Jungle: Towards Dynamically Adjustable Key-Value Store by Combining LSM-Tree and Copy-On-Write B+Tree

Jung-Sang Ahn (junahn@ebay.com),
Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane

Platform Architecture & Data Infrastructure, eBay Inc.

USENIX HotStorage 2019

Outline

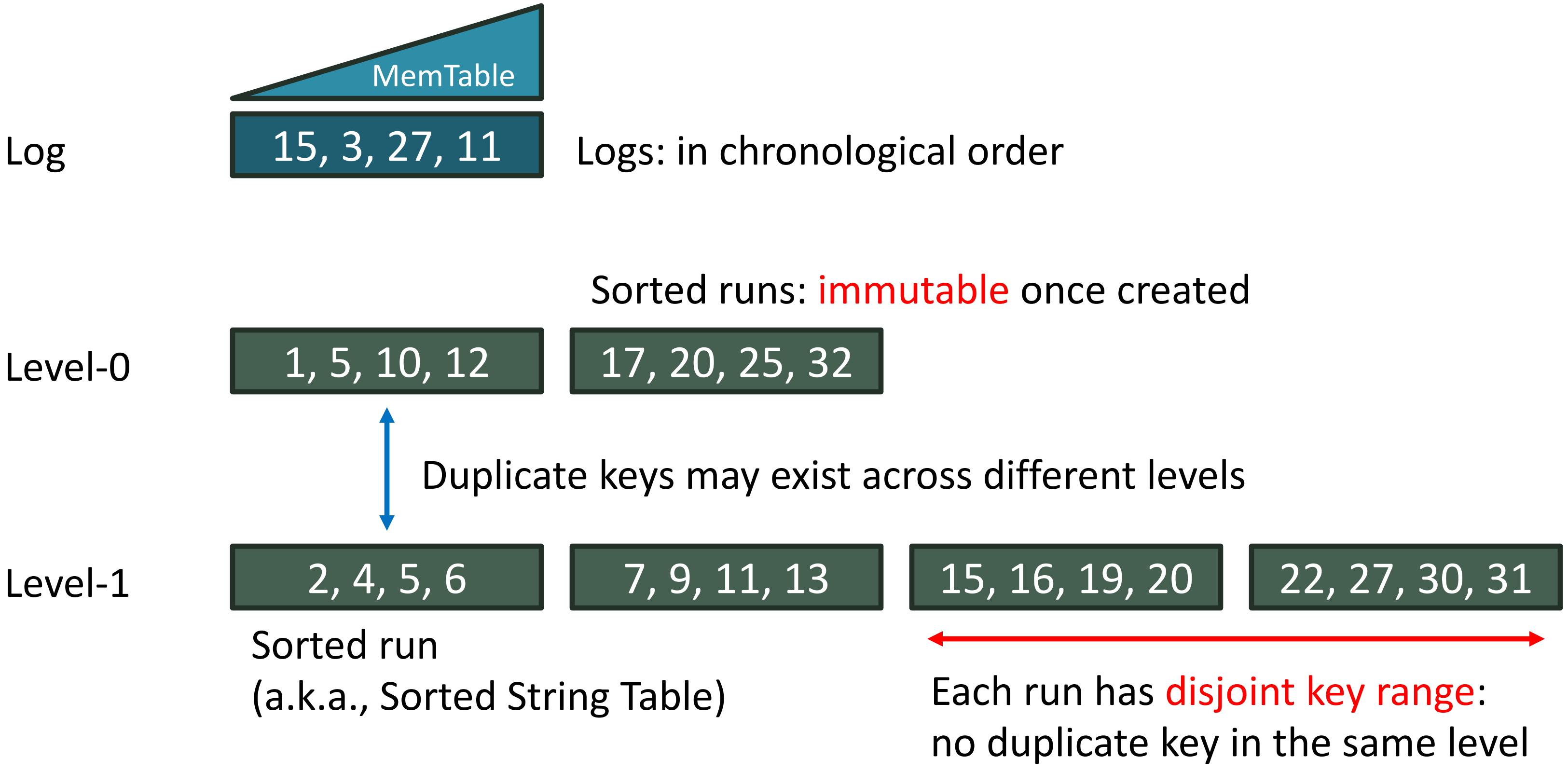
- Background: LSM-tree overview
  - Write amplification
  - Tiering merge: trade-offs
- Jungle
  - (Copy-on-write B+tree overview)
  - Combining LSM-tree and copy-on-write B+tree: why and how?
- Brief evaluation
- What's next?

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

- Lots of recent key-value stores & databases are using it (or its variants)
- Compared to B+tree
  - Better write performance
  - Relatively degraded (but acceptable) read performance

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

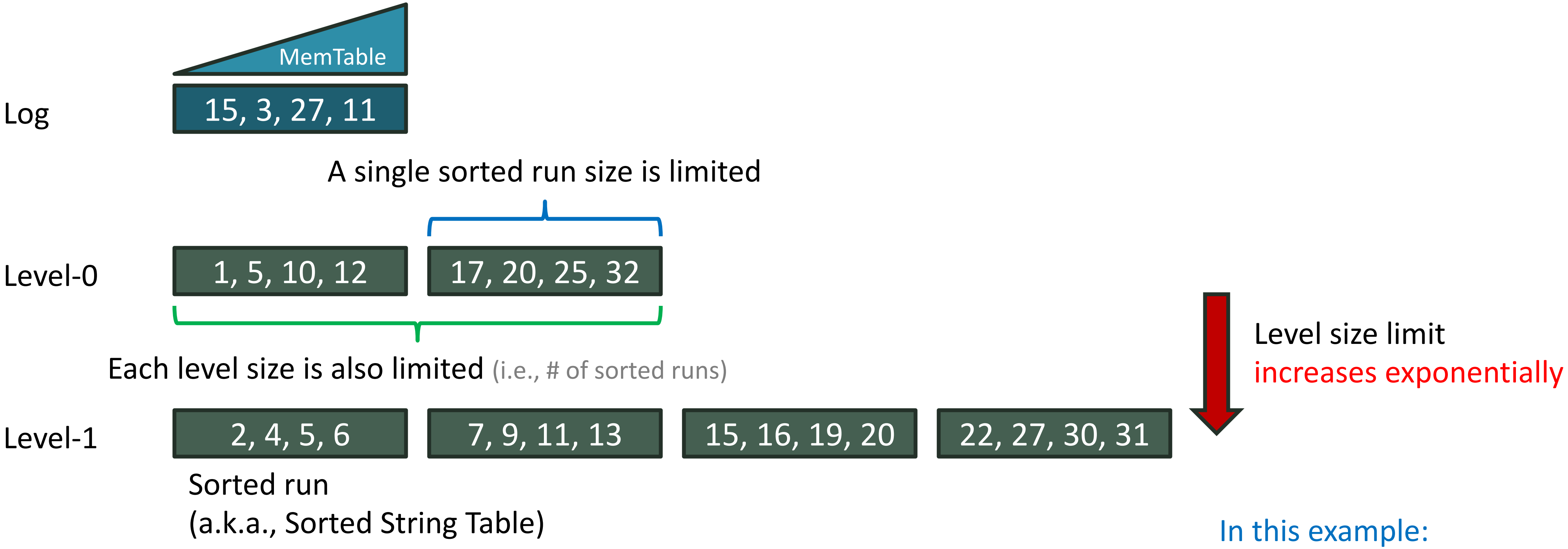
- Basic algorithm – write and merge



Disclaimer:
Some implementations allow exception for top-most level, i.e., level-0

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

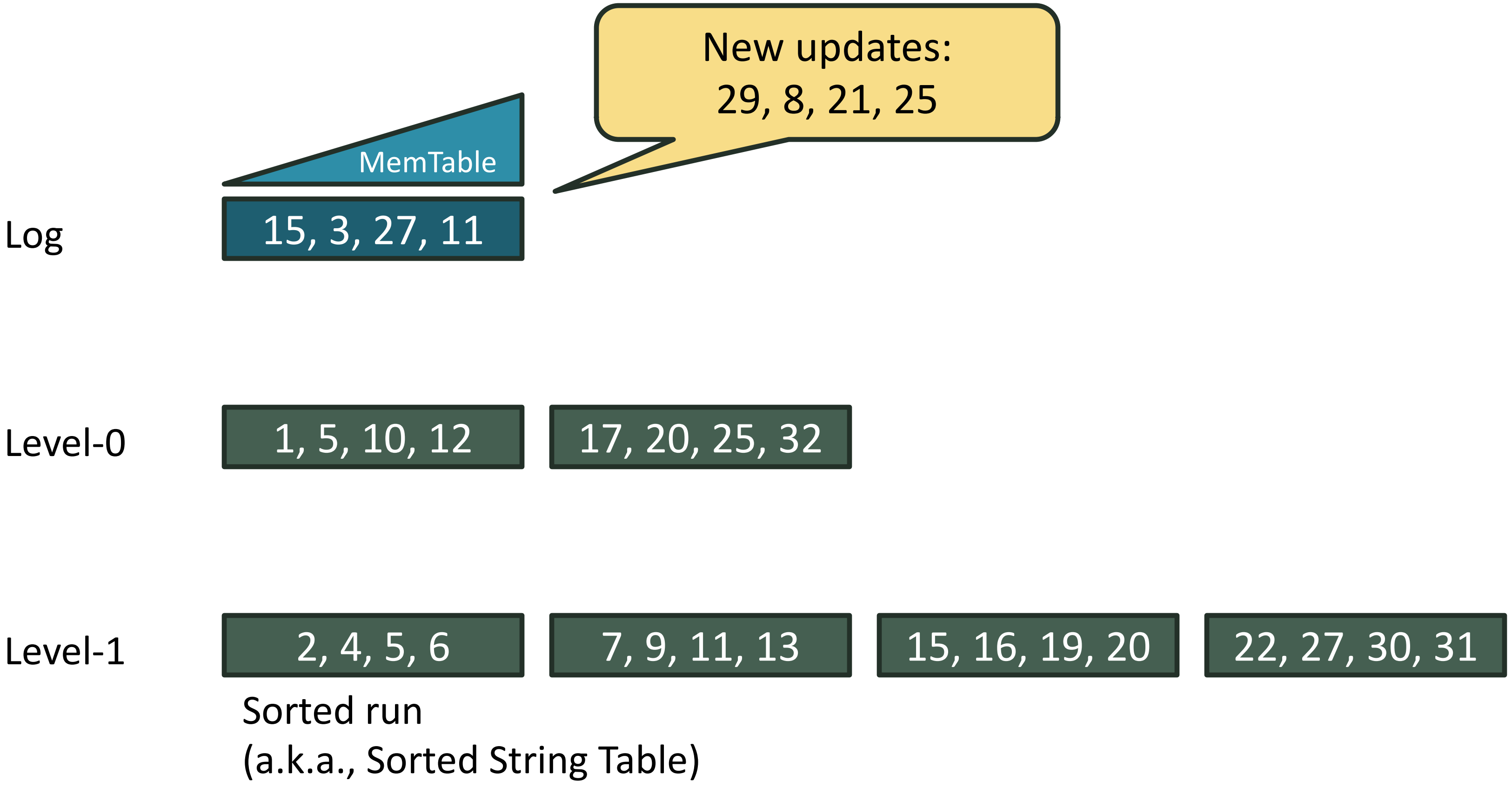
- Basic algorithm – write and merge



In this example:
- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

- Basic algorithm – write and merge



In this example:

- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree)

(P. O'Neil et al. 1996)

- Basic algorithm – write and merge

MemTable

Log: 15, 3, 27, 11, 29, 8, 21, 25 — Just append

Level-0: 1, 5, 10, 12 | 17, 20, 25, 32

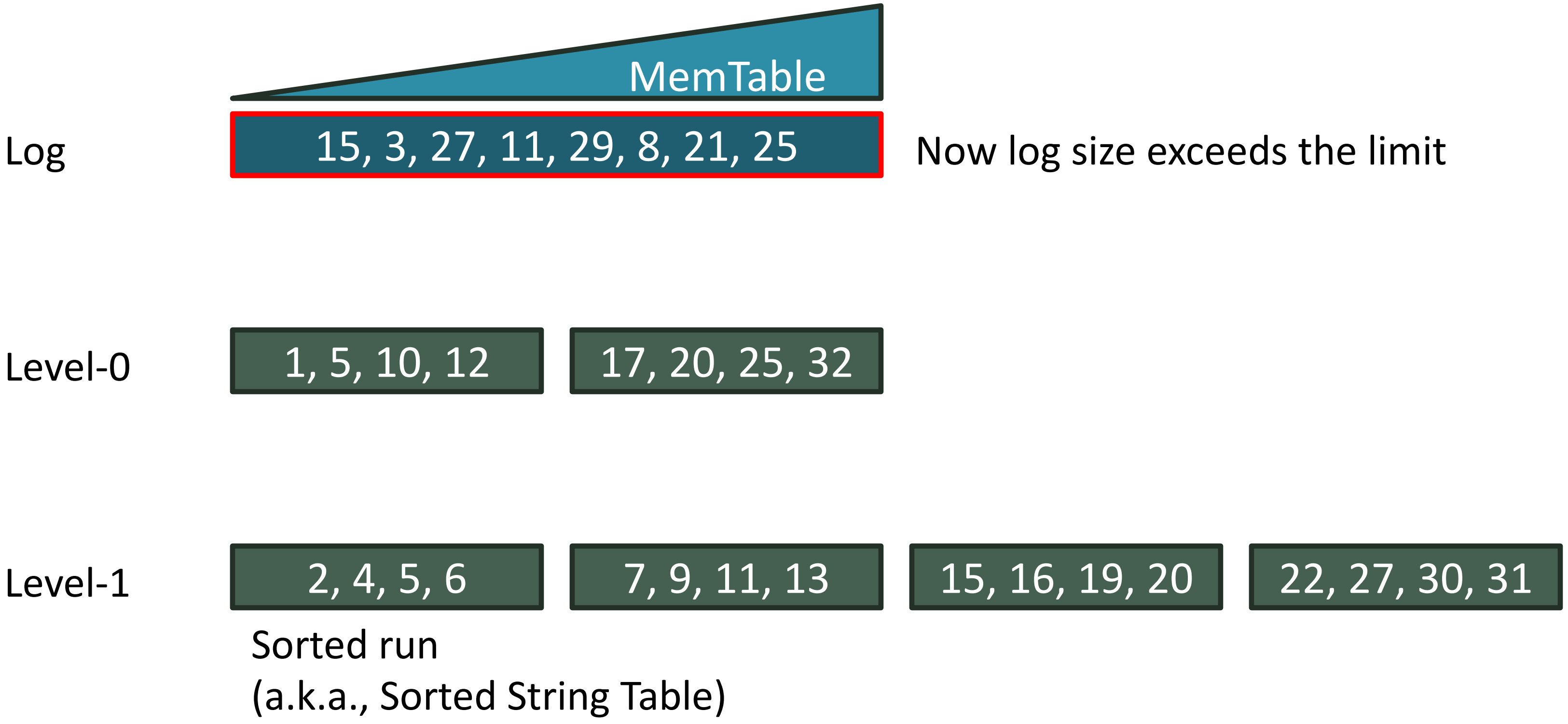
Level-1: 2, 4, 5, 6 | 7, 9, 11, 13 | 15, 16, 19, 20 | 22, 27, 30, 31

Sorted run
(a.k.a., Sorted String Table)

In this example:
- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

- Basic algorithm – write and merge

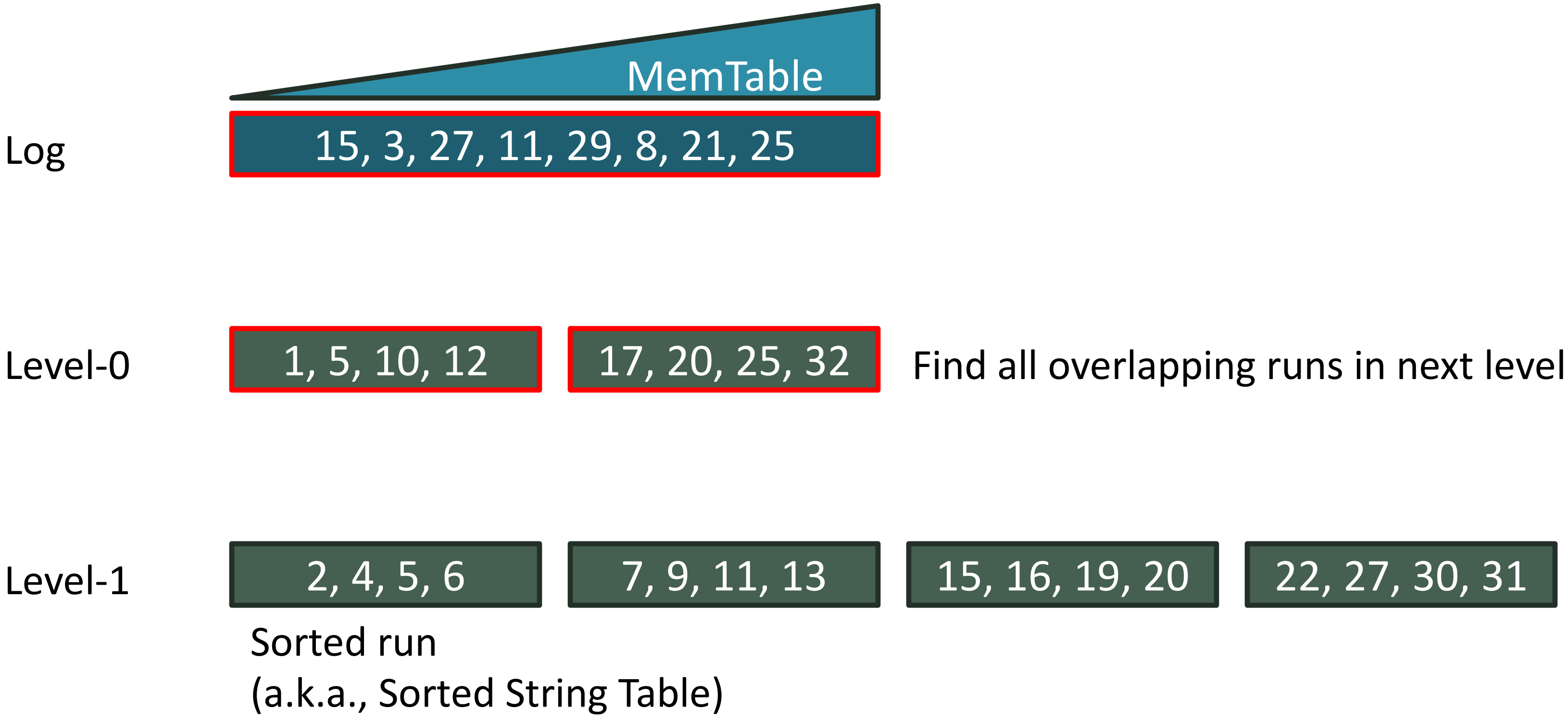


In this example:
- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree)

(P. O'Neil et al. 1996)

- Basic algorithm – write and merge



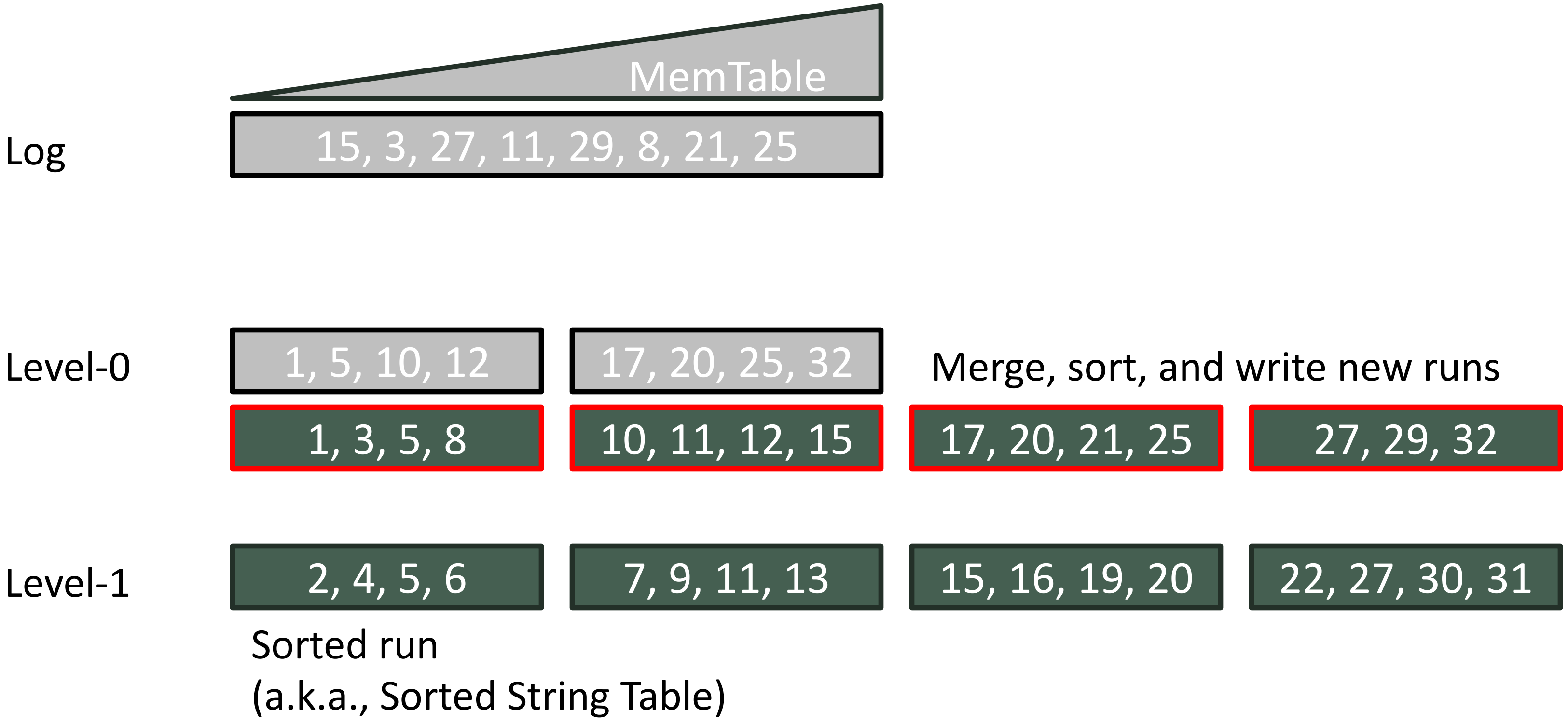
In this example:

- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree)

(P. O'Neil et al. 1996)

- Basic algorithm – write and merge



In this example:

- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

- Basic algorithm – write and merge

Log

Remove old runs & logs

Level-0: | 1, 3, 5, 8 | 10, 11, 12, 15 | 17, 20, 21, 25 | 27, 29, 32 |

Level-1: | 2, 4, 5, 6 | 7, 9, 11, 13 | 15, 16, 19, 20 | 22, 27, 30, 31 |

Sorted run
(a.k.a., Sorted String Table)

In this example:

- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

- Basic algorithm – write and merge

Log

Merge can be cascaded:
now level-0 becomes full

Level-0	1, 3, 5, 8	10, 11, 12, 15	17, 20, 21, 25	27, 29, 32
Level-1	2, 4, 5, 6	7, 9, 11, 13	15, 16, 19, 20	22, 27, 30, 31

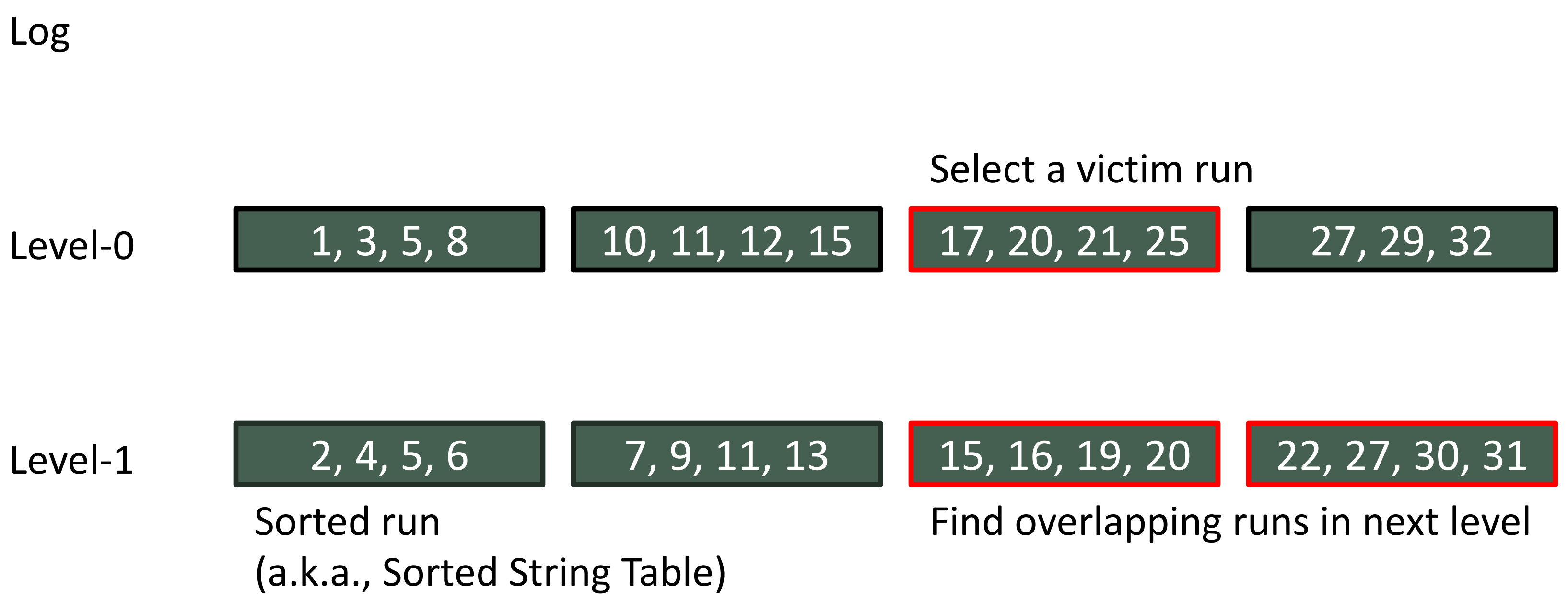
Sorted run
(a.k.a., Sorted String Table)

In this example:

- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

- Basic algorithm – write and merge



In this example:

- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

- Basic algorithm – write and merge

Log

Level-0: 1, 3, 5, 8 | 10, 11, 12, 15 | 17, 20, 21, 25 | 27, 29, 32

Level-1: 2, 4, 5, 6 | 7, 9, 11, 13 | 15, 16, 19, 20 | 22, 27, 30, 31

15, 16, 17, 19 | 20, 21, 22, 25 | 27, 30, 31

Sorted run
(a.k.a., Sorted String Table)

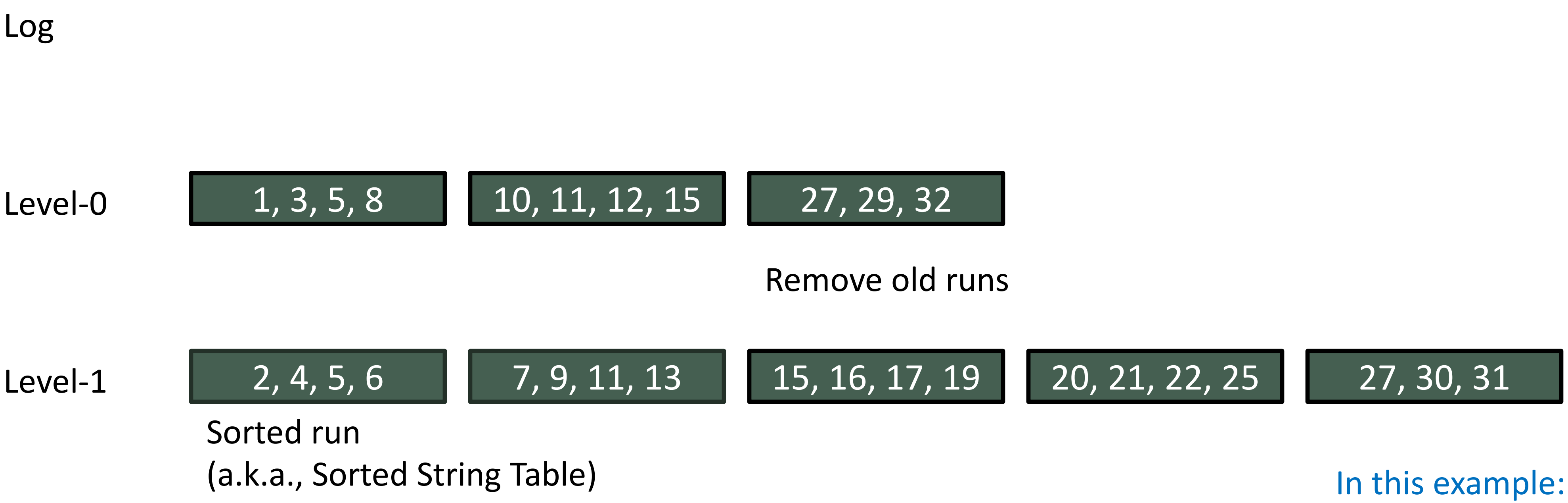
Merge, sort, and write new runs

In this example:

- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

- Basic algorithm – write and merge

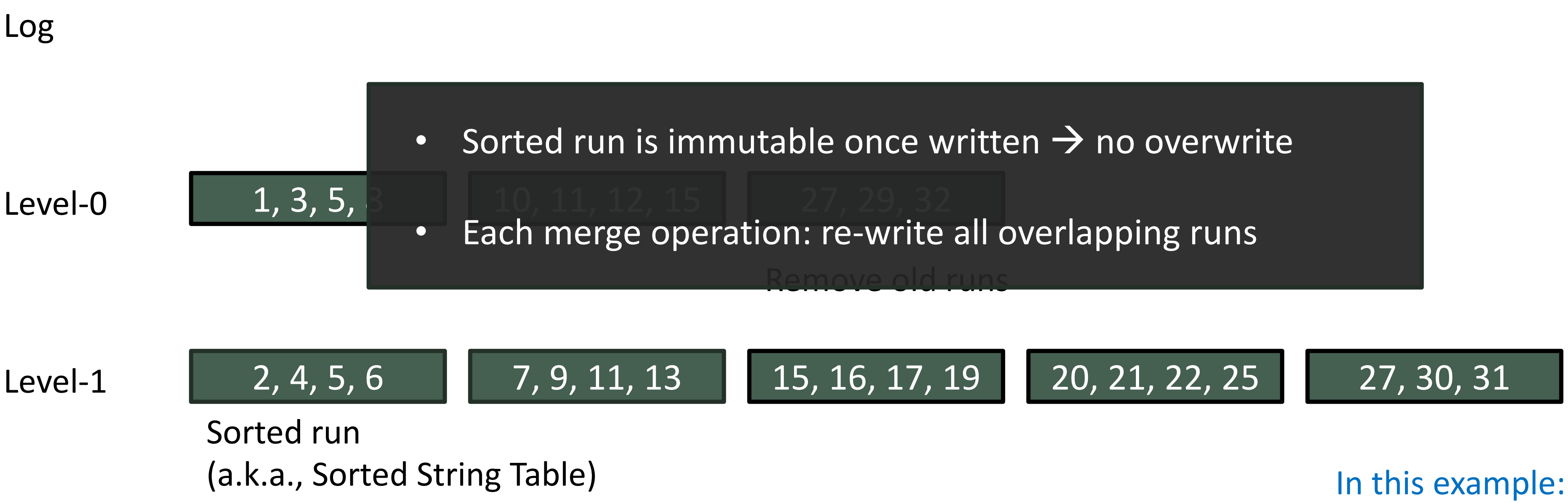


In this example:

- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

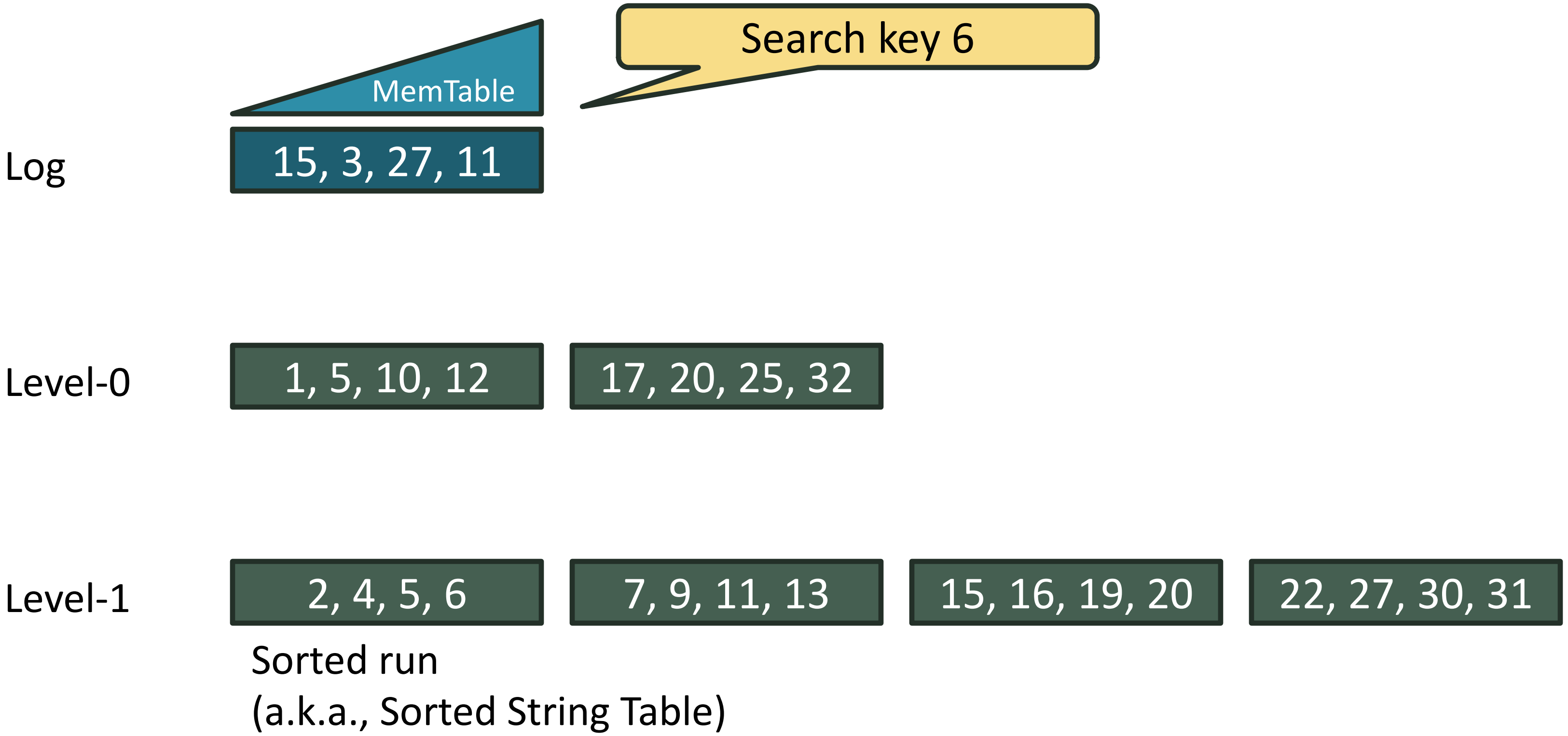
- Basic algorithm – write and merge



In this example:
- Run size limit: 4 keys
- Level-0: 2 runs
- Level-1: 4 runs (2x)

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

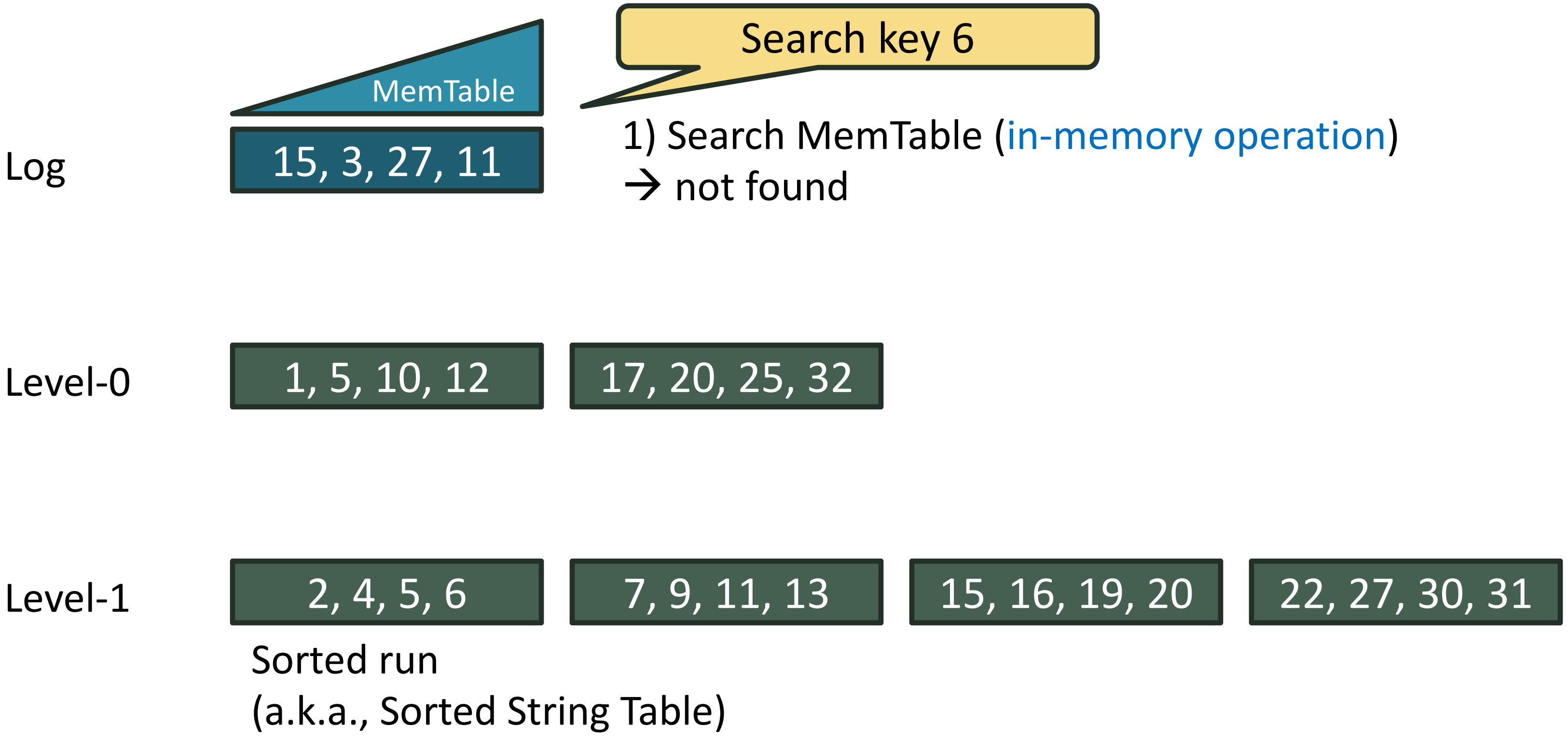
- Basic algorithm – read



- Inner-level search: logarithmic
- Inter-level search: linear

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

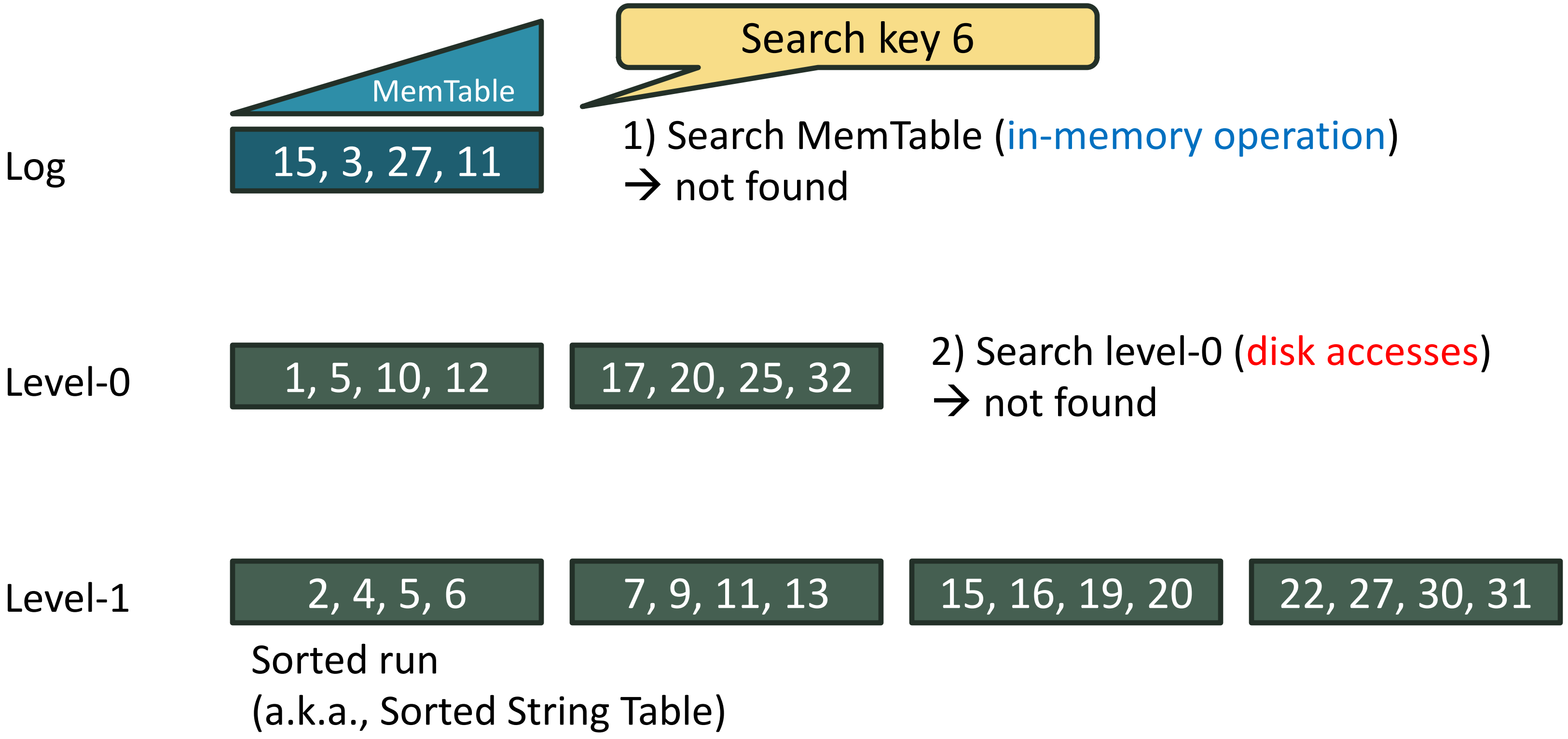
- Basic algorithm – read



- Inner-level search: logarithmic
- Inter-level search: linear

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

- Basic algorithm – read



- Inner-level search: logarithmic
- Inter-level search: linear

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

- Basic algorithm – read

Search key 6

Log | MemTable | 15, 3, 27, 11

1) Search MemTable (in-memory operation)
→ not found

Level-0 | 1, 5, 10, 12 | 17, 20, 25, 32

2) Search level-0 (disk accesses)
→ not found

Level-1 | 2, 4, 5, 6 | 7, 9, 11, 13 | 15, 16, 19, 20 | 22, 27, 30, 31

Sorted run
(a.k.a., Sorted String Table)

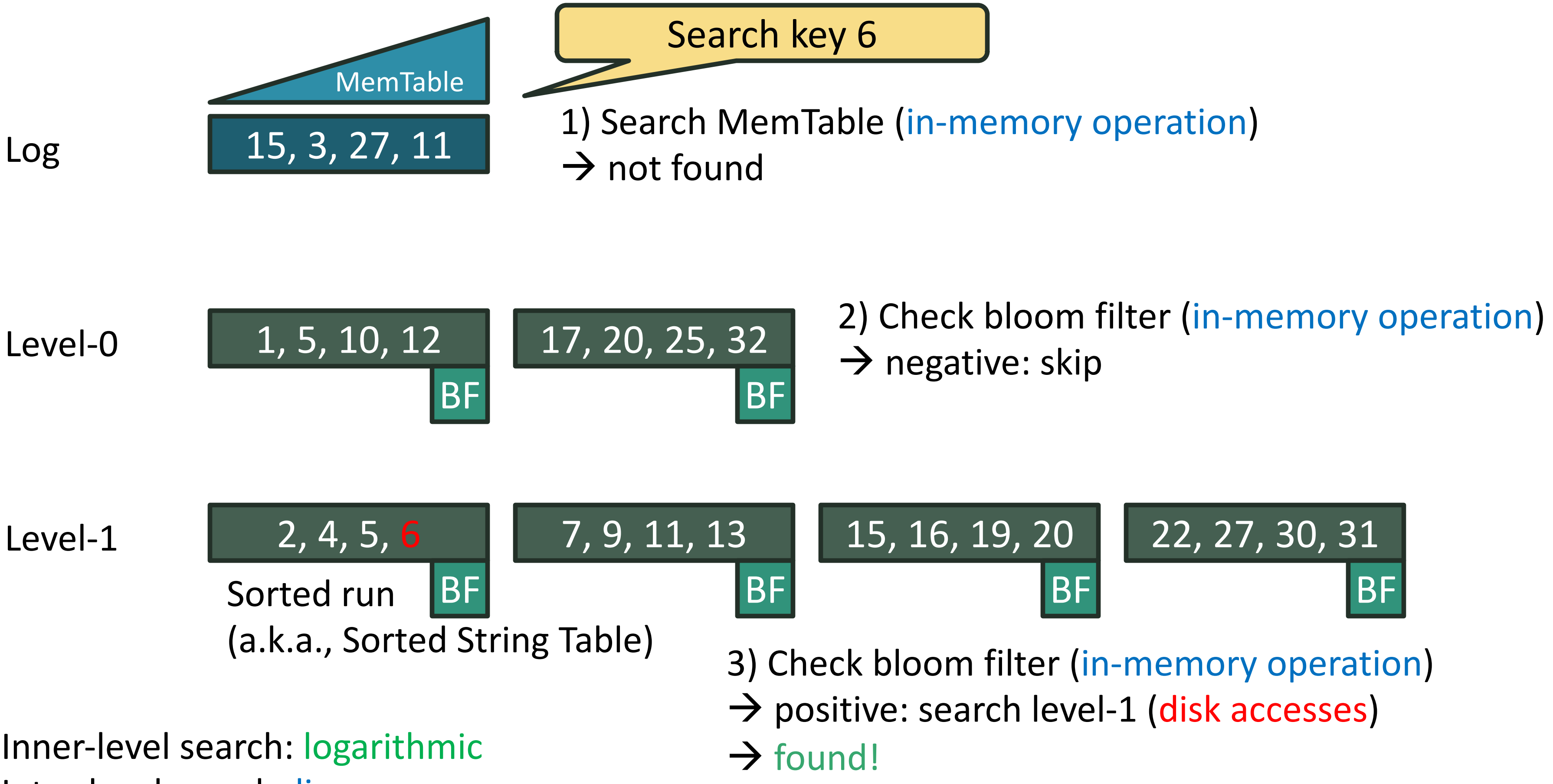
3) Search level-1 (disk accesses)
→ found!

- Inner-level search: logarithmic
- Inter-level search: linear

Search each level one by one

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

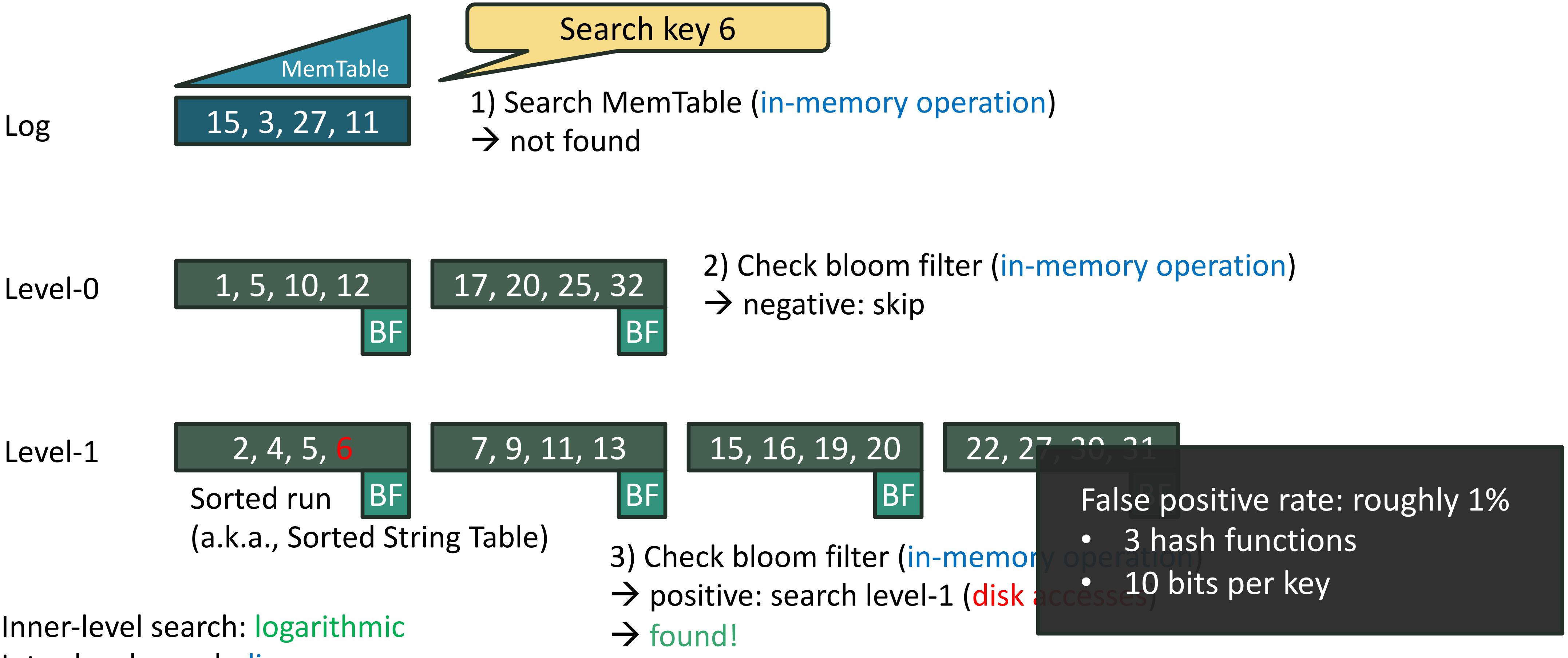
- Basic algorithm – read
  - Bloom filter: skip examining unnecessary runs



- Inner-level search: logarithmic
- Inter-level search: linear

Log-Structured Merge-Tree (LSM-Tree) (P. O'Neil et al. 1996)

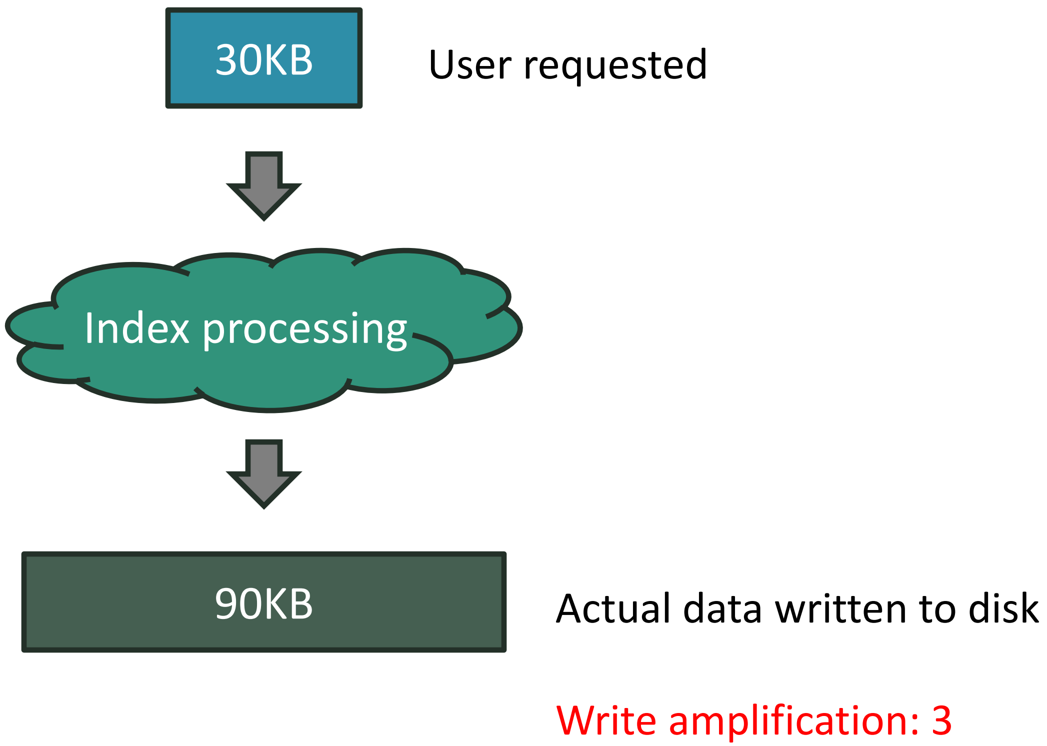
- Basic algorithm – read
  - Bloom filter: skip examining unnecessary runs



- Inner-level search: logarithmic
- Inter-level search: linear

Write Amplification

- Ratio between
  - Amount of data requested by user
  - Amount of data actually written to disk

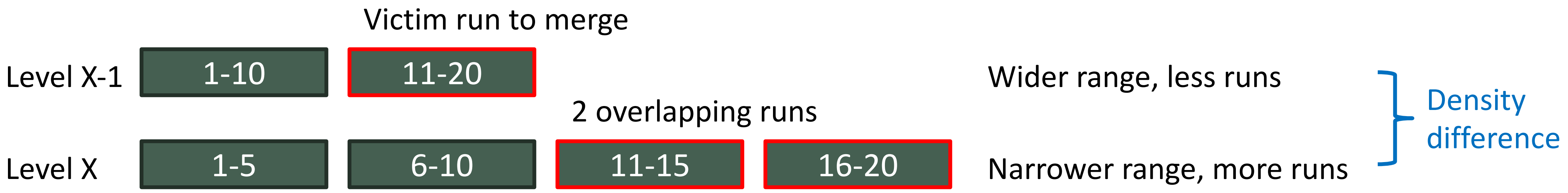


Write Amplification

- Ratio between
  - Amount of data requested by user
  - Amount of data actually written to disk
- LSM-tree: merge amplifies amount of writes

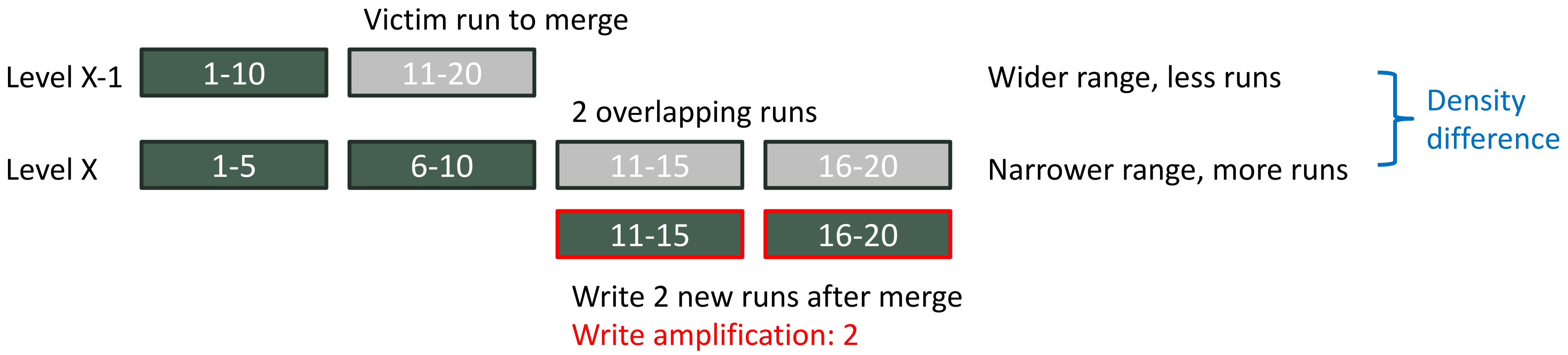
Write Amplification

- Ratio between
  - Amount of data requested by user
  - Amount of data actually written to disk
- LSM-tree: merge amplifies amount of writes



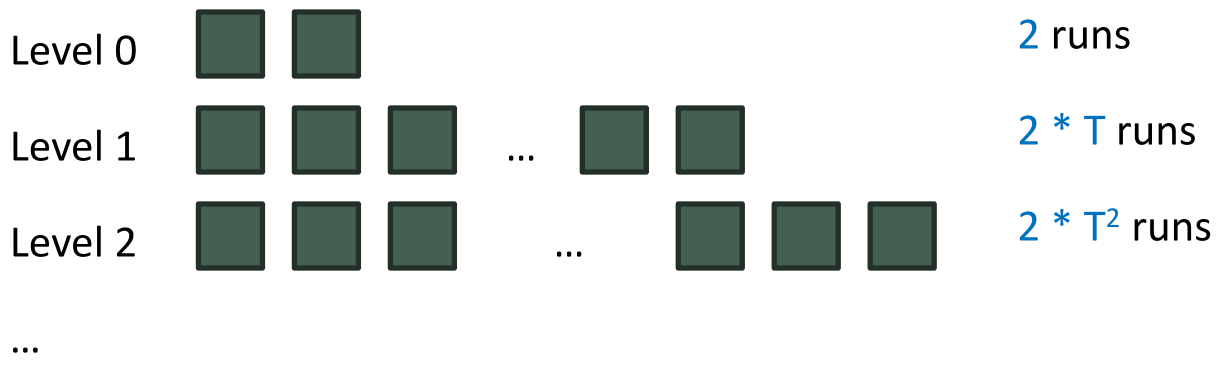
Write Amplification

- Ratio between
  - Amount of data requested by user
  - Amount of data actually written to disk
- LSM-tree: merge amplifies amount of writes



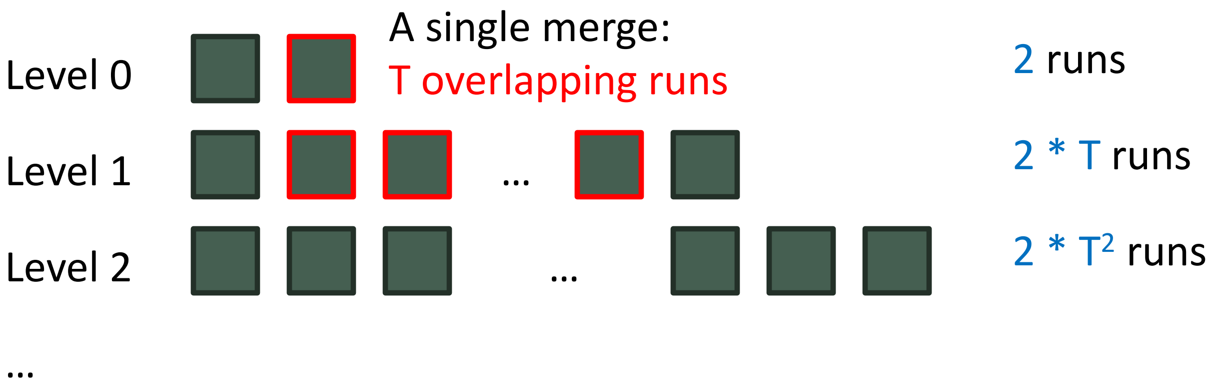
Write Amplification

- Ratio between
  - Amount of data requested by user
  - Amount of data actually written to disk
- LSM-tree: merge amplifies amount of writes
  - T: size ratio of adjacent levels



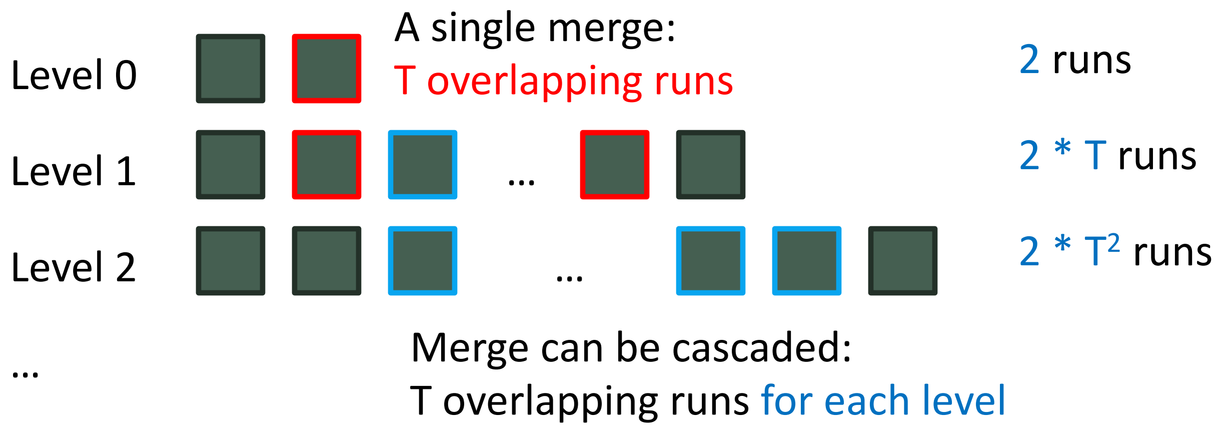
Write Amplification

- Ratio between
  - Amount of data requested by user
  - Amount of data actually written to disk
- LSM-tree: merge amplifies amount of writes
  - T: size ratio of adjacent levels



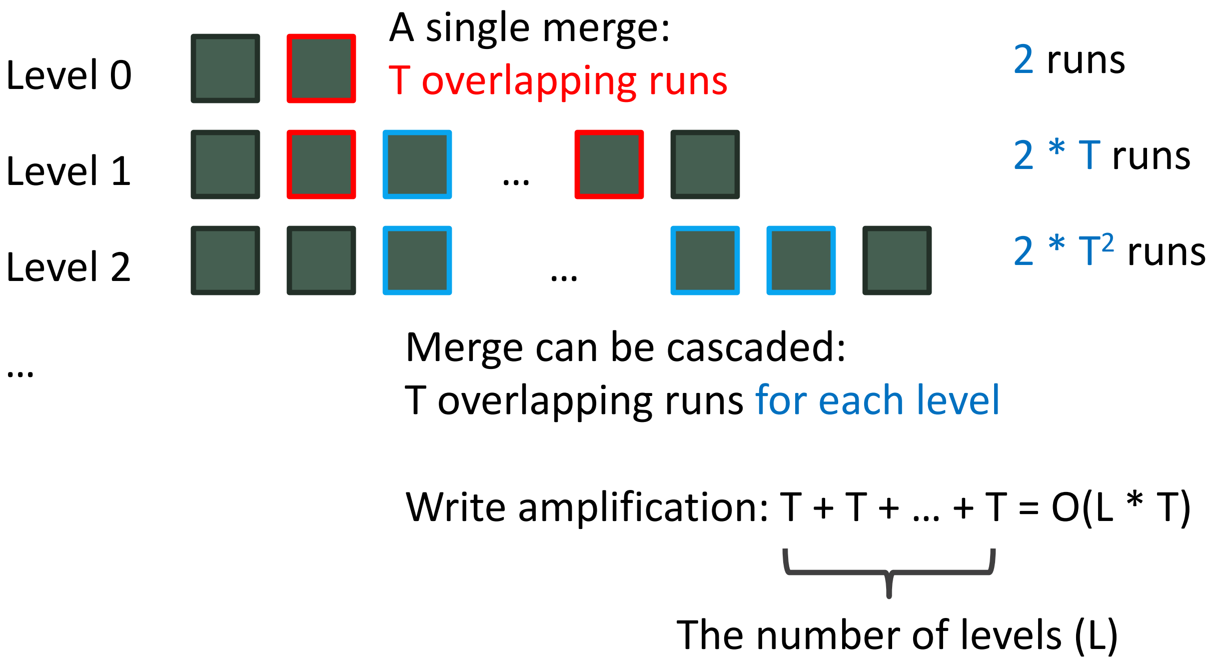
Write Amplification

- Ratio between
  - Amount of data requested by user
  - Amount of data actually written to disk
- LSM-tree: merge amplifies amount of writes
  - T: size ratio of adjacent levels



Write Amplification

- Ratio between
  - Amount of data requested by user
  - Amount of data actually written to disk
- LSM-tree: merge amplifies amount of writes
  - T: size ratio of adjacent levels

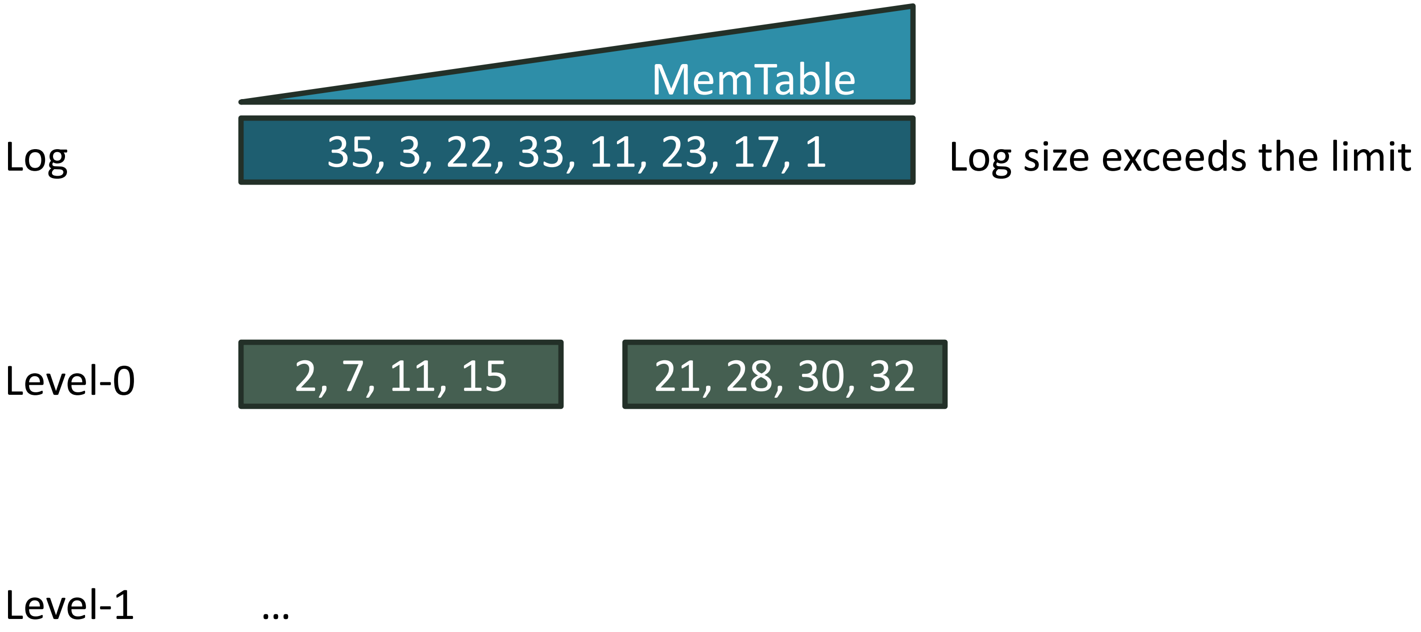


Write Amplification

- Ratio between
  - Amount of data requested by user
  - Amount of data actually written to disk
- LSM-tree: merge amplifies amount of writes
  - T: size ratio of adjacent levels
  - Write amplification: O(L * T)
- Why write amplification matters?
  - User: put 2KB records at 10,000 records/sec rate (20MB/s)
  - Write amplification: 20
  - Actual traffic to disk: 400MB/s
    - Easily hit the upper bound of disk bandwidth
    - Affect read latency or throughput as well

Tiering Merge (H.V. Jagadish et al. VLDB 1997, PebblesDB SOSP 2017, Dostoevsky SIGMOD 2018)

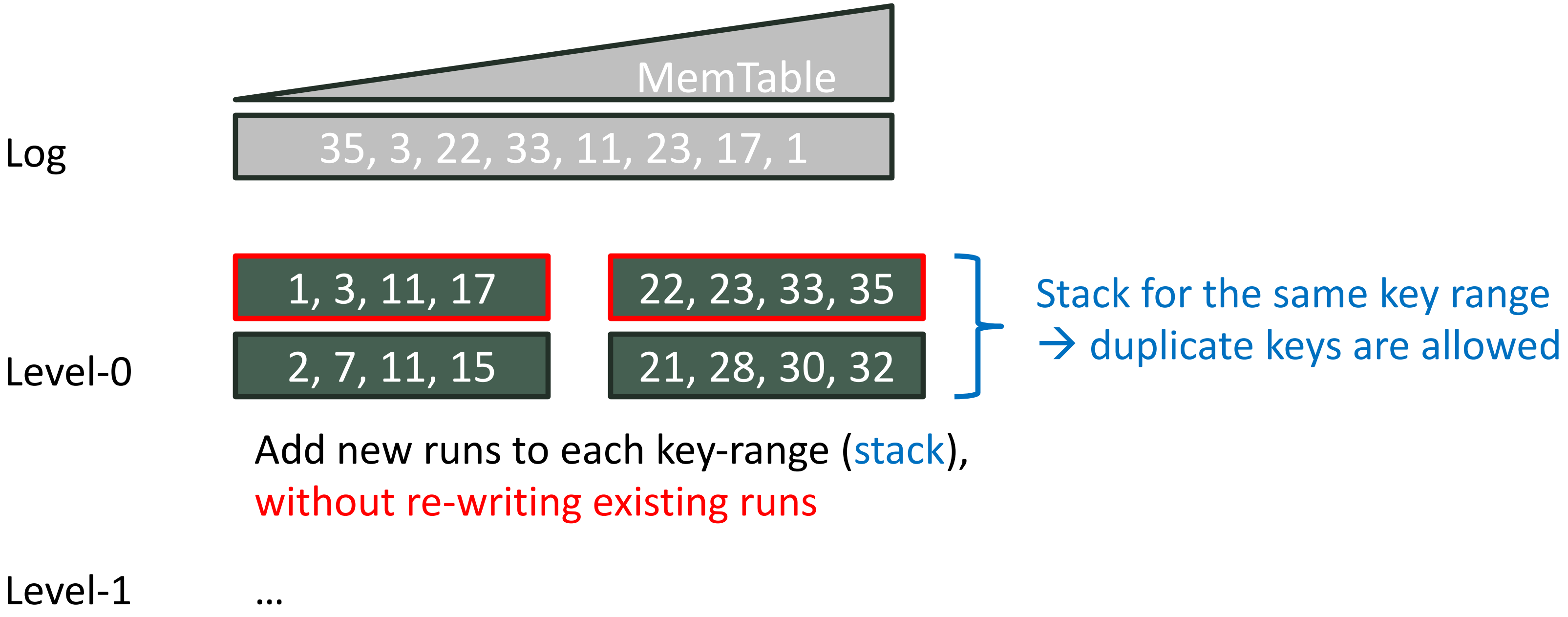
- Delay merge, and keep stacks of runs (for the same key range)



Tiering Merge

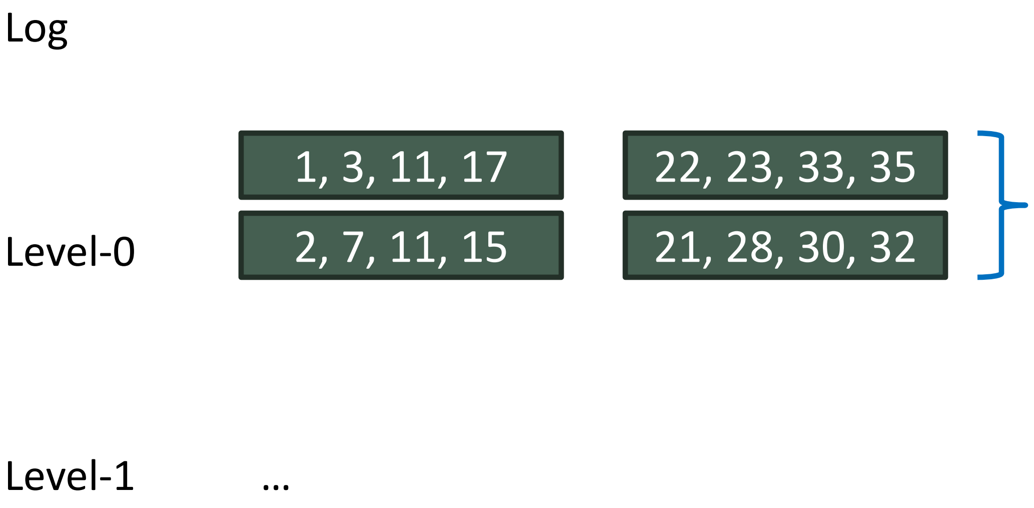
(H.V. Jagadish et al. VLDB 1997, PebblesDB SOSP 2017, Dostoevsky SIGMOD 2018)

- Delay merge, and keep stacks of runs (for the same key range)



Tiering Merge
(H.V. Jagadish et al. VLDB 1997, PebblesDB SOSP 2017, Dostoevsky SIGMOD 2018)

- Delay merge, and keep stacks of runs (for the same key range)



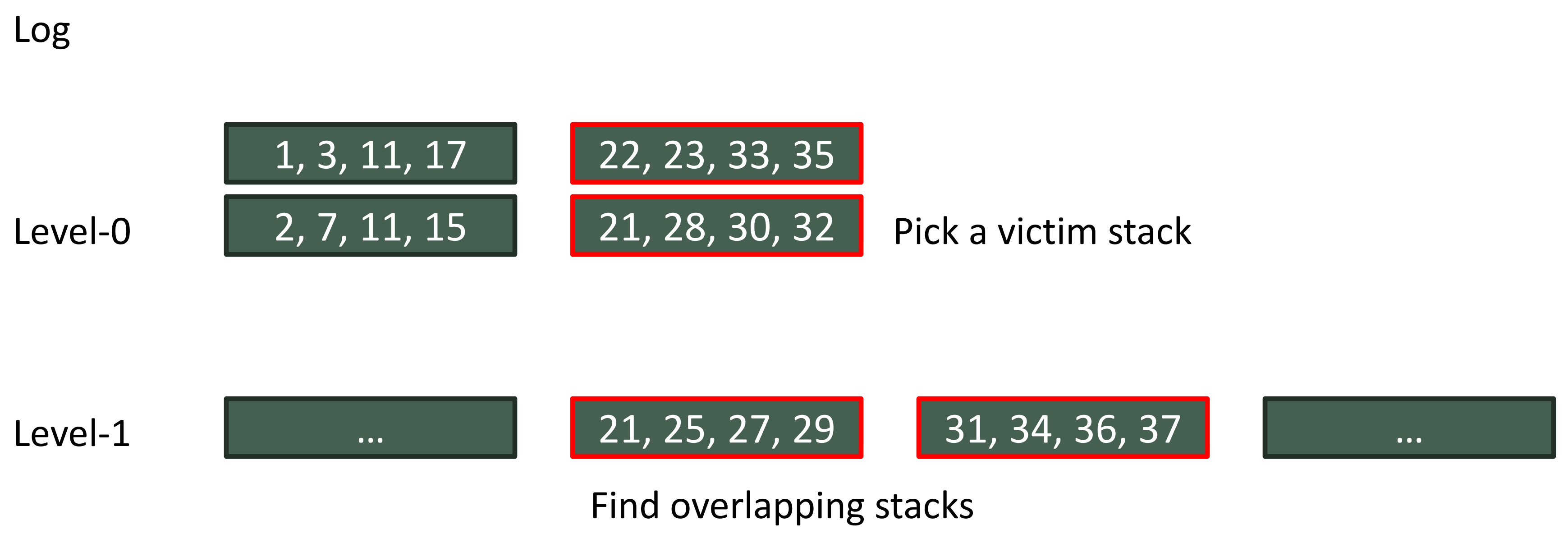
Stack for the same key range
→ duplicate keys are allowed

Stack size limit: up to M runs

Stack exceeds the limit:
merge into next level

Tiering Merge (H.V. Jagadish et al. VLDB 1997, PebblesDB SOSP 2017, Dostoevsky SIGMOD 2018)

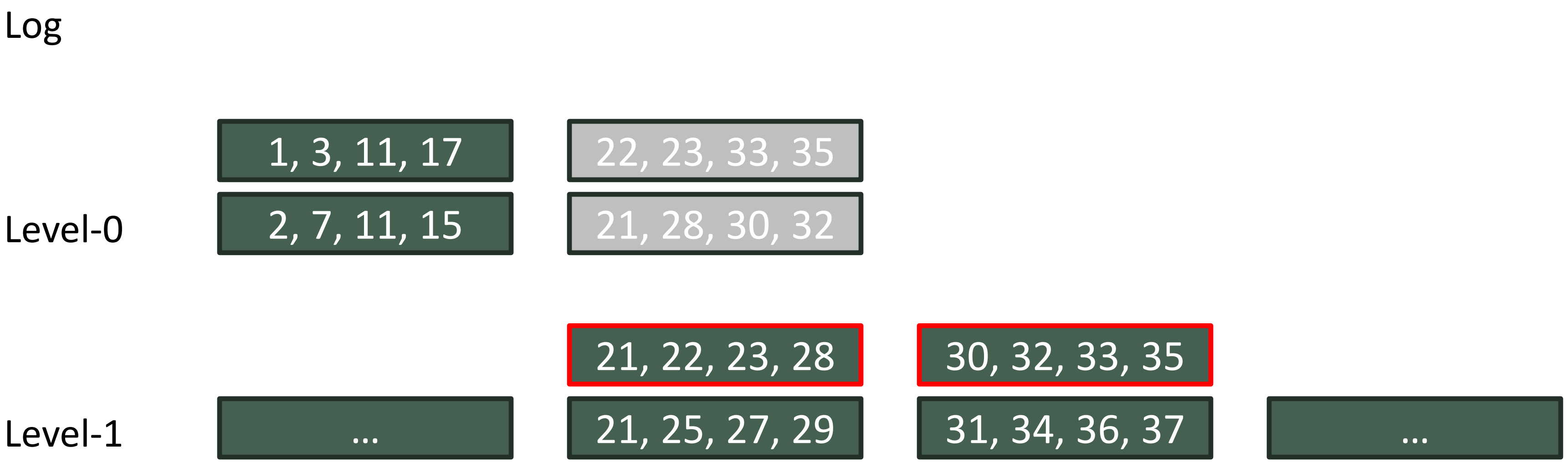
- Delay merge, and keep stacks of runs (for the same key range)



Tiering Merge

(H.V. Jagadish et al. VLDB 1997, PebblesDB SOSP 2017, Dostoevsky SIGMOD 2018)

- Delay merge, and keep stacks of runs (for the same key range)

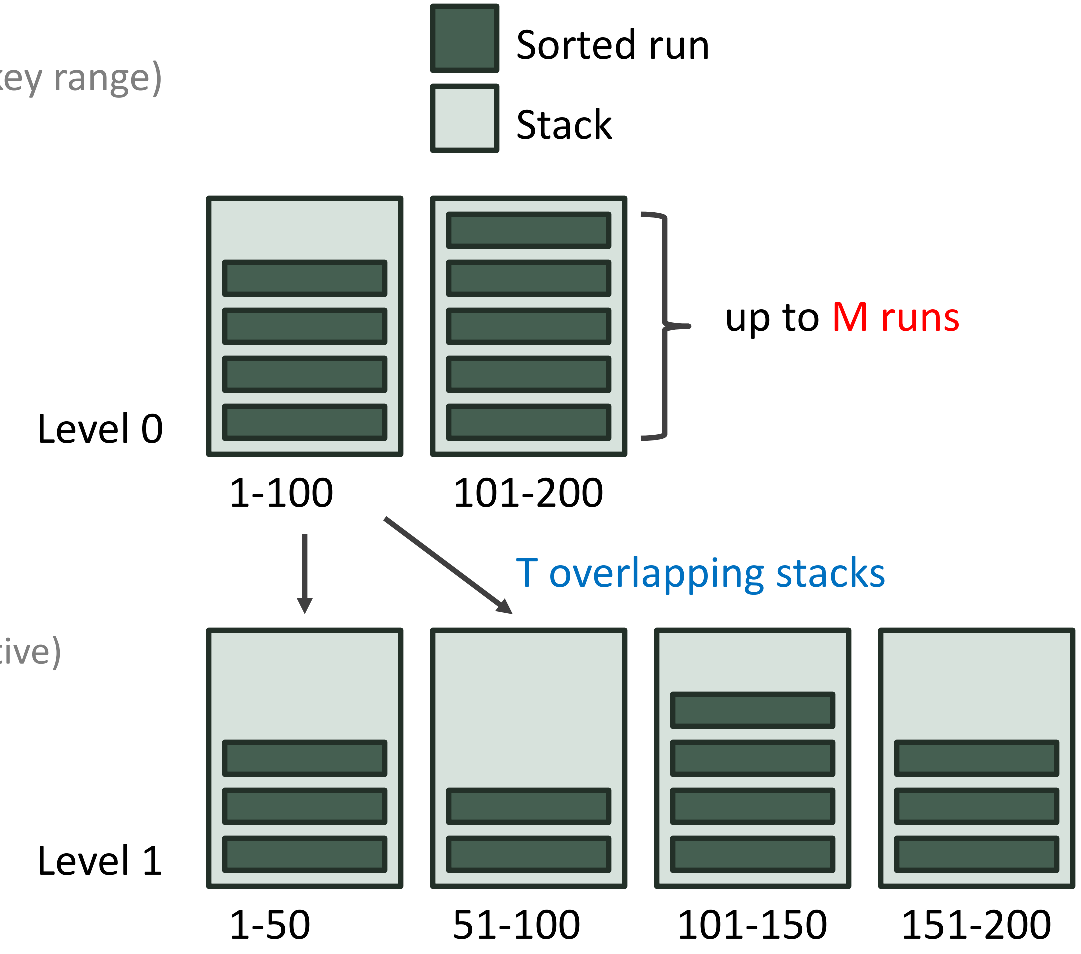


Add new runs to each stack,
without re-writing existing runs

Tiering Merge

(H.V. Jagadish et al. VLDB 1997, PebblesDB SOSP 2017, Dostoevsky SIGMOD 2018)

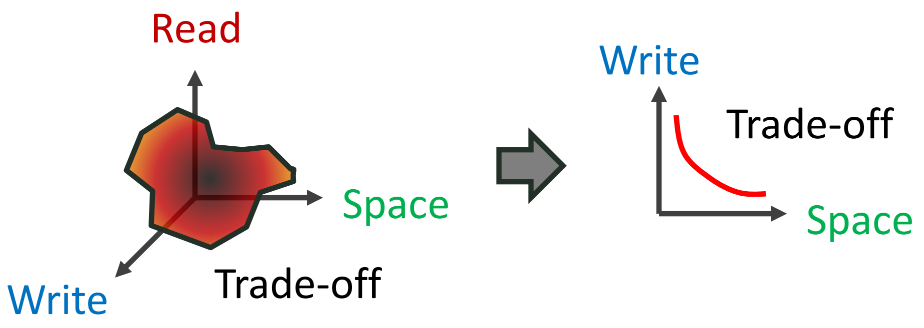
- Delay merge, and keep stacks of runs (for the same key range)
- Trade-offs
  - Stack size limit: up to M runs
  - Write cost
    - Can delay merge M times
    - M times smaller write amplification
  - Read cost
    - M times more searches
    - (even with Bloom filter, more searches upon false positive)
  - Space cost
    - M times more space



Write cost vs. read cost and space cost

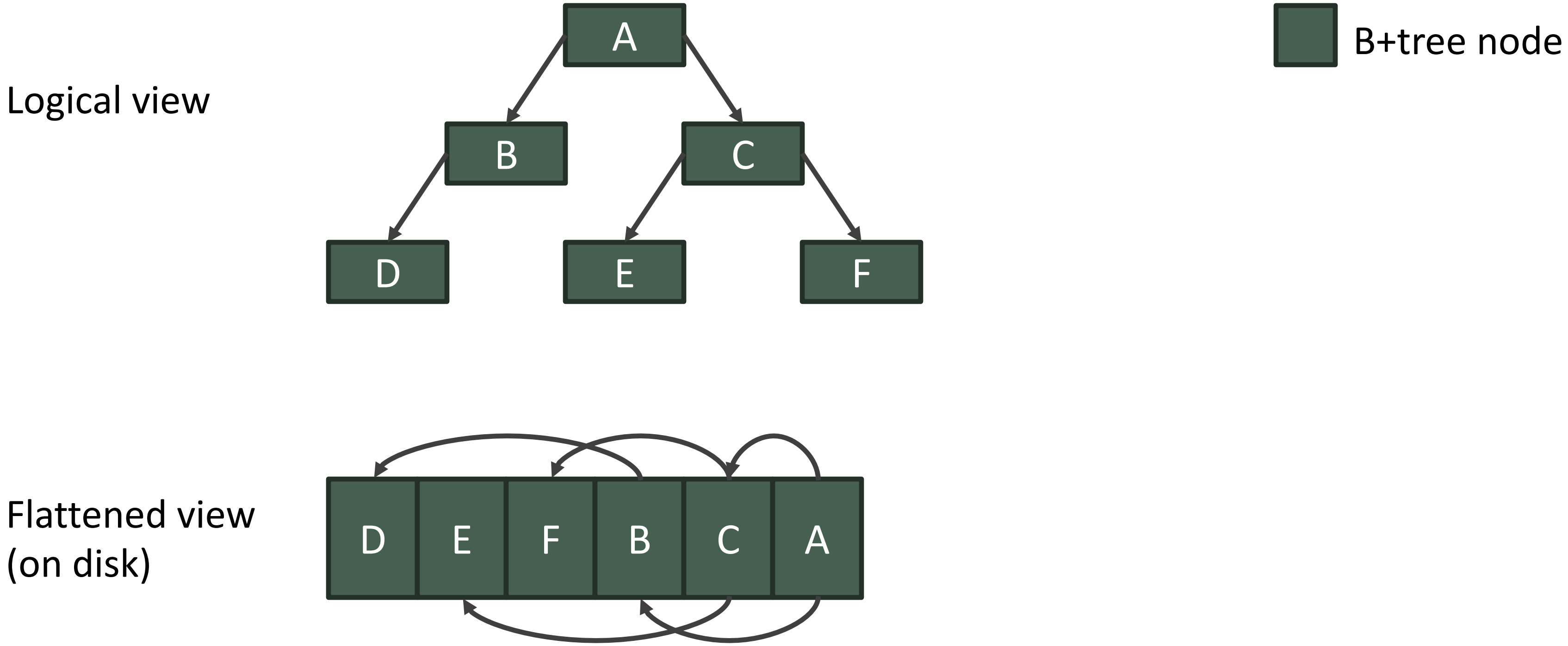
Jungle: Motivation

- What if we can
  - Reduce write cost
  - Without sacrificing read cost
- Trade-off change: 3-dimensional → 2-dimensional
- How?
  - Replace **each stack** with **copy-on-write B+tree**



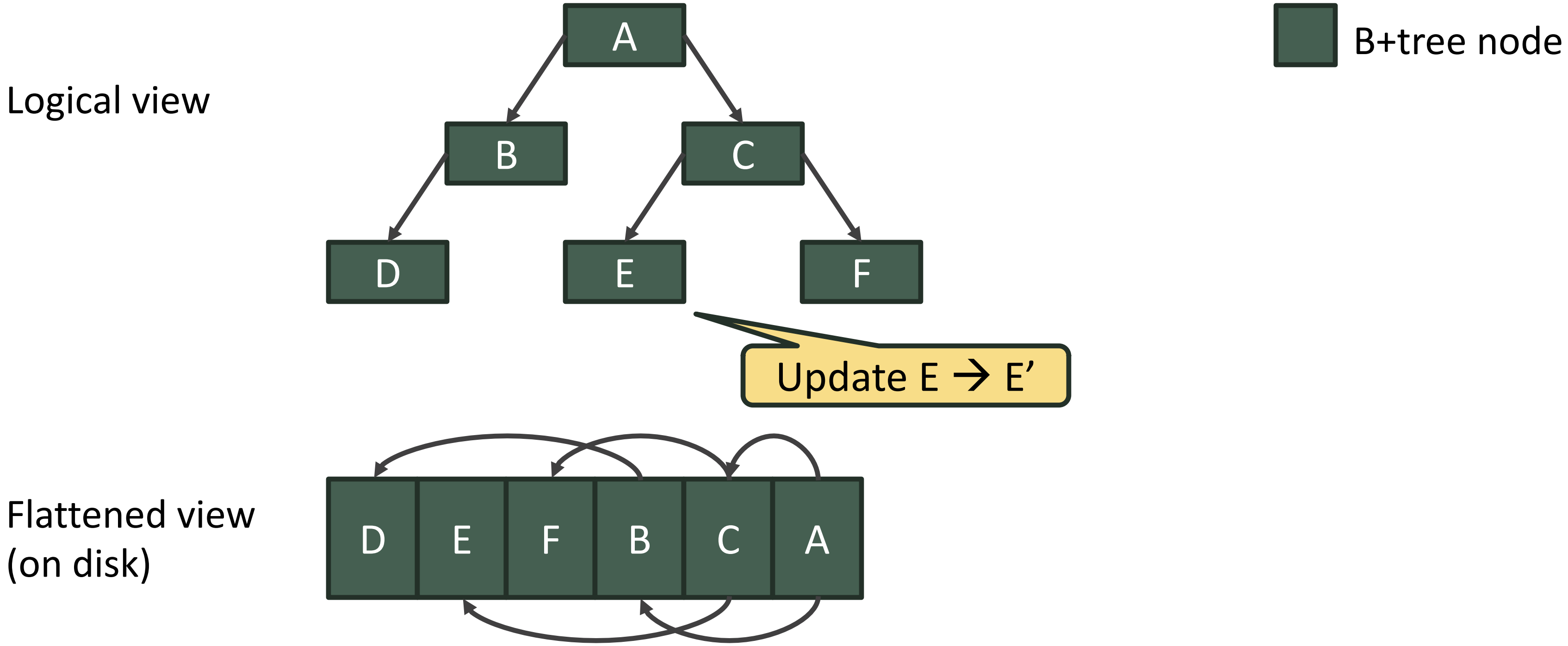
Copy-On-Write (CoW) B+Tree

- Logically the same, but about how to write B+tree nodes to disk
  - Out-of-update manner: no overwrite, but append
- All nodes are immutable, written in chronological order



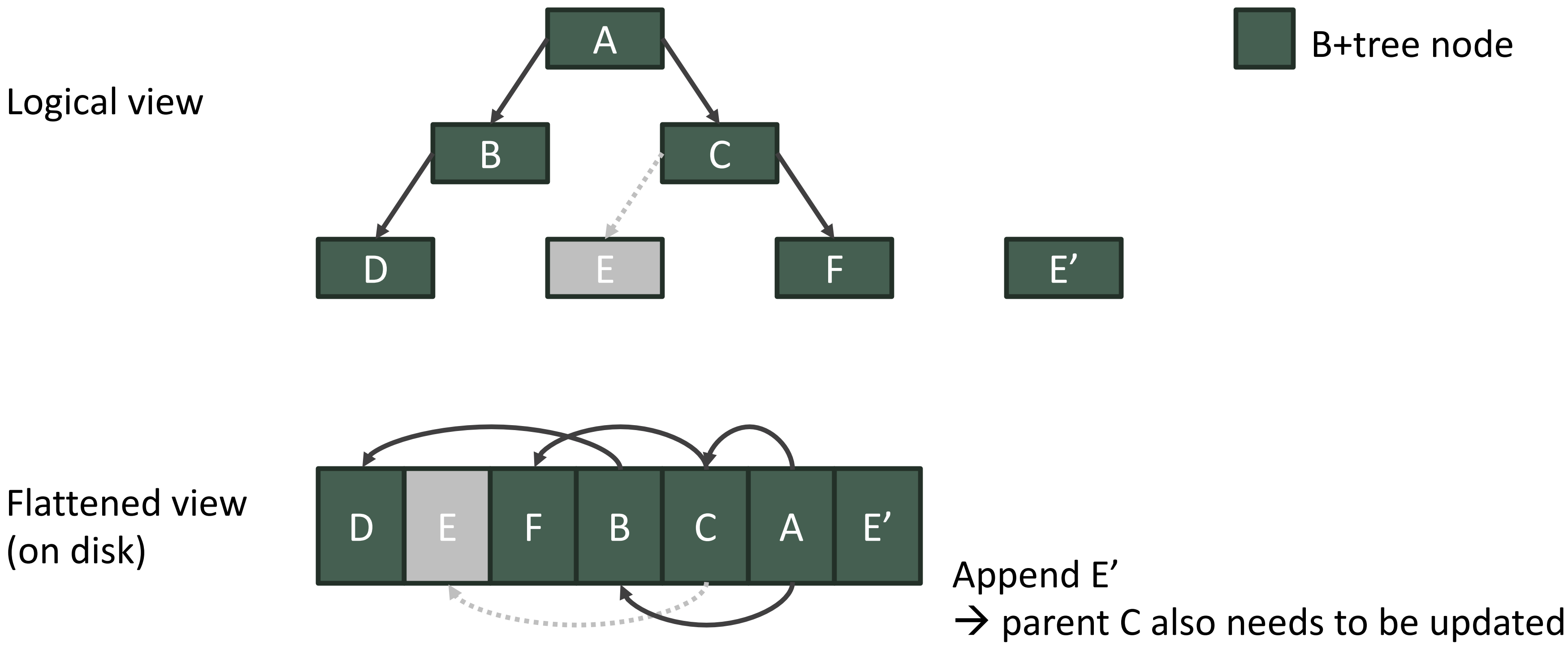
Copy-On-Write (CoW) B+Tree

- Logically the same, but about how to write B+tree nodes to disk
  - Out-of-update manner: no overwrite, but append
- All nodes are immutable, written in chronological order



Copy-On-Write (CoW) B+Tree

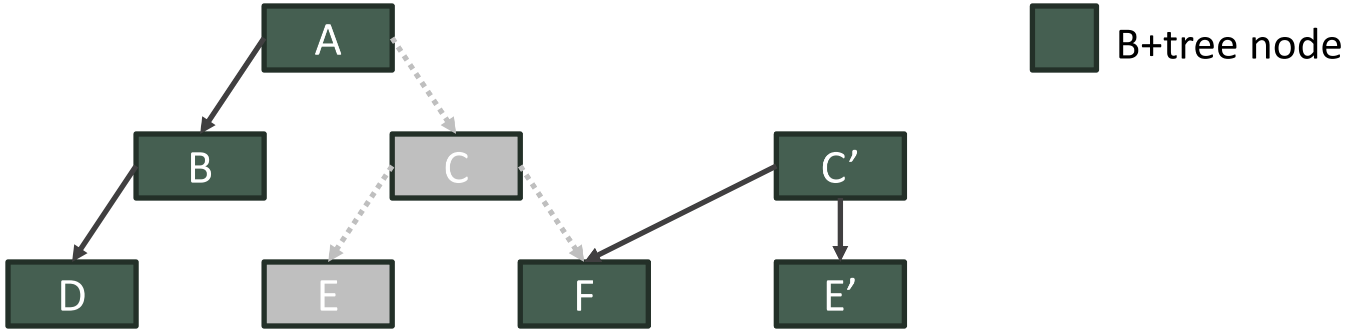
- Logically the same, but about how to write B+tree nodes to disk
  - Out-of-update manner: no overwrite, but append
- All nodes are immutable, written in chronological order



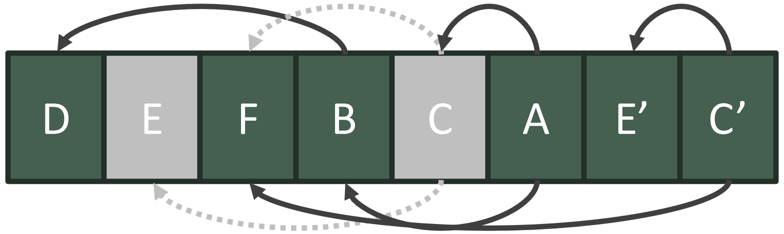
Copy-On-Write (CoW) B+Tree

- Logically the same, but about how to write B+tree nodes to disk
  - Out-of-update manner: no overwrite, but append
- All nodes are immutable, written in chronological order

Logical view



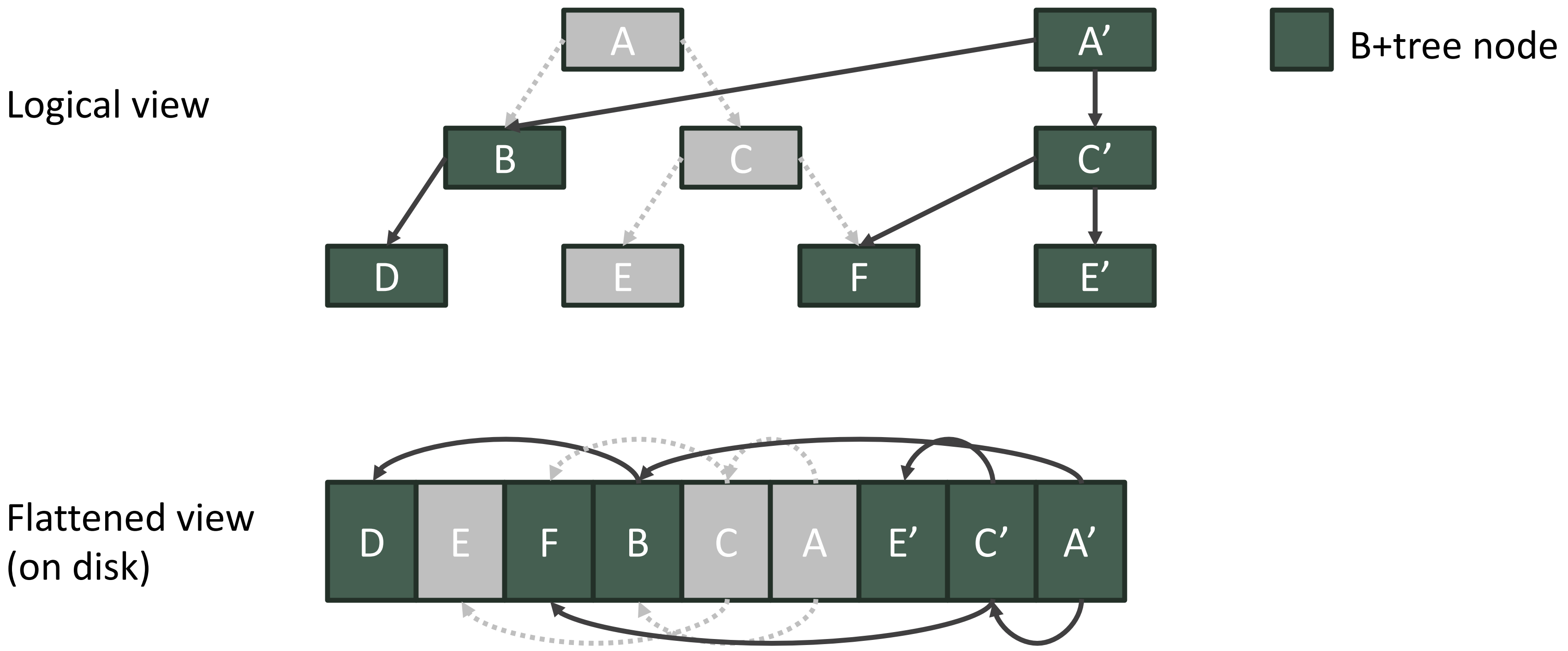
Flattened view
(on disk)



Append C'
→ parent A also needs to be updated

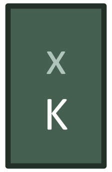
Copy-On-Write (CoW) B+Tree

- Logically the same, but about how to write B+tree nodes to disk
  - Out-of-update manner: no overwrite, but append
- All nodes are immutable, written in chronological order

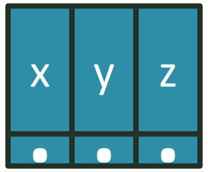


Copy-On-Write (CoW) B+Tree

- Batching + decoupling value from B+tree node
  - To reduce write amplification + to make B+tree compact
  - Append values first, and then append updated B+tree nodes
  - B+tree leaf nodes: contain {key, pointer to value} pairs



K: value of key x

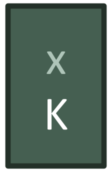


B+tree node with key x, y, and z

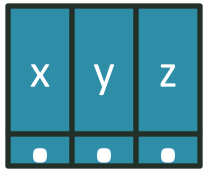
Insert {10, A}, {15, B}, {27, C}, {50, D}

Copy-On-Write (CoW) B+Tree

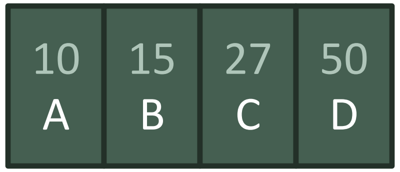
- Batching + decoupling value from B+tree node
  - To reduce write amplification + to make B+tree compact
  - Append values first, and then append updated B+tree nodes
  - B+tree leaf nodes: contain {key, pointer to value} pairs



K: value of key x



B+tree node with key x, y, and z

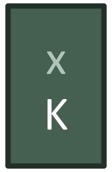


Append values (in key order)

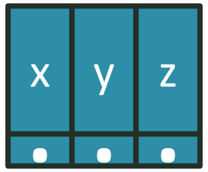
Insert {10, A}, {15, B}, {27, C}, {50, D}

Copy-On-Write (CoW) B+Tree

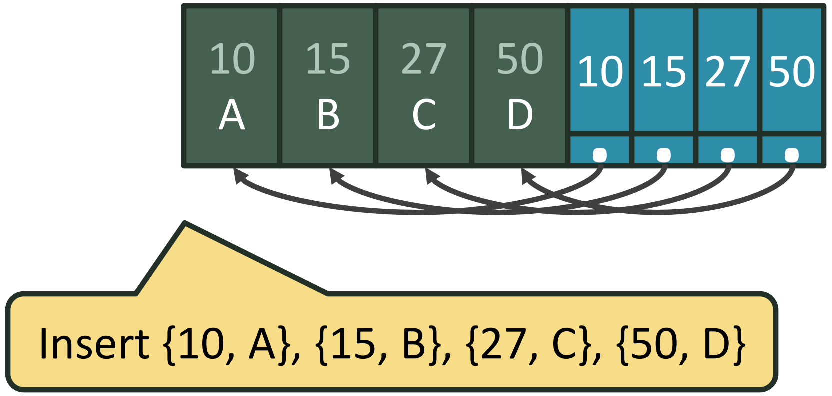
- Batching + decoupling value from B+tree node
  - To reduce write amplification + to make B+tree compact
  - Append values first, and then append updated B+tree nodes
  - B+tree leaf nodes: contain {key, pointer to value} pairs



K: value of key x



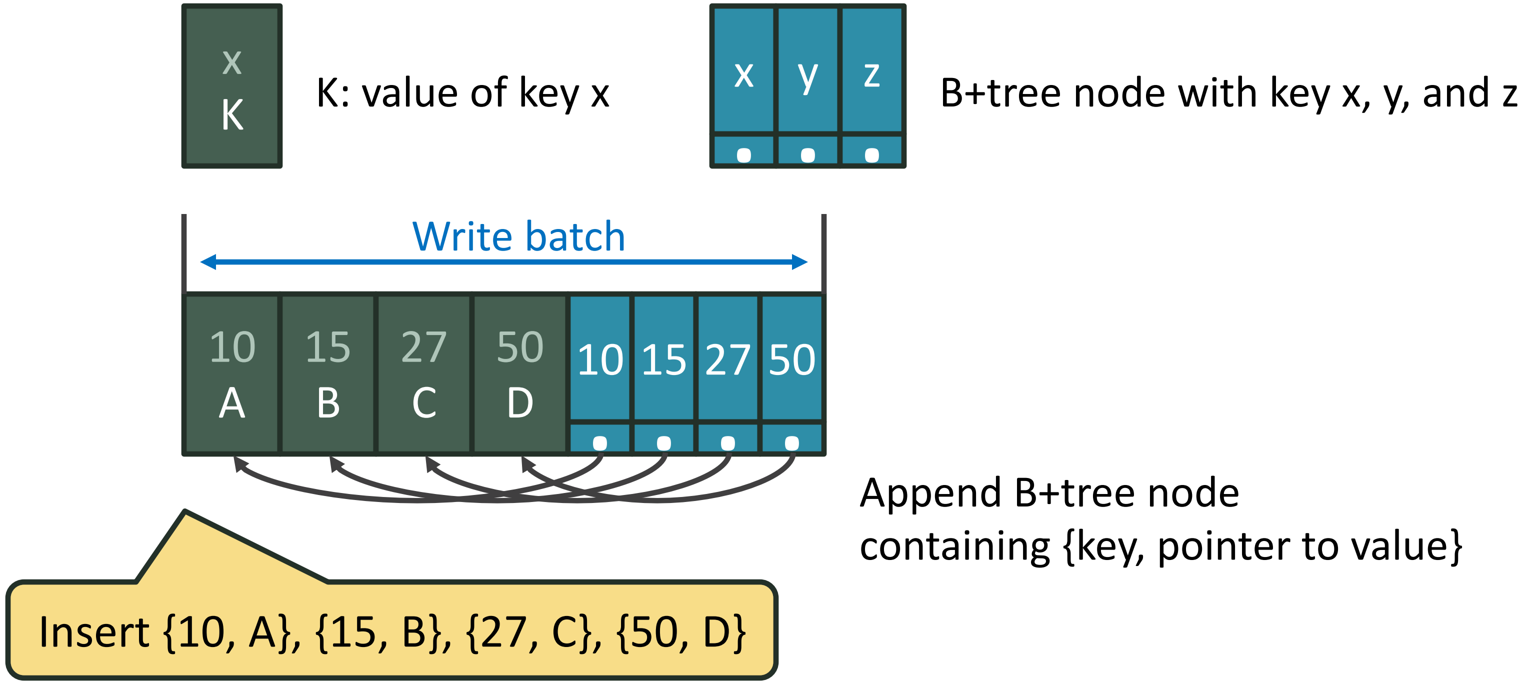
B+tree node with key x, y, and z



Append B+tree node
containing {key, pointer to value}

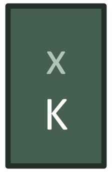
Copy-On-Write (CoW) B+Tree

- Batching + decoupling value from B+tree node
  - To reduce write amplification + to make B+tree compact
  - Append values first, and then append updated B+tree nodes
  - B+tree leaf nodes: contain {key, pointer to value} pairs

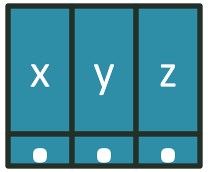


Copy-On-Write (CoW) B+Tree

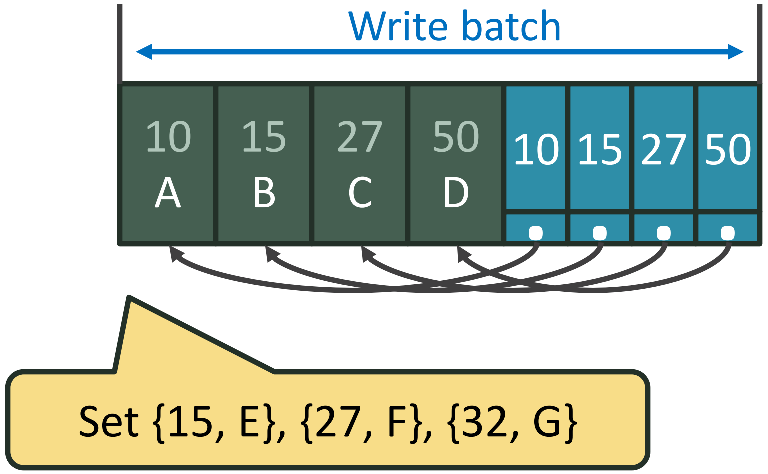
- Batching + decoupling value from B+tree node
  - To reduce write amplification + to make B+tree compact
  - Append values first, and then append updated B+tree nodes
  - B+tree leaf nodes: contain {key, pointer to value} pairs



K: value of key x

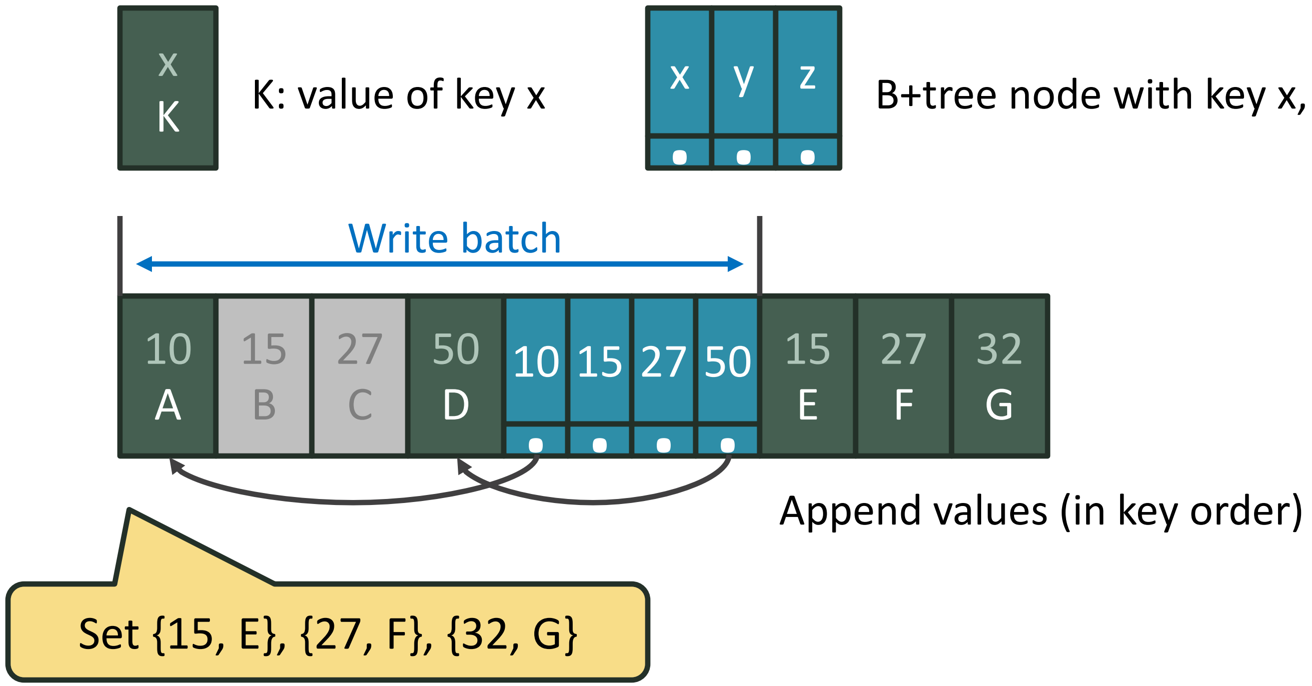


B+tree node with key x, y, and z



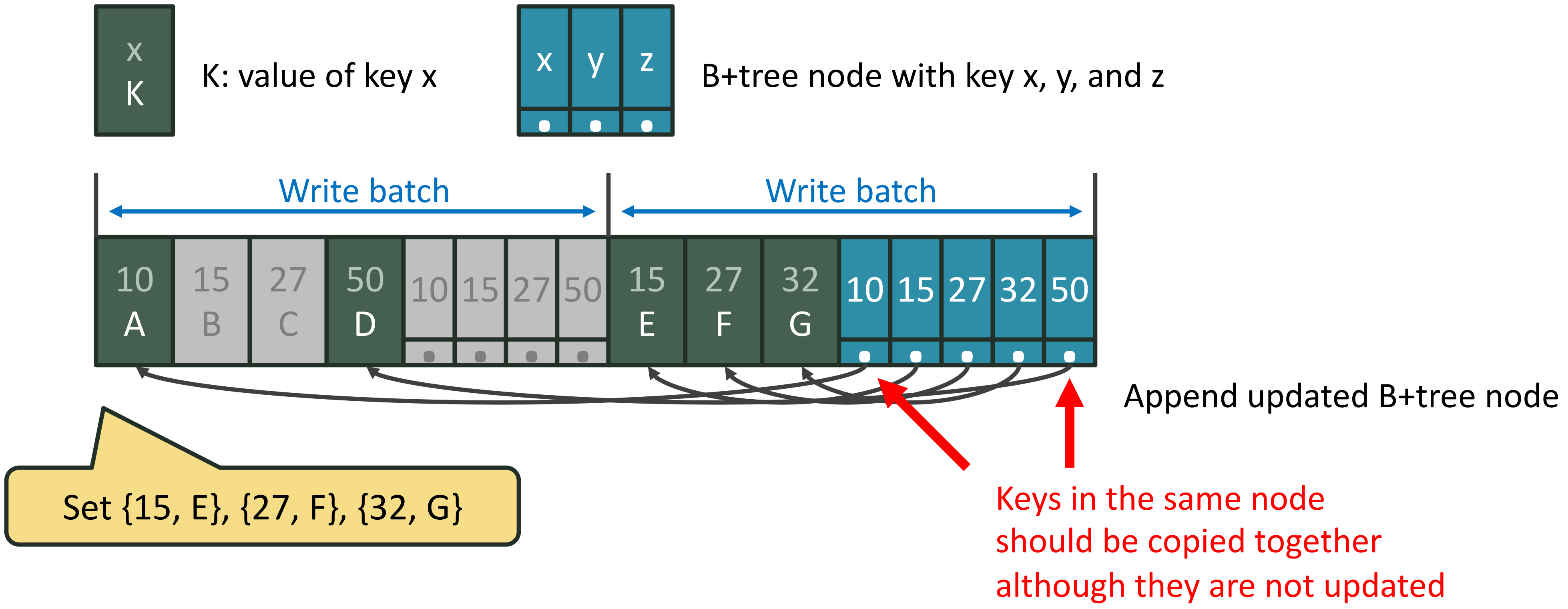
Copy-On-Write (CoW) B+Tree

- Batching + decoupling value from B+tree node
  - To reduce write amplification + to make B+tree compact
  - Append values first, and then append updated B+tree nodes
  - B+tree leaf nodes: contain {key, pointer to value} pairs



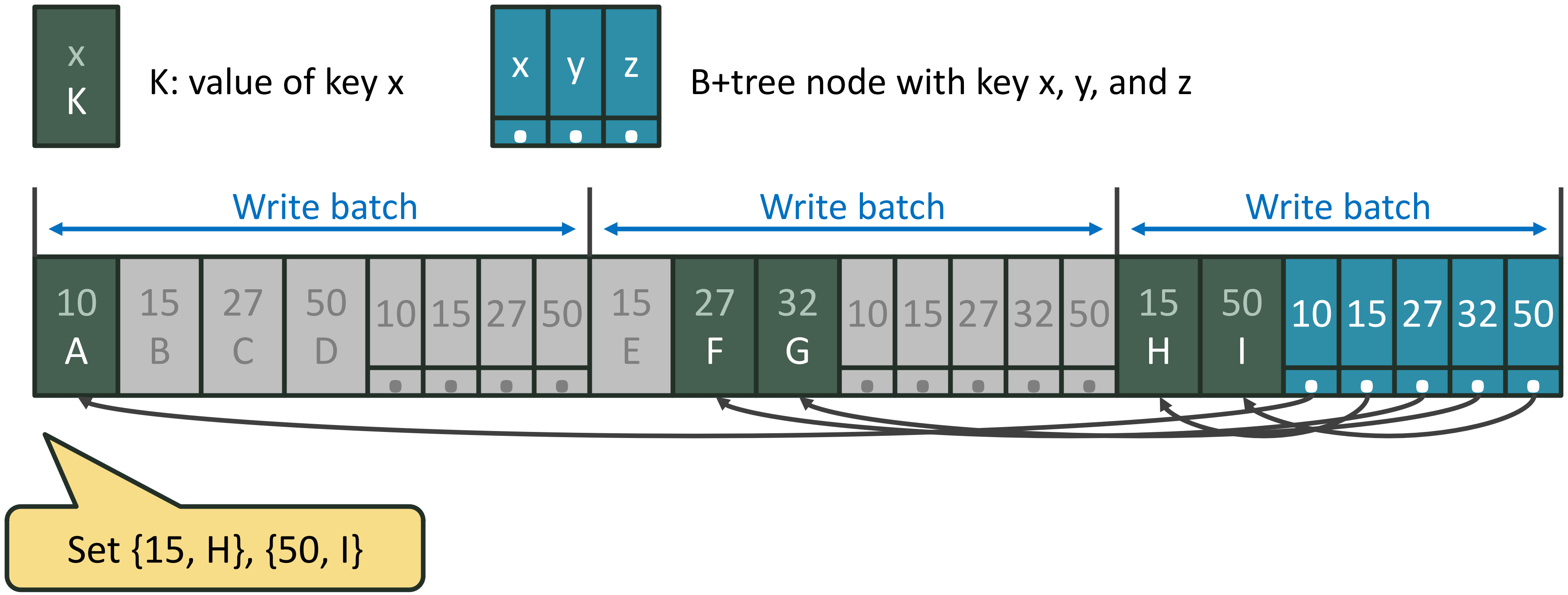
Copy-On-Write (CoW) B+Tree

- Batching + decoupling value from B+tree node
  - To reduce write amplification + to make B+tree compact
  - Append values first, and then append updated B+tree nodes
  - B+tree leaf nodes: contain {key, pointer to value} pairs



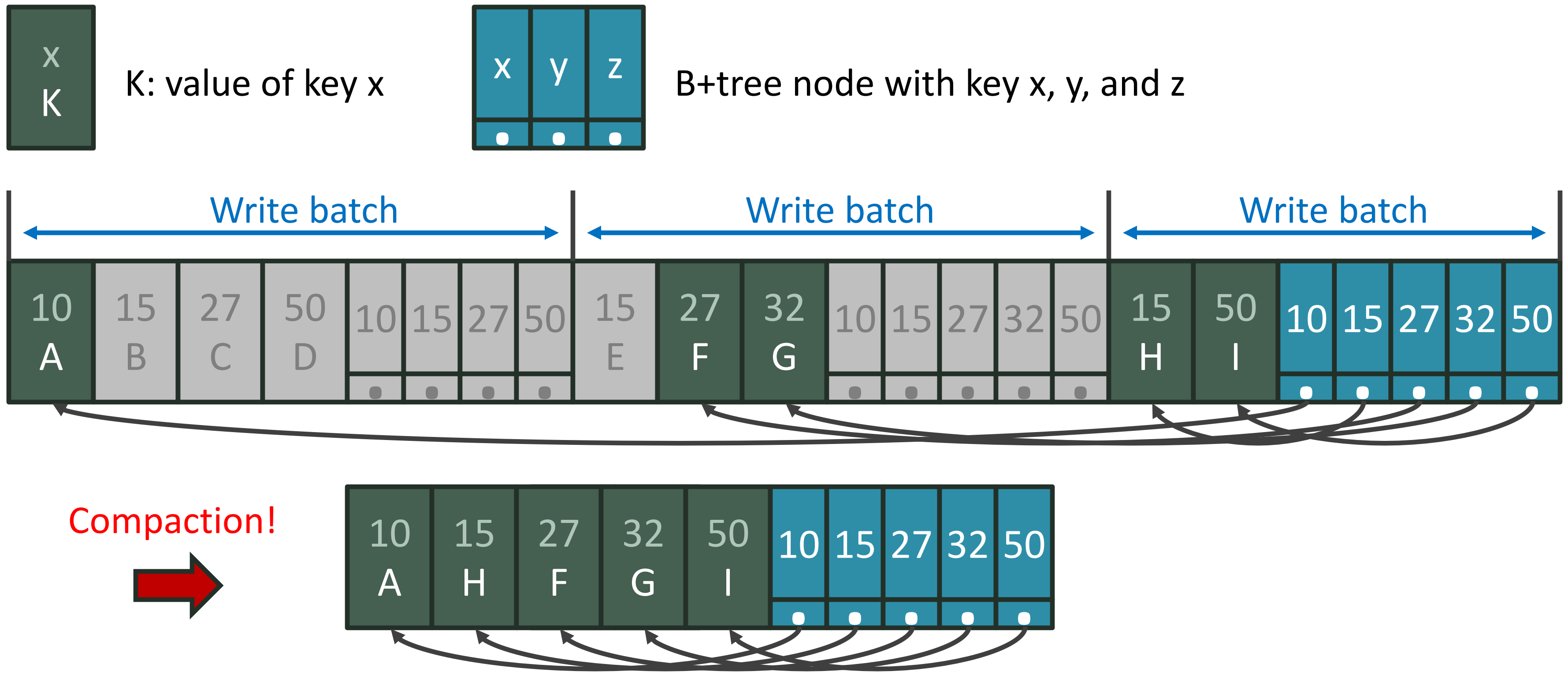
Copy-On-Write (CoW) B+Tree

- Batching + decoupling value from B+tree node
  - To reduce write amplification + to make B+tree compact
  - Append values first, and then append updated B+tree nodes
  - B+tree leaf nodes: contain {key, pointer to value} pairs



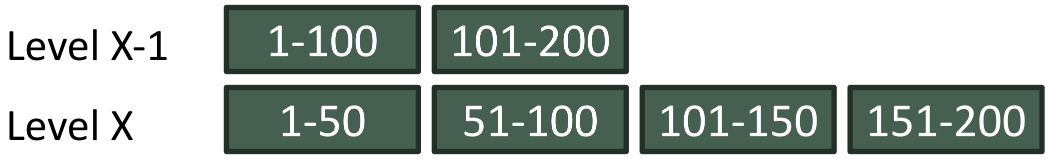
Copy-On-Write (CoW) B+Tree

- Batching + decoupling value from B+tree node
  - To reduce write amplification + to make B+tree compact
  - Append values first, and then append updated B+tree nodes
  - B+tree leaf nodes: contain {key, pointer to value} pairs



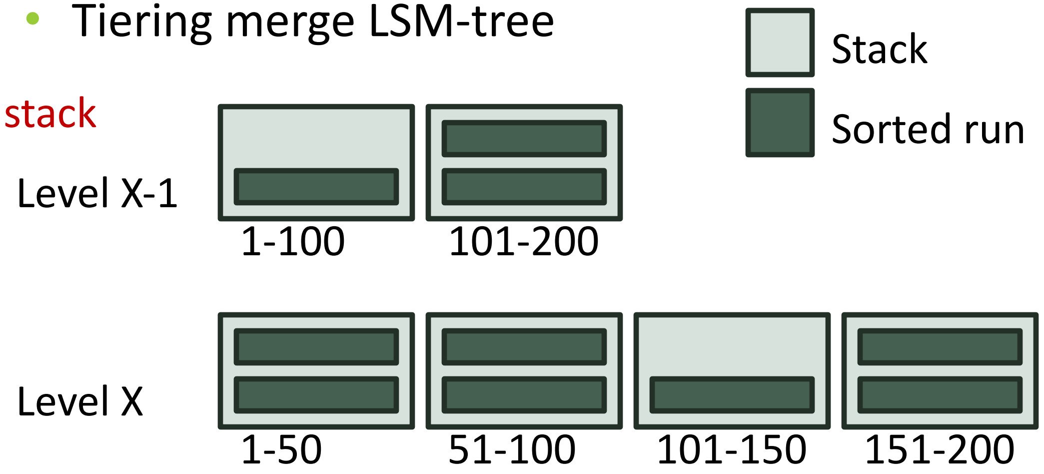
Jungle: Replacing Stack with CoW B+Tree

- Original LSM-tree (leveling)



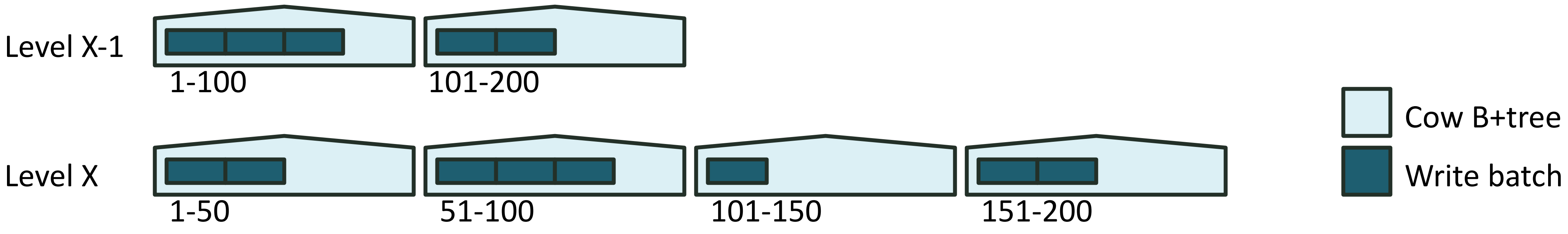
Sorted run → stack

- Tiering merge LSM-tree



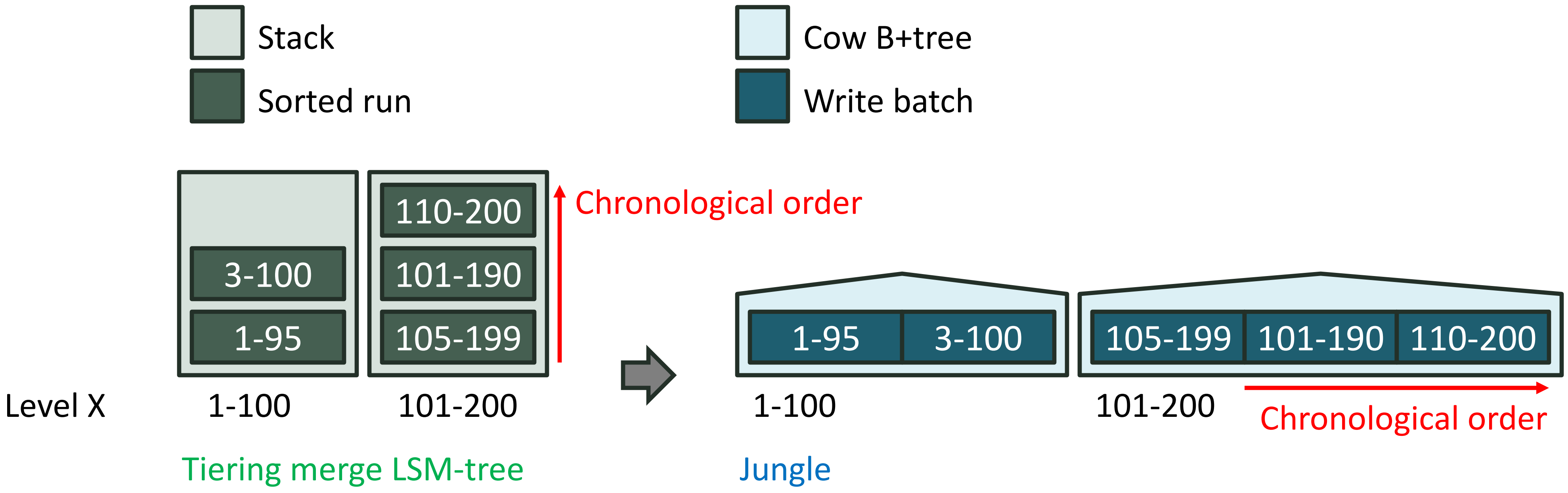
Stack → cow B+tree

- **Our approach: Jungle**



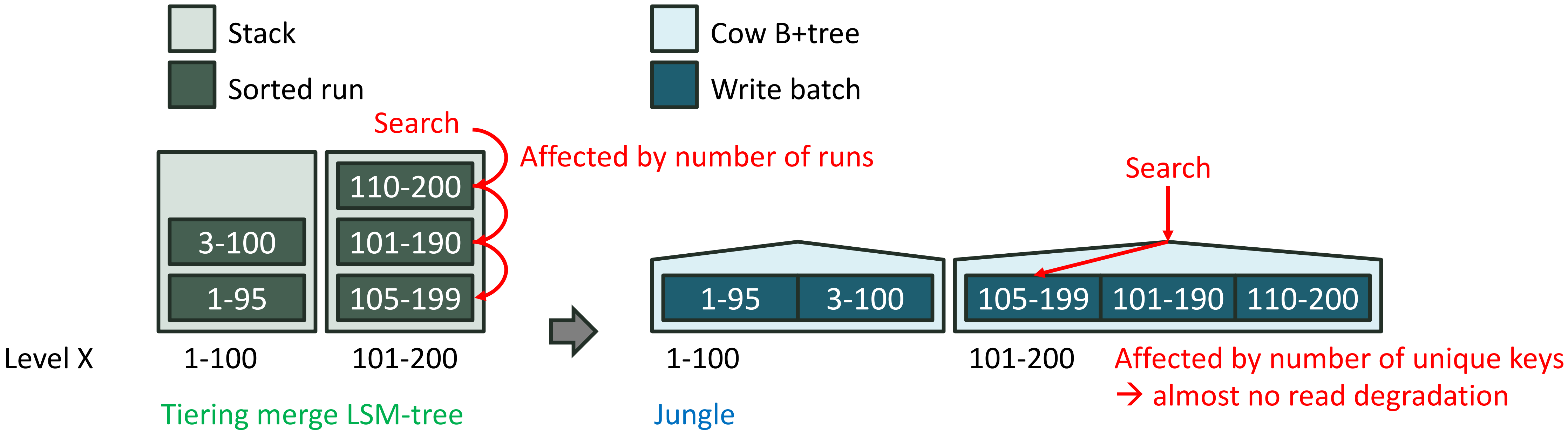
Jungle: Replacing Stack with CoW B+Tree

- Similarities between tiering stack vs. cow B+tree
  - Stack → cow B+tree
  - Sorted run → write batch (if locally sorted in key order)
    - Immutable: no overwrite
    - Appended in chronological order



Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)

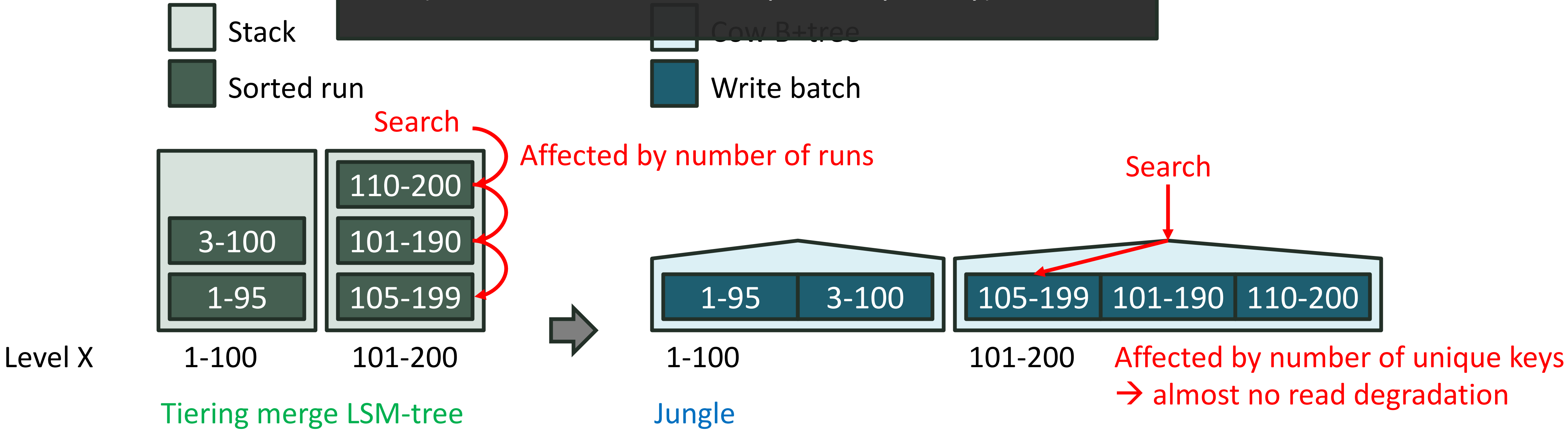


Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)

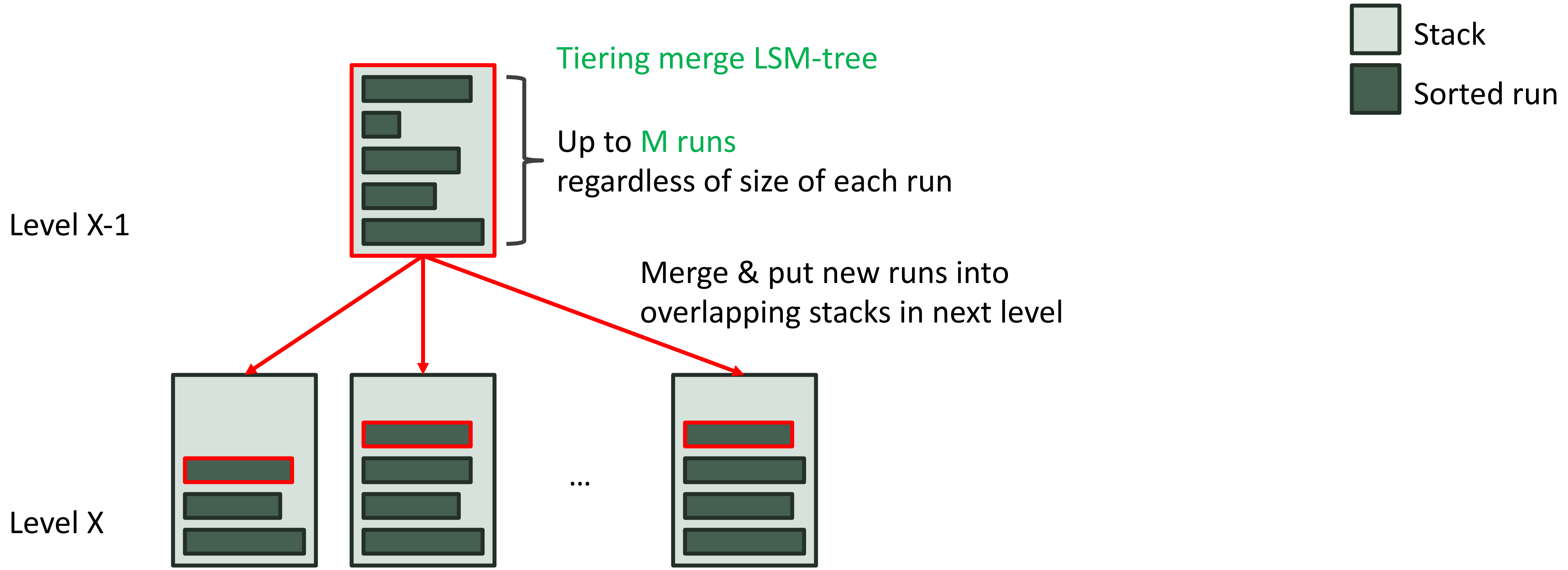
With Bloom filter?

- No stacking → M times smaller false positive rate (or M times smaller false positive penalty)



Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)



Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)

Jungle

Compaction factor: C

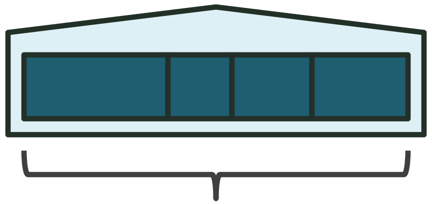
(1) Trigger in-place merge if
- space usage / live (unique) data size > C

(2) Trigger inter-level merge if
- live (unique) data size > limit (C * original sorted run size limit)

Cow B+tree

Write batch

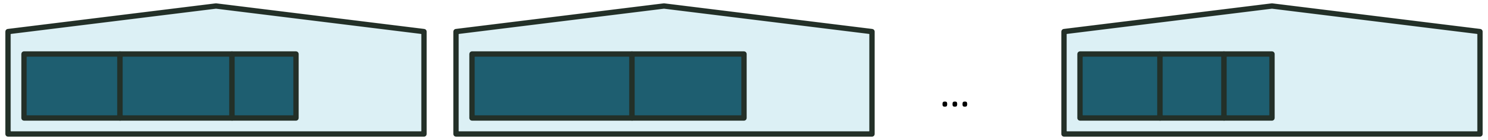
Level X-1



Based on actual size,
the number of batches is flexible

We can keep track of current live data size in B+tree meta section

Level X



Jungle

Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)

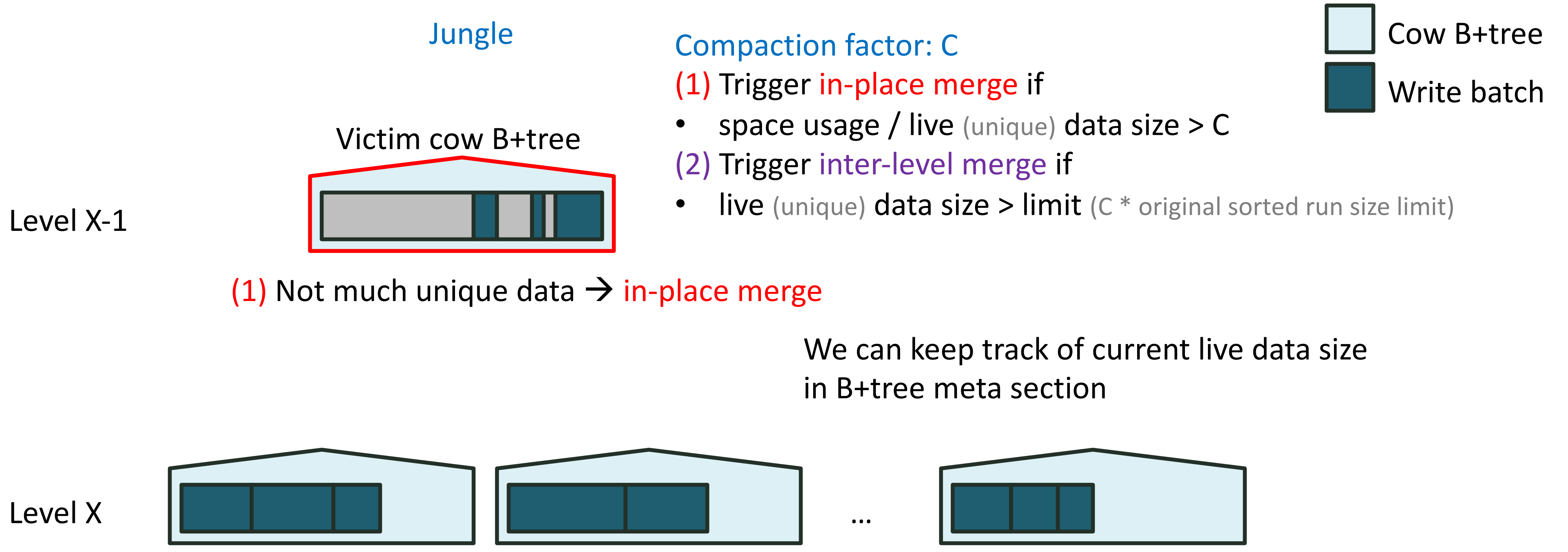
Compaction factor: C

(1) Trigger in-place merge if

- space usage / live (unique) data size > C

(2) Trigger inter-level merge if

- live (unique) data size > limit (C * original sorted run size limit)



Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)

Jungle

Cow B+tree

Write batch

Victim cow B+tree

Level X-1

Same as cow B+tree compaction

Compaction factor: C

(1) Trigger in-place merge if

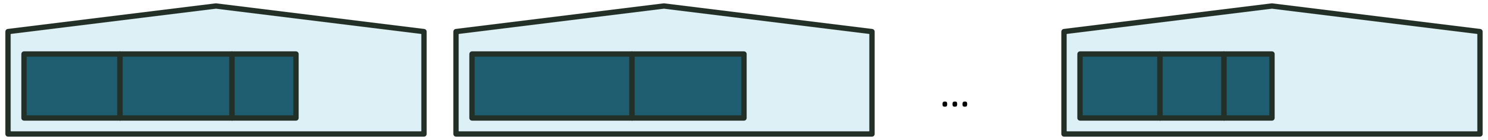
- space usage / live (unique) data size > C

(2) Trigger inter-level merge if

- live (unique) data size > limit (C * original sorted run size limit)

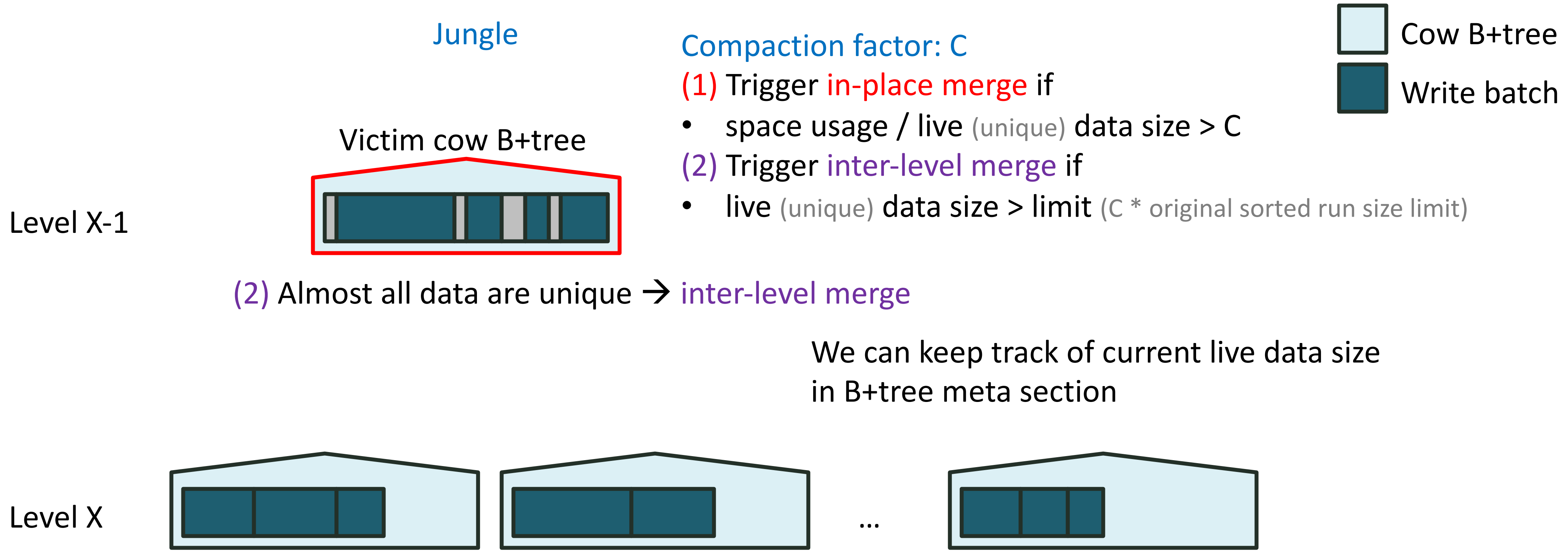
We can keep track of current live data size in B+tree meta section

Level X



Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)



Compaction factor: C

(1) Trigger in-place merge if

- space usage / live (unique) data size > C

(2) Trigger inter-level merge if

- live (unique) data size > limit (C * original sorted run size limit)

(2) Almost all data are unique → inter-level merge

We can keep track of current live data size in B+tree meta section

Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)

Jungle

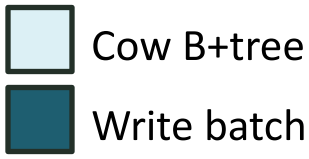
Compaction factor: C

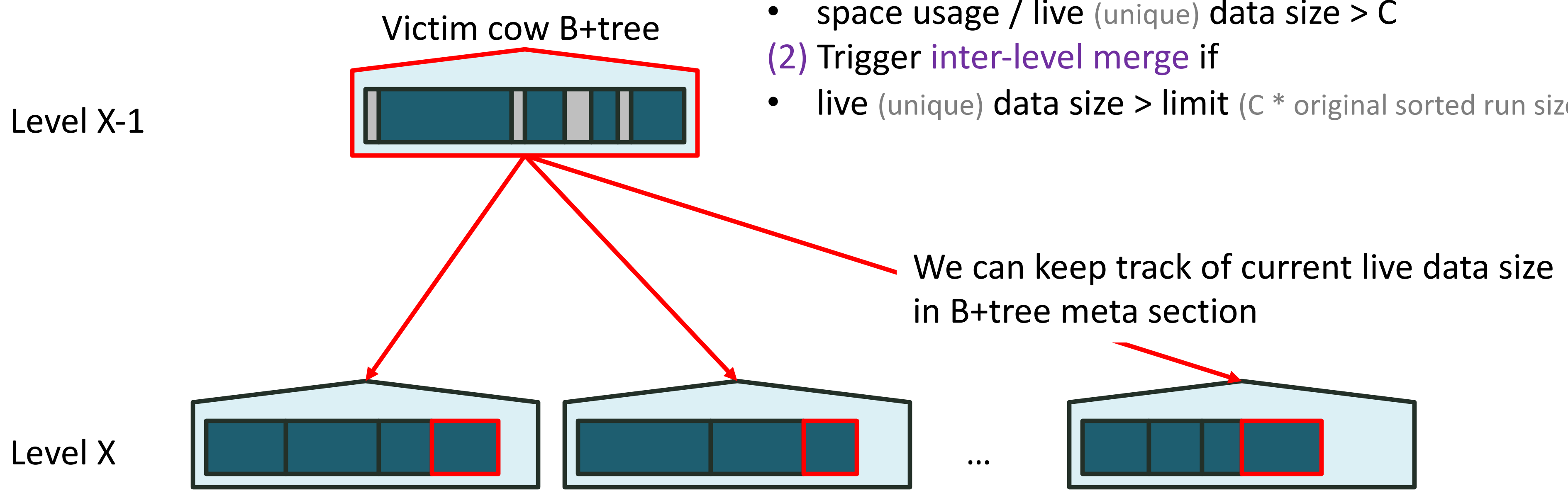
(1) Trigger in-place merge if

- space usage / live (unique) data size > C

(2) Trigger inter-level merge if

- live (unique) data size > limit (C * original sorted run size limit)





Same as tiering merge:
merge & put new batches into overlapping cow B+trees in next level

Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)

Jungle

Compaction factor: C

(1) Trigger in-place merge if
- space usage / live (unique) data size > C

(2) Trigger inter-level merge if
- live (unique) data size > limit (C * original sorted run size limit)

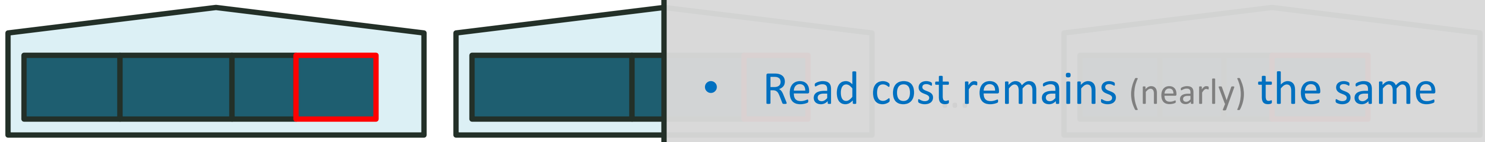
Cow B+tree

Write batch

Level X-1

- Smaller C
  - Frequent merge: more writes, less space
- Bigger C
  - Infrequent merge: less writes, more space
- Read cost remains (nearly) the same

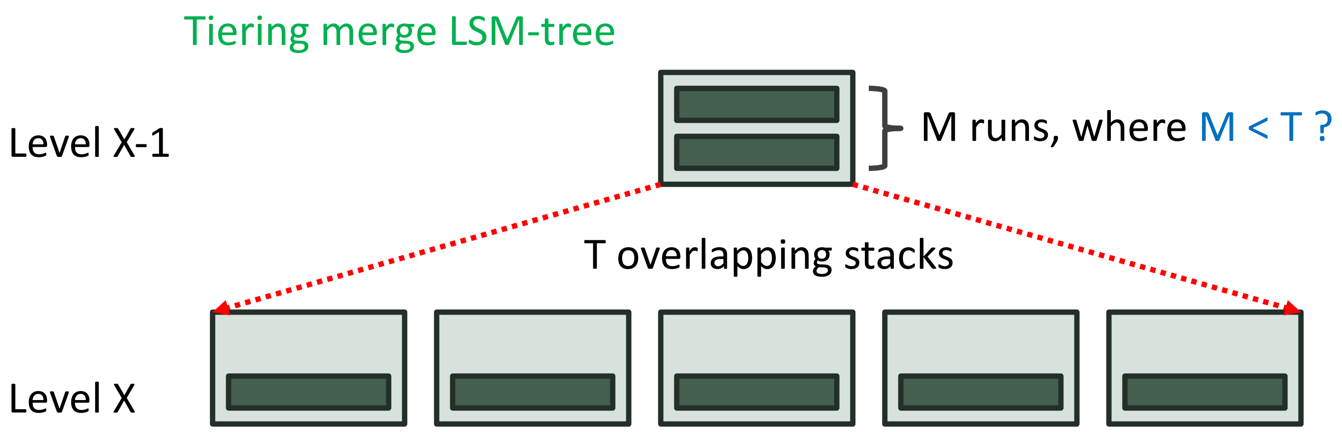
Level X



Same as tiering merge:
merge & put new batches into overlapping cow B+trees in next level

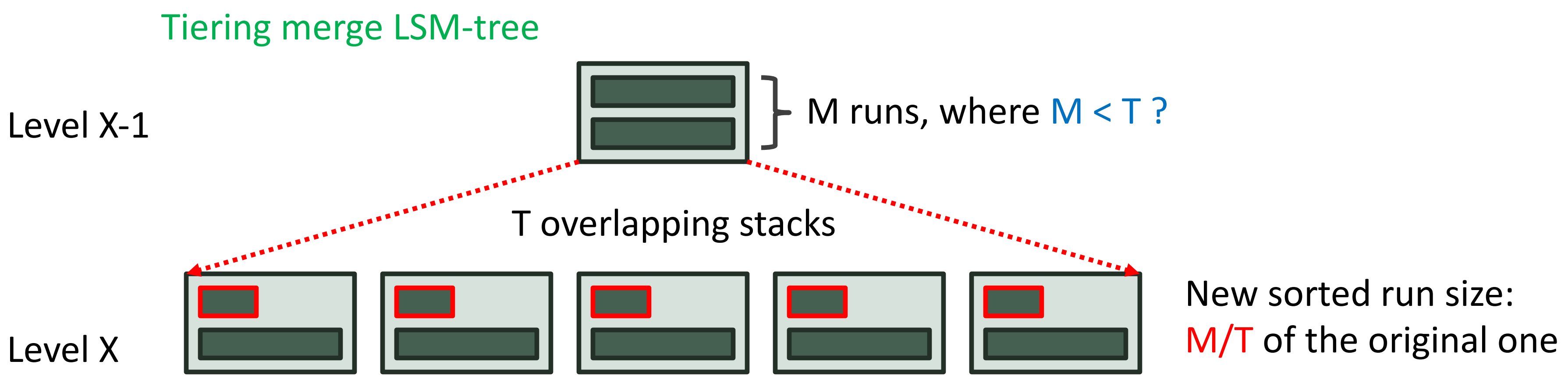
Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)



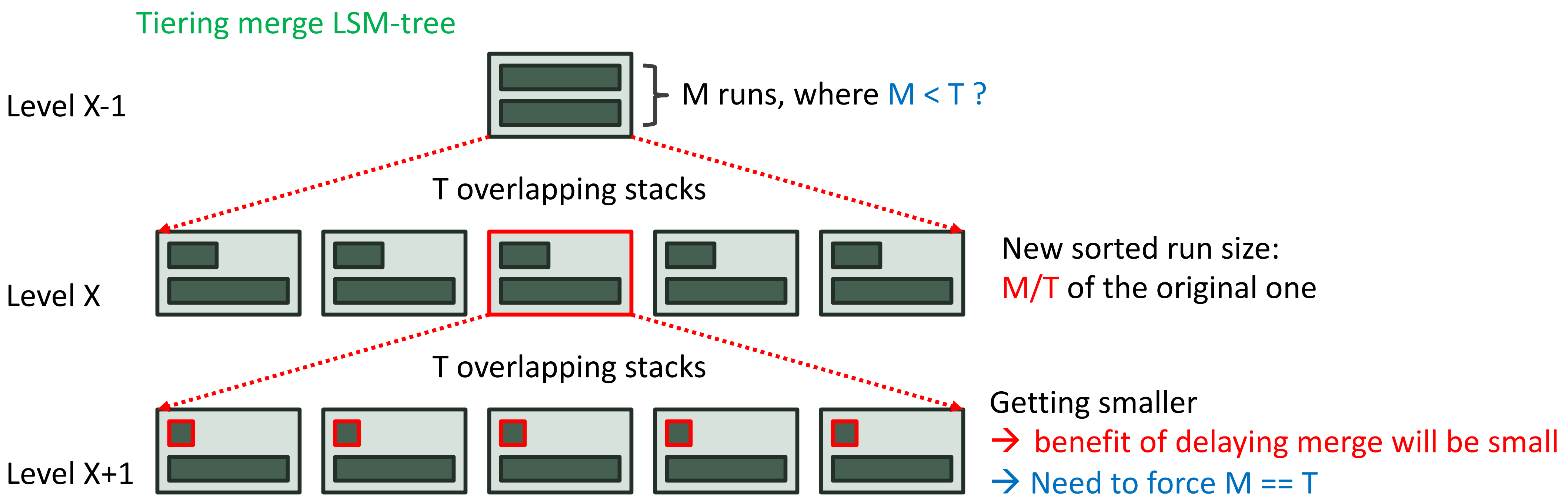
Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)



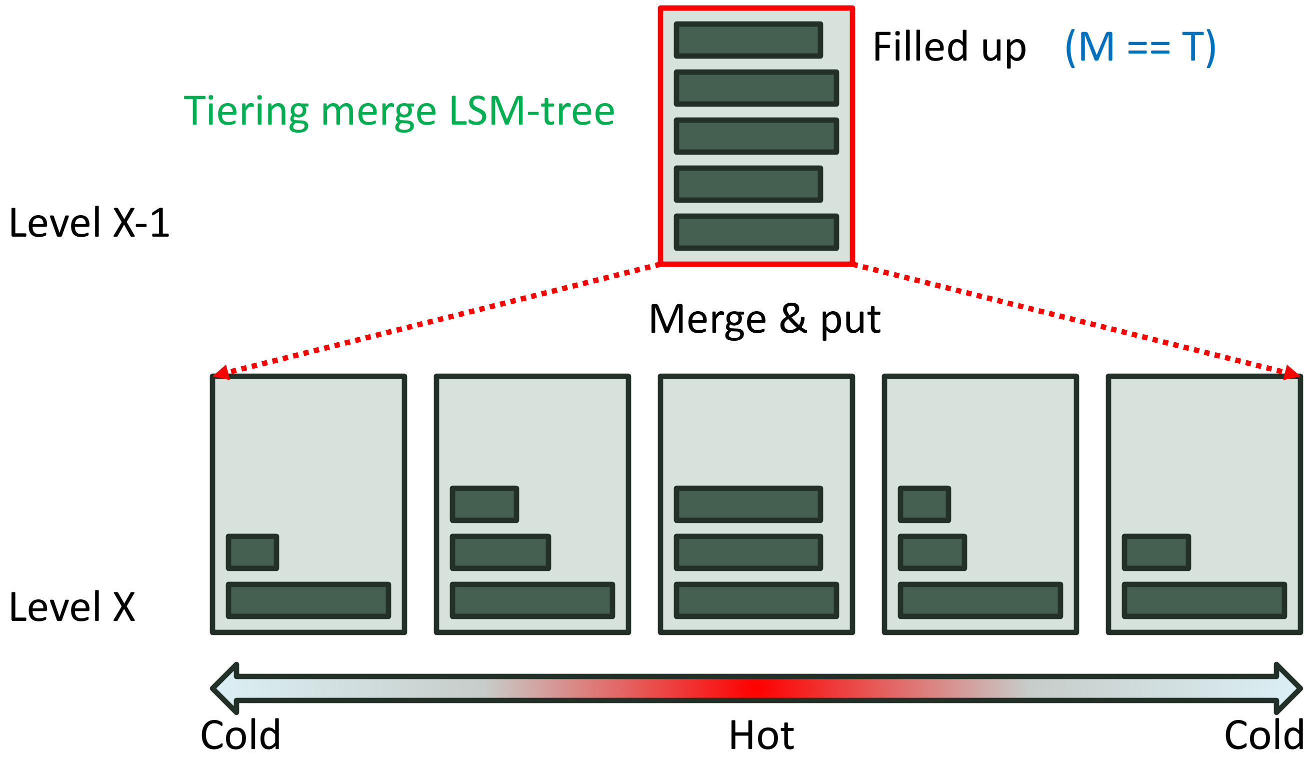
Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)



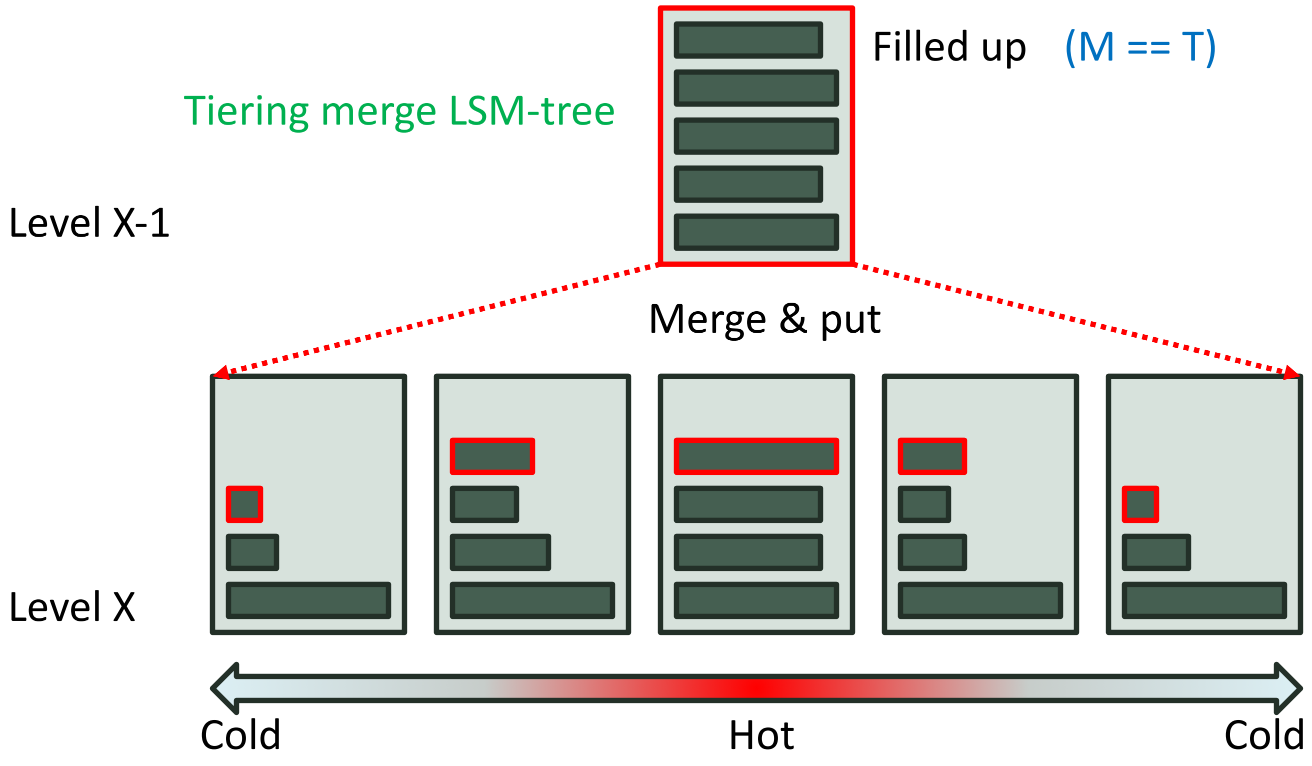
Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)
    - Locality?



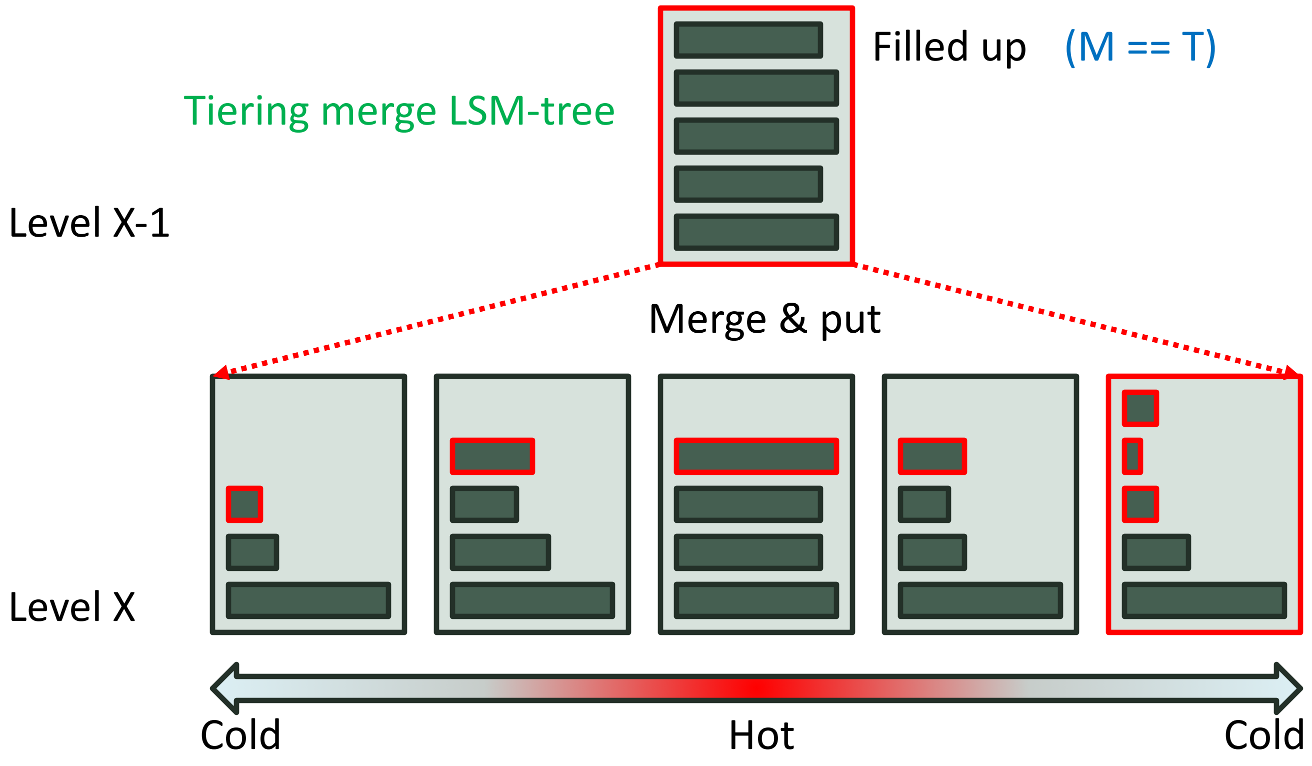
Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)
    - Locality?



Jungle: Replacing Stack with CoW B+Tree

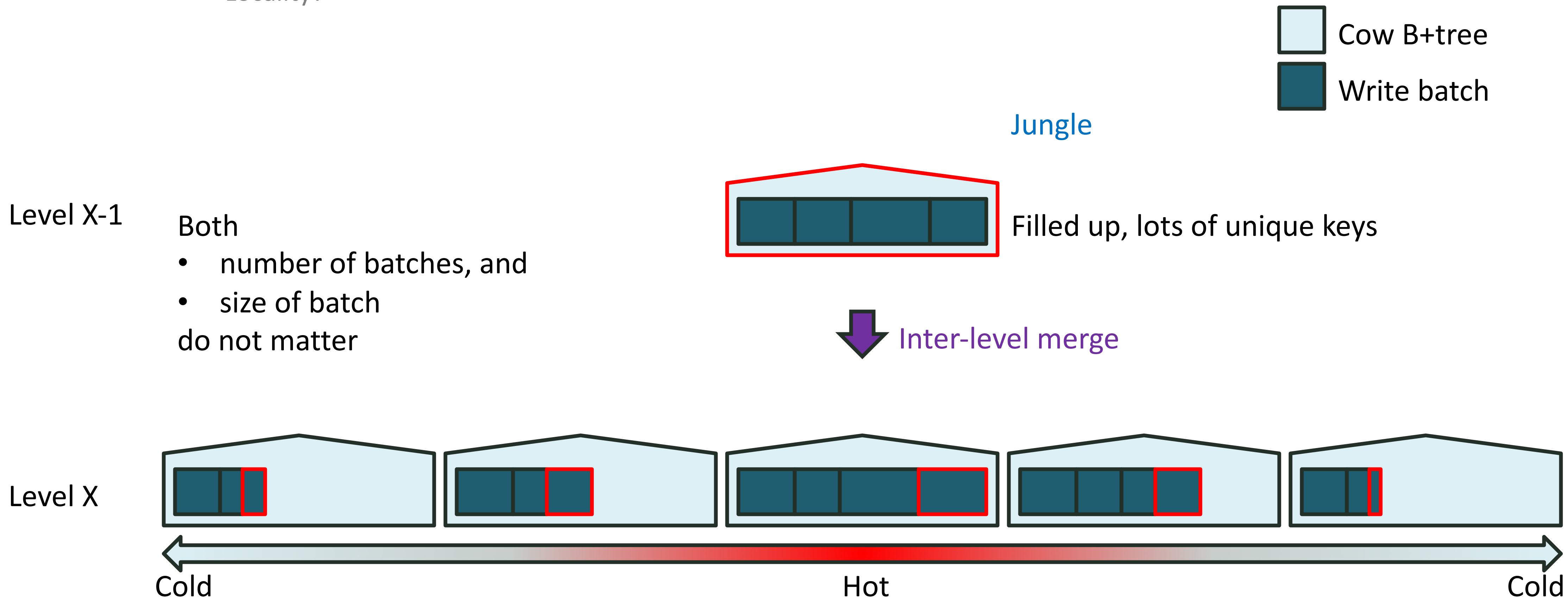
- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)
    - Locality?



Cold range:
will be filled up with small runs
→ benefit of delaying merge is small
→ cascaded: will make small runs in next level!

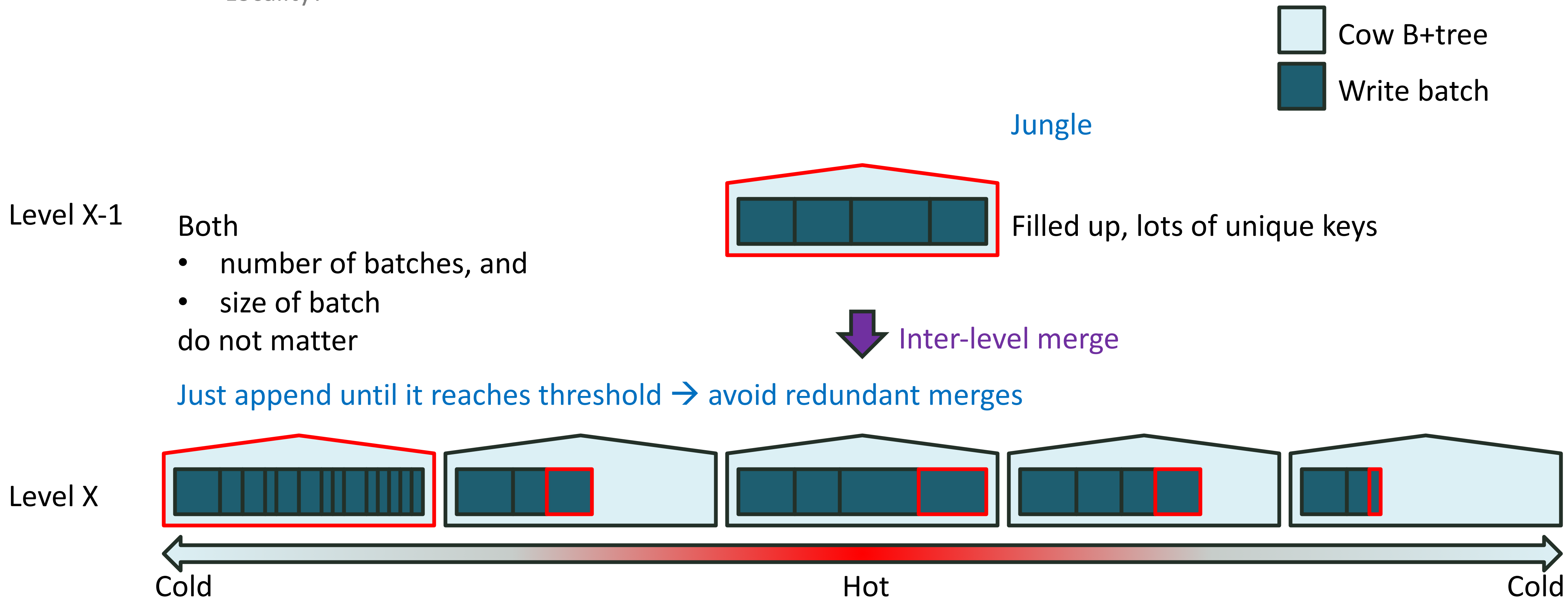
Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)
    - Locality?



Jungle: Replacing Stack with CoW B+Tree

- Differences between tiering stack vs. cow B+tree
  - Search: linear (stack) vs. logarithmic (cow B+tree)
  - Unit of limit: the number of runs (stack) vs. actual data size (cow B+tree)
    - Locality?

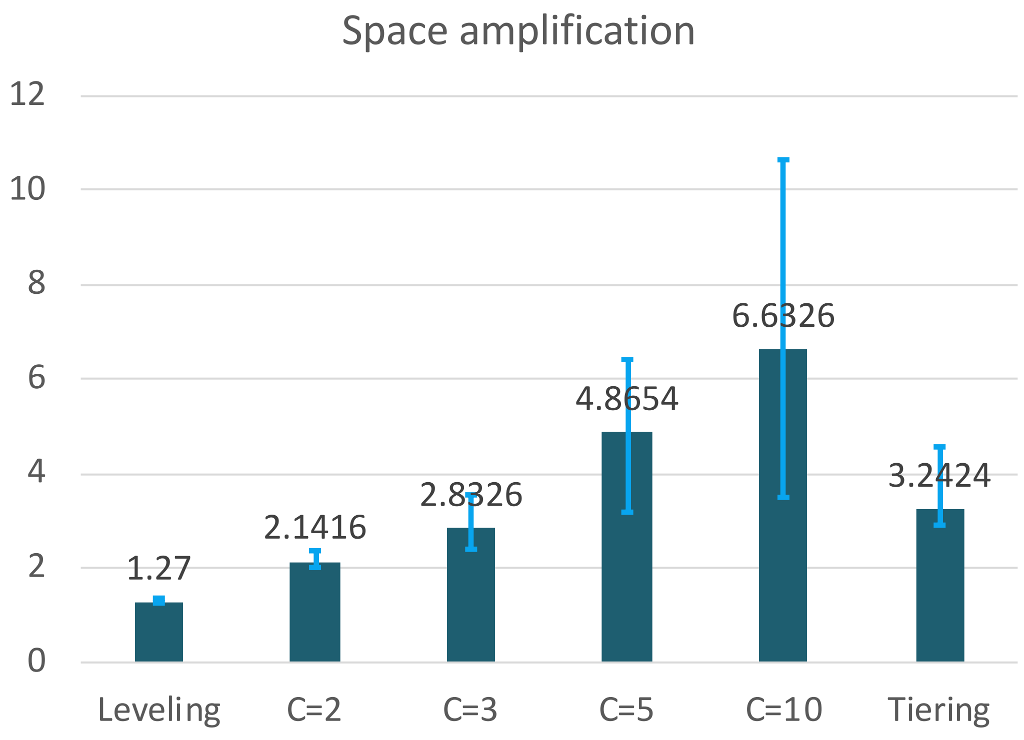


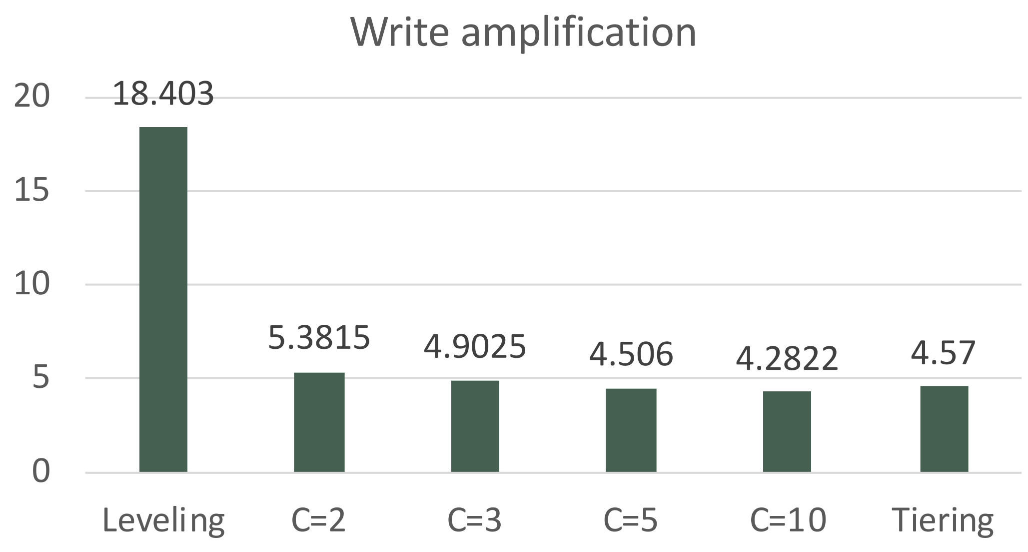
Brief Evaluation

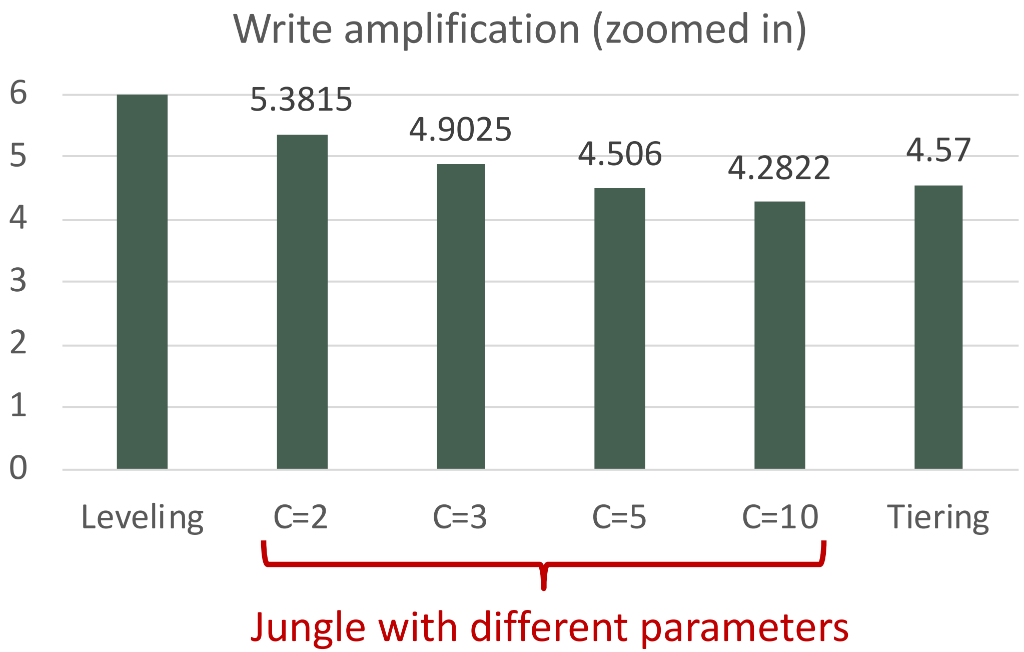
- To prove the concept of Jungle
  - Comparison against leveling (original LSM-tree) and tiering
  - Widely used LSM-based approaches: leveling
- Environment
  - Samsung 860 QVO 1TB, Ext4
  - 20M random key-value pairs
    - Key: 8 bytes, value 1024 KB
  - RAM size: limited to 2.5 GB
  - LSM-tree settings
    - Max sorted run size: 64MB, L0 size limit: 256MB
    - Size ratio between levels (T): 10
    - Max stack size of tiering (M): 10 runs (==T)
    - Bloom filter: 10 bits per key (~1% false positive rate)

Brief Evaluation

- Write and space amplification
  - Issued 400M uniform random updates (no locality → best case for tiering)

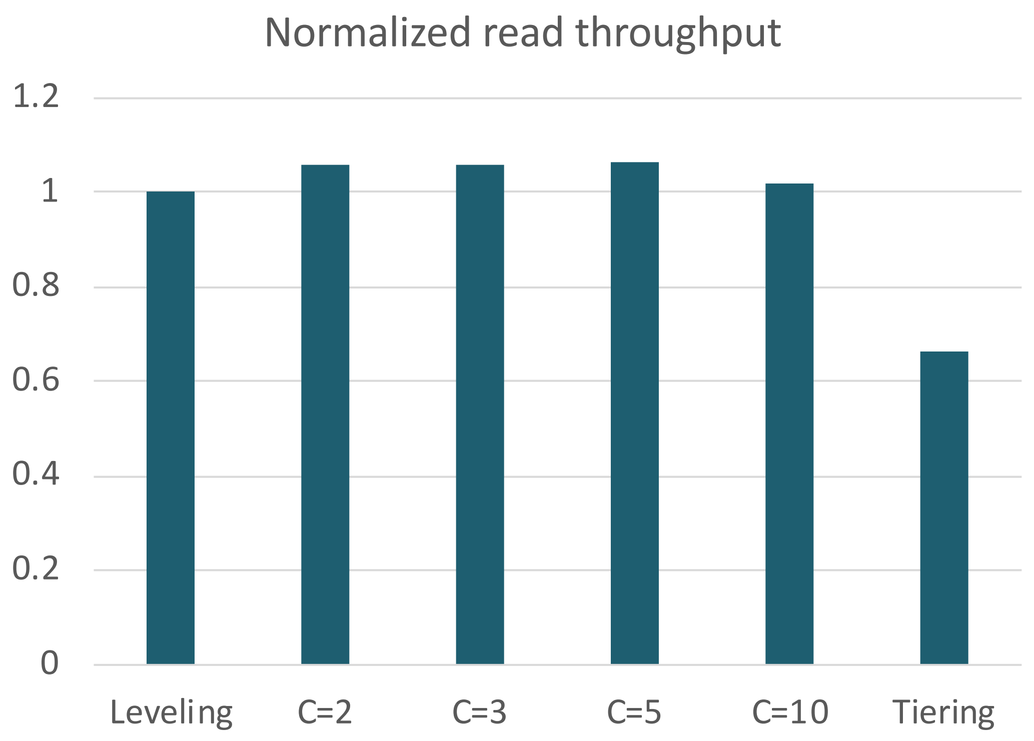


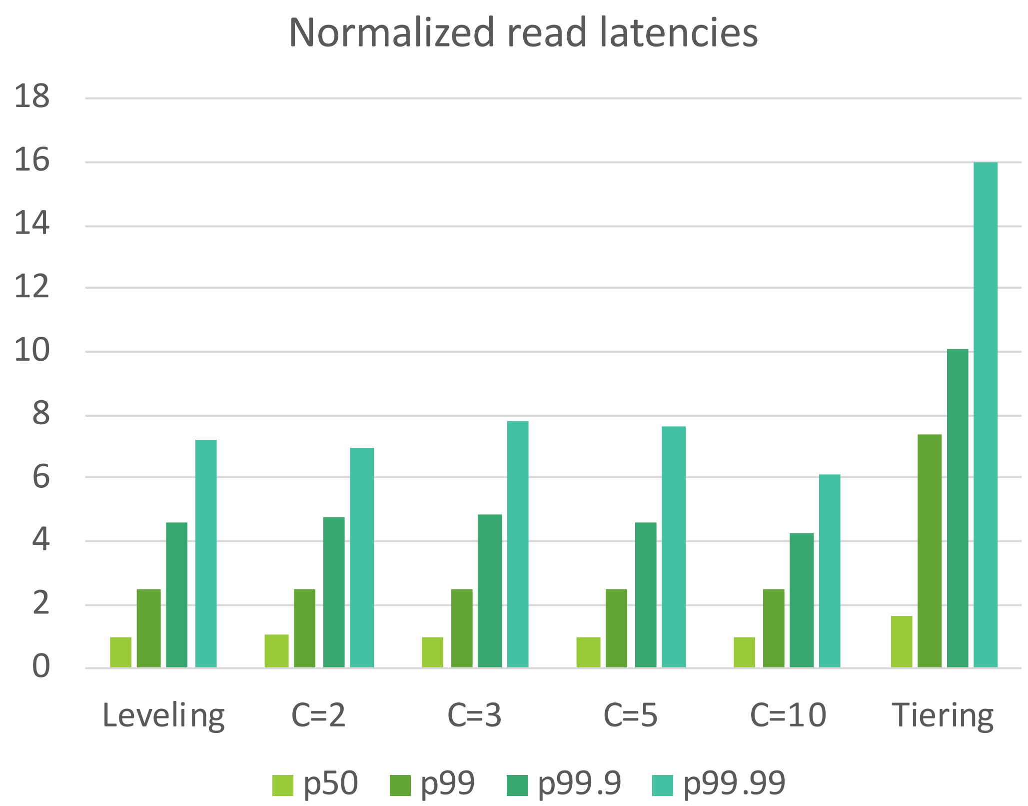




Brief Evaluation

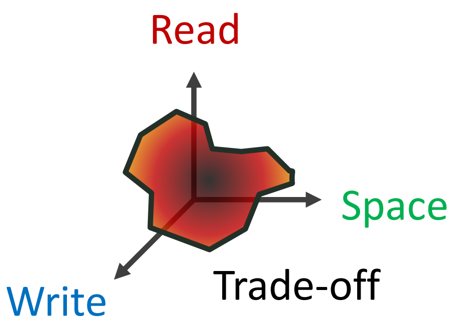
- Point read throughput and latencies
  - Issued random read operation on aged indexes



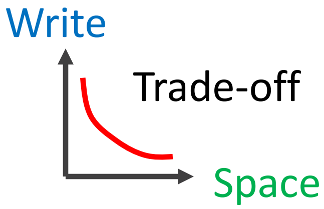


Summary

- Traditional LSM-tree trade-offs
  - Read, write, and space



- Jungle
  - Transplant copy-on-write B+tree into LSM-tree
  - Get rid of read cost from the trade-off
  - Introduce a practical parameter to adjust write and space



- What's next?
  - Fundamental ways to re-think LSM-tree structure
  - More chances to deform or optimize

Thank You