Indexing Near-Sorted Data

Aneesh Raman  
aneeshr@bu.edu

Matthaios Olma  
maolma@microsoft.com

Subhadeep Sarkar  
ssarkar1@bu.edu

Manos Athanassoulis  
mathan@bu.edu
Indexes in Databases

Efficient queries

write amplification

space amplification
Read-Write Tradeoff

- **read cost**
  - **logarithmic search** $O(\log(n))$ or faster
  - **append** $O(1)$
  - **scan** $O(n)$

- **write cost**
  - **in-order insertion** $O(\log(n))$
Data Sortedness

- Inversions, length of sub-sorted sequences, etc.
- Focus on a single metric to bring a data stream into sorted order
- K-L metric by Ben-Moshe (ICDT 2011)

1 8 3 4 5 6 7 2 9 10

#. unordered entries = K

max. displacement among unordered entries = L
Problem

Ingestion Latency

Scrambled | Less-sorted | Nearly-sorted | Sorted

Data Sortedness

B-tree | Bulk loading | Ideal tree
Our Vision
Our Vision

- logarithmic search $O(\log(n))$ or faster
- scan $O(n)$
- read cost
- append $O(1)$
- write cost
- in-order insertion $O(\log(n))$
- Data sortedness
- unsorted
- sorted
Contributions

1. “Sortedness” as a resource
2. Index meta-design
3. SWARE design paradigm
4. Sortedness-aware Index (SA $B^+$-tree)
SWARE Design Elements

- Rightmost leaf-appends
- Bulk loading
- Variable fill & split factor
- Buffering
Right-most Leaf Appends

Normal Insertion (top-insert)

Insert Key 65
Right-most Leaf Appends

Normal Insertion (top-insert)

Insert Key 65
Right-most Leaf Appends

Normal Insertion (top-insert)

Insert Key 65
Right-most Leaf Appends

Normal Insertion (top-insert)

Insert Key 65

Insert to tail leaf

Insert Key 65  
**tail-leaf-ptr**  
*min_val(55)*
Right-most Leaf Appends

Normal Insertion (top-insert)

Insert Key 65

Insert to tail leaf

Insert Key 65

\( \text{tail-leaf-.ptr} \)  \( \text{min_val(55)} \)

Is 65 >= min_val?
Right-most Leaf Appends

Normal Insertion (top-insert)

Insert Key 65

Insert Key 65

tail-leaf-ptr

\[ \text{min_val}(55) \]

is \( 65 \geq \text{min_val} \)?

yes!

add key to tail leaf directly!

Insert to tail leaf
Bulk loading

Insert Keys [85, 90, 95]

tail_leaf_parent
Bulk loading

Insert Keys \([85, 90, 95]\)

tail_leaf_parent

new leaf
Bulk loading

Insert Keys [85, 90, 95]

deadline

tail_leaf_parent

new leaf
Bulk loading

Insert Keys [85, 90, 95]

tail_leaf_parent

bulk loaded leaf
Adjusting Node Fill/Split Factor

Split Ratio = 50:50
Adjusting Node Fill/Split Factor

Split Ratio = 50:50

Split Ratio = 80:20
SWARE Ingestion

Rightmost leaf-appends

Opportunistic Bulk loading

Variable fill & split factor

Smart buffering

Faster ingestion
SWARE Ingestion

- Rightmost leaf-appends
- Opportunistic bulk loading
- Variable fill & split factor

Smart buffering = Faster ingestion
SWARE Ingestion

- Rightmost leaf-appends
- Opportunistic Bulk loading
- Variable fill & split factor

meta-data to identify sortedness

sorted entries

out of order entries

40 42 44 50 60 52 54 70

Smart buffering

Faster ingestion
SWARE Ingestion

Rightmost leaf-appends

Opportunistic Bulk loading

Variable fill & split factor

Opportunistically bulk load

Smart buffering

Faster ingestion
SWARE Ingestion

- Rightmost leaf-appends
- Opportunistic Bulk loading
- Variable fill & split factor

= Smart buffering

= Faster ingestion

move remaining entries
SWARE Ingestion

- Rightmost leaf-append
- Opportunistic bulk loading
- Variable fill & split factor

Smart buffering

Faster ingestion

52 54 60 70
sort remaining entries
Optimizing Reads

zonemaps  Bloom filters  query-driven partial sorting
Overall Structure

Global Bloom filter

Zonemap layer

Bloom filter layer

SWARE Buffer

B⁺-tree

buffer page
Overall Structure

- Global Bloom filter
- Zonemap layer
- Bloom filter layer
- SWARE Buffer
  - sorted section
  - use zonemap & Bloom filters in unsorted section
  - buffer page

B^+ tree
Overall Structure

Global Bloom filter

Zonemap layer

Bloom filter layer

SWARE Buffer

sorted section

B^+-tree

use zonemap & Bloom filters in unsorted section

scan qualifying pages in unsorted section

buffer page
Overall Structure

- Global Bloom filter
- Zonemap layer
  - SWARE Buffer
    - sorted section
    - Use interpolation search ($O(\log(\log(n))$) for sorted section
    - Scan qualifying pages in unsorted section
- Bloom filter layer
  - use zonemap & Bloom filters in unsorted section
- $B^+$-tree
  - buffer page
Scanning Unsorted Section

unsorted section scan is still expensive!
Scanning Unsorted Section

increase sorted component

Query-driven partial sorting
Query-driven Partial Sorting

use faster search algorithm (e.g., interpolation search / binary search)

simply merge sorted components => also helps speed up inserts!
Tuning Decisions

- sorting algorithm
- fill & split factor
- buffer size
- query sorting threshold
Experimental Setup

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

**Workload Generator:**
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 (5M); flush <= 50%
2. BFVs = 10 BPK; Murmur Hash
3. Split = 80:20; Bulk load = 95%
Overall Performance

\[ \cong 9x \text{ speedup} \]

\[ K=0\% \]

\[ K=5\%, L=5\% \]

\[ K=50\%, L=50\% \]

Fully sorted
Near-sorted
Less sorted
Scrambled
B\textsuperscript{+}-tree Cost

Read:Write Ratio

\begin{array}{cccccc}
\end{array}
Overall Performance

- $\approx 9x$ speedup
- $\approx 4x$ for mixed reads & writes

$K = 0\%$

$K = 5\%, L = 5\%$

$K = 50\%, L = 50\%$

B$^+$-tree Cost

Fully sorted
Near-sorted
Less sorted
Scrambled

Read:Write Ratio

Speedup

BOSTON UNIVERSITY
Overall Performance

compared to state-of-the-art with less sortedness
Raw Performance

(a) Insert Latency ($\theta$ s)

(b) Lookup Latency ($\theta$ s)

 ingestion latency reduced between 27-90%
Raw Performance

- Ingestion latency reduced between 27-90%
- Overhead in lookups between 5-26%
Summary

Indexes can use “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!