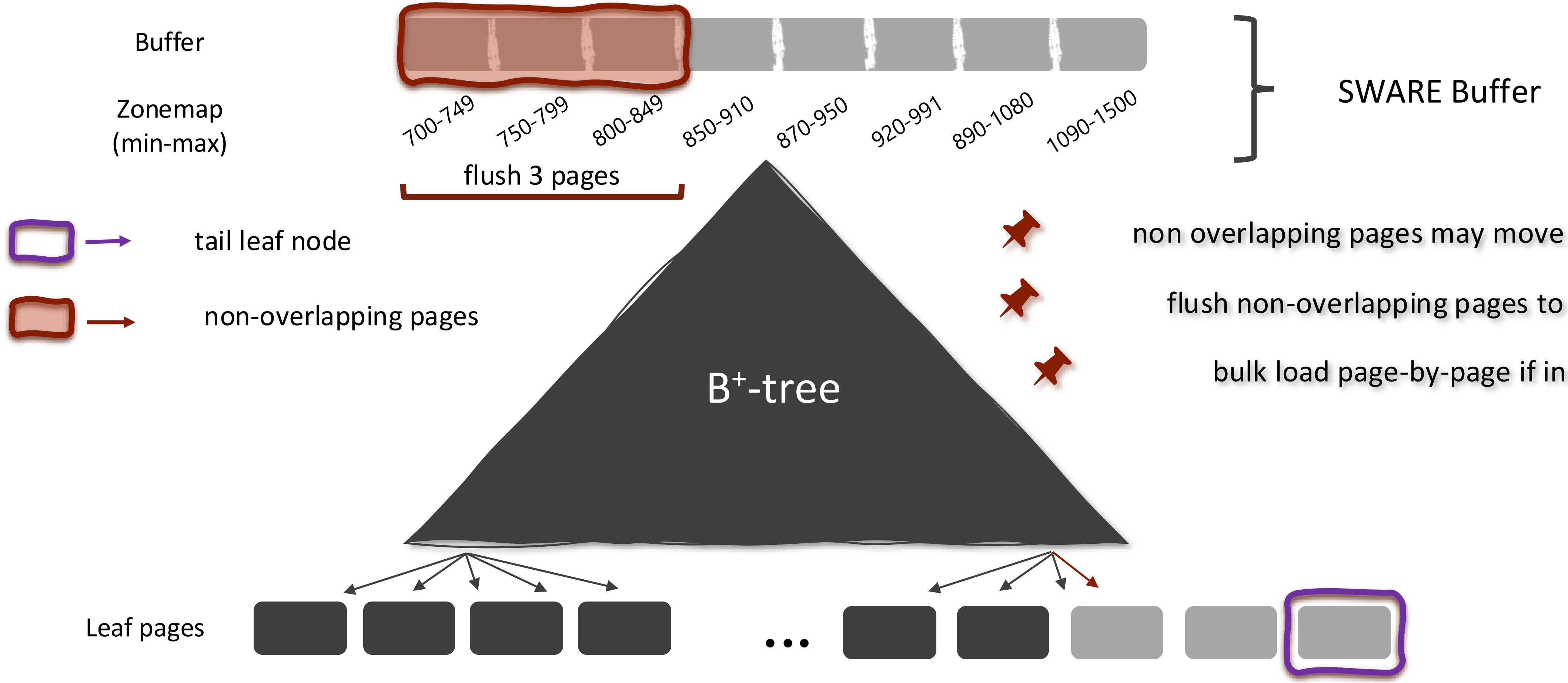
SWARE Ingestions



📌 non overlapping pages may move

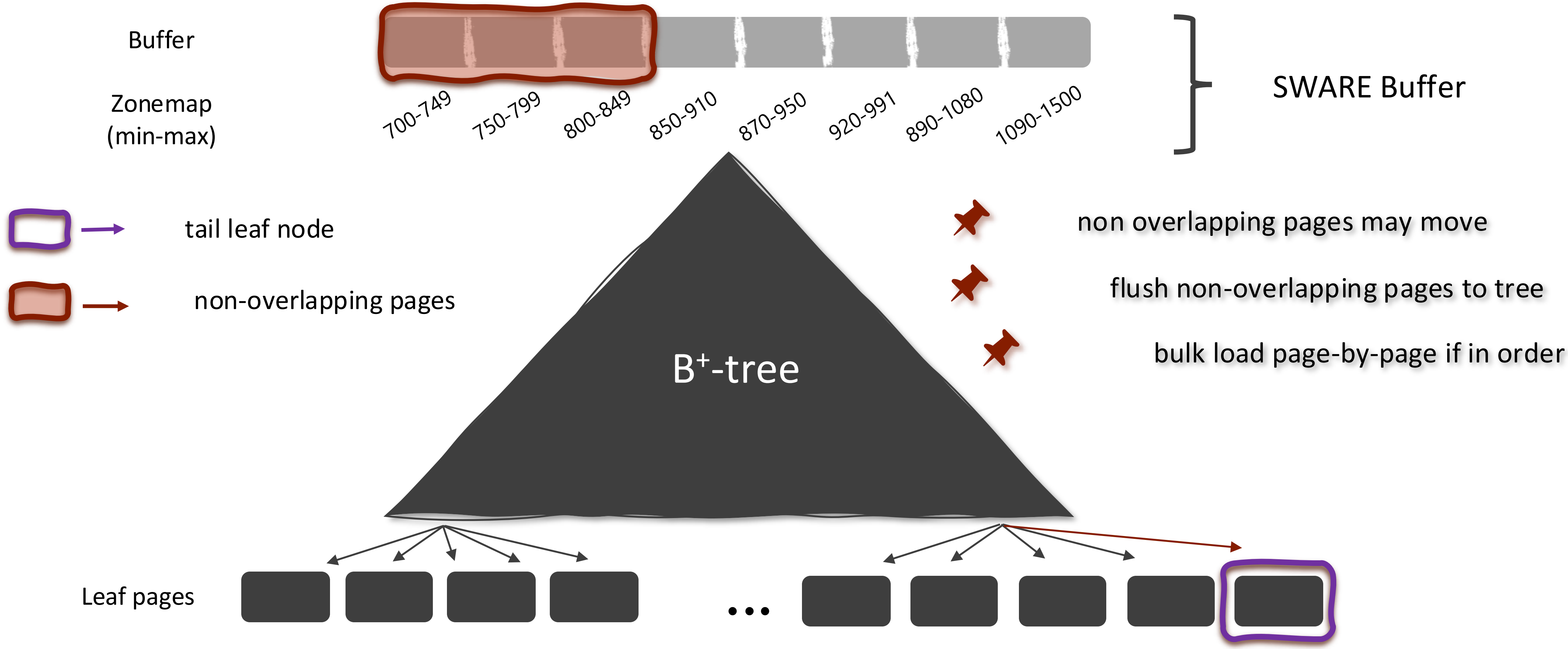
📌 flush non-overlapping pages to tree

SWARE Ingestions



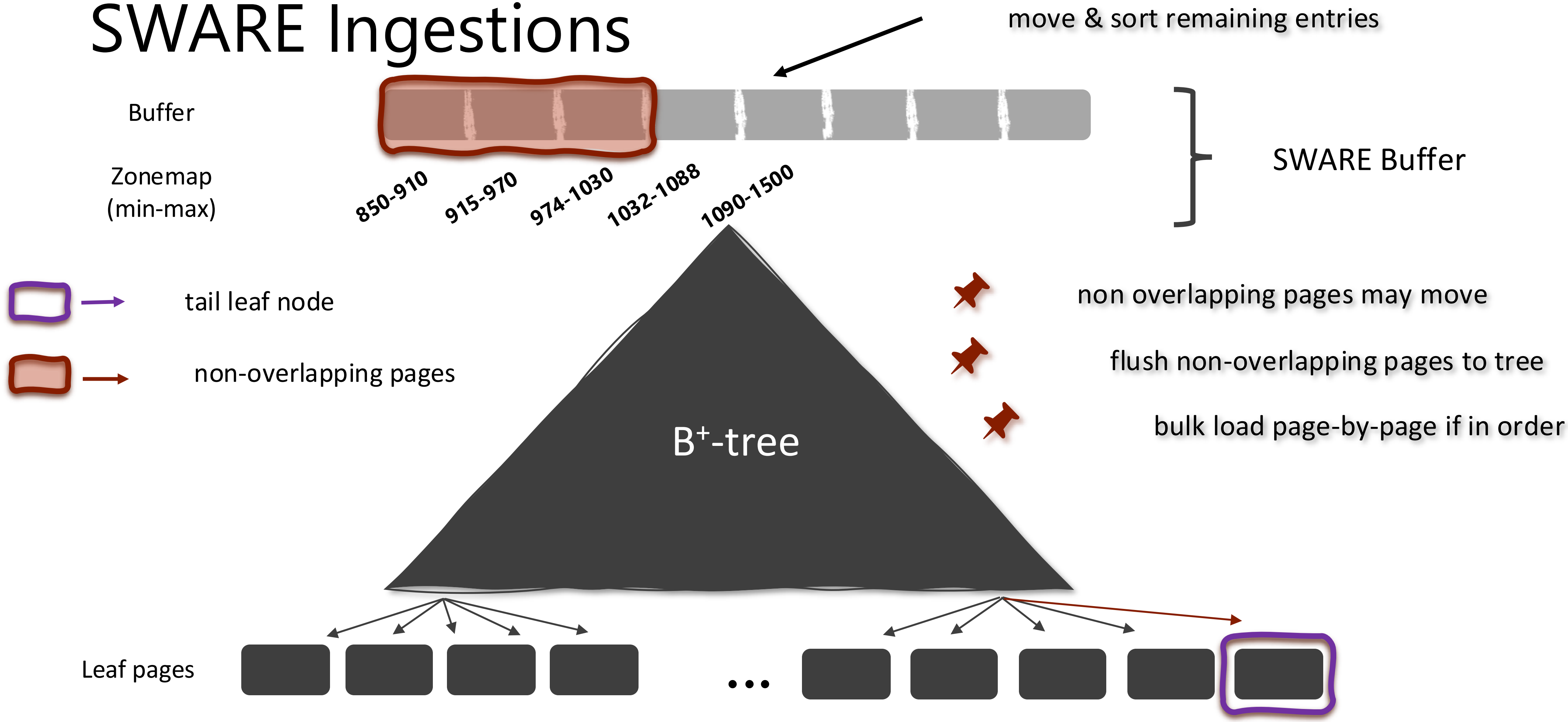
- non overlapping pages may move
- flush non-overlapping pages to tree
- bulk load page-by-page if in order

SWARE Ingestions



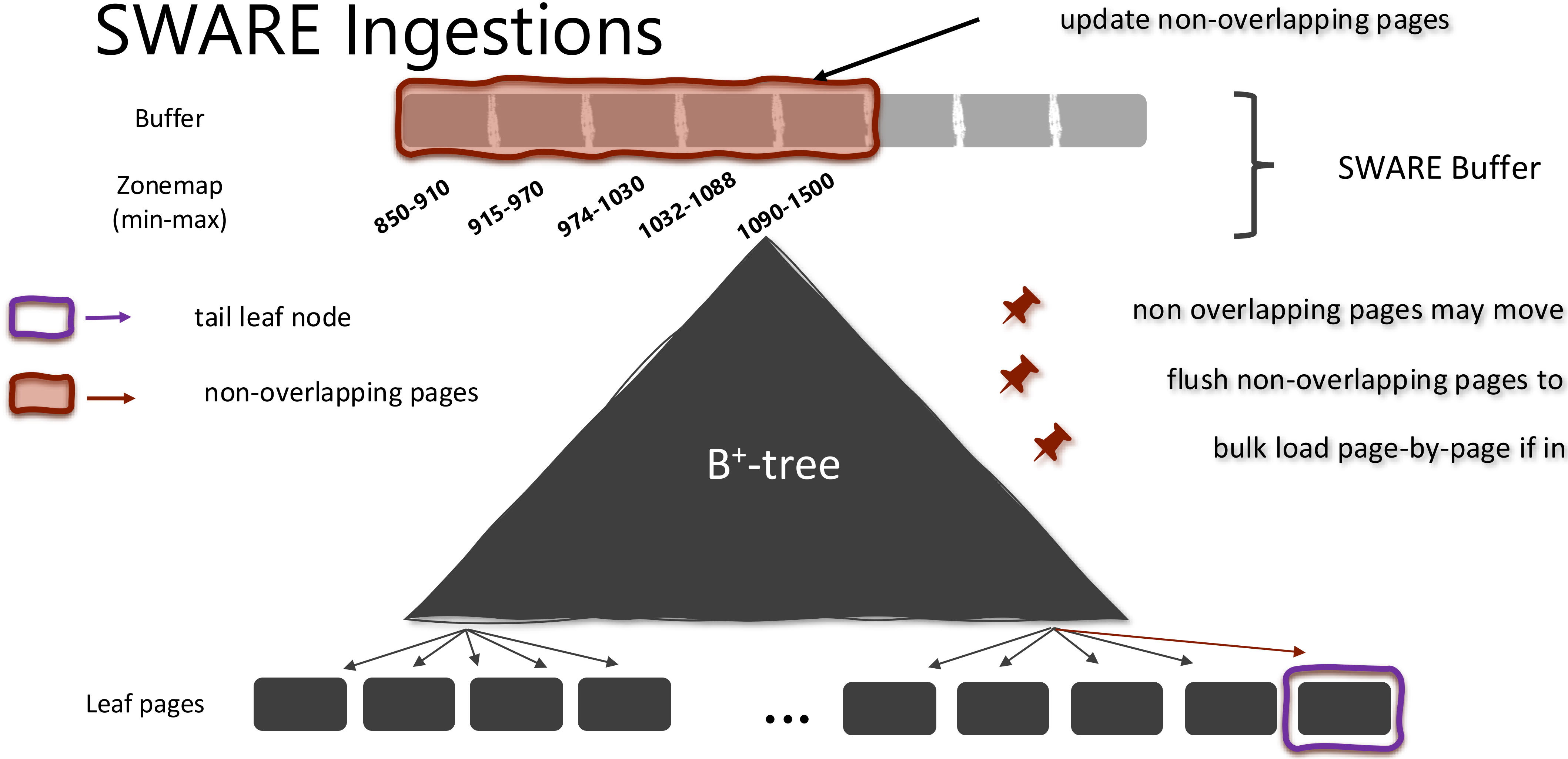
- non overlapping pages may move
- flush non-overlapping pages to tree
- bulk load page-by-page if in order

SWARE Ingestions



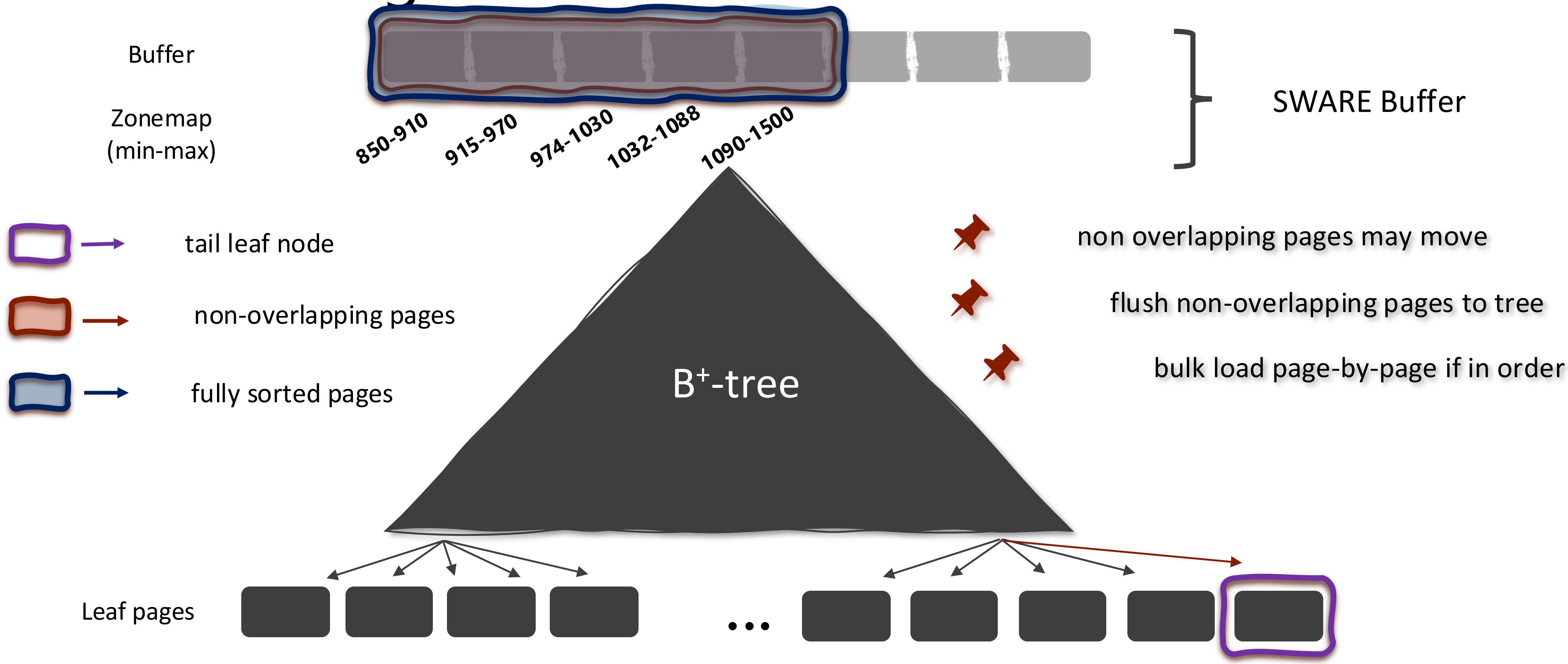
- non overlapping pages may move
- flush non-overlapping pages to tree
- bulk load page-by-page if in order

SWARE Ingestions



- non overlapping pages may move
- flush non-overlapping pages to tree
- bulk load page-by-page if in order

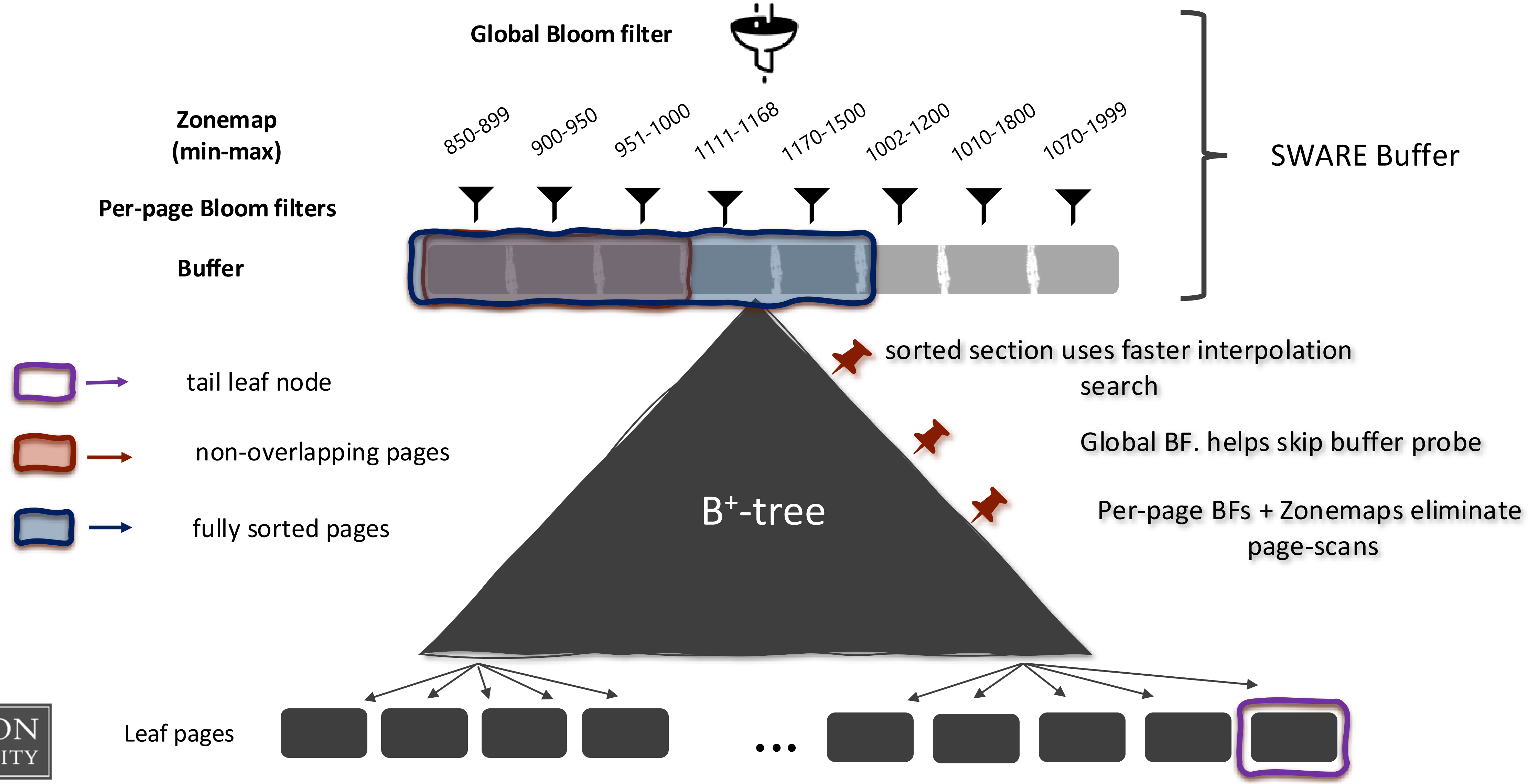
SWARE Ingestions



- non overlapping pages may move
- flush non-overlapping pages to tree
- bulk load page-by-page if in order

How do lookups work?

SWARE Lookups



- ▭ tail leaf node
- 🟥 non-overlapping pages
- 🟦 fully sorted pages

sorted section uses faster interpolation search

Global BF. helps skip buffer probe

Per-page BFs + Zonemaps eliminate page-scans

Experimental Evaluation

System Setup:

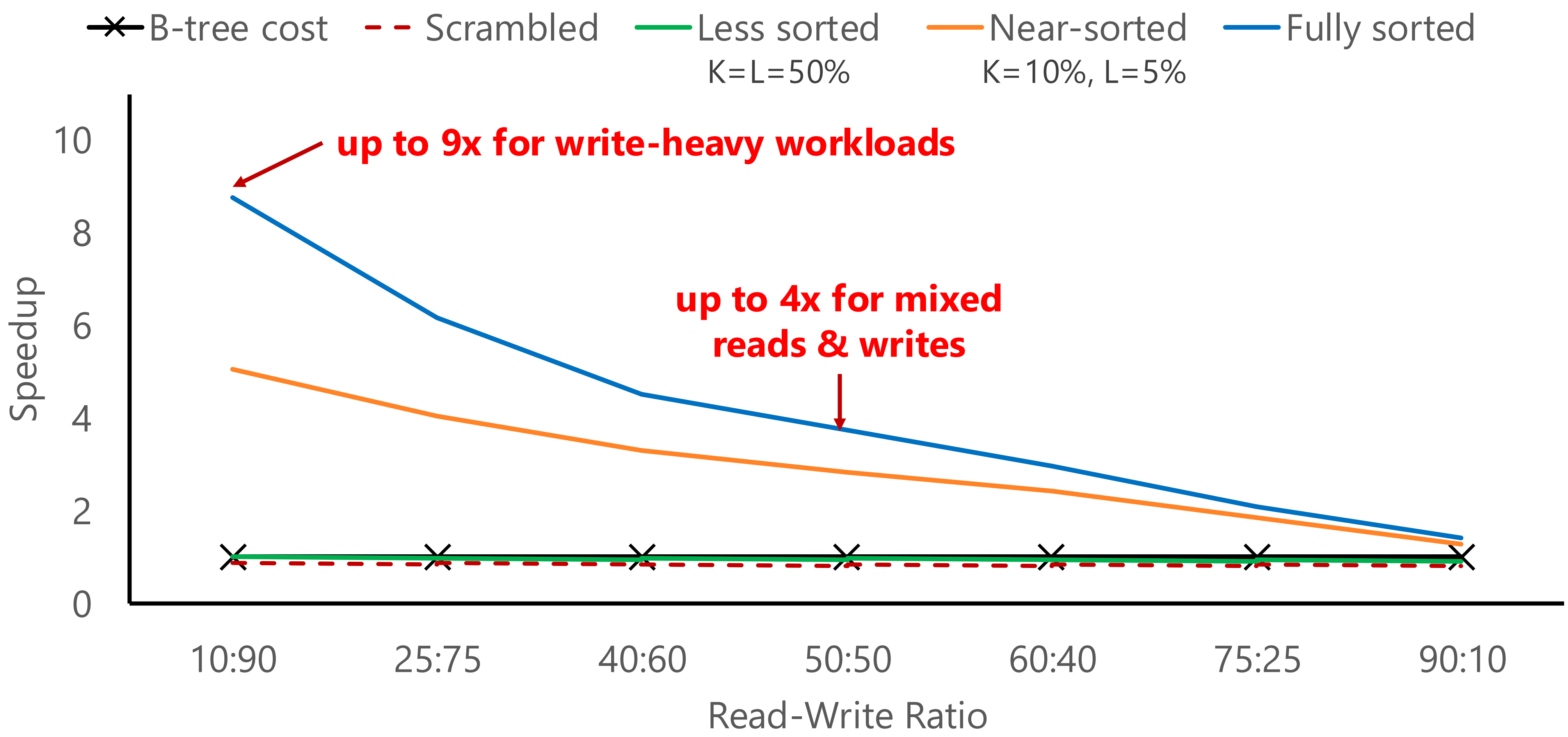
- Intel Xeon Gold 5230
- 2.1GHZ processor w. 20 cores
- 384GB RAM, 28MB L3 cache

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$^\varepsilon$-tree

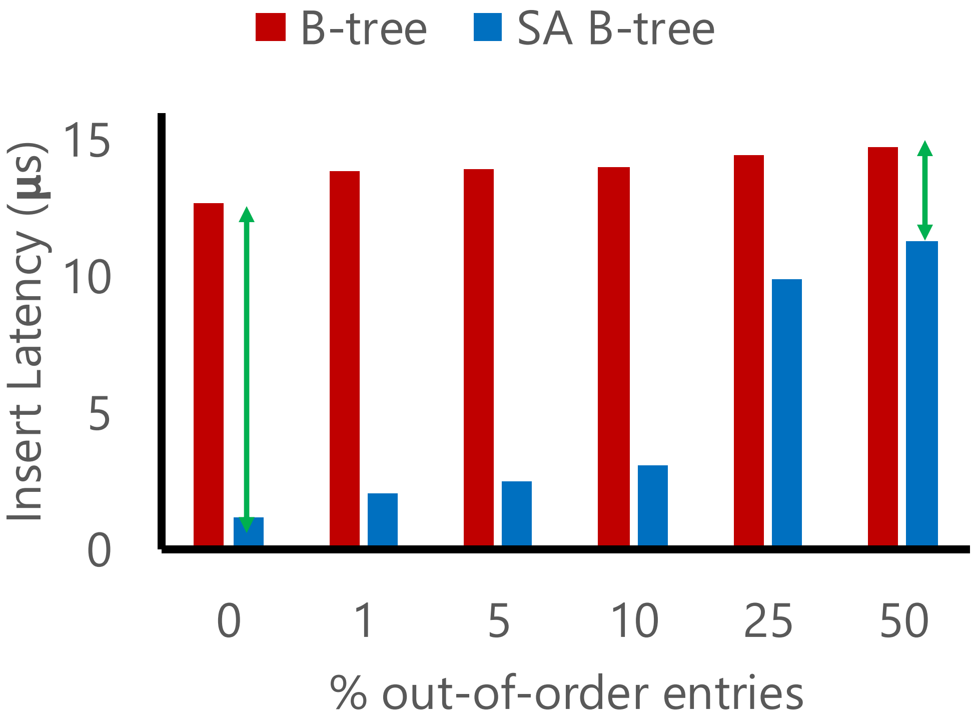
Evaluating SWARE Under Varying Sortedness



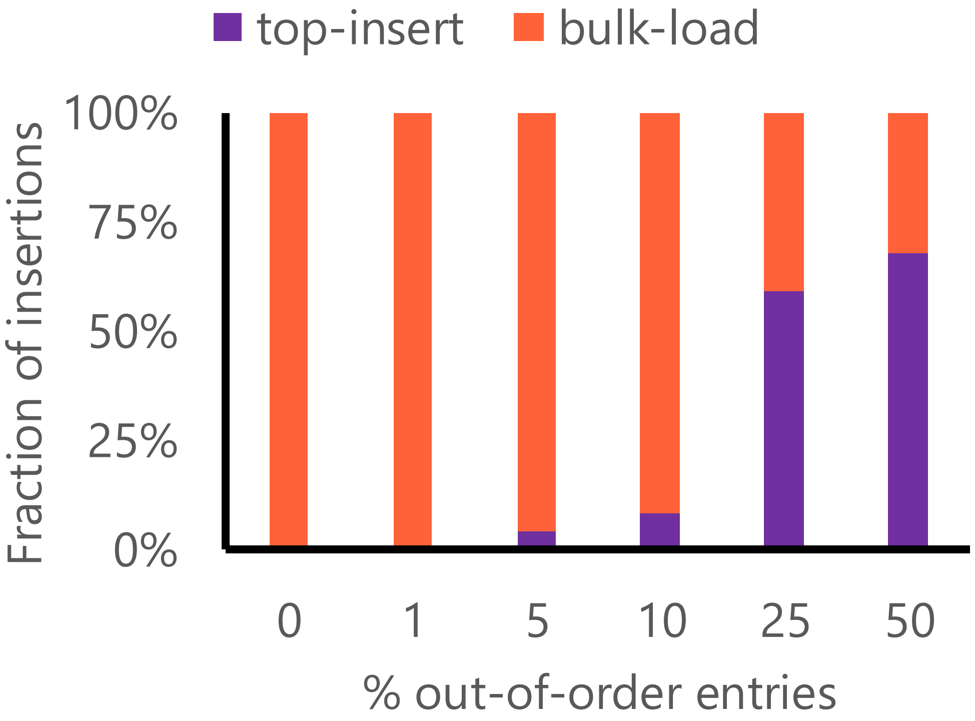
up to 9x for write-heavy workloads

up to 4x for mixed reads & writes

Raw Ingestion Performance

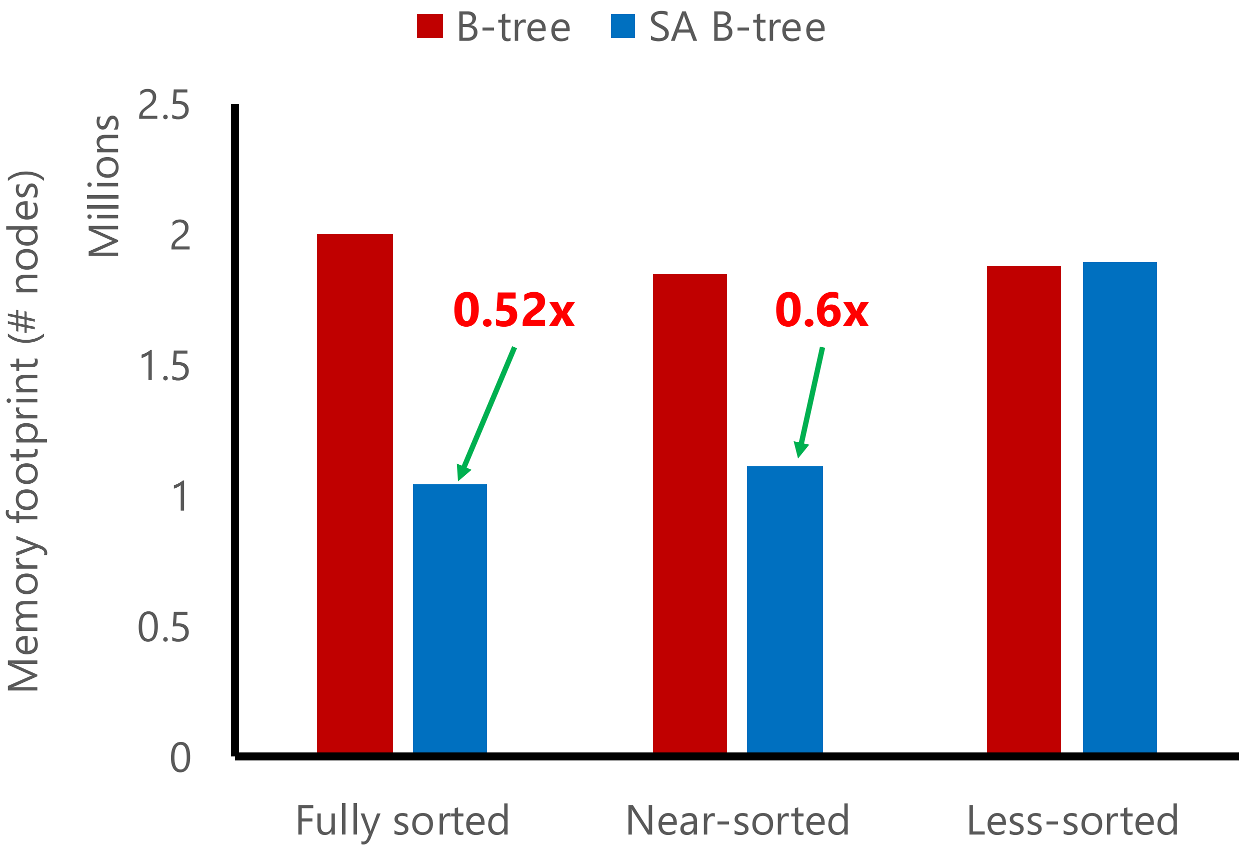


ingestion latency reduced between 27-90%



bulk loading maximized with high data sortedness

SWARE Improves Space Utilization



increased fill/split factor helps reduce memory footprint

Summarizing SWARE [ICDE 2023]

intelligent buffering + opportunistic bulk loading

Improves performance by exploiting sortedness

Any downsides to wider applicability?

Summarizing SWARE [ICDE 2023]

intelligent buffering + opportunistic bulk loading

Improves performance by exploiting sortedness

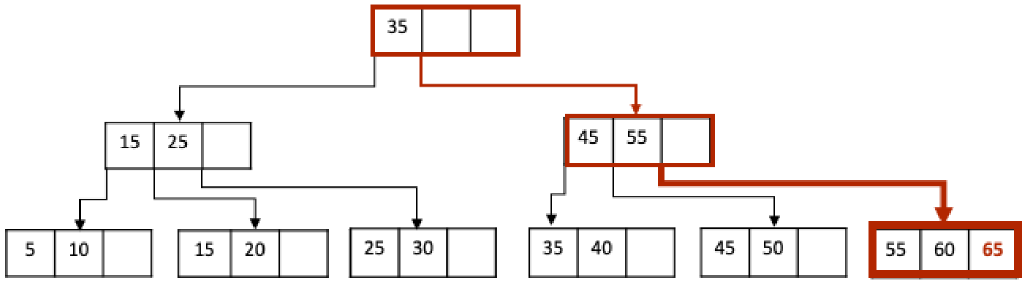
Increases Complexity in Design!

Can we achieve fast ingestions <u>without</u> buffering?

Inserting to the Tail-leaf (PostgreSQL & MySQL)

Normal Insertion (top-insert)

Insert Key 65

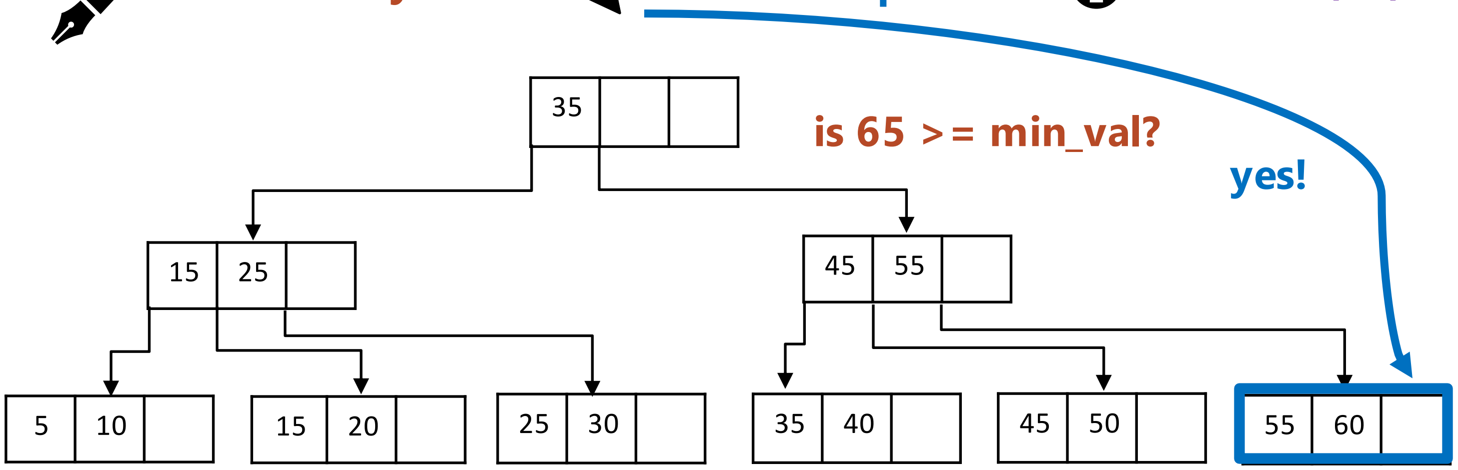


Tail-leaf Insertion

Insert Key 65

tail-leaf-ptr

min_val(55)



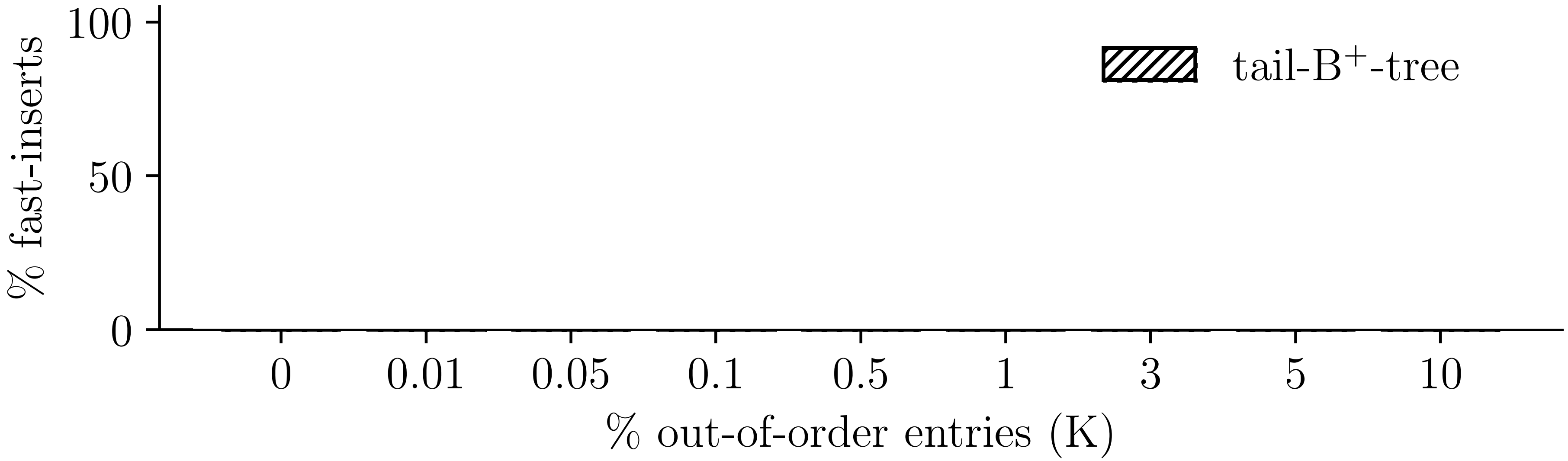
is 65 >= min_val?

yes!

add key to tail leaf directly!

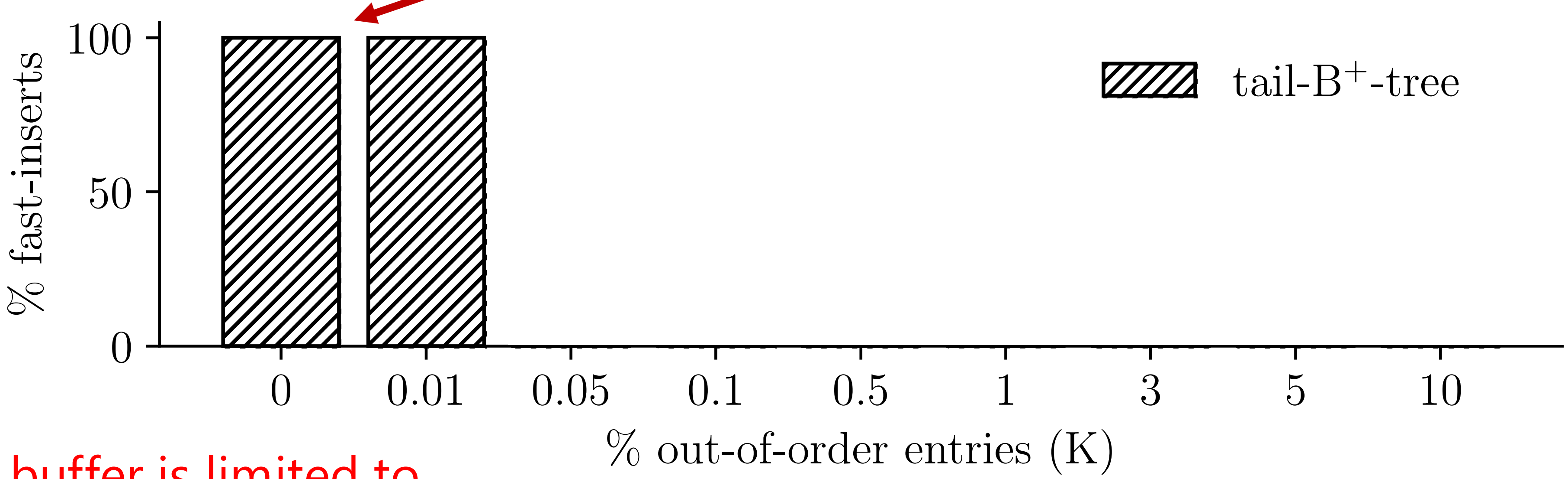
Is the tail-leaf optimization the solution?

Does Tail-leaf Insertion Work?



Does Tail-leaf Insertion Always Work?

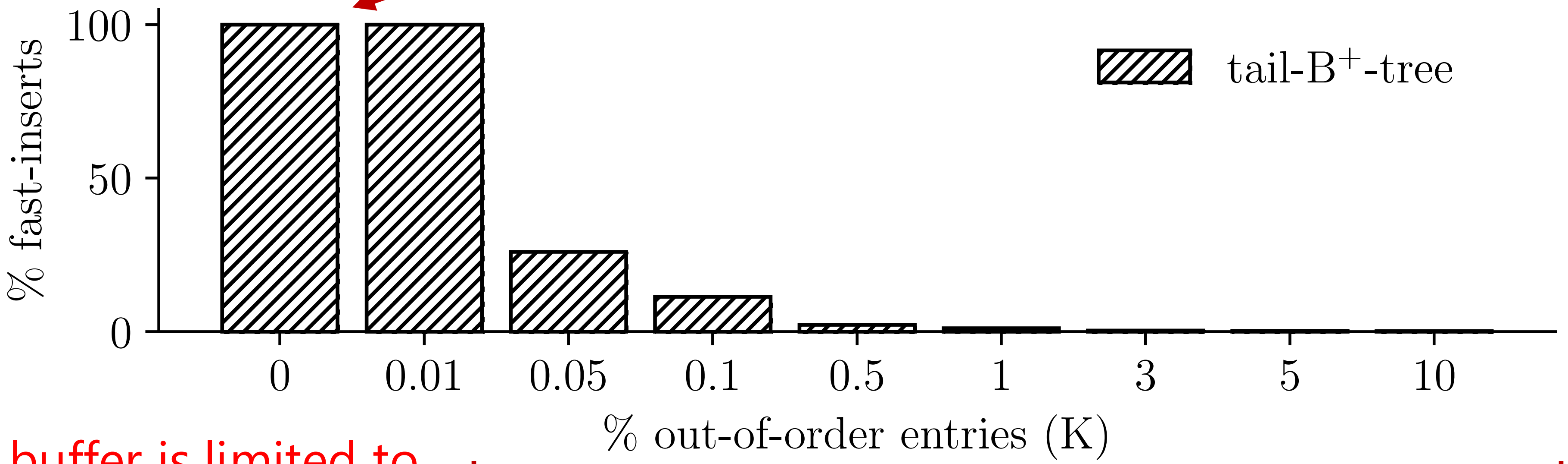
Works for fully and very highly sorted data



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

Does Tail-leaf Insertion Always Work?

Works for fully and very highly sorted data

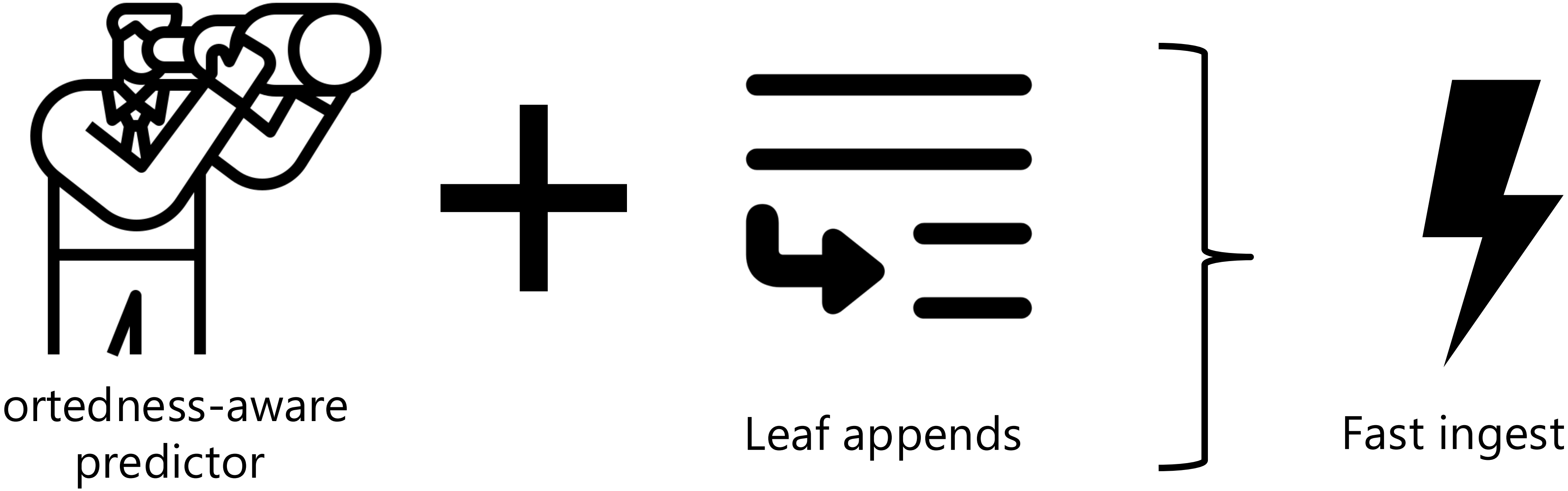


Tail-leaf's buffer is limited to leaf node!

Degrades very quickly

However, tail-leaf points us to the right direction…

Key Idea – Predicting the Ordered LEaf (POLE)



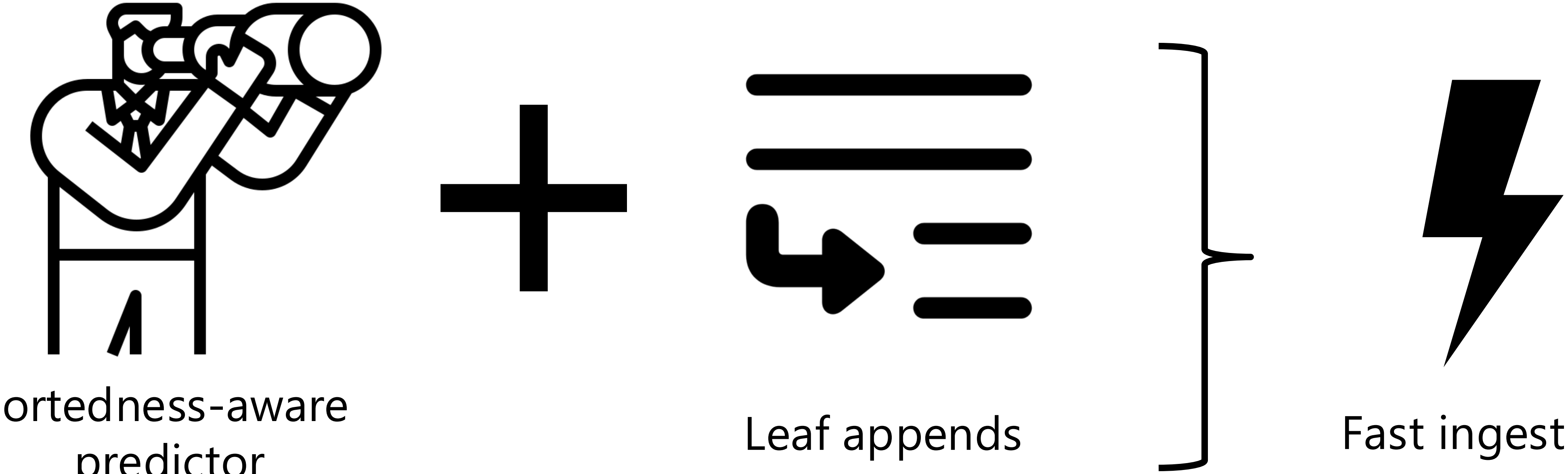
Sortedness-aware predictor

Leaf appends

Fast ingestion

Key Idea – Predicting the Ordered LEaf (POLE)

it could be *<u>any</u>* <u>node</u>!

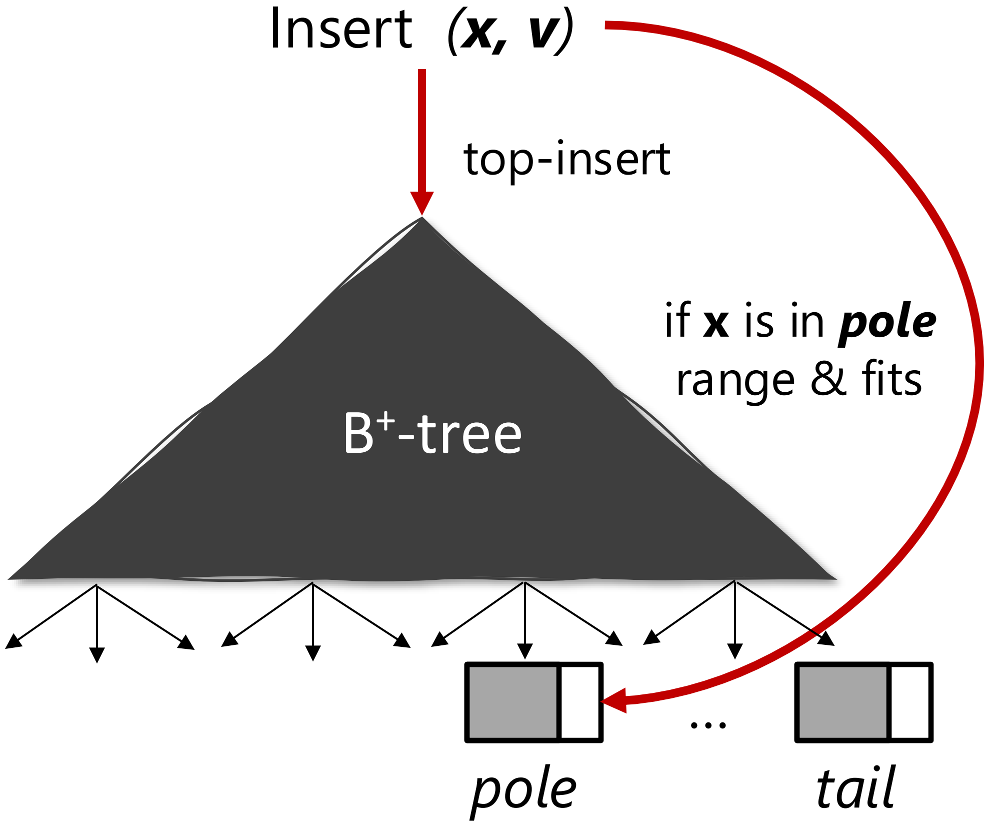


Sortedness-aware predictor

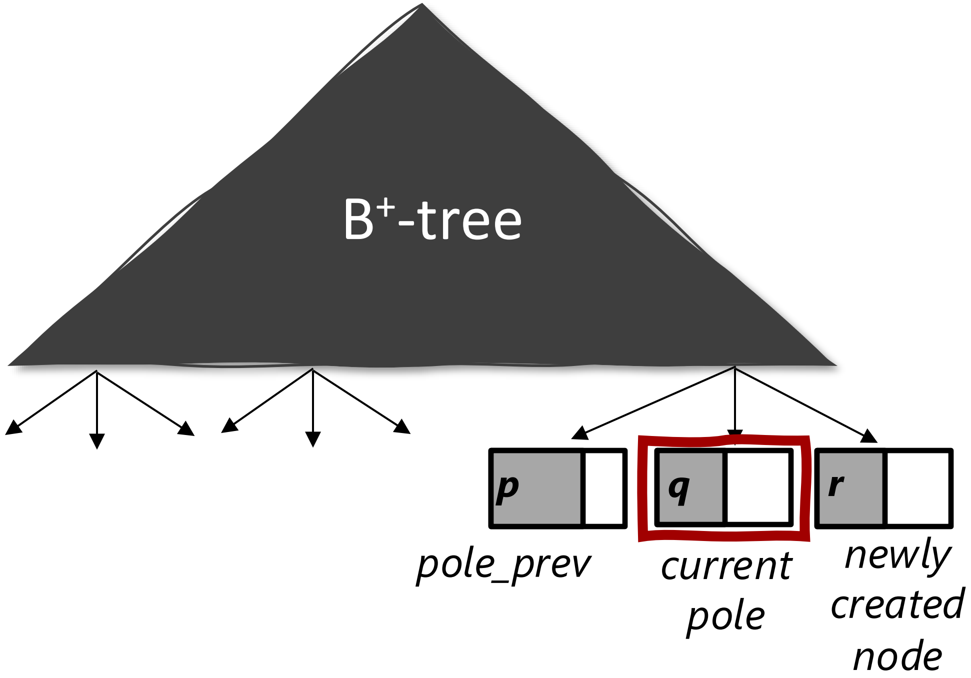
Leaf appends

Fast ingestion

Insertions in Steady-State



When Pole Splits



Legend

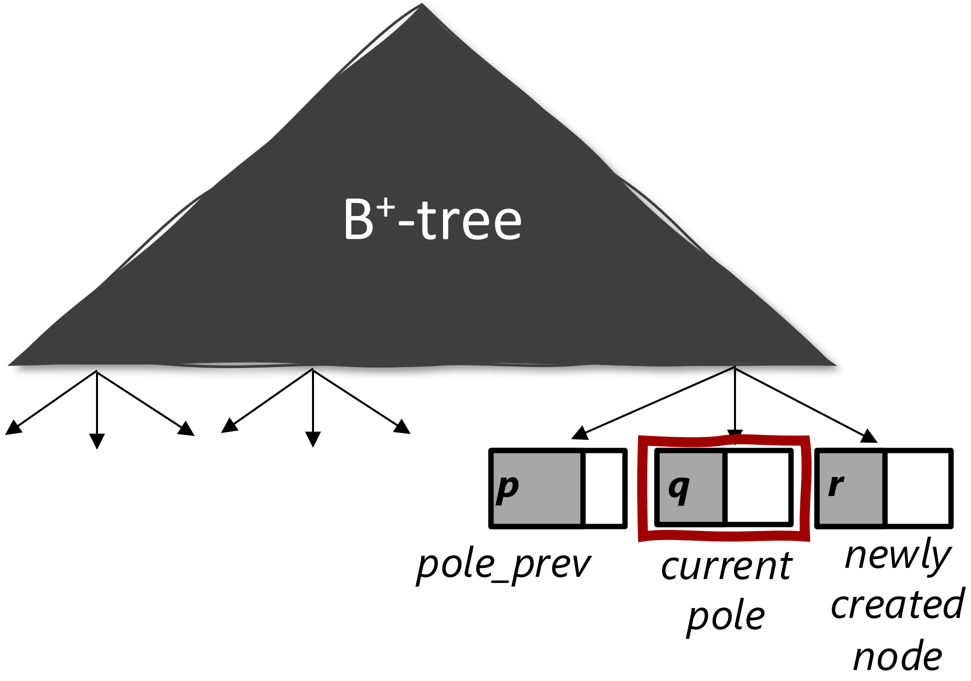
p = smallest entry in node previous to *pole;*
q = smallest entry in *pole*
r **=** smallest entry in newly created node
□**=** pointer to *pole* node

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

$$x = q + \underbrace{\left(\frac{q-p}{pole\_prev_{size}}\right)}_{\text{density between two non-outliers}} \cdot pole_{size} \cdot (1.5)$$

When Pole Splits

if r > x, new node has outliers



pole stays as is

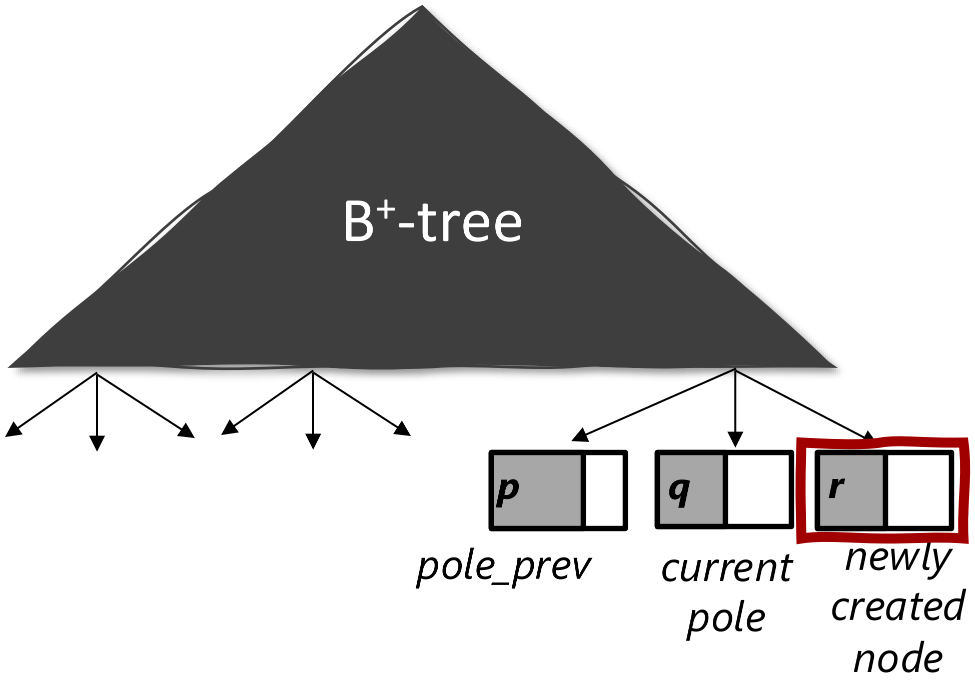
Legend
p = smallest entry in node previous to *pole;*
q = smallest entry in *pole*
r **=** smallest entry in newly created node
□**=** pointer to *pole* node

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

$$x = q + \underbrace{\left(\frac{q-p}{pole\_prev_{size}}\right)}_{\text{density between two non-outliers}} \cdot pole_{size} \cdot (1.5)$$

When Pole Splits

if r <= x, new node has at least one non-outlier value



Legend

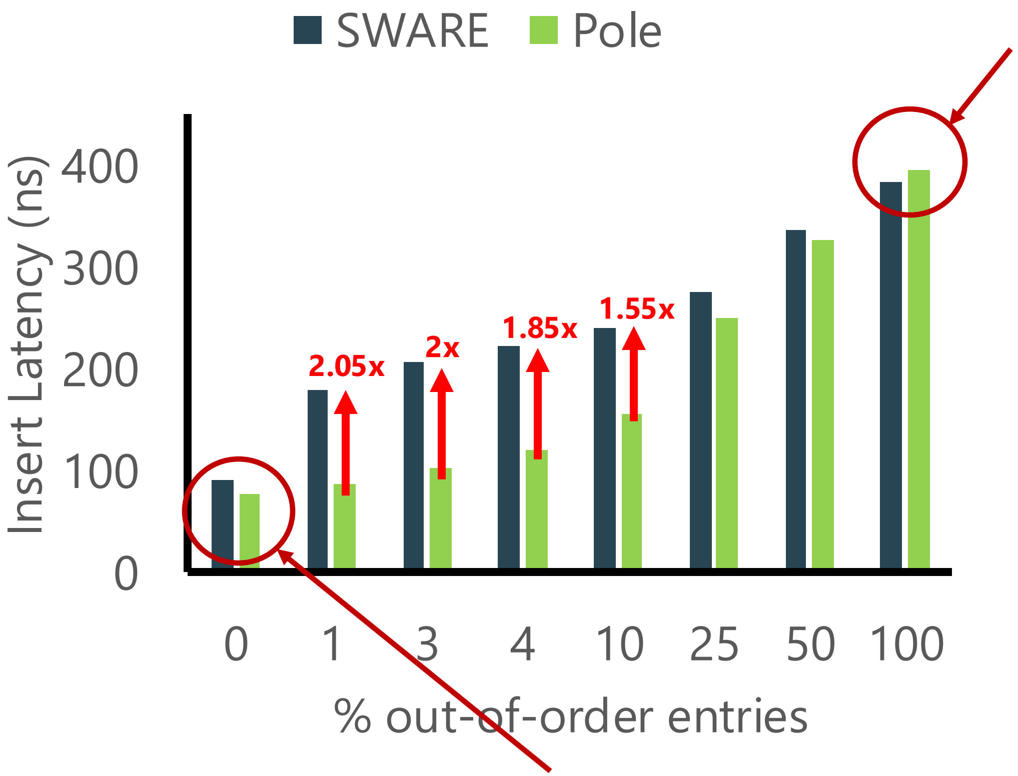
p = smallest entry in node previous to *pole;*
q = smallest entry in *pole*
r **=** smallest entry in newly created node
□ **=** pointer to *pole* node

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

$$x = q + \underbrace{\left(\frac{q - p}{pole\_prev_{size}}\right)}_{\text{density between two non-outliers}} \cdot pole_{size} \cdot (1.5)$$

Update ***pole*** to newly created node from split

Comparing with SWARE



buffer helps: full bulk loading
Pole is still faster!

Buffer pays off: some vs. none fast ingestion

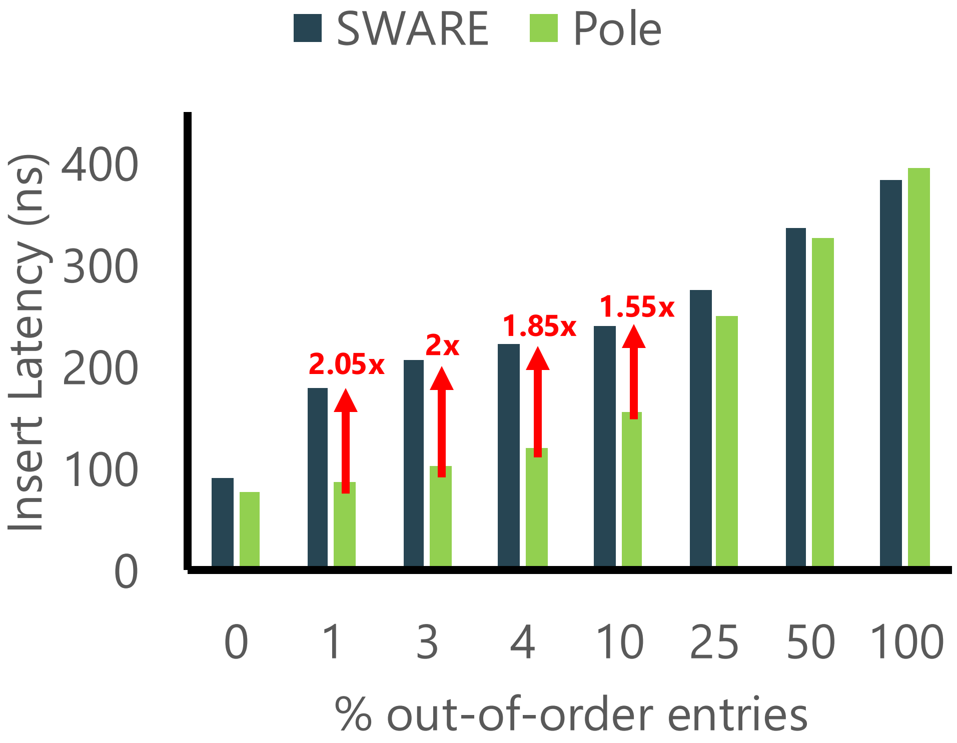
up to 2.05x faster

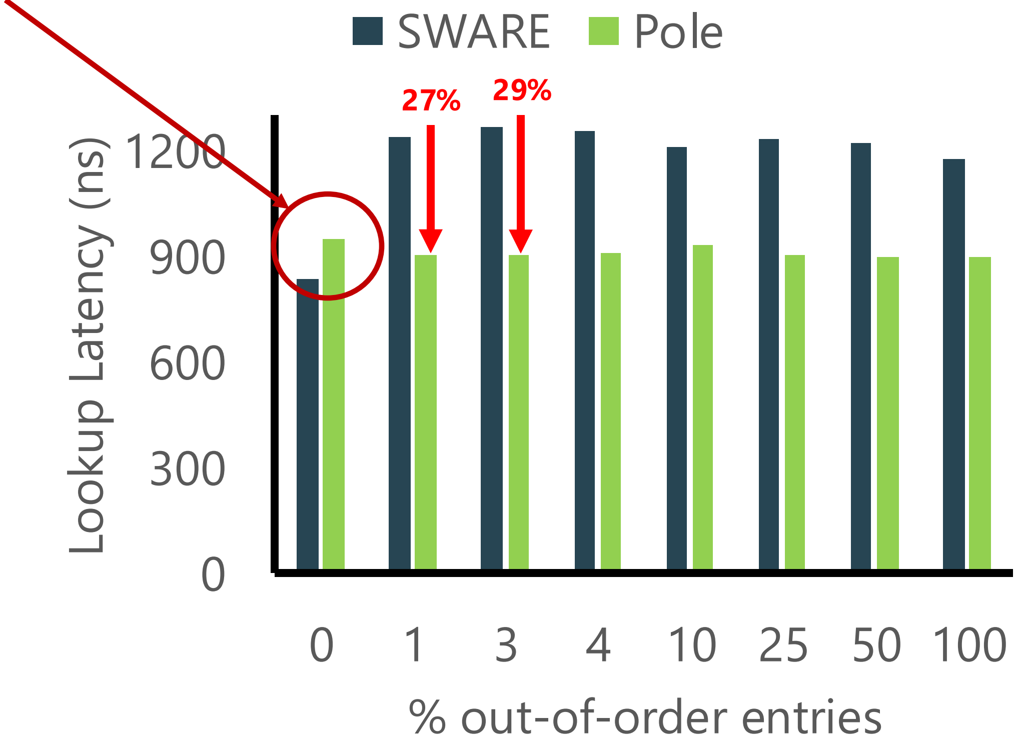
minimal metadata ✓

avoids SWARE buffer management ✓

Comparing with SWARE

full bulk loading ⇒ smaller tree

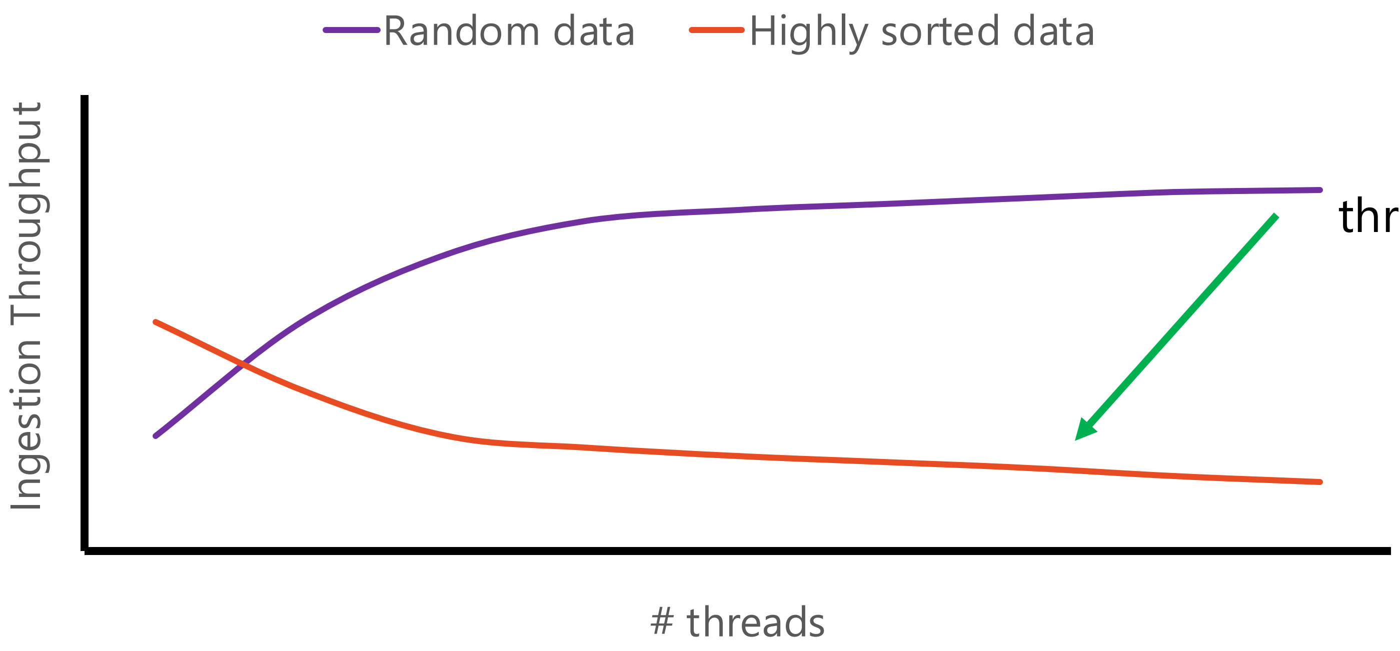




up to 29% faster for point lookups

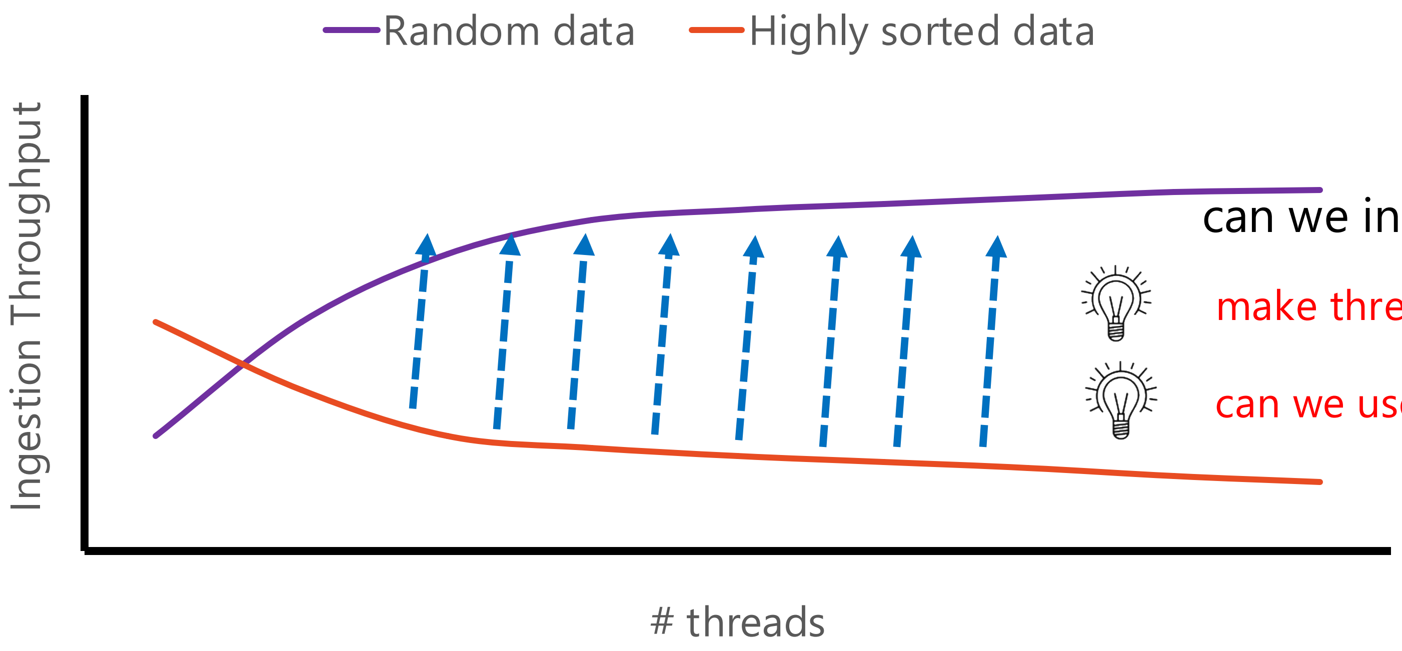
No buffering ⇒ no read overhead!

Future Work - Concurrency in Fast Path



throughput drops due to high contention

Future Work - Concurrency in Fast Path



can we increase throughput for fast-path?

💡 make threads work for and not against each other!

💡 can we use something like consolidation array?

Summary

Identify "sortedness" as a resource

Classical indexes do not exploit sortedness by design!

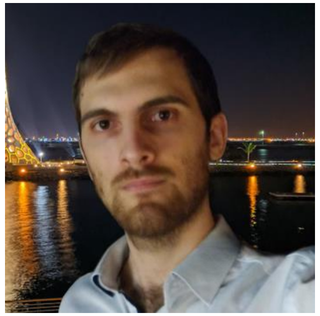
SWARE paradigm & Pole optimization optimize for sortedness

Further research required for learned indexes + joins

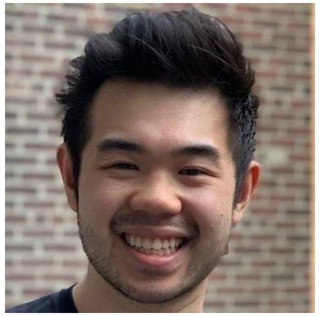
Scan here to learn more about our work

The Team

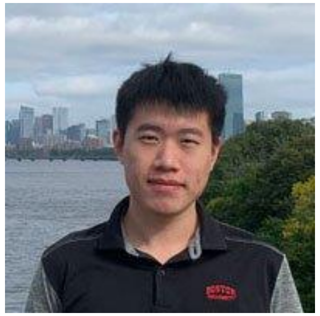
Aneesh Raman



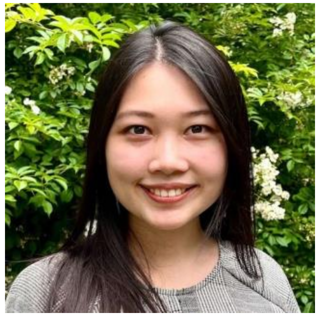
Konstantinos
Karatsenidis



Andy Huynh



Jinqi Lu



Shaolin Xie

Subhadeep Sarkar

Matthaios Olma

BOSTON UNIVERSITY

Microsoft

mongoDB®

DiSC lab