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

Logs: in chronological order

Level-0

<table>
<thead>
<tr>
<th>MemTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>15, 3, 27, 11</td>
</tr>
</tbody>
</table>

Level-1

Log

<table>
<thead>
<tr>
<th>Sorter runs: immutable once created</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 5, 10, 12</td>
</tr>
</tbody>
</table>

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

Duplicate keys may exist across different levels

Sorted runs:

- 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

Each run has disjoint key range:
- no duplicate key in the same level

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

- A single sorted run size is limited

- Each level size is also limited (i.e., # of sorted runs)

- Level size limit increases exponentially

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

New updates: 29, 8, 21, 25

MemTable

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

<table>
<thead>
<tr>
<th>Log</th>
<th>MemTable</th>
<th>Just append</th>
</tr>
</thead>
<tbody>
<tr>
<td>15, 3, 27, 11, 29, 8, 21, 25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level-0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 5, 10, 12</td>
<td>17, 20, 25, 32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level-1</th>
<th>Sorted run (a.k.a., Sorted String Table)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 4, 5, 6</td>
<td>7, 9, 11, 13</td>
</tr>
</tbody>
</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

Log

<table>
<thead>
<tr>
<th>MemTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>15, 3, 27, 11, 29, 8, 21, 25</td>
</tr>
</tbody>
</table>

Log Level-0

<table>
<thead>
<tr>
<th>Run</th>
<th>Sorted run (a.k.a., Sorted String Table)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 5, 10, 12</td>
<td>2, 4, 5, 6</td>
</tr>
<tr>
<td>17, 20, 25, 32</td>
<td>7, 9, 11, 13</td>
</tr>
</tbody>
</table>

Merge, sort, and write new runs

<table>
<thead>
<tr>
<th>Run</th>
<th>Sorted run (a.k.a., Sorted String Table)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 3, 5, 8</td>
<td>15, 16, 19, 20</td>
</tr>
<tr>
<td>10, 11, 12, 15</td>
<td>22, 27, 30, 31</td>
</tr>
<tr>
<td>17, 20, 21, 25</td>
<td>27, 29, 32</td>
</tr>
</tbody>
</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

<table>
<thead>
<tr>
<th>Level-0</th>
<th>Level-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 3, 5, 8</td>
<td>2, 4, 5, 6</td>
</tr>
<tr>
<td>10, 11, 12, 15</td>
<td>7, 9, 11, 13</td>
</tr>
<tr>
<td>17, 20, 21, 25</td>
<td>15, 16, 19, 20</td>
</tr>
<tr>
<td>27, 29, 32</td>
<td>22, 27, 30, 31</td>
</tr>
</tbody>
</table>

### Remove old runs & logs

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

<table>
<thead>
<tr>
<th>Log</th>
<th>Merge can be cascaded: now level-0 becomes full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level-0</td>
<td>1, 3, 5, 8</td>
</tr>
<tr>
<td>Level-1</td>
<td>2, 4, 5, 6</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Level-0</th>
<th>Level-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 3, 5, 8</td>
<td>2, 4, 5, 6</td>
</tr>
<tr>
<td>10, 11, 12, 15</td>
<td>7, 9, 11, 13</td>
</tr>
<tr>
<td>17, 20, 21, 25</td>
<td>15, 16, 19, 20</td>
</tr>
<tr>
<td>27, 29, 32</td>
<td>22, 27, 30, 31</td>
</tr>
</tbody>
</table>

Select a victim run

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

Find overlapping runs in next level

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

<table>
<thead>
<tr>
<th>Level-0</th>
<th>Level-1</th>
<th>Sorted run (a.k.a., Sorted String Table)</th>
<th>Merge, sort, and write new runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 3, 5, 8</td>
<td>2, 4, 5, 6</td>
<td>15, 16, 17, 19</td>
<td>27, 30, 31</td>
</tr>
<tr>
<td>10, 11, 12, 15</td>
<td>7, 9, 11, 13</td>
<td>15, 16, 19, 20</td>
<td>20, 21, 22, 25</td>
</tr>
<tr>
<td>17, 20, 21, 25</td>
<td></td>
<td>15, 16, 17, 19</td>
<td></td>
</tr>
<tr>
<td>27, 29, 32</td>
<td></td>
<td>20, 21, 22, 25</td>
<td></td>
</tr>
</tbody>
</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

<table>
<thead>
<tr>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level-0</td>
</tr>
<tr>
<td>1, 3, 5, 8</td>
</tr>
<tr>
<td>10, 11, 12, 15</td>
</tr>
<tr>
<td>27, 29, 32</td>
</tr>
<tr>
<td>Level-1</td>
</tr>
<tr>
<td>2, 4, 5, 6</td>
</tr>
<tr>
<td>7, 9, 11, 13</td>
</tr>
<tr>
<td>15, 16, 17, 19</td>
</tr>
<tr>
<td>20, 21, 22, 25</td>
</tr>
<tr>
<td>27, 30, 31</td>
</tr>
</tbody>
</table>

Remove old runs

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

Level-0

1, 3, 5, 8

10, 11, 12, 15

27, 29, 32

• Sorted run is immutable once written \(\rightarrow\) no overwrite

• Each merge operation: re-write all overlapping runs

Remove old runs

Level-1

2, 4, 5, 6

7, 9, 11, 13

15, 16, 17, 19

20, 21, 22, 25

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 – read

  ![Diagram of Log-Structured Merge-Tree](image)

  - **Log**: 15, 3, 27, 11
  - **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)

- Inner-level search: logarithmic
- Inter-level search: linear
Log-Structured Merge-Tree (LSM-Tree)  
(P. O’Neil et al. 1996)

- Basic algorithm – read

Log

| 15, 3, 27, 11 |

1) Search MemTable (in-memory operation)

→ not found

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)

- Inner-level search: logarithmic
- Inter-level search: linear
Log-Structured Merge-Tree (LSM-Tree) (P. O’Neil et al. 1996)

- Basic algorithm – read

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

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

Inner-level search: logarithmic
Inter-level search: linear
Log-Structured Merge-Tree (LSM-Tree) (P. O’Neil et al. 1996)

- Basic algorithm – read

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

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

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

Log - 15, 3, 27, 11
MemTable - Search key 6

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

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

2) Check bloom filter (in-memory operation)  
   ⇒ negative: skip

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)
  BF  BF  BF  BF

3) Check bloom filter (in-memory operation)  
   ⇒ positive: search level-1 (disk accesses)  
   ⇒ found!

- 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

Log

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

Level-0

1, 5, 10, 12

17, 20, 25, 32

2) Check bloom filter (in-memory operation)
   ➔ negative: skip

Level-1

2, 4, 5, 6

7, 9, 11, 13

15, 16, 19, 20

22, 27, 30, 31

3) Check bloom filter (in-memory operation)
   ➔ positive: search level-1 (disk access)
   ➔ found!

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

False positive rate: roughly 1%
- 3 hash functions
- 10 bits per key

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

MemTable

Search key 6

BF

BF

BF

BF

BF

BF
Write Amplification

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

30KB User requested

Index processing

90KB Actual data written to disk

Write amplification: 3
Write Amplification

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

- LSM-tree: merge amplifies amount of writes

<table>
<thead>
<tr>
<th>Level X-1</th>
<th>1-10</th>
<th>11-20</th>
<th>Wider range, less runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level X</td>
<td>1-5</td>
<td>6-10</td>
<td>11-15</td>
</tr>
</tbody>
</table>

Density difference
Write Amplification

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

- LSM-tree: merge amplifies amount of writes

<table>
<thead>
<tr>
<th>Level X-1</th>
<th>Level X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>1-5</td>
</tr>
<tr>
<td>11-20</td>
<td>6-10</td>
</tr>
<tr>
<td><strong>2 overlapping runs</strong></td>
<td><strong>11-15</strong></td>
</tr>
</tbody>
</table>

Victim run to merge

Wider range, less runs

Narrower range, more runs

Density difference
Write Amplification

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

- LSM-tree: merge amplifies amount of writes

Victim run to merge

- Level X-1: 1-10, 11-20
- Level X: 1-5, 6-10, 11-15, 16-20

2 overlapping runs

- Wider range, less runs
  - Density difference

- Narrower range, more runs

Write 2 new runs after merge

Write amplification: 2
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

<table>
<thead>
<tr>
<th>Level 0</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image.png" alt="Diagram" /></td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

- Level 0
- Level 1
- Level 2

- 2 runs
- 2 \* T runs
- 2 \* T^2 runs
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

A single merge: $T$ overlapping runs

<table>
<thead>
<tr>
<th>Level 0</th>
<th>2 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>2 * $T$ runs</td>
</tr>
<tr>
<td>Level 2</td>
<td>2 * $T^2$ runs</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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

- A single merge: T overlapping runs
  - Level 0: 2 runs
  - Level 1: 2 * T runs
  - Level 2: 2 * T² runs

- Merge can be cascaded: T overlapping runs for each level
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

<table>
<thead>
<tr>
<th>Level 0</th>
<th>2 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>2 * T runs</td>
</tr>
<tr>
<td>Level 2</td>
<td>2 * T^2 runs</td>
</tr>
</tbody>
</table>

- A single merge: T overlapping runs
- Merge can be cascaded: T overlapping runs for each level

Write amplification: T + T + ... + T = O(L * T)

The number of levels (L)
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)

---

**Log**

- **MemTable**
- Log size exceeds the limit

**Level-0**

- 35, 3, 22, 33, 11, 23, 17, 1
- 2, 7, 11, 15
- 21, 28, 30, 32

**Level-1**

- ...

---
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 key-range (stack), without re-writing existing runs

Stack for the same key range

⇒ duplicate keys are allowed
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)

Log

<table>
<thead>
<tr>
<th>Level-0</th>
<th>Stack for the same key range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 3, 11, 17</td>
<td>-&gt; duplicate keys are allowed</td>
</tr>
<tr>
<td>2, 7, 11, 15</td>
<td>21, 28, 30, 32</td>
</tr>
</tbody>
</table>

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)

Log

Level-0

- 1, 3, 11, 17
- 2, 7, 11, 15

Level-1

- 21, 25, 27, 29
- 31, 34, 36, 37

Find overlapping stacks

Pick a victim stack
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)

Log

<table>
<thead>
<tr>
<th>Level-0</th>
<th>Level-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 3, 11, 17</td>
<td>22, 23, 33, 35</td>
</tr>
<tr>
<td>2, 7, 11, 15</td>
<td>21, 28, 30, 32</td>
</tr>
<tr>
<td>21, 25, 27, 29</td>
<td>31, 34, 36, 37</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 $\rightarrow$ 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

![Diagram of B+tree nodes]

Logical view

![Flattened view of B+tree nodes (on disk)]

Flattened view (on disk)
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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

Flattened view (on disk)

| D | E | F | B | C | A | E' |

Append E' → parent C 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

Logical view

Flattened view (on disk)

Append $C'$
$\rightarrow$ 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

Logical view

Flattened view (on disk)
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

![Diagram showing a B+tree node with keys 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

\[
\begin{align*}
  \text{K: value of key } x & \quad \text{B+tree node with key } x, y, \text{ and } z \\
  \begin{array}{c|c|c|c}
    x & y & z \\
    \\ hline
    x & y & z \\
  \end{array}
\end{align*}
\]

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

Append values (in key 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

\[
x \quad K: \text{value of key } x
\]

\[
x \quad y \quad z
\]

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

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

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

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

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

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>15</td>
<td>27</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>
```

- Write batch

```
Set {15, E}, {27, F}, {32, G}
```
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

|x| K: value of key x

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

Set \{15, E\}, \{27, F\}, \{32, G\}

Append values (in key order)

Write batch
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

\[
\begin{align*}
\text{K: value of key } x \\
\text{B+tree node with key } x, y, \text{ and } z
\end{align*}
\]

\[
\begin{array}{cccccccc}
10 & 15 & 27 & 50 & 10 & 15 & 27 & 50 \\
A & B & C & D & E & F & G & \\
\end{array}
\]

- Set \{15, E\}, \{27, F\}, \{32, G\}
- Keys in the same node should be copied together although they are not updated
- Append updated B+tree node

\[
\begin{array}{cccccccc}
10 & 15 & 27 & 32 & 50 \\
\end{array}
\]
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

Set \{15, H\}, \{50, I\}
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

![Diagram showing B+Tree operations and compaction](image-url)
Jungle: Replacing Stack with CoW B+Tree

- Original LSM-tree (leveling)

<table>
<thead>
<tr>
<th>Level X-1</th>
<th>Level X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-100</td>
<td>1-50</td>
</tr>
<tr>
<td>101-200</td>
<td>51-100</td>
</tr>
<tr>
<td></td>
<td>101-150</td>
</tr>
<tr>
<td></td>
<td>151-200</td>
</tr>
</tbody>
</table>

- Tiering merge LSM-tree

<table>
<thead>
<tr>
<th>Level X-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-100</td>
</tr>
<tr>
<td>101-200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-50</td>
</tr>
<tr>
<td>51-100</td>
</tr>
<tr>
<td>101-150</td>
</tr>
<tr>
<td>151-200</td>
</tr>
</tbody>
</table>

- Our approach: Jungle

<table>
<thead>
<tr>
<th>Level X-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-100</td>
</tr>
<tr>
<td>101-200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level X</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-50</td>
</tr>
<tr>
<td>51-100</td>
</tr>
<tr>
<td>101-150</td>
</tr>
<tr>
<td>151-200</td>
</tr>
</tbody>
</table>

Stack → cow B+tree

Sorted run → stack
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

![Diagram showing the comparison between Stack and Cow B+tree]

- Level X
  - Stack
    - 1-100
    - 101-200
  - Sorted run
    - 3-100
    - 101-190
    - 105-199

- Cow B+tree
  - 1-100
  - 101-200
  - Write batch
    - 1-95
    - 3-100
    - 105-199
    - 101-190
    - 110-200

- Tiering merge LSM-tree
- Jungle
Jungle: Replacing Stack with CoW B+Tree

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

Level X

Stack
Sorted run

Tiering merge LSM-tree

1-95
3-100
101-200
110-200
105-199

1-100
101-200

Affected by number of runs

Search

Cow B+tree
Write batch

1-100
3-100

101-200
105-199
101-190
110-200

Affected by number of unique keys
Almost no read degradation
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 $\rightarrow$ M times smaller false positive rate (or M times smaller false positive penalty)

![Diagram showing differences between Stack and Cow B+Tree](image)

**Affected by number of runs**
- Stack
- Sorted run
- Level X
- Tiering merge LSM-tree

**Affected by number of unique keys**
- 1-95
- 3-100
- 105-199
- 101-200
- 1-100
- 110-200

With Bloom filter?
- No stacking $\rightarrow$ 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)

![Diagram of tiering merge LSM-tree](image)

- Stack
- Sorted run

Level X-1

Merge & put new runs into overlapping stacks in next level

Level X

Up to $M$ runs regardless of size of each run
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)

Based on actual size, the number of batches is flexible. 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)

![Diagram of Jungle and Cow B+tree]

(1) Not much unique data $\rightarrow$ in-place merge

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

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 $> \text{limit (} C \times \text{original sorted run size limit})$

Cow B+tree
Write batch
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 ($C \times$ original sorted run size limit)

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

Same as cow B+tree compaction
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)

Level X

Level X-1

Victim 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)

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

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

Cow B+tree
Write batch
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)

- Level X-1
- Victim cow B+tree

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

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

Cow B+tree
Write batch

Victim cow B+tree

Jungle

1/26
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)

- Smaller C
  - Frequent merge: more writes, less space
  - Bigger C
  - Infrequent merge: less writes, more space

- Read cost remains (nearly) the same

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)

Tiering merge LSM-tree

Level X-1

Level X

M runs, where \( M < T \)?

T overlapping stacks

New sorted run size: \( \frac{M}{T} \) of the original one
**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)

Tiering merge LSM-tree

Level X-1

Level X

Level X+1

M runs, where $M < T$?

T overlapping stacks

T overlapping stacks

New sorted run size: $M/T$ of the original one

Getting smaller

→ benefit of delaying merge will be small
→ Need to force $M = T$
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?

![Diagram of tiering merge LSM-tree]

Filled up \( (M = T) \)

Level X-1

Merge & put

Level X

Cold Hot Cold
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?

Tiering merge LSM-tree

Filled up \((M == T)\)

Level X-1

Merge & put

Level X

Cold Hot Cold
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?

Tiering merge LSM-tree

Level X-1

Level X

Filled up \((M == T)\)

Merge & put

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?

Level X-1

- Both
  - number of batches, and
  - size of batch
do not matter

Filled up, lots of unique keys

Inter-level merge

Level X

Cold

Hot

Cold
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?

![Diagram of Jungle]

- Level X-1
  - Both
    - number of batches, and
    - size of batch
  - do not matter
  - Filled up, lots of unique keys
  - Inter-level merge
    - Just append until it reaches threshold → avoid redundant merges

- Level X
  - Cold
  - Hot
  - Cold
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)

**Space amplification**

<table>
<thead>
<tr>
<th>Leveling</th>
<th>C=2</th>
<th>C=3</th>
<th>C=5</th>
<th>C=10</th>
<th>Tiering</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.27</td>
<td>2.1416</td>
<td>2.8326</td>
<td>4.8654</td>
<td>6.6326</td>
<td>3.2424</td>
</tr>
</tbody>
</table>

**Write amplification**

<table>
<thead>
<tr>
<th>Leveling</th>
<th>C=2</th>
<th>C=3</th>
<th>C=5</th>
<th>C=10</th>
<th>Tiering</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.403</td>
<td>5.3815</td>
<td>4.9025</td>
<td>4.506</td>
<td>4.2822</td>
<td>4.57</td>
</tr>
</tbody>
</table>

**Write amplification (zoomed in)**

<table>
<thead>
<tr>
<th>Leveling</th>
<th>C=2</th>
<th>C=3</th>
<th>C=5</th>
<th>C=10</th>
<th>Tiering</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3815</td>
<td>4.9025</td>
<td>4.506</td>
<td>4.2822</td>
<td>4.57</td>
<td></td>
</tr>
</tbody>
</table>

*Jungle with different parameters*
Brief Evaluation

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

![Normalized read throughput](image)

![Normalized read latencies](image)
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