2017 IEEE 33rd International Conference on Data Engineering

On Log-Structured Merge for Solid-State Drives

Risi Thonangi
VMware Inc.
rthonangi@vmware.com

Jun Yang
Duke University
junyang@cs.duke.edu

***Abstract*—Log-structure merge (LSM) is an increasingly prevalent approach to indexing, especially for modern write-heavy workloads. LSM organizes data in levels with geometrically increasing sizes. Records enter the top level; whenever a level fills up, it is merged down into the next level. Hence, the index is updated only through merges and records are never updated in-place. While originally conceived to avoid slow random accesses of hard drives, LSM also turns out to be especially suited to solid-state drives, or any block-based storage with expensive writes.**

We study how to further reduce writes in LSM. Traditionally, LSM always merges an overflowing level fully into the next. We investigate in depth how partial merges save writes and prove bounds on their effectiveness. We propose new algorithms that make provably good decisions on whether to perform a partial merge, and if yes, which part of a level to merge. We also show how to further reduce writes by reusing data blocks during merges. Overall, our approach offers better worst-case guarantees and better practical performance than existing LSM variants.

I. Introduction

Log-structured merge (*LSM*) [16] has become an increasingly popular alternative to B-tree for indexing data in recent years. An LSM index consists of multiple *levels* $L_0, L_1, L_2, \ldots$, whose capacities increase geometrically. L_0 is small and resides entirely in main memory. New data enters L_0 first; as a level L_i overflows, its contents are *merged* with the next level L_{i+1}. Hence, except in L_0, index records are never updated in-place; instead, index is updated by merging adjacent levels, leading to higher-bandwidth sequential writes. This feature of LSM has proven indispensable to modern write-heavy workloads [18]. Many popular recent systems, such as *LevelDB* [10], *HBase* [2], *Cassandra* [1], *Druid* [23], and *bSLM* [18], use LSM or its variants.

Traditionally, LSM has worked well for magnetic hard drives, where sequential I/Os are much faster than random I/Os. Recently, flash-based solid-state drives (SSDs) have become commonplace and have successfully replaced magnetic hard drives in many enterprise as well as consumer settings. Although SSDs do not have a significant performance gap between sequential and random I/Os, LSM remains a good idea, because it eliminates in-place updates on SSDs, which are far more expensive than other types of accesses.

This paper considers the problem of optimizing LSM for SSDs. Beyond avoiding in-place updates, we focus on how to minimize the number of writes for LSM in general. This goal is motivated by the pronounced cost asymmetry between reads and writes on SSDs: compared with reads, writes are more expensive in terms of time and energy, and they also have a wear effect on SSDs, which decreases drive life.

The key observation is that we can achieve good reduction of writes by using a more intelligent *merge policy* for LSM. The original LSM and many of its implementations use what we call a *Full* policy, where *all* records of an overflowing level are merged into the next level. On the other hand, some of the recent LSM variants, including LevelDB, HBase, and Cassandra, have explored alternative policies that are *partial*, which carry out only a part of a full merge at a time. Their rationale for having shorter merges is to increase the index's availability for other operations. However, how partial policies affect the total write cost over time is not well-understood. One contribution of this paper is an in-depth study of the total write costs of various merge policies.

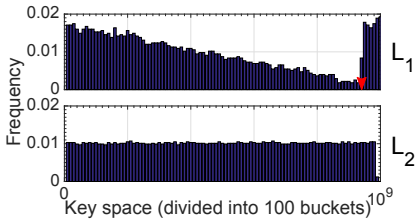


Fig. 1: Distribution of keys in the lowest two levels of a 3-level LSM-tree, at a random time instant while processing a workload of random inserts and deletes. For L_1, the arrow marks the beginning of the key space to be merged with L_2 next. Setup is the same as Figure 2a, with 20MB data.

To see why an in-depth study is warranted, we note that it is not even clear whether partial merges will result in fewer writes than full merges over time. Consider the following.

Example 1. *Consider a simple round-robin partial merge policy (which is roughly what LevelDB implements). Whenever a level L_i overflows, we merge a fraction, say $1/10$, of its data into the next level. The first merge will pick the first $1/10$ of L_i (in key order); each subsequent merge will pick the next $1/10$, starting from the largest key involved in the previous merge from this level (and wrapping around upon reaching the end of the key range). We call this policy* RR.

Suppose that the record keys within all levels follow a uniform distribution. Then, each merge from L_i is expected to hit about $1/10$ of L_{i+1}'s range, thereby writing about $1/10$ of the index blocks in L_{i+1}. For RR to merge one full L_i's worth of data into L_{i+1}, it would take 10 partial merges and a total number of block writes roughly equal to the size of L_{i+1}—which is no better than what a full merge would cost.

However, in practice, we observe something different when comparing RR and Full on a workload of uniform record inserts and deletes—RR in fact incurs fewer I/Os over time (see Section V for more results). But why?

It turns out that partial policies like RR may create different key distributions across levels—in a way that benefits partial merges—even for a uniform workload. Figure 1 shows a

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.121

snapshot of the key distributions within the lowest two levels of a 3-level LSM-tree. As expected, L_2, the lowest and largest level where most records reside, has a uniform key distribution matching that of the workload. However, L_1 has a skewed key distribution, which bottoms in the region last merged (to the left of the arrow in Figure 1) and peaks in the region to be merged next (to the right of the arrow). Because the region to be merged next is dense, $1/10$ of the L_1 records in this region would span only a small key range—narrower than $1/10$ of the entire key space. When these records get merged into L_2, they are expected to disturb less than $1/10$ of the blocks.

Interestingly, this behavior sustains itself. Merging records in the densest key range of L_1 empties that range. The most recently merged region becomes the sparsest while the least recently merged region is the densest. The cycle repeats.

It would be premature to conclude that RR always works well based on a single example. In the worst case, the workload could be skewed itself in a way such that new records arriving in a level disproportionately hit the most recently merged range, leaving the least recently merged range sparse, which causes a larger, instead of smaller, key range in the next level to be disturbed. How bad can RR be? Can its merges repeatedly disturb large portions of the next level? Does there exist a more robust partial merge policy that cannot be thwarted so easily?

Contributions In this paper, we show an easy-to-implement partial merge policy called *ChooseBest*, with guaranteed good performance in the worst case, and better performance in practice than Full and RR for a variety of workloads.

Furthermore, we show how to do even better than ChooseBest, by judiciously alternating it with Full. This improvement may seem counter-intuitive, as ChooseBest consistently outperforms Full in all workloads we considered. The key insight, however, is that the shape of LSM-tree affects performance and in general exhibits a *periodic* steady-state behavior during a workload. At certain times, it may make sense to invoke Full to bring the index into a state that would lower the cost of future partial merges in a period. The trade-off is rather intricate, but we show how to "learn" the right decisions automatically for this hybrid policy we call *Mixed*.

Besides studying the merge policies, we exploit another opportunity to reduce writes: when performing a merge, instead of always creating new index blocks, we consider the possibility of preserving existing blocks. We show how to modify the LSM data structure and the merge procedure to enable this optimization, which is effective when records are large (relative to block size).

II. Enabling Partial & Block-Preserving Merge

This section begins with a review of a basic LSM-tree. Then, we show how to modify LSM to enable partial merges in a way more general than previous work, and to implement the block-preserving optimization during merges. These modifications are easy to incorporate—we only relax the property of compact, sequential level storage and change the merge operation; LSM's general structure and other operations stay the same.

Beyond the basic LSM-tree described here, there are other techniques and optimizations for LSM. Many of them are orthogonal to and complement our contributions in this paper (e.g., our technical report [19] discusses how our techniques work with concurrency control and Bloom filters). Other merge-related techniques are not as general as our techniques or chose different trade-offs; we survey them in Section VI.

A. LSM Basics

Structure Let B denote the maximum number of records that can be stored in one block of storage. An LSM-tree consists of a sequence of *levels* $L_0, L_1, \ldots, L_{h-1}$ ordered from top to bottom. Each level stores a list of index records ordered by key. The *capacity* of L_i, denoted K_i, is the maximum size of L_i as measured in the number of blocks required to store its index records. The level capacities form an increasing geometric series: $K_i = K_0 \cdot \Gamma^i$, where $\Gamma > 1$ is a constant which we call the *order* of the LSM-tree. L_0, the top level, always resides completely in main memory and is implemented as an in-memory sorted index. The lower levels reside on secondary storage with a block access interface, e.g., hard drives or SSDs. Each lower level is implemented as a B+tree, whose data blocks (leaves) lie sequentially on consecutive blocks and store records as compactly as possible (leaving no gaps between records). In practice, the internal B+tree nodes of these levels are cached in main memory.

Operations The main idea behind LSM is to "log" data modifications first in the memory-resident L_0. If any level overflows, we merge all its records down to the lower level. Since the levels are sorted, merge is a one-pass operation.

In more detail, for an insert request, we simply insert a new index record into L_0. For a delete or update request with key k, we check whether k is in L_0. If yes, we execute the request in L_0; otherwise, we log the request with an index record.

When L_i overflows, we execute a *merge* between L_i and L_{i+1} in one pass, by scanning their contents simultaneously in order and writing out the result records compactly and sequentially. During the merge, if two index records refer to the same key, only their net effect (if any) will be produced; e.g., a delete record in L_i would "cancel out" a normal (insert) record in L_{i+1} with the same key. As the result records of L_{i+1} fill up its B+tree leaves, the upper levels of the B+tree is generated at the same time. When the merge ends, L_i becomes empty, while L_{i+1} becomes (likely) larger. If L_{i+1} in turn overflows, a merge with the next level is triggered, and so on. If there is no next level, the overflowing bottom level L_{h-1} becomes L_h, increasing the LSM-tree's number of levels by one.

A lookup for key k starts at L_0 and continues to lower levels until a match is found. Within each level, we follow the standard lookup procedure for the index on that level. If the match is an insert or update record, we return its corresponding payload. If the match is a delete record or if there is no match, we signal that k is not found.

B. Our Modifications

Relaxed Level Storage Requirements Recall that one of our goals is to enable partial merges in a general way. Many LSM-based indexes, such as LevelDB, divide each level into partitions with disjoint key ranges, and when the level overflows, merge only one partition into the next level. For maximum flexibility, however, we do not want to pre-partition the key space. Instead, we want to support dynamic selection of

the key range to merge. This requirement—as well as support for block-preserving merge, to be discussed next—makes it difficult to ensure that the index records are always stored sequentially with no gaps in between.

Therefore, with the exception of memory-resident L_0, we relax the requirement that storage of LSM index records in each level is sequential and compact. First, we allow the B+tree data blocks in a level to reside at non-contiguous physical block addresses. Because of SSDs' fast random block accesses, this relaxation does not incur a performance penalty for scans like it would on hard drives.

Second, we do not require B+tree leaves to be full. To guard against waste, however, we impose two constraints:

(Level-wise waste) Define the *waste factor* of a level to be the fraction of empty (unused) record slots in its data blocks. We require the waste factor of each level (with at least two data blocks) to be no more than a preset threshold $\varpi \leq 0.5$. We use $\varpi = 0.2$; i.e., each level is at least 80% full.

(Pairwise waste) We require any two consecutive data blocks to store strictly more than B records total.

We require the level-wise overall waste factor to be no more than ϖ, which can be much tighter than the standard 50% of B+tree. On the other hand, we allow an individual leaf to be nearly empty, so long as adjacent leaves are nearly full. In this regard, our pairwise waste constraint is looser than the standard B+tree capacity constraint, which applies to *every* block.

Note that the pairwise waste constraint is important for partial merges (or any operation involving a subsequence of blocks in a level) to avoid worse-case scenarios. Without this constraint, even if a level satisfies the level-wise waste constraint overall, there may exist a subsequence of nearly empty blocks that can lead to poor performance.

New Merge Operation Our new, flexible merge operation takes a list of data blocks from a level L_i—in general only a subsequence of its B+tree leaves—and merges the records therein into the next level L_{i+1}. The decision of which subsequence to merge is made by a *merge policy*, which we further discuss starting Section III.

Let X denote the subsequence of data blocks from L_i, and let $[k_{\min}, k_{\max}]$ denote X's key range. Let Y denote the list of data blocks in L_{i+1} whose key ranges overlap $[k_{\min}, k_{\max}]$. We merge the records in X and Y—with the block-preserving optimization further described below—and produce a list of data blocks Z. We then bulk-delete X from L_i and Y from L_{i+1}, and bulk-insert Z into L_{i+1}. Each of these bulk operations affects at most one key range per internal level of B+tree. Their costs are negligible compared with the cost of merging data blocks.

Additionally, our merge operation handles possible waste constraint violations. Upon removal of X from L_i ($i \geq 1$):

1) We check if the two blocks immediately before and after X meet the pairwise waste constraint. If not, we replace them with a block with all their contents.
2) We check if the resulting L_i meets the level-wise waste constraint. If not, we compact L_i in one pass.

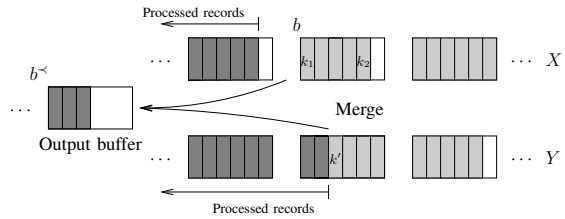
Upon modification (removal of Y and addition of Z) to L_{i+1}:

3) We check if the last block of Z and the following block in L_{i+1} meet the pairwise waste constraint. If not, we replace them with a block with all their contents. (The block-preserving merge procedure below will ensure that no violations occur among Z blocks and the preceding block.)
4) We check if the resulting L_{i+1} meets the level-wise waste constraint. If not, we compact L_{i+1} in one pass.

We will prove later that maintaining our waste constraints adds only a small overhead that does not affect the amortized costs of our merges asymptotically. In practice, we find compactions to be extremely rare in our experiments.

Block-Preserving Merge Given X and Y, each a list of blocks storing records in key order, we want to output all records in key order into a single list Z of blocks, such that we reuse blocks of X and Y (unmodified) in the output as much as possible while still meeting the waste constraints (for L_{i+1}, where the output is going).

We use a simple yet practically effective greedy algorithm. It proceeds like a standard merge, scanning X and Y in parallel a block at a time while buffering output records in a block of memory. The only difference is what we do when the next input record to be added to the output happens to start an input block. Suppose this input block b is from X (the case of Y is symmetric), and its key range is $[k_1, k_2]$. Let $b^{\prec}$ denote the block currently buffered for output (if it is not empty), and let k' denote the key value of the first record yet to be processed in Y. Pictorially, we have:



If $k_2 < k'$—in other words, the entirety of b can be "squeezed in" before the next record from Y—we consider the option of writing $b^{\prec}$, reusing b in the output, and then resuming with an empty output buffer and the next block from X. To avoid violating the waste constraints, we perform a *waste check* and preserve b only if all conditions below are met:

- The pairwise waste constraint must be met for $b^{\prec}$ and its preceding block (the last block written by this merge, or, if none is written yet, the block preceding Y in L_{i+1}), and for $b^{\prec}$ and b.
- Let m_{i+1} denote the running count of merges (including the current one) into L_{i+1} since its last compaction, and let w_{i+1} denote the cumulative net increase in the number of empty record slots due to these merges. The latter is updated during the current merge by adding the number of empty slots in the blocks written so far and subtracting those in the Y blocks already processed (completely or partially). Preserving b must not cause w_{i+1} to exceed $m_{i+1}\varpi\delta K_i B - B + 1$, where δ is the *merge rate*, or the fraction of L_i that we choose to merge into L_{i+1} each time.

The second condition above guarantees that at the end of the current merge, $w_{i+1} \leq m_{i+1}\varpi\delta K_i B$ (as in the worst case we

may be forced to write the last block of Z with only one record and hence $B-1$ empty slots). One can think of $\varpi\delta K_i B$ as the amount of "slack" that each merge is allowed to introduce (and any unused slack can be claimed by subsequent merges). Naturally, the bigger the maximum waste factor and the larger the merges, the more slack is allowed. This slack is needed to upper-bound the amortized cost of compactions; we defer its discussion (and the overhead of maintaining waste constraints in general) to Section III-D.

It should be clear that the *only* opportunity to preserve an input block is when its key range contains no record in the other input list. Our greedy algorithm considers all such opportunities, but may decide not to take an opportunity if the waste check fails. In practice, when the input is not adversarial, we found our simple greedy algorithm to be more than adequate. In the workloads we tried (Section V), it was rare for the waste check to fail, meaning that our algorithm in fact usually made the optimal decision.

III. Full vs. Partial Merge Policies

Our simple modifications to LSM in Section II allow us to formulate different merge strategies as *policies* that select the two lists of input blocks to participate in each merge. The basic LSM policy, which we call *Full*, always chooses all blocks in the two levels to merge. A policy that does not always choose to merge all blocks is called *partial*. We start by analyzing Full and the simple round-robin partial policy *RR* introduced in Section I. We then propose a provably better partial policy *ChooseBest*, and end with some experimental results that help motivate an even better policy to be presented in Section IV.

Some clarification is in order before we delve into analysis. First, we focus on modification (insert, update, and delete) workloads, and specifically, I/Os on the non-memory-resident levels (L_1 and below) to process modifications using merges. We do not consider lookups as their costs are largely independent of the techniques proposed in this paper. Second, when measuring cost, we count the number of writes (in terms of number of blocks). We do not count reads because there are roughly as many reads as writes in each merge (discounting the effect of consolidating records sharing the same key, which applies to all merge policies, and reads in L_0, which incur no I/O). Third, to simplify analysis, we ignore the effect of block preservation discussed in Section II-B, because this optimization benefits all merge policies. Also, for simplicity, our analysis will first ignore the issues of wasted space in data blocks and the additional overhead of maintaining our waste constraints; we discuss their effects in Section III-D.

For analysis, we break the cost down by level, considering the cost of merging *into* each L_i. An important observation—which complicates analysis and design here as well as later in Section IV—is that merges into the same level may not always cost the same, even if the workload has reached a steady state where the total number of records indexed remains a constant. Therefore, we need to consider *amortized* cost over time. To make the definition of amortization precise, we assume that the index size is in a steady state where the workload has a balanced mix of inserts, deletes, and updates such that the number of records stays constant over time. In this case, we can assume that the bottom level has a constant size, since the majority of records reside in the bottom level.

A. Full Policy

We focus on the cost of merging into a non-bottom level (the cost of a full merge into a constant-size bottom level is just its size in blocks). Consider $i \in [1, h-1)$. The cost of merging a full L_{i-1} into L_i is bounded by the total number of blocks in L_{i-1} and L_i. L_{i-1} is full at the time of the merge, so it has K_{i-1} blocks. L_i's size varies over time, however—it is empty immediately following a merge into L_{i+1}, but grows with each merge from L_{i-1} until it becomes full and triggers another merge into L_{i+1}. It is easy to see that the cost of merging into L_i is periodic, where each period, which we call a *full cycle* of L_i, sees L_i growing from empty to full. Following this line of reasoning, we derive the amortized cost of merges under Full by considering a full cycle.

Proposition 1. *Assume that a full merge into L_i increases the size of L_i by Δ blocks and its cost is the same as the resulting size of L_i. Under Full, the worst-case cost of a merge into L_i is K_i, and the amortized cost of merges into L_i is $\frac{1}{2}(K_i+\Delta)$ per merge.*

Corollary 1. *If each full merge into L_i increases the size of L_i by K_{i-1} blocks, then the amortized cost of merging into L_i under Full is $\frac{1}{2}(\Gamma+1)$ per block merged into L_i.*

For all proofs, please refer to our technical report [19].

B. RR Policy

As discussed in Section I, when L_{i-1} overflows, RR always selects a small constant fraction of L_{i-1} whose key range follows the one selected when L_{i-1} last overflowed. We call the constant fraction the *merge rate* and denote it by δ. We assume δK_0 to be a positive integer. In practice, δK_0 can easily be in hundreds. More precisely, RR remembers the largest key k in the fraction selected for the last merge; when L_{i-1} overflows again, RR selects the sequence of up to δK_{i-1} blocks starting with the first block whose smallest key is greater than k. If there is no such block left in L_{i-1}, RR selects the first δK_{i-1} blocks of L_{i-1}.

Under RR, in the steady state, all non-bottom levels are between $1-\delta$ and completely full between merges, and the bottom level remains roughly constant in size. The cost of merging the selected range of L_{i-1} into L_i, however, can vary depending on the key distributions. Although Example 1 illustrates a case when the partial merge touches less than δ fraction of L_i, in general the selected range of L_{i-1} may touch more, or nearly all of L_i in the worst case.

Despite this worst case for a single merge, we can still upper-bound the amortized cost of merges over time. The crux of the proof is to view RR merges as steadily progressing through the key space of L_{i-1}. The total write cost over one entire pass can be upper-bounded, while the number of merges (and hence the number of blocks merged into L_i) in each pass can be lower-bounded. In other words, if one merge is very costly, the others in the same pass will necessarily cost less.

Theorem 1. *Under RR, the worst-case cost of a merge into L_i is at least $K_i \cdot \min\{\delta B/\Gamma, 1-1/\Gamma\}$, and the amortized cost of merges into L_i is $(\frac{1}{1-\delta}+o(1))\Gamma+o(1)$ (for sufficiently large K_i) per block merged into L_i.*

The bounds above imply that a single RR merge can cost nearly as much as a full merge, and the amortized cost of

merges into L_i is not much more than Γ per block merged into L_i. While we can take some consolation in the amortized write cost of RR, it is still not appealing compared with Full from a theoretical perspective. RR has a higher amortized cost than Full in the worst case, and RR's worst-case cost for a single merge is similar to Full—the entire next level may still need to be rewritten. Our quest for a better policy hence continues.

C. ChooseBest Policy

It turns out that a simple idea of selecting the "best" range to merge works really well. RR may get unlucky and select a particularly bad range that overlaps many blocks in the next level, but by doing a bit more work, we can select the rangethat overlaps the fewest blocks. We call this policy ChooseBest.

More precisely, given merge rate δ, when L_{i-1} overflows, we look for the sequence X of δK_{i-1} consecutive blocks in L_{i-1} such that X's key range overlaps those of the fewest number of L_i blocks. We can find this sequence in a single, simultaneous scan of the metadata about L_{i-1} and L_i blocks.

There is no need to scan actual data to make the decision; instead, all information necessary is available from the internal nodes (those immediately above the data blocks) of the two B+trees for L_{i-1} and L_i, which in practice are cached in memory. Conceptually, the algorithm steps through two sorted lists of key ranges, ℓ for the L_{i-1} blocks and ℓ' for the L_i blocks. In turn, we examine each candidate subsequence q with δK_{i-1} entries in ℓ. Meanwhile, we maintain the subsequence q' in ℓ' enclosed by the first and last entries in ℓ' that overlap the span of q, and we remember the length (in entries) and location of the shortest q' seen so far. When moving to the next q, we update q' by scanning forward from its two endpoints. At the end of a single pass, we will have found the best q. The overhead of running this algorithm is in CPU processing, and is small as we shall see in Section V.

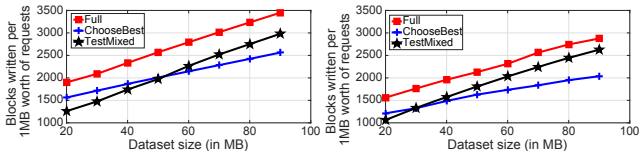
Theorem 2. *Under ChooseBest, each merge into L_i costs no more than $\delta(\frac{1}{\Gamma}+1)K_i$, which is $\Gamma+1$ per block merged.*

Compared with Full and RR, the bounds are much improved. Most importantly, the bounds here apply to *every* ChooseBest merge. Hence, unlike Full and RR, ChooseBest will never experience any particularly costly merges that rewrite the entire next level. Even though Full's worst-case amortized cost is still better by a half, we will see next in Sections III-E and V that ChooseBest in practice outperforms both RR and Full in experiments.

D. On Waste and Compaction

So far our analysis has ignored the effect of wasted space in data blocks and the overhead of maintaining waste constraints (Section II-B). We now show that they are small compared with merge costs, and do not change the overall amortized costs asymptotically. First, it is clear that enforcing the pairwise waste constraint in the new merge operation (Cases 1 and 3 in Section II-B) only incurs at most one extra write per merge. Second, the amortized compaction cost is also small compared with merge—$\frac{1}{1-\delta}+o(1)$ vs. $O(\Gamma)$—under the reasonable assumption that the index is growing or in a steady state:

Theorem 3. *The amortized cost of compactions for L_i is $\frac{1}{1-\delta}+o(1)$ (for sufficiently large K_i) per block merged into L_i, assuming: merges (from L_{i-1}) into L_i do not decrease its number of blocks; and the number of blocks in L_i does not decrease between two compactions if there is no merge from L_i to L_{i+1} during this period.*



(a) *Uniform* (b) *Normal*($\sigma = 0.5\%, \omega = 10{,}000$)

Fig. 2: Amortized cost comparison of Full, ChooseBest ($\delta = 1/20$), and TestMixed in steady state for different index sizes. Insert/delete ratio is 50/50; K_0 is 1MB (or 250 blocks); additional buffer cache is 1MB; see Section V for other default settings and explanation of σ and ω for *Normal*.

E. Some Experimental Results

Following the theoretical analysis above, we preview some experimental results illustrating the practical, common-case behaviors of full and partial policies. Here we focus on Full versus ChooseBest and only a small range of dataset sizes; additional results as well as comparison with RR and other policies can be found in Section V.

We consider two workloads. *Uniform* generates insert keys uniformly at random from the entire key space; *Normal* draws insert keys from a normal distribution, whose mean moves randomly from time to time. Both workloads generate deletes from existing records uniformly at random. Figure 2 compares the amortized costs of Full and ChooseBest across different dataset sizes (ignore TestMixed for now). For each dataset size, we run the workloads with equal insert and delete rates long enough to measure the steady-state amortized write costs. Across the dataset sizes shown, all indexes have 3 levels but differ in the sizes of their bottom levels.

We make several observations from Figure 2 (they also hold for additional and larger-scale experiments in Section V):

- ChooseBest consistently costs less than Full across all dataset sizes and the full range of bottom level sizes (relative to the maximum)—with 20MB data, the bottom level size is about 20% of its maximum, while with 100MB, the bottom level is about full.
- Costs generally rise linearly as the bottom level size grows, but the rate of growth is lower for ChooseBest than for Full.
- ChooseBest has a bigger advantage over Full in *Normal* than in *Uniform*. As we shall see in Section V, this advantage is more pronounced with more skewed workloads.

To sum up, earlier in Section III-C we have seen that ChooseBest is better than Full in capping the cost of each individual merge, which translates to lower worst-case latency. Here, we see that in practice, for all configurations we tested, ChooseBest also beats Full in amortized merge cost.

IV. Mixed Merge Policy

In this section, we propose a new merge policy called *Mixed* that further improves ChooseBest by judiciously mixing it with Full. However, it is far from obvious what could be gained by combining one policy with another that is apparently always inferior (as shown in Section III). Thus, before presenting Mixed, we start with a closer examination of merges across levels and over time, to identify opportunities for cost saving.

A. Motivation for Mixing Full and Partial

A closer look Let us drill down to one of the scenarios we experimented with in Figure 2a. Here, the dataset size is 20MB. In Figure 3, we show the cumulative merge costs (blocks written) by level over a period of time when the index is under *Uniform* in a steady state. For each of Full and ChooseBest, there are two cumulative cost plots, one for merges into L_1 and the other for merges into L_2 (the bottom level). Some explanations are in order:

- The L_2 cumulative cost plot for Full exhibits a step behavior. The plot jumps whenever a merge into L_2 is triggered, but remains flat otherwise. The jumps have the same height because L_2 is constant in size in the steady state.
- The L_1 plot for Full is also a step function, but richer in behavior than L_2 (clearly seen in the zoomed version):
  - Each jump corresponds to a merge into L_1, which occurs at least $\Gamma = 10$ times more frequently than merges into L_2 (in this trace we see more because record consolidation causes L_2 to grow slower).
  - Jumps have different heights. They start out small (when they are merging into small L_1) and grow bigger (as L_1 grows). Eventually, L_1 will become full and it will be merged into L_2; then the cycle repeats. The boundaries between cycles align with jumps in the L_2 plot, because they both correspond to times of merges into L_2.
- The two ChooseBest plots are much smoother than their Full counterparts. Although not visible from the figure, the L_2 (or L_1) plot jumps occur whenever a partial merge into L_2 (or L_1) is triggered, which happens at least $\delta^{-1} = 20$ times more frequently than Full. Furthermore, the slope of each plot remains constant (i.e., jumps have the same height) over time. The reason is that in the steady state, L_2 is constant in size, and L_1 is almost always full (with no fewer than $(1-\delta)K_1$ blocks) under ChooseBest; therefore, over time, merges into the same level cost roughly the same.

While ChooseBest beats Full in terms of the *overall* steady-state cost, we note that there are *moments* during which Full appears to have lower amortized costs—these moments occur when L_1 is small and the cost of merging into L_1 is low. As observed in Section III-E, both full and partial merges benefit from a smaller next level. Unfortunately, such opportunities never arise under ChooseBest, because in its steady state, L_1 almost always remains full.

But we can introduce these opportunities by (temporarily) switching from ChooseBest to Full. Suppose we do a full merge from L_1 to L_2. Then, L_1 becomes empty; subsequently, merges from L_0 into L_1—full or partial—become cheaper (until L_1 is filled up again). However, there is a trade-off: a full merge from L_1 to L_2 is costly in itself. According to Figure 2a, full merges actually have higher amortized costs than partial merges of ChooseBest. Luckily, in this particular experiment (Figure 3), L_2 is quite small, so the disadvantage of Full is not so bad. As seen in Figure 2a, merges into L_1 cost far more than merges into L_2, so it is a good deal to sacrifice some performance in L_2 to gain more improvement in L_1.

A test policy To test this idea, we experiment with a policy which we call *TestMixed* for the 3-level LSM-tree. This policy simply does ChooseBest for all merges from L_0 to L_1, and Full for all merges from L_1 to L_2. For the same configuration in Figure 3, TestMixed is better than both ChooseBest and Full—its cost is about 34% lower than Full and 20% lower than ChooseBest. Figure 4 basically augments Figure 3 with the same cumulative cost plots by level for TestMixed. As we can see in Figure 4, compared with ChooseBest and Full, TestMixed incurs substantially lower cost when merging into L_1—it beats ChooseBest by periodically making L_1 smaller and cheaper to merge into; it beats Full by using partial merges into L_1. On the other hand, for merges into L_2, TestMixed has practically the same cost as Full, which is higher than ChooseBest. Nonetheless, TestMixed is better overall.

We further compare TestMixed with others in all scenarios in Figure 2. Again, recall that for each dataset size, we run workload long enough to measure the steady-state amortized cost. According to the figures, TestMixed does not always beat ChooseBest. When L_2 is small, TestMixed's full merges into L_2 are a good deal. As L_2 grows in size, however, the disadvantge of full merges into L_2 becomes more pronounced, and eventually outweighs the savings they bring in L_1, which do not increase with the size of L_2. Therefore, a take-away point here is that the trade-off depends on the size of L_2 (which depends on the dataset size).

Another take-way point is that this trade-off depends also on the workload. Comparing Figures 2a and 2b, we see that under *Uniform*, TestMixed beats ChooseBest for a wider range of L_2 sizes than under *Normal*. As we will see later in Section V, in general, under less skewed workloads, there are more opportunities for gains by switching some merges to Full.

B. Thresholds-Based Mixed Merge

Taking the idea in Section IV-A further, we now propose the Mixed merge policy. While TestMixed always does full merges into the bottom level, Mixed judiciously invokes full merges at opportune times across all levels. Roughly speaking, when merging to a level L_i, Mixed chooses a full merge if the current size of L_i is below a prescribed threshold that is specific to the level and the workload.

More precisely, suppose that the index has reached a steady state, with h levels. Let $S(L_i) \leq K_i$ denote the (current) size of L_i in the number of data blocks. In addition to the merge rate δ, Mixed is parameterized by $(\tau_2, \tau_3, \ldots, \tau_{h-2}, \beta)$, where each τ_i is a fraction in $[0, 1]$ called the *threshold* of L_i ($2 \leq i \leq h-2$), and β is a Boolean value called the *decision* for L_{h-1}, the bottom level. Mixed operates as follows:

- A merge from L_0 to L_1 is always a partial merge with rate δ, determined in the same way as ChooseBest.
- A merge from L_{i-1} to L_i (for $2 \leq i \leq h-2$) is a full merge if $S(L_i) < \tau_i K_i$; otherwise, it is a partial merge with rate δ, determined in the same way as ChooseBest.
- A merge from L_{h-2} to L_{h-1} will be a full merge if β is `true`; otherwise, it is a partial merge with rate δ, determined in the same way as ChooseBest.

The first rule concerning the top level is easy to understand. Unlike a full merge from a lower level, which empties that level to make future merges into it cheaper, a full merge from L_0 provides no such benefit because there is no merge into L_0. Instead, L_0 is in memory, and we incur no I/O when operating

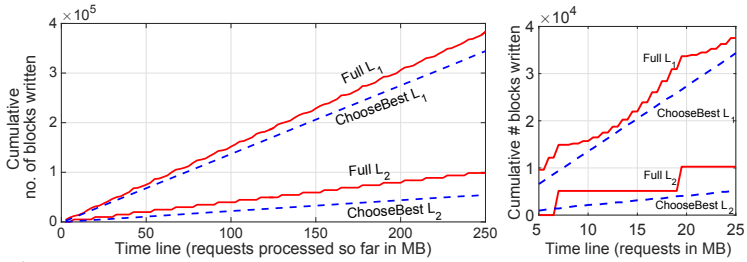


Fig. 3: Cumulative costs of Full and ChooseBest ($\delta = 1/20$) by level over time for an index with dataset size of 20MB in a steady state under *Uniform*. Other settings remain the same as in Figure 2a. On the left, we plot the cumulative costs (starting with 0) every 2.5MB worth of requests in the workload. On the right, we show a zoomed version over a representative time period, where the costs are plotted every 0.5MB worth of requests.

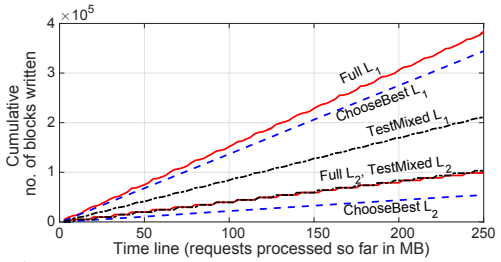


Fig. 4: Cumulative costs by level over time for TestMixed in comparison with Full and ChooseBest. All settings are the same as Figure 3.

on it. Furthermore, leaving records in L_0 increases the chance of record consolidation, which reduces merge frequency.

The last two rules can be seen as generalization of the idea in Section IV-A. The last rule, concerning the bottom level L_{h-1}, is just a simplification—since the $S(L_{h-1})$ is fixed in the steady state, we only need to know whether to use a full or partial merge into L_{h-1} given $S(L_{h-1})$. The second rule, however, requires a threshold to operate because $S(L_i)$ $(1 \leq i \leq h-2)$ may vary periodically during the steady state. For example, suppose merges from L_{h-2} to L_{h-1} are full. Then, $S(L_{h-2})$ periodically becomes 0 and grows back to K_{h-2}. When $S(L_{h-2}) < \tau_{h-2}K_{h-2}$, we perform full merges from L_{h-3} to L_{h-2}; once $S(L_{h-2})$ reaches $\tau_{h-2}K_{h-2}$, we go back to performing partial merges from L_{h-3} to L_{h-2}, until L_{h-2} becomes full and then empty again.

C. Learning Parameters for Mixed

For many applications, we expect the index to reach a steady state, where the workload characteristics remain stable and the dataset size does not change or changes very slowly (relative to the amount of data already indexed). For these scenarios, we can "learn" the appropriate settings of the Mixed policy parameters through experimentation and observation. This learning process incurs some cost, but it is small compared with the benefit over time, which continues until the workload or data characteristics change significantly.

The remainder of this section details our process for learning the Mixed parameters. We focus on two challenges. The first challenge is that optimal parameter settings across levels can be interrelated—one cannot set the optimal parameter for a given level independently of all others. Trying all combinations of settings across levels would incur cost exponential in h, the height of the tree. We analyze the dependencies among the parameters, and show how to learn them in a particular order with cost linear in h and without sacrificing optimality.

The second challenge lies in speeding up learning of the optimal threshold for an internal level. Let $\mathcal{D}_\tau \subset [0,1]$ denote the discretized domain of possible threshold settings. Testing all possible settings has cost linear in $|\mathcal{D}_\tau|$. We show how to exploit the properties of the cost function to find an optimal threshold with $O(\log|\mathcal{D}_\tau|)$ cost.

Top-down learning of optimal parameters Although Mixed operates on each level independently according to that level's parameter and current size, we cannot cost merges into a level independently of others. For example, cost of merging into L_i depends not only on τ_i (which controls merges from L_{i-1} to L_i), but also on how we merge L_i into L_{i+1}. If we use a full merge into L_{i+1}, L_i will periodically grow from empty to full; we can measure the overall amortized cost of merges into L_i—some full and some partial—under τ_i over this period. However, if we choose to do partial merges into L_{i+1}, L_i will remain nearly full at all times; in this case, merges into L_i will all be partial, so the overall amortized cost of merges into L_i will be very different.

Despite the intricacies, it turns out that the optimal parameter setting for a level does not depend on the settings of lower levels. This property allows us to set the parameters one level at a time in a top-down fashion, starting from τ_2 and ending with β. In the following, we first define the sequence of settings $\tau_2^\star, \ldots, \tau_{h-2}^\star, \beta^\star$ inductively, and then show that together they are optimal for Mixed.

Definition 1. *Given a workload, an index with h levels, and Mixed parameters $\tau_2, \ldots, \tau_{h-2}, \beta$, we define $\mathcal{C}(\tau_2, \ldots, \tau_{h-2}, \beta)$ as*

$$\frac{\textit{total cost incurred by merges into all levels}}{\textit{total number of records merged into } L_1}$$

measured in a steady state of the index operating under Mixed with the given parameter settings.

Given a prefix of the parameter settings $\tau_2, \ldots, \tau_i$ $(2 \leq i \leq h-2)$, we define $\mathcal{C}(\tau_2, \ldots, \tau_i)$ as

$$\frac{\textit{total cost incurred by merges into } L_1, \ldots, L_i}{\textit{total number of records merged into } L_1}$$

measured in a steady state of the index operating as follows:

- *for merges into L_i and above, run Mixed with $\tau_2, \ldots, \tau_i$;*
- *for merges from L_i to L_{i+1}, run Full;*
- *for merges into levels below L_{i+1}, run ChooseBest.*

Note that when figuring $\mathcal{C}(\tau_2, \ldots, \tau_i)$, what happens to merges into levels below L_{i+1} does not matter. We assume ChooseBest for these levels, to make the definition for the steady state of the index more precise.

Definition 2. *For an index with h levels, we define $\tau_2^\star, \ldots, \tau_{h-2}^\star, \beta^\star$:*

- $\tau_2^\star \leftarrow \arg\min_{\tau_2} \mathcal{C}(\tau_2)$;
- *for* $3 \leq i \leq h-2$, $\tau_i^\star \leftarrow \arg\min_{\tau_i} \mathcal{C}(\tau_2^\star, \ldots, \tau_{i-1}^\star, \tau_i)$;
- $\beta^\star \leftarrow \arg\min_{\beta} \mathcal{C}(\tau_2^\star, \ldots, \tau_{h-2}^\star, \beta)$.

Theorem 4. *$(\tau_2^\star, \ldots, \tau_{h-2}^\star, \beta^\star)$ is optimal, i.e.:*
$(\tau_2^\star, \ldots, \tau_{h-2}^\star, \beta^\star) = \arg\min_{\tau_2, \ldots, \tau_{h-2}, \beta} \mathcal{C}(\tau_2, \ldots, \tau_{h-2}, \beta)$.

Theorem 4 leads to a procedure for finding optimal parameter settings top-down. For an internal level L_i, we find the best setting $\tau_i^\star$ for τ_i—using a procedure to be detailed next—under the assumption that we follow the optimal threshold settings found for the upper levels and always perform full merges from L_i to L_{i+1}. Of course, we may later find out that full merges into L_{i+1} are suboptimal, but we show (again, see appendix for all proofs) that the assumption we use to obtain $\tau_i^\star$ does not affect overall optimality.

Learning threshold for a given level Given the optimal threshold settings $\tau_2^\star, \ldots, \tau_{i-1}^\star$ at upper levels, we want to find the setting of τ_i that minimizes $\mathcal{C}(\tau_2^\star, \ldots, \tau_{i-1}^\star, \tau_i)$. In ensuing discussion, the context is clear, so we omit the constant arguments $\tau_2^\star, \ldots, \tau_{i-1}^\star$ and the subscript i from τ_i, and write $\mathcal{C}(\tau)$ for brevity. In general, the problem is hard because the optimal solution depends on workload and data characteristics. We thus empirically measure $\mathcal{C}(\tau)$ for different settings of τ and pick the setting that minimizes $\mathcal{C}(\tau)$. Each measurement requires observing a cycle in the workload—we begin the cycle with an empty L_i as the result of a full merge into L_{i+1}, apply Mixed with $\tau_2^\star, \ldots, \tau_{i-1}^\star$ and the τ setting of interest, and end the cycle when L_i becomes full again (which would trigger a full merge into L_{i+1} and start a new cycle).

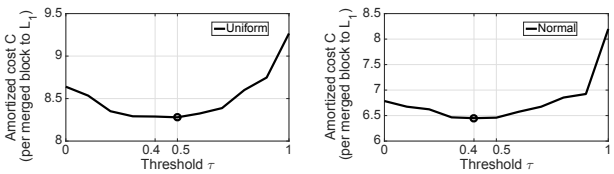
Overall, each measurement amounts to inserting K_i blocks of data into L_i, which may be just a small fraction of the total data indexed (since L_i is an internal level). Nonetheless, the total cost would add up with if we naively try all $|\mathcal{D}_\tau|$ possible settings of τ_i at each level. We would like to do better.

The key insight is that $\mathcal{C}(\tau)$ is not an arbitrary function. Under reasonable assumptions—namely, merges (full or partial with fixed rate) into an internal level have cost linear in its current size and increase this size steadily—we can show that $-\mathcal{C}(\tau)$ is unimodal. That is, $\mathcal{C}(\tau)$ is monotonically decreasing when $\tau < \tau^\star$ and monotonically increasing when $\tau > \tau^\star$, where $\tau^\star$ is the optimal setting.

Theorem 5. *Assume that: each full merge into L_i increases $S(L_i)$ by Δ_f blocks and costs $a_f(S(L_i) + \Delta_f) + b_f$; each partial merge of rate δ into L_i, selected by ChooseBest, increases $S(L_i)$ by Δ_p blocks and costs $a_p(S(L_i) + \Delta_p) + b_p$. Then $\mathcal{C}(\tau)$ is a quadratic function of τ, with second order derivative $c\left(\frac{a_f}{\Delta_f} - \frac{a_p}{\Delta_p}\right)$, where c is a positive constant.*

Note that the two terms in the second order derivative above, $\frac{a_f}{\Delta_f}$ and $\frac{a_p}{\Delta_p}$, roughly correspond to amortized costs of full merges and partial merges under ChooseBest, respectively. In practice, as we have observed in Section III-E, $\frac{a_f}{\Delta_f} > \frac{a_p}{\Delta_p}$, so $\mathcal{C}(\tau)$ is concave up, with a unique local minimum. It follows that $-\mathcal{C}(\tau)$ is unimodal. With a unimodal $-\mathcal{C}(\tau)$, we can use the standard *golden section search* [17] to find $\tau^\star$ with $O(\log|\mathcal{D}_\tau|)$ measurements. See [19] for details. In practice, $\mathcal{D}_\tau$ is not big—we consider τ settings that are multiples of 10%. Hence, even a linear search provides adequate performance: start from $\tau = 0$ and stop when $\mathcal{C}(\tau)$ starts to increase.

As an example, let us look at a pair of real $\mathcal{C}(\cdot)$ curves—Figure 5 plots $\mathcal{C}(\tau_2)$ measured under *Uniform* and *Normal* using a setup similar to Figure 2. We see that both $\mathcal{C}(\tau_2)$'s are indeed roughly quadratic in shape, each with one unique local minimum. Comparing the *Uniform* and *Normal* figures, we see that the optimal threshold setting is smaller under *Normal*—i.e., Mixed should switch to ChooseBest sooner—because partial merges benefit more from a skewed workload, as we have seen earlier in Figure 2.



(a) *Uniform* (b) *Normal*($\sigma = 0.5\%, \omega = 10{,}000$)

Fig. 5: $\mathcal{C}(\tau_2)$ measured in τ_2 increments of 10%. All settings remain the same as in Figure 2, except here we use a larger index with four levels (there is no need to learn τ_2 for the 3-level tree in Figure 2 as β would suffice).

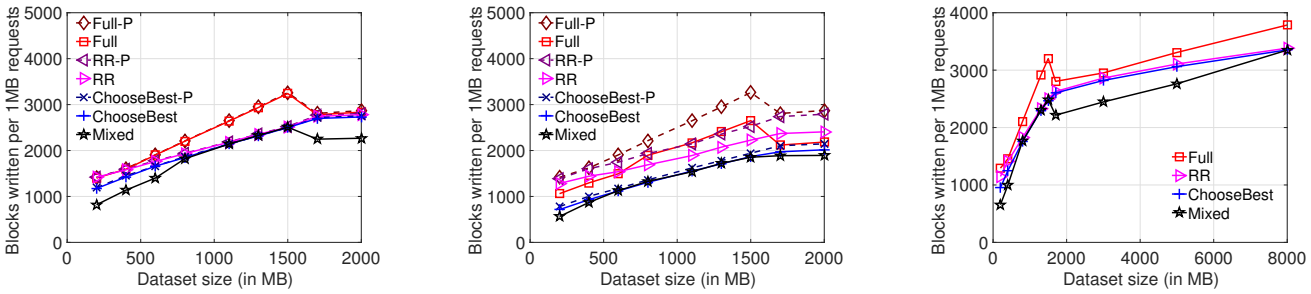
Putting it together Overall, given a workload, we learn the optimal Mixed parameter settings level by level from top to bottom, invoking the method above for finding the threshold for each level. The complete algorithm is given in [19].

V. Experimental Evaluation

Merge policies compared We compare seven merge policies. The four policies of primary interest are Full, RR, ChooseBest, and Mixed as discussed earlier. The other three policies, Full-P, RR-P, and ChooseBest-P (all with "-P" suffix), are variants of Full, RR, and ChooseBest without the block-preserving merge feature (Section II-B). In other words, the "-P" policies use standard merges that simply write out result sequentially and do not look for opportunities to reuse existing data blocks. We are interested in such policies not only because we want to understand the benefit of block-preserving merges, but also because they approximate the behaviors of some of the well-known systems:

- Full-P is the basic LSM merge policy. Newer, better variants such as bLSM [18] have been proposed, but the improvements focus on aspects (e.g., concurrency control, merge scheduling, Bloom filters) orthogonal to our concern. Without these improvements, bLSM is basically Full-P.
- RR-P is a reasonable approximation of the merge policy of LevelDB [10], again without orthogonal improvements in concurrency control and burst handling.
- ChooseBest-P is a stronger version of the merge policy of *HyperLevelDB* [3]. HyperLevelDB pre-partitions the key space into ranges, and picks the best range to merge only among this set. ChooseBest-P examines all possible ranges and can find potentially cheaper options. Thus, we can regard the cost of ChooseBest-P as a lower bound on that of HyperLevelDB (again without orthogonal improvements).

Workloads We use three synthetic workloads, *Uniform*, *Normal*, and *TPC*, with insert and delete requests. For *Uniform* and *Normal*, our workload generator allows the specification of the key domain, key and record sizes, as well as the ratio between insert and delete requests. The generator picks the type (insert or delete) for each request according to the specified ratio. For *Uniform*, we draw insert keys uniformly at random from keys that are not currently indexed; we draw delete keys uniformly at random from keys that are currently indexed. For *Normal*, we draw insert keys from a normal distribution (truncated to the key space) whose mean periodically moves to a location selected uniformly at random in the key space.



(a) *Uniform* (b) *Normal*($\sigma = 0.5\%, \omega = 10{,}000$) (c) *TPC*

Fig. 6: Comparison of steady-state write costs in an index across various dataset sizes. Workloads have a $50/50$ insert/delete ratio.

More precisely, *Normal* is parameterized by (σ, ω), where σ is the standard deviation of the normal distribution divided by the length of the key domain, and ω is the number of inserts to generate before we move the distribution mean. Deletes are generated in the same way as *Uniform*.

TPC is loosely based on the TPC-C benchmark [4]. We simulate modifications to `NEW_ORDER` only. Each insert transaction picks a warehouse, a district, and a customer at random. Each delete transaction picks a warehouse and a district at random, and removes the 10 oldest orders. We deviate from the standard by coding the primary key of `NEW_ORDER` using a bit string of the given key size, and allowing each record to carry an extra payload of given size.

Implementation and setup All algorithms are implemented in C++. We run our experiments on virtual machines leased from Amazon EC2. All machines are of the type `m3.xlarge` running the Ubuntu Server 14.04 LTS 64-bit operating system. Each machine comes with 2×40GB of local SSD storage that we use for workload execution. To access the SSD storage, we use an `ext4` file system without journaling, but with direct I/O using `O_DIRECT` and `O_SYNC` flags; on-disk caching is left on. The block size is 4KB.

Unless otherwise noted, keys are 4-byte unsigned integers in $[0, 10^9]$, and payloads (separate from keys) are 100 bytes each. The order of the LSM-tree is $\Gamma = 10$ (same as the default LevelDB setting) and the top level capacity is $K_0 = 4000$ blocks, or 16MB. The maximum waste factor $\varpi = 0.2$. See [19] for more details on the B+trees used by lower levels. For policies using partial merges, the merge rate $\delta = 0.07$ by default, and 0.05 for Section V-A because of larger indexes. Besides reserving 16MB for the top level, we also use an LRU buffer cache. Its size is 16MB by default, but 100MB for Section V-A because of larger indexes. For policies involving partial merges, this cache is large enough to pin the internal B+tree nodes (non-data blocks) of lower LSM-tree levels in memory for our workloads. For Full(-P), we do not pin these nodes, as Full(-P) benefits more from caching L_1 data blocks.

Metrics of comparison Our primary measure of performance is the number of writes of data blocks on the SSD.[1] We instrument our code to keep track of this number precisely, independent of the platform running experiments. We care about writes because they are particularly expensive on SSDs, and as mentioned in Section III, write cost is a good surrogate for the overall I/O cost of merges, because the number of reads is roughly the same as writes (discounting reads in L_0, which incur no I/O, and the effect of record consolidation, which applies to all merge policies).

We use request processing time as a secondary measure of performance. Processing time is heavily dependent on implementation and platform, and less reliable than write counts as our implementation still has room to improve. Nonetheless, we hope to get a general sense of the processing overhead.

Because of space constraint, this section presents only a subset of our results; see [19] for additional experiments. Notably, these experiments include workloads with lookups and range queries, and show that our techniques introduce little overhead in terms of query performance even when compared with Full-P, which has the most compact storage possible.

A. Steady-State Workloads

First, we compare the merge policies using an index in a steady state running a workload with a $50/50$ insert/delete ratio, where the dataset size is stable over time. We experiment with workloads *Uniform*, *Normal*($\sigma = 0.5\%, \omega = 10{,}000$), and *TPC*. We also study how the policies scale by varying the steady-state dataset size. For *Uniform* and *Normal*, the size varies from 200MB to 2000MB; for *TPC*, it varies from 200MB to 8GB. Between 200MB and 1600MB the index has three levels; beyond 1600MB the index has four levels. Before measuring the steady-state performance for each configuration, we first run the workload with inserts only until the dataset reaches the desired size; we then switch to a $50/50$ insert/delete ratio and wait until at least one full second-to-last level worth of data has been merged down to the bottom level. Furthermore, we wait until Mixed finishes learning parameters and is operating with optimal parameter settings.

Figure 6 compares the merge policies in terms of writes per 1MB of requests. The high-level observation is that, throughout the range of dataset sizes tested, Mixed either has the lowest number of writes among all policies or ties with other policies based on partial merges. We zoom in on some details below.

Writes under *Uniform* Under *Uniform* (Figure 6a), when the dataset size is 200MB, Mixed is 30% cheaper than ChooseBest, but its lead over ChooseBest narrows as the

[1] We do not count writes to the internal B+tree nodes. As discussed earlier, for policies with partial merges, they are always cached in our experiments. For Full(-P), there is no guarantee that they are cached, but we discount them for fair comparison. In reality, the number of such writes is anyway negligible compared with writes to data blocks. In fact, we can also design LSM-trees such that internal B+tree nodes can be reconstructed from data blocks and hence need not be persisted.

dataset size increases. At the beginning of the range, since the bottom level L_2 is small, Mixed uses full merges into L_2 to gain advantage over ChooseBest. From around 800MB to 1500MB, the two policies have same performance, because L_2 is now large and Mixed resorts back to ChooseBest for merges into L_2. From 1700MB to 2000MB, the index has a new bottom level L_3, whose size is still small (relative to its full capacity), so Mixed switches to full merges into L_3 again.

Compared with Full, Mixed is always better—by about 42% at 200MB, and by 23% at around 1500MB. Even in ranges where Mixed uses full merges into the bottom level, its merge policies in upper levels allow it to outperform Full. An interesting phenomenon occurs when the index grows to four levels—we see a drop in cost for both Full and Mixed, which is counter-intuitive as a larger index typically implies higher maintenance cost. It turns out that when the last level L_3 is relatively empty, full merges into this level are very cost-effective in reducing the merge costs at higher levels. In contrast, prior to having L_3, L_2 would remain close to full at all times, which are expensive to merge into (for both full and partial merges). This phenomenon raises the question of whether we can increase the number of levels strategically to gain performance in certain situations; see [19] for more.

One may wonder what happens when the dataset size grows beyond 2000MB—we defer that discussion to when we study the *TPC* workload. The short answer is that, when the index stays at four levels, write costs follow the same trend as that observed when the index stays at three levels. The phenomenon described above occurs only when the index gains a new level.

As for RR, it starts out worse than ChooseBest but quickly becomes close to ChooseBest in Figure 6a. The reason is that under *Uniform*, as explained in Example 1, RR's round-robin scheme happens to pick the best (densest) region to merge. However, we will see shortly that it is not always lucky.

The last point to note in Figure 6a is that the "-P" policies have nearly identical performance as their counterparts with the block-preserving merge feature. The reason is that the record size in this experiment (4-byte key plus 100-byte payload) is fairly small. With more records in data block, the chance that an entire block of records can be squeezed into the gap between two consecutive records in an adjacent level is smaller. Therefore, there are few opportunities to preserve blocks. In Section V-C, however, we will see situations where block-preserving merges produce more gains.

Writes under *Normal* We now turn to *Normal* (Figure 6b), which is skewed. Overall, Mixed remains the best among all policies, and the shapes and relative ordering of the curves are largely consistent with the *Uniform* case. We highlight a few notable differences.

RR(-P) no longer works as well as ChooseBest(-P)—now there is discernible, consistent gap. Even Full outperforms RR(-P) sometimes. Unlike ChooseBest, which finds the densest regions to merge, RR works well only if the least recent merged region happens to be dense—a property that holds for *Uniform* but not in general when the workload itself is skewed.

All policies with block preservation outperform their "-P" counterparts, which was not the case for *Uniform*. The reason is that the skewed *Normal* leads to regions with higher key concentration, increasing the chance for block preservation. We investigate the effect of skewness further in Section V-B.

The leads of ChooseBest(-P) and Mixed over Full-P are wider than under *Uniform*, thanks to the workload skew. Also, Mixed has the same performance as ChooseBest(-P) over a wider range of dataset sizes than under *Uniform*. The reason is that as ChooseBest becomes more effective on a skewed workload, switching to full merges is less appealing, so Mixed switches back to partial at lower thresholds. Nonetheless, switching to full still brings considerable gains when the bottom level is small compared with its maximum capacity, as evidenced by the gaps between Mixed and ChooseBest(-P) towards the two ends of the dataset size range tested.

Writes under *TPC* Finally, we turn to *TPC* (Figure 6c), for which we have pushed the dataset size to 8GB. In the beginning part of the range (up to 1500MB), the plots are very similar to those under *Uniform* in Figure 6a. This similarity should not be surprising, because *TPC* generates inserts and deletes uniformly over warehouses and districts; given the warehouse and district, the order ids in inserts and deletes are essentially sequential. Overall, like *Uniform*, *TPC* is skewless, so RR works almost as well as ChooseBest.

The range of dataset sizes between 1700MB and 8GB, where the index stays at four levels, gives a good picture of the performance trend beyond what was illustrated in Figures 6a. Here, following the temporary decrease in write costs as the index grows from three to four levels, the write costs revert back to an increasing trend, and the relative ordering among polices is largely consistent with that observed while the index had three levels. Again, Mixed has a big lead when the bottom level is relatively empty, but beyond a threshold (7500MB) it resorts back to ChooseBest. Finally, note that across policies, write costs increase at a slower rate when the index has more levels—a desirable property of the LSM-tree which implies complexity sublinear in the dataset size.

Running time under *Normal* Figure 7 compares the merge policies in terms of request processing time under *Normal*. Results for *Uniform* and *TPC* are similar; see [19] for details. Note that as the index grows in size (and hence height), merges (and other index operations in general) become more expensive, so we see an increasing trend in this figure.

As mentioned earlier, running time measurements are highly dependent on the platform and implementation, and our prototype implementation has not been highly tuned for CPU. In our experiments, the overhead in intelligently selecting key ranges to merge constitutes as low as 2% (for the largest workload of 8GB, under *TPC*) and as high as 16% (for one of the smallest workloads of 200MB) of the total time; this percentage decreases as the workload becomes bigger, so the trend is favorable to practical dataset sizes. Despite the overhead, from Figure 7, we see that the ranking of policies by running time is largely consistent with the ranking by writes, and Mixed remains a winner—only sometimes losing to ChooseBest by a small margin.[2]

[2]Note that Figure 7 plots only one series of experiments on one machine. We repeated the experiments 5 more times on different machines of the same configuration on Amazon EC2. While we observed variations in running times across machines (despite perfectly repeatable I/O counts), the overall trends are similar and the conclusion about Mixed remains valid; see [19] for details.

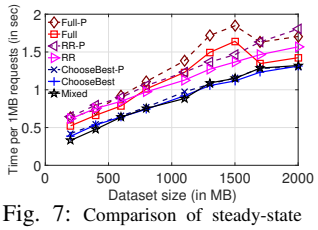


Fig. 7: Comparison of steady-state request processing time for *Normal*; same settings as Figure 6b.

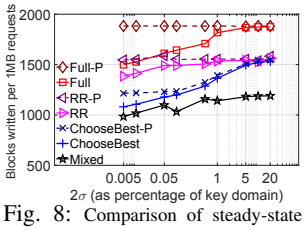


Fig. 8: Comparison of steady-state write costs for a 300MB dataset under *Normal* as σ varies.

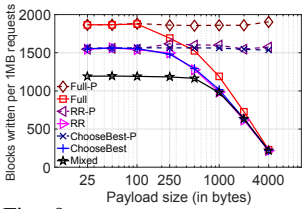


Fig. 9: Comparison of steady-state write costs for a 300MB dataset with given record payload sizes. *Uniform* with a $50/50$ insert/delete ratio.

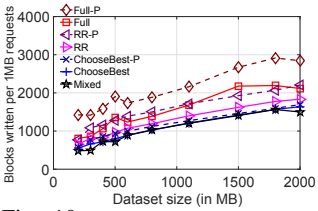


Fig. 10: Comparison of amortized write costs over time as index grows. Insert-only *Normal*($\sigma = 0.5\%, \omega = 10{,}000$).

B. Effect of Key Distribution

We now turn to how skewness in the workload affects merge policies. We experiment with *Normal* workloads with $\omega = 10{,}000$ but different σ, such that the percentage of the key space covered by one standard deviation around the mean ranges from 0.005% to 20%. For each σ setting, we compare the merge policies using an index with 300MB data in a steady state running a *Normal* workload with given σ and $50/50$ insert/delete ratio. Figure 8 compares the merge policies in terms of amortized write costs.

Viewing the figure *from right to left*, we see that as σ decreases and the workload becomes more skewed, ChooseBest(-P) outperforms RR(-P) more, because it is easier to find dense key ranges that cost less to merge. As σ decreases, we also see that policies with block-preserving merges outperform "-P" policies more. The reason is that a tighter σ leads to a higher concentration of keys in a region to be merged down, which in turn leads to a higher chance for block preservation.

Even when σ is large—where we would expect lackluster performance from both block-preserving merges and intelligent selection of key ranges to merge—Mixed remains extremely effective. It maintains a comfortable lead over the other merge policies throughout the range of σ tested.

C. Effect of Payload Size

We now study how record size affects merge policies. Record size—which determines the number of records in a block—mainly influences the effectiveness of block preservation. Intuitively, the bigger the records, the fewer of them fit in a block, and we have a higher chance for all keys in a block to fit in a gap between some two consecutive keys in the other level to be merged. In the extreme case, when each block holds only one record, we can preserve all blocks during a merge.

Figure 9 shows the steady-state amortized write cost for each policy, as the payload size varies from 25 to 4000 bytes. Note the logarithmic scale for the horizontal axis. With 25-byte payload, a block can store 136 records; with 4000-byte payload, a block can store only one record. We fix the dataset size to be 300MB; otherwise the workload settings remain identical to those in Section V-A. From Figure 9, we see the clear trend that as the payload size increases, policies with block-preserving merges perform progressively better than their counterparts without it (i.e., the "-P" policies), confirming our intuition. The performance of the "-P" policies remains flat across payload sizes as they simply do not benefit from higher chances for block preservation.

The second observation to make is how policies with block-preserving merges compare among themselves. From Figure 9, we see that under *Uniform*, at small payload sizes, Mixed holds a significant lead over others—about 33% better than Full and about 23% better than RR and ChooseBest. As the payload size increases, the four policies all improve, and the savings achieved by block preservation start to overwhelm their differences. In the extreme case of 4000-byte payloads, all blocks can be preserved during merges, so the four policies have the same write costs. Similar results can be seen under *Normal*; see [19] for details.

D. Insert-Only Workloads

We now compare the merge policies in the setting where the index accepts a stream of inserts. We use *Normal*($\sigma = 0.5\%, \omega = 10{,}000$) again, but now with inserts only (see [19] for results on *Uniform*). For each policy, we start with an empty index and let it grow while tracking its performance over time. Mixed uses the same thresholds learned for steady-state experiments. Figure 10 shows the amortized write cost for each merge policy over the course of growing the index. The horizontal axis shows the size of the dataset indexed so far, which serves as the time line. Unlike the steady-state experiments shown in Figures 6, there are no steady states here. Each point is the average measured *since the beginning* of workload. Thus, compared with Figure 6, we expect write cost to be generally lower here, because the accounting reflects earlier periods when the index was smaller.

The conclusions on insert-only workloads are similar to those on steady-steady workloads—Mixed is the overall winner, and Full performs the worst. One notable difference is that policies with the block-preserving merge feature perform much better than their counterparts without it (i.e., the "-P" policies). This advantage is much less visible under *Uniform* [19], nor did we see it in the steady-state experiments even under *Normal* (Figure 6b). One reason, as observed in Section V-B, is that more skewed workloads lead to a higher concentration of keys and hence higher chances for block preservation. Less obviously, insert-only *Normal* leads to higher key concentration than *Normal* with a $50/50$ insert/delete ratio (used by the steady-state experiments), because delete keys are generated uniformly. Thus, the benefit of block-preserving merges is more pronounced under insert-only workloads.

VI. Related Work

LSM-tree [16] is a seminal work proposing a log-structured organization for database indexes. Basic LSM-tree always uses full merges. *Partitioned Exponential File* [12] improves LSM by partitioning the key space and indexing each partition with a separate LSM-tree, which can be well suited for skewed workloads. bLSM [18] improves LSM by carefully scheduling

merges across different levels to reduce latency and using Bloom filters to speed up lookups. These schemes above essentially use Full-P for merges.

Stepped-Merge [11], developed independently from LSM-tree, allows each level to accumulate multiple runs before merging them fully into a run at the lower level. Cassandra [1] and HBase [2] offer merge options that are basically Stepped-Merge. In reducing merge costs, however, Stepped-Merge sacrifices lookups, which must now examine multiple runs per level. In contrast, partial merges, popular in recent implementations of LSM, are able to reduce merge cost without penalizing lookups; we follow the same philosophy.

There are notable uses of partial merges in popular systems. LevelDB [10] partitions each level into multiple *SSTables* (of 2MB in size by default) with non-overlapping key ranges, and merges a single SSTable from one level to the next in a round-robin fashion. Thus, the policy is essentially RR-P. HyperLevelDB [3] is a fork of the LevelDB code with a main difference being its merge policy, which selects the "best" SSTable to merge to the next level. As discussed in Section V, this policy amounts to a weaker version of our ChooseBest-P, where the candidate key ranges are limited to those of the SSTables. The newest merge option offered by Cassandra, called *leveled*, is similar to LevelDB. The main motivation for partial merges in these systems is to reduce merge latency (compared with Full). For the first time, our study provides an in-depth understanding of how these RR- and ChooseBest-like policies behave, both theoretically and experimentally.

Logging has been a recurring technique in much of the work on indexing for SSDs, which have cheap random reads and expensive random writes. *BFTL* [22], *FlashDB* [15], and *LA-tree* [5] extend the classical B-tree with log-based maintenance of individual nodes or sub-trees. *SkimpyStash* [9] and *SILT* [14] are based on hashing. Although our techniques are designed for the LSM-tree, it might be possible to apply similar ideas to improve their log-based maintenance procedures. There has also been work on effectively using SSDs in LSM-based systems from a systems perspective. *Hybrid HBase* [6] studies the cost and benefit of moving components of HBase to SSDs. *LOCS* [21] is a hardware-software hybrid system that makes the parallelism inside SSDs available to a LevelDB-like implementation of LSM. Our contributions are orthogonal.

Among work on indexing for SSDs, *FD-tree* [13] and *FD+tree* [20] are extensions of the LSM-tree that use full merges by default; neither considers block preservation. FD-tree's de-amortization method roughly corresponds to our RR-P. Our results are directly applicable to them.

VII. Conclusion

Given the prevalence of SSDs and importance of modern write-heavy workloads, LSM has become a popular approach to indexing. On SSDs, writes are more expensive than reads, and they also have a wear effect which decreases drive life. This paper offers a close look at how to reduce writes in LSM. We provide the first in-depth study of how partial merges help save writes, and establish theoretical bounds on their effectiveness. We propose novel merge policies with better theoretical bounds and practical performance. We also show how to further reduce writes by reusing data blocks during merges.

From a theoretical angle, it remains an open problem to show why policies like RR and ChooseBest easily outperform Full for "common" cases as observed in experiments, despite the fact the worst-case bounds for their amortized costs are about twice as high as that of Full. Such analysis is challenging, because a partial merge policy can have interesting effects on the key distributions of data within levels as well as records moving down the levels (as shown in Example 1).

While our work focuses on SSDs, some of the methods are relevant to LSM on hard drives as well. Mixed's intelligent switching between Full and ChooseBest directly benefits LSM for hard drives. On the other hand, with hard drives' slow random accesses, block preservation and partial merges of arbitrary ranges may lead to poor performance because they decluster the data blocks on a level. Nonetheless, both optimizations can still be applied on a coarser granularity, e.g., on the order of "super" blocks (indeed, HyperLevelDB can be seen as ChooseBest applied on the granularity of SSTables). Further study in this direction may be fruitful.

Acknowledgment The authors were supported by the U.S. National Science Foundation Award IIS-1320357.

References

[1] Apache Cassandra. http://cassandra.apache.org/.
[2] Apache HBase. http://hbase.apache.org/.
[3] HyperLevelDB. http://hyperdex.org/performance/leveldb/.
[4] The TPC-C benchmark. http://www.tpc.org/tpcc/.
[5] Agrawal, Ganesan, Sitaraman, Diao, and Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *PVLDB*, 2(1):361–372, 2009.
[6] Awasthi, Nandini, Bhattacharya, and Sehgal. Hybrid hbase: Leveraging flash ssds to improve cost per throughput of hbase. In *COMAD* 2012.
[7] Bentley and Saxe. Decomposable searching problems I: static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.
[8] Chazelle and Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986.
[9] Debnath, Sengupta, and Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *SIGMOD* 2011.
[10] Ghemawat and Dean. LevelDB 1.18. https://github.com/google/leveldb, 2014.
[11] Jagadish, Narayan, Seshadri, Sudarshan, and Kanneganti. Incremental organization for data recording and warehousing. In *VLDB* 1997.
[12] Jermaine, Omiecinski, and Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16(4):417–437, 2007.
[13] Li, He, Yang, Luo, and Yi. Tree indexing on solid state drives. *PVLDB*, 3(1):1195–1206, 2010.
[14] Lim, Fan, Andersen, and Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *SOSP* 2011.
[15] Nath and Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *IPSN* 2007.
[16] O'Neil, Cheng, Gawlick, and O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.
[17] Press, Teukolsky, Vetterling, and Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 2007.
[18] Sears and Ramakrishnan. bLSM: A general purpose log structured merge tree. In *SIGMOD* 2012.
[19] Thonangi and Yang. On log-structured merge for solid-state drives. Technical report, Duke University, 2016. http://db.cs.duke.edu/papers/2016-ThonangiYang-partial_merge_lsm.pdf.
[20] Thonangi, Babu, and Yang. A practical concurrent index for solid-state drives. In *CIKM* 2012.
[21] Wang, Sun, Jiang, Ouyang, Lin, Zhang, and Cong. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *EuroSys* 2014.
[22] Wu, Chang, and Kuo. An efficient B-tree layer for flash-memory storage systems. In *International Conference on Real-Time and Embedded Computing Systems and Applications*, 2003.
[23] Yang, Tschetter, Léauté, Ray, Merlino, and Ganguli. Druid: A real-time analytical data store. In *SIGMOD* 2014.