Sortedness-Aware Indexing

Aneesh Raman
aneeshr@bu.edu

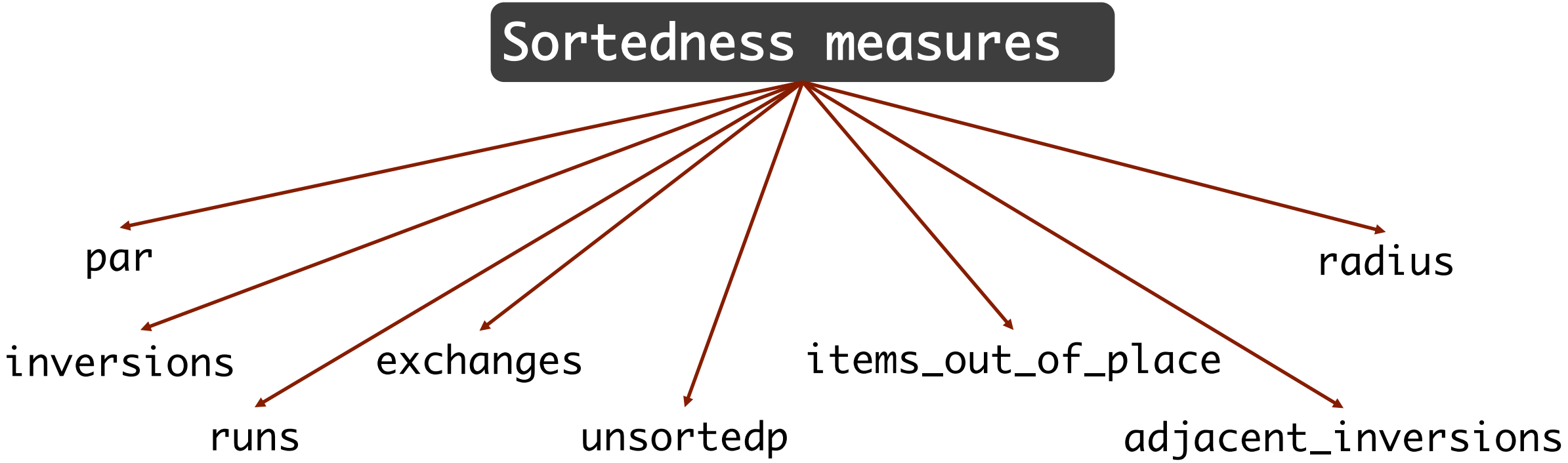
BOSTON UNIVERSITY

DiSC lab

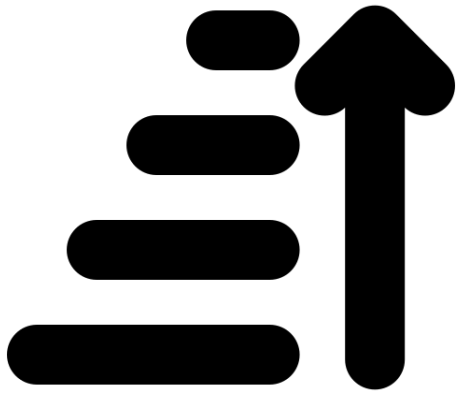
Sortedness

Refers to the ***structure*** or ***order*** in data

Quantifying Sortedness



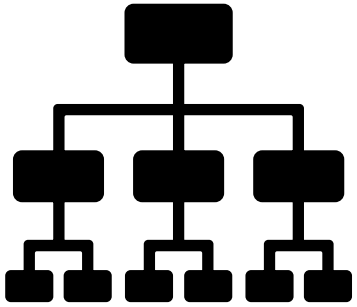
Research Focus



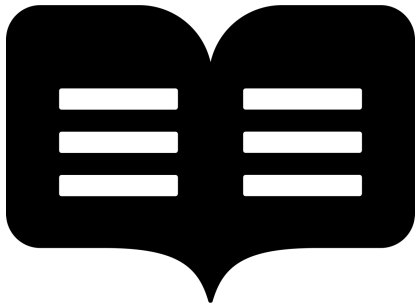
sorting algos. have
adapted to pre-sortedness

can other components of a
data system exploit
intrinsic data ordering?

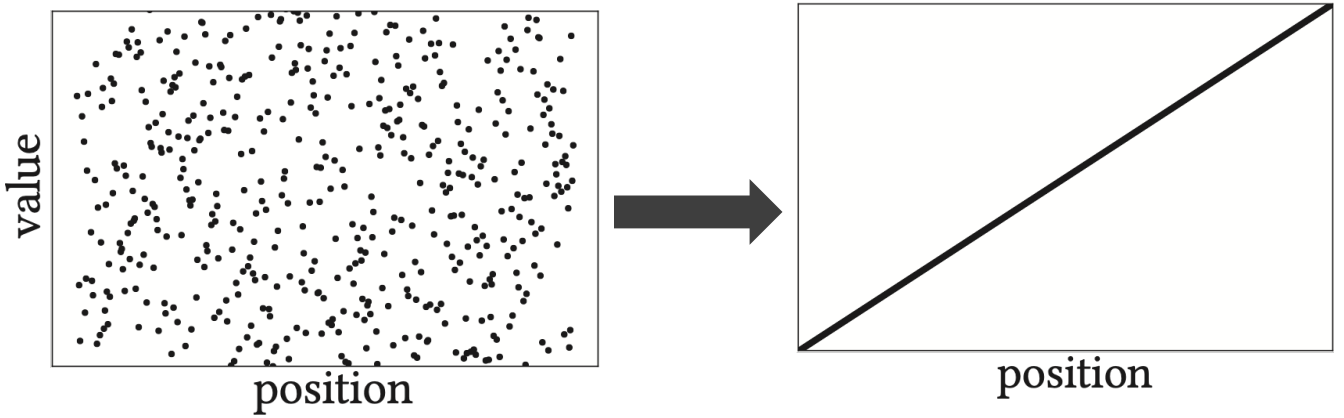
Indexes in Databases



organize data



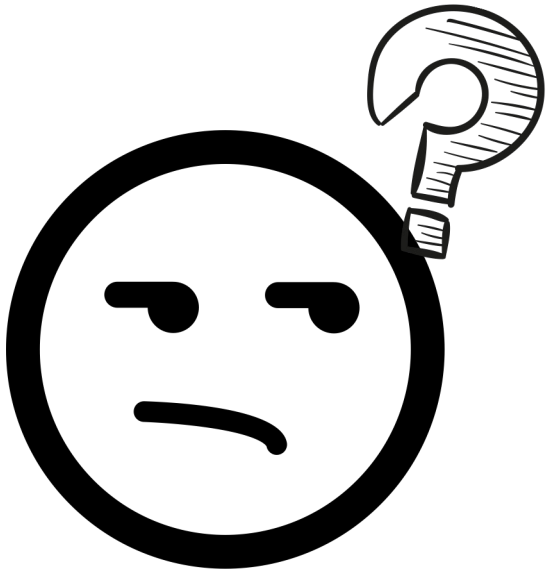
efficient queries



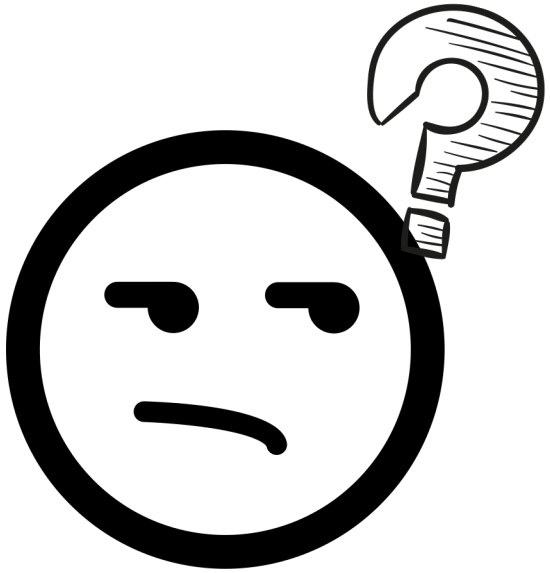
unstructured data

structured data

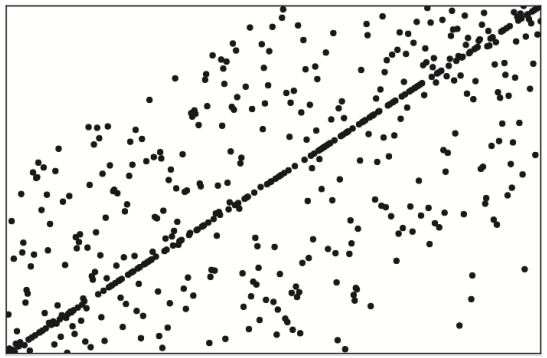
The process of inducing "*sortedness*" to an otherwise unsorted data collection

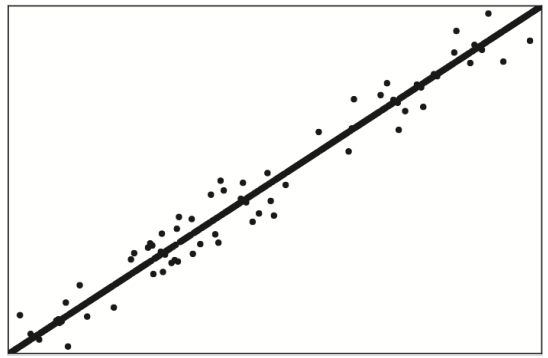


What if we already have some structure?

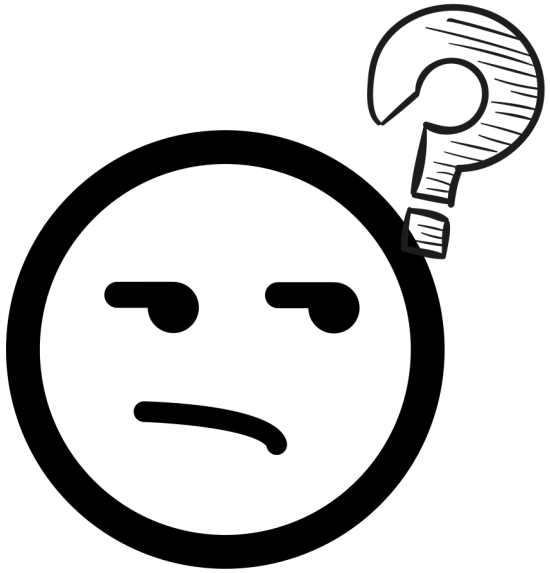


What if we already have some structure?

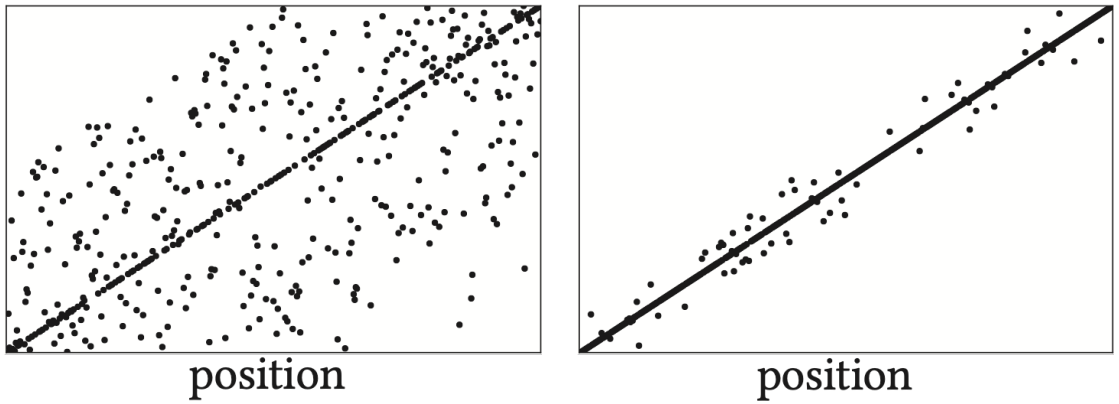




Near-sorted data

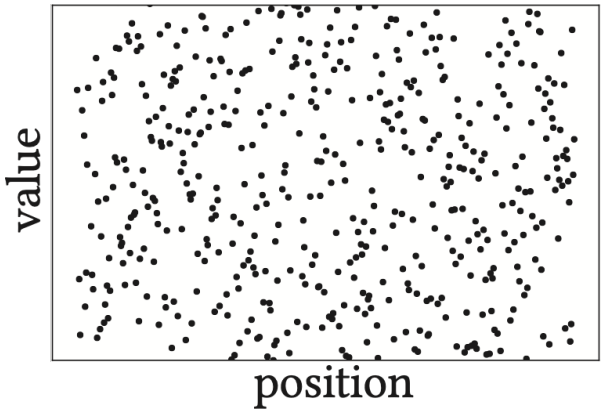


What if we already have some structure?



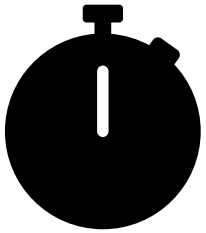
Near-sorted data

$\approx$



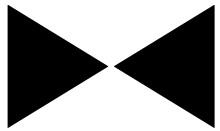
treated same as unstructured data!

Intermediate-Sortedness in Practice

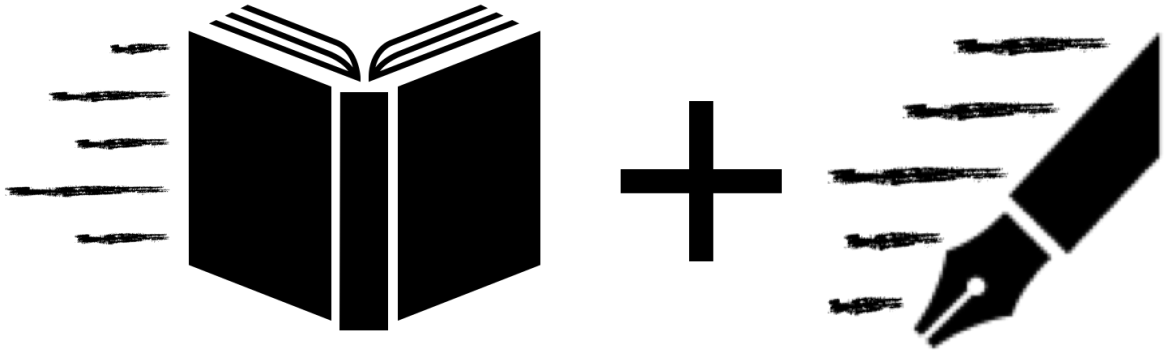


Time Series

Stock market



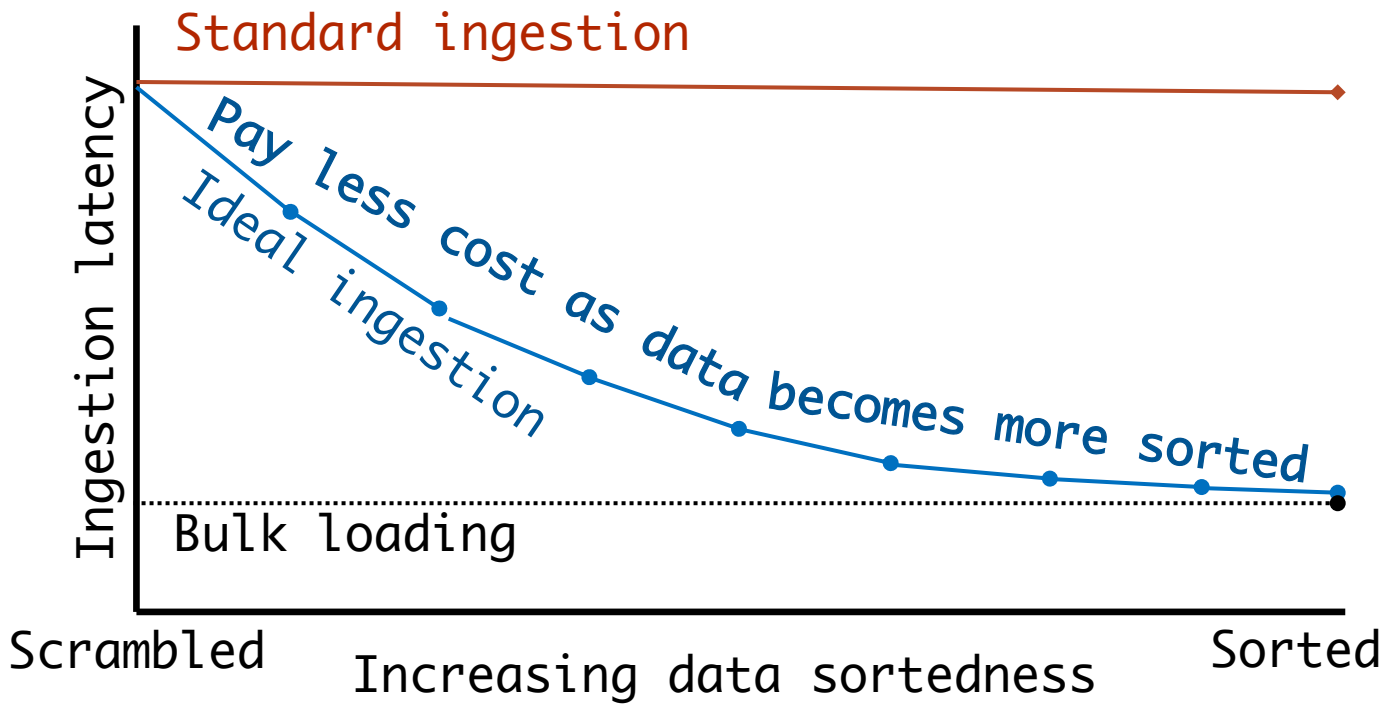
Join/query



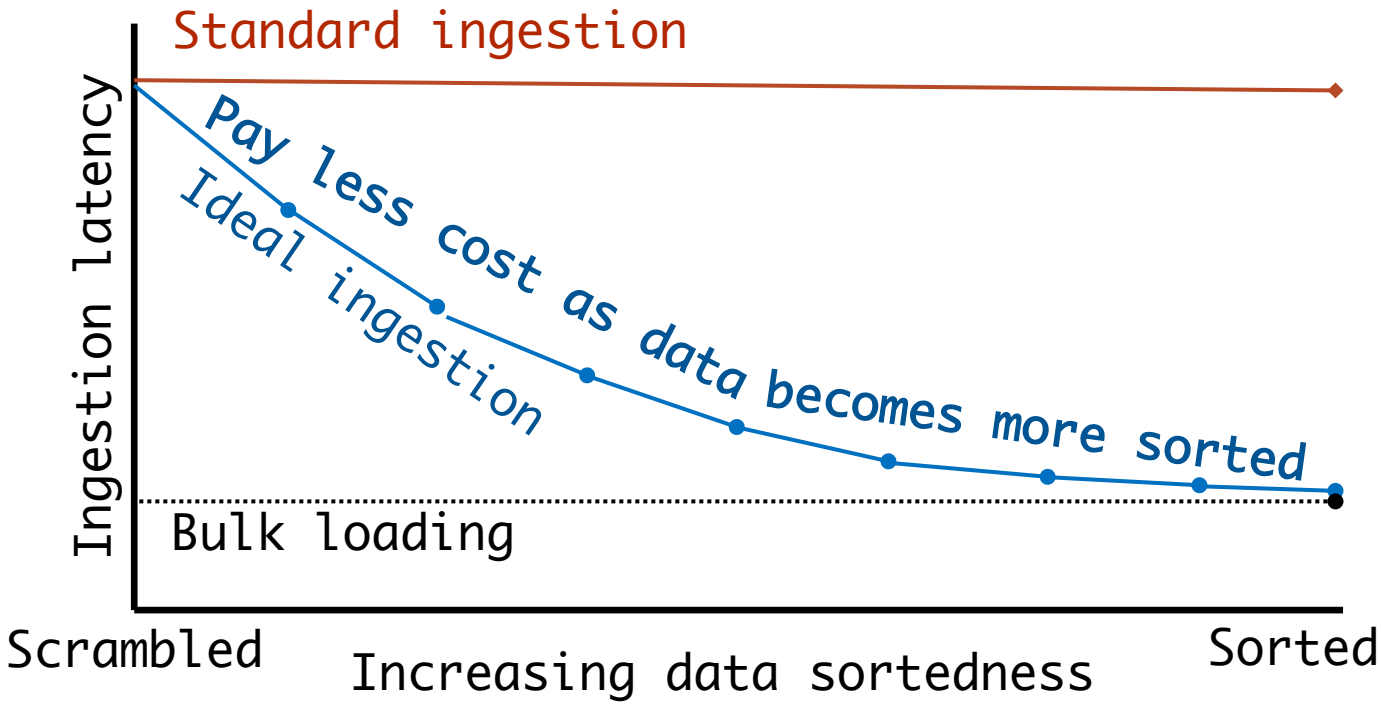
efficient reads + fast writes

classical indexes carry *redundant* effort!

In an Ideal Tree...



In an Ideal Tree...

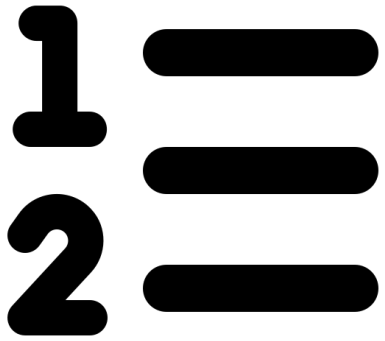


Is this possible?

1. Unexplored study with sortedness
2. Lack of testing framework
3. No such existing index design

The Benchmark on Data Sortedness (BoDS)

Benchmark on Data Sortedness (BoDS)



Variable Sortedness
Data Generator

+

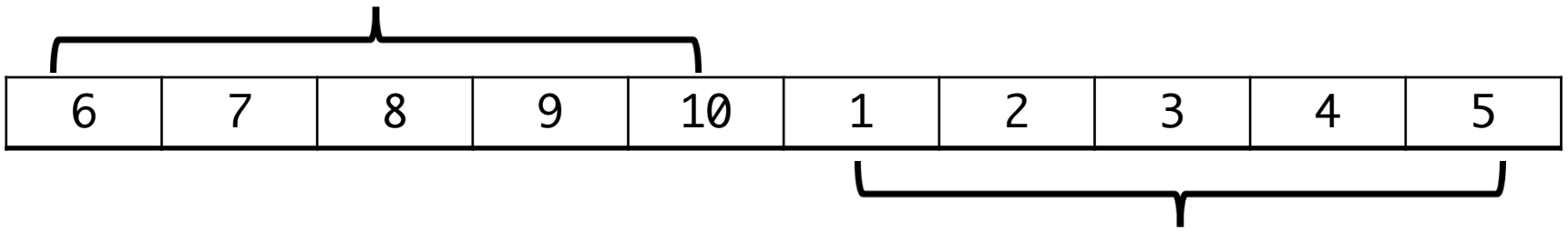
Benchmarking
Suite

Quantifying Data Sortedness

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

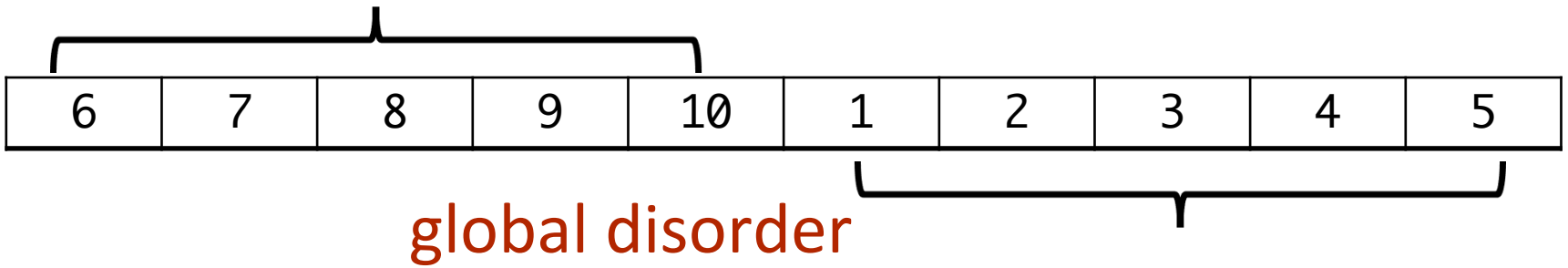
Quantifying Data Sortedness

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



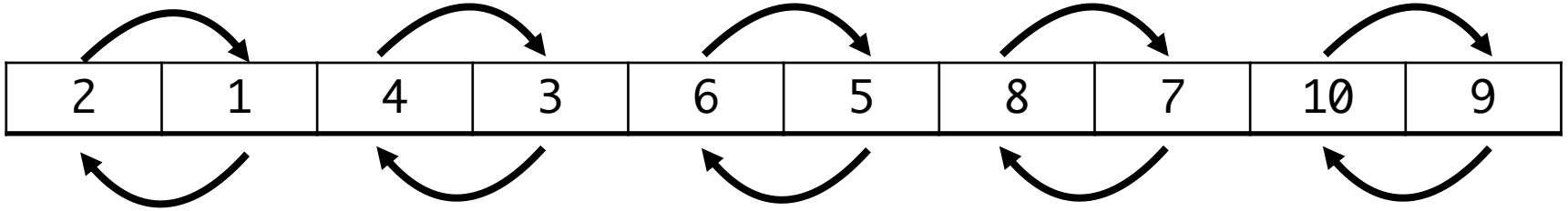
Quantifying Data Sortedness

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



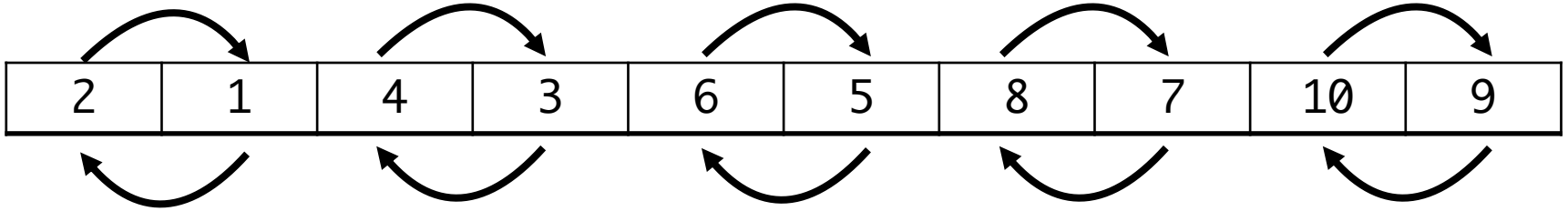
Quantifying Data Sortedness

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



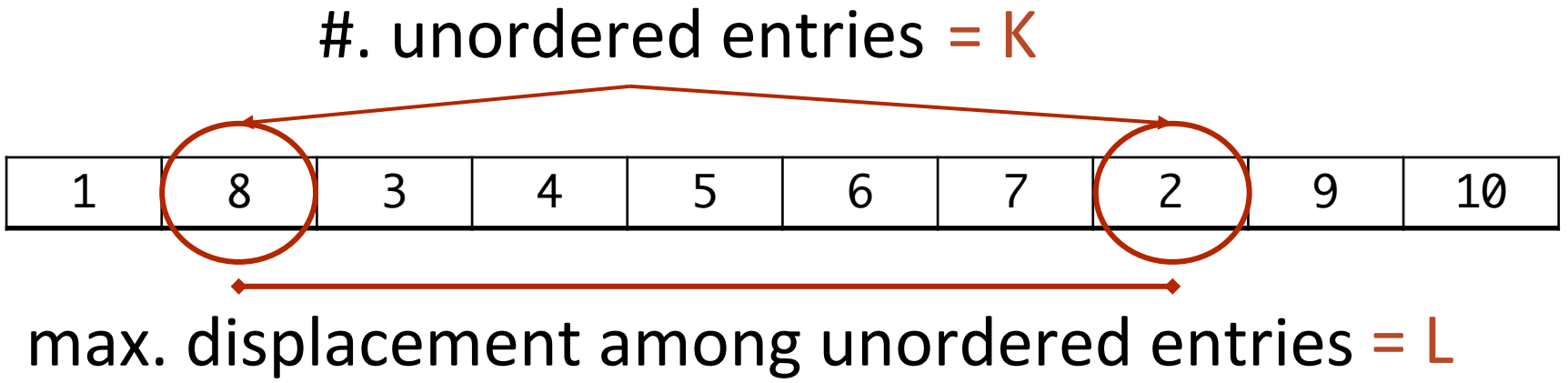
Quantifying Data Sortedness

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



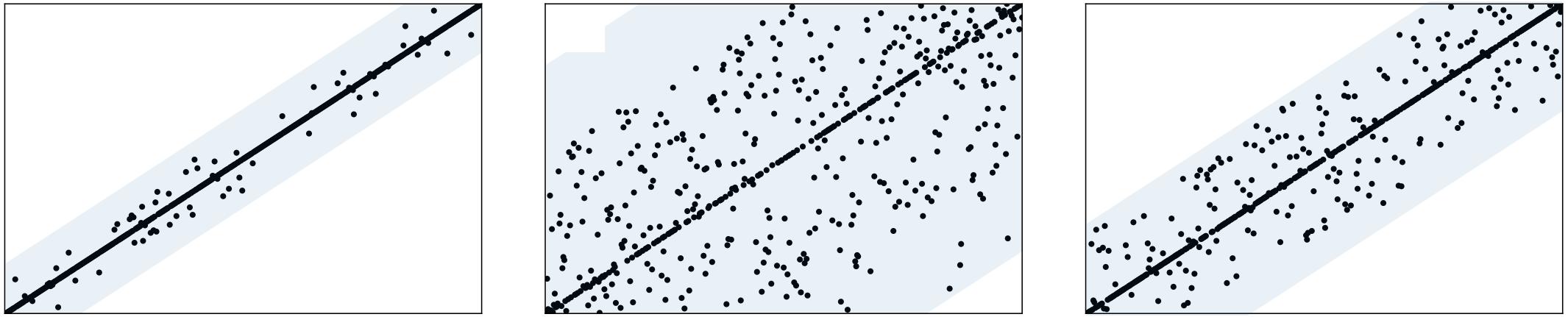
local disorder

(K, L)-Sortedness Metric

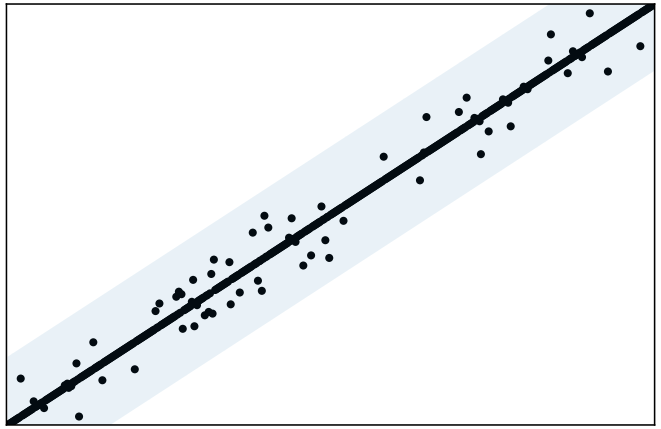


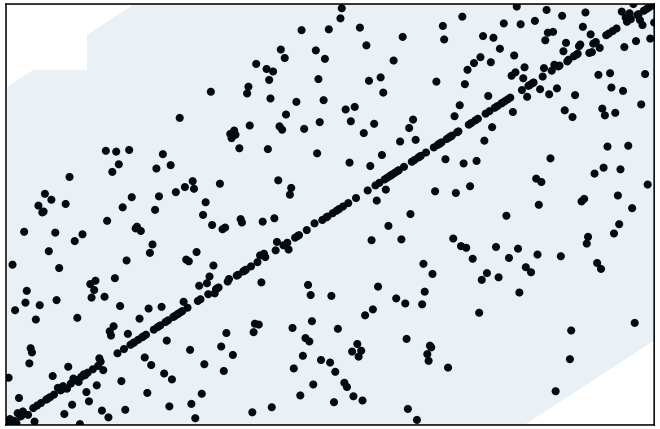
[BenMoshe, ICDT 2011]

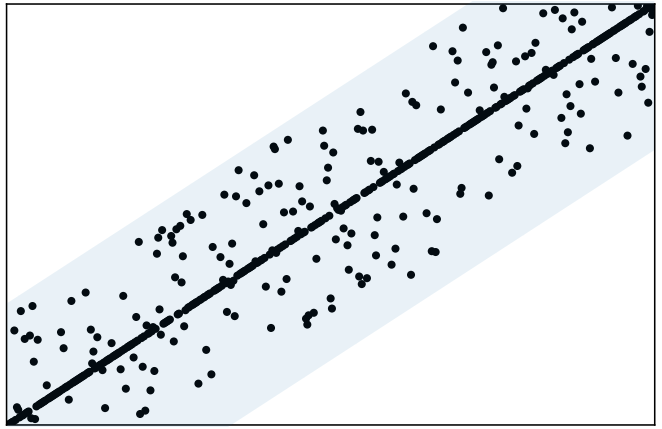
Differently Sorted Data

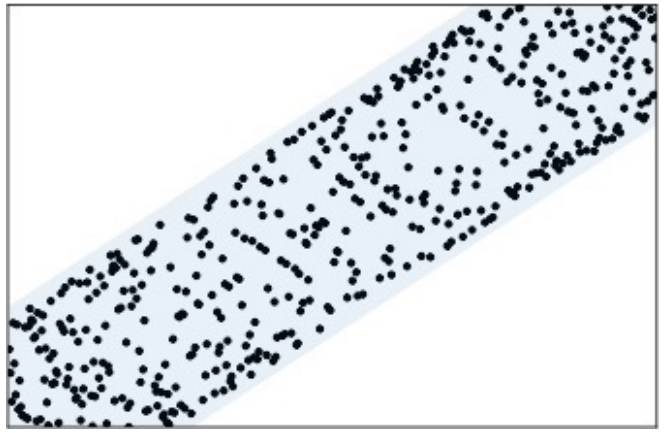


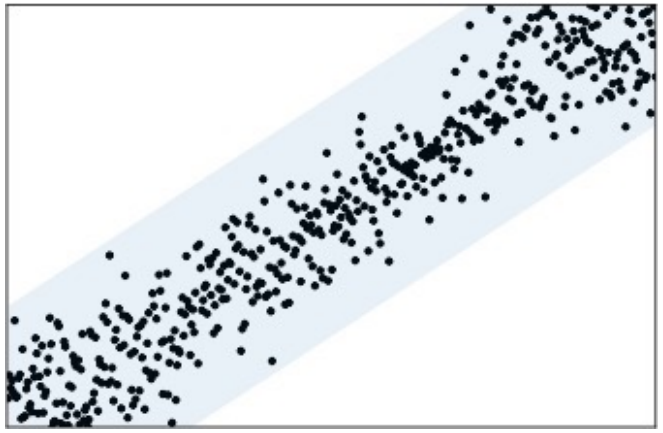
Is (K,L) Enough?











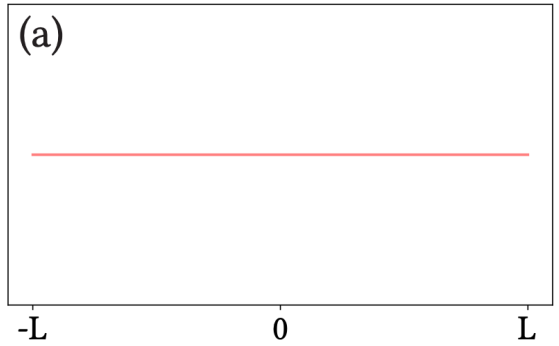
What if we want to distribute the unordered entries differently?

Displacement (L) Distribution

$B(\alpha, \beta)$ **bounded between** $[-L, L]$

Displacement (L) Distribution

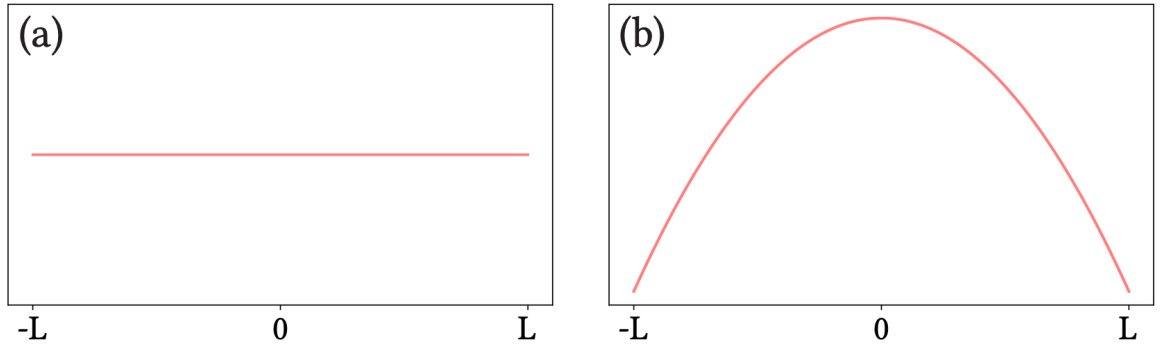
$B(\alpha, \beta)$ **bounded between** $[-L, L]$



$\alpha = \beta = 1$

Displacement (L) Distribution

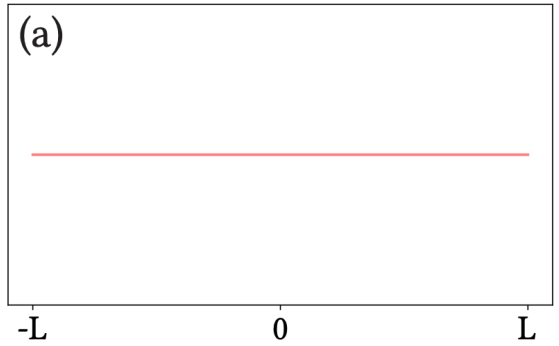
$B(\alpha, \beta)$ **bounded between** $[-L, L]$



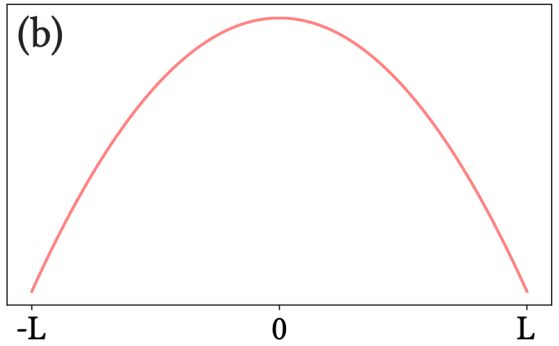
$\alpha = \beta = 1$ $\alpha = \beta = 2$

Displacement (L) Distribution

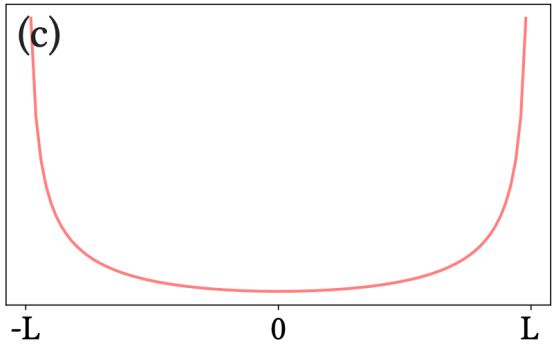
$B(\alpha, \beta)$ **bounded between** $[-L, L]$



$\alpha = \beta = 1$



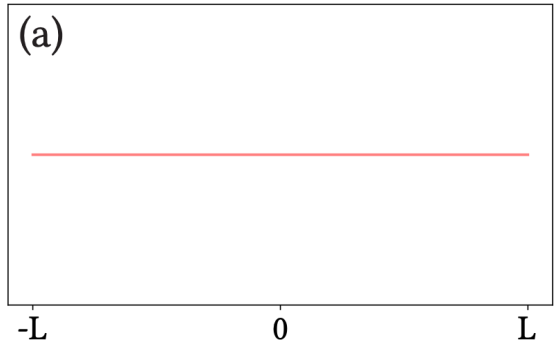
$\alpha = \beta = 2$



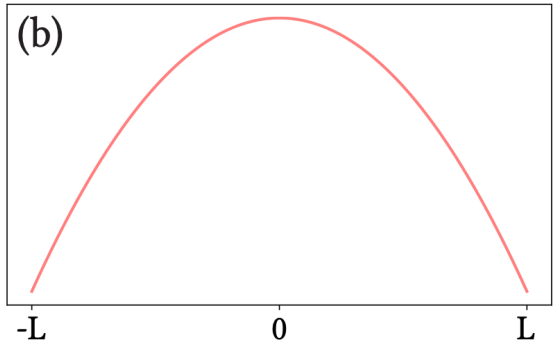
$\alpha = \beta = 0.5$

Displacement (L) Distribution

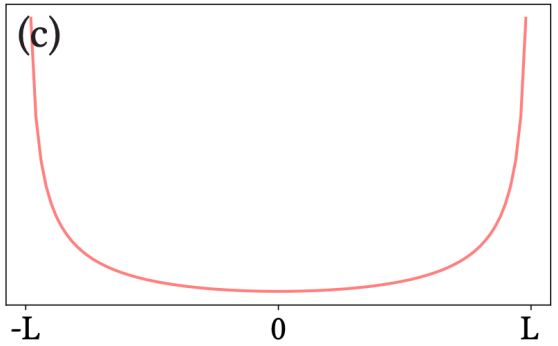
$B(\alpha, \beta)$ **bounded between** $[-L, L]$



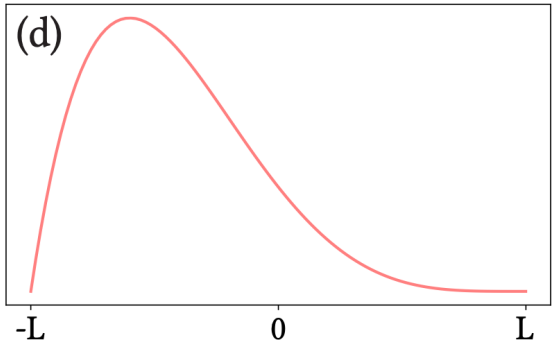
$\alpha = \beta = 1$



$\alpha = \beta = 2$



$\alpha = \beta = 0.5$



$\alpha = 2, \beta = 5$

Supported Workloads

Type	Workload	Data Loading		Operations	
		Method	% of data	R-W ratio	% of data
Insert only	A	Bulk loading	100%	-	-
Mixed reads & writes					

Supported Workloads

Type	Workload	Data Loading		Operations	
		Method	% of data	R-W ratio	% of data
Insert only	A	Bulk loading	100%	-	-
	B	Individual inserts	100%	-	-
Mixed reads & writes					

Supported Workloads

Type	Workload	Data Loading		Operations	
		Method	% of data	R-W ratio	% of data
Insert only	A	Bulk loading	100%	-	-
	B	Individual inserts	100%	-	-
Mixed reads & writes	C	-	0%	17%-83%	100%

Supported Workloads

Type	Workload	Data Loading		Operations	
		Method	% of data	R-W ratio	% of data
Insert only	A	Bulk loading	100%	-	-
	B	Individual inserts	100%	-	-
Mixed reads & writes	C	-	0%	17%-83%	100%
	D	Bulk loading	80%	50%-50%	20%

Supported Workloads

Type	Workload	Data Loading		Operations	
		Method	% of data	R-W ratio	% of data
Insert only	A	Bulk loading	100%	-	-
	B	Individual inserts	100%	-	-
Mixed reads & writes	C	-	0%	17%-83%	100%
	D	Bulk loading	80%	50%-50%	20%
	E	Individual inserts	80%	50%-50%	20%

Supported Workloads

Type	Workload	Data Loading		Operations	
		Method	% of data	R-W ratio	% of data
Insert only	A	Bulk loading	100%	-	-
	B	Individual inserts	100%	-	-
Mixed reads & writes	C	-	0%	17%-83%	100%
	D	Bulk loading	80%	50%-50%	20%
	E	Individual inserts	80%	50%-50%	20%

Benchmarking Action

Metrics:

- Ingestion latency
- Overall operational latency

Benchmarking Action

Metrics:

- Ingestion latency
- Overall operational latency

Data Setup:

- 16M K-V pairs (~ 4GB)
- Key = 4B, Payload = 252B

Benchmarking Action

Metrics:

- Ingestion latency
- Overall operational latency

System Setup:

- AWS EC2 instance (t2.medium)
- 2 Intel Xeon CPU v4 @2.3GHz
- 4GB RAM, 40GB SSD

Data Setup:

- 16M K-V pairs (~ 4GB)
- Key = 4B, Payload = 252B

Benchmarking Action

Metrics:

- Ingestion latency
- Overall operational latency

System Setup:

- AWS EC2 instance (t2.medium)
- 2 Intel Xeon CPU v4 @2.3GHz
- 4GB RAM, 40GB SSD

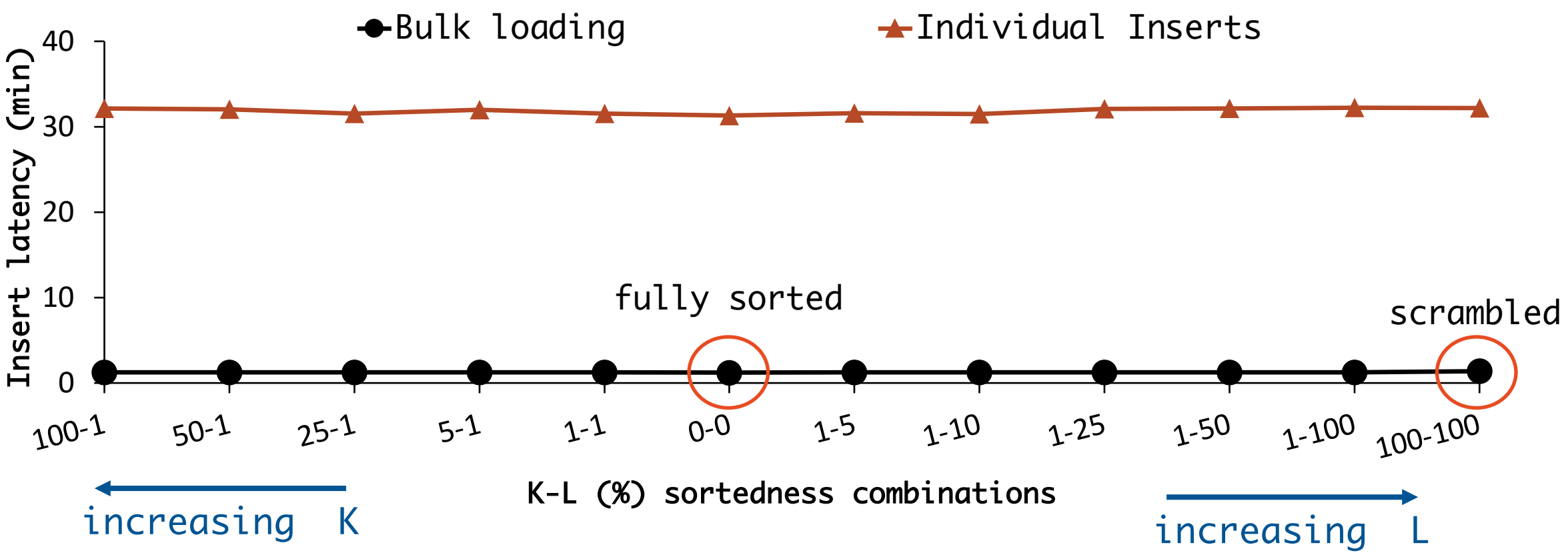
Data Setup:

- 16M K-V pairs (~ 4GB)
- Key = 4B, Payload = 252B

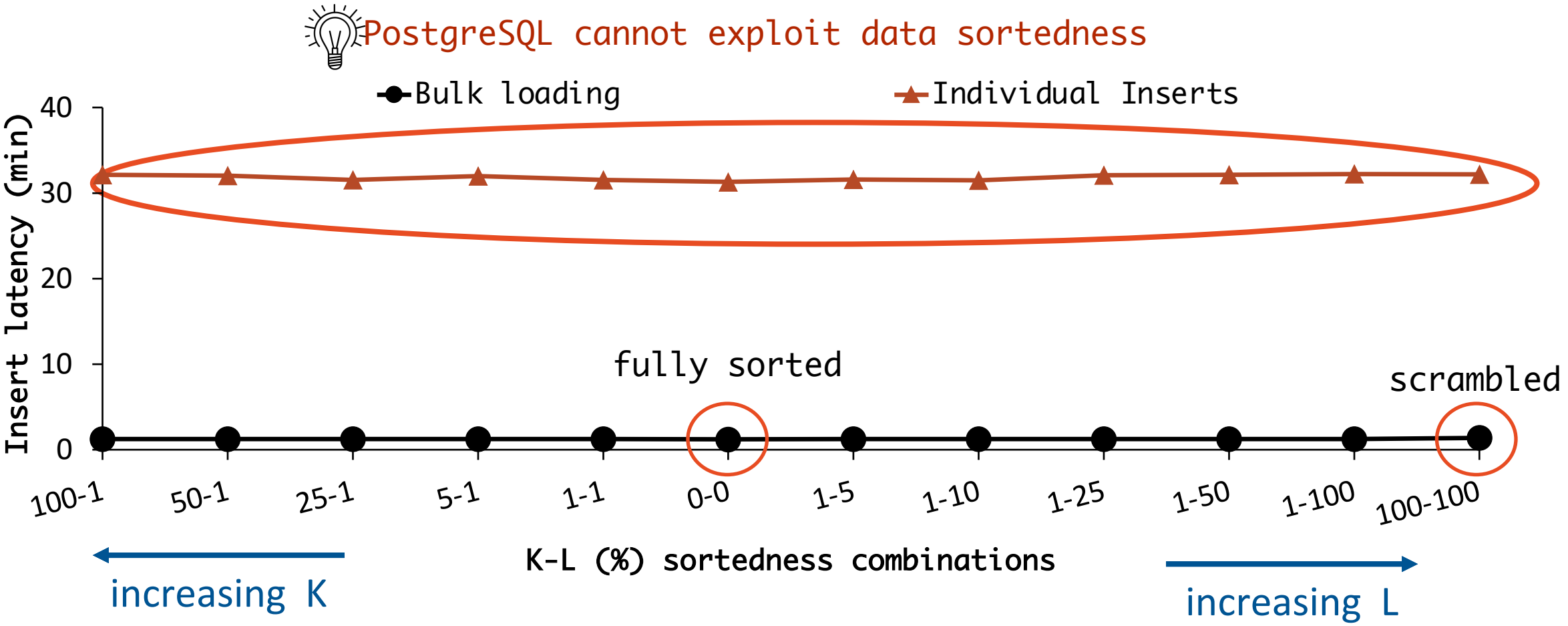
Default Index Setup:

- PostgreSQL (Unlogged tables)
- B-tree on key (id_col)

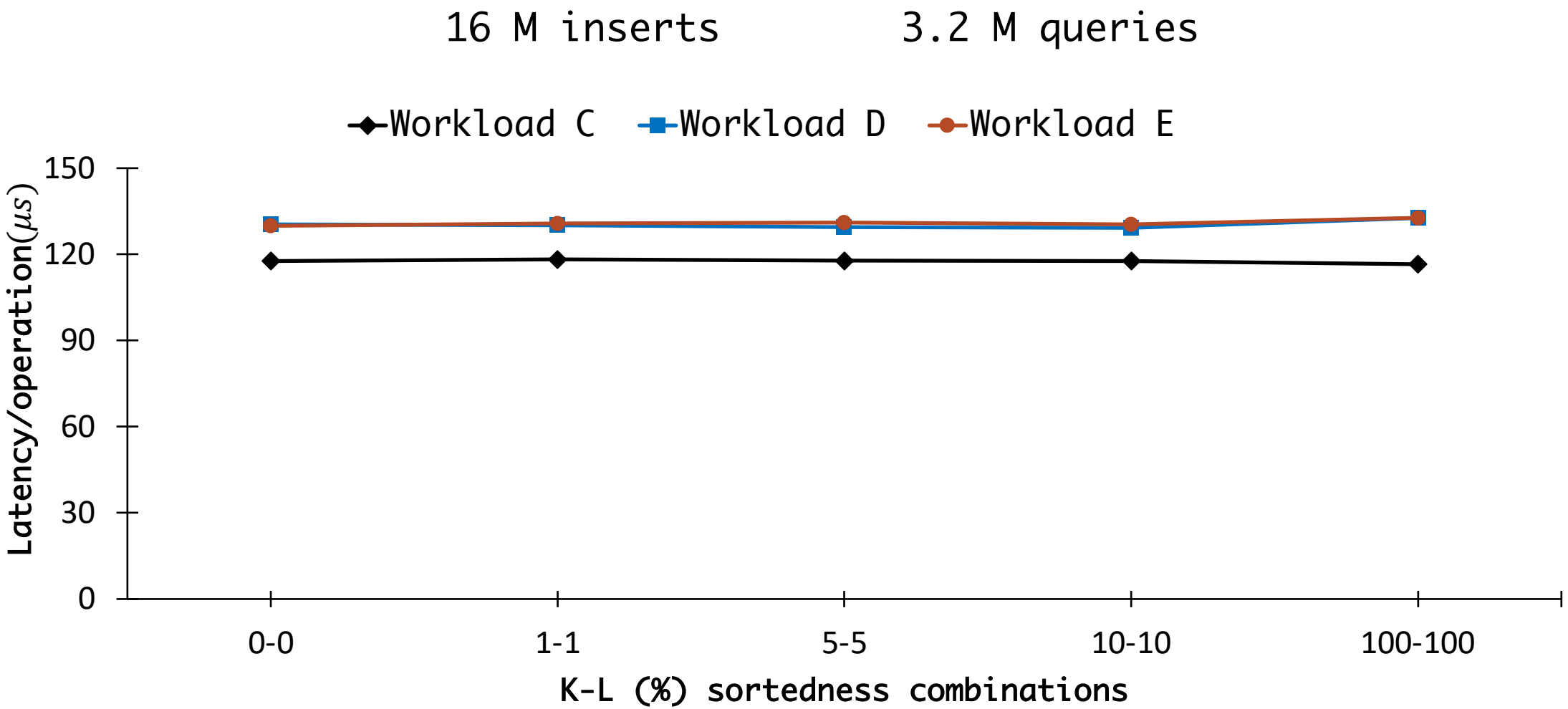
Raw Ingestion Performance



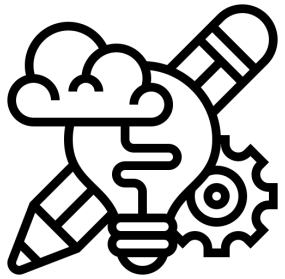
Raw Ingestion Performance



Mixed Workload Performance

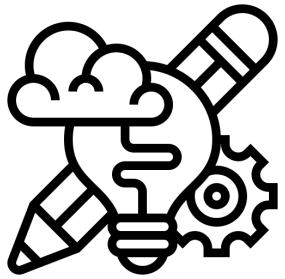


Column-Store Systems



Fundamental
design changes

Column-Store Systems

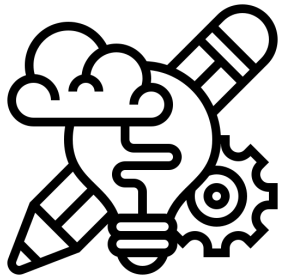


Fundamental
design changes

1. MonetDB

- ✕ invalidated by updates

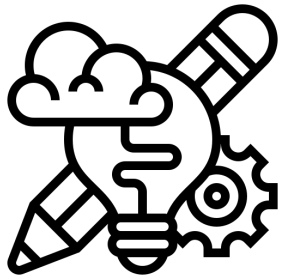
Column-Store Systems



Fundamental
design changes

1. MonetDB
   - ✗ invalidated by updates
2. Vertica
   - ✗ no live updates to sorted column

Column-Store Systems



Fundamental
design changes

1. MonetDB
   - ✗ invalidated by updates
2. Vertica
   - ✗ no live updates to sorted column
3. Actian Vector
   - ⚠ PDT similar to b-trees

The Sortedness-Aware (SWARE) Paradigm

Key Ideas in SWARE Paradigm

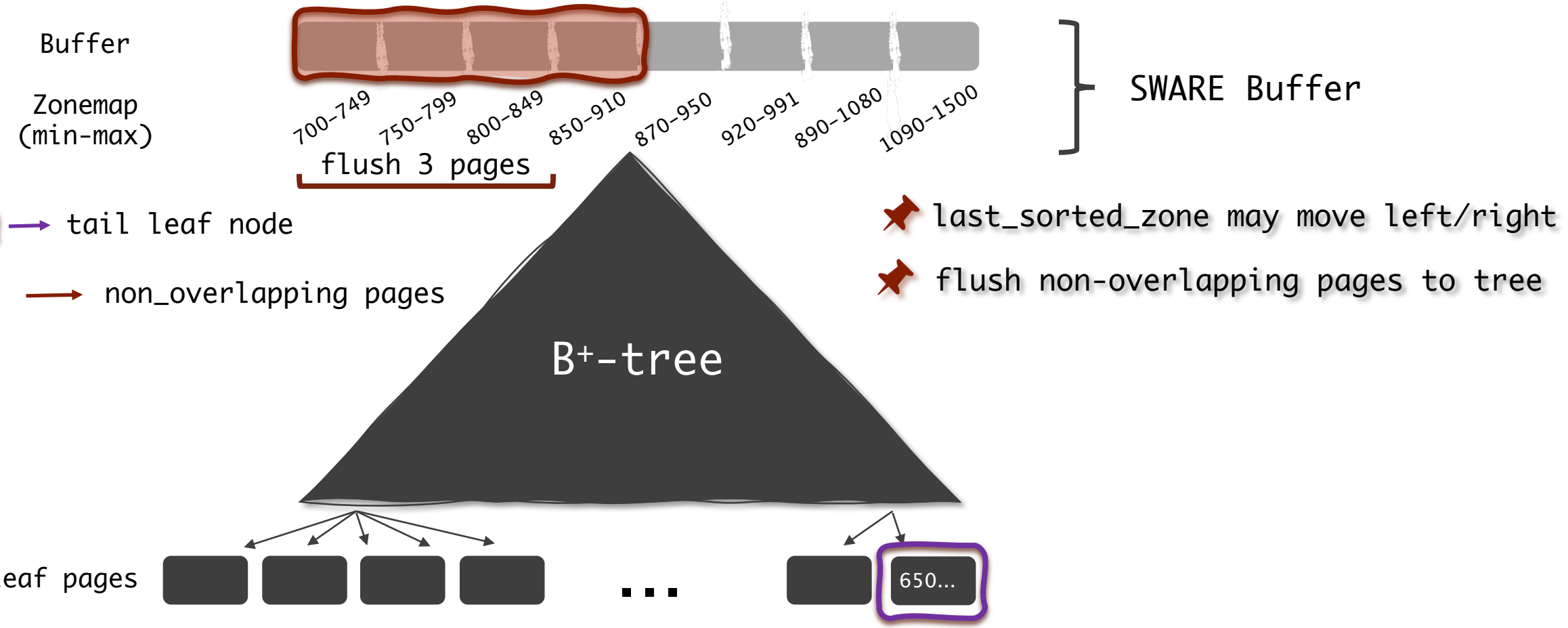
intelligent buffering

opportunistic bulk loading

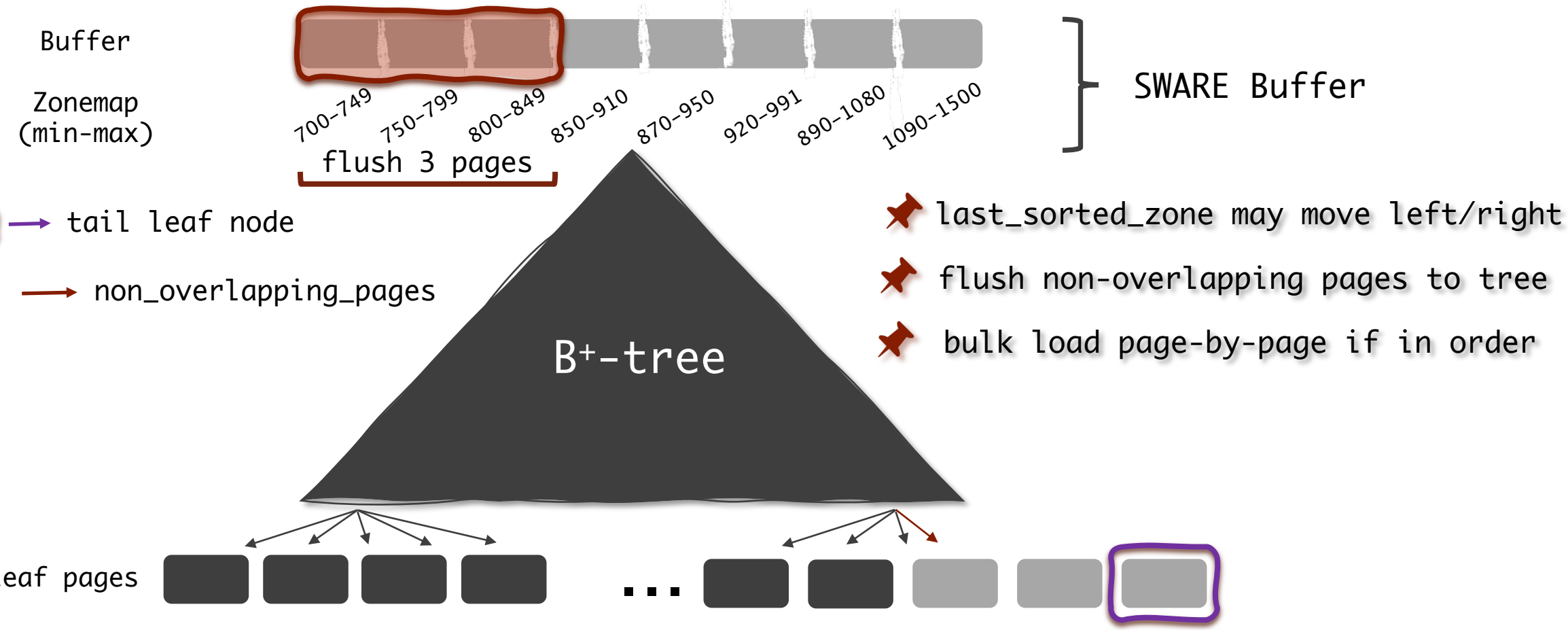
increased fill and split factor

Can be applied to any tree-index!

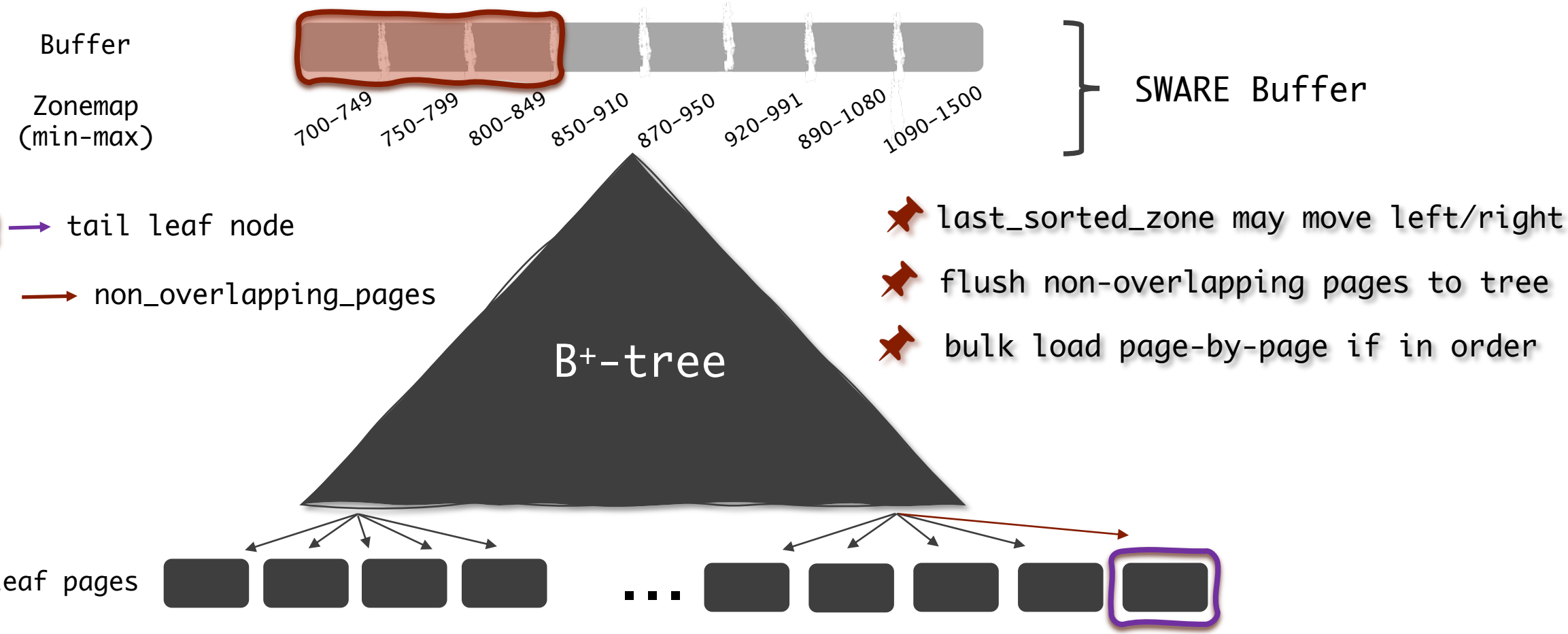
SWARE Ingestions



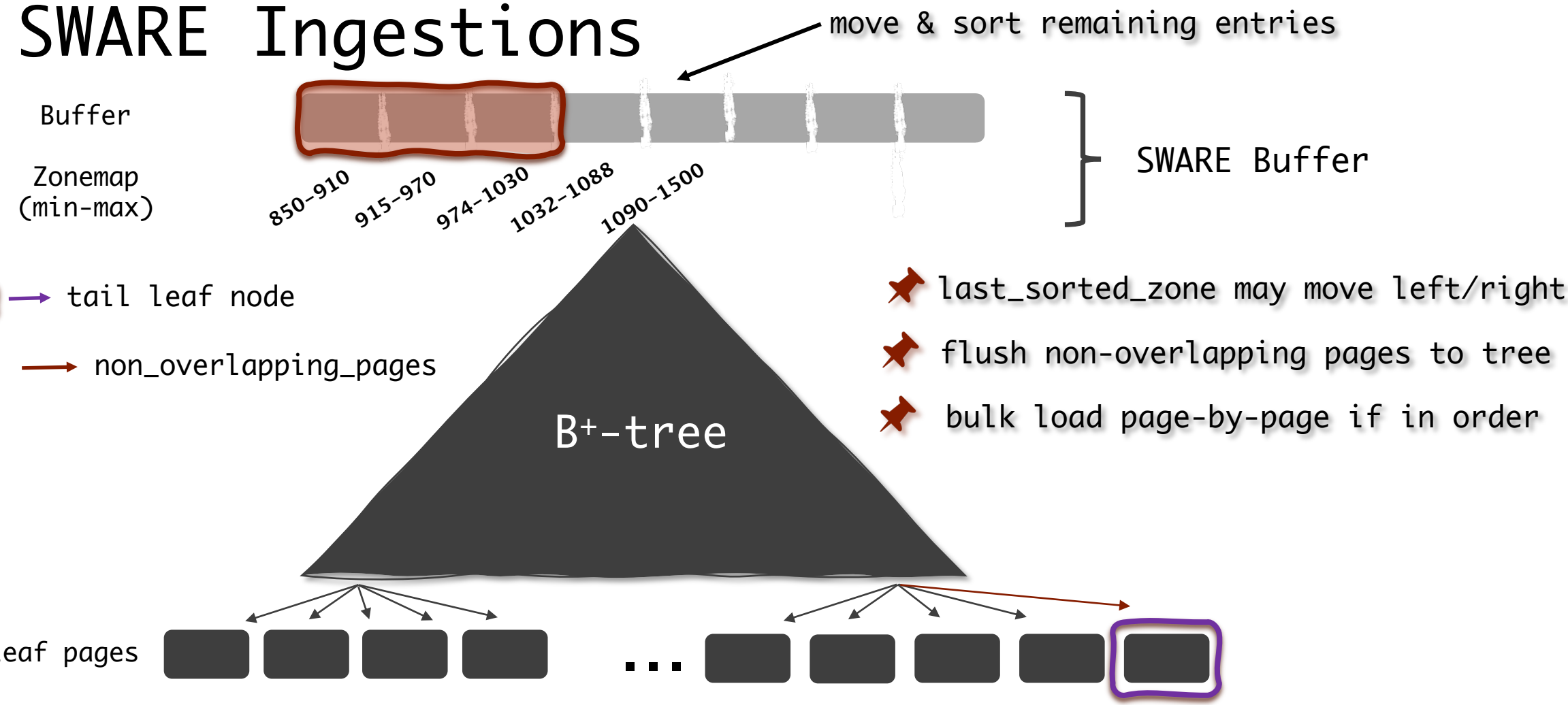
SWARE Ingestions



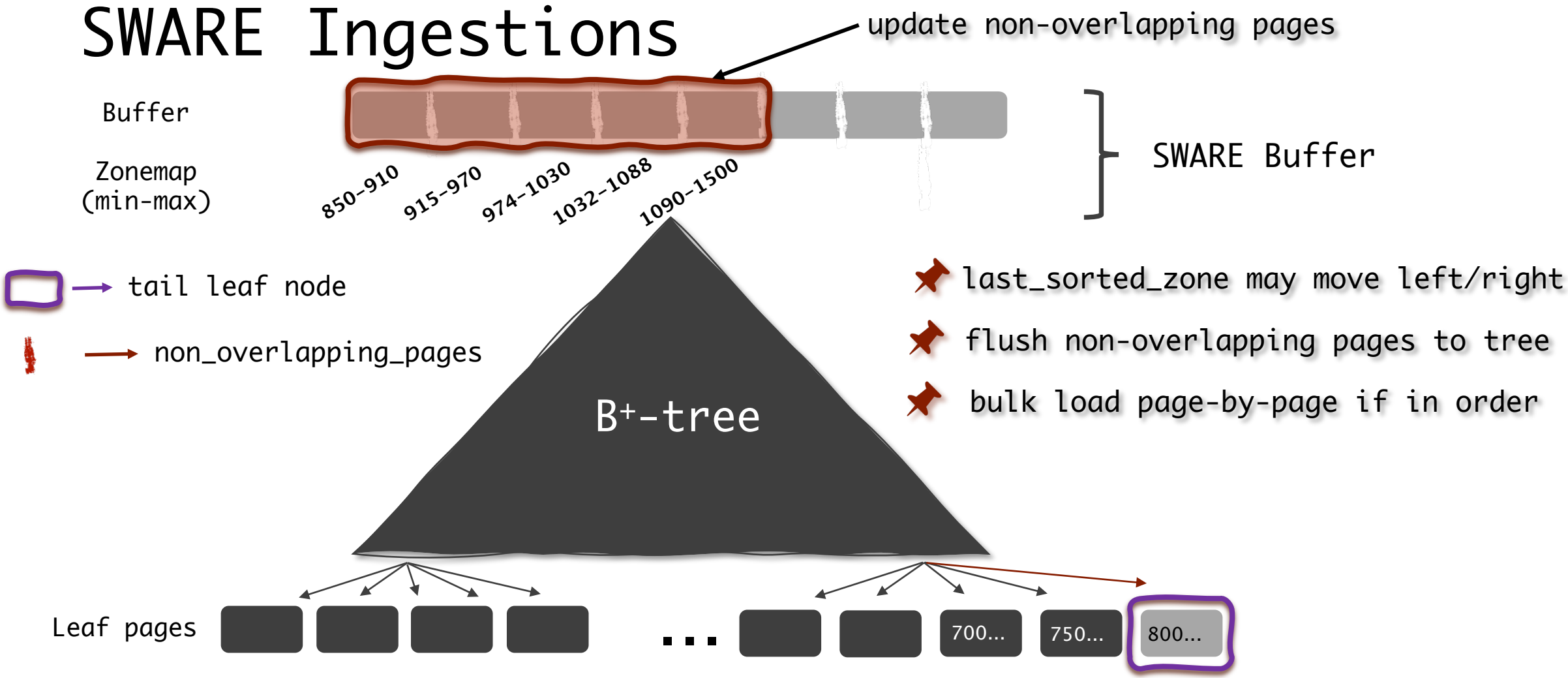
SWARE Ingestions



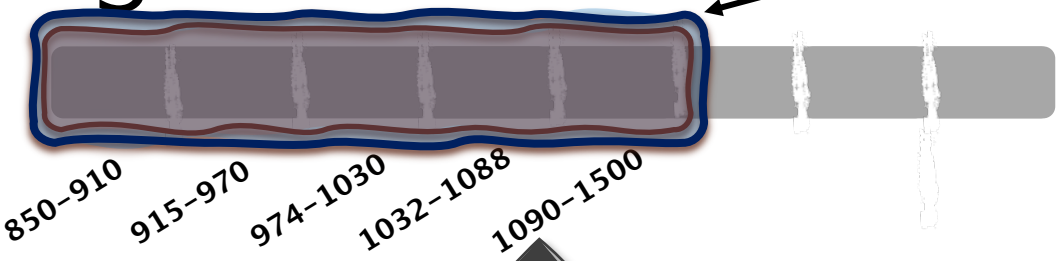
SWARE Ingestions



SWARE Ingestions



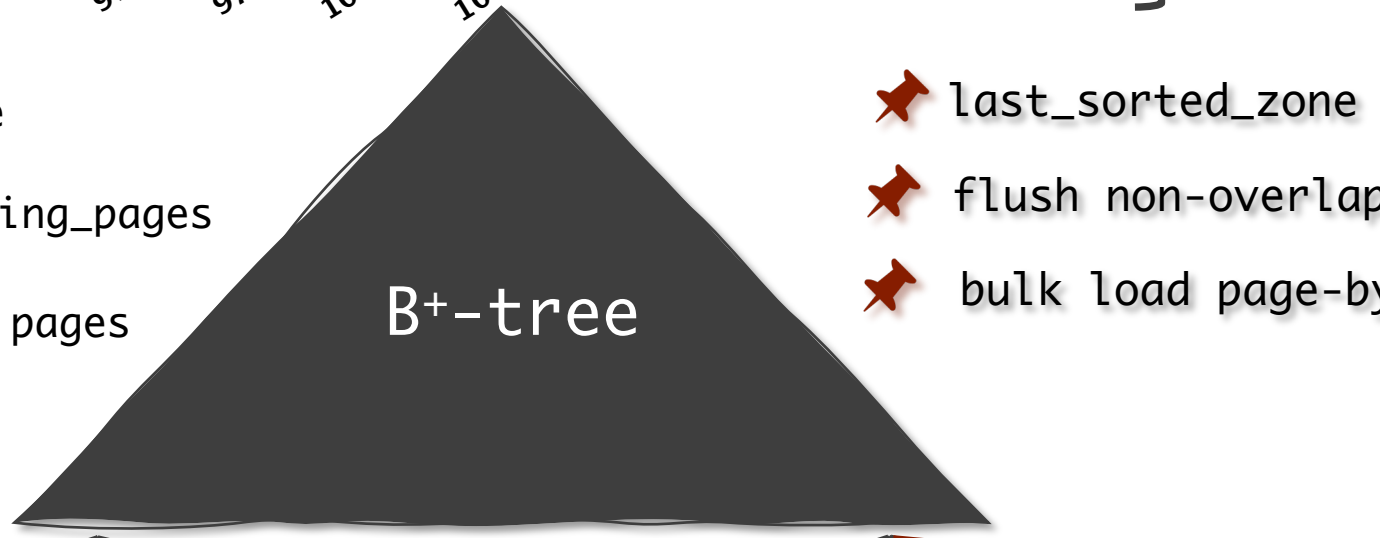
SWARE Ingestions



tail leaf node

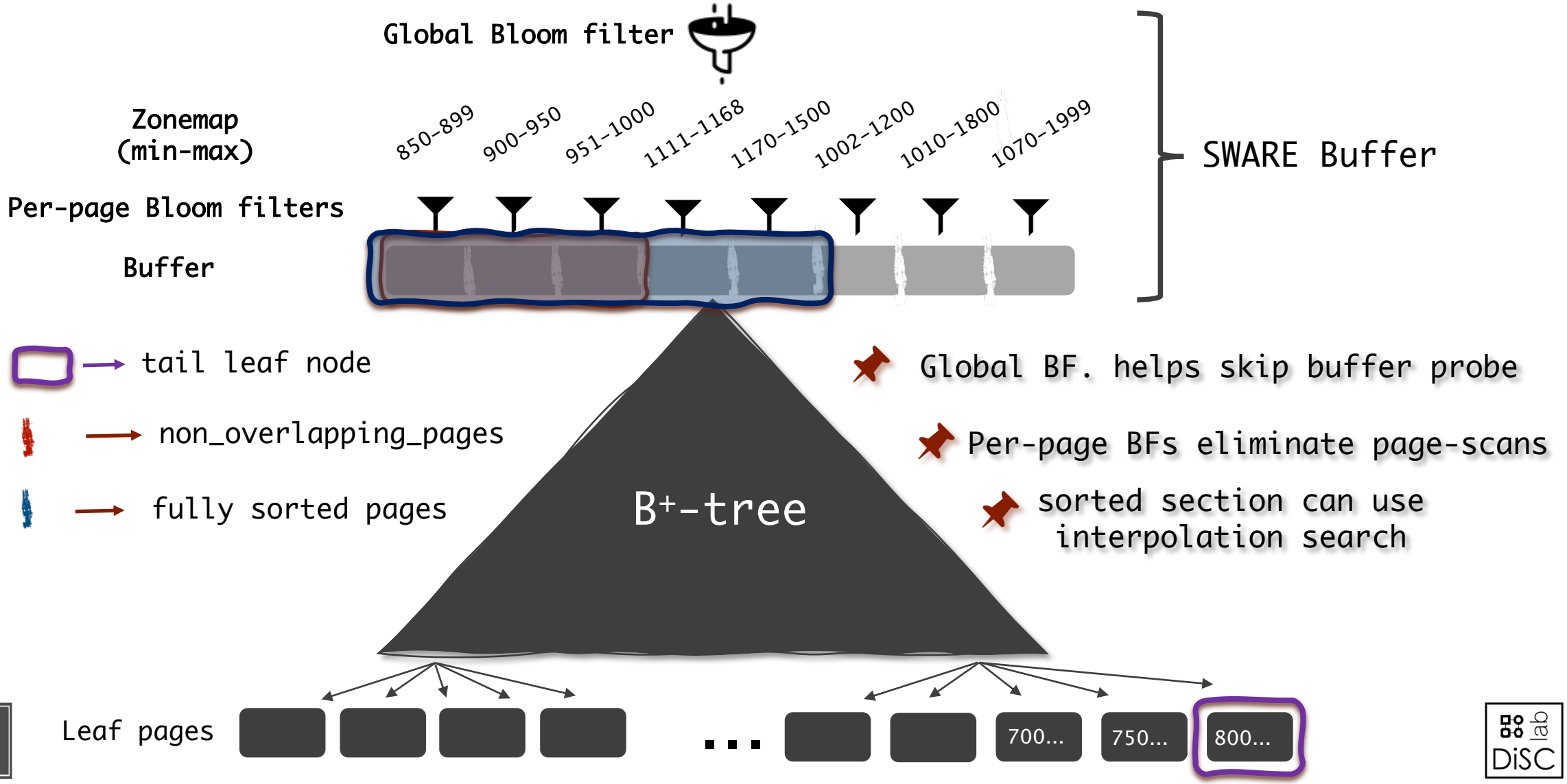
non_overlapping_pages

fully sorted pages

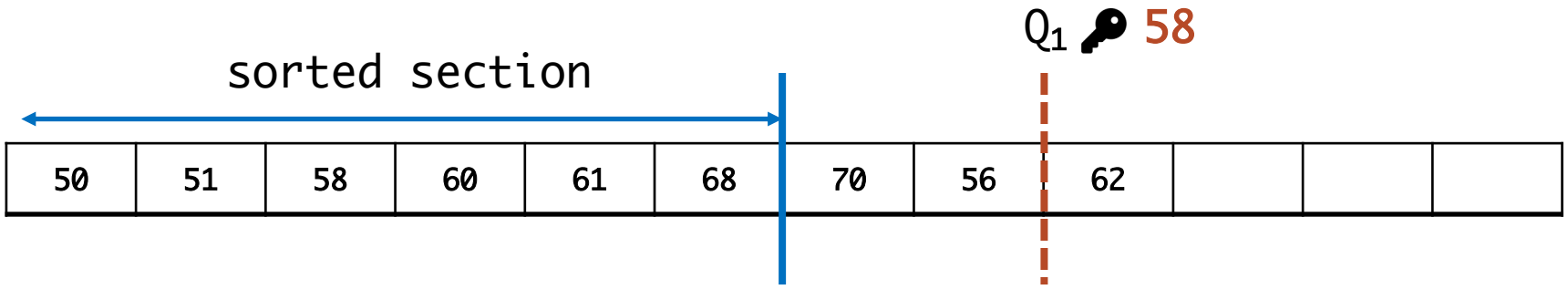


- last_sorted_zone may move left/right
- flush non-overlapping pages to tree
- bulk load page-by-page if in order

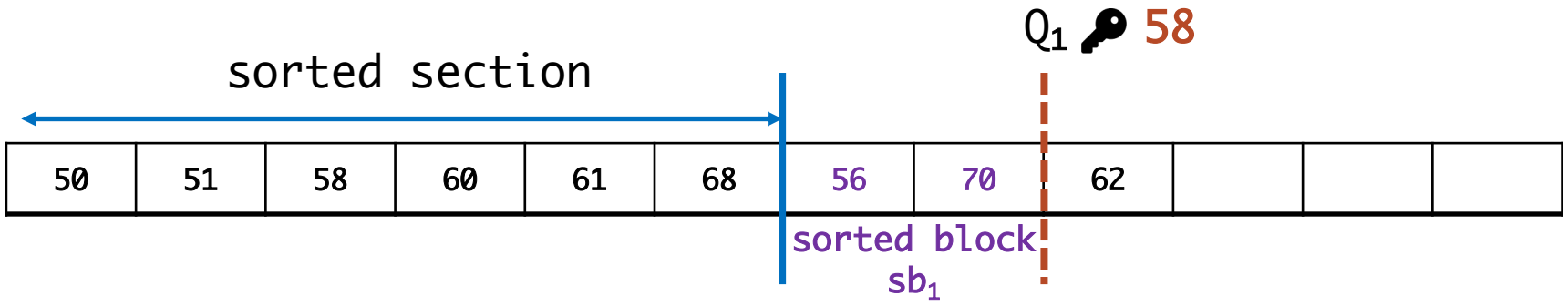
Overall Structure for Queries



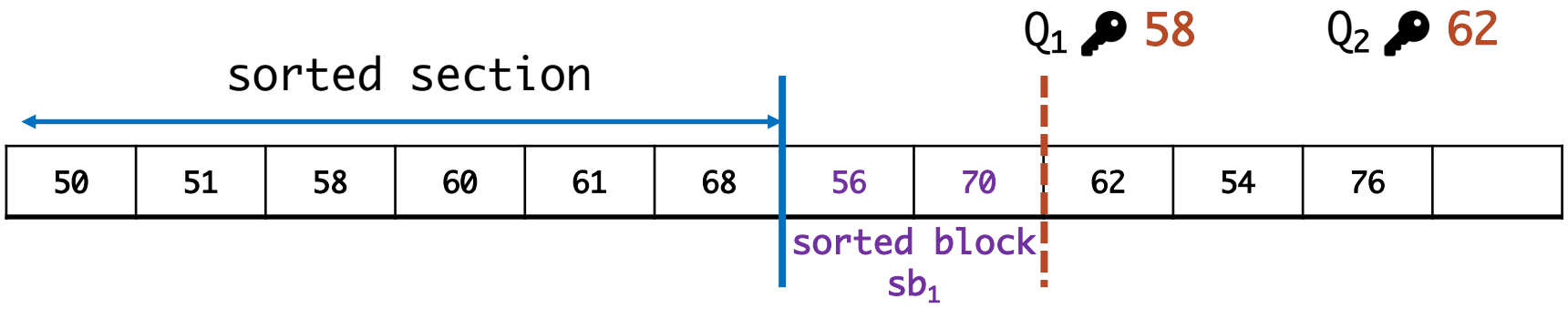
Query-driven Partial Sorting



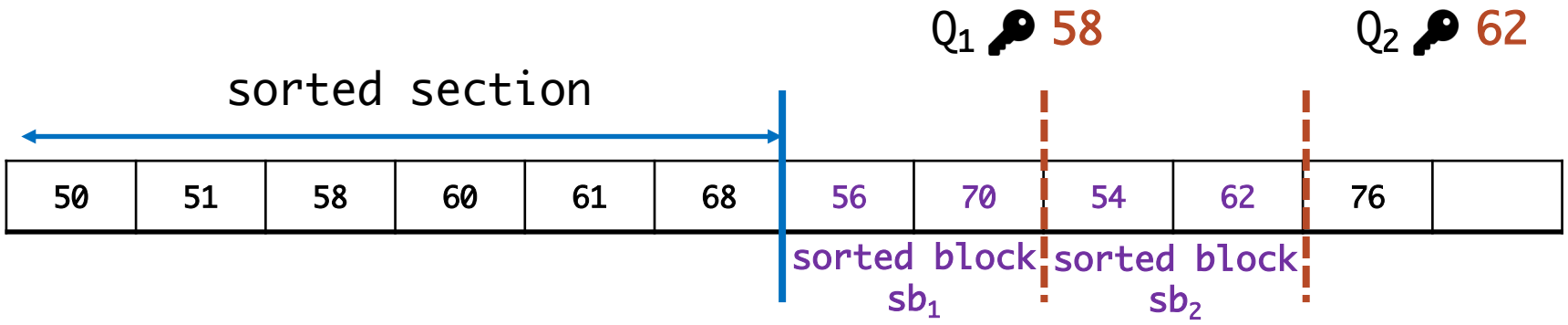
Query-driven Partial Sorting



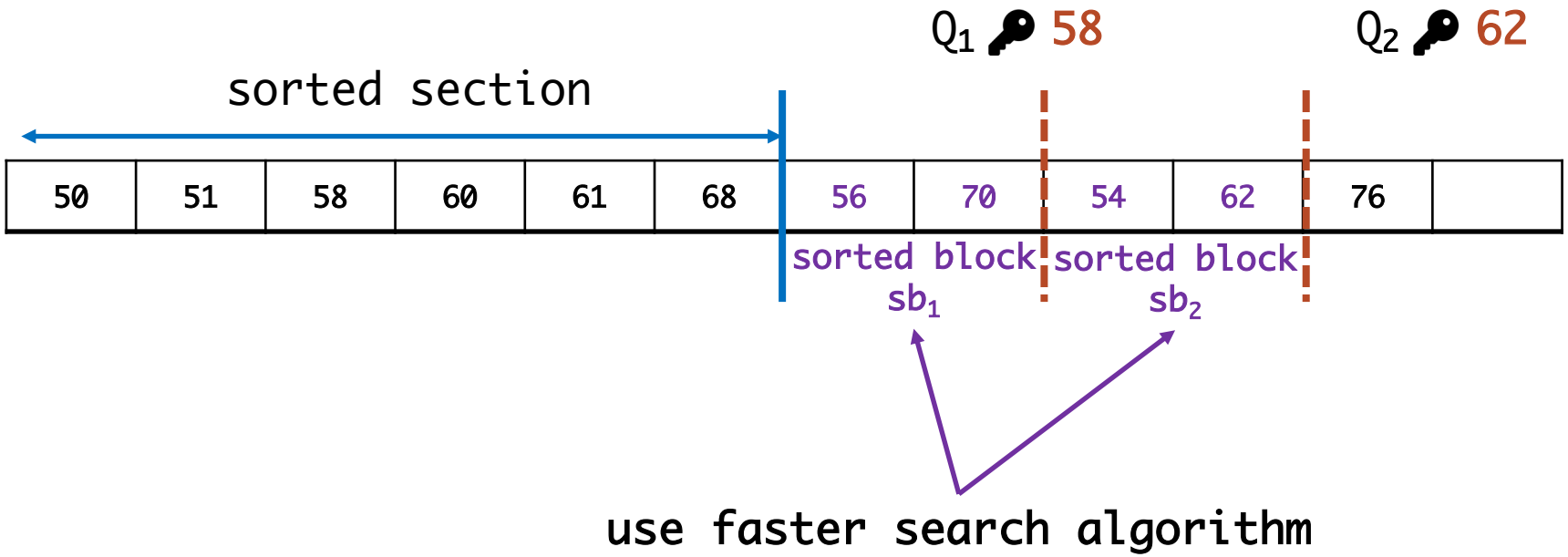
Query-driven Partial Sorting



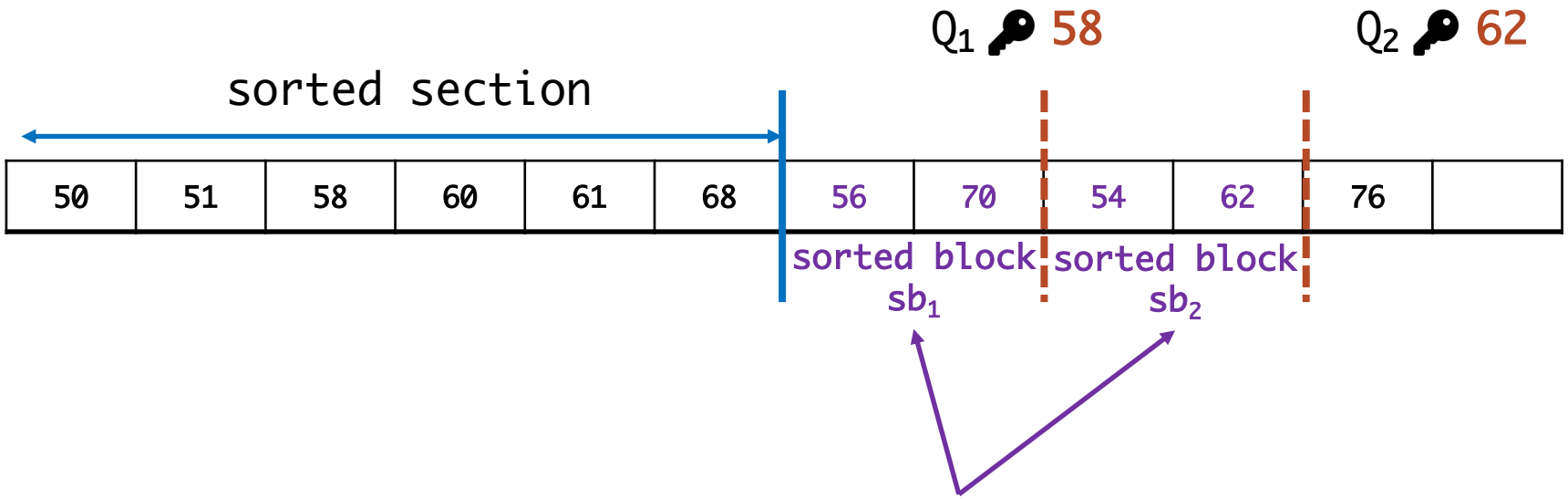
Query-driven Partial Sorting



Query-driven Partial Sorting



Query-driven Partial Sorting



Experimental Setup

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

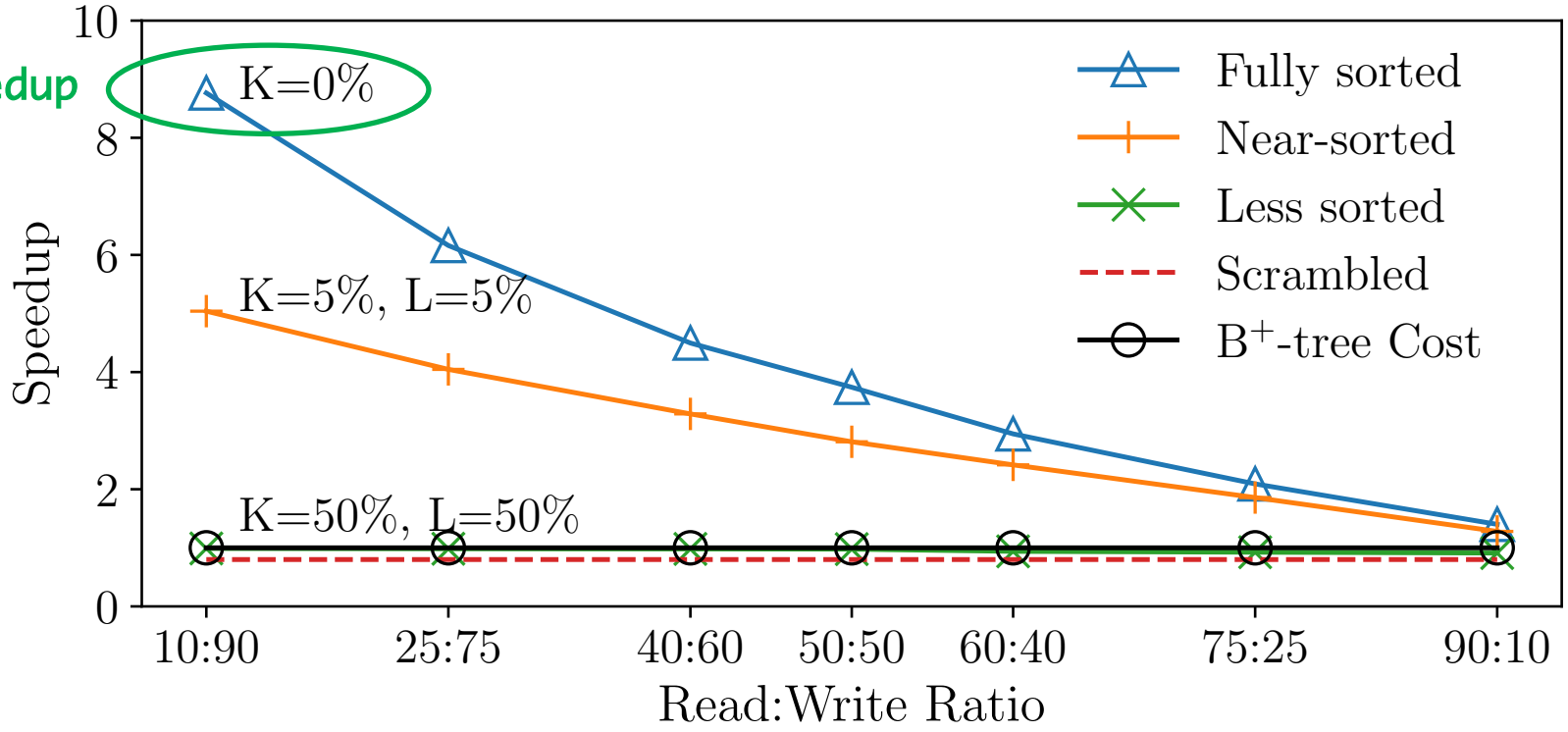
Workload Generator: **BoDS**
1. 500M Integer keys (~ 4GB)
2. Random lookups on existing keys

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 = 80:20; Bulk load = 95%

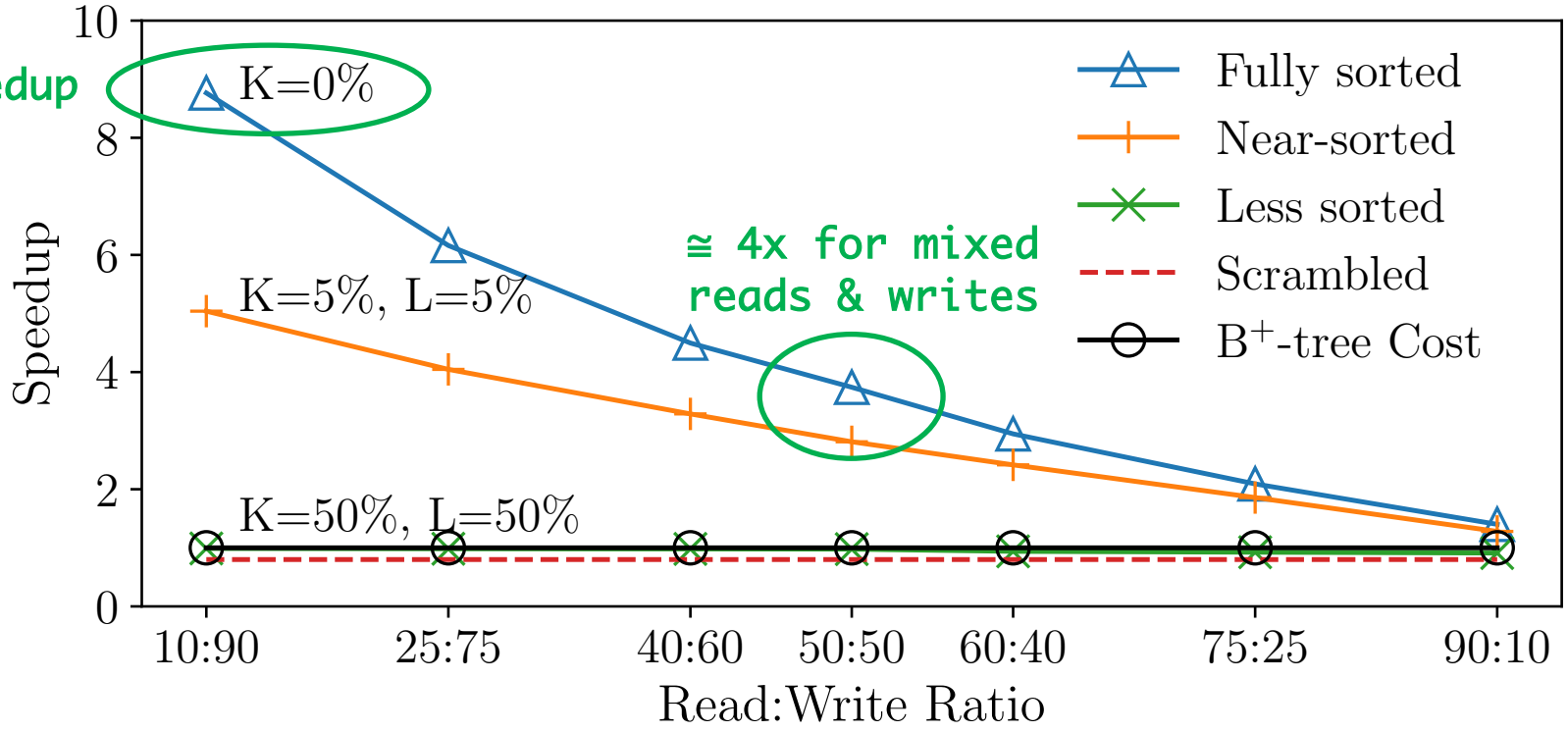
Overall Performance

≅ 9x speedup

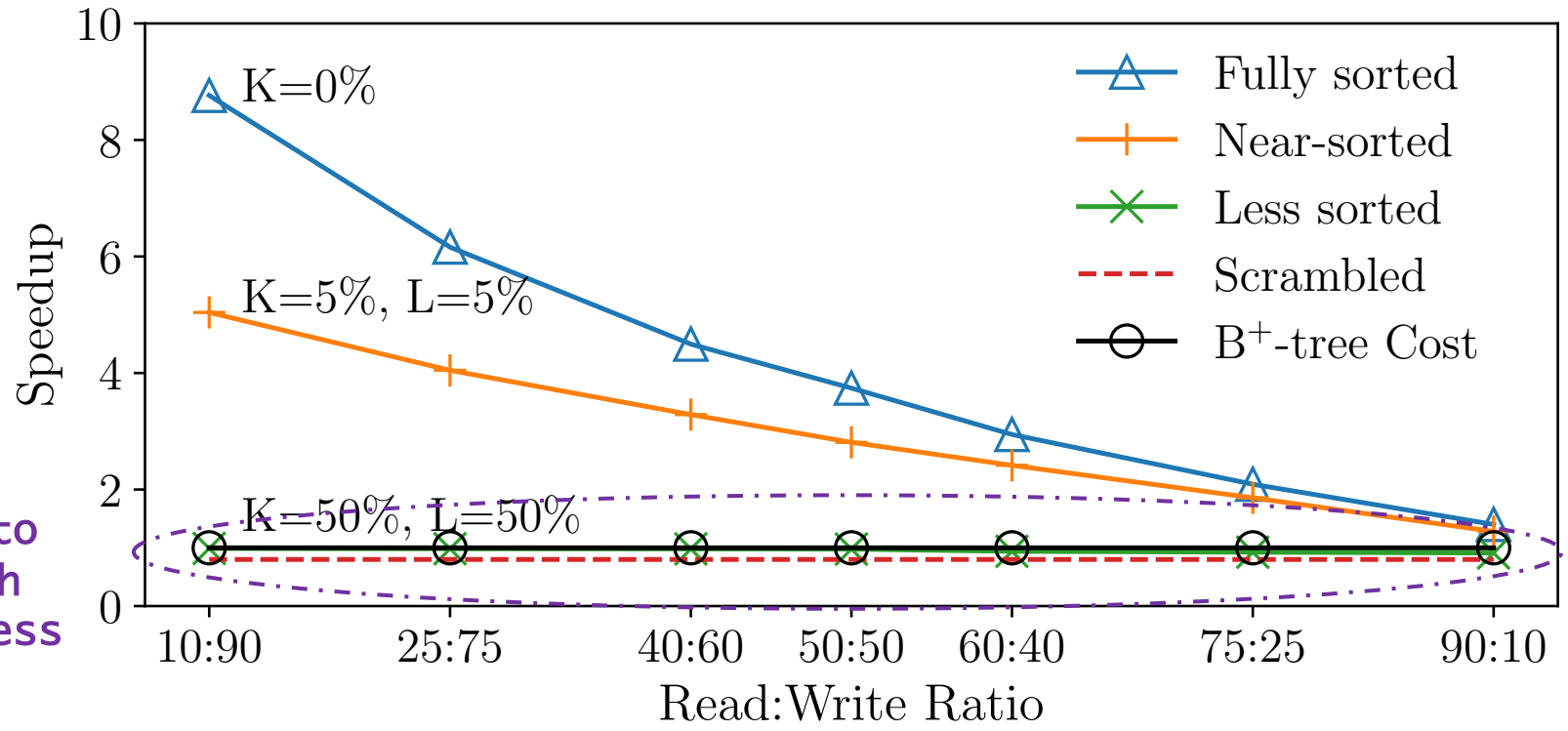


Overall Performance

≅ 9x speedup

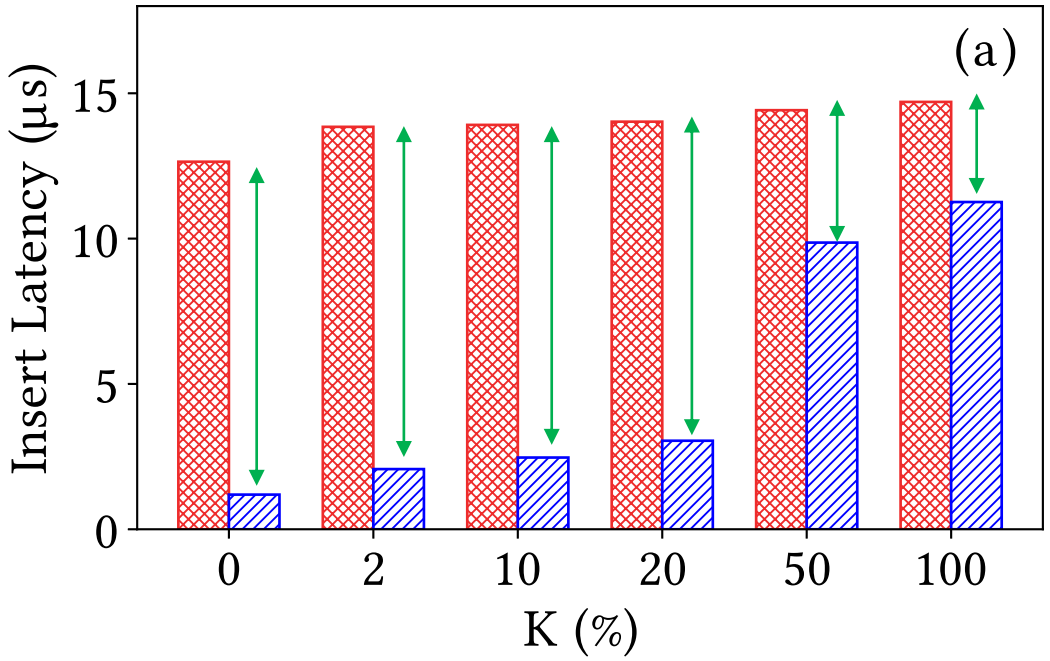


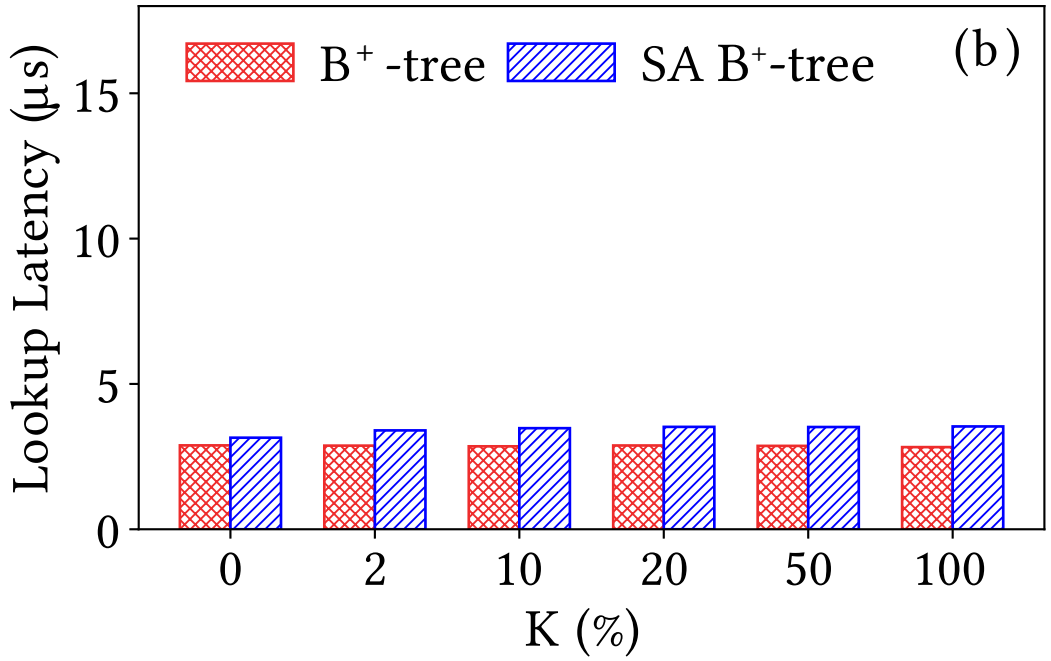
Overall Performance



comparable to B+-tree with less sortedness

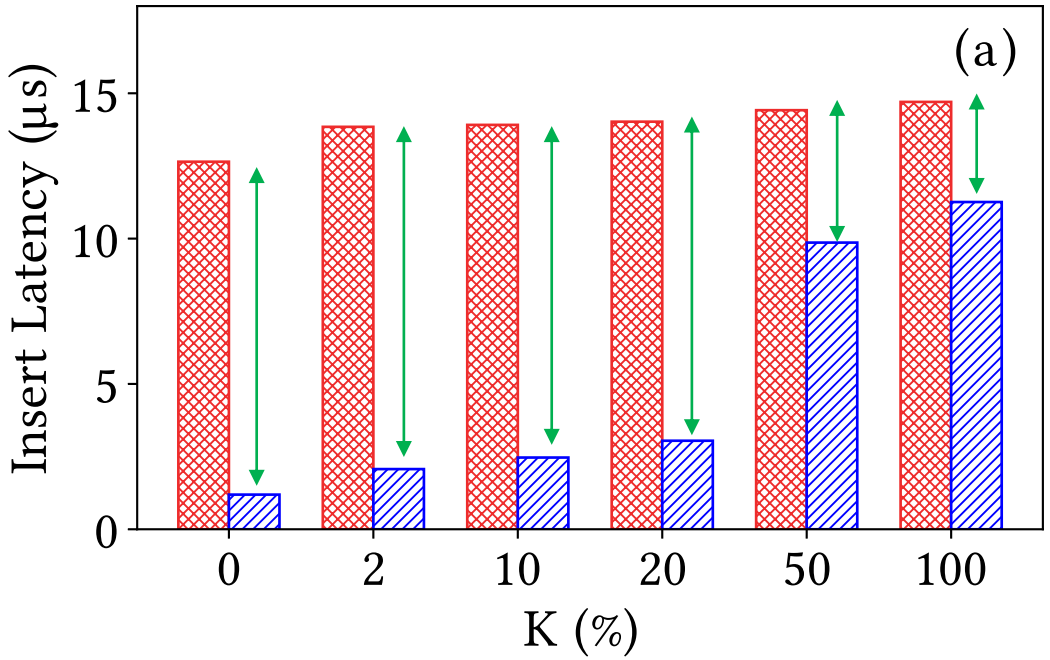
Raw Performance



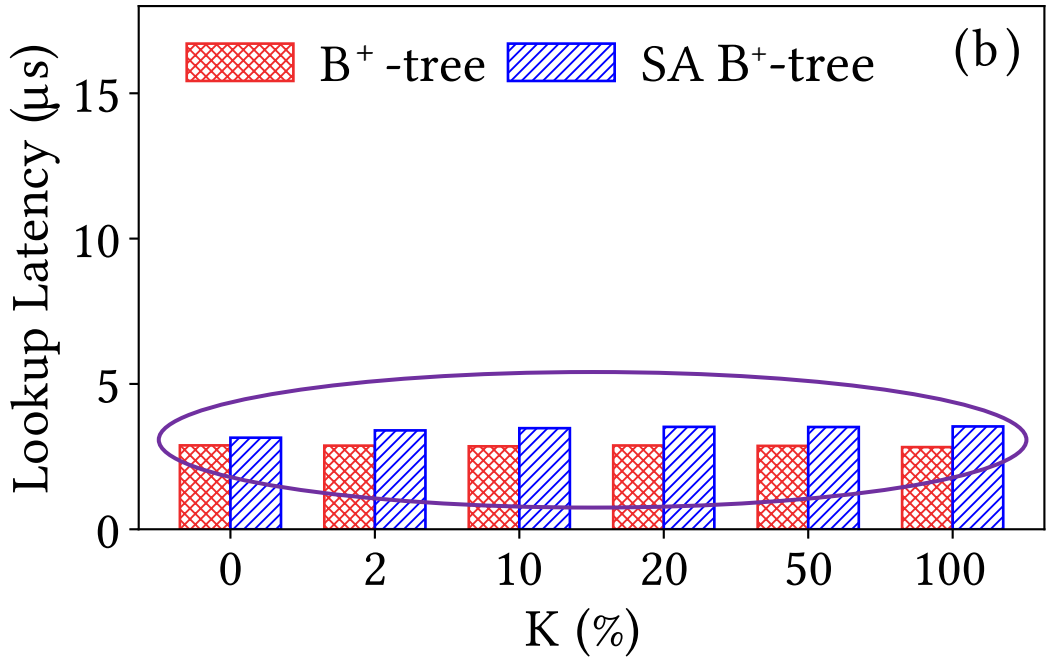


ingestion latency reduced between 27-90%

Raw Performance

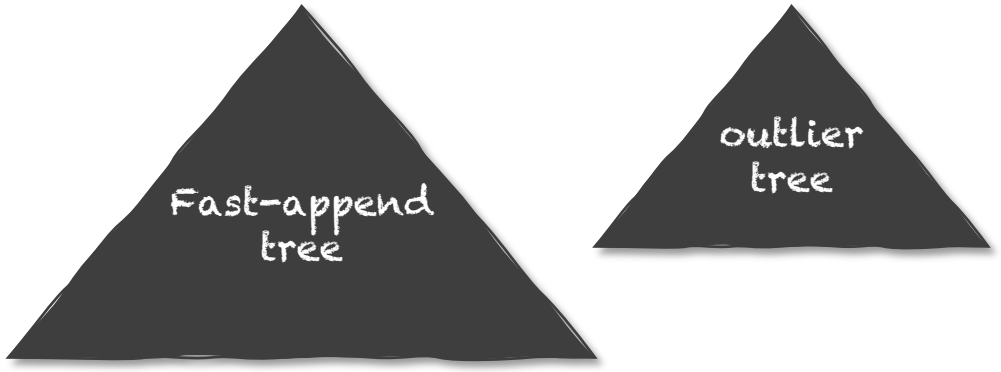


ingestion latency reduced between 27-90%

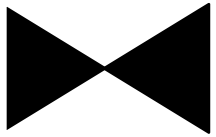


overhead in lookups between 5-26%

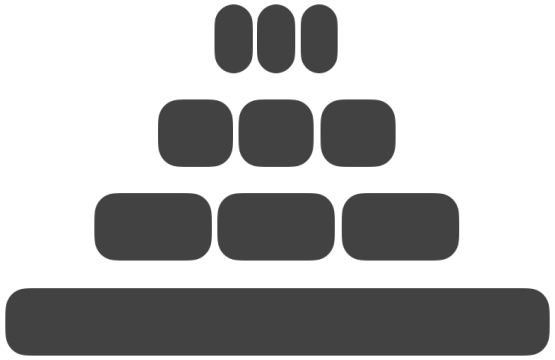
Future Work



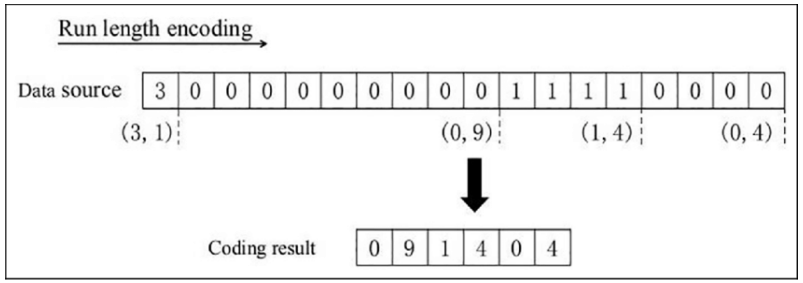
Dual B⁺-tree



Speed up SMJ for near-sorted data?



Can we build LSM-trees bottom up?



Can compression algorithms exploit sortedness?

BOSTON UNIVERSITY

DiSC lab

Summary

Identify “**sortedness**” as a resource

Smart buffering + bulk index appends = **faster** inserts

8.8x speedup with **SWARE** meta-design

Framework can be **extended** to other indexes

Thank You!