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

Jung-Sang Ahn Mohiuddin Abdul Qader Woon-Hak Kang Hieu Nguyen
Guogen Zhang Sami Ben-Romdhane
eBay Inc.

Abstract

Designing key-value stores based on log-structured merge-tree (LSM-tree) encounters a well-known trade-off between the I/O cost of update and that of lookup as well as of space usage. It is generally believed that they cannot be improved at the same time; reducing update cost will increase lookup cost and space usage, and vice versa. Recent works have been addressing this issue, but they focus on probabilistic approaches or reducing amortized cost only, which may not be helpful for tail latency that is critical to server applications. This paper suggests a novel approach that transplants copy-on-write B$^+$-tree into LSM-tree, aiming at reducing update cost without sacrificing lookup cost. In addition to that, our scheme provides a simple and practical way to adjust the index between update-optimized form and space-optimized form. The evaluation results show that it significantly reduces update cost with consistent lookup cost.

1 Introduction

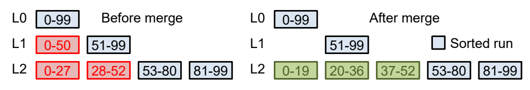
Lots of modern key-value stores use LSM-tree [17] due to its high write throughput and acceptable read performance. It is commonly implemented as a leveled index structure, where each level consists of multiple sorted runs (a.k.a. sorted string tables – SSTables [7]). Incoming updates are buffered in memory and then merged into sorted runs at the topmost level. There is a capacity limit of each level, and the limit increases exponentially over levels. Once a level becomes full, some sorted runs in that level are merged (or compacted) into the overlapping sorted runs in the next level. Since all disk I/Os during merge are fully sequential, LSM-tree has much better write performance than B$^+$-tree.

However, the amount of actual writes for a single merge operation is huge compared to the size of actual data to be merged, as all affected sorted runs in the next level are rewritten in an out-of-place update manner. As merge keeps happening throughout the entire lifetime of LSM-tree, the cost of merge has a great impact on not only the amount of disk I/O but also CPU usage. One of recent optimizations for reducing merge overhead is *tiering* [10, 11, 15, 19], which defers merging and instead maintains a stack of multiple sorted runs for the same key range in the same level. Since it does not rewrite existing sorted runs, we can reduce the overall update cost. Instead, the space occupied by the index will increase as much as deferred merges, since there will be duplicate sorted runs for the same key range. Moreover, lookup cost also increases due to visiting more sorted runs to find the given key. As a result, it introduces a three-dimensional relationship between update cost, lookup cost, and space cost [4, 5].

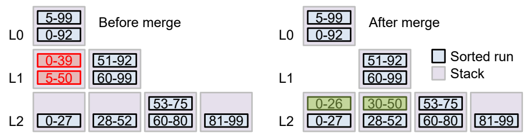
To moderate update cost without degrading lookup cost, this paper suggests a new LSM-tree approach *Jungle*, which replaces each sorted run with copy-on-write (CoW) B$^+$-tree [6, 8, 9, 13, 20] that stores keys and values separately [2, 3, 14]. When a new write batch of key-value pairs comes in, Jungle sorts it by the key, appends the values to the end of the B$^+$-tree file, and then appends the updated nodes which store only the keys and references to their values. Separating keys and values is an important enabler for our target use cases, where values are much larger than keys.

Since multiple batches are accumulated in a chronological order, Jungle has many similarities to the stack of sorted runs in tiering. The biggest difference from existing tiering is the fact that the lookup cost of CoW B$^+$-tree is not affected by the number of batches (i.e., sorted runs) appended to the tree; we do not need to visit multiple sorted runs for a single lookup. It makes tiering more flexible so that we can break free from the limit on the number of sorted runs in a stack.

Due to its append-only characteristics, CoW B$^+$-tree requires periodic *compaction* task which merges all appended batches and then rewrites the entire B$^+$-tree in a sorted order. By adjusting the frequency of this compaction, we can easily tune LSM-tree; if compaction happens less frequently, CoW B$^+$-tree will have more batches so that overall update cost will decrease, but instead it will have increased space cost. More frequent compaction will reduce the space cost but the update cost will be close to that of the original LSM-tree. Note that lookup cost will remain almost the same regardless of the frequency. The experimental results show that Jungle can adjust update cost and space cost according to compaction



(a) Leveling merge ($T = 2$)



(b) Tiering merge ($T = 2$)

Figure 1: Examples of LSM-tree merge.

frequency, without sacrificing the lookup cost.

2 Background

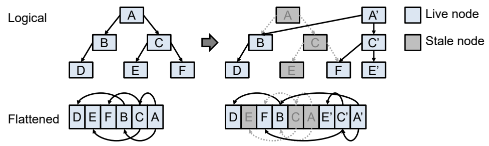
2.1 LSM-Tree Merge and Trade-Offs

The basic LSM-tree merge operation is commonly referred to as *leveling*, as described in Figure 1(a). Each box indicates a sorted run, and the number in the box denotes its key range. L0, L1, and L2 mean each level, and boxes in red and green background represent sorted runs to be merged and sorted runs as a result of merge, respectively.

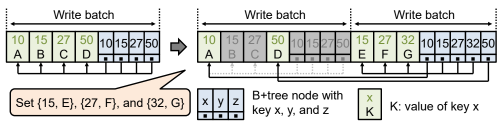
Once a level becomes full, it picks a victim sorted run in that level (i.e., 0-50 in L1), and then finds all sorted runs in the next level whose key range is overlapping (i.e., 0-27 and 28-52 in L2). After doing merge sort, new sorted runs are written in the next level and the previous sorted runs in each level are removed. There is a size limit of a single sorted run, and a sorted run that exceeds the size limit needs to be split. Hence, the next level may have more sorted runs after merge due to the increased number of keys. Since it rewrites all overlapping sorted runs in the next level, the ratio between the size of original data to be merged and the amount of data actually written, called *write amplification*, will be proportional to the size ratio between each level. If the size of level $l+1$ is T times bigger than that of level l, the write amplification of a merge operation will be T, when keys are uniformly distributed.

Figure 1(b) represents tiering merge. Outer box indicates a stack of sorted runs within the same key range. The upper run is newer than lower runs, and they can have duplicate keys. There is a limit of the number of runs in a stack, and maintained up to T runs. When a stack becomes full, T runs in the victim stack (i.e., 0-39 and 5-50) are merged, split, and then added to stacks in the next level (i.e., 0-26 and 30-50). Since it does not rewrite existing runs in the next level, the write amplification of a merge operation gets much smaller. Instead, the ratio between the space occupied by the entire index and the actual live data size, called *space amplification*, will be increased up to T times more than that of leveling, and lookup cost will be up to T times worse as well.

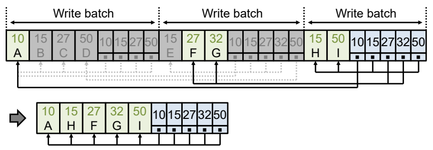
Modern LSM-trees improve lookup performance by maintaining a Bloom filter [15, 16] which is used to quickly reduce the set of sorted runs that need to be examined during a lookup.



(a) Updating node E. Instead of overwriting E, append the updated node E′. It requires the update of parent nodes C and A, and appends C′ and A′ as well.


Write batch
Write batch
Write batch
Set {15, E}, {27, F}, and {32, G}
B+tree node with key x, y, and z
K: value of key x

(b) Set key-value pairs {15,E}, {27,F}, and {32,G} in batch. Append the value part (i.e., E, F, and G) first, and then append the entire B^+-tree node with updated references. It involves copying existing, but not updated keys in the same node: 10 and 50.



(c) Compaction. All values as well as B^+-tree nodes are rewritten in a key order.

Figure 2: Examples of copy-on-write B^+-tree.

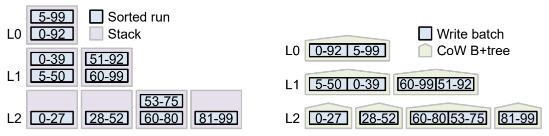
Bloom filters can return false positives, so the tail latencies such as the 99-th or the 99.9-th percentiles are still influenced by the number of levels and stack sizes.

2.2 Copy-On-Write B^+-Tree

Copy-on-write (or append-only) B^+-tree is a variant of B^+-tree which handles incoming updates in an out-of-place update manner, as illustrated in Figure 2(a). Once a node E is updated, the modified new node E′ is appended to the end of the file instead of overwriting E. Since the location of the node has been changed, the update should be cascaded from leaf to root. As a result, its parent nodes C and A are also updated to C′ and A′, respectively.

Since appending all nodes from leaf to root for every key-value update is too expensive, CoW B^+-trees usually append updates in batch, which reduces write amplification a lot. A common optimization to decrease write amplification further is decoupling values from B^+-tree leaf nodes [2, 3]: append the contents of values outside the B^+-tree node first, and put the pairs of key and reference to its value (i.e., byte offset) into leaf node, as shown in Figure 2(b). If 1) each write batch contains enough number of key-value pairs and 2) the sizes of values are sufficiently bigger than keys, the additional cost of copying B^+-tree nodes will be small compared to the cost of appending the entire write batch.

Once the space usage of data referred to by CoW B^+-tree exceeds a certain threshold, it triggers a compaction task which iterates the tree in a key order, copies latest key-value



(a) Tiering LSM-tree (b) Jungle

Figure 3: Tiering LSM-tree and corresponding Jungle.

pairs to new space, constructs a new B$^+$-tree in a bottom-up manner, and then finally removes the previous tree, as described in Figure 2(c).

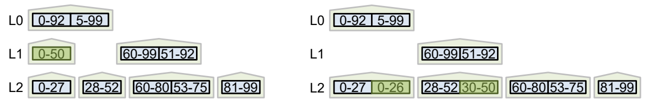
3 Jungle: Planting LSM-Tree and Copy-On-Write B$^+$-Tree Together

To address the goals outlined in the introduction, we propose Jungle, which is a hybrid LSM-tree approach where sorted runs are replaced by CoW B$^+$-trees. Since a CoW B$^+$-tree can contain multiple write batches in a chronological order, it shares a number of common features with a stack of sorted runs in tiering, if all write batches are locally sorted. Hence, Jungle can be treated as a variant of tiering merge as described in Figure 3. Each CoW B$^+$-tree in Jungle is corresponding to the stack of sorted runs in tiering LSM-tree, where each sorted run in a stack matches up with each write batch in a CoW B$^+$-tree.

One big benefit of using CoW B$^+$-tree is lookup cost. In tiering LSM-tree, the lookup cost is roughly proportional to the size of stack (i.e., the number of sorted runs in a stack) as it should search each sorted run until it finds the given key. Suppose that the maximum number of keys that a single sorted run can contain is R and there are T runs in a stack, the lookup cost per level is $O(T \log R)$. Even with bloom filter, the stack sizes directly affect the possibility of false positives, which is one of reasons why there is a limit on the maximum number of sorted runs in a stack: usually T.

However, the lookup cost of a CoW B$^+$-tree is solely affected by the total number of unique keys in the tree, regardless of the number of write batches. If T different write batches in a tree have duplicate keys so that there are R unique keys, the lookup cost per level will be $O(\log R)$. If all keys in T write batches are unique, the lookup cost per level increases to $O(\log(T \cdot R)) = O(\log T + \log R)$, which is still significantly smaller than that of tiering, as T is a very small constant number compared to R. Consequently, it enables us to have the flexible number of write batches while keeping nearly the same lookup cost as in leveling merge: $O(\log R)$ per level. This approach is orthogonal to Bloom filter, which we can still use to improve read performance.

Jungle has two types of merge operations: *in-place merge* and *inter-level merge*. In-place merge is the same as CoW B$^+$-tree's compaction, while inter-level merge is identical to the tiering merge of LSM-tree. Figure 4 depicts the examples of Jungle merge operations with the victim CoW B$^+$-tree in `L1`, containing write batches `5-50` and `0-39` in Figure 3(b).



(a) After in-place merge (b) After inter-level merge

Figure 4: Jungle merge operations.

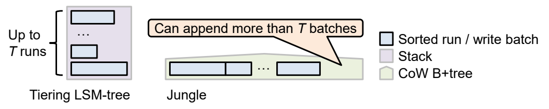


Figure 5: Merge threshold comparison when the sizes of sorted runs are not even.

An advantage of CoW B$^+$-tree over the original tiering is that we can easily get the total number of unique keys (i.e., the latest version of key-value pairs) and their space usage (i.e., the space occupied by those pairs) without causing extra I/O. If the space occupied by unique key-value pairs is smaller than the size limit of the level, the level is not full yet, so merging them into the next level is not necessary. In such case, we trigger an in-place merge so that the victim CoW B$^+$-tree is compacted without affecting other trees, as presented in Figure 4(a). If the actual data size exceeds the limit of the level, then an inter-level merge will be invoked. It iterates the victim CoW B$^+$-tree and appends new write batches into proper CoW B$^+$-trees in the next level. Figure 4(b) describes the outcome of inter-level merge, where the overall shape is the same as the result of tiering merge in Figure 1(b).

Merge Frequency The compaction threshold of CoW B$^+$-tree C is defined as the ratio between the amount of space occupied by the tree including the stale data and the size of the live data. If $C = 3$, an in-place merge will be triggered when the space usage of a CoW B$^+$-tree and its data becomes bigger than 3x of the size of its live key-value pairs. That is similar to triggering merge upon 3 sorted runs in tiering LSM-tree. Even before reaching the threshold, an inter-level merge may happen if the total size of the level exceeds the limit, which is C times bigger than the limit in leveling.

Having threshold in terms of size has much more benefits than that in terms of the number of sorted runs. In tiering, the number of sorted runs is strictly limited to T, and merge happens regardless of the actual size of sorted runs. If the average size of runs in a stack is smaller than its limit S, merge will happen earlier than that of Jungle with $C = T$, as Jungle will trigger merge upon the actual size limit, i.e., $C \cdot S$. In Jungle's perspective, neither the size of individual write batch nor the total number of write batches in the tree matters; thus it can accommodate more write batches if their average size is smaller than S, as illustrated in Figure 5.

As a result, the total number of merges invoked in tiering will be the same as that in Jungle with $C = T$, only when all sorted runs are evenly sized to S. Otherwise, tiering will likely have more merges than Jungle, which will increases the

Table 1: Cost comparison.

	Point	(w/ Bloom filter)	Range	Write amplification	Space amplification
Leveling	$O(L \cdot \log R)$	$O\big((1+p\cdot L)\log R\big)$	$O(L)$	$O(L\cdot T)$	$O\big(\frac{T}{T-1}\big)$
Tiering	$O(L\cdot T\cdot \log R)$	$O\big((1+p\cdot L\cdot T)\log R\big)$	$O(L\cdot T)$	$O(L)$	$O\big(\frac{T^2}{T-1}\big)$
Jungle	$O\big(L(\log C+\log R)\big)$	$O\big((1+p\cdot L)(\log C+\log R)\big)$	$O(L)$	$O\big((L+\frac{1}{C})(1+B_w)\big)$	$O\big(\frac{C\cdot T}{T-1}(1+B_s)\big)$

overall write amplification. The merge frequency gap between tiering and Jungle gets bigger with more skewed incoming data. Under non-uniform key distribution, there will be a number of cold stacks whose keys are updated rarely and sparsely. Most likely those stacks are filled up with small sorted runs and contribute to the increasing of write amplification.

Making matters worse, although we put uniform random data, the actual size distribution of sorted runs is not even in practice. It is influenced by various factors such as key range partitioning in a level, victim stack selection policy, or probabilistic problems like *balls into bins* [18].

Note that the reason why tiering cannot allow more than T runs to avoid such issues is because of lookup performance: more runs will make the lookup cost more expensive. In addition, tiering cannot reduce the size limit of a stack to be less than T either, due to the aforementioned small run issue. Suppose that the stack limit is M runs, where $M < T$. Assuming the ideal case where the size of each run before merge is S, and given that the key range of a stack in level l is overlapping with T stacks in the next level $l+1$, the merge will generate T runs whose size is $\frac{M\cdot S}{T}$ each. Since $M < T$, new runs become smaller than old runs, and accordingly it will make tiering invoke merge even more frequently. Hence, the number of sorted runs in a stack in tiering should be the same as the size ratio between adjacent levels, which is T.

In Jungle, the compaction threshold of CoW B$^+$-tree C is easily adjustable, as it hardly influences the lookup performance. In other words, we can use it as a tunable knob between update cost and space cost. With smaller C, merge will happen more frequently so that the space cost will be reduced. If we increase the value of C, it makes merge less frequent and accordingly the overall update cost will decrease. There are multiple options for setting threshold: 1) a single global value, 2) vertically different values per level, and 3) horizontally different values per key range according to workload patterns and locality. In this paper, we only cover the first option, and will explore the potential improvements by the second and third options in the future.

Cost Analysis Table 1 summarizes the cost of each operation in leveling, tiering, and Jungle. R means the maximum number of keys that a single sorted run can contain, and L denotes the number of levels in LSM-tree. p indicates the false positive rate of Bloom filter: $p = (1-e^{-\frac{kn}{m}})^k$ [16]. m and n denote the number of bits in Bloom filter and the total number of keys, while k indicates the number of hash functions. With $\frac{n}{m} = \frac{1}{10}$ and $k = 3$, p is around 1.7%. We assume that the same values of $\frac{n}{m}$ and k are used for all levels. Since leveling and tiering have been analyzed well [10–12, 15], we omit their derivation details here.

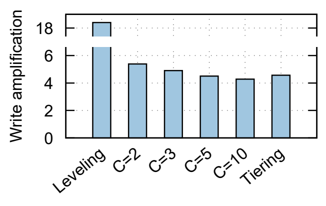
In Jungle, the point lookup cost is basically identical to that in leveling, except for the number of keys in each CoW B$^+$-tree: up to $C\cdot R$ keys at worst. The range lookup cost will remain the same, as it is not affected by the height of B$^+$-tree. Both write and space amplification are influenced by the compaction threshold C. If $C = T$, they become very close to those of tiering, except for B_w and B_s, which represent the extra write and space overhead of CoW B$^+$-tree nodes.

Suppose that the average size of values is V and that of key and reference pairs in B$^+$-tree node is K. For each write batch containing R key-value pairs, we first append a set of values which is $R\cdot V$ big, and then append a set of B$^+$-tree nodes whose size is roughly $R\cdot K$. Hence, the value of both B_w and B_s will be $\frac{R\cdot K}{R\cdot V} = \frac{K}{V}$. However, in the worst-case scenario that all keys in C batches are unique, the size of existing CoW B$^+$-tree nodes before appending a new batch will be $C\cdot R\cdot K$. We may need to rewrite all those nodes for a single batch append, and accordingly the value of B_w and B_s will be increased to $\frac{C\cdot K}{V}$. With V =1 KB, K =16 bytes, and $C = 10$, they will be ranged between 0.016 and 0.16. Those values are usually smaller than 1 when keys are smaller than values, as mentioned in Section 2.2.

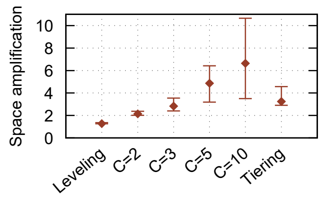
With smaller C, the space cost of Jungle becomes close to that of leveling. However, the write amplification still remains much smaller than leveling since Jungle just appends a new write batch to existing CoW B$^+$-tree, except for in-place merge. Most likely in-place merge will happen at the last level, where all keys are eventually settled down. Write batches for the last level will probably contain keys that already exist, so that we may trigger in-place merge after appending C write batches. Hence, C mostly affects the frequency of in-place merge. This is one of benefits of Jungle as we can maintain the value of C smaller than T, which results in less space usage than tiering while keeping similar write amplification.

4 Evaluation

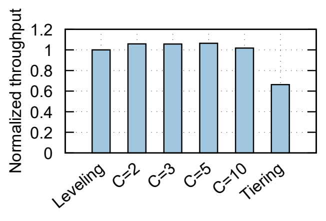
We evaluate Jungle to see if it efficiently adjusts write and space amplification according to its compaction threshold C, with minimum lookup performance degradation. Jungle is implemented as a stand-alone library embedded in eBay's home-grown distributed server platform, which is used for multiple difference services. There are internal options to run itself in pure LSM leveling or tiering mode, so that we



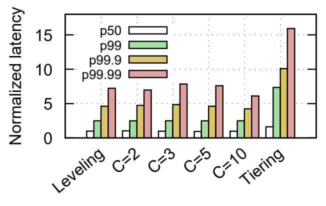
(a) Write amplification



(b) Space amplification



(c) Point lookup throughput



(d) Tail latencies of point lookup

Figure 6: Evaluation results, `C=x` denotes Jungle with compaction threshold `x`. The throughput and latencies of point lookup are normalized to the throughput and the 50-th percentile latency of leveling, respectively.

compare the costs of each mode with Jungle. In this paper, we do not measure the overall performance against widely used key-value stores such as RocksDB [1], as they include a number of optimizations that are orthogonal to the improvement of Jungle over the original leveling and tiering schemes.

The evaluation was performed on Dell T7820, equipped with Samsung 860 QVO 1 TB SSD, formatted using the Ext4. We initialize each key-value store with 20 million random records, whose key and value size are 8 bytes and 1,024 bytes on average, so that the total space usage is expected to be ranged from 22 GB (leveling) to 222 GB (`C=10` or tiering). To minimize the effect of OS page cache, we limit the available RAM size to 2.5 GB. Level size ratio T and the size of level-0 are set to 10 and 256 MB, respectively. The size limit of each sorted run is 64 MB, and the size of Bloom filter is set to 10 bits per key. CoW B$^+$-tree node size is set to 4 KB. After initial load, we randomly issue more than 400 million update operations so that the entire records are updated at least 20 times. We believe the enough number of merge operations happen during the evaluation.

We first plot the write amplification of each scheme in Figure 6(a). The original leveling shows the highest write amplification, as it always rewrites all overlapping sorted runs during merge. By contrast, the write amplification of tiering is almost 4 times smaller than leveling. Jungle has similar write amplification to tiering, and it increases as C gets smaller. A notable fact is that the write amplification of Jungle is more than 3 times smaller than that of leveling, even with `C=2`. This is because C most likely affects the in-place merge frequency of the last level, as explained in Section 3.

Figure 6(b) illustrates space amplification. Since space usage fluctuates over time, we measure the average number as well as the maximum and minimum numbers. Leveling has the smallest space usage which is close to 1.2 times of the original working set size. The space amplification of tiering is expected to be 11.1, but the actual measured number is much smaller than that: 3.24 on average. There are two reasons why it happens. Firstly, merge cannot happen on all stacks at the same time, so that some stacks are full while others are not. Thus the average stack size is always smaller than the maximum size. Secondly, the size of sorted runs in a stack is not even in practice, as we mentioned in Section 3. Eventually it cannot fully utilize the space capacity, and merge operations are triggered more often, thus we observe tiering shows bigger write amplification than that of even `C=5`.

On the other hand, Jungle's space amplification range is more predictable, even with the same merge timing and victim policy. That is because its compaction threshold is based on the actual data size ratio, not the total number of write batches in a CoW B$^+$-tree.

Next, we randomly invoke point lookup on those aged indexes, and measure the throughput and latencies as shown in Figure 6(c) and Figure 6(d), respectively. No write operation is performed during read, to get pure latency without any impact on sharing I/O bandwidth or lock contention. Even though there are up to 10 times more sorted runs to visit, the point lookup throughput of tiering is only 30% worse than that of leveling, by help of Bloom filter. On the contrary, Jungle shows even better performance than leveling. Since the capacity of each level in Jungle is bigger than that in leveling, more key-value pairs can be found in upper level and then returned earlier, and consequently it improves the read performance slightly.

Unlike throughput, Bloom filter can barely help reducing tail latencies; the 99-th percentile latency of tiering is 3 times higher than that of leveling. Jungle keeps showing comparable latencies regardless of the value of C, as CoW B$^+$-tree provides nearly the same lookup performance no matter how many write batches are in the tree.

5 Conclusion

This paper presents Jungle, a hybrid approach by combining LSM-tree and CoW B$^+$-tree. The main objective of Jungle is to cut off the relation between update cost and lookup cost, so as to make the index of key-value store more flexible for optimizing. Our evaluation results show that the write and space amplification of Jungle can be easily tuned by adjusting the compaction threshold of CoW B$^+$-tree, with the minimum impact on lookup cost.

Acknowledgments

We would like to thank our shepherd Peter Macko and the anonymous reviewers for their invaluable comments.

Discussion Topics

Chances of LSM-tree deformation Jungle shows the feasibility of breaking LSM-tree's limits, which have been strictly bounded by the relationship between lookup cost, update cost, and space cost. By eliminating the lookup cost from the trade-offs, it provides new chances of tackling existing design problems. For instance,

- The reason why LSM-tree implementations have been maintaining multiple levels, despite of the read performance degradation compared to B$^+$-tree, is to keep the ratio between adjacent levels T constant. Fewer levels make T bigger, which results in higher write amplification.

- LSM-tree approaches have been passive about adopting different merge policies for hot/cold separation, under non-uniform workloads. Deferring merge for hot key range may incur augmented lookup cost of the same range.

We can re-think such fundamentals.

Limitations There are potential limitations we need to study more.

- **Range lookup.** If there are a number of tiny write batches in a CoW B$^+$-tree, the access pattern will be more random which results in range lookup degradation. We believe less than T batches will be fine by help of readahead.

- **Long keys.** Longer keys make B_w and B_s values bigger. We may use variants of CoW B$^+$-tree [2] to moderate this overhead.

State-of-the-art approaches In Dostoevsky [11] paper, Dayan et al. have suggested an approach about setting the size limit of stack in each level differently. Since the last level has a dominant impact on the cost of update and space, we can reduce the overall cost by setting the stack size of the last level smaller. They also have proposed a model to find optimal stack size of each level, which is similar to their precedent paper Monkey [10] that explores the optimal ratio of Bloom filter size across each level. These schemes are orthogonal to Jungle as the stack size limit can be translated to compaction threshold in Jungle. We can use the same approach to reduce operation costs further.

PebblesDB [19] has been proposed for key-range partitioning policy of tiering merge, where it calls the boundary of stack key range as *Guard*. Jungle may adopt its approach for creating or splitting CoW B$^+$-trees efficiently, to avoid key range skew that may cause superfluous merging.

References

[1] *RocksDB: A persistent key-value store for fast storage environments*. https://rocksdb.org/.

[2] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. Forestdb: A fast key-value storage system for variable-length string keys. *IEEE Transactions on Computers (TC)*, 65(3):902–915, 2016.

[3] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide: Time to Relax*. "O'Reilly Media, Inc.", 2010.

[4] Manos Athanassoulis and Stratos Idreos. Design tradeoffs of data access methods. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 2195–2200. ACM, 2016.

[5] Manos Athanassoulis, Michael S Kester, Lukas M Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. Designing access methods: The rum conjecture. In *EDBT*, volume 2016, pages 461–466, 2016.

[6] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. Technical report, 2003.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[8] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[9] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, Robert N. Sidebotham, and Transarc Corporation. The episode file system. pages 43–60, 1992.

[10] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 79–94. ACM, 2017.

[11] Niv Dayan and Stratos Idreos. Dostoevsky: Better spacetime trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 505–520. ACM, 2018.

[12] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, et al. Design continuums and the path toward self-designing key-value stores that know and learn. In *Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[13] Wook-Hee Kim, Beomseo Nam, Dongil Park, and Youji Won. Resolving journaling of journal anomaly in android i/o: multi-version b-tree with lazy split. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 273–285, 2014.

[14] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: separating keys from values in ssd-conscious storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 133–148, 2016.

[15] Chen Luo and Michael J. Carey. Lsm-based storage techniques: A survey. *CoRR*, abs/1812.07527, 2018.

[16] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge university press, 2005.

[17] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[18] Martin Raab and Angelika Steger. Balls into bins—a simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

[19] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 497–514. ACM, 2017.

[20] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.