Indexing for Near-Sorted Data

Aneesh Raman

Subhadeep Sarkar

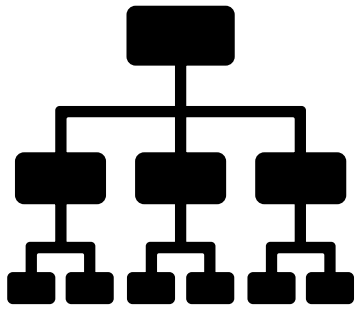
Matthaios Olma

Manos Athanassoulis

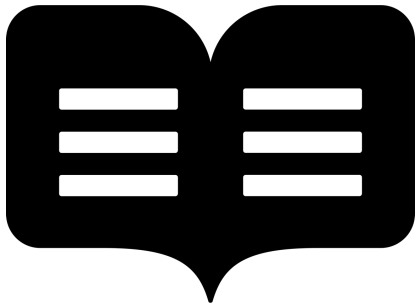
BOSTON UNIVERSITY

DiSC lab

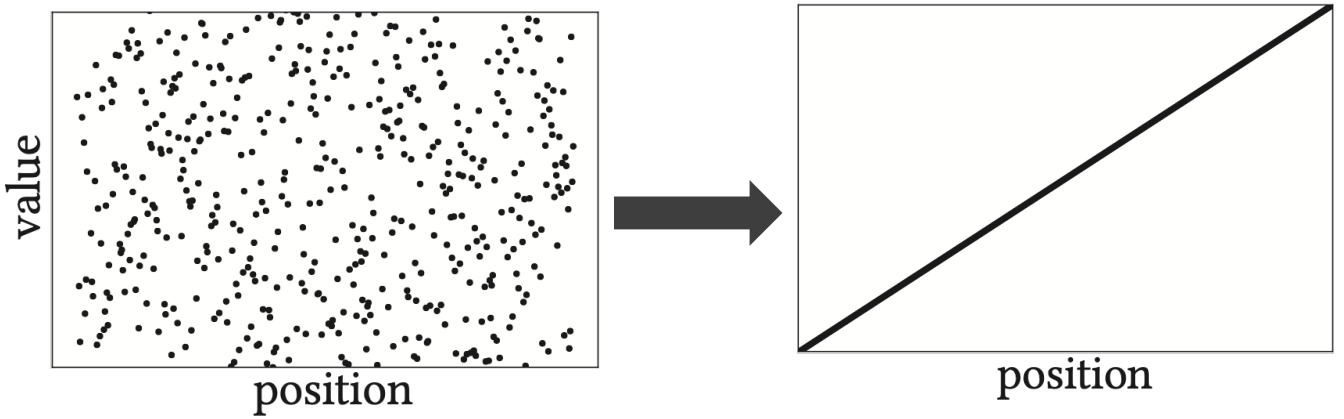
Indexes in Databases



organize data



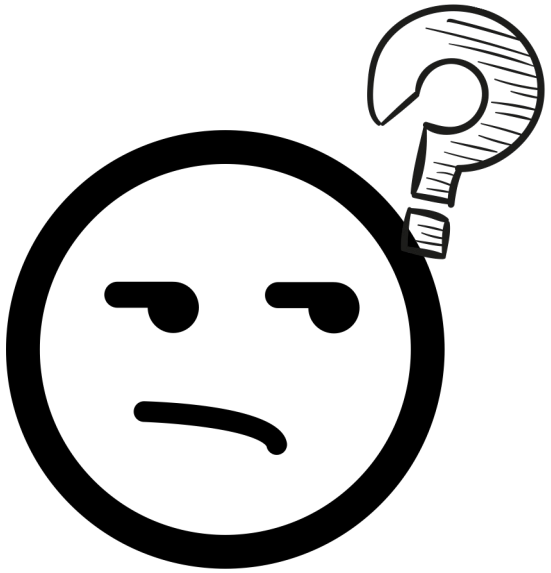
efficient queries



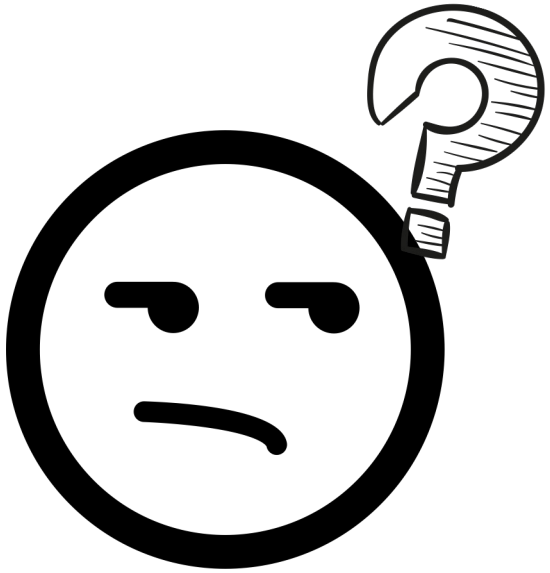
unstructured data

structured data

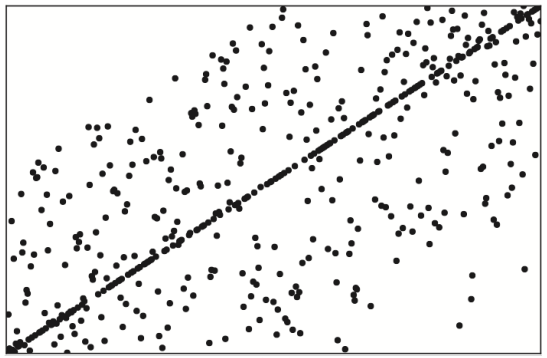
The process of inducing "*sortedness*" to an otherwise unsorted data collection

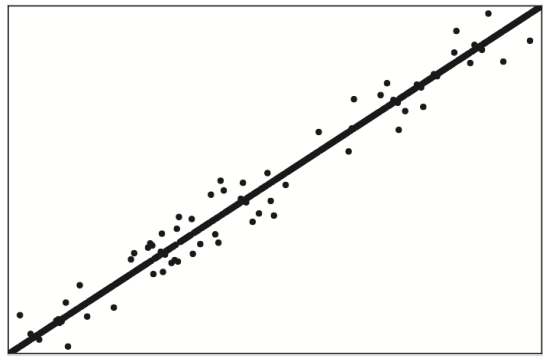


What if data already has some structure?

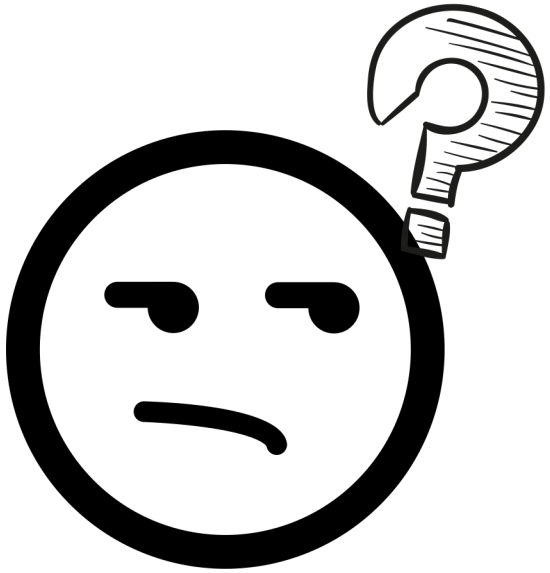


What if data already has some structure?

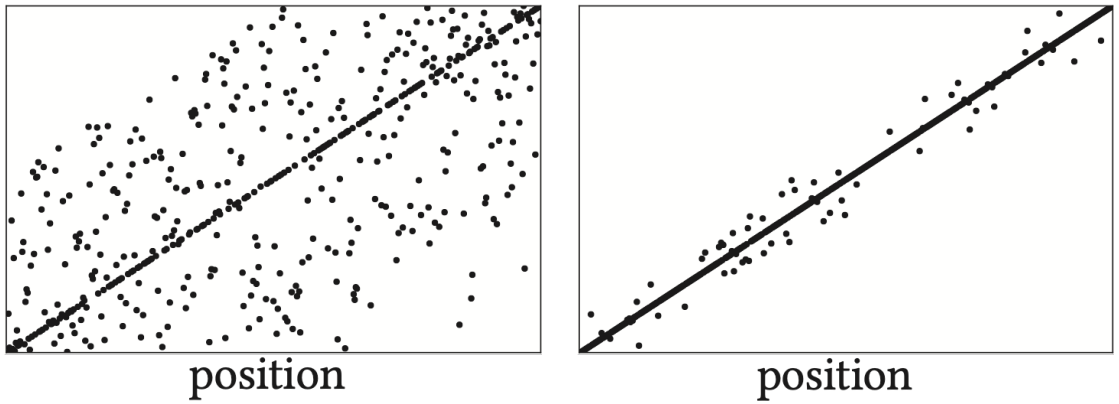




Near-sorted data

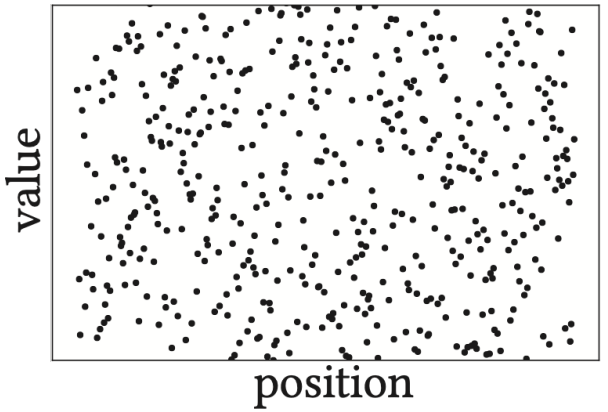


What if data already has some structure?



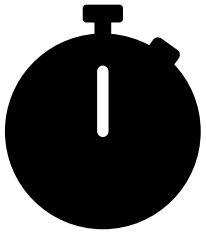
Near-sorted data

$\approx$



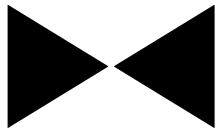
treated same as unstructured data!

Intermediate-Sortedness in Practice

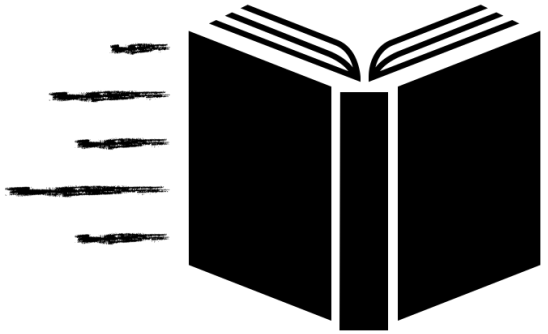


Time Series

Stock market



Join/query

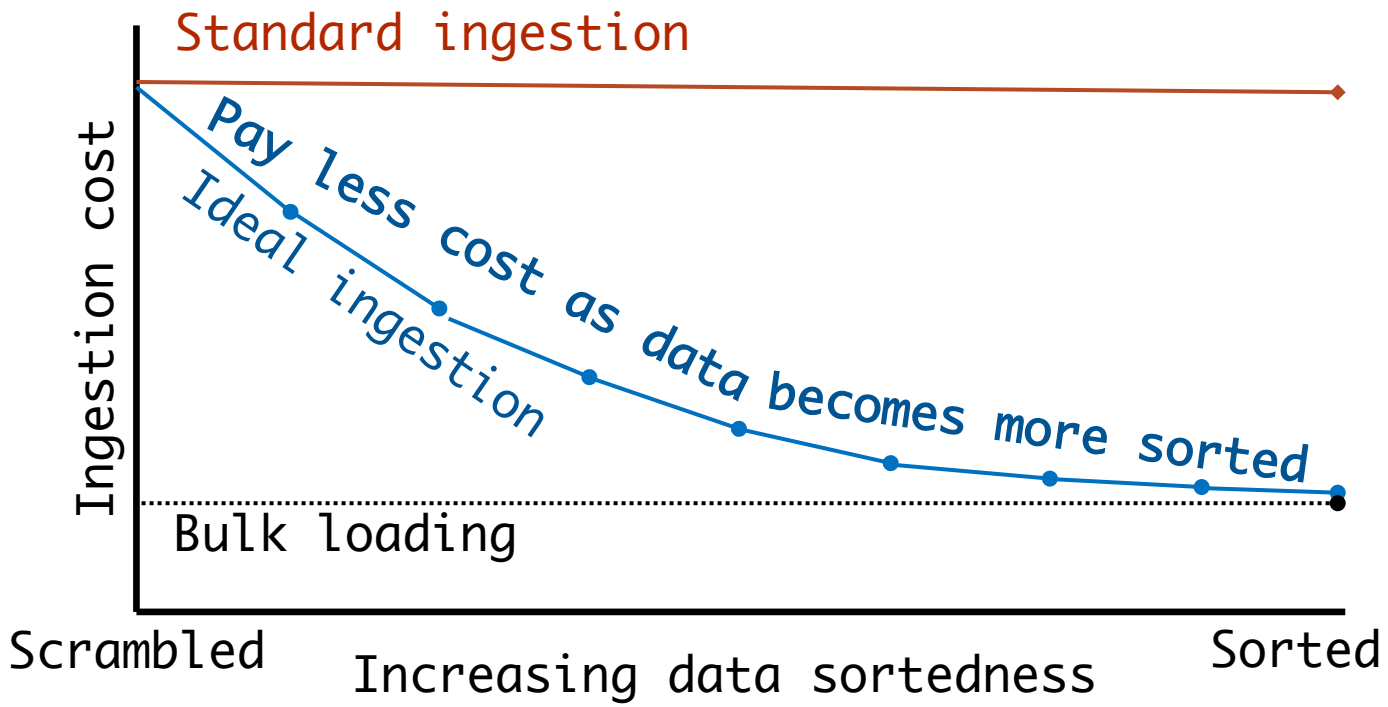


efficient reads

fast writes

classical indexes carry *redundant* effort!

Ideally...


Standard ingestion
Pay less cost as data becomes more sorted
Ideal ingestion
Bulk loading
Ingestion cost
Scrambled
Increasing data sortedness
Sorted

The Sortedness-Aware (SWARE) Paradigm

Sortedness-Aware (SWARE) Paradigm

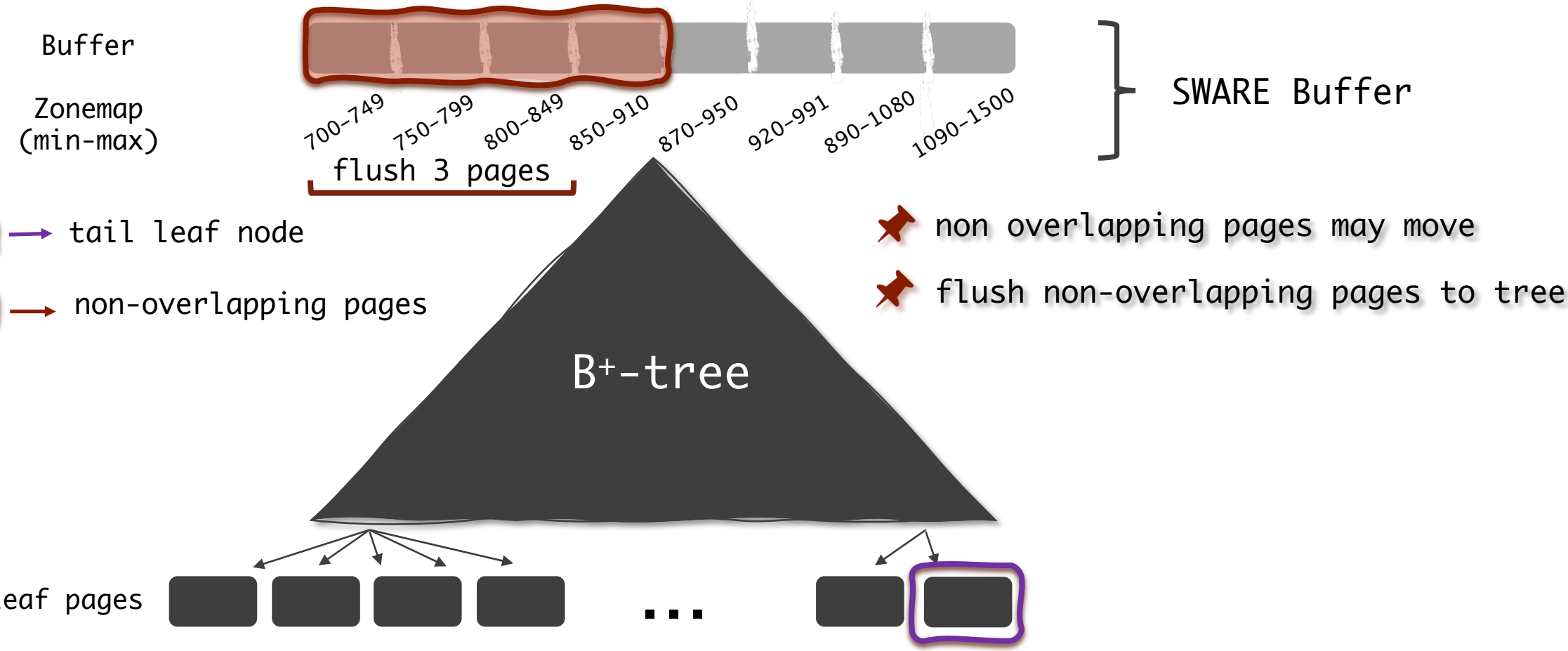
intelligent buffering

opportunistic bulk loading

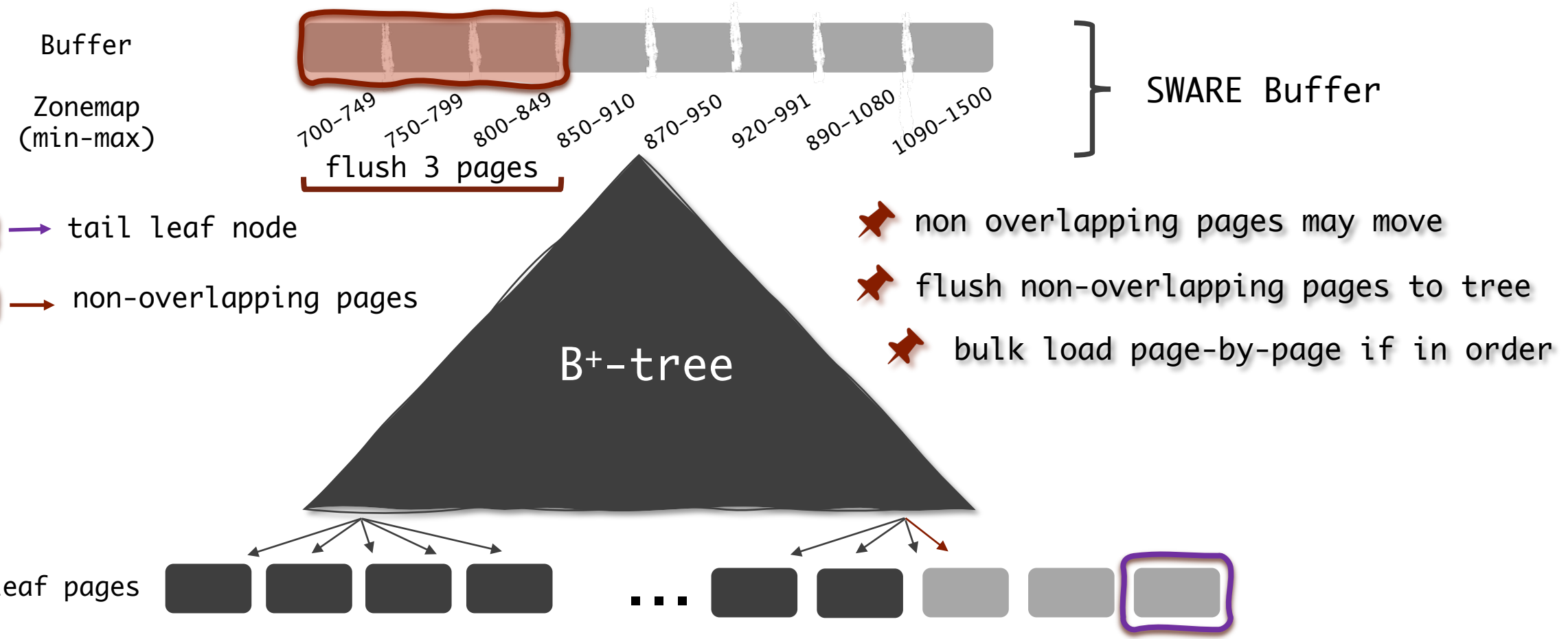
increased fill and split factor

SWARE framework can be applied to any tree-index!

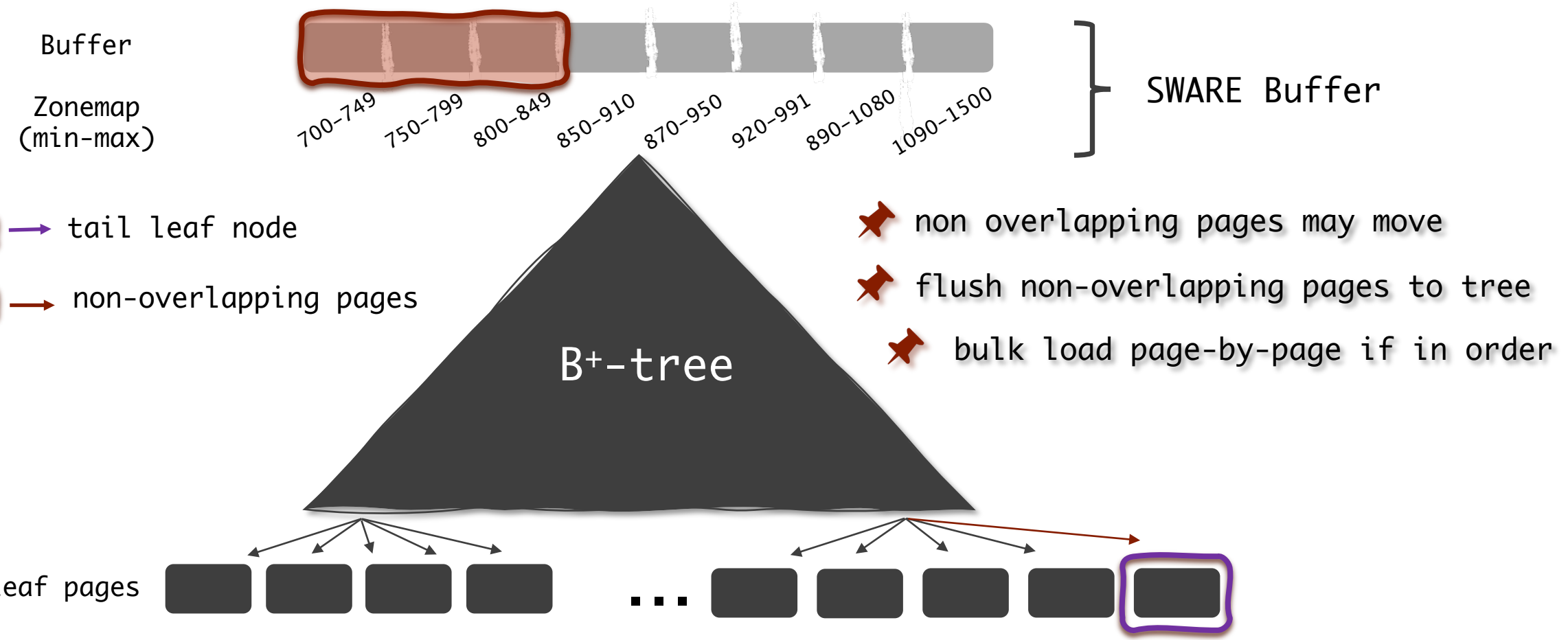
SWARE Ingestions



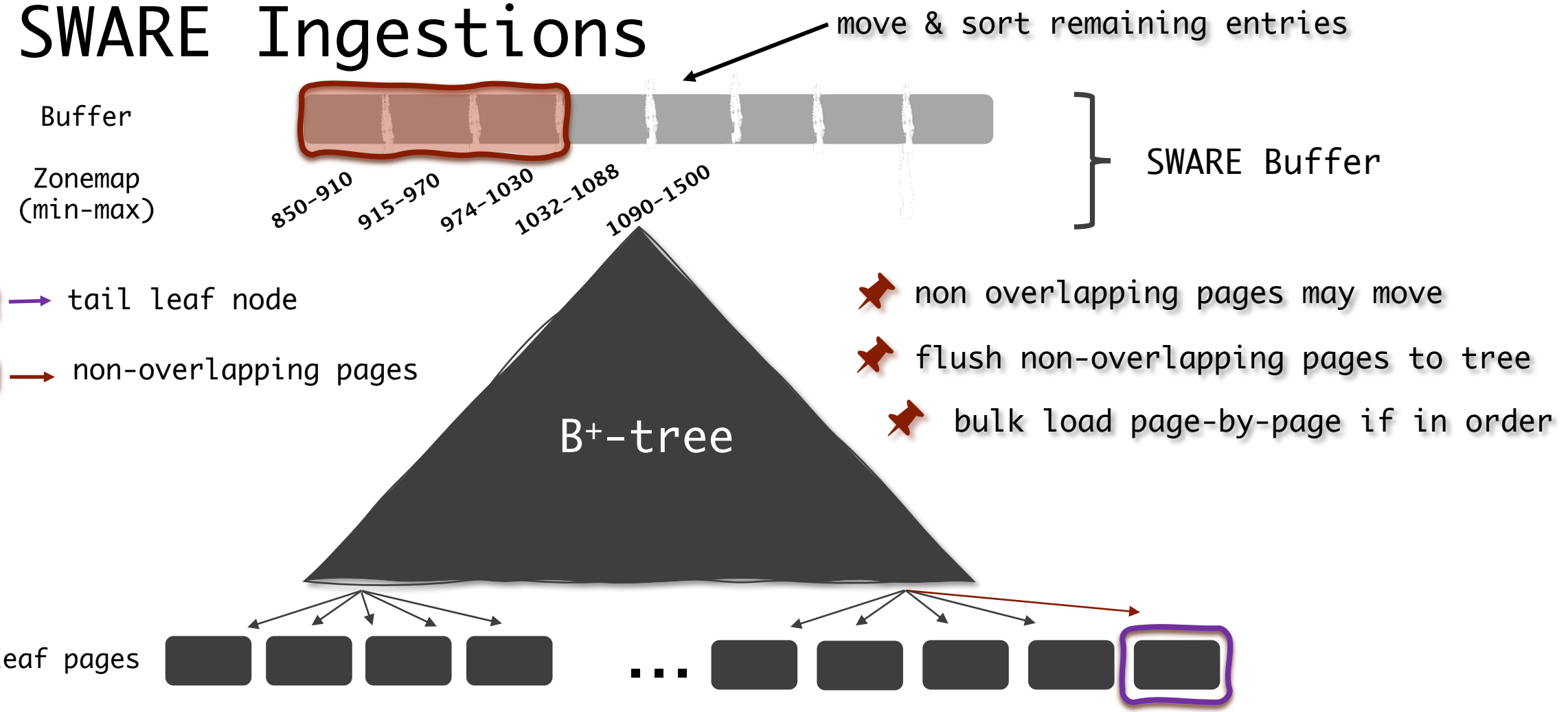
SWARE Ingestions



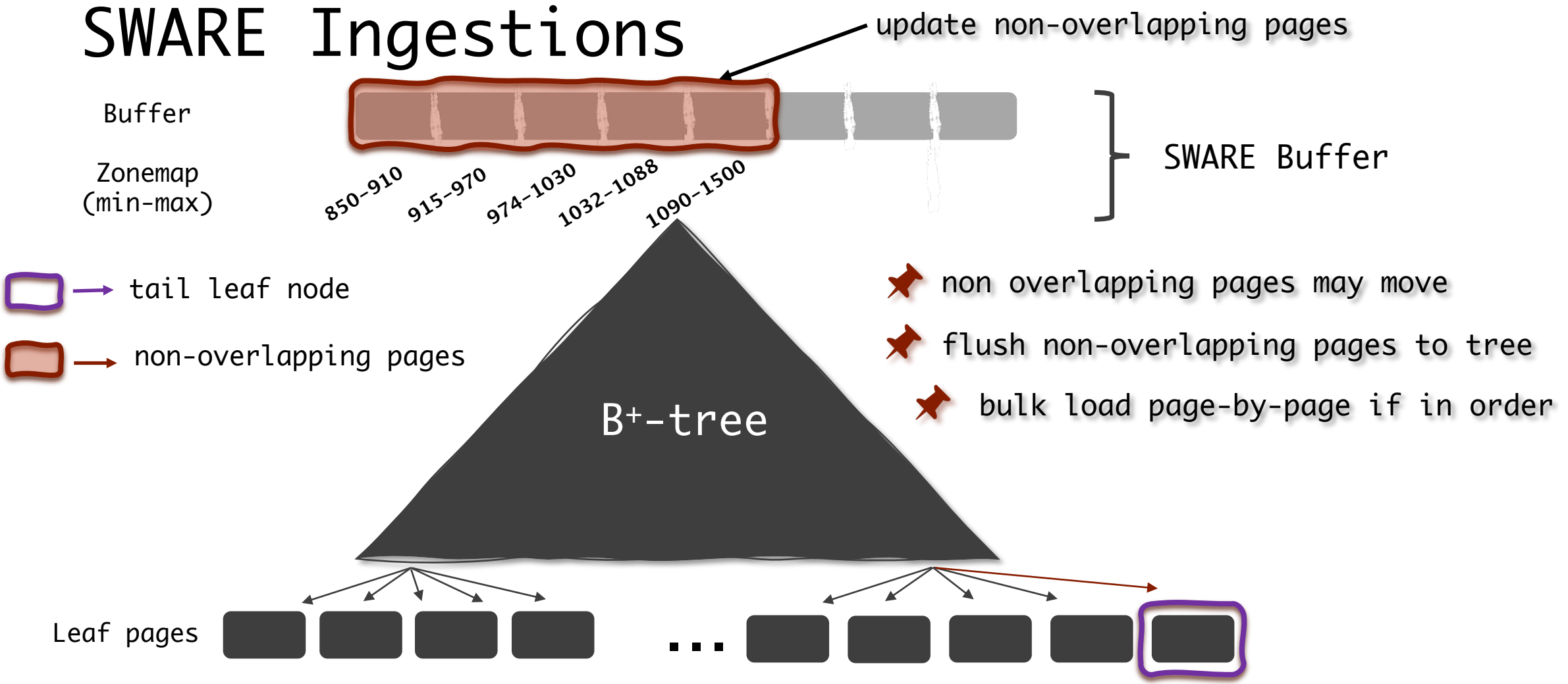
SWARE Ingestions



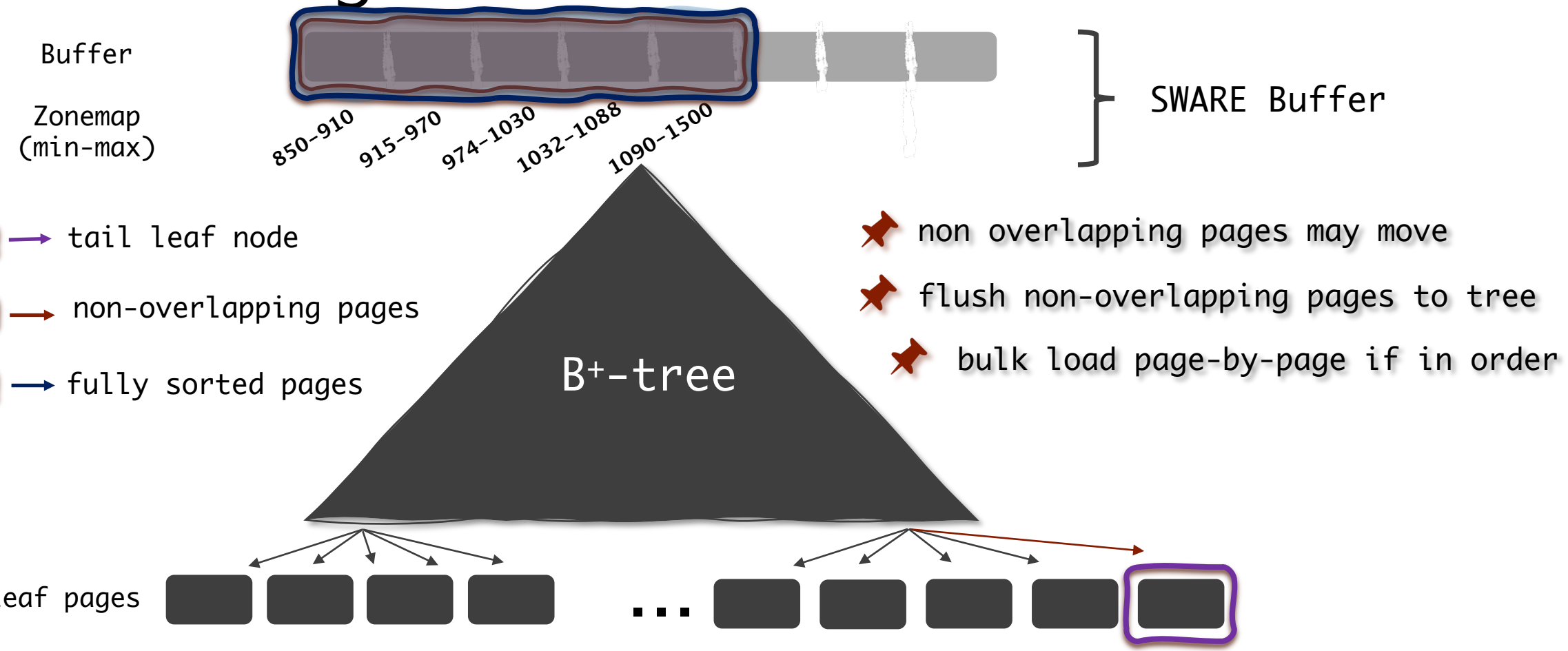
SWARE Ingestions



SWARE Ingestions

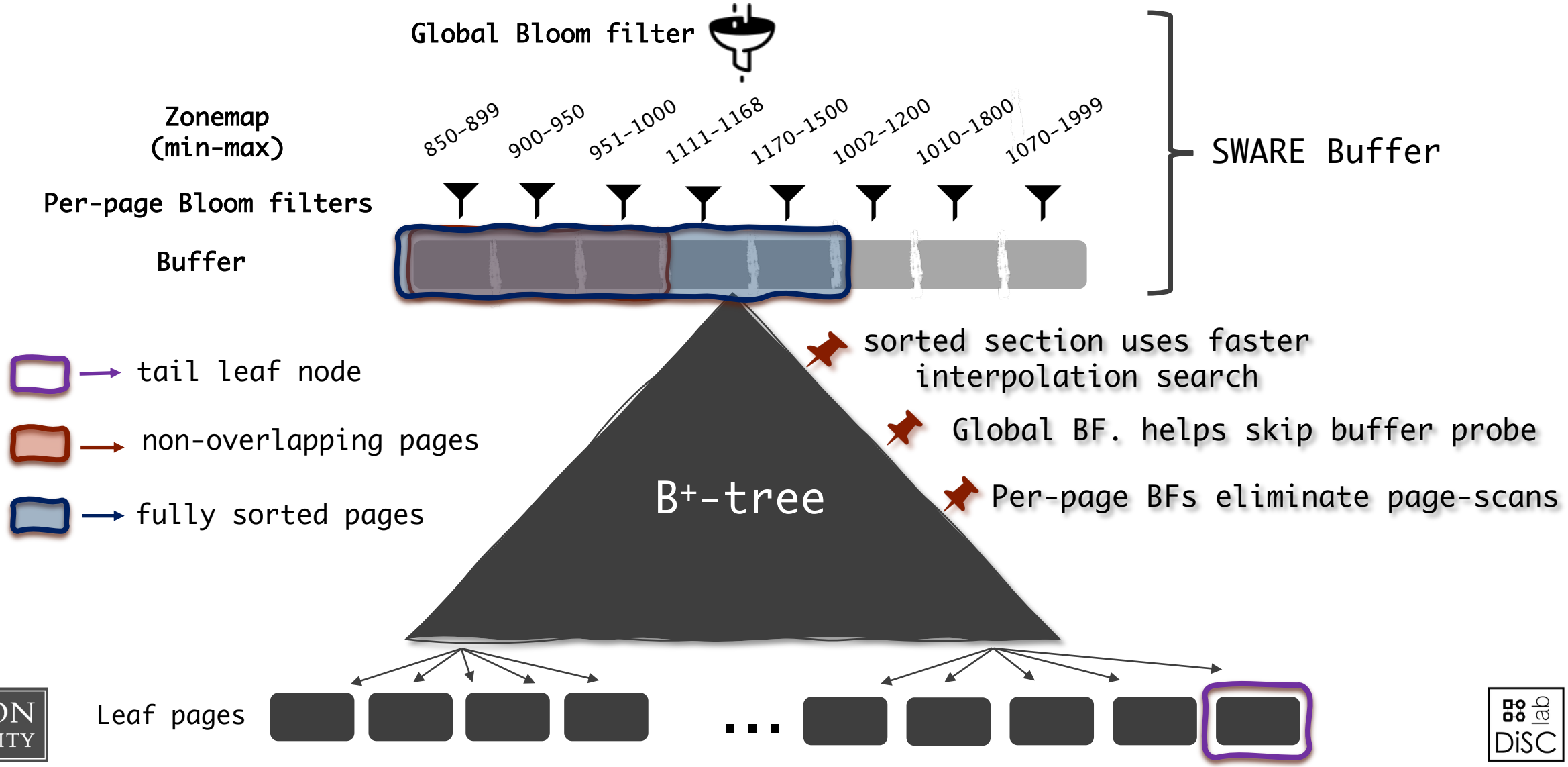


SWARE Ingestions



How do lookups work?

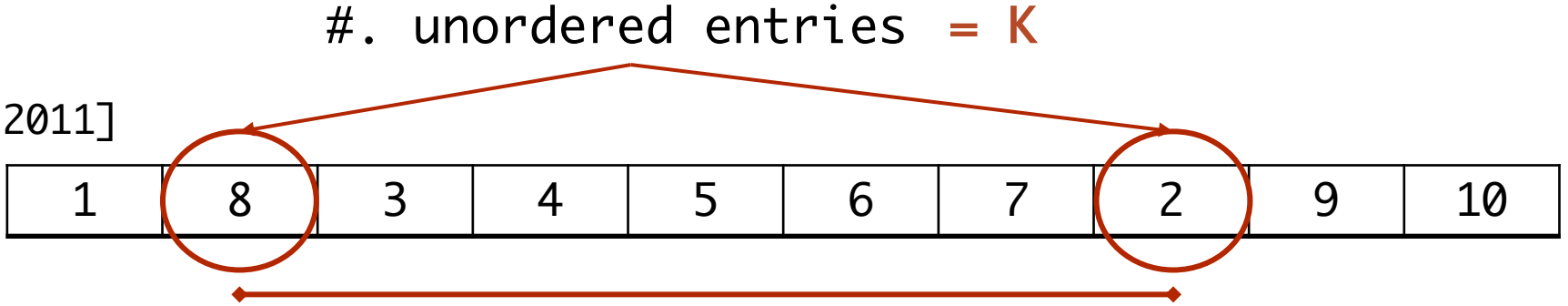
Overall Structure for Queries



How do we evaluate SWARE?

Benchmark on Data Sortedness (TPCTC 2022)

[BenMoshe, ICDT 2011]



Insert Only

Mixed Workloads
(interleaved reads and writes)

Experimental Setup

Metrics:
1. Overall performance (speedup)
2. Raw performance (latency)

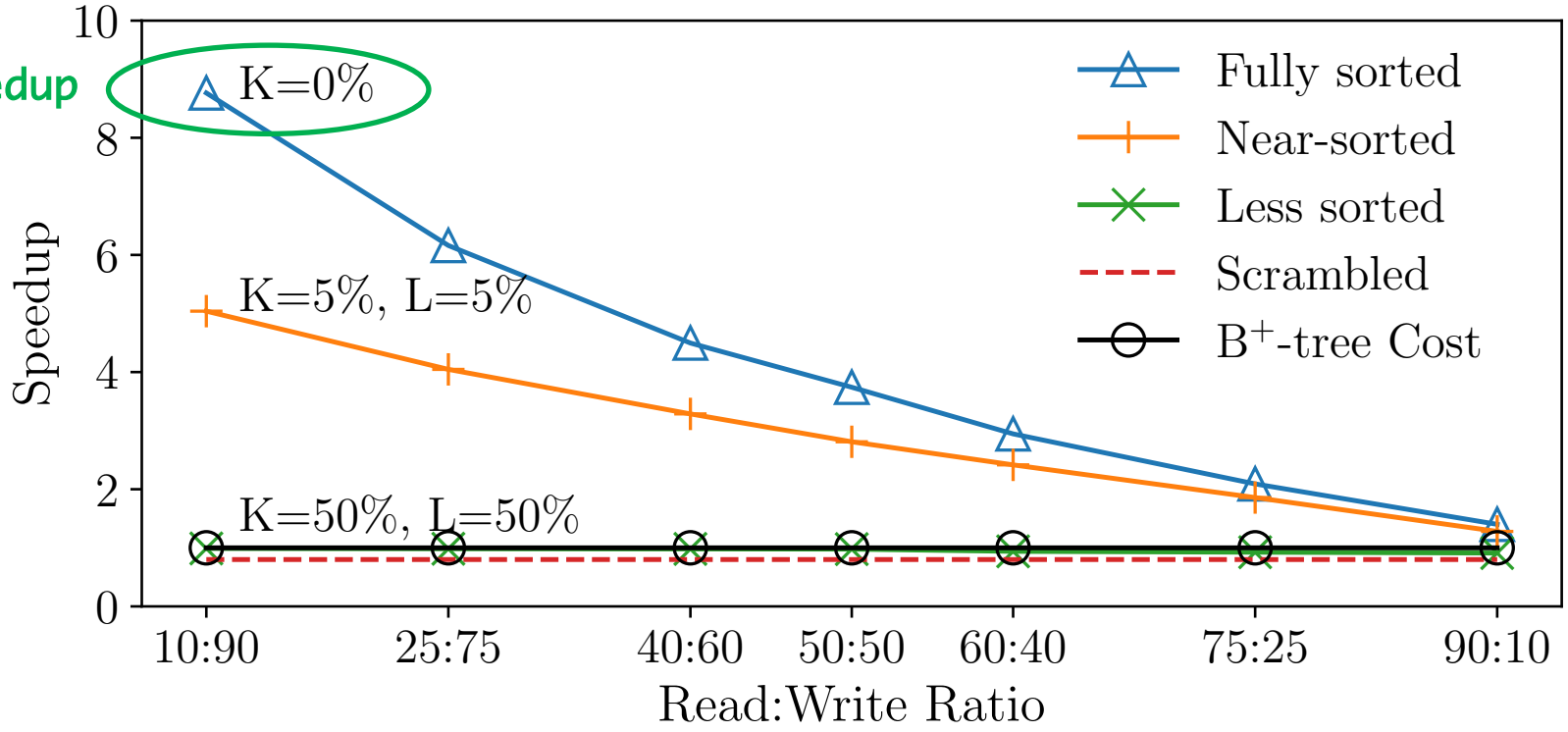
Workload Generator: **BoDS**
1. 500M Integer keys (~ 4GB)
2. Random existing lookups

System Setup:
1. Intel Xeon Gold 5230
2. 2.1GHZ processor w. 20 cores
3. 384GB RAM, 28MB L3 cache

Default Index Setup:
1. Buffer = 40MB; flush <= 50%
2. BFs = 10 BPK; Murmur Hash
3. Split at 80%

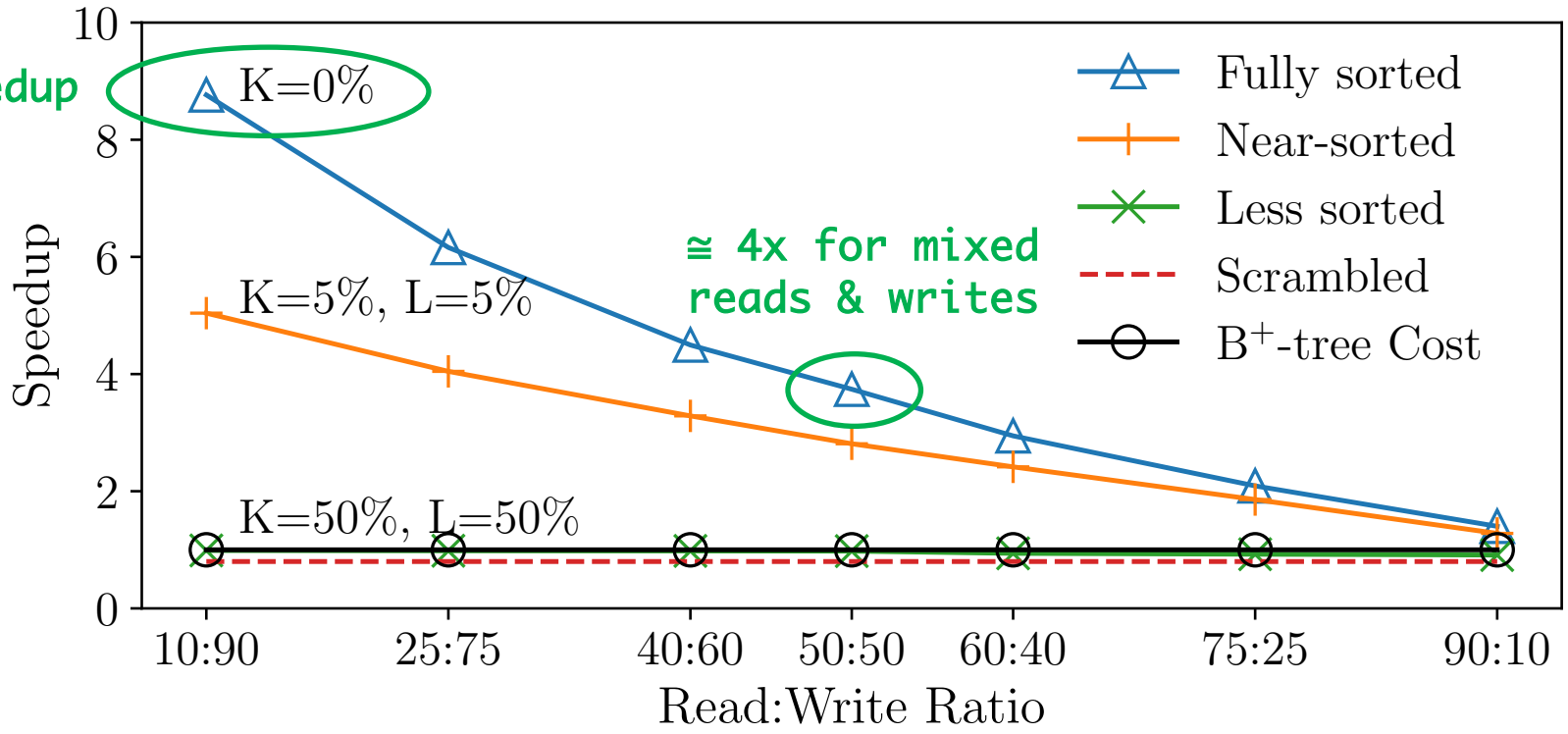
Overall Performance

≅ 9x speedup

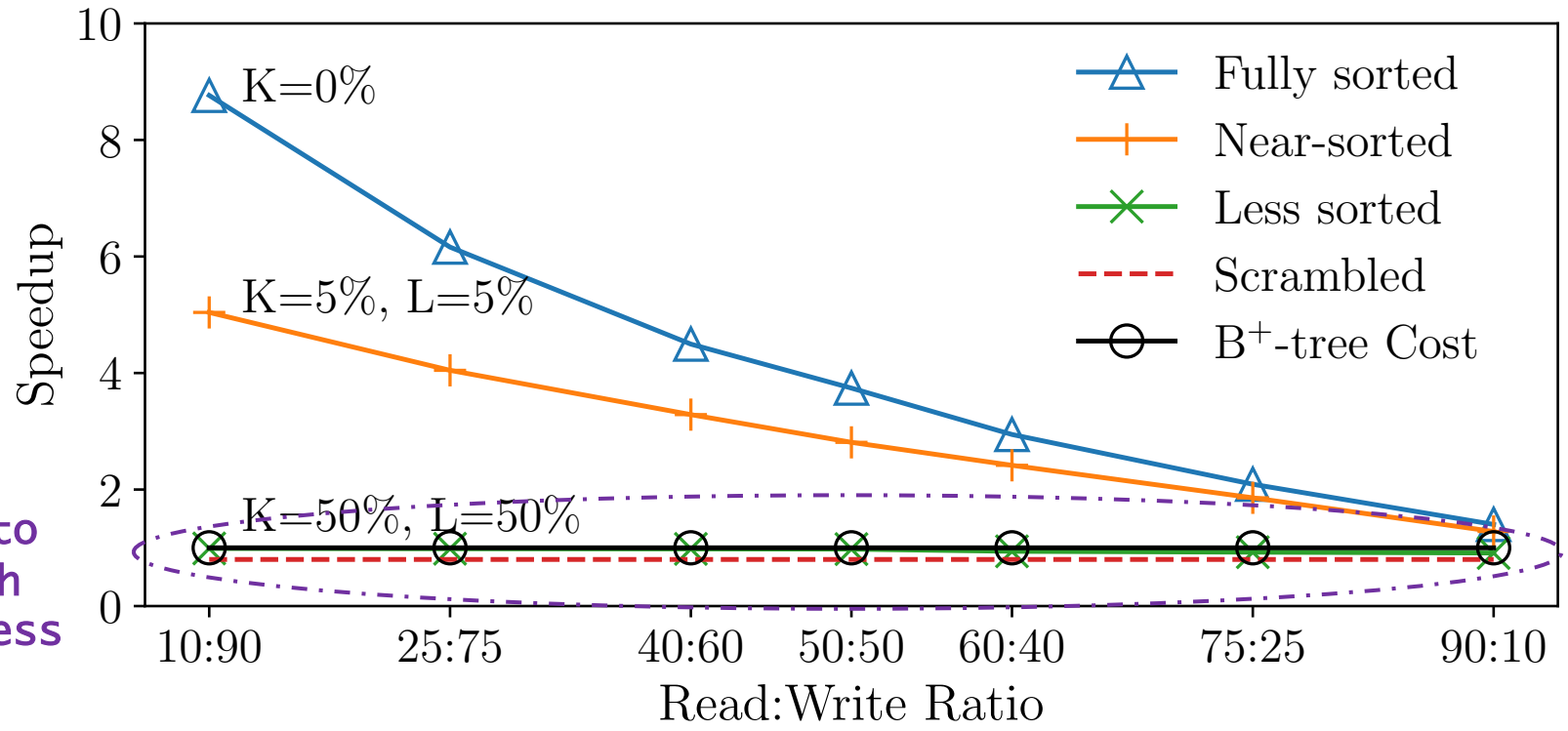


Overall Performance

≅ 9x speedup

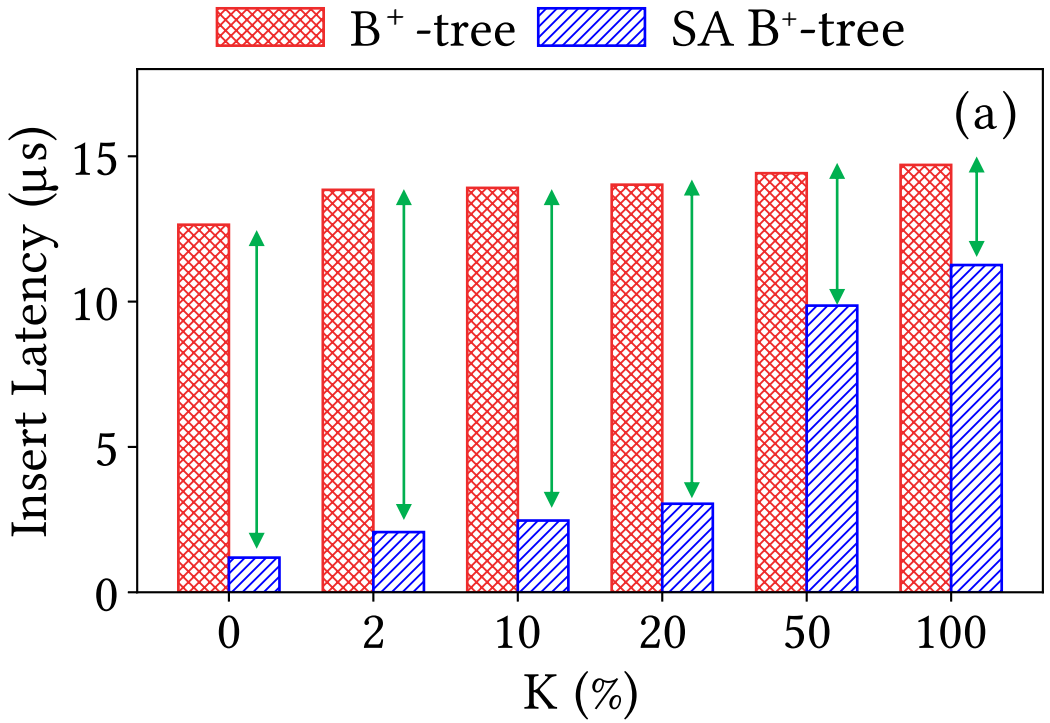


Overall Performance



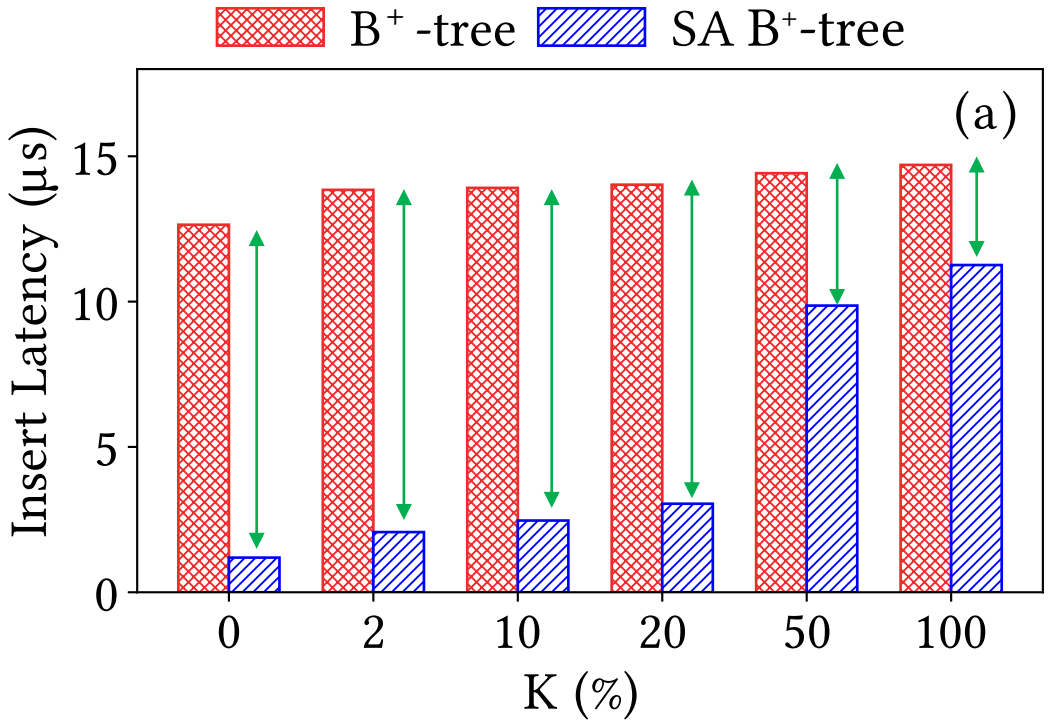
comparable to B+-tree with less sortedness

Raw Ingestion Performance

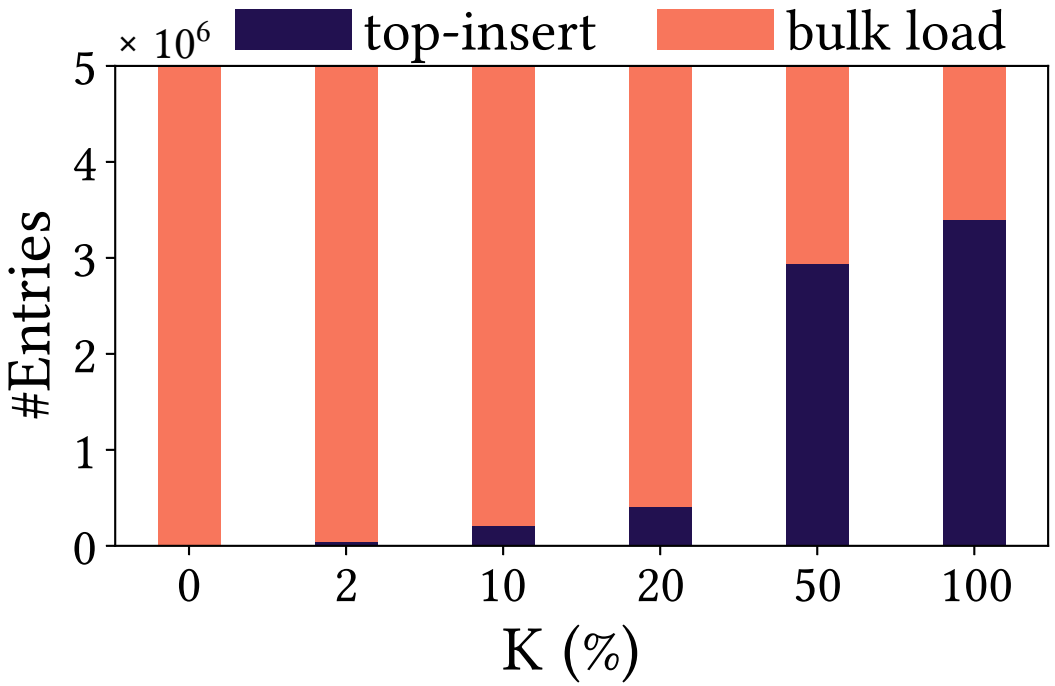


ingestion latency reduced between 27-90%

Raw Ingestion Performance



ingestion latency reduced between 27-90%



bulk loading is maximized
with high data sortedness

Space Efficiency

Sortedness Degree	#. Nodes (#. Internal, #. Leaf)	
	B+ tree	SA B+ tree
Fully Sorted	2.004M (8K, 1.996M)	0.52x
Near-Sorted	1.847M (7K, 1.840M)	0.6x
Less-Sorted	1.878M (4.3K, 1.873M)	1.01x

Space Efficiency

Sortedness Degree	#. Nodes (#. Internal, #. Leaf)	
	B+ tree	SA B+ tree
Fully Sorted	2.004M (8K, 1.996M)	0.52x
Near-Sorted	1.847M (7K, 1.840M)	0.6x
Less-Sorted	1.878M (4.3K, 1.873M)	1.01x

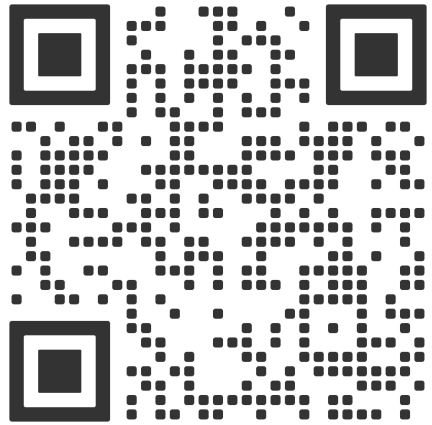
increased fill/split factor
helps reduce memory footprint

Summary

Identify “sortedness” as a resource

Smart buffering + bulk index appends
= faster ingestion

Works well with write-heavy or mixed
read-write workloads



Thank You!

: aneeshr@bu.edu

: ramananeesh.com/

BOSTON UNIVERSITY

DiSC lab