Splaying Log-Structured Merge-Trees

Thomas Lively, Luca Schroeder, Carlos Mendizábal
Harvard University, Cambridge, MA
{tlively,schroeder,carlosmendizabal}@college.harvard.edu

ABSTRACT

Modern persistent key-value stores typically use a log-structured merge-tree (LSM-tree) design, which allows for high write throughput. Our observation is that the LSM-tree, however, has suboptimal performance during read-intensive workload windows with non-uniform key access distributions. To address this shortcoming, we propose and analyze a simple decision scheme that can be added to any LSM-based key-value store and dramatically reduce the number of disk I/Os for these classes of workloads. The key insight is that copying a frequently accessed key to the top of an LSM-tree ("splaying") allows cheaper reads on that key in the near future.

ACM Reference Format:
Thomas Lively, Luca Schroeder, Carlos Mendizábal. 2018. Splaying Log-Structured Merge-Trees. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3183713.3183723

1 INTRODUCTION

Modern persistent key-value stores usually use a log-structured merge-tree (LSM-tree) design [4], in which updates are written to a sequential log to avoid random writes and thus achieve high write throughput. The LSM-tree is typically organized into levels of exponentially increasing size, with the smallest level being stored in-memory and the rest stored on disk. When a level becomes full, it merges with the next larger level, and merges may cascade to the bottom of the tree. LSM-tree designs will often include in-memory Bloom filters [1] for every level as well as fence pointers, both of which reduce the I/O cost of point lookups [2].

The Problem: Suboptimal Read Performance. Although LSM-tree designs are deployed in write-intensive applications, reads are inevitably an important part of the workload of any key-value store. Moreover, even in write-heavy workloads there may be short windows of read-heavy activity. However, the LSM-tree is suboptimal for reads, since the tree is not organized based on what is likely to be accessed in the future; the top of the tree contains the most recently updated key-value pairs, which might not be read soon. A frequently accessed key that is in a lower level of the LSM-tree may thus cause expensive reads in multiple levels of the tree. The standard solution is to add a cache, but this has several drawbacks: first, memory given to the cache could instead be used for the in-memory level, which would decrease the frequency of costly merges; second, when the workload shifts back to predominantly writes, the cache becomes less useful; third, the hot set of frequently accessed keys may exceed the size of the cache, suggesting the cache is only a partial solution.

The Solution: Splaying. In this paper, we show how a simple decision scheme that can be added to *any* LSM-based key-value store can dramatically reduce the number of disk I/Os for read-intensive workloads with non-uniform key access distributions. The core idea, inspired by the classic splay tree data structure [5], is to copy (or "splay") frequently accessed keys to the top of the LSM-tree. The cost of making such a copy—namely the disk I/O associated with the future merges the splayed copy will participate in—will be offset by faster gets on that key in the near future. Unlike a cache, splaying does not consume memory that, in the worst case, becomes useless during write-intensive periods. Moreover, since a splayed copy can be merged down to disk, splayed keys confer a benefit even if they are no longer in memory, as long as the splayed key is in a level above the level of the original key; thus in contrast to the cache, splaying is still able to provide substantial performance gains even if the set of frequently accessed keys exceeds the size of main memory.

2 SPLAYING

There are many possible schemes for deciding whether to splay a key or not. The natural point at which to make this decision is right after a read on a key, since we will then have information on what level a key resides in.

AlwaysSplay. A naïve strategy is to always splay; that is, to have every 'get' on a key trigger a 'put' on the same key. While simple, this strategy is likely suboptimal as it does not distinguish between frequently accessed keys (which may be worth splaying) and less frequently accessed keys (which may not be). Moreover, use of this strategy will create many splayed copies of accessed keys in the LSM-tree, inflating the number of levels and the worst-case point lookup time. If there is a shift in either the read/write ratio of the workload or the set of frequently accessed keys, these copies may no longer be helpful—meaning we have temporarily raised the cost of 'get' operations without any benefit. (Note that splayed copies behave like normal key-value pairs and will eventually be merged with the original key.)

FlexSplay. To address the shortcomings of the always-splay strategy, we introduce an adaptive strategy called "FlexSplay." Under FlexSplay, every 'get' operation triggers a 'put' as before, but the duplicate key-value pair is marked as splayed. During every downward merge, we decide whether to delete or keep the splayed copies, using the information available at run-time to estimate the expected benefit and cost (in disk I/Os) of keeping them. Deleting splayed keys is always safe because they are copies of keys lower in the tree. Note that splaying every read key to the in-memory level has zero disk I/O cost, as we can always delete the splayed copy when merging to disk later on.

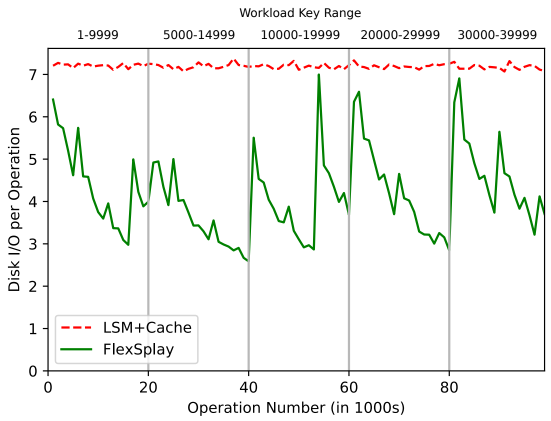

Figure 1: Splaying allows LSM-trees to adapt to workload shifts (indicated by dotted vertical lines)

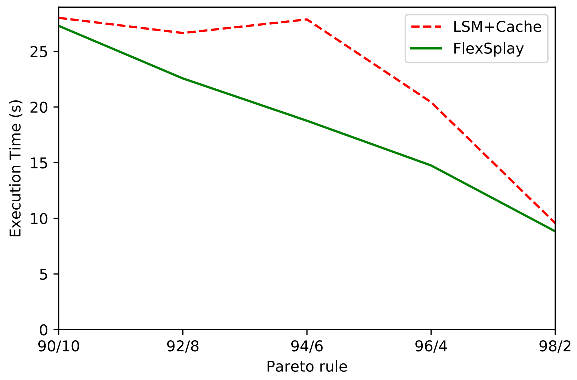

Figure 2: In RocksDB, splaying is able to achieve up to 33% speedup for read-intensive workloads

The expected cost of keeping a splayed key-value pair during a merge is the cost of the additional merges the splayed copy will participate in. The expected benefit is that the single next access on the key (after which the pair is again splayed to the in-memory layer) is perhaps cheaper. By making more incremental decisions, FlexSplay is better able to adapt to workload changes: if the set of frequently accessed keys changes or the number of reads decreases, splayed copies will gradually be removed from the tree. This adaptivity comes at the cost of some additional computational work and a small increase in the size of key-value pairs to accommodate a few bits of metadata.

3 ANALYSIS

In this section we present promising results that highlight conditions under which splaying is particularly advantageous.

Adaptivity through Splaying. To rapidly explore the space of possible workloads and splaying strategies, we developed a leveled LSM-tree simulator in Python. The simulator models an idealized system and computes disk I/O cost over workloads with varying read/write ratios, access distributions, and main memory budgets.

Our simulator experiment demonstrates that splaying can allow LSM-trees to adapt to changes in workload characteristics and thereby achieve better performance. We load 100,000 key-value pairs in random order into two LSM-trees with the same main memory budget. The first tree splits the memory evenly between the in-memory level (L0) and an LRU cache. The second tree uses the memory for L0 and the metadata required for the FlexSplay strategy, which it employs. Neither tree uses Bloom filters. Each tree reads uniformly at random from a sequence of overlapping key ranges, none of which fits in memory.

The average disk I/O per read across these simulated workloads is graphed for each tree in Figure 1. The cost for a no-splay LSM-tree is constant, since the insertion order is random and the structure of the tree does not change (no writes). The cache only fits 5% of the key-value pairs in a 'hot' key range and thus has negligible impact. In contrast, the per-operation I/O cost for the FlexSplay tree rapidly decreases as the 'hot' key range is brought to the top levels of the tree. Spikes in the graph correspond to merges, which are more frequent during workload shifts when many keys are splayed.

Performance Gains of Splaying. Although the previous experiment is highly stylized, we find that the superior performance of splaying also holds in a real-world system, RocksDB [3]. We conducted experiments on a 2.7 GHz Intel Core i7 Apple MacBook Pro (2016). RocksDB was configured to use Bloom filters with 1% false positive rate. Each level on disk is twice the size of the previous level. We compare a non-splaying implementation that evenly splits a 256KB main memory budget between L0 and the cache with a splaying implementation that uses the memory only for L0. Overall, this configuration is a small-scale model of a production system. The splaying strategy used is an approximation of FlexSplay, in which keys are always copied to the in-memory level but never merged down. Since in some cases it may have been optimal to merge down, the following results *underapproximate* the performance achievable by FlexSplay. The data set consists of one million keys, loaded into the database in random order prior to each test.

Figure 2 shows the performance of these implementations for different access distributions under read-only workloads (when the cache is most beneficial). The access distributions are given by the Pareto rule; '98/2' means 98% of the operations touch 2% of the keys. The splaying implementation is faster in all cases and achieves up to 33% speedup. Splaying performs comparatively well when the set of frequently accessed keys is large enough to exceed the size of the cache but small enough to fit in the topmost levels.

4 CONCLUSION

In this paper, we have shown how a simple decision scheme can be added to any LSM-based key-value store to reduce disk I/Os for read-intensive workloads with non-uniform access distributions. The key insight is to copy frequently accessed keys to the top levels of the LSM-tree to make future reads cheaper. Our results show that splaying can potentially provide substantial performance gains and workload adaptivity; a simplified splaying implementation in RocksDB achieves up to 33% speedup over a cache design, even in an all-reads workload where a cache is most useful.

ACKNOWLEDGMENTS

We are indebted to Stratos Idreos for his mentorship and guidance throughout this project.

REFERENCES

[1] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13(7): 422-426, 1970.
[2] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Navigable Key-Value Store. In *SIGMOD*, 2017.
[3] Facebook. RocksDB. *https://github.com/facebook/rocksdb*.
[4] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4): 351-385, 1996.
[5] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the Association for Computing Machinery*, 32 (3): 652-686, 1985.